Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Příprava kurzu Platforma JavaScript

2022

Bc. Jan Kvapil

Vedoucí práce:
RNDr. Martin Trnečka, Ph.D.

Studijní program: Aplikovaná informatika,
Specializace: Vývoj software

Bibliografické údaje

Autor:	Bc. Jan Kvapil
Název práce:	Příprava kurzu Platforma JavaScript
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2022
Studijní program:	Aplikovaná informatika, Specializace: Vývoj software
Vedoucí práce:	RNDr. Martin Trnečka, Ph.D.
Počet stran:	32
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliographic info

Author:	Bc. Jan Kvapil
Title:	Preparation of JavaScript Platform Course
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2022
Study program:	Applied Computer Science, Specialization: Software Development
Supervisor:	RNDr. Martin Trnečka, Ph.D.
Page count:	32
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Cílem této diplomové práce je vytvoření kurzu zaměřeného na platformu JavaScript, který by bylo možné zařadit do studijního plánu navazujícího magisterského programu Aplikovaná informatika. Kurz by měl být zaměřen na moderní použití JavaScript technologií. Součástí práce je i příprava veškerých studijních materiálů, slidů, skript a řešených příkladů.

Synopsis

The aim of this diploma thesis is to create a course focused on the JavaScript platform, which could be included in the study plan of the follow-up master's program Applied Informatics. The course should focus on the modern use of JavaScript technology. Part of the work is also the preparation of all study materials, slides, scripts and solved examples.

Klíčová slova: JavaScript kurz; moderní webové technologie

Keywords: JavaScript course; modern web technologies

Chtěl bych poděkovat RNDr. Martinu Trnečkovi, Ph.D. za ochotný přístup a užitečné připomínky ke zpracování této práce.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Úvod

Hlavní motivací pro vytvoření této diplomové práce byla absence magisterského kurzu zaměřeného především na JavaScriptové technologie. Předmět by měl představovat alternativu k již existujícím kurzům typu *Platforma .NET (KMI/PNE)* nebo *Platforma Java (KMI/PJA)*. Kvůli mohutnému rozšíření technologie *Node.js* lze na JavaScript nahlížet jako na plnohodnotnou platformu pro vývoj aplikací.

Kapitoly textu tématicky odpovídají jednotlivým seminářům. V úvodu bude vždy shrnut obsah semináře, po kterém bude následovat teorie k dané problematice. Text práce by měl sloužit jako příručka pro vyučujícího, proto zde není příliš kladen důraz na vysvětlení všech základních pojmů. Tato práce pokrývá pouze stěžejní témata, kterým by se kurz měl věnovat. Samotné semináře by z nich měly vycházet, ale měly by být zároveň obohaceny o řadu doplňujících informací (individuálně dle preferencí vyučujícího).

Cíle předmětu

Cílem předmětu bude seznámit studenty s použitím JavaScriptových technologií na klientovi (webový prohlížeč) a serveru. V průběhu kurzu budou představeny nástroje a návrhové vzory, které vývoj aplikací usnadní.

Obsah kurzu

V úvodu se studenti seznámí s novinkami v rámci samotného jazyka dle nejnovějšího *ECMAScript* standardu. Stěžejní je samotná práce s *Node.js* knihovnami a tvorba jednoduchých webových služeb na straně serveru. Dále se kurz zaměřuje na vývoj frontendu, práci s databází a tvorbu *API*. Druhá polovina kurzu seznámí studenty s možnostmi práce s *websockety* a *Message Queue* knihovnami. Následovat budou další užitečné nástroje pro vývoj aplikací a práci s aplikačním stavem. Jelikož je v dnešní době možné použít JavaScript pro téměř jakékoliv zařízení, je zde zařazena i část, zabývající se vývojem multiplatformních aplikací. V samotném závěru jsou pak nastíněny možnosti funkcionálního přístupu v rámci JavaScript platformy.

Studijní materiály

Součástí této práce jsou také skripta [1], která představují hlavní studijní oporu. Skripta poskytují studentům ke každé kapitole úvod do problematiky, sadu řešených příkladů, včetně popisu řešení a referencí na další studijní materiály.

Předpoklady

Pro úspěšné absolvování kurzu by měl mít student alespoň základní zkušenosti s tvorbou webových aplikací. Měl by rozumět principům z oblasti databázových systémů a počítačových sítí (zejména aplikační vrstvě).

1 Úvod do JavaScriptu

V úvodním semináři proběhne seznámení s obsahem kurzu a podmínkami pro jeho absolvování. Podstatné bude nastavení vývojového prostředí (včetně instalace běhového prostředí Node.js) pro testování ukázek kódu ze skript a plnění zadaných úkolů.

Po dokončení technických záležitostí přijde na řadu téma první lekce. Jelikož je pravděpodobné, že se studenti s jazykem JavaScript již setkali v rámci jiných kurzů, je důležité sjednotit různou úroveň znalostí zopakováním základů tohoto jazyka. Mimo zopakování syntaxe a sémantiky je vhodné uvést tuto technologii také do historického kontextu v rámci vývoje webu. S tím také souvisí nutnost poukázat na rozdíly mezi verzemi, včetně nových vlastností, které přibyly s aktuální specifikací standardu ECMAScript.

1.1 Vývojové prostředí

Pro účely testování příkladů ze skript je možné použít například webové IDE Repl.it [2]. Preferovaným editorem je VSCode [3], který spolu s Linuxovou konzolí (případně alternativou pro Windows v podobě MinGW) bude sloužit jako základní nástroj pro vývoj JS[1] aplikací. Nutnou součástí bude samozřejmě také JavaScriptové běhové prostředí Node.js [4] spolu s balíčkovacím systémem *Yarn*. Kvůli testování příkladů v kapitole o staticky typovaném JavaScriptu bude užitečné mít nainstalované také běhové prostředí pro TypeScript – Deno [5].

1.2 JavaScript

První z příkladů se zaměřuje na rozdíly v rozsahu platnosti proměnných a funkcí. Následovat bude práce s funkcemi vyšších řádů. Dále budou představeny základní datové struktury, které jsou realizovány jako referenční datové typy. Z toho plynou určitá omezení, která se projeví zejména při jejich duplikaci nebo modifikaci. Dále budou porovnány destruktivní a nedestruktivní operace nad datovými strukturami. V poslední části se zaměříme na možnosti práce s asynchronními funkcemi, včetně proxy v podobě příslibu (Promise [6]) pro předem neznámé hodnoty.

V průběhu lekce bude kladen důraz na pochopení specifických vlastností tohoto jazyka. Student by měl chápat omezení plynoucí ze slabého typového systému a dvojího rozsahu platnosti, dále by měl umět pracovat s referenčními datovými typy a asynchronním kódem.

[1] V textu se bude často používat místo názvu programovacího jazyka JavaScript zkratka JS.

2 Node.js

Jednou z hlavních výhod JavaScriptu je možnost sdílení kódu na klientu i serveru. Tématem druhého semináře je právě JS na straně serveru. K tomu bylo vytvořeno běhové prostředí Node.js [4], které by měli mít studenti připravené již z minulého semináře.

