Webové inženýrství

3. ročník
Skripta

Autor
Ing. Matěj Cajthaml

Verze
14. 12. 2025

Obsah

Seznam zkratek

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

CLI Command Line Interface

CORS Cross-Origin Resource Sharing

CSS Cascading Style Sheets

DOM Document Object Model

ECMA European Computer Manufacturers Association

ES ECMAScript

FTP File Transfer Protocol

GUI Graphical User Interface

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

HTTP Hypertext Transfer Protocol

I/O Input/Output

ID Identifikátor

IDE Integrated Development Environment

ISO International Organization for Standardization

JS JavaScript

JSON JavaScript Object Notation

LESS Leaner Style Sheets

MPA Multi Page Application

OOP Objektově orientované programování

REST Representational State Transfer

SASS Syntactically Awesome Stylesheets

SASS Sassy CSS

SHA Secure Hash Algorithm

SLA Service Level Agreement

SMS Short Message Service

SPA Single Page Application

SQL Structured Query Language

SSH Secure Shell

SVN SubVersion

TS TypeScript

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

UTC Universal Time Coordinated

UX User Experience

WBA Webové aplikace

WIP

Vše je rozpracováno

Všechny kapitoly jsou rozpracovány. Nic zde není dokončeno a může se to kdykoliv změnit. Nejblíže se k dokončení mají kapitoly „Rozšíření znalostí“, „JavaScript“ a „TypeScript“.

Pokud naleznete chybu, zvažte, zda není možné, že je to tím, že je kapitola rozpracována.

Předmluva

2.1 Úvod

Webové aplikace nás obklopují na každém kroku. Pro mě osobně představují jednu z nejzajímavějších oblastí programování a informačních technologií obecně. Ale to již víte ze druhého ročníku, kdy jste se s nimi (a možná semnou) poprvé setkali.

Tento text je určen především studentům 3. ročníku Smíchovské střední průmyslové školy a gymnázia v Maturitním odborném předmětu Webové inženýrství. Cílem je předat velmi pokročilé znalosti z oblasti tvorby (nejen) webových aplikací a připravit vás na další studium či praxi v tomto oboru.

Jako webový inženýři víte, že každý rok se v rámci webu objevují nové technologie, trendy a přístupy. Letos se v tomto předmětu vrhneme na pokročilou tvorbu webových aplikací. A to včetně témat které souvisí s kompletní tvorbou služeb na webu. Jsem si jist, že vám ale všechny témata dají nové pochopení pro to, jak máte webové aplikace tvořit, aby jste mohli mít populární a úspěšné weby.

Stejně jako se webové technologie posunují dopředu, tak i tato skripta budou postupně rozšiřována a aktualizována, aby držela krok s vývojem oboru.

Pevně věřím, že pro vás budou přínosná a užitečná na cestě za hlubším porozuměním webovým aplikacím. Přeji vám mnoho radosti při jejich studiu.

2.2 Poděkování

Tato skripta vznikala v průběhu mnoha let výuky na SSPŠ. Část obsahu vychází z dřívějších materiálů, které jsem postupně tvořil a využíval při hodinách. Tyto texty prošly rukama stovek studentů, kteří svými připomínkami a upozorněním na chyby významně přispěli k jejich vylepšení. Všem jim patří velké poděkování za cennou zpětnou vazbu.

Velký dík si zaslouží také tým stojící za nástrojem Typst, bez něhož by tato podoba materiálu nemohla vzniknout.

Pokud během čtení narazíte na chybu, nepřesnost nebo budete mít návrh na zlepšení, budu rád, když vytvoříte issue na GitHubu.

2.3 Jak používat skripta

Skripta slouží jako podpůrný materiál k předmětu MVOP Webové inženýrství ve třetím ročníku a svým obsahem z velké části kopírují probíranou látku. Místy obsahují doplňující informace, které se během hodin nestihnou probrat, a naopak – některé části výuky se zde nemusí objevit. Je proto důležité brát je jako doplněk, nikoliv náhradu výuky.

Je třeba zdůraznit, že samotným čtením se programovat nenaučíte. Programování (a i tvorba webových stránek) je dovednost, která se osvojuje především praktickým cvičením. Proto doporučuji při čtení vždy zkoušet ukázky v editoru a experimentovat s kódem na vlastní pěst.

Pokročilejší příklady naleznete často také v prezentacích z hodin, které mohou sloužit jako rozšíření a inspirace.

2.4 Návaznost

Celý předmět a i tato skripta zcela navazují na předchozí znalosti, které jste měli získat v předmětech Webové aplikace ve druhém ročníku. Skoro ve všech částech se tyto základy předpokládají a často se referuje na již zmíněné koncepty. Podobně to je i s dalšími předměty, jako je Programování a vývoj aplikací a často i Počítačové sítě.

Pro začátek (a tak to děláme i v hodinách) doporučuji si připomenout obsah tohoto předmětu a případně si projít i skripta z nich. Vše naleznete na archívu mých materiálů.

Rozšíření znalostí

3.1 Omezení jednoho roku

Během jednoho roku se toho dá stihnout opravdu hodně. Hlavně tehdy, když se všichni učí něco, co je baví a zajímá.

V rámci WBA ve druhém ročníku jste získali základní přehled o tvorbě webových aplikací, resp. spíše jenom webových stránek. Spousta věcí, které byly probrány v hodinách, byly často zjednodušené, abychom lidi, kteří s weby nechtějí pokračovat, nezahltili zbytečnými detaily.

V této sekci se podíváme na nějaké věci, které pro moderní webové aplikace jsou důležité, ale z WBA je neumíme dostatečně dobře.

3.2 Styly

CSS obsahuje spoustu vlastností, které jsme nezmínili. Během roku si jich moc nových nebudeme ukazovat a spíše budeme předpokládat, že si je sami dohledáte podle potřeby.

Nicméně z CSS jsme velmi vynechali pár důležitých konceptů, které je dobré znát.

3.2.1 Specificita selektorů

Víme, že v CSS existuje věc, které se říká kaskáda. Jedna z vlastností kaskády je to, že pokud je nějaký prvek definován více pravidly, tak se použije to pravidlo, které je „specifičtější“. Ve WBA to pro nás znamenalo to, že poslední pravidlo v CSS souboru má přednost před předchozím. To stále platí, ale je tu ještě jeden důležitý koncept, kterému se říká specificita selektorů.

Specificita říká, že některé selektory jsou „silnější“ než jiné. Tedy vybírají prvek konkrétněji a proto by měli mít přednost před jinými pravidly.

Například selektor podle ID je silnější než selektor podle třídy. To si můžeme zdůvodnit jednoduše – ID je na stránce unikátní a má ho jeden prvek. Naopak například výběr pomocí tagu je velmi obecný a může se týkat mnoha prvků. Když dáme styly na všechny tagy, tak poté neočekáváme, že by tyto styly byly důležitější, než ty, které jsme dali na konkrétní třídu.

CSS specificitu počítá na základě několika pravidel. Používá proto bodový systém, kdy každý selektor má určitý význam. Celý selektor pak spočítá body dle selektorů v něm použitých a výsledné číslo určí jeho specificitu. Specificita se pak porovnává právě mezi sebou v kategoriích. Kategorie jsou tři[1] a jsou seřazeny podle důležitosti z leva doprava. Často je zapisujeme pomocí pomlček, například `0-1-0` znamená 0 bodů v první kategorii, 1 bod ve druhé a 0 bodů ve třetí.

Například `1-1-0` je silnější než `0-2-0`, protože první kategorie je důležitější než druhá.

Níže je seznam pravidel, které mění jakou kategorii a o kolik bodů:

- **1. úroveň** jsou selektory dle ID,
- **2. úroveň** jsou selektory dle třídy, atributu nebo pseudo-třídy,
- **3. úroveň** jsou selektory dle tagu nebo pseudo-elementu[2].

[1] Čistě teoreticky jsou čtyři, ale o významu té nejvyšší si povíme později.

