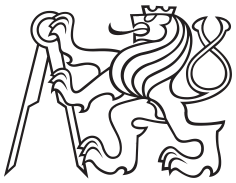
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Gamestats – kontrola statistik soutěží ledního hokeje

Michal Toman

Školitel: Bc. Petr Huťák
Obor: Softwarové inženýrství a technologie
Květen 2021

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Toman** Jméno: **Michal** Osobní číslo: **483502**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Gamestats – kontrola statistik soutěží ledního hokeje

Název bakalářské práce anglicky:

Gamestats – checking of ice hockey statistics

Pokyny pro vypracování:

Firma eSports.cz zajišťuje prostřednictvím aplikace Gamestats sběr statistik z hokejových utkání v několika evropských ligách. Data jsou sbírána přímo v arénách prostřednictvím mobilních aplikací a následně proudí na hlavní server, kde jsou dále zpracovávána. Aplikace poskytuje zpracovaná data na oficiální weby soutěží, klubů, do mobilních aplikací nebo přímých TV přenosů a je třeba zajistit dostupnost a správnost dat. Cílem projektu je připravit real-time monitorovací službu, jejíž hlavním úkolem bude kontrola, že ze všech právě probíhajících utkání do systému proudí statistická data a jsou správně zpracována. V případě, že objeví chybu, notifikuje nastavitelný kontakt zprávou obsahující podrobnosti o incidentu. Kontrolní aplikace bude vyvíjena v TypeScriptu a poběží na runtime Deno. V rámci bakalářská práce dojde také k porovnání ekosystémů a výkonu technologií Deno a Node.js pomocí implementace zjednodušených webových serverů a jejich následného benchmarkování.

Seznam doporučené literatury:

1. Deno - A secure runtime for JavaScript and TypeScript. Deno - A secure runtime for JavaScript and TypeScript [online]. Dostupné z: https://deno.land/
2. TypeScript: Handbook - The TypeScript Handbook. TypeScript: Typed JavaScript at Any Scale. [online]. Copyright © 2012 [cit. 20.11.2020]. Dostupné z: https://www.typescriptlang.org/docs/handbook/intro.html
3. Architectural Styles and the Design of Network-based Software Architectures- Dissertation, University of California, Irvine, 2000 [online]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
4. Documentation | Node.js. [online]. Copyright © OpenJS Foundation. All Rights Reserved. Portions of this site originally [cit. 12.11.2020]. Dostupné z: https://nodejs.org/en/docs/
5. JSON Web Tokens - jwt.io. JSON Web Tokens - jwt.io [online]. Dostupné z: https://jwt.io/
6. RFC 8725: JSON Web Token Best Current Practices. » RFC Editor [online]. Dostupné z: https://www.rfc-editor.org/rfc/rfc8725.html

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Bc. Petr Huťák, katedra počítačové grafiky a interakce FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **10.02.2021** Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **30.09.2022**

Bc. Petr Huťák podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Poděkování

Děkuji vedoucímu práce, panu bakaláři Petru Huťákovi, za trpělivost a vstřícnost při průběžných konzultacích po celou dobu psaní této studie. V neposlední řadě také děkuji firmě eSports.cz, s.r.o., že mi umožnila realizovat projekt kontrolního systému v rámci bakalářské práce. Jmenovitě potom děkuji kolegovi Lukáši Peroutkovi, se kterým jsem řešil klientskou část zadání monitorovací služby.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 20. května 2021

.....

Abstrakt

Tato bakalářská práce se zabývá vývojem systému kontrolujícího sbírané statistiky soutěží ledního hokeje. Aplikace byla vytvořena pro společnost eSports.cz, s.r.o. v nové technologii Deno, jejímuž výzkumu se dokument z velké části věnuje. Součástí práce je analýza existujícího systému sběru statistik, konkretizace klientského zadání kontrolní služby a její návrh spolu s implementací. Dokument obsahuje také bližší popis technologie Deno a prozkoumání jejího ekosystému. Kromě toho si také práce klade za cíl porovnat platformu Deno s jejím předchůdcem Node.js. Za účelem komparace byly vyvinuty dva podobně rozsáhlé webové servery, které byly složeny z technologií použitých při implementaci klientem zadané monitorovací služby. Studie podrobuje obě webové aplikace výkonnostním testům a také popisuje proces jejich tvorby z pohledu vývojářské přívětivosti.

Klíčová slova: Deno, Node.js, TypeScript, REST, web

Školitel: Bc. Petr Huťák

Abstract

This bachelor thesis focuses on the development of a system checking the collected statistics of ice hockey competitions.The application was created for the company eSports.cz, s.r.o. in the new Deno technology, the research of which is primarily devoted to the document. The work includes an analysis of the existing system for collecting statistics, the specification of the client assignment of the control service and its design, together with the implementation. The document also provides a more detailed description of the Deno technology and explores its ecosystem. In addition, the work also aims to compare the Deno platform with its predecessor Node.js. For comparative purposes, two similarly large web servers were developed and were composed of technologies used in the implementation of the client-specified monitoring service. The study subjects both web applications to performance tests and describes their creation from a developer-friendliness perspective.

Keywords: Deno, Node.js, TypeScript, REST, web

Title translation: Gamestats – checking of ice hockey statistics

Obsah

Obrázky

Tabulky

Kapitola 1
Úvod

Moderní sport se točí kolem čísel. Fanoušky zajímají góly, střely, zásahy brankářů a další statistiky. Data o sportovních utkáních ovšem nevznikají sama. Je zapotřebí zaměstnanců na stadionech, kteří nějakým způsobem zaznamenávají potřebné statistiky. Nejčasteji se data sbírají elektronicky a jsou dále zpracovávána na serverech, odkud proudí do světa v nejrůznějších statistických přehledech.

Sportovním statistikám se věnuje i firma eSports.cz[1]. Pomocí své aplikace Gamestats zajišťuje sběr základních i rozšířených statistik z hokejových utkání v několika evropských ligách. Data jsou sbírána v arénách prostřednictvím mobilní aplikace a následně proudí na server, kde jsou dále zpracovávána. Systém poskytuje výstupy na oficiální webové stránky soutěží, klubů, do mobilních aplikací nebo přímých televizních přenosů, a proto je důležité zajistit dostupnost a správnost dat v reálném čase.

Z toho důvodu chce firma připravit real-time monitorovací službu, jejíž hlavním úkolem by byla kontrola statistických dat z právě probíhajících utkání, která mají proudit do systému. Hlídat by měla všechny typy statistik, které se přes tablety sbírají, tedy střely, buly, čas na ledě a hity. Služba musí být schopná rozpoznat očekávané stavy, kdy data do systému téct nemusejí, tzn. komerční pauzy, přestávky mezi třetinami, samostatné nájezdy atp. V případě objevení chyby by měla informovat o incidentu osobu zodpovídající za správnost sběru dat z konkrétního utkání.

Cílem této práce je navrhnout systém, který bude řešit výše popsaný problém. Kapitola *Analýza* (2) se zabývá sběrem požadavků na systém a výběrem technologií i architektonických principů, pomocí kterých dojde k realizaci zadání. V části *Implementace* (3) se poté práce věnuje výběru konkrétních modulů či knihoven a popisu realizace monitorovací služby.

Kromě popisu realizace produkčního kontrolního systému se tento dokument zabývá také použitým runtimem Deno a jeho srovnáním s alternativním Node.js. V rámci bakalářské práce byly pro účely komparace vytvořeny dva srovnatelně rozsáhlé servery, na nichž byly provedeny testy výkonu a velikosti spustitelných souborů. Obě testované aplikace obsahovaly technologie použité při vývoji statistické monitorovací služby.

Srovnání Deno a Node.js se věnuje kapitola 4, v níž jsou k nalezení výsledky

[1]Profil firmy dostupný na: `https://www.esportsmedia.cz/` [cit. 2021-01-07]

testů naprogramovaných aplikací i porovnání obou technologií z pohledu vývojářské přívětivosti a zralosti.

Kapitola 2
Analýza

Během výběru tématu své bakalářské práce jsem narazil na nový, čerstvě vydaný runtime pro JavaScript (JS) a TypeScript (TS) Deno. Při četbě dokumentace mě velice zaujala možnost importu funkcionality z webu pomocí URL a také vestavěná podpora TS.

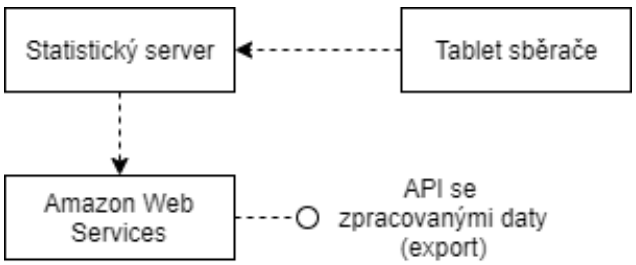
Věděl jsem, že bych se této technologii chtěl v rámci závěrečné práce věnovat. Zároveň jsem však chtěl vyvinout systém, který by měl využití v praxi. Rozhodl jsem se tedy se svým nápadem obrátit na nadřízené v zaměstnání.

Netrvalo dlouho a dostal jsem kladnou odpověď spolu s návrhem tématu, a tak začala základní část vývoje každého systému - analýza. V tomto případě navíc okořeněná o poznávání nové technologie a jejího ekosystému.