Na začátku semináře bude představen balíčkovací systém Npm, který je instalován společně s Node.js. Dále budou studenti seznámeni s možnostmi práce s asynchronními funkcemi na straně serveru. Následovat bude vytvoření webového serveru, obsluhujícího HTTP požadavky. Na základě požadavků bude server vracet jak požadovaná data, tak i HTML[2] stránku spolu s načtenými daty.

2.1 Npm

Npm neboli *Node Package Manager* je balíčkovací systém pro Node.js, který umožňuje spravovat závislosti na JavaScriptových knihovnách. V době psaní této práce čítá okolo 1.5 milionu knihoven. V porovnání s ostatními platformami je sice největším balíčkovacím systémem, s tím je však spojeno také určité riziko a ne každou knihovnu je vhodné používat. Obecně je lepší spoléhat na dlouhodobě vyvíjené a masivně používané knihovny s velkým zázemím a podporou ze strany komunity, případně knihovny, za kterými stojí velké společnosti a tudíž by měly zaručovat určitou kvalitu a podporu do budoucna. V rámci tohoto kurzu budou prezentované technologie tyto vlastnosti splňovat v co největší míře.

2.2 Axios

Jelikož v rámci serverové části nemáme k dispozici metodu `fetch`, která je standardně implementovaná v rámci API prohlížeče, je nutné zvolit jiný způsob realizace asynchronních požadavků. Jednou z možností je použití externí knihovny Axios [7]. Na té si v rámci prvního příkladu ve skriptech ukážeme práci s externími knihovnami.

2.3 Express

Další základní knihovnou, na kterou se v rámci semináře zaměříme, je knihovna Express [8]. Ta umožňuje snadno vytvořit vlastní webový server, obsluhující HTTP požadavky, na kterém bude demonstrováno základní použití technologie Node.js.

[2]Tento přístup se nazývá Server Side Rendering (SSR). Server na základě požadavku vrací vygenerovanou HTML stránku. Jednou z technologií, která vychází primárně z tohoto přístupu, je například technologie PHP.

3 React I

Tématem této lekce bude vývoj frontendu pomocí knihovny React. V úvodu budou zmíněny aktuální trendy v rámci vývoje uživatelského rozhraní (UI) a rozdíly mezi jednotlivými přístupy UI knihoven. Poté se zaměříme konkrétně na knihovnu React a její samotné fungování. Následovat bude seznámení s frameworkem Next, který staví nad knihovnou React. Mimo tvorby frontendu je v rámci Next.js skvěle integrovaná také tvorba API. Posledním tématem tohoto semináře pak bude zejména tvorba REST API,[3] na které se později naváže pokročilejším *GraphQL* API.

3.1 Knihovny pro vývoj UI

V rámci vývoje uživatelského rozhraní (v době psaní této práce) dominují v zásadě tři hlavní knihovny – *Angular*, *Vue* a *React*. Nejstarší z nich je knihovna Angular [10], vyvíjená společností Google. Ta upřednostňuje přístup, který vychází spíše ze šablonovacích systémů. Zavádí do HTML pomocné atributy, které realizují například cyklus průchodu pole a následné vykreslení HTML elementů z daného pole. Obohacuje tak HTML o určitou nadstandardní funkcionalitu.

Naopak nejnovější a v současné době velmi populární knihovnou je Vue [11]. Její nevýhodou je, že momentálně prochází poměrně razantním vývojem a vytrácí se z ní minimalismus předešlých verzí. Navíc se jedná spíše o komunitní projekt, za kterým nestojí žádná velká firma.

```
1  <App>
2    <Header />
3    <Content>
4      ...
5    </Content>
6    <Footer />
7  </App>
```

Zdrojový kód 1: Kompozice React komponent

3.2 React

Poslední z výše jmenovaných knihoven je React [12]. Samotná knihovna je velmi minimalistická. Hlavní myšlenkou je odstínění programátora od práce s objektovým modelem HTML dokumentu - *DOM (Document Object Model)*.

[3] REST (Representational State Transfer) [9] je architektura rozhraní, navržená pro distribuované prostředí. Rozhraní REST je použitelné pro jednotný a snadný přístup ke zdrojům (resources). Zdrojem mohou být data, stejně jako stavy aplikace (pokud je lze popsat konkrétními daty). REST je tedy, na rozdíl od známějších XML-RPC či SOAP, orientován datově, nikoli procedurálně.

3.3 Reconciliation

V Reactu komponenta = JavaScriptová funkce, která transformuje data na výsledné HTML, neboli *View*. V průběhu práce s aplikací se pak samotná data mění a React přepočítává změny pomocí *Diffing algoritmu* [13], na základě kterých ve vhodné chvíli překresluje View.

$$f(d_0) = v_0$$

$$df(d_0, d_1) = v_1$$

Aby se zajistilo efektivní překreslení pouze těch komponent, ve kterých došlo ke změně, je zapotřebí zavést koncept virtuálního DOMu. Virtuální DOM představuje abstraktní reprezentaci aktuálního stavu uživatelského rozhraní. Ve virtuálním DOMu se změny provedou okamžitě, zatímco do reálného DOMu se promítnou až ve chvíli, kdy má dojít k celkovému překreslení. Tento proces se nazývá *Reconciliation* [13].

React je čistě client-side technologie. Dá se jednoduše vložit do webové stránky pomocí skriptu a napojit na určitý element, který má být kořenem stromu React komponent. Tímto způsobem se však React v rámci tohoto kurzu používat nebude. Je však dobré vědět, že je možné jej použít bez problému v již existující aplikaci.

3.4 Next.js

Jelikož React řeší pouze překreslování View, pro účely tohoto kurzu je vhodnější použít plnohodnotný framework pro tvorbu webových aplikací – *Next.js* [14]. Jeho základ tvoří právě knihovna React spolu s řadou dalších nástrojů a knihoven. Obsahuje vlastní router, který poněkud vytlačuje typický SPA přístup při klasickém použití Reactu. Kvůli *SEO*[4] optimalizaci používá ve výchozím nastavení statické předgenerování stránek (SSG), které jsou vytvořeny v okamžiku sestavení aplikace. Další možností je server-side rendering (SSR), díky kterému může dynamicky předgenerovat HTML stránku na serveru při každém requestu. Samozřejmě je možné použít také (pro React typický) client-side rendering (CSR). Pokročilou vlastností je pak statický HTML export, který umožňuje z aplikace vygenerovat statické HTML stránky, které je možné provozovat bez Node.js serveru. To však nese určitá omezení spojená s absencí serverové části.

[4]SEO (Search Engine Optimalization) umožňuje vyhledávačům lépe indexovat stránku. Proto se při vývoji webových aplikací často používají techniky, které mohou indexaci ovlivnit a přimět vyhledávače zobrazovat požadovanou stránku na dřívějších pozicích ve vyhledávání.