[2] Co jsou pseudoelementy si povíme později.

Například selektor `#header .menu li` má specificitu `1-1-1`, protože obsahuje jeden selektor dle ID (`#header`), jeden selektor dle třídy (`.menu`) a jeden selektor dle tagu (`li`).

Důležité je, že do specificity nepočítáme univerzální selektor *, ani selektory pro potomky a další[3]. Tyto selektory poté mění jen jaké prvky se vyberou, ne jejich specificitu.

Největší specificitu (můžeme si představit jako 0. úroveň) mají dvě věci:
- Inline styly, tedy styly přímo v HTML elementu pomocí atributu `style`, například `<div style="color: red;">`.
- Atribut `!important` u vlastnosti v CSS, například `color: red !important;`.

Tyto věci mají nejvyšší prioritu a přepíší jakékoliv jiné pravidlo, i když má větší specificitu.

Používání !important

Používání `!important` se obecně nedoporučuje, protože ztěžuje údržbu kódu a může vést k nečekaným výsledkům. Obecně použití `!important` naznačuje, že něco není v pořádku s CSS kódem – buď používáte nevhodnout knihovnu (framework) nebo máte špatně navržené styly, protože nechápete jak funguje kaskáda a specificita.

3.2.2 Dědičnost vlastností

Některé vlastnosti v CSS jsou dědičné. To znamená, že pokud je nastavíme na rodičovském prvku, tak se automaticky použijí i na jeho potomky. Například vlastnost `color` (barva textu) je dědičná.

Pokud tedy nastavíme `color: red;` na nějakém rodičovském prvku, tak všechny texty uvnitř tohoto prvku budou červené, pokud je nepřepíšeme jiným pravidlem. Naopak vlastnost `background-color` (barva pozadí) není dědičná. To znamená, že pokud nastavíme `background-color: yellow;` na rodičovském prvku, tak jeho potomci nebudou mít žluté pozadí, pokud jim to výslovně nenastavíme.

Mezi dědičné vlastnosti patří například:
- `color`,
- `font-family`,
- `font-size`,
- `font-style`,
- `font-weight`,
- `line-height`,
- `text-align`,
- A další.

3.2.3 Pseudo-elementy

Jak víme, tak pseudotřídy jsou speciální selektory, které vybírají prvky na základě jejich stavu nebo pozice v dokumentu. Pseudotřídy zapisujeme pomocí dvojtečky, například `a:hover` vybírá odkazy, když nad nimi uživatel přejede myší.

Podobně v CSS existují i pseudoelementy, které umožňují stylovat určité části prvků, které nejsou přímo reprezentovány v HTML. Pseudoelementy zapisujeme pomocí dvojité dvojtečky,

[3] Těmto typům selektorů se říká kombinační selektory.

například `::before`. Používáme je tehdy, když chceme přidat obsah nebo stylovat části prvku, které nejsou přímo v HTML.

Selektor: Obsah před obsahem

Syntax: `E::before`

Vybere obsah uvnitř elementu `E`, ale před jeho skutečným obsahem. Tedy vloží něco před to, co je v HTML. Tento pseudoelement se často používá k přidání dekorativního obsahu, jako jsou ikony nebo speciální znaky, bez nutnosti měnit HTML.

Pokud chceme přidat jakýkoliv obsah, je nutné použít vlastnost `content`.

Podobně funguje `E::after` pro výběr obsahu za obsahem elementu.

css
```
1 div::before {
2   content: ">  ";
3   color: red;
4 }
```

content: Obsah

Určuje obsah, který se má zobrazit v daném tagu. **Definice hodnoty**:

Můžeme vkládat:

- Text, například `'Hello'`, ve kterém můžeme používat i speciální znaky jako `\A` pro nový řádek.
- URL obrázku, například `url(image.png)`,
- Speciální znaky, například `attr(data-info)` pro vložení hodnoty
- Počítadla, například `counter(section)` pro číslování sekcí,
- Prázdný obsah pomocí `""` (dvojité uvozovky),
- A další.

Vložený obsah v prohlížeči není interaktivní a nelze ho například zkopírovat.

css
```
1 div::before {
2   content: "Hello \A World";
3   white-space: pre; /* Aby se respektovaly nové řádky */
4 }
```

Selektor: První řádek

Syntax: `E::first-line`

Vybere první řádek textu uvnitř elementu `E`. Tento pseudoelement se často používá k přidání speciálního stylu pro první řádek textu, například pro zvýraznění nebo změnu písma.

css

```css
p::first-line {
  font-weight: bold;
  color: blue;
}
```

Selektor: První písmeno

Syntax: `E::first-letter`

Vybere první písmeno textu uvnitř elementu `E`. Tento pseudoelement se často používá k vytvoření dekorativního efektu, například pro zvýraznění prvního písmene odstavce.

```css
p::first-letter {
  font-size: 2em;
  color: red;
}
```

Selektor: Výběr textu

Syntax: `::selection`

Vybere část textu, kterou uživatel označil (vybral) myší nebo klávesnicí. Tento pseudoelement se používá k přizpůsobení vzhledu označeného textu.

Je omezené, jaké vlastnosti můžeme u tohoto pseudoelementu použít. Mezi podporované vlastnosti patří například `color`, `background-color`, `text-shadow` a `cursor`.

```css
::selection {
  background: yellow;
  color: black;
}
```

Selektor: Zástupný text

Syntax: `::placeholder`

Vybere text, který se zobrazuje jako nápověda v prvcích formuláře, jako jsou `<input>` nebo `<textarea>`, když jsou prázdné. Tento pseudoelement se používá k přizpůsobení vzhledu tohoto nápovědného textu.

```css
input::placeholder {
  color: gray;
  font-style: italic;
}
```

Selektor: Marker

Syntax: `::marker`

Vybere značky (bullet points) v seznamu, jako jsou `<ul>` nebo `<ol>`. Tento pseudoelement se používá k přizpůsobení vzhledu těchto značek.

```css
li::marker {
  color: red;
  font-size: 1.5em;
}
```

3.2.4 Selektory

V CSS existuje spousta různých selektorů, které nám umožňují vybírat prvky na základě různých kritérií. Některé z nich jsme již viděli ve WBA, ale existuje jich mnohem více.

Selektor: Vedlejší potomci

Syntax: `E + F`

Vybere všechny prvky `F`, které jsou přímými sousedy prvku `E`. Tedy, prvek `F` musí být hned za prvkem `E` na stejné úrovni v hierarchii dokumentu.

```css
h1 + p {
  margin-top: 0;
}
```

Selektor: Vedlejší přímí potomci

Syntax: `E ~ F`

Vybere všechny prvky `F`, které jsou sourozenci prvku `E`. Tedy, prvek `F` musí být na stejné úrovni v hierarchii dokumentu jako prvek `E`, ale nemusí být hned za ním, pouze kdekoliv po něm.

```css
h1 ~ p {
  color: gray;
}
```

Selektor: Atribut

Syntax: `E[atribut]` nebo `E[atribut*"hodnota"]`, kde * může být =, ~=, ^=, $= nebo *=.

Vybere všechny prvky `E`, které mají daný atribut, nebo atribut s určitou hodnotou.