2.1 Aplikace Gamestats

Nejprve bylo potřeba se zaměřit na analýzu již implementovaného a fungujícího systému sběru statistik, jehož kontrolu měla mnou vyvíjená aplikace zajišťovat.

Aplikace Gamestats byla vyvinuta firmou eSports.cz, s.r.o. Systém zajišťuje sběr základních i rozšířených statistik z hokejových utkání v několika evropských ligách. Statistická data jsou zaznamenávána v arénách prostřednictvím mobilní aplikace na tabletech a následně proudí na server, kde jsou dále zpracovávána. Server posléze poskytuje výstupy na oficiální webové stránky soutěží, klubů, do mobilních aplikací a přímých televizních přenosů pomocí Amazon Web Services. Proces sběru dat je znázorněn na obrázku 2.1.



Obrázek 2.1: Diagram systému pro sběr statistik

Aktuálně se sbírají a uchovávají data o následujících herních činnostech či událostech:

- Vhazování
- Střely
- Čas na ledě
- Hity
- Držení puku

Ne každá ze skupiny statistických dat je ovšem sbíraná ve všech soutěžích, ve kterých se aplikace Gamestats používá. Například v první české hokejové lize nedochází ke sběru hitů a držení puku. Způsob zadání dat a jejich přenosu na server je ovšem jasně definovaný pro všechny soutěže.

Implementovaná kontrola zadávaných dat v rámci mobilní aplikace a serveru však není momentálně plně dostačující. Pro každý hrací den v dané soutěži je tedy nutné, aby sbíraná data zběžně prohlížel zaměstnanec, který v případě objevení incidentu telefonicky kontaktuje sběrače na stadionu, případně manuálně upraví data v administrátorské sekci aplikace. Zaměstnanec však musí být ve střehu po celou dobu, kdy se hraje zápas, aby chybu objevil.

2.2 Byznys požadavky

Prvních několik měsíců jsem věnoval schůzkám s klientem, na kterých jsem se snažil co nejvíce konkretizovat zadání. Soustředil jsem se také na identifikaci nejpalčivějších problémů, s nimiž se aktuálně zaměstnanci starající se o kontrolu dat potýkají.

Během sezení došlo k definici hlavní funkcionality systému, jež je popsaná v bodech níže.

1. Kontrola proudění dat z tabletů na server
2. Kontrola správného zpracování dat serverem
3. Kontrola serverového generování feedů na Amazon Web Services (AWS)
4. Notifikace nastavitelného kontaktu o incidentu pomocí e-mailu

Došlo také k rozšíření zadání o webové Graphical User Interface (GUI), jelikož klient chce větší kontrolu nad správou uživatelů, soutěží a služeb. Vývoji uživatelského rozhraní se však tento dokument nebude věnovat. V rámci bakalářské práce bylo vyvinuto GUI pouze v základní verzi.

Dále byly definovány nejčastější problémy, se kterými se kontroloři statistik setkávají. Každému problému je také přiřazena úroveň závažnosti. Popis jednotlivých úrovní je následující:

- *Warning* - chyba je pouze zaznamenána v systému u daného zápasu, nedochází k automatické notifikaci pomocí jiných informačních kanálů
- *Error* - chyba je závažná a je třeba okamžitě řešit, systém tedy odešle e-mailovou notifikaci službě, která bude předem zvolena

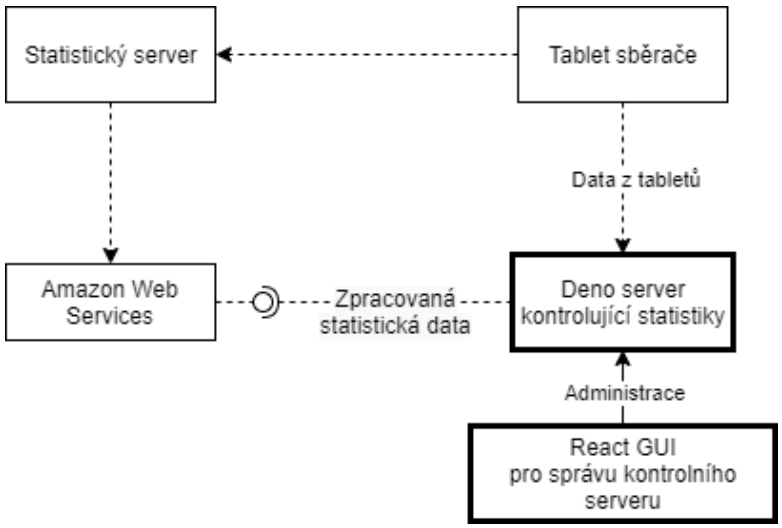
Konkrétní chyby, které bude monitorovací služba kontrolovat jsou uvedeny v následujícím seznamu. Problémy jsou seřazeny podle vrstev kontroly dat do dvou skupin a v rámci každé je jim přiřazena úroveň závažnosti.

1. Kontrola mobilní aplikace
   - Error
     - Nechodí vhazování na začátku třetiny
     - Nechodí dostatečně často data o střelách
     - Nechodí dostatečně často data o času na ledě
     - Nechodí dostatečně často data o držení puku
   - Warning
     - Přišla střela na branku (gól/chycená) bez vyplněného ID brankáře
2. Kontrola AWS feedu generovaného serverem
   - Warning
     - Zápasový čas ve statistickém záznamu není nižší než čas konce zápasu
     - Počet střel s výsledkem gól není stejný jako počet gólů ve skóre zápasu
     - Po konci utkání není vyplněn počet diváků

2.3 Technologie

Vzhledem k prvotnímu nápadu na téma práce bylo jasné, že hlavní backendová část bude vyvinuta pro runtime Deno (2.3.2). Server-side JS kromě toho vyhovuje požadavku na real-time kontrolu dat. Deno také v základu obsahuje kompilátor TS, který osobně preferuji před JS kvůli výhodám popsaným v sekci 2.3.1.

Pro aplikační rozhraní jsem se rozhodl použít zásad RESTu (2.3.3), jelikož s ním mám největší zkušenosti. Poslední volbou byla technologie pro zabezpečení jednotlivých API endpointů. Opět jsem se rozhodl podle svých zkušeností a zvolil jsem technologii JWT (2.3.4).



Obrázek 2.2: Diagram návrhu systému s kontrolou statistik

2.3.1 TypeScript

TypeScript (TS) je open-source programovací jazyk vyvinutý společností Microsoft. Jde o nadstavbu nad velice populárním JS, která svého předchůdce doplňuje o typovou kontrolu. TS kód je nejprve zkompilovaný do JS a až poté může být spuštěn. [1]

Rozšíření programovacího jazyka o definici typů pomůže vývojáři s odhalením velkého množství chyb v kódu ještě před spuštěním programu. Velice příjemná je typová kontrola při práci s objekty. V kódu lze popsat strukturu objektu, což následně umožní TS kontrolovat, jestli je objekt správně inicializován a zda uživatel nepřistupuje k proměnným a metodám, které nejsou definované.

Kód obohacený o definici typů je navíc přehlednější pro ostatní vývojáře, kteří v něm mají něco změnit. Zdrojový kód je díky definici typů lépe dokumentovaný. Například u funkce, jež je dobře definovaná, se nemůže stát, že bude vracet jiný typ, než má.

Definování typů však není v TS povinné, ba naopak. Ve spoustě případů dokáže TS správný typ k proměnné přiřadit sám pomocí odvozování. V takovém případě se kompilátor TS sám postará o to, že vývojář neudělal ve svém kódu typovou logickou chybu.

2.3.2 Deno

Deno je runtime a package manager pro JS a TS vyvinutý Ryanem Dahlem, jenž stojí i za vznikem staršího a alespoň prozatím populárnějšího runtimu pro JS Node.js.

Velikou výhodou Dena je zabudovaná podpora TS. Deno umožňuje spustit program napsaný v TS bez předchozí kompilace. O kompilaci se Deno postará samo a následně spustí přeložený kód.

Další vlastností Dena je bezpečnost. Program spuštěný v Denu nemá v základu přístup k souborům, síti ani proměnným prostředí. Pokud uživatel požaduje nějaký externí zdroj pro svůj program, musí jej při spuštění explicitně povolit.

Deno nabízí také sadu standardních knihoven, které neobsahují externí závislosti. Samotný import knihoven je v Denu velmi jednoduchý. Lze importovat veškeré exporty ze zdrojových kódů dostupných přes Content Delivery Network (CDN)[1].

Bez potřeby externích závislostí také Deno podporuje různé pro vývojáře užitečné vychytávky. Jde například o bundler, který transformuje TS projekt do jediného JS souboru, do něhož vloží i kód všech potřebných závislosti. Deno také umožňuje kompilovat projekty do samostatně spustitelných binárních souborů či formátovat zdrojový kód. [2]

Srovnání Deno a Node.js se v této bakalářské práci věnuje samostatná kapitola 4.

2.3.3 REST

Representational State Transfer (REST) je architektura pro tvorbu aplikačních rozhraní. Ve své disertační práci ji popsal Roy Fielding v roce 2000 a dnes jde o asi nejrozšířenější způsob realizace programového přístupu k aplikacím.