4 React II

V návaznosti na předešlý seminář se budeme nyní zabývat správou aplikačního stavu v rámci React aplikace. Důležitým bodem bude zejména správné překreslování View v závislosti na změnách aplikačního stavu. Dále se zaměříme na problémy, které mohou nastat při nevhodné práci s aplikačním stavem.

Od aplikačního stavu následně přejdeme ke stylování aplikace importováním externích CSS souborů, ale také pomocí inline CSS v JavaScriptu. Téma stylování bude probráno však pouze okrajově, jelikož tento kurz není primárně o návrhu uživatelského rozhraní nebo práci s vizuální stránkou aplikace.

Často není žádoucí psát veškeré React komponenty od základu ručně. Posledním tématem této lekce proto bude použití komponent třetích stran. S tím by mělo být také spojeno doporučení, který typ knihoven je vhodné používat a kterým je naopak lepší se vyhnout.

4.1 UseState

Základním *hookem*,[5] se kterým se při práce s aplikačním stavem v Reactu setkáme, je *useState*. Ten umožňuje komponentě udržovat vnitřní stav, který při jeho změně vyvolá požadavek na překreslení komponenty. Při práci s hookem useState je nutné používat pouze nedestruktivní operace nad datovými strukturami. V opačném případě React nebude správně překreslovat View.

4.2 UseEffect

Dalším ze základních hooků je *useEffect*. Ten umožňuje dynamicky reagovat na změny stavu aplikace. Mimo propojení s určitými daty, na jejichž změnu chceme reagovat, umí hook useEffect reagovat také na změny životního cyklu komponenty. To se hodí použít třeba ve chvíli, kdy chceme zavolat funkci po načtení a vykreslení celé komponenty.

4.3 UseSWR

Jedním z hooků, který není součástí Reactu a představuje tzv. *"custom hook"*, je *useSWR*. Jedná se o hook, který realizuje asynchronní načítaní dat a následnou změnu aplikačního stavu. Tento hook je součástí knihovny třetí strany. Custom hooky umožňují extrahovat aplikační logiku do znovupoužitelných funkcí. V tomto případě však useSWR poskytuje pouze abstrakční vrstvu, usnadňující asynchronní načítání dat. Kombinuje tak několik základních hooků dohromady.

[5] Hooky jsou funkce, které umožňují pracovat s aplikačním stavem a vedlejšími efekty v rámci funkcionální React komponenty.

5 Databáze I

Zásadní roli ve vývoji aplikací hraje perzistentní uložení dat. Tématem tohoto a následujícího semináře bude práce s datovou vrstvou aplikace prostřednictvím knihoven nižší úrovně až po *objektově relační mapování* databázových entit. Základem je práce s relačními databázemi, demonstrována pomocí databázového systému *SQLite* [15]. Ta je vhodná především kvůli velmi snadnému zprovoznění bez nutnosti instalace jakéhokoliv dalšího software. Příklady ve skriptech popisují tvorbu REST API. Prostřednictvím API bude možné pracovat s databází pomocí *query builderu* na základě přijatých HTTP requestů.

5.1 Abstrakční vrstvy

K databázi lze přistupovat pomocí knihoven na různých úrovních abstrakce. Míra abstrakce reflektuje jednoduchost a rychlost vývoje, ale také vnáší určitá omezení. Zde si definujeme základní kategorie přístupů:

- **ORM**, neboli objektově relační mapování databázových entit poskytuje nejvyšší úroveň abstrakce při práci s databází. Lze mapovat celé tabulky, nebo pouze výsledky dotazů, přináší však určitá rizika ve formě neefektivity dotazů (*n+1 problem* [16]).
- **Query/Schema buildery** - knihovny pro tvorbu dotazů a schémat sjednocují zápis dotazů a příkazů pro různé typy databázových systémů pomocí JS syntaxe. Knihovny generují a vykonávají výsledné dotazy na pozadí. Odstiňují tak uživatele od přímého přístupu k databázi.
- **Raw SQL** - ne vždy je možné spolehnout se na vyšší úroveň abstrakce a je tedy nutné přistupovat k databázi přímo - v tomto případě pomocí SQL. Díky knihovnám nejnižší úrovně (databázovým *driverům*) je možné psát přímo SQL dotazy, které jsou pro určité situace nevyhnutelné (optimalizace dotazů).
- **Database-First** - způsobu práce s již existující databází říkáme *Database-First* přístup. Databázové entity je možné namapovat a vygenerovat z nich aplikační model - realizován pomocí tříd nebo typů.
- **Code-First** - tento přístup umožňuje pomocí kódu celou databázi vygenerovat. Například v rámci platformy .NET se často používá tak, že se vytvoří model databáze pomocí tříd a z něj se poté generuje samotná struktura databáze. Z jednotlivých úprav těchto tříd vznikají tzv. *migrace*. Jedná se o části kódu, které transformují databázové schéma do nové podoby.
- **Schema-First** - velmi populární je použití alternativních databázových schémat a *SDL* (*Schema definition language*), ze kterých lze generovat jak samotnou strukturu databáze, tak aplikační třídy.

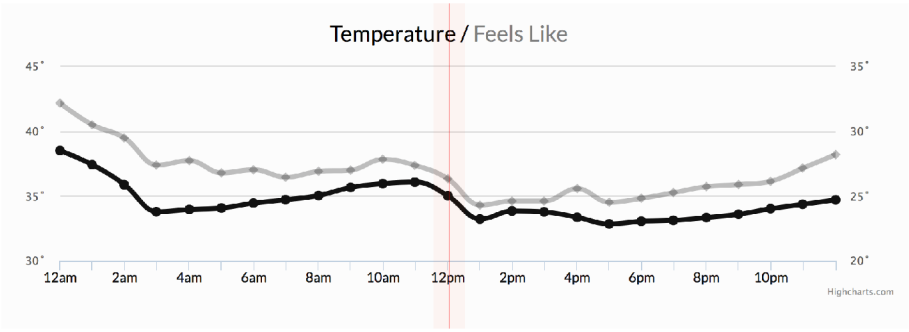
6 Databáze II - MongoDB

Jednou z nejpopulárnějších alternativ klasického relačního databázového modelu je právě model dokumentový. Jeho zástupcem pro účely tohoto semináře bude technologie MongoDB [17], která používá dokumenty ve formátu *BSON* (*Binary-encoded Javascript Object Notation*) a kvůli skvělé integraci s Node.js je proto vhodným kandidátem. Výhodou tohoto modelu je jeho jednoduchost, rychlost, robustnost a snadná škálovatelnost - jak vertikální (rychlejší hardware, více paměti), tak i horizontální (více distribuovaných databází běžících zároveň). Na druhou stranu nepodporuje některé z typických vlastností relačních databázových systémů (např. použití jazyka SQL, triggerů nebo procedur). Součástí této lekce bude také demonstrace přístupu k databázi MongoDB, jak pomocí nízkoúrovňového databázového driveru, tak i s použitím ORM.