- `E[atribut]` vybere všechny prvky `E`, které mají atribut `atribut`, bez ohledu na jeho hodnotu,
- `E[atribut="hodnota"]` vybere všechny prvky `E`, kde atribut `atribut` má přesně hodnotu `hodnota`,
- `E[atribut~="hodnota"]` vybere všechny prvky `E`, kde atribut `atribut` obsahuje slovo `hodnota` (slova jsou oddělena mezerami),
- `E[atribut^="hodnota"]` vybere všechny prvky `E`, kde atribut `atribut` začíná na `hodnota`,
- `E[atribut$="hodnota"]` vybere všechny prvky `E`, kde atribut `atribut` končí na `hodnota`,
- `E[atribut*="hodnota"]` vybere všechny prvky `E`, kde atribut `atribut` obsahuje `hodnota` kdekoli v hodnotě.

css

```css
a[href^="https"] {
  color: green;
}
```

Selektor: Negace

Syntax: `E:not(F)`

Vybere všechny prvky `E`, které neodpovídají selektoru `F`.

css

```css
div:not(.highlight) {
  background: lightgray;
}
```

Selektor: Obsahuje potomka

Syntax: `E:has(F)`

Vybere všechny prvky `E`, které obsahují alespoň jeden prvek odpovídající selektoru `F` jako svého potomka.

css

```css
div:has(p) {
  border: 1px solid black;
}
```

JavaScript

4.1 Co je JavaScript

JavaScript v tom nejzákladnějším smyslu je skriptovací jazyk. Byl vytvořen v roce 1995 Brendanem Eichem během jeho krátkého působení ve společnosti Netscape. Původně byl navržen jako jednoduchý jazyk pro přidání interaktivity na webové stránky.

JavaScript se v moderním smyslu stal velmi populárním a všestranným programovacím jazykem. Jeho schopnosti sahají daleko za hranice webových prohlížečů a zahrnují vývoj serverových aplikací, mobilních aplikací, desktopových aplikací a dokonce i vývoj her.

4.1.1 Typy jazyků

Mimo JavaScript je dobré, aby weboví inženýři rozuměli základním konceptům programovacích jazyků.

Programovací jazyky můžeme měnit do několika kategorií podle různých kritérií. Urovně tohoto rozdělení bývají různé a často se překrývají. Neznamená tedy, že máme pouze odpověď „ano“ či „ne“ na otázku, zda patří jazyk do určité kategorie.

Nízkoúrovňové jazyky (low-level languages) jsou blíže strojovému kódu a poskytují větší kontrolu nad hardwarem. Často musíme spravovat paměť a další systémové zdroje ručně. Příkladem nízkoúrovňového[4] jazyka je například Assembler nebo C.

Vysokoúrovňové jazyky (high-level languages) jsou blíže lidskému jazyku a abstraktnější. Poskytují vyšší úroveň abstrakce a často spravují paměť a další systémové zdroje automaticky. Příkladem vysokoúrovňového jazyka je Python, Ruby nebo JavaScript.

Dalším rozdělením, co máme, je podle účelu jazyka. Například máme skriptovací jazyky (scripting languages), které jsou určeny pro automatizaci úkolů a psaní skriptů. Upravují totiž existující aplikace nebo systémy. Například již zmíněný JavaScript, Bash i další. JS je primárně skriptovací jazyk, protože upravuje webové stránky a aplikace. Na druhou stranu máme programovací jazyky (programming languages), které jsou určeny pro vývoj samostatných aplikací a systémů.

Rozdělení dle spuštění je také běžné. Máme interpretované jazyky (interpreted languages), které jsou spuštěny přímo interpretem bez předchozí kompilace do strojového kódu. Potřebujeme tedy další softwarovou vrstvu, která kód překládá a spouští. JavaScript je interpretovaný jazyk, protože je spuštěn přímo v prohlížeči nebo v běhovém prostředí. Na druhou stranu máme kompilované jazyky (compiled languages), které jsou před spuštěním přeloženy do strojového kódu pomocí kompilátoru.

Některé jazyky nelze rozdělit do interpretovaných či kompilovaných, protože mohou být jak interpretované, tak kompilované. Ukázkou je například Java, která je kompilována do bytecode a poté interpretována virtuálním strojem. Podobně to má moderní C# s .NET.

Další rozdělení je podle typování. Máme jazyky s dynamickým typováním (dynamically typed languages), kde jsou datové typy určeny za běhu programu. JavaScript je dynamicky typovaný jazyk, protože proměnné mohou měnit svůj datový typ během běhu programu. Na druhou stranu máme jazyky s statickým typováním (statically typed languages), kde jsou datové typy určeny při kompilaci. Proměnné musíme deklarovat s konkrétním datovým typem a ten se nemění. Příkladem staticky typovaného jazyka[5] je například Java nebo C.

[4] A tady již vidíme tu nejistotu toho, kam každý jazyk patří. C je podle některých high-level jazyk.

[5] V některých jazycích nemusíme proměnné deklarovat s typem, ale typ je zjištěn hodnotou při deklaraci.

Další rozdělení je podle programovacího paradigmatu.

Máme procedurální jazyky (procedural languages), které jsou založeny na procedurách nebo funkcích. Příkladem procedurálního jazyka je C.

Máme objektově orientované jazyky (object-oriented languages), které jsou založeny na objektech a třídách. Příkladem objektově orientovaného jazyka je Java nebo C++.

Máme funkcionální jazyky (functional languages), které jsou založeny na funkcích jako prvotních stavebních kamenech. Příkladem funkcionálního jazyka je Haskell nebo Lisp.

Máme datově orientované jazyky (data-oriented languages), které jsou založeny na práci s daty a jejich transformacemi. Příkladem datově orientovaného jazyka je SQL.

Posledním rozdělením, které si ukážeme, je dle způsobu práce. Imperativní jazyky (imperative languages) jsou založeny na příkazech, které mění stav programu. Příkladem imperativního jazyka je C nebo JavaScript. Deklarativní jazyky (declarative languages) jsou založeny na popisu toho, co má být provedeno, spíše než jak to má být provedeno. Příkladem deklarativního jazyka je SQL nebo HTML.

JavaScript je multiparadigmatický jazyk, což znamená, že podporuje více programovacích paradigmat. Podporuje procedurální, objektově orientované i funkcionální programování. JavaScript je také imperativní, dynamicky typovaný, interpretovaný a vysokoúrovňový jazyk.

4.1.2 ECMAScript

JavaScript je standardizován organizací ECMA International pod názvem ECMAScript. Čistě teoreticky nikdo již čistý JavaScript nepíše, ale prohlížeče podporují různé verze ECMAScriptu.

ECMAScript postupem času přidával nové funkce a vylepšení. A to jeden z důvodů, proč se JavaScript stal tak populárním a všestranným jazykem. Mezi nejvýznamnější verze ECMAScriptu patří:

- ECMAScript 3 (ES3) – vydán v roce 1999, přinesl základní funkce jako regulární výrazy, výjimky a další,
- ECMAScript 5 (ES5) – vydán v roce 2009, přidal funkce jako strict mode, JSON podporu a další,
- ECMAScript 6 (ES6) – vydán v roce 2015, přinesl významné změny jako třídy, moduly, šipkové funkce, let a const, šablonové řetězce a další,
- ECMAScript 2016 a novější – každoroční aktualizace přinášejí menší vylepšení a nové funkce.

4.1.3 Historie JavaScriptu

Jak bylo již řečeno, JS vzniknul dávno a hlavně rychle. Jeho rychlý vývoj vedl k určitým nedostatkům a nejasnostem v jazyce. To vedlo k různým implementacím a nekompatibilitám mezi prohlížeči. A celkově vzniklo spoustu hrůzných kódů, které se dnes snažíme nepsat.

Naštěstí se situace zlepšila s příchodem moderních verzí ECMAScriptu a lepší podpory prohlížečů.

Programování uvnitř JavaScriptu je jiné nehledě na verzi ECMAScriptu. Dynamické typování pro některé znamená svobodu, pro jiné chaos. Nejhorší důsledek dynamického typování je to, že chyby se často projeví až za běhu programu. Proto je nutné v každé funkci ověřovat

parametry a jejich typy. Typy se ověřují taktéž nevhodně, a to například díky nevhodně navrženému typu porovnání (viz @operators).

Další špatnou částí[6] JS je, že musí tvořit zpětnou kompatibilitu. To znamená, že nové funkce a vylepšení musí být přidávány tak, aby neporušily stávající kód. To vede k tomu, že některé starší funkce a konstrukce zůstávají v jazyce i přesto, že jsou nevhodné nebo zastaralé. Příkladem je například použití `var` pro deklaraci proměnných, které má nevhodný rozsah (scope) a chování.