REST API pracuje se základní jednotkou, která se nazývá *resource*. Resource je jakákoliv ucelená informace, která může být pomocí API přenesena. Každý resource je také jednoznačně identifikován, např. pomocí URL.

K jednotlivým resourcům se přistupuje pomocí tzv. *resource methods*. Roy Fielding u nich však nespecifikuje, jaký význam má která metoda mít. V rámci jednoho API však musí mít každá metoda jednoznačný a unikátní význam. U protokolu HTTP je však známou praxí využívat metody GET, POST, PUT a DELETE, GET pro získání existujícího záznamu, POST pro vytvoření nového záznamu, PUT pro změnu existujícího záznamu a DELETE pro vymazání existujícího záznamu. [3]

Alternativou k REST architektuře bylo například použití GraphQL. Jedná se o dotazovací jazyk pro API a runtime pro exekuci dotazů, který byl vytvořen společností Facebook v roce 2012 a je při tvorbě API stále více používaný.

GraphQL na rozdíl od RESTu potřebuje jen jeden endpoint. Datové zdroje totiž nejsou identifikovány pomocí URL, ale pomocí typů a funkcí. Uživatel může v dotazu specifikovat, jaký typ dat chce obdržet a také jaké informace z něj potřebuje. Oproti RESTu lze tedy přenést data šitá klientovi přímo na míru bez nepotřebných informací. [4]

S GraphQL jsem ovšem neměl žádné dřívější zkušenosti. Navíc jsem při průzkumu Deno ekosystému nenarazil na žádnou solidní knihovnu, jež by GraphQL podporovala. Pro realizaci projektu jsem si tedy vybral REST.

[1] `https://deno.land/x` [cit. 2021-05-19] nebo `https://nest.land/` [cit. 2021-05-19]

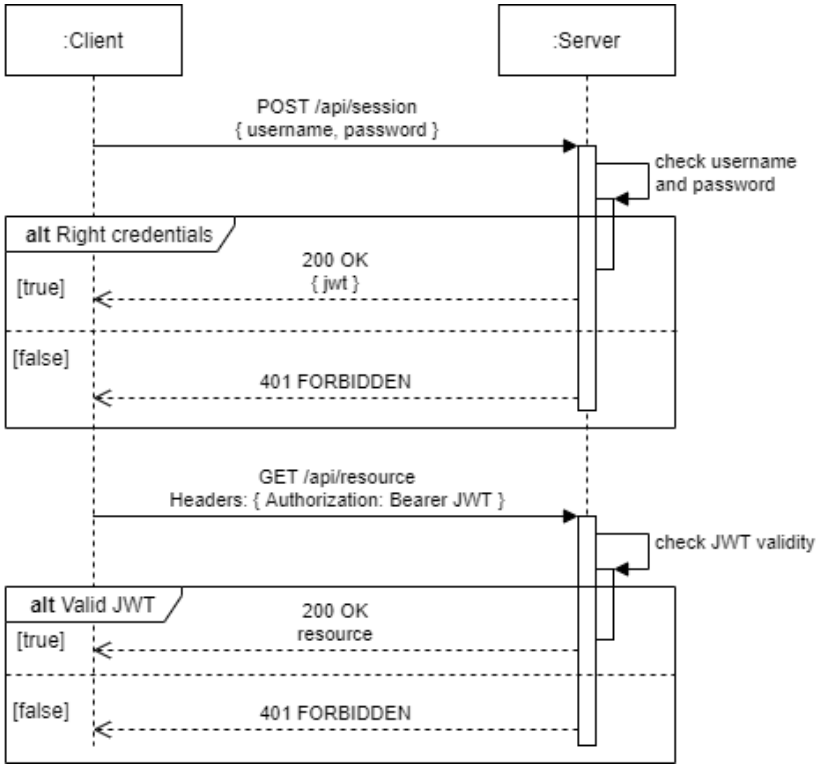
2.3.4 JWT

JSON Web Token (JWT) je podle oficiální dokumentace otevřený standard, který definuje způsob zabezpečeného přenosu informací mezi dvěma komunikujícími stranami. [5]

Nejběžnějším použitím JWT je autentizace, neboli prokázání uživatelské identity. Proces autentizace budiž znázorněn na sekvenčním diagramu na obrázku 2.3. Klient na začátku komunikace nejprve odešle na server své přihlašovací údaje, např. uživatelské jméno a heslo. Server následně ověří správnost obdržených údajů a vygeneruje JWT, který odešle klientovi. Klient si vygenerovaný token uloží a odesílá ho v HTTP hlavičce s každou žádostí na zabezpečený zdroj. Server při zpracování požadavku na zabezpečený zdroj kontroluje, zda je token v hlavičce HTTP dotazu platný a nepoškozený. Jestliže kontrola proběhne bezchybně, pak klientovi odešle požadovaná data. Pokud se při kontrole objeví chyba, server klientovu žádost zamítne.

JWT se skládá ze tří částí: *hlavičky (header)*, *těla (payload)* a *podpisu (signature)*. Jednotlivé části tokenu jsou zakódované pomocí `base64url` kódování a jsou oddělené tečkou, tzn. tvar JWT je následující - `X.Y.Z`, kde X odpovídá hlavičce, Y tělu a Z podpisu. Význam jednotlivých částí JWT je následující:

- **Hlavička** - obsahuje informaci o typu tokenu (JWT) a algoritmu, který byl použit k podpisu, např. HS256.
- **Tělo** - obsahuje sadu tvrzení, která jsou předdefinovaná, ale uživatel může přidat i libovolné jiné tvrzení. Každé tvrzení je podle dokumentace nepovinné a nemusí být v těle tokenu obsaženo. V praxi ale záleží na konkrétní implementaci. Často používaná jsou například následující tvrzení:
  - *iss* - řetězec jednoznačně určující uživatele, který žádal o vytvoření tokenu
  - *sub* - řetězec jednoznačně určující uživatele, který je předmětem tokenu, tzn. ke zdroji se přistupuje jeho jménem
  - *exp* - číslo definující čas, po kterém je token brán jako expirovaný, tudíž neplatný
  - *nbf* - číslo definující čas, před kterým je token brán jako nedozrálý, tudíž neplatný
- **Podpis** - je vygenerován za použití algoritmu uvedeného v hlavičce tokenu. Nejprve se spojí obsah hlavičky a obsah těla tokenu. Obě části jsou zakódované pomocí `base64url`. Vzniklý řetězec je nakonec zašifrován pomocí daného algoritmu a výsledek je brán jako podpis. [6]



Obrázek 2.3: Sekvenční diagram autentizace pomocí JWT

Kapitola 3
Implementace

Po analýze problému, zformování zadání a definici technologií, které mají být použity, přišla na řadu samotná implementace aplikace.

3.1 Deno ekosystém

Implementace pro runtime Deno přinesla hodně práce ještě před samotným kódováním. Prvním velkým úkolem bylo zjistit, zda v Deno ekosystému existuje podpora a knihovny pro zvolené technologie a požadovanou funkcionalitu implementovaného systému.

První problém se objevil už při hledání driveru pro spojení s databází. Aplikace měla totiž využívat databázi MariaDB verze 10.1 a jediný MariaDB driver *mysql* pro Deno podporoval spojení jen s MariaDB verze 10.2 a vyšší. Po malé úpravě kódu však driver pro základní operace fungoval i s verzí 10.1 a po pár týdnech se objevila nová verze driveru, jež začala podporovat i MariaDB verze 10.0. Spojení s databází bylo tedy vyřešeno.

Další funkcionalitou, po níž jsem se sháněl, bylo Object–relational Mapping (ORM). Objevil jsem několik pěkných modulů, ale nakonec mě nejvíce oslovil *Cotton*. Pro mapování využívá dekorátory v TS, které jsou syntakticky podobné anotacím v Javě. Použití knihovny Cotton se v této práci více věnuje kapitola 3.3.2.

Pro práci s datovou vrstvou jsou také velice potřebné migrace, pomocí kterých se řeší správa jednotlivých verzí databáze. Migrace jsem se nejprve rozhodl řešit modulem obsaženým v knihovně Cotton. To se ale nakonec neukázalo jako vhodné řešení, jelikož modul nedostával od svého autora dostatečnou podporu. Sáhl jsem tedy nakonec po separátní knihovně *Nessie*, která je v Deno ekosystému velice populární a nabízí lepší podporu. Práci s ní se více věnuje kapitola 3.3.3.

Systém měl být vyvinut jako webová aplikace, a tak bylo třeba najít vhodný webový framework. Velice populární je framework Oak, který vychází z frameworku Koa pro Node.js. Mě však více oslovil *Drash*. Jde o framework, jenž v základu obsahuje obdobu controllerů, tzv. *resources*, které jsou popsány pomocí URI a následně se předají serveru, jenž se stará o jejich obsluhu. Drash umožňuje jednoduše postavit webovou aplikaci na REST principech, což byl jeden z požadavků v zadání. Přístup k resourcům může být kontrolován

pomocí middlewaru. Lze tedy jednotlivé zdroje zabezpečit a zpřístupnit pouze oprávněným uživatelům. Drash uživateli zjednodušuje také práci s HTTP hlavičkami a cookies, takže umožňuje vše, co bylo stanoveno v zadání. Vývojáři frameworku navíc uvádějí, že implementují v souladu s Documentation Driven Developmentem a v oficiální dokumentaci je opravdu k nalezení vše potřebné. [9] V rámci této práci se Drash frameworku podrobněji věnuje kapitola 3.4.