Speciálním případem nerelačních databází jsou *databáze časových řad*, které jsou v rámci systému MongoDB podporovány. V poslední části semináře budou představeny základy práce s tímto typem databáze.

6.1 Databáze časových řad (TSDB)

V době masivně se rozšiřujícího *IoT*[6] bylo zapotřebí přizpůsobit i možnosti ukládání obrovských objemů dat z nejrůznějších senzorů. Totéž například platí i pro nástroje monitorující pohyb cen akcií, kryptoměn a dalších finančních instrumentů. K tomu slouží *TSDB* neboli *time series databases*. Ty jsou optimalizovány pro ukládání velkých objemů dat, kde jsou hodnoty asociovány s časovým razítkem, kdy byla tato data pořízena. Typickým zástupcem je například *InfluxDB* [19]. Pro účely tohoto kurzu si však vystačíme s MongoDB. V rámci MongoDB jsou nyní k dispozici tzv. *Time Series Collections*.



Obrázek 1: TSDB příklad [19]

[6]Internet věcí (anglicky Internet of Things, zkratka IoT) [18] je v informatice označení pro síť fyzických zařízení, vozidel, domácích spotřebičů a dalších zařízení, která jsou vybavena elektronikou, softwarem, senzory, pohyblivými částmi a síťovou konektivitou, která umožňuje těmto zařízením propojit se a vyměňovat si data.

6.2 Terminologie TSDB

Abychom porozuměli základní terminologii v rámci časových řad, mějme tento graf (obrázek č. 1) jako příklad.

- **Měření** (Measurement) je v tomto případě teplota (Temperature) a představuje kolekci dat
- **MetaField** (Label) je označení časové řady (pocitová teplota/reálná teplota)
- **Granularita** - hustota naměřených dat v určitém časovém období (teplota měřená každou hodinu)
- **Expirace** - pokud není nutné uchovávat celou historii dat, lze nastavit, po jakém časovém úseku budou data promazávána

7 GraphQL

GraphQL je dotazovací jazyk nad libovolným datovým zdrojem. Používá dvě základní operace - *Query* (vrací data) a *Mutace* (změna v datech). GraphQL API je určitým rozšířením klasického REST API. Obsluhuje však pouze jediný endpoint `/graphql`, přes který proudí veškeré dotazy.

```
1    query {
2      users {
3        id
4        name
5      }
6    }
```

Zdrojový kód 2: GraphQL dotaz

Pro všechna data, která chceme vystavit v rámci GraphQL API, je nutné vytvořit *resolvery* - funkce vracející data z databáze. Totéž platí i pro mutace. Query a Mutation jsou základní typy - *Root type*.[7] Syntaxe může připomínat formát JSON, ve kterém jsou definovány pouze názvy atributů, pro které chceme získat hodnoty.

Zásadní roli při práci s GraphQL API hraje GraphQL schéma. Jedná se o popis API pomocí SDL (GraphQL schema language). Schéma API tvoří typy (např. typ User). Resolveru pak můžeme předat parametry daného typu, které chceme

[7] Root type, neboli kořenový typ, představuje základ pro vytváření dalších typů. Konvencí je vytvářet dotazy nad kořenovým typem Query. Příkazy, které provádí změnu v datech (v terminologii HTTP metod operace POST, PUT, REMOVE a PATCH), se vytváří pod kořenovým typem Mutation

načíst. Schéma je volně dostupné v rámci GraphQL endpointu, tudíž je možné jej načíst a zjistit tak strukturu samotného API. V kapitole o TypeScriptu se budeme zabývat pokročilými možnostmi práce se schématem jako je automatické generování typů. Verzování GraphQL API by mělo dodržovat určitou zásadu "evolvingu". Parametry by se neměly měnit, ale pouze přidávat. Nemůže pak dojít k situaci, kdy klient bude požadovat data s neexistujícími atributy.

```
type Query {
  users: [User]
}

type User {
  id: Int!
  name: String!
  posts: [Post]
}

type Post {
  id: Int!
  title: String!
  text: String
  authorId: Int!
}
```

Zdrojový kód 3: GraphQL schéma

```
query {
  users {
    id
    name
    posts {
      id
      title
    }
  }
}
```

Zdrojový kód 4: GraphQL dotaz 2

Zřejmě největší výhodou oproti RESTu je možnost načítat všechna potřebná data v jednom dotazu, odeslaném na GraphQL server. Navíc lze jednoduše parametrizovat dotaz pomocí různých restrikcí a filtrů nad daty. Velmi intuitivní je také práce se spojenými daty (spojené tabulky v rámci relačního modelu). Lze tak velmi jednoduše vytvářet komplexní dotazy, které ale zároveň mohou být načítány velmi efektivně. Zajistit takovou variabilitu a efektivitu dotazování nad klasickým REST API je téměř nemožné. Typickým problémem, spojeným

s neefektivním dotazováním nad RESTovými službami, je například "overfetching", kdy je v jednom dotazu načítáno zbytečně mnoho dat. Opačným problémem je "underfetching", kdy je naopak nutné vytvořit více dotazů, abychom dostali všechna požadovaná data - což znamená další režii.

8 WebSocket & Message Queue

Cílem tohoto semináře bude seznámit studenty s možnostmi real-time komunikace, jak v rámci webového prohlížeče (pomocí websocketů), tak i na straně serveru (pomocí Message Queue knihoven).

8.1 WebSocket

Se zvyšujícími nároky na webové aplikace a potřebou real-time komunikace mezi prohlížečem a serverem (chatovací aplikace, hry v prohlížeči nebo streamovací aplikace), se do prohlížečů zaintegrovalo WebSockets API [20], implementující komunikační protokol WebSocket [21] (RFC 6455). Jedná se o plně duplexní způsob komunikace pomocí protokolu TCP. Komunikace se zahájí vytvořením komunikačního kanálu, který je řízen událostmi. Událost může představovat například přijetí zprávy nebo výskyt chyby.

8.2 WebSocket API

Cílem semináře bude seznámit studenty s WebSocket API - jak v čisté podobě, tak s nadstavbami, usnadňujícími práci s websockety. Stěžejní je také schopnost rozlišit vhodnost použití websocketů, nebo případných nadstaveb, pro konkrétní případy užití v porovnání s jinými možnostmi komunikace (jako je REST).

8.3 Socket.io

Socket.io je nadstavba nad websockety pro prohlížeč i server. Jedná se o externí knihovnu, která poskytuje řadu nadstandardních funkcí spolu s jednodušším rozhraním pro práci s websockety.

Nevýhodou použití Socket.io v prohlížeči je však nutnost stažení externí knihovny (na rozdíl od základní WebSocket API). To se může negativně projevit například u mobilních zařízení s omezeným přístupem k internetu, kde každá externí knihovna navíc znamená znatelnou zátěž.