Odstraněním těchto nevhodných částí by se jazyk stal jednodušším a přehlednějším ale to by vedlo, že staré webové stránky by přestaly fungovat. A to je pro prohlížeče nepřijatelné. Naštěstí moderní JavaScript nabízí mnoho nových funkcí a konstrukcí, které umožňují psát čistý, dobře strukturovaný a udržovatelný kód.

> **Změna přístupu**
>
> Všem studentům bych doporučil se vzdát takové teze, že JavaScript je špatný jazyk. Problém jazyků většinou bývá na tom, jak ho používají programátoři.
>
> JavaScript je velmi mocný a flexibilní jazyk, který umožňuje vytvářet složité aplikace. Pokud si záměrně budete dávat „klacky pod nohy“, tak vás sebelepší jazyk nezachrání.
>
> V JS vám tedy doporučuji se naučit (jednou a provždy) správný přístup k psaní kódu. To zahrnuje například nepoužívání zvláštních konstrukcí jazyka, které vedou k nečitelnému kódu. Dále se naučit používat moderní funkce jazyka a psát čistý, dobře strukturovaný kód. A hlavně se naučit používat nástroje, které vám pomohou udržet kód kvalitní[7].

4.1.4 Použití JavaScriptu na webu

Na webových stránkách se JavaScript používá k přidání interaktivity a dynamického obsahu. To z obyčejných webových stránek poté tvoří webové aplikace.

JavaScript má v rámci prohlížeče přístup k různými API, které dovolují modifikovat obsah stránky, reagovat na uživatelské akce, komunikovat se servery a další. S různými API jste se již setkali a v průběhu následujících dvou let se s nimi setkáte ještě více.

Nejdůležitější částí skritpovacího jazyka je Document Object Model (DOM), který představuje strukturu HTML dokumentu jako strom uzlů. Uvnitř DOM můžeme pomocí JavaScriptu měnit obsah stránky, přidávat nebo odebírat prvky, měnit jejich styly a reagovat na události jako kliknutí myší, stisky kláves a další. Změny uvnitř DOM se okamžitě projeví na stránce.

Nejdůležitějším konceptem je vybírání prvků uvnitř DOM. Pro výběr prvků můžeme použít různé metody, například `getElementById`, `getElementsByClassName`, `getElementsByTagName` nebo modernější `querySelector` a `querySelectorAll`. Modernější alternativy vřele doporučuji, protože dovolují vybírat prvky za pomocí CSS selektorů.

Po vybrání prvku můžeme měnit jedno vlastnosti.

`innerHTML` dovoluje měnit vnitřní HTML prvku,
`textContent` dovoluje měnit vnitřní text prvku,
`style` dovoluje měnit styly prvku,

[6] Respektive spíše prohlížečů.
[7] Třeba TypeScript, viz Kapitola 5.1.

`classList` dovoluje přidávat, odebírat nebo přepínat třídy prvku,
`setAttribute` a `getAttribute` dovolují měnit nebo získávat atributy prvku.

Dále můžeme přidávat posluchače událostí pomocí metody `addEventListener`[8]. Mezi nejznámější události patří `click`, `input`, `change`, `submit`, `mouseover`, `mouseout` a další.

Uvnitř DOM a událostí je taktéž důležité zmínit to, jak se jednotlivé události volají. My totiž víme, že na jeden prvek můžeme přidat více posluchačů pro stejnou událost. Podobně když klikneme na rodiče prvku a ten rodič i potomek mají přidán posluchač pro `click`, tak se oba posluchače zavolají. To, v jakém pořadí se zavolají, závisí na tom, v jaké fázi se událost nachází. Uvnitř DOM máme dvě fáze – capture a bubble.

Fáze capture začíná od kořene dokumentu a postupuje směrem dolů k cílovému prvku. Tímto hledáme hlavní prvek události, pro který nastala a případně řešíme poslouchače. Po nalezení cílového prvku se událost dostane do fáze bubble.

Ve fázi bubble se událost šíří zpět nahoru od cílového prvku k rodičům. To znamená, že nejdříve se zavolají posluchače na cílovém prvku a poté na jeho rodičích až k celému dokumentu. V rámci buble fáze můžeme ukončit šíření události pomocí metody `stopPropagation`.

Vše, co jsme s DOM prozatím dělali se dělo právě v bubble fázi. My se však můžeme zachytit na událost ještě v době, kdy prochází capture fází. To dovoluje přidat třetí parametr do `addEventListener`, který může být objekt s různými možnostmi. Pro nás je zajímavá možnost `capture`, která pokud je nastavena na `true`, tak se posluchač přidá do capture fáze.

javascript

Přidání posluchače pro fázi capture

```
1 element.addEventListener('click', handler, { capture: true });
```

Dále můžeme zabránit výchozímu chování události pomocí metody `preventDefault`. Tím řekneme prohlížeči, aby nevolal původní akci s daným prvkem spojenou. Například u odkazu zabráníme přechodu na jinou stránku, u formuláře zabráníme odeslání formuláře a podobně.

DOM si implementuje vlastní pomocné struktury, například `NodeList`, což je kolekce uzlů, která se chová podobně jako pole, ale není to pole. Tyto struktury můžeme převádět na pole pomocí `Array.from` nebo rozšiřujícího operátoru (`...`).

Na webu píšeme JS uvnitř HTML pomocí tagu `<script>`.

html

Základní použití JavaScriptu uvnitř HTML

```
1 <script>
2   console.log("Ahoj světe!");
3 </script>
```

Další varianta je psaní JavaScriptu v externím souboru a jeho připojení pomocí atributu `src`.

html

Připojení externího JavaScriptového souboru

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

[8] Proboha prosím nepoužívejte inline event handlery pomocí `on*`, kde `*` je název události. To stejné prosím ani v kódu

4.1.5 JavaScript mimo web

JavaScript se místo skriptování webových stránek používá i mimo web. Mimo webové stránky říkáme JavaScriptu již programovací jazyk. Pro spuštění potřebujeme intepreter (běhové prostředí, runtime), které je schopno JavaScript spustit. Nejznámější je Node.js, které je založeno na V8 enginu z Google Chrome. Node.js umožňuje spouštět JavaScript na serveru, vytvářet serverové aplikace, skripty pro automatizaci úkolů, nástroje příkazové řádky a další. Alternativou k Node.js je například Deno nebo Bun.

Během roku se naučíme vše možné o Node.js, ale prozatím si ukážeme jen základní použití. Pro spuštění JavaScriptu v Node.js potřebujeme mít nainstalovaný Node.js. To můžeme udělat z oficiálních stránek https://nodejs.org.

Po instalaci máme k dispozici příkaz `node`, který nám dovolí nejen spustit JavaScriptové soubory. Soubor, který chceme spustit uvedeme jako první parametr.

bash

Spuštění JavaScriptového souboru

```
1 node soubor.js
```

Uvnitř takto spuštěného souboru nemáme přístup k API prohlížeče, ale máme přístup k API Node.js. Podobné API je například konzole `console`, která nám dovolí vypisovat informace na standardní výstup. Nové API je například `process`, které nám dovolí pracovat s procesem, ve kterém je Node.js spuštěn.

Proces nám přidává například následující:
`process.argv` pole argumentů příkazové řádky,
`process.env` objekt s proměnnými prostředí,
`process.exit()` ukončí proces s daným návratovým kódem,
`process.stdin`, `process.stdout`, `process.stderr` proudy pro standardní vstup, výstup a chybový výstup.

Právě první z nich se hojně používá pro předávání argumentů skriptům.

bash

Předání argumentů skriptu

```
1 node script.js arg1 "arg 2" arg3
```

Výše uvedené spuštění předá skriptu tři argumenty, ale pozor, že předává všechny části – včetně cesty k interpretu a k souboru. Uvnitř `script.js` je můžeme získat pomocí `process.argv`. Na jednotlivých indexech postupně budou:
`process.argv[0]` cesta k interpretu (například `/usr/bin/node`),
`process.argv[1]` cesta k souboru (například `/cesta/k/souboru/script.js`),
`process.argv[2]` první argument (v našem případě `arg1`),
`process.argv[3]` druhý argument (v našem případě `arg 2`) – jak vidíme, můžeme použít mezery, pokud je argument uzavřen v uvozovkách,
`process.argv[4]` třetí argument (v našem případě `arg3`).