Pro zabezpečení zdrojů aplikace má být použita technologie JWT a pro její podporu jsem našel v Deno ekosystému velmi jednoduchou, ale dostačující knihovnu *djwt*. Metody `create` pro vytvoření tokenu a `verify` pro jeho ověření a získání v něm obsažených tvrzení.

Aplikace má podle specifikace podporovat odesílání e-mailových notifikací. I pro tuto funkcionalitu jsem v Deno ekosystému našel velice jednoduchou knihovnu pro obsluhu SMTP klienta *smtp*. Klient se po vytvoření připojí k zadanému SMTP serveru a následně je možné pomocí metody `send` odesílat e-maily.

Jako poslední jsem hledal modul, který by mi usnadnil konfiguraci aplikace po jejím zabalení a nasazení do produkce. K tomuto účelu slouží `.env` soubory, jež jsou ve světě JS a TS velice populární. V takových souborech se definují parametry charakteristické pro dané prostředí, na kterém aplikace běží. Typicky se jedná o testovací a produkční prostředí. Pro čtení parametrů z `.env` souborů jsem zvolil knihovnu *Dotenv*, jež je opět velmi kompaktní a jednoduchá.

3.2 Struktura projektu

Struktura kódu projektu vychází z mé zkušenosti organizace kódu v systémech psaných v jazycích PHP a Java. Samotný kód aplikace a kód testů jsou rozděleny do dvou adresářů. Mimo hlavní kód jsou v samostatném adresáři také uloženy skripty pro správu databáze. V kořenovém adresáři je potom k nalezení kód se vstupním bodem aplikace, konfigurace a také import závislostí.

Hlavní sekce projektového adresáře vypadají následovně:

- *db* - skripty pro správu databáze
  - *migrations* - migrační skripty, popis jednotlivých verzí databáze
  - *seeds* - skripty pro naplnění databáze iniciačními daty
- *src* - aplikační kód
  - *middleware* - funkce, které lze v Drash frameworku zaregistrovat jako middleware ke konkrétním resourcům; umožňují například kontrolovat práva pro přístup k resourcu
  - *model* - definice objektů datového modelu aplikace; pomocí dekorátorů v TS a knihovně Cotton jsou mapovány na databázové tabulky

  - *repositories* - třídy definující databázové operace nad modelem
  - *resources* - obdoba controllerů v jiných webových frameworcích; Drash je nazývá resources
  - *services* - třídy s byznys logikou, která se provádí nad modelem pomocí repositářů
  - *utils* - kód s pomocnou funkcionalitou
- *tests* - testovací kód
  - *mocks* - funkcionalita pro generování testovacích dat a simulování připojení k databázi
  - *resources* - testy tříd z *src/resources*
  - *services* - testy tříd z *src/services*
  - *deps.ts* - import závislostí použitých pro testování
- *app.ts* - vstupní bod aplikace
- *config.ts* - čtení konfiguračního .env souboru
- *database.ts* - spojení s databází
- *deps.ts* - import závislostí; u větších aplikací vyvíjených pro Deno je vhodné externí závislosti importovat v jednom souboru, aby byly snadno k nalezení
- *server.ts* - tvorba serveru, registrace potřebných služeb a resourců (controllerů)

3.3 Datová vrstva

Systém pro kontrolu statistik bude obsahovat administrační modul a bude si udržovat informace o chybách, které odhalí. Z těchto důvodů potřebuje databázovou vrstvu, v níž budou uchována systémová data. Proto bylo třeba definovat, jaká data a v jaké formě budou ukládána.

3.3.1 Datový model

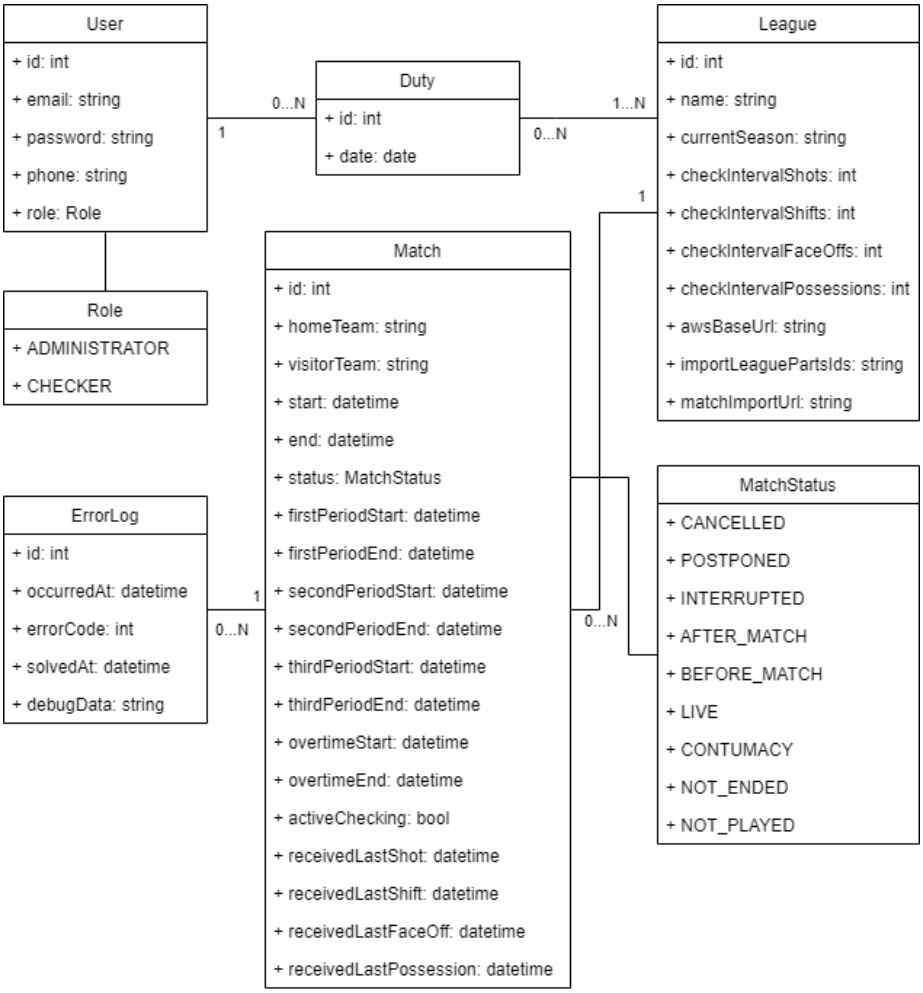
Základní entitou je uživatel systému *(User)*. Uživatelé mají následně přiřazené služby *(Duty)* na konkrétní den pro určené ligy *(League)*. Uživatel může mít v jeden den službu pro více lig a zároveň více uživatelů může mít v jeden den službu pro stejnou ligu.

Každý uživatel má přiřazenou roli, která určuje jeho práva v rámci systému. Administrátor může pracovat s uživateli i soutěžemi. Může přidávat nové uživatele i ligy a také je upravovat. Zároveň má administrátor přístup ke stejným funkcím jako kontrolor. Není tedy potřeba implementovat systém podporující více rolí pro jednoho uživatele.

Kontrolor oproti tomu může pouze koukat na jemu přiřazené služby a dostává notifikace o objevených chybách. V rámci systému může následně kontrolor označit chyby za vyřešené.

V rámci ligy systém eviduje zápasy *(Match)* a u zápasů jsou následně ukládány logy odhalených chyb *(ErrorLog)*. U každého zápasu je možné vypnout kontrolu statistik, čehož je dosaženo pomocí atributu `activeChecking`.

Všechny datové entity a vztahy mezi nimi jsou znázorněny na diagramu na obrázku 3.1.



Obrázek 3.1: Datový model kontrolního systému

3.3.2 Cotton

Pro správu databáze jsem si vybral knihovnu Cotton. Cotton podporuje Object–relational Mapping (ORM). Umí tedy mapovat objekty v kódu na tabulky v databázi. Cotton toho dosahuje pomocí dekorátorů v TS, které připomínají anotace v Javě.

Ukažme si práci s dekorátory na příkladu kódu z kontrolního systému 3.1. Jde o třídu reprezentující zápasy. Pomocí dekorátoru `@Model` registrujeme jako datový model. Pomocí parametru *matches* říkáme, že dekorovaná třída odpovídá databázové tabulce s názvem *matches*.

U parametrů třídy vidíme několik dalších důležitých dekorátorů. Dekorátor `@Primary` označuje primární klíč. V databázi se obvykle jedná o typ integer, který v TS odpovídá typu number. Primární klíč však může být například i řetězec. Záleží na definici sloupce v databázi. Podle něj musíme poté přizpůsobit typ v kódu.

Dekorátor `@Column` označuje mapování běžného sloupce a je možné mu předat objekt s nastavením určujícím typ a jméno sloupce v databázi. Jméno je potřeba uvést pouze v případě, že se nerovná názvu parametru třídy, viz řádek 6 v příkladu.

Posledním dekorátorem je `@BelongsTo`. Ten mapuje vztah *many to one* a akceptuje dva parametry - první určuje, s jakým modelem je sloupec provázán, druhý definuje, jaký je název sloupce v databázi.