8.4 Message Queue knihovny

Message Queue (MQ)[8] knihovny nacházejí široké uplatnění v distribuovaných systémech. Zajišťují komunikaci (obecně) mezi producenty a konzumenty pomocí různých komunikačních patternů. Časté použití MQ knihoven je například pro integraci systémů, kdy je možné rozdělit systém do menších celků (služeb) a zajistit mezi nimi komunikaci prostřednictvím MQ knihoven. Jednotlivé části systému mohou být implementovány nad libovolnou platformou (Java, .NET, Python). Typické oblasti použití MQ knihoven mohou být třeba monitorovací systémy (obecně IoT software) nebo programy pro analýzu trendů akcií na burze, případně algoritmické obchodování.

8.5 Základní komunikační patterny

Typicky se využívá komunikace pomocí zpráv na úrovni socketů. Na rozdíl od websocketů, u MQ knihoven počítáme s použitím výhradně mimo prohlížeč. V rámci MQ knihoven je možné používat jak synchronní, tak asynchronní komunikaci, případně streamování zpráv. Níže si představíme výčet základních komunikačních patternů.

- **Request/Reply** - blokující komunikace, konzument čeká na odpověď od producenta
- **Push/Pull** (Pipeline pattern) - neblokující komunikace, zprávy jsou ukládány na zásobník. Ve chvíli, kdy je konzument dostupný, si uložené zprávy postupně vyzvedne
- **Publish/Subscribe** - neblokující komunikace, "rádio" - producent vysílá zprávy, konzument se připojí na "kanál" a začne zprávy přijímat

8.6 ZeroMQ

ZeroMQ [22] je velmi jednoduchá a rychlá MQ knihovna. Má velkou podporu napříč programovacími jazyky a platformami, nicméně je omezena spíše na nižší úroveň komunikace mezi jednotlivými uzly. Výhodou je, že nepotřebuje žádný middleware (brokera, jako např. RabitMQ - proto předpona Zero). Může tak fungovat i na té nejjednodušší úrovni peer-to-peer komunikace implementovatelné pomocí několika řádků kódu.

Další výhodou je právě odstínění od nízko-úrovňových problémů, jako je například výpadek spojení. Jestliže u publishera dojde k selhání (výpadku internetového spojení), subscribers čekají, dokud publisher znovu neobnoví svoji činnost. Knihovna ZeroMQ sama ošetří situace neočekávaného ukončení spojení.

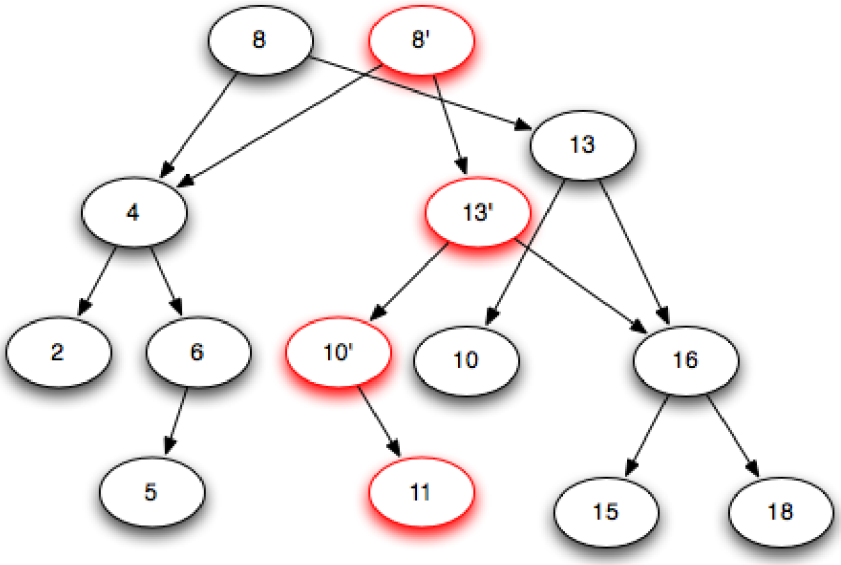
[8] Zde je nejspíš zkratka MQ obsahující "Queue" poněkud zavádějící, jelikož pouze v patternu Push/Pull, jak zjistíme později, figuruje fronta jako stěžejní prvek.

9 Aplikační stav

V úvodních lekcích o Reactu se studenti seznámí s prací s aplikačním stavem. Zásadním pravidlem pro práci s aplikačním stavem v rámci Reactu je použití nedestruktivních operací nad daty. Destruktivní manipulace přináší řadu problémů spojených se snadnějším zanášením programátorských chyb, nepředvidatelným chováním a také znemožněním správného překreslování React komponent. Mezi nedestruktivní operace nad polem patří funkce jako `map`, `filter` nebo `reduce`. Ty nám vracejí vždy pole nová, nemodifikují původní.

9.1 Imutabilní datové struktury

Zavedením imutabilních (neměnných) datových struktur zajistíme, že nebude možné s aplikačním stavem pracovat destruktivně. Při modifikaci objektu dochází k vytvoření objektu nového. To by se mohlo zdát jako náročná operace. Nicméně imutabilní datové struktury mají vlastnost Structural sharing. Jedná se o struktury typu Persistent Search Tree.[9]



Obrázek 2: Vložení prvku 11 do perzistentního vyhledávacího stromu [24]

[9]Perzistentní vyhledávací strom [23] si pamatuje historii všech svých změn a umí vyhledávat nejen v aktuálním stavu, ale i ve všech stavech z minulosti. Přesněji řečeno, po každé operaci, která mění stav stromu, vznikne nová verze stromu a operace pro dotazy dostanou jako další parametr identifikátor verze, ve které mají hledat. Strom spotřebuje $O(n \log n)$ paměti a dotazy vyřizuje v čase $O(\log n)$. Perzistence datových struktur je přirozená pro striktní funkcionální programovací jazyky (například Haskell). V nich neexistují vedlejší efekty příkazů, takže jednou sestrojená data již nelze modifikovat, pouze vyrobit novou verzi datové struktury s provedenou změnou.

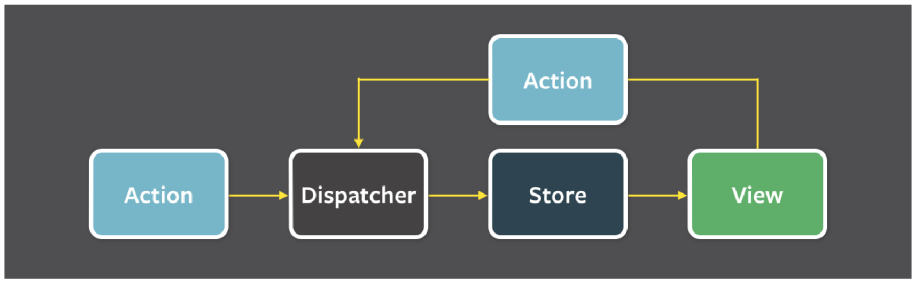
Obrovským přínosem za cenu přijatelné režie je robustnost, kterou tento přístup poskytuje. Dále se pro React proces počítání změn (Reconciliation) výrazně urychlí, jelikož se nemusí provádět porovnávání zanořených objektů, ale pouze porovnání reference a snadnější lokalizace elementu v rámci DOM, který se má překreslit.