K Node.js se váže i spousta dalších knihoven a nástrojů, které nám dovolí dělat téměř cokoliv. Nejznámější je pravděpodobně npm, což je správce balíčků pro Node.js. S tím se setkáme později, viz Kapitola 4.5.

4.2 Použití JavaScriptu

Nyní si připomeneme, respektive více podrobně vysvětlíme základní koncepty JavaScriptu. Spoustu základu již znáte z jiných jazyků, respektive předmětu Programování a vývoj aplikací.

4.2.1 Datové typy

JavaScript má několik základních datových typů:
`number` reprezentuje čísla, jak celá, tak desetinná,
`string` reprezentuje textové řetězce,
`boolean` reprezentuje logické hodnoty `true` a `false`,
`undefined` reprezentuje nedefinovanou hodnotu, často používáme pro indikaci, že opravdu něco není nastaveno,
`BigInt` reprezentuje celá čísla větší než `Number.MAX_SAFE_INTEGER`, obecně používáme málokdy,
`symbol` reprezentuje jedinečné identifikátory, které se používají hlavně pro klíče v objektech,
`function` reprezentuje funkce,
`object` reprezentuje složitější datové struktury, jako jsou pole, mapy, množiny a další,
`null` reprezentuje prázdnou hodnotu.

Některé výše uvedené typy nejsou nutně zcela primitivní, ale pro jednoduchost je takto uvádíme. Ukázkou je třeba hodnota `null`, která je interně reprezentována jako objekt. Podobně je na tom i `function`, která je speciálním typem objektu.

JavaScript je dynamicky typovaný jazyk, což znamená, že proměnné nemusíme deklarovat s konkrétním datovým typem. Proměnné mohou měnit svůj datový typ během běhu programu.

4.2.2 Proměnné

Pro deklaraci proměnných můžeme použít tři různá klíčová slova.

Nejstarší a nejméně doporučovanou variantou je `var`. `var` využívá tzv. hoistování. Hoistování znamená, že deklarace proměnné (ale i jiných konstrukcí) je přesunuta na začátek svého rozsahu (scope). To může vést k nečekanému chování, protože proměnná je dostupná ještě před svou deklarací. V rozsahu `var` ale není blok, ale funkce a navíce při prvotní přístupnosti se její hodnota nastaví na `undefined`. Hodnotu tato proměnná obrží až v místě přiřazení.

javascript

Hoistování proměnné deklarované pomocí `var`

```
1 console.log(a); // undefined, nehodí chybu
2 var a = 5;
3 console.log(a); // 5
```

Druhou variantou je `let`, která byla zavedena v ECMAScript 6 (ES6). `let` nemá hoistování a má blokový rozsah (scope). To znamená, že proměnná deklarovaná pomocí `let` je dostupná pouze v bloku, ve kterém je deklarována.

javascript

Chyba při přístupu k proměnné deklarované pomocí `let` před její deklarací

```
1 console.log(b); // ReferenceError: b is not defined
2 let b = 10;
```

```
3 console.log(b); // 10
```

Třetí variantou je `const`, která je podobná `let`, ale deklarovaná proměnná musí být inicializována při deklaraci a její hodnota nemůže být změněna. Tady je důležité si uvědomit, že v jazycích existují dva nejzákladnější typy dat – primitivní a referenční. Primitivní (datové) typy jsou například `number`, `string`, `boolean` a další. Referenční (datové) typy jsou například `object`, `array`, `function` a další. Referenční typy místo předávání hodnoty předávají referenci (adresu) na hodnotu v paměti. To znamená, že pokud deklarujeme proměnnou pomocí `const` a přiřadíme jí referenční typ, můžeme měnit obsah tohoto typu, ale nemůžeme změnit samotnou referenci.

javascript

Chování `const` s referenčními typy

```
1 const c = { name: "Alice" };
2 c.name = "Bob"; // Toto je povoleno, měníme obsah objektu
3 console.log(c.name); // Bob
4 c = { name: "Charlie" }; // Toto není povoleno, měníme referenci
5 console.log(c.name); // TypeError: Assignment to constant variable.
```

4.2.3 Operátory

Operátor je speciální symbol nebo klíčové slovo, které provádí operaci na jednom nebo více operandech. Operand je hodnota, na které operátor pracuje.

Operátory řadíme do určitých kategorií a zde si je stručně popíšeme.

4.2.3.1 Aritmetické operátory

Aritmetické operátory provádí matematické operace na číselných hodnotách.

Operátor	Popis	Příklad
+	Sčítání	`5 + 3`
-	Odčítání	`5 - 3`
*	Násobení	`5 * 3`
/	Dělení	`5 / 3`
%	Zbytek po dělení (modulo)	`5 % 3`
**	Umocňování	`5 ** 3`

4.2.3.2 Logické operátory

Logické operátory pracují s logickými hodnotami (`true` a `false`) a vrací logickou hodnotu.

Operátor	Popis	Příklad
&&	Logický AND (a zároveň)	`true && false`
||	Logický OR (nebo)	`true || false`
!	Logický NOT (negace)	`!true`

4.2.3.3 Bitové operátory

Bitové operátory pracují na úrovni jednotlivých bitů čísel.

Operátor	Popis	Příklad
&	Bitový AND	`5 & 3`
|	Bitový OR	`5 | 3`
^	Bitový XOR	`5 ^ 3`
~	Bitový NOT	`~5`
<<	Posun vlevo	`5 << 1`
>>	Posun vpravo se znaménkem	`5 >> 1`
>>>	Posun vpravo bez znaménka	`5 >>> 1`

4.2.3.4 Přiřazovací operátory

Přiřazovací operátory přiřazují hodnoty proměnným.

Operátor	Popis	Příklad
=	Přiřazení	`a = 5`
+=	Přiřazení se sčítáním	`a += 3`
-=	Přiřazení s odčítáním	`a -= 3`
*=	Přiřazení s násobením	`a *= 3`
/=	Přiřazení s dělením	`a /= 3`
%=	Přiřazení se zbytkem po dělení	`a %= 3`

Operátor	Popis	Příklad
`**=`	Přiřazení s umocňováním	`a **= 3`
`<<=`	Přiřazení s posunem vlevo	`a <<= 1`
`>>=`	Přiřazení s posunem vpravo se znaménkem	`a >>= 1`
`>>>=`	Přiřazení s posunem vpravo bez znaménka	`a >>>= 1`
`&=`	Přiřazení s bitovým AND	`a &= 3`
`|=`	Přiřazení s bitovým OR	`a |= 3`
`^=`	Přiřazení s bitovým XOR	`a ^= 3`
`i++`	Inkrementace (zvýšení o 1)	`a++`
`i--`	Dekrementace (snížení o 1)	`a--`
`++i`	Předinkrementace (zvýšení o 1 před použitím)	`++a`
`--i`	Předdekrementace (snížení o 1 před použitím)	`--a`

4.2.3.5 Porovnávací operátory

Porovnávací operátory porovnávají dvě hodnoty a vrací logickou hodnotu.

Operátor	Popis	Příklad
`==`	Rovnost (volné porovnání)	`5 == '5'`
`===`	Rovnost (přísné porovnání)	`5 === '5'`
`!=`	Nerovnost (volné porovnání)	`5 != '3'`
`!==`	Nerovnost (přísné porovnání)	`5 !== '5'`
`>`	Větší než	`5 > 3`
`<`	Menší než	`5 < 3`
`>=`	Větší nebo rovno	`5 >= 3`

Operátor	Popis	Příklad
`<=`	Menší nebo rovno	`5 <= 3`

V JavaScriptu máme dva typy porovnávacích operátorů pro rovnost a nerovnost – volné (== a !=) a přísné (=== a !==). Volné porovnání provádí konverzi typů před porovnáním, což může vést k nečekaným výsledkům. Přísné porovnání neprovádí konverzi typů a porovnává hodnoty i jejich typy. Doporučuji vždy používat přísné porovnání, aby se předešlo nejasnostem a chybám v kódu.