Opačný směr vztahu, tedy *one to many* lze definovat dekorátorem `@HasMany`, který přijímá totožné parametry jako dekorátor `@BelongsTo`. Uložit informace o vazbě do databáze lze z obou stran entitního vztahu. Uživatel tedy může ukládat vazbu pomocí entity s označením relace dekorátorem `@BelongsTo` nebo entity používající `@HasMany`. [7]

```
1  @Model("matches")
2  class Match {
3      @Primary()
4      id!: number;
5
6      @Column({ type: DataType.String, name: "home_team" })
7      homeTeam!: string;
8
9      @Column({ type: DataType.Boolean, name: "visitor_team" })
10     visitorTeam!: boolean;
11
12     @Column({ type: DataType.Date })
13     start!: Date;
14
15     @BelongsTo(() => League, "league_id")
16     league!: League;
17 }
```

Kód 3.1: Cotton - datový model

3.3.3 Nessie

Pro verzování databáze pomocí migrací jsem použil knihovnu Nessie. Migrace je v rámci Nessie reprezentována třídou, která obsahuje asynchronní metody `up` a `down`. Pojďme se na realizaci migrace podívat v ukázce kódu 3.2.

Soubor s migrací musí mít výchozí export, který odpovídá migrační třídě, jak je vidět na řádku 1. Třída musí dědit od abstraktní migrace a jako generický parametr přijímá třídu reprezentující použitý SQL dialekt.

Abstraktní migrace obsahuje objekt `client` s metodou `query`, jež přijímá SQL příkaz, aby ho následně provedla nad databází. Metoda `up` na řádku 3 má provádět změny potřebné k tomu, aby se databáze posunula na novou verzi. Metoda `down` potom slouží k tomu, aby vrátila změny provedené předchozí metodou a dostala databázi na předchozí verzi.

```
1  export default class extends AbstractMigration<ClientMySQL> {
2      /** Runs on migrate */
3      async up(_info: Info): Promise<void> {
4          await this.client.query(
5              // SQL statement; update the database
6          );
7      }
8      /** Runs on rollback */
9      async down(_info: Info): Promise<void> {
10         await this.client.query(
11             // SQL statement; rollback the changes
12         );
13     }
14 }
```

Kód 3.2: Nessie - migrace

Nessie obsahuje také podporu pro naplnění databáze výchozími daty, tzv. *seedy*. Seed je stejně jako migrace třída, která musí být obsažena ve výchozím exportu z TS souboru.

Na rozdíl od migrační třídy však dědí od třídy reprezentující abstraktní seed a obsahuje pouze jednu metodu nazvanou `run`. Ta se opět s pomocí objektu `client` postará o vykonání SQL příkazů, jež přidají do databáze požadovaná data. Vše je vidět na příkladu 3.3. Na řádku 1 exportujeme vytvořenou třídu seedu a na řádku 4 v metodě `run` spouštíme SQL příkaz.

```
1  export default class extends AbstractSeed<ClientMySQL> {
2      /** Runs on seed */
3      async run(_info: Info): Promise<void> {
4          await this.client.query(
5              // SQL statement; insert data to the database
6          );
7      }
8  }
```

Kód 3.3: Nessie - seed

Seed lze v podstatě zapsat i jako migraci, které jen bude chybět metoda `down`. I v rámci migrace může uživatel pomocí SQL příkazů vkládat data do databáze. V tom případě ale poté nelze při spouštění migrací oddělit logiku plnění databáze výchozími daty od tvorby struktury databáze. Tento problém řeší právě seedy. Spouštění migrací a seedů je totiž odděleno a uživatel má možnost vynechat generování výchozích dat. Seedy jsou také v projektu umístěny v jiném adresáři, takže je pak snazší najít kód generující výchozí data.

3.4 Drash

Drash je webový mikroframework pro tvorbu webových aplikací běžících na Deno HTTP serveru. Drash respektuje zásady RESTu a nemá žádné externí závislosti. Vychází pouze ze standardních Deno modulů. Velkou výhodou Drashe je rozsáhlá oficiální dokumentace a tým vývojářů, kteří se starají o jeho aktualizace pro nejnovější verze Dena.

Routování v tomto frameworku funguje odlišně od většiny JS webových knihoven pro Deno i Node.js. Drash používá jako základní stavební kámen tzv. *resource*.

Resource, neboli zdroj, je podle dokumentace na webu MDN cílem HTTP požadavku a jeho podoba není nijak dále specifikována. Může se jednat o jakýkoliv datový zdroj. Každý resource je jednoznačně definován pomocí URI. [8]

Ukažme si práci s knihovnou na příkladu kódu 3.4. Na řádku 4 definujeme třídu, která dědí od třídy `Drash.Http.Resource`.

Statická proměnná `paths` na řádku 5 určuje, jaká všechna URI budou obsluhována definovanou třídou. Při dotazu na webový server Drash porovnává URI dotazu s cestami, jež jsou uvedeny ve všech zaregistrovaných resource třídách.

Následně definujeme veřejné metody, které se musí jmenovat podle HTTP metod, tzn. GET, POST apod. Drash při přístupu k některé URI z proměnné `paths` zavolá metodu, jež má stejný název jako HTTP metoda použitá pro přístup k URI. Pokud použijeme náš příklad a iniciujeme HTTP dotaz `GET /`, provede se kód v metodě definované na řádku 6.

V popisu URI můžeme uvést také parametr, který označíme pomocí dvojtečky. Parametr je v základu povinný a musí v URI existovat. Toto chování ale můžeme změnit pomocí otazníku za názvem parametru (řádek 5). K označenému parametru poté můžeme v metodách přistupovat pomocí metody `getPathParam()`, jak je ukázáno na řádku 7.

Generování odpovědi probíhá ve veřejné metodě nazvané podle HTTP metody pomocí modifikování objektu `response`. Na řádku 8 například nastavujeme tělo HTTP odpovědi. Každá resource metoda potom musí objekt s odpovědí vrátit, viz řádek 9.

```
1  import { Drash } from
2      "https://deno.land/x/drash@v1.3.1/mod.ts";
3
4  class MyResource extends Drash.Http.Resource {
5    static paths = ["/:name?"];
6    public GET() {
7      let name = this.request.getPathParam("name");
8      this.response.body = `Hello, ${name ? name : "stranger"}!`;
9      return this.response;
10   }
11 }
```

Kód 3.4: Drash - definice resource

Každý resource musí být následně zaregistrován v serveru. Toto je ukázáno na příkladu 3.5. Na řádku 7 registrujeme při tvorbě serveru všechny resourcy, které mají být přístupné. Na řádku 10 poté spouštíme HTTP server na adrese http://localhost:1447.

Třída `Drash.Http.Server` nabízí více možností konfigurace a všechny lze prostudovat v podrobně dokumentovaném kódu[1].

Stejně tak je v oficiální dokumentaci možné nalézt podrobnější popis práce s parametry. [9]

```
1  import { Drash } from
2      "https://deno.land/x/drash@v1.3.1/mod.ts";
3  import MyResource from "./myResource.ts";
4
5  const server = new Drash.Http.Server({
6      response_output: "application/json",
7      resources: [MyResource]
8  });
9
10 server.run({
11     hostname: "localhost",
12     port: 1447,
13 });
```

Kód 3.5: Drash - registrace resource

[1] Dostupné z: `https://deno.land/x/drash@v1.3.1/src/interfaces/server_configs.ts` [cit. 2020-01-01]

3.5 Testování

Deno v základu obsahuje svůj modul pro testování, jenž je velice používaný. Sám o sobě však neobsahuje podporu pro seskupování testů do větších skupin. Nemá tak ani podporu pro test hooky - funkce, které se spouští před nebo po každém testu či skupině testů. Z těchto důvodů není příliš pohodlné psát testy pomocí základní testovací knihovny od tvůrců Dena.

Pro testování jsem nakonec zvolil framework *Rhum*, který je vyvinut pro Deno od stejných vývojářů jako webový framework Drash. Stejně jako u Drashe jsem ocenil bohatou dokumentaci a také se mi velmi zalíbil způsob, jakým se unit testy pomocí Rhumu píší.

Rhum dělí testy do tří úrovní. Je to proto, aby byly výstupy z testů přehledné a vývojář okamžitě viděl, kde nastala v kódu chyba. Jednotlivé úrovně jsou vypsány níže.

1. **Test plan** - třída či soubor, který má být testovaný; seskupuje test suity
2. **Test suite** - metoda či funkce, která má být otestovaná; seskupuje test casy
3. **Test case** - základní jednotka testů; na této úrovni se testuje konkrétní funkcionalita, tzn. uživatel definuje, jak má vypadat návratová hodnota funkce se zadanými vstupy

Ukažme si nyní několik základních znaků testů v knihovně Rhum na následujícím příkladu 3.6. Kód je převzat z testování byznys logiky v aplikaci vyvíjené v rámci této práce.

Na řádku 6 registrujeme test plan pro třídu `UserService`. V rámci test planu následně definujeme test suite pro metodu `checkCredentials` a pod test suitem uvádíme samostné test casy, jež pro přehlednost opatříme popisem funkcionality, kterou testujeme. Samotný test provádíme v test casu pomocí vybrané kontrolní metody z jmenného prostoru `Rhum.asserts`, jejichž názvy a funkcionalita odpovídají standardní testovací knihovně od vývojářů Dena[2].