Některé datové struktury (viz poznámka pod čarou na předešlé stránce) mají v sobě implementovanou možnost uchovat historii všech změn. To by představovalo opravdu silný nástroj pro případné debugování, kdy lze poslat report o sledu aplikačních změn, které vedly k chybě za běhu programu. Cílem lekce, odpovídající této kapitole, bude k takovémuto řešení dojít pomocí JS knihoven.

9.2 Flux

Architektura Flux [25] sjednocuje práci s aplikačním stavem na globální[10] úrovni s využitím imutabilních datových struktur. Stejně jako React, pochází tento návrhový vzor od vývojářů z Facebooku.

Základními stavebními prvky jsou *Actions*, *Store* a *Dispatcher*. Akce představují typy událostí, které pomocí Dispatcheru mění aplikační stav, který je udržován ve Store (na jednom místě v aplikaci).



Obrázek 3: Flux: Data flow diagram [25]

Všechny změny aplikačního stavu jsou transparentní, jelikož Flux používá jednosměrný tok dat. Je jasně daná posloupnost akcí, které vedly k aktuálnímu stavu aplikace. Všechny akce proudí skrz Dispatcher, který může akce ukládat a mít tak záznam o tom, co se v aplikaci stalo. Jak bylo naznačeno v úvodu této kapitoly, tato možnost je skvělým nástrojem pro debugování. Mimo to je možné jednoduše implementovat například operace jako Undo nebo Redo.

[10] Jelikož Store by měl být ze své podstaty imutabilní, mělo by být možné k němu bez omezení přistupovat odkudkoliv z aplikace i bez použití Dispatcheru. K tomu se dá použít třeba nástroj Use-global-hook [26]. Samotnou imutabilitu datových struktur je možné zajistit například pomocí knihovny Immer [27]. Kombinací takovýchto dvou nástrojů můžeme získat minimalistického správce aplikačního stavu bez nutnosti implementace jednotlivých akcí pro jakoukoliv změnu v datech. Funkce, vykonávající změny na základě přijatých akcí, se nazývají "reducery".

10 Staticky typovaný JavaScript

Jednou z problematických charakteristik JavaScriptu je slabá dynamická typovost, díky které často dochází k chybám typu "undefined is not a function". Ty mohou být zapříčiněny například chybným předáním parametrů funkce a bývají odhaleny většinou až za běhu programu. Přidáním statické kontroly typů lze získat nejen oznámení případných chyb při kompilaci, ale zároveň také automatické doplňování a nápovědu již při psaní kódu.

Nicméně je třeba stále brát v potaz, že data, která jsou získána ze serveru, nemusejí vždy odpovídat struktuře, kterou si programátor jednou, v rámci aplikace, nadefinuje. Řešením může být například použití monorepozitáře.[11]

10.1 TypeScript

TypeScript [29] je nadmnožina JavaScriptu, vytvořena společností Microsoft. Obohacuje JavaScript o definici typů, rozhraní a typové anotace. Jedná se o kompilovaný jazyk, takže jej není možné použít přímo v prohlížeči. Studenti by měli pochopit výhody i kompromisy, spojené s použitím TypeScriptu v rámci platformy JS, a dále znát alternativní přístupy pro statickou kontrolu typů.

```
type Person = {
  name: String
  age: Number
}
```

Zdrojový kód 5: TypeScript

10.2 Schémata

Předávání informací o datových typech mezi klientem a serverem lze obecně řešit pomocí schémat. V kapitole o GraphQL jsme si jedno ze schémat již ukázali. Jedním z témat semináře, který se váže k této kapitole, je také automatické generování typů z GraphQL schématu.

Mimo GraphQL schéma existují také JSON Schema/OpenAPI,[12] které umožňují ověřovat konzistenci dat za běhu. Protokol Transit umožňuje posílat data (JSON) zároveň s informacemi o typech.

[11] Monorepozitář [28] je speciální případ repozitáře, kdy je více logicky nezávislých projektů vyvíjeno a jejich kód uchováván v rámci jednoho repozitáře. Protože tento způsob umožňuje snadné sdílení kódu mezi jednotlivými podprojekty, často se jedná o společný vývoj backendu a frontendu jedné aplikace, případně o vývoj několika frontendových klientů jedné aplikace. Na rozdíl od aplikace, kde jsou jednotlivé moduly rozděleny mezi více repozitářů a kde může být například testování více modulů napříč repozitáři komplikované, je testování aplikace v rámci jednoho repozitáře jednodušší.

[12] JSON Schema/OpenAPI nacházejí využití především pro popis dat RESTových endpointů.

10.3 Deno

Nevýhodou použití TypeScriptu v Node.js je nutnost kompilace. Deno [5] je běhové prostředí pro TypeScript. Je vytvořeno autorem Node.js a stejně jako Node.js má Deno svůj REPL a interpret. Zatím se jedná o čerstvý projekt, ale má potenciál zásadním způsobem nahradit Node.js na serveru.

10.4 Algebraické datové typy

K definici typů lze využít mimo kompozici také další z matematických operací jako sjednocení (Union type) nebo průnik (Intersection type).

```
type Gender = 'male' | 'female'

type Person = {
  name: String
  age?: Number
  gender: Gender
}

const john: Person = {
  name: 'John',
  gender: 'male'
}
```

Zdrojový kód 6: Union type

V tomto případě funguje Gender podobně jako výčtový typ. Pokud bychom předali jiný řetězec než "male" nebo "female", překladač by nám zahlásil chybu.

```
type Props = {
  visible: boolean
}

const Popup = (props: Props & React.HTMLAttributes<HTMLDivElement>)
    => (
  <div {...props} style={ props.visible ? {display: 'block'} :
      {display: 'none'}}>
    { props.children }
  </div>
)

export default Popup
```

Zdrojový kód 7: Intersection type

Intersection type je možné použít například pro popis rozhraní React komponenty, která má zároveň převzít vlastnosti nějakého obecného elementu.

10.5 JSDoc

V JavaScript ekosystému se používá standardně pro psaní dokumentace knihovna JSDoc [30]. Typy se dají považovat za součást dokumentace. Vývojová prostředí jako například VSCode umožňují pomocí doplňků (některá i nativně) analyzovat komentáře a poskytovat automatickou nápovědu a statickou kontrolu typů na základě typových anotací v komentářích. Z komentářů lze následně vygenerovat dokumentaci v podobě HTML dokumentů.

10.6 Domain Driven Design

Jedním z návrhových vzorů, benefitující z bohatého typového systému, je právě *Domain Driven Design* (dále DDD), který je úzce spojen s aplikační logikou.

Pomocí algebraických datových typů lze snadno definovat aplikační doménu, typy se stanou určitou dokumentací aplikační logiky, snadno čitelnou i pro lidi bez znalostí programování. Zároveň také programátorům názvy typů naznačují, jaké hodnoty proměnné daného typu mohou obsahovat. Samotný název typu by měl být co nejspecifičtější.