4.2.3.6 Další operátory

Níže jsou uvedeny další operátory, které se často používají v JavaScriptu, ale nemají jasně zařazenou kategorii.

Operátor	Popis	Příklad
`? :`	Ternární operátor (podmíněný)	`true ? 'ano' : 'ne'`
`typeof`	Operátor typu	`typeof 5`
`instanceof`	Operátor instance	`obj instanceof MyClass`
`super`	Operátor super (přístup k metodám rodičovské třídy)	`super.method()`
`new`	Operátor new (vytvoření instance)	`new MyClass()`
`this`	Operátor this (odkaz na aktuální instanci)	`this.property`
`in`	Operátor vlastnosti v objektu	`'name' in obj`
`...`	Rozšiřující operátor (spread/rest)	`[...array1, ...array2]`
`void`	Operátor void (ignoruje hodnotu)	`void 0`
`delete`	Operátor delete (odstraní vlastnost objektu)	`delete obj.name`
`??`	Operátor nulového sloučení (nullish coalescing)	`value ?? 'default'`
`?.`	Volitelný řetězec (optional chaining)	`obj?.property`
`+`	Sčítání (konkatenace pro řetězce)	`'Hello, ' + 'world!'`

Operátor	Popis	Příklad
`...`	Rozšiřující operátor (spread/rest)	`[...array1, ...array2]`

A mnoho dalších, které se používají méně často nebo jsou specifické pro určité situace.

4.2.4 Řídící struktury

Řídící struktury nám dovolují řídit tok programu na základě podmínek nebo opakování.

4.2.4.1 Podmínky

Pro podmínky můžeme použít `if`, `else if` a `else`. Povolená kombinace je vždy `if` následované libovolným počtem `else if` a nakonec volitelným `else`. Dovnitř jako podmínku můžeme vložit libovolný výraz, který vrací logickou hodnotu. Pokud se vloží hodnota, která není logická, JavaScript ji převede na logickou hodnotu podle svých pravidel (pravdivé a nepravdivé hodnoty). Po prvním zachycení pravdivé podmínky se vykoná příslušný blok kódu a zbytek se přeskočí.

javascript

Základní použití podmínky `if`

```javascript
if (condition1) {
  // Kód se vykoná, pokud je condition1 pravdivá
} else if (condition2) {
  // Kód se vykoná, pokud je condition1 nepravdivá a condition2 je pravdivá
} else {
  // Kód se vykoná, pokud jsou všechny předchozí podmínky nepravdivé
}
```

4.2.4.2 Přepínač

Podmínky přepínače (switch) jsou užitečné, pokud máme více možných hodnot pro jednu proměnnou.

javascript

Základní použití přepínače `switch`

```javascript
switch (expression) {
  case value1:
    // Kód se vykoná, pokud expression === value1
    break;
  case value2:
    // Kód se vykoná, pokud expression === value2
    break;
}
```

V přepínači můžeme použít libovolný výraz jako `expression`. Hodnoty v jednotlivých případech (`case`) jsou porovnávány s výrazem pomocí přísného porovnání (`===`). Pokud chceme, aby se vykonal kód pro více hodnot, můžeme je jednoduše řadit za sebe bez `break`. Pro

jednotlivé případy můžeme obalit kód do bloku {}, pokud potřebujeme deklarovat proměnné s blokovým rozsahem.

javascript

Více hodnot pro jeden případ v přepínači switch

```
switch (expression) {
  case value1:
  case value2: {
    // Kód se vykoná, pokud expression === value1 nebo expression === value2
    let temp = "nějaká hodnota";
    console.log(temp);
    break;
  }
}
```

Blok default se vykoná, pokud žádný z předchozích případů neodpovídá výrazu. Obecně ho píšeme na konec přepínače.

4.2.4.3 Cykly

Cykly nám dovolují opakovat blok kódu, dokud je splněna určitá podmínka. V JS známe několik typů cyklů.

Cyklus	Popis
`for`	Opakuje blok kódu pro každý prvek v kolekci nebo pro daný počet opakování
`while`	Opakuje blok kódu, dokud je podmínka pravdivá
`do...while`	Opakuje blok kódu alespoň jednou a poté dokud je podmínka pravdivá
`for...in`	Iteruje přes vlastnosti objektu
`for...of`	Iteruje přes hodnoty iterovatelných objektů (např. pole, řetězce)

javascript

Základní použití cyklu for

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

javascript

Základní použití cyklu while

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

javascript

Základní použití cyklu `do...while`

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

javascript

Základní použití cyklu `for...in`

```
let obj = { a: 1, b: 2, c: 3 };
for (let key in obj) {
  console.log(key, obj[key]);
}
```

javascript

Základní použití cyklu `for...of`

```
let array = [1, 2, 3];
for (let value of array) {
  console.log(value);
}
```

4.2.4.4 Práce s cykly

Průchod cyklem můžeme ovlivnit pomocí dvou klíčových slov – `break` a `continue`. `break` okamžitě ukončí cyklus a pokračuje vykonávání kódu za cyklem. `continue` přeskočí zbytek aktuální iterace a pokračuje další iterací cyklu.

javascript

Použití `break` v cyklu

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break;
  }
  console.log(i);
}
```

4.2.5 Funkce

Funkce jsou bloky kódu, které můžeme volat opakovaně. Funkce nám dovolují organizovat kód, znovu používat kód a předávat hodnoty mezi různými částmi programu. Funkce můžeme deklarovat uvnitř JS několika způsoby.

Funkce v JS nemá povinný výstup a ani vstup pomoci parametrů. Můžeme tedy udělat funkci tzv. se side-effect, která něco udělá, ale nic nevrací.

Uvnitř funkci máme kličové slovo `return`, které nám dovolí vrátit hodnotu z funkce. Pokud funkce nedosáhne na `return`, vrátí `undefined`.

Uvnitř funkce poté můžeme používat kličové slovo `this`, které odkazuje na kontext, ve kterém byla funkce volána. Uvnitř tříd, viz @objekty, `this` odkazuje na aktuální instanci třídy a mimo

třídy odkazuje na globální objekt (v prohlížeči je to `window`, v Node.js je to `global`). Různými deklaracemi funkcí se chování `this` může lišit.

4.2.5.1 Deklarace funkcí

Nejstarší a nejběžnější způsob deklarace funkce je pomocí klíčového slova `function`. Jméno takovéto funkce je hoistováno[9] na začátek svého rozsahu (scope). `this` uvnitř takovéto funkce odkazuje na kontext, ve kterém byla funkce volána.

javascript

Deklarace funkce pomocí `function`

```
1 function greet(name) {
2   return `Hello, ${name}!`;
3 }
4 console.log(greet("Alice"));
```

Další možností je přiřadit anonymní funkci do proměnné. Podle použití můžeme použít `var`, `let` nebo `const` a tím se poté liší i hoistování a rozsah (scope). `this` uvnitř takovéto funkce odkazuje na kontext, ve kterém byla funkce volána.

javascript

Deklarace anonymní funkce přiřazené do proměnné

```
1 const greet = function(name) {
2   return `Hello, ${name}!`;
3 };
4 console.log(greet("Bob"));
```

Nejmodernější možností je použití šipkových funkcí (arrow functions), které byly zavedeny v ECMAScript 6 (ES6). Šipkové funkce mají kratší syntaxi a nehoistují se. `this` uvnitř šipkové funkce neodkazuje na kontext, ve kterém byla funkce volána, ale na kontext, ve kterém byla funkce definována. To znamená, že `this` uvnitř šipkové funkce je stejné jako `this` v okolním kódu. Což je asi nejdůležitější rozdíl oproti běžným funkcím a proto se šipkové funkce velmi často používají.

javascript

Deklarace šipkové funkce

```
1 const greet = (name) => {
2   return `Hello, ${name}!`;
3 };
4 console.log(greet("Charlie"));
```

4.2.5.2 Parametry funkcí

Funkce mohou přijímat libovolný počet parametrů, které jsou uvedeny v závorkách za jménem funkce. Parametry jsou odděleny čárkou a můžeme jim přiřadit výchozí hodnoty. Výchozí hodnoty řadíme pouze na konec seznamu parametrů.

javascript

Parametry s výchozími hodnotami

[9] Tady je hoistování poměrně žádaná funkce.