Definici test hooku vidíme na řádku 9. Jedná se o hook `beforeEach`, tzn. kód ve funkci, kterou hooku předáme jako parametr, se provede před každou položkou z nižší úrovně testů, než je úroveň stávající. V našem případě je hook definován v rámci test suitu `checkCredentials`, tedy se provede před každým test casem, který je uvedený pod test suitem `checkCredentials`.

Testy lze spustit příkazem `deno test`. Avšak aby příkaz fungoval, musí se testy v kódu nejprve zaregistrovat, což v naší ukázce kódu provádíme pomocí řádku 31.

Popis psaní testů je podrobněji vysvětlen v oficiální dokumentaci, pomocí níž jsem se snažil v této práci přiblížit použití Rhumu a nastínit jeho výhody. [10]

[2]Dostupné z: `https://deno.land/std@0.83.0/testing` [cit. 2021-01-01]

```
import { UserService } from
    "../../src/services/userService.ts";
import { Rhum } from "../deps.ts";
import { getMockUserService } from "../mocks/users.ts";

Rhum.testPlan("UserService", () => {
    let userService: UserService;
    Rhum.testSuite("checkCredentials()", () => {
        Rhum.beforeEach(() => {
            userService = getMockUserService();
        });
        Rhum.testCase("Returns false with wrong credentials",
        async () => {
            const result = await userService.checkCredentials(
                "checker@xyz.com",
                "hello"
            );
            Rhum.asserts.assertEquals(result, false);
        });
        Rhum.testCase("Returns true with right credentials",
        async () => {
            const result = await userService.checkCredentials(
                "admin@admin.com",
                "hello"
            );
            Rhum.asserts.assertEquals(result, true);
        });
    });
});

Rhum.run();
```

Kód 3.6: Rhum - struktura testů

Kapitola 4

Deno vs. Node.js

Ryan Dahl v roce 2018 poprvé představil serverový runtime Deno pro JS a TS. Jeho oznámení vzbudilo v komunitě JS/TS vývojářů poměrně velký rozruch. Vždyť Deno bylo symbolicky představeno v rámci Dahlovy přednášky *10 Things I Regret About Node.js* jako jakýsi nástupce svého staršího sourozence Node.js. [11]

Runtime Node.js je na světě už od roku 2009, kdy byla vydána jeho verze 0.0.1. [12]. Platforma za dobu své existence urazila pořádný kus cesty a stala se jednou z nejpopulárnějších, co se výběru exekučního prostředí pro backend webových aplikací týká. Od doby vzniku Node.js však uběhla pěkná řádka let a Dahl si až s novými trendy uvědomil nedokonalosti, kterých se při vývoji Node.js dopustil. [11]

Deno slibovalo odstranění palčivých problémů, se kterými se vývojáři při používání Node.js potýkali. Avšak vyrovnat se technologii, která má za sebou více než deset let života a ekosystém s nespočtem knihoven, není nic jednoduchého. Téměř tři roky po zveřejnění první verze je ovšem Deno ekosystém rozvinutější a konkurenceschopnější.

V rámci této bakalářské práce jsem se rozhodl obě technologie porovnat. Z tohoto důvodu jsem naprogramoval jednoduché webové servery pro Deno i pro Node.js. Oba zdrojové kódy jsou psané v TS stejně jako kontrolní systém aplikace Gamestats (3).

Deno server jsem složil z technologií a knihoven, jež byly použity pro vývoj kontrolního systému aplikace Gamestats a jsou popsané v kapitole 3. Použil jsem tedy webový framework Drash, jenž obsahuje i vlastní logger. Pro práci s datovou vrstvou jsem potom sáhl po knihovně Cotton, která podporuje připojení k MariaDB serveru. Validaci uživatelských dat jsem implementoval pomocí knihovny Zod, která funguje velice efektivně spolu s TS. Pro autentizaci jsem poté sáhl po implementacích JWT a bcryptu pro Deno.[1]

U vývoje serveru pro Node.js jsem volil knihovny podle popularity v komunitě vývojářů, ale i podle podobnosti s technologiemi, které jsem použil při implementaci aplikace pro Deno. Ve volbě webového frameworku s přehledem vyhrál Express. Ten je asi nejpopulárnějším webovým frameworkem pro Node.js a sám jsem s ním měl dřívější zkušenosti. Nejpoužívanějším

[1] Dostupné z: `https://github.com/mitom18/example-server-deno` [cit. 2021-05-04]

loggerem pro Node.js je Winston. Pro práci s daty jsem si vybral TypeORM, který je svým stylem velice podobný Cottonu a také podporuje připojení k MariaDB databázovému serveru. Validaci uživatelského vstupu jsem opět provedl s pomocí knihovny Zod a pro autentizaci jsem použil implementace JWT a bcryptu pro Node.js.[2]

Následující podkapitoly obsahují porovnání obou implementovaných serverů z pohledu volby knihoven, výkonu a velikosti zdrojových kódů.

4.1 Ekosystémy

První projekt poháněný Deno runtimem jsem naprogramoval téměř před rokem. Potýkal jsem se tehdy s nevelkým výběrem knihoven a poměrně malou podporou různých databázových driverů. I v době začátku psaní této bakalářské práce jsem pociťoval nedostatek vývojářské podpory.

Největším problémem, na který jsem narazil, byl postupný úbytek vývojářské podpory u některých knihoven, jež jsem si na začátku práce zvolil. To mě nakonec vedlo k tomu, že jsem musel hledat obdobně fungující nástroje, které ovšem zaštiťovali aktivnější vývojáři. Není se však čemu divit. Deno je stále ještě velice nová technologie a v produkčním světě není stále tolik používaná, takže vývojáři knihoven nemají tak velikou uživatelskou základnu, aby jim stálo za to věnovat údržbě knihovny tolik času.

Deno však na rozdíl od Node.js nabízí v základu několik velice pěkných vychytávek, jako je například instalace uživatelských skriptů, formátování kódu a bundler. Mimoto také Deno umožňuje uživatelskou aplikaci zkompilovat do binárního kódu, a to jak pro Linux, tak i pro Windows a Mac OS.

Kompilátor je ovšem relativně nová funkcionalita, jež byla přidána do Dena ve verzi 1.6.0 a je stále označena jako nestabilní. Osobně jsem u ní narazil na veliký problém, a sice ten, že ještě nepodporuje Web Worker API. To pro mě tedy znamenalo, že moje aplikace podporující autentizaci nemohla být ve zkompilované verzi používána. Mnou použitá implementace hashování hesel pomocí algoritmu bcrypt je totiž výpočetně náročná a vyžaduje použití workerů, aby neblokovala další exekuci kódu. To je velice důležité u implementace webového serveru, jelikož blokování exekuce kódu znamená také blokování zpracování HTTP požadavků a má veliký dopad na výkon aplikace. Jiná obstojná implementace hashování hesel pro Deno zatím neexistuje, a tak jsem kompilátor mohl použít jen pro prozkoumání velikosti vygenerovaného spustitelného souboru.

Další problém jsem objevil při použití bundleru. Ten transpiluje napsaný TS kód a i se všemi závislostmi ho zapíše do jediného JS souboru, který je následně možné spustit. Bundler ovšem neumí stáhnout a zkompilovat soubory spouštěné pomocí Web Workerů. Při použití implementace hashování hesel pomocí bcrypt algoritmu tedy opět nastal problém. Při každém HTTP požadavku na přihlašovací endpoint musel server stáhnout a zkompilovat TS soubor. To se výrazně podepsalo na počtu možných připojení serveru, jelikož

[2] Dostupné z: `https://github.com/mitom18/example-server-nodejs` [cit. 2021-05-04]

Deno runtime při stovkách připojení za sekundu narážel na to, že si přepisoval cachovaný worker soubor. Řešením bylo tedy použít bundler separátně na soubor pro workera a následně upravit JS bundle aplikace, aby importoval lokální bundle workera místo stahování TS souboru z webu.

Deno bundler mi nabídl ještě jednu kuriozitu k vyřešení. Narazil jsem na ni při použití validační knihovny Zod. Pro reprezentaci knihovních validačních funkcí použil bundler jiný objekt, než měl. Musel jsem tedy balíček zrevidovat a chybu manuálně upravit.

Všechny popsané chyby jsou ovšem dané hlavně nevyzrálostí Dena jako celku a je třeba je brát s rezervou. Deno tyto chyby postupně opraví a na rozdíl od Node.js bude mít výhodu v tom, že uvedenou funkcionalitu podporuje *out of the box*. Není nutné instalovat žádný jiný software či jiné externí závislosti, aby mohl uživatel použít bundler, kompilátor apod.

V době psaní této práce jsem ale měl jednodušší práci při implementaci webové aplikace pro Node.js. Transpilování TS jsem sice musel řešit instalací TS kompilátoru a pomocí závislosti jsem řešil i kompilování spustitelného souboru. Nicméně vše proběhlo bez větších problémů a i při výběru knihoven jsem pocítil rozdíl. Ekosystém kolem Node.js má přeci jen oproti Denu skoro deset let náskok. Knihovny mají velikou autorskou i komunitní podporu a vývojář má tak při výběru skládání závislostí svého projektu spoustu variant.