Vezměme si například funkci dělení. Zavedeme typ `NonZeroNumber`, z jehož názvu vyplývá, že pro dělitel není hodnota nula definovaná.

```
1 type NonZeroNumber = Number
2 type Divide = (a: Number, b: NonZeroNumber) => Number
```

Zdrojový kód 8: DDD - Příklad dělení

Funkce dělení je primitivní příklad použití DDD, pravou sílu odhalí až s komplexnější architekturou doménového modelu. Vhodnějším příkladem může být doménový návrh karetní hry. V přístupu DDD se začíná právě návrhem typů, na základě kterých se poté staví samotná aplikace.

```
1  type Suit = 'Club' | 'Diamond' | 'Spade' | 'Heart'
2  type Rank = 'Two' | 'Three' | 'Four' | 'Five' | 'Six' | 'Seven' |
3     'Eight' | 'Nine' | 'Ten' | 'Jack' | 'Queen' | 'King' | 'Ace'
4  type Card = [Suit, Rank]
5  type Hand = Array<Card>
6  type Deck = Array<Card>
7  type Player = { name: String, hand: Hand }
8  type Game = { deck: Deck, players: Array<Player> }
9  type Deal = (deck: Deck) => [Deck, Card]
10 type PickupCard = (hand: Hand, card: Card) => Hand
```

Zdrojový kód 9: DDD - Příklad karetní hry [31]

11 Multiplatformní JS technologie

Již dávno neplatí, že JavaScript je technologií, která je spojená výhradně s prohlížečem a webovými stránkami. JavaScript nachází uplatnění nejen na serveru, ale také na mobilních zařízeních a desktopech. V současné době se velké množství softwarových firem potýká s problémem jak stavět udržitelné multiplatformní aplikace, které by běžely současně na mobilních zařízeních i desktopech. Alternativním řešením jsou samozřejmě webové aplikace. Tématem předposledního semináře bude vývoj nativních aplikací v rámci platformy JavaScript - ať už s integrací jádra prohlížeče nebo přímo kompilováním do nativního kódu cílového zařízení.

11.1 Electron

Jedním z dlouhodoběji stabilních trendů je technologie zvaná Electron [32]. Ta je použita pro řadu moderních desktopových aplikací jako je Visual Studio Code, Microsoft Teams nebo Slack.

Jedná se o open-source framework, který je vyvíjen společností GitHub. Používá renderovací jádro prohlížeče Chromium a JavaScriptové běhové prostředí Node.js. Jedná se o způsob, jakým se webová aplikace zabalí do desktopové podoby, která navíc umožňuje pracovat se systémovými zdroji. Způsob takovéto tvorby aplikací je velmi zajímavý, jelikož jsou vývojáři odstíněni od celé řady problémů, spojených s vývojem pro více platforem. Navíc mohou využít znalosti z tvorby webových aplikací, takže je snadné pro tým vývojářů se zaměřením na webové aplikace přejít na vývoj pro desktop nebo mobil.

11.2 Nextron

Příklady ve skriptech využívají upravenou verzi Electronu - Nextron [33]. Jedná se o projekt, který kombinuje frameworky Next.js a Electron. Electron sám o sobě pracuje s API prohlížeče, tudíž pro tvorbu dynamického frontendu je nutné pracovat s DOM. Jelikož Next má v sobě integrovaný React spolu s celou řadou dalších užitečných nástrojů, je pak velmi snadné díky znalosti Next.js začít tvořit desktopové aplikace.

11.3 React Native

Za zmínku určitě stojí také technologie React Native [34]. Ta umožňuje vyvíjet nativní mobilní aplikace cílené na Android a iOS s použitím JavaScriptových technologií a Reactu. React Native je vyvíjen společností Facebook. Extrémním příkladem je pak implementace React Native od Microsoftu [35], která cílí na platformy Windows a MacOS. Sice se nezdá, že by Microsoft měl ambice tímto způsobem nahradit desktopový vývoj nad platformou .NET, pro určitý typ projektu by se však dalo zvážit použití právě této technologie.

12 Funkcionální programování na platformě JS

V průběhu kurzu budou přiblíženy výhody spojené s použitím funkcionálního přístupu při vývoji v JavaScriptu. Počínaje předvídatelnou prací s daty pomocí nedestruktivních operací jako jsou `map` nebo `reduce`, přes funkcionální kompozici React komponent, až po relativně sofistikovaný funkcionální state management. Stále však platí, že samotný JavaScript čistě funkcionální není. Funkcionální prvky jsou spíše součástí externích knihoven, jako je Immer. Tématem první části závěrečného semináře bude knihovna *fp-ts* [36], která integruje funkcionální principy do TypeScriptu. Druhá část bude věnovaná úplně odlišnému přístupu, a to zcela samostatnému jazyku *Clojure*, který je hostován nad platformou JavaScript.

12.1 Fp-ts

Knihovna fp-ts implementuje řadu funkcionálních principů z jazyků jako je *Haskel* nebo *F#*. Poskytuje pokročilé algebraické datové typy jako `Option` nebo `Either`, umožňující bezpečněji pracovat s daty, případně validovat předem neznámá data.

```
1  type Option<A> = { _tag: 'None' } | { _tag: 'Some'; value: A }
2
3  interface Left<E> {
4    readonly _tag: 'Left'
5    readonly left: E
6  }
7
8  interface Right<A> {
9    readonly _tag: 'Right'
10   readonly right: A
11 }
12
13 type Either<E, A> = Left<E> | Right<A>
```

Zdrojový kód 10: Příklad z GitHub repozitáře Fp-101 [37]

12.2 Option & Either

Pomocí algebraických datových typů a generik můžeme eliminovat problematické primitivní datové typy jako jsou `undefined` a `null` tak, že zavedeme jednotný typ `Option`. Ten udržuje meta-informace o tom, zda je proměnná inicializovaná a pokud ano, jaká je její hodnota. Knihovna fp-ts poskytuje další pomocné funkce pro práci s tímto typem.

Dalším ze základních algebraických datových typů z knihovny fp-ts je typ `Either`. Ten se používá pro ošetření návratových hodnot funkcí, které mohou skončit chybou. Definují se dva scénáře - `Right` a `Left`. Výsledek typu `Right`

značí skutečnost, že vše proběhlo v pořádku a vrací se požadovaný výsledek. Scénář `Left` pak oznamuje, že v průběhu výpočtu došlo k chybě, která se zároveň vrací. To umožňuje zřetězit řadu funkcí a tímto generickým typem označit výsledek výpočtu, od kterého očekáváme, že by v průběhu mohl skončit chybou. Díky abstrakcím z fp-ts není nutné explicitně ošetřovat kód výjimkami a větvením. Tento přístup se nazývá Railway Oriented Programming.[13]

12.3 ClojureScript

ClojureScript [38] je překladač z funkcionálního jazyka Clojure [39] do JavaScriptu. Clojure je dialektem Lispu a je primárně navržen jako hostovaný jazyk pro platformu Java. Je možné ho však použít i nad platformami .NET a Node.js.