```javascript
function greet(name, greeting = "Hello") {
  return `${greeting}, ${name}!`;
}
console.log(greet("Alice")); // Hello, Alice!
console.log(greet("Bob", "Hi")); // Hi, Bob!
```

Funkce mohou přijímat libovolný počet parametrů, ale pokud jich přijmou více, než je uvedeno v deklaraci, zbytek je ignorován. Pokud jich přijmou méně, než je uvedeno v deklaraci, chybějící parametry jsou nastaveny na `undefined`. Pro práci s libovolným počtem parametrů můžeme použít rozšiřující operátor (spread/rest) `...`, který shromažďuje všechny zbývající argumenty do pole.

javascript

Použití rozšiřujícího operátoru pro shromažďování parametrů do pole

```javascript
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3)); // 6
```

4.2.5.3 Rekurze

Funkce může volat sama sebe, což se nazývá rekurze. Rekurze je užitečná pro řešení problémů, které lze rozdělit na menší podproblémy stejného typu. Při použití rekurze je důležité mít podmínku pro ukončení, aby se zabránilo nekonečné smyčce.

javascript

Příklad rekurzivní funkce pro výpočet faktoriálu

```javascript
function factorial(n) {
  if (n <= 1) {
    return 1;
  }
  return n * factorial(n - 1);
}
console.log(factorial(5)); // 120
```

4.2.6 Nejčastější data a funkce

V této části si popíšeme nejčastěji používané datové struktury a funkce v JavaScriptu.

Konzole:

Máme k dispozici objekt `console`, který nám dovolí vypisovat informace na standardní výstup. V rámci prohlížeče se jedná o konzoli uvnitř nástrojů pro vývojáře (DevTools). V rámci Node.js se jedná o terminál, ve kterém je Node.js spuštěn.

`console.log(a)` vypíše zprávu na standardní výstup,
`console.error(a)` vypíše chybovou zprávu na standardní chybový výstup,
`console.warn(a)` vypíše varovnou zprávu,
`console.info(a)` vypíše informační zprávu,
`console.debug(a)` vypíše ladicí zprávu,
`console.table(a)` vypíše data ve formě tabulky,

`console.group(a)` začne nový skupinový blok zpráv,
`console.groupEnd(a)` ukončí aktuální skupinový blok zpráv,
`console.time(a)` začne měřit čas pro daný štítek,
`console.timeEnd(a)` ukončí měření času pro daný štítek a vypíše výsledek.

Matematické funkce:

Máme k dispozici objekt `Math`, který nám dovolí provádět různé matematické operace.
`Math.abs(x)` vrátí absolutní hodnotu čísla x,
`Math.ceil(x)` vrátí nejmenší celé číslo větší nebo rovno x,
`Math.floor(x)` vrátí největší celé číslo menší nebo rovno x,
`Math.round(x)` vrátí nejbližší celé číslo k x,
`Math.max(a, b, ...)` vrátí největší číslo z daných argumentů,
`Math.min(a, b, ...)` vrátí nejmenší číslo z daných argumentů,
`Math.pow(base, exponent)` vrátí hodnotu base umocněnou na exponent,
`Math.sqrt(x)` vrátí druhou odmocninu čísla x,
`Math.random()` vrátí náhodné číslo mezi 0 (včetně) a 1 (exkluzivně),
`Math.sin(x)`, **`Math.cos(x)`**, **`Math.tan(x)`** vrátí sinus, kosinus a tangens úhlu x (v radiánech).
`Math.log(x)` vrátí přirozený logaritmus čísla x,
`Math.exp(x)` vrátí hodnotu e umocněnou na x (kde e je Eulerovo číslo),
`Math.PI` vrátí hodnotu π (pí),
`Math.E` vrátí hodnotu e (Eulerovo číslo).

Řetězce:

Řetězce (stringy) jsou sekvence znaků uzavřené v uvozovkách (`'`, `"`, `` ` ``). Zápis pomocí zpětných uvozovek (`` ` ``) nám dovolí vkládat proměnné a výrazy přímo do řetězce pomocí `${expression}`.

javascript

Použití šablonových řetězců s vloženými výrazy

```javascript
let name = "Alice";
let greeting = `Hello, ${name}!`;
console.log(greeting);
```

`str.length` vrátí délku řetězce,
`str.toUpperCase()` vrátí řetězec převedený na velká písmena,
`str.toLowerCase()` vrátí řetězec převedený na malá písmena,
`str.charAt(index)` vrátí znak na daném indexu,
`str.indexOf(substring)` vrátí index prvního výskytu podřetězce nebo −1, pokud není nalezen,
`str.lastIndexOf(substring)` vrátí index posledního výskytu podřetězce nebo −1, pokud není nalezen,
`str.includes(substring)` vrátí true, pokud řetězec obsahuje podřetězec, jinak false,
`str.startsWith(substring)` vrátí true, pokud řetězec začíná podřetězcem, jinak false,
`str.endsWith(substring)` vrátí true, pokud řetězec končí podřetězcem, jinak false,
`str.slice(start, end)` vrátí podřetězec od indexu start do end (end není zahrnut),
`str.substring(start, end)` vrátí podřetězec od indexu start do end (end není zahrnut),
`str.replace(searchValue, newValue)` vrátí nový řetězec, kde je první výskyt searchValue nahrazen newValue,
`str.replaceAll(searchValue, newValue)` vrátí nový řetězec, kde jsou všechny výskyty searchValue nahrazeny newValue,
`str.split(separator)` rozdělí řetězec na pole podřetězců podle zadaného oddělovače,
`str.trim()` vrátí řetězec bez bílých znaků na začátku a na konci,

`str.padStart(targetLength, padString)` vrátí řetězec doplněný na začátku na zadanou délku pomocí padString,
`str.padEnd(targetLength, padString)` vrátí řetězec doplněný na konci na zadanou délku pomocí padString.

Čísla:

Čísla v JavaScriptu jsou reprezentována různými způsoby, včetně celých čísel, desetinných čísel a speciálních hodnot jako `Infinity`, `-Infinity` a `NaN` (Not a Number).
`Number.isNaN(value)` vrátí true, pokud je hodnota NaN, jinak false,
`Number.isFinite(value)` vrátí true, pokud je hodnota konečné číslo, jinak false,
`Number.parseInt(string, radix)` převede řetězec na celé číslo v zadané soustavě (radix),
`Number.parseFloat(string)` převede řetězec na desetinné číslo,
`num.toFixed(digits)` vrátí řetězec reprezentující číslo zaokrouhlené na zadaný počet desetinných míst,
`num.toString(radix)` vrátí řetězec reprezentující číslo v zadané soustavě (radix),
`Number(value)` převede hodnotu na číslo.

Pro převod čísel vždy používejte funkci `Number()`, protože jiné způsoby mohou vést k nečekaným výsledkům.

4.2.7 Struktury

V JavaScriptu máme několik základních datových struktur, které nám dovolí organizovat a spravovat data. Jednotlivé struktury si nyní popíšeme.

4.2.7.1 Pole

Pole (arrays) jsou uspořádané kolekce hodnot, které mohou obsahovat různé datové typy. Oproti jiným jazykům jsou pole v JavaScriptu dynamická, což znamená, že jejich velikost může měnit během běhu programu. V podstatě se jedná o list. Interně je pole reprezentováno jako objekt s číselnými klíči.

javascript

Základní použití pole