Deno se postupně dostává do fáze, kdy bude mít k dispozici plnohodnotný ekosystém obsahující řešení téměř jakéhokoliv implementačního problému. Pomáhá tomu rozrůstající se komunita a také vývojáři, kteří vytvářejí porty svých modulů z Node.js na Deno. Je tedy jen otázkou času, než se Deno dostane v rozsahu ekosystému na úroveň Node.js. A možná ho i kvalitativně předčí, vezmeme-li v potaz nástroje, které Deno nabízí samo o sobě.

4.2 Výkon

Prvním kritériem, které mě po vyvinutí dvou příkladových webových aplikací zajímalo, byl jejich výkon. Bude aplikace běžící na Denu rychlejší než na Node.js? Zvládne více požadavků za minutu? Jaká bude latence obou serverů?

Abych si odpověděl na výše uvedené otázky, rozhodl jsem se oba webové servery otestovat pomocí nástroje *Autocannon*. Připravil jsem si pro tento účel jednoduchý skript běžící na Node.js a využívající Autocannon API. Testovací program má za cíl otestovat oba servery co nejkomplexněji, a tak se postupně dotazuje na všechny vystavené serverové endpointy. Kód testovacího programu je k nahlédnutí na platformě GitHub[3].

Autocannon API nabízí spoustu pokročilých nastavení. Pro účely této práce jsem však měnil jen základní parametry a to na následující hodnoty:

[3]Dostupné z: `https://github.com/mitom18/deno-vs-nodejs-autocannon` [cit. 2021-05-09]

- *Počet spojení* - 100
- *Počet HTTP pipeline* - 1 (provede požadavky sekvenčně, čeká na odpovědi serveru)
- *Čas testu* - 10 sekund
- *Počet vláken* - 4 (každé vlákno dostane na starost 100/4=25 spojení) [13]

V každém spojení provádí testovací skript sekvenčně následující HTTP dotazy:

1. *POST /sessions* - přihlášení pomocí předpřipravených přístupových údajů, v odpovědi obdrží JWT, který použije v následujících dotazích
2. *GET /sessions/current* - obdrží informace o aktuálně přihlášeném uživateli
3. *POST /users* - vytvoření nového uživatele, v odpovědi obdrží informace o vytvořeném uživateli, jeho ID použije v následujících dvou dotazech
4. *PUT /users/:id* - úprava právě vytvořeného uživatele včetně změny hesla, aby na serveru došlo k jeho zahashování
5. *GET /users/:id* - dotaz na upraveného uživatele
6. *GET /users* - dotaz na všechny uživatele

Původně jsem chtěl testování podrobit zkompilované binární soubory, u kterých jsem očekával největší výkon. Nicméně tento přístup jsem musel zavrhnout vzhledem k problému s Deno kompilátorem, jenž nepodporuje Web Worker API, viz sekce 4.1. U Deno serveru jsem se tedy rozhodl otestovat bundle upravený podle popisu v sekci 4.1 a u Node.js serveru jsem testoval verzi transpilovanou do JS.

Test byl proveden na stroji se specifikací, která je uvedena v tabulce 4.1. Stejně tak jsou v tabulce k nalezení verze runtimů Deno a Node.js, na nichž byly oba servery testovány.

PC	Acer Aspire V15 Nitro
Procesor	2.6 GHz 2jádrový Intel Core i7
RAM	8 GB DDR4
Deno	verze 1.9.2
Node.js	verze 14.16.0

Tabulka 4.1: Srovnání testovaných aplikací - specifikace systému

Výsledky nakonec vyzněly pro aplikaci napsanou pro Node.js poměrně přesvědčivě. V průměrných hodnotách všech statistik byl Node.js server o třídu lepší, jak je vidět v tabulce 4.2.

Latence serveru pro Deno byla v průměru o 2,3 sekundy delší než u druhé testované aplikace. Node.js server zvládl za sekundu obsloužit průměrně 66,5 požadavků, což je téměř třikrát tolik, než kolik zvládl obsloužit program pro Deno.

Statistika	Deno	Node.js
Latence [ms]	3664,71	1316,3
Požadavky za sekundu	23,4	66,5

Tabulka 4.2: Srovnání testovaných aplikací - prům. výkon

Tak drtivou převahu ve výkonu nemohla aplikace pro Node.js získat jen díky použitému runtimu. Problém ve špatném výkonu serveru pro Deno musel být v některé z použitých knihoven. Provedl jsem tedy test Deno aplikace ještě jednou a tentokrát jsem využil profilovací schopnosti Dena.

Deno podporuje V8 Inspector Protocol. Je tedy možné debugovat programy poháněné Deno runtimem například pomocí Chrome Devtools. Při spouštění aplikace stačí do příkazu `deno run` přidat pouze příznak `--inspect` nebo `--inspect-brk`. [14]

Pomocí profilovacího nástroje v Chrome Devtools jsem následně zjistil, že nejvíce času při testu Deno aplikace zabere práce s hesly pomocí knihovny bcrypt. Implementace tohoto hashovacího algoritmu pro Deno využívá Web Workers API a jeho rychlost bohužel není pro zátěžové testy dostačující.

V implementaci algoritmu bcrypt pro Node.js se oproti tomu využívá podpora nativních addonů. Addony poskytují rozhraní mezi JS kódem a kódem napsaným v C/C++. Je tedy možné výpočetně náročné operace z JS kódu jen spouštět a provádět je na nižší vrstvě programu ve zkompilovaném C++, což vede ke značnému zrychlení aplikace. [15]

Zajímalo mě tedy, jakého výkonu dosáhne testovaný Deno server po odstranění práce s hesly. A zlepšení bylo výrazné, jak je vidět v tabulce 4.3. Latence se zlepšila více než čtyřikrát a počet zpracovaných požadavků za sekundu se zvýšil téměř pětkrát.

Statistika	Deno bez alg. bcrypt
Latence [ms]	840,22
Požadavky za sekundu	113,5

Tabulka 4.3: Deno server - prům. výkon bez alg. bcrypt

Data ukazují, že Deno server neprohrál v testu kvůli výkonu runtimu, ale kvůli vybranému modulu pro práci s hesly. Bohužel však pro Deno prozatím neexistuje lepší knihovna, která by řešila daný problém. Test v rámci této práce měl za cíl srovnat výkon serverů složených z běžně používaných technologií. A v tomto testu vyhrála implementace pro Node.js.

4.3 Velikost

Další metrikou pro mě byla datová velikost obou webových serverů. Důležité je zmínit, že jsem nechtěl porovnávat velikost zdrojových kódů v TS, ale až JS kódu, který byl také podroben testu výkonu. Do výsledků jsem zahrnul také velikost potřebných závislostí obou aplikací.

U Dena bylo měření velikosti aplikace přímočaré. Bundler dodaný v rámci Dena totiž tvoří jeden soubor včetně všech používaných závislostí. K jeho velikosti jsem poté přičetl ještě velikost JS souboru s bcrypt workerem, který bylo třeba manuálně transpilovat kvůli důvodům popsaným v sekci 4.1.

Měření Node.js aplikace bylo o poznání složitější. Node.js aplikace napsané v TS se do produkce nenasazují v žádném balíčku, ale pouze v podobě JS kódu, jenž je vytvořen TS kompilátorem. Velikost čistého JS kódu testovaného serveru byla 52 kB.

Následně však bylo potřeba přidat velikost všech použitých závislostí. Pro porovnání jsem chtěl použít čistě jen produkční závislosti aplikace, abych nijak nezvýhodnil Deno. Vyloučil jsem tedy závislosti pro vývoj, což byly převážně moduly podporující vývoj v TS nebo modul pro kompilaci aplikace do spustitelného binárního kódu. Celá složka *node_modules* jen s produkčními závislostmi zabírala na disku 45056 kB, což je v součtu s dříve změřenými 52 kB několikanásobně více, než kolik zabíral balíček Deno aplikace.

Dále jsem musel oba servery zkompilovat do samostatně spustitelných souborů, abych mohl porovnat jejich velikost. U aplikace pro Deno jsem k tomuto účelu použil standardní funkcionalitu `deno compile`, která vytvoří spustitelný soubor s obsaženým Deno runtimem. Node.js oproti tomu žádný standardní kompilátor spustitelných souborů neobsahuje. Musel jsem tedy použít externí modul *pkg* od společnosti Vercel, jenž funguje na stejném principu jako Deno kompilátor. [16] U obou testovaných serverů jsem provedl kompilaci do binárního kódu s povolenou kompresí, aby byly soubory co nejmenší.

Ve srovnání velikostí spustitelných binárních souborů si vedlo lépe Deno. Pokud se podíváme na tabulku 4.4, všimneme si výrazného rozdílu u souborů spustitelných na strojích s operačním systémem Windows. Soubor vygenerovaný Deno kompilátorem byl o téměř 13 MB menší než soubor vygenerovaný pomocí kompilátoru pro Node.js.

Druh kompilace	Deno [kB]	Node.js [kB]
Produkční JS kód	2104	52+45056
Binární kód - Linux	58705	58813 *(pkg)*
Binární kód - Windows	39672	52574 *(pkg)*
Binární kód - Mac OS	56116	54957 *(pkg)*

Tabulka 4.4: Srovnání testovaných aplikací - datová velikost

Pokud se ale podíváme na soubory pro platformy Linux a Mac OS, nenalezneme žádný významný rozdíl. U Dena lze však předpokládat, že se

velikost zkompilovaných souborů ještě více zmenší, neboť jeho vývojáři na této funkcionalitě v posledních verzích intenzivně pracují.