Veškeré datové struktury jsou implicitně imutabilní. Pro práci s mutabilním stavem je vytvořen transakční systém. Velkou předností jazyka Clojure je datová orientace, která umožňuje pracovat s kódem jako s daty. Další typickou předností Lispových jazyků je pokročilá práce s makry. Clojure je sám o sobě velmi minimalistický, ale zároveň umožňuje snadnou rozšiřitelnost.

12.4 REPL Driven Development

Clojure poskytuje dynamické vývojové prostředí, které umožňuje interagovat se systémem pomocí REPLu.[14] Nad Node.js je možné použít například nástroj Lumo [40], který poskytuje rychlé, multiplatformní prostředí pro Clojure. Primárně je použití Clojure tímto způsobem vhodné pro tvorbu menších utilit a webových služeb. Pro rozsáhlejší aplikace je vhodnější použít ClojureScript překladač v kombinaci s platformou Java.

```
1 (require 'fs)
2
3 (fs/readFile
4  "file.txt"
5  (fn [err data] (println data)))
```

Zdrojový kód 11: Clojure interoperabilita s Node.js (Lumo)

[13] Více o Railway Oriented programování lze najít například v přednáškách Scotta Wlaschina [31].

[14] REPL (Read Eval Print Loop) je programovací prostředí, které umožňuje programátorovi komunikovat s běžícím programem a upravovat jej vyhodnocováním výrazů [39].

Závěr

V této práci jsme se seznámili se základními principy vývoje webových aplikací pomocí současných JavaScriptových technologií. Vzhledem k rychle se měnícím trendům v oblasti webových technologií je nutné brát v potaz, že podoba předmětu by se jim měla v průběhu následujících let přizpůsobovat. Důležitou součástí výuky je schopnost abstrahovat od konkrétních technologií a zaměřit se zejména na principy, které stojí v pozadí vývoje. Jednotlivé semináře by měly studentům poskytnout sadu základních nástrojů a programátorských technik pro vývoj JavaScriptových aplikací.

Conclusions

In this work, we got acquainted with the basic principles of web application development using current JavaScript technologies. Due to the rapidly changing trends in the field of web technologies, it is necessary to take into account that the form of the subject should adapt to them in the coming years. An important part of teaching is the ability to abstract from specific technologies and focus mainly on the principles that underlie the development. Individual seminars should provide students with a set of basic tools and programming techniques for developing JavaScript applications.

A Obsah přiloženého datového média

doc/
: Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

src/
: Kompletní materiály ke všem seminářům včetně zdrojových souborů.

U veškerých cizích převzatých materiálů obsažených na médiu jejich zahrnutí dovolují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy na médiu, je uveden jejich zdroj (např. webová adresa) v bibliografii nebo textu práce nebo v souboru `readme.txt`.

Literatura

[1] *JavaScript Platform Course*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/jkvapil6/kmi-pjs⟩.

[2] *Repl.it – Free, collaborative, in-browser IDE to code in 50+ languages*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://replit.com/⟩.

[3] *Visual Studio Code is a code editor redefined and optimized for building and debugging modern web and cloud applications*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://code.visualstudio.com/⟩.

[4] *NodeJS: JavaScript runtime*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/nodejs/node⟩.

[5] *Deno: A modern runtime for JavaScript and TypeScript*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/denoland/deno⟩.

[6] *The Promise object represents the eventual completion*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise⟩.

[7] *Axios, Promise based HTTP client for the browser and node.js*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/axios/axios⟩.

[8] *Express, Fast, unopinionated, minimalist web framework for node*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/expressjs/expres⟩.

[9] *Representational State Transfer*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://cs.wikipedia.org/wiki/Representational_State_Transfer⟩.

[10] *Angular: The modern web developer's platform*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/angular/angular⟩.

[11] *Vue: An approachable, performant and versatile framework for building web user interfaces*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/vuejs/core⟩.

[12] *React: A declarative, efficient, and flexible JavaScript library for building user interfaces*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/facebook/react⟩.

[13] *The Diffing Algorithm*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://reactjs.org/docs/reconciliation.html⟩.

[14] *Next-js: The React Framework*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/vercel/next.js⟩.

[15] *SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/sqlite/sqlite⟩.

[16] Ignjatovic, Alexandre. *N+1 problem* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://medium.com/doctolib/understanding-and-fixing-n-1-query-30623109fe89⟩.

[17] *MongoDB is a source-available cross-platform document-oriented database program.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/mongodb/mongo⟩.

[18] *Internet vecí.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://cs.wikipedia.org/wiki/Internet_v%C4%9Bc%C3%AD⟩.

[19] *InfluxDB – the Time Series Data Platform where developers build IoT, analytics, and cloud applications.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://www.influxdata.com/⟩.

[20] *The WebSocket API (WebSockets).* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API⟩.

[21] *RFC 6455, The WebSocket Protocol.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://datatracker.ietf.org/doc/html/rfc6455⟩.

[22] *ZeroMQ: An open-source universal messaging library.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/zeromq⟩.

[23] Martin Mareš, Tomáš Valla. *Průvodce labyrintem algoritmů.* Praha: CZ.NIC, z. s. p. o., 2017. ISBN 978-80-88168-19-5.

[24] *Immutable binnary tree.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://massivealgorithms.blogspot.com/2015/10/immutable-binary-tree.html⟩.

[25] *Flux – the application architecture that Facebook uses for building client-side web applications.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://facebook.github.io/flux/docs/in-depth-overview⟩.

[26] *Easy state management for react using hooks in less than 1kb.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/use-global-hook/use-global-hook⟩.

[27] *Create the next immutable state by mutating the current one.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/immerjs/immer⟩.

[28] *Monorepozitáž.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://cs.wikipedia.org/wiki/Monorepo⟩.

[29] *TypeScript is a superset of JavaScript that compiles to clean JavaScript output.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/microsoft/TypeScript⟩.

[30] *JSDoc, An API documentation generator for JavaScript.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/jsdoc/jsdoc⟩.

[31] Wlaschin, Scott. *F# for Fun and Profit* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://fsharpforfunandprofit.com/⟩.

[32] *Framework Electron.* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/electron⟩.

[33] *Nextron: Electron + Next.js*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/saltyshiomix/nextron⟩.

[34] *A framework for building native applications using React*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/facebook/react-native⟩.

[35] *A framework for building native Windows apps with React*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/microsoft/react-native-windows⟩.

[36] *Fpts, Functional programming in TypeScript*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/gcanti/fp-ts⟩.

[37] Boutté, Victor. *Fp-101, repo for functional programming 101 youtube video series* [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/monsieurBoutte/fp-101-series/tree/master/src⟩.

[38] *ClojureScript, Clojure to JS compiler*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/clojure/clojurescript⟩.

[39] *Clojure, The Clojure programming language*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/clojure/clojure⟩.

[40] *Fast, cross-platform, standalone ClojureScript environment*. [online]. [cit. 2022-4-24]. Dostupný z: ⟨https://github.com/anmonteiro/lumo⟩.