```javascript
let array = [1, "two", true, null];
console.log(array[0]);
console.log(array.length);
array.push(5);
console.log(array);
array[2] = "three";
```

V poli objekty indexujeme čísly, které začínají na nule. Uvnitř JS máme koncept hodnoty `empty` (prázdná hodnota), což je hodnota, která není definována. Toto je pouze vnitřní koncept a nemáme k ní přímý přístup. Pokud přistoupíme k indexu, který není definován, vrátí se nám `undefined`.

Pole mají mnoho vestavěných metod, které nám dovolí s nimi pracovat.

`array.push(value)` přidá hodnotu na konec pole,

`array.pop()` odstraní a vrátí poslední hodnotu z pole,

`array.shift()` odstraní a vrátí první hodnotu z pole,

`array.unshift(value)` přidá hodnotu na začátek pole,

`array.slice(start, end)` vrátí nový podřetězec od indexu start do end (end není zahrnut),

`array.splice(start, deleteCount, item1, item2, ...)` odstraní deleteCount prvků od indexu start a vloží nové položky,

`array.indexOf(value)` vrátí index prvního výskytu hodnoty nebo −1, pokud není nalezena,

`array.includes(value)` vrátí true, pokud pole obsahuje hodnotu, jinak false.

`array.forEach(callback)` provede funkci callback pro každý prvek pole,

`array.map(callback)` vrátí nové pole s výsledky volání funkce callback pro každý prvek,

`array.filter(callback)` vrátí nové pole obsahující prvky, pro které funkce callback vrací true,

`array.reduce(callback, initialValue)` vrátí jedinou hodnotu získanou aplikací funkce callback na každý prvek pole,

`array.find(callback)` vrátí první prvek, pro který funkce callback vrací true, nebo undefined, pokud žádný takový prvek neexistuje,

`array.findIndex(callback)` vrátí index prvního prvku, pro který funkce callback vrací true, nebo −1, pokud žádný takový prvek neexistuje,

`array.every(callback)` vrátí true, pokud funkce callback vrací true pro všechny prvky, jinak false,

`array.some(callback)` vrátí true, pokud funkce callback vrací true pro alespoň jeden prvek, jinak false.

`array.sort(compareFunction)` seřadí prvky pole podle zadané funkce porovnání,

`array.reverse()` obrátí pořadí prvků v poli,

`array.join(separator)` vrátí řetězec vytvořený spojením všech prvků pole pomocí zadaného oddělovače,

`array.concat(array2, array3, ...)` vrátí nové pole, které je spojením původního pole s dalšími poli,

`array.length` vrátí počet prvků v poli,

`Array.isArray(value)` vrátí true, pokud je hodnota pole, jinak false.

Pole procházíme pomocí cyklů, například `for`, `for...of` nebo metod jako `forEach`. Metodu `forEach` avšak nedoporučuji používat, protože je to cyklus, který avšak nedovoluje použít `break` nebo `continue`.

S polem je spojený operátor destrukturingu, který nám dovolí rozložit pole na jednotlivé prvky. Například můžeme přímo z pole přiřadit hodnoty do proměnných.

javascript

Destrukturing s polem

```
1 let [first, second, ...rest] = [1, 2, 3, 4, 5];
2 console.log(first); // 1
3 console.log(second); // 2
4 console.log(rest); // [3, 4, 5]
```

4.2.7.2 Množiny

Množiny (sets) jsou kolekce unikátních hodnot, což znamená, že každá hodnota se v množině může objevit pouze jednou.

javascript

Základní použití množiny

```
let set = new Set();
set.add(1);
set.add(2);
set.add(2);
```

V rámci množiny máme několik užitečných metod.

`set.add(value)` přidá hodnotu do množiny,
`set.delete(value)` odstraní hodnotu z množiny,
`set.has(value)` vrátí true, pokud množina obsahuje hodnotu, jinak false,
`set.clear()` odstraní všechny hodnoty z množiny,
`set.size` vrátí počet hodnot v množině.

4.2.7.3 Mapy

Mapy (maps) jsou kolekce párů klíč-hodnota, kde klíče mohou být libovolného datového typu. Oproti bojektům, viz další Kapitola, mapy zachovávají pořadí vložení a umožňují libovolné datové typy jako klíče.

javascript

Základní použití mapy

```
let map = new Map();
map.set("name", "Alice");
map.set(1, "one");
map.set(true, "boolean");
console.log(map.get("name"));
```

V rámci mapy máme několik užitečných metod.

`map.set(key, value)` přidá nebo aktualizuje pár klíč-hodnota v mapě,
`map.get(key)` vrátí hodnotu spojenou s klíčem nebo undefined, pokud klíč neexistuje,
`map.has(key)` vrátí true, pokud mapa obsahuje klíč, jinak false,
`map.delete(key)` odstraní pár klíč-hodnota z mapy,
`map.clear()` odstraní všechny páry klíč-hodnota z mapy,
`map.size` vrátí počet párů klíč-hodnota v mapě.

4.2.7.4 Objekty

Objekty (objects) jsou kolekce vlastností, kde každá vlastnost je pár klíč-hodnota. Klíče jsou vždy řetězce nebo symboly a hodnoty mohou být libovolného datového typu.

javascript

Základní použití objektu

```
let obj = {
  name: "Alice",
```

```
3   age: 30,
4   isStudent: false
5 };
6 console.log(obj.name);
7 obj.age = 31;
8 console.log(obj);
```

K jednotlivým vlastnostem objektu můžeme přistupovat pomocí tečkové notace (`obj.key`) nebo hranatých závorek (`obj["key"]`). Hranaté závorky nám dovolí použít klíče, které nejsou platnými identifikátory (např. obsahují mezery nebo speciální znaky) nebo klíče, které jsou uloženy v proměnné.

V rámci objektu máme několik užitečných metod a vlastností.
`Object.keys(obj)` vrátí pole obsahující všechny klíče objektu,
`Object.values(obj)` vrátí pole obsahující všechny hodnoty objektu,
`Object.entries(obj)` vrátí pole obsahující všechny páry klíč-hodnota jako podpole,
`Object.assign(target, source)` zkopíruje všechny vlastní vlastnosti z objektu source do objektu target,
`obj.hasOwnProperty(key)` vrátí true, pokud objekt obsahuje daný klíč jako vlastní vlastnost, jinak false.

Pokud přistoupíme k neexistující vlastnosti objektu, vrátí se nám `undefined`.

Destrukturing nám dovolí rozložit objekt na jednotlivé vlastnosti, stejně jako u pole.

javascript

Destrukturing s objektem

```
1 let { name, age } = { name: "Alice", age: 30 };
2 console.log(name); // Alice
3 console.log(age); // 30
```

4.2.7.5 Datum a čas

Pro práci s datem a časem máme v JavaScriptu objekt `Date`.

javascript

Základní použití objektu Date

```
1 let now = new Date();
2 console.log(now.toString());
3 let specificDate = new Date("2023-01-01T00:00:00Z");
4 console.log(specificDate.toISOString());
```

Datum a čas se v počítači ukládají jako počet milisekund od 1. ledna 1970 (tzv. Unix epoch) nebo jako formátovaný řetězec. V rámci objektu Date máme několik užitečných metod.
`new Date()` vytvoří nový objekt Date s aktuálním datem a časem,
`new Date(dateString)` vytvoří nový objekt Date z formátovaného řetězce,
`new Date(year, month, day, hours, minutes, seconds, milliseconds)` vytvoří nový objekt Date z jednotlivých komponent,
`date.getTime()` vrátí počet milisekund od 1. ledna 1970,
`date.getFullYear()` vrátí rok,
`date.getMonth()` vrátí měsíc (0-11)[10],
`date.getDate()` vrátí den v měsíci (1-31),

`date.getHours()` vrátí hodiny (0-23),
`date.getMinutes()