4.4 Závěr

Tato kapitola měla za cíl srovnat technologie Deno a Node.js z různých úhlů pohledu. Pokud se podíváme na srovnání jejich ekosystémů a pohodlnosti práce pro vývojáře, vyjde z testu mírně lépe Node.js. Deno je stále poměrně nová technologie, a proto za sebou nemá zatím tak velikou vývojářskou komunitu. Stejně tak i standardní funkcionalita nabízená přímo od vývojářů Dena není zatím pořádně otestovaná a hodně vychytávek je označeno jako nestabilní.

Problémy při vývoji aplikace pro Deno potkaly i mě. Nejpalčivějším problémem pro mě byla ztráta vývojářské podpory u knihoven, které jsem se rozhodl použít v jejich začátcích. Narazil jsem taky na potíže s Deno bundlerem a kompilátorem, s čímž je ale třeba počítat, jelikož jsou zatím v Denu označeny jako nestabilní stejně jako Web Worker API.

Avšak co se týče práce se závislostmi, má Deno z pohledu vývojáře velkou výhodu. Externí kód není třeba stahovat do separátní složky, jako je tomu u Node.js a adresáře *node_modules*. V Deno kódu uživatel importuje závislosti pomocí URL vedoucí na kód umístěný v některé z oficiálních CDN. Runtime si při prvním spuštění závislosti uloží do globální cache a následně používá uložený kód. Uživatel má ale možnost závislosti v cache aktualizovat pomocí příkazu v příkazovém řádku.

Jedna z výhod odlišného přístupu ke správě závislostí se projevila také v testu velikosti spustitelného JS kódu. Deno bundler vložil do kódu jen potřebné knihovny a oproti velikosti Node.js produkčních závislostí vytvořil nesrovnatelně menší kód.

Deno také na rozdíl od Node.js podporuje přímou kompilaci TS. Vývojář tedy nemusí stahovat externí závislosti pro převod TS kódu do JS a může rovnou vyvíjet a testovat svou aplikaci v TS podobě.

V testu výkonu ovšem na plné čáře zvítězil testovaný sever pro Node.js. Aplikace měla nižší latenci, zvládla více požadavků za sekundu a i rychlost přenosu dat byla větší než u protistrany běžící na Denu. Nesmíme ale výsledky interpretovat tak, že Node.js je samo o sobě rychlejším runtimem. Na výsledcích se projevuje zejména volba knihoven použitých k vývoji a ty pro Deno za sebou nemají tolik testování v produkčním prostředí jako ty pro Node.js.

Deno je rozhodně zajímavá alternativa ke svému staršímu sourozenci. Ryan Dahl a ostatní vývojáři kolem Dena opravdu naslouchali potřebám JS/TS komunity a podařilo se jim odstranit nejpalčivější problémy Node.js. Deno ale prozatím není připravené pro plnohodnotné produkční použití. V tom je zatím bezesporu lepší zaběhnuté Node.js. Deno však nabízí pro vývojáře mnoho zajímavých vychytávek a v budoucnu se s ním rozhodně musí počítat. Podle všeho se z něj však nestane náhrada Node.js, ale spíše jeho alternativa.

Kapitola 5

Závěr

Tato práce se věnovala sběru požadavků a návrhu řešení zadaného problému kontroly statistických dat v již existující aplikaci Gamestats. Kromě toho došlo v práci k průzkumu ekosystému Deno a objevení technologií, pomocí nichž bylo zadání realizováno. V rámci bakalářské práce byla také vyvinuta backend část kontrolní aplikace a jednoduché frontendové GUI vytvořené pomocí frameworku ReactJS.

V analytické části (2) se tento dokument věnuje popisu existující aplikace pro sběr statistických dat. Dále je ve stejné části rozvedeno zadání samotného systému pro kontrolu statistických dat a následně jsou popsány základní technologie a pricipy, které byly použity při realizaci projektu.

Implementační část (3) podrobněji rozvádí výzkum ekosystému technologie Deno a uvádí konkrétní knihovny, jež byly při realizaci projektu použity. Důležitým a rozsáhlejším modulům je věnováno více prostoru, aby bylo možno lépe popsat jejich použití.

V rámci práce také došlo ke srovnání technologií Deno a Node.js. Porovnání bylo provedeno na dvou testovaných webových aplikacích, jež byly pro účely této práce vyvinuty. Jedna aplikace byla vyvinuta pro Deno, druhá pro Node.js. Obě byly navrženy tak, aby obsahovaly co nejvíce technologií použitých v rámci kontrolního systému aplikace Gamestats.

Ve srovnávací části (4) jsou popsány ekosystémy obou runtimů na příkladu vývoje obou testovaných serverů. V rámci stejné části práce také došlo k porovnání obou technologií z pohledu výkonu a datové velikosti obou testovaných aplikací. Výkon byl změřen pomocí tzv. *stress testingu*.

Hlavní cíl práce, kontrolní aplikace Gamestats, je nyní ve fázi uživatelského testování. Její nasazení do produkce je s klientem plánováno na začátek následující hokejové sezony v srpnu 2021. Využije se pro kontrolu statistik z nejvyšších hokejových soutěží v Česku a Německu a také pro Champions Hockey League.

Příloha A

Seznam použitých zkratek

API Application Programming Interface. 5, 7, 22–25, 27

AWS Amazon Web Services. 3–5

CDN Content Delivery Network. 7, 27

GUI Graphical User Interface. 4, 29

HTTP HyperText Transfer Protocol. 8, 12, 17, 18, 22, 24

JS JavaScript. 3, 5–7, 12, 17, 21–27

JWT JSON Web Token. vii, viii, 5, 8, 9, 12, 21, 22, 24

MDN Mozilla Developer Network. 17

ORM Object–relational Mapping. 11, 15

REST Representational State Transfer. vii, 5, 7, 11, 17

SMTP Simple Mail Transfer Protocol. 12

SQL Structured Query Language. 16, 17

TS TypeScript. 3, 5–7, 11, 12, 15, 16, 21–23, 26, 27

URI Uniform Resource Identifier. 11, 17

URL Uniform Resource Locator. 3, 7, 27

Příloha B

Ukázky kódu

Příloha C

Literatura

[1] TypeScript: Typed JavaScript at Any Scale. *TypeScript* [online]. 2012 [cit. 2020-12-14]. Dostupné z: `https://www.typescriptlang.org/`

[2] Deno - A secure runtime for JavaScript and TypeScript. *Deno* [online]. [cit. 2020-12-18]. Dostupné z: `https://deno.land/manual`

[3] FIELDING, Roy. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Dostupné také z: `https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`. Disertace. University of California, Irvine.

[4] Introduction to GraphQL. *GraphQL* [online]. c2021 [cit. 2021-05-12]. Dostupné z: `https://graphql.org/learn/`

[5] JONES, Michael, et al. *RFC 7519 - JSON Web Token (JWT)* [online]. 2015 [cit. 2020-12-29]. Dostupné z: `https://tools.ietf.org/html/rfc7519`

[6] JSON Web Token Introduction. *JSON Web Tokens* [online]. [cit. 2020-12-18]. Dostupné z: `https://jwt.io/introduction/`

[7] FADHIL, Rahman. *Cotton* [online]. 2020 [cit. 2021-01-02]. Dostupné z: `https://rahmanfadhil.github.io/cotton/guide/`

[8] Identifying resources on the Web. *MDN Web Docs* [online]. c2005-2021 [cit. 2021-05-08]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web`

[9] Drash - Introduction. *Drash Land* [online]. [cit. 2020-12-30]. Dostupné z: `https://drash.land/drash/v1.x`

[10] Rhum - Introduction. *Drash Land* [online]. [cit. 2020-12-30]. Dostupné z: `https://drash.land/rhum/v1.x`

[11] JSConf, 2018, *10 Things I Regret About Node.js - Ryan Dahl - JSConf EU*, YouTube video. [cit. 2020-05-13]. Dostupné z: `https://www.youtube.com/watch?v=M3BM9TB-8yA`

[12] DAHL, Ryan, et al. Release v0.0.1 - nodejs/node. *GitHub* [online]. c2021, 2009 [cit. 2021-05-13]. Dostupné z: `https://github.com/nodejs/node/releases/tag/v0.0.1`

[13] COLLINA, Matteo, et al. Release v7.3.0 - mcollina/autocannon. *GitHub* [online]. c2021, 2021 [cit. 2021-05-13]. Dostupné z: `https://github.com/mcollina/autocannon/releases/tag/v7.3.0`

[14] Debugging your code. *Deno Manual* [online]. [cit. 2021-05-19]. Dostupné z: `https://deno.land/manual@v1.9.2/getting_started/debugging_your_code`

[15] C++ addons. *Node.js v14.17.0 Documentation* [online]. [cit. 2021-05-19]. Dostupné z: `https://nodejs.org/docs/latest-v14.x/api/addons.html`

[16] Release 5.1.0 - vercel/pkg. *GitHub* [online]. c2021, 2021 [cit. 2021-05-19]. Dostupné z: `https://github.com/vercel/pkg/releases/tag/5.1.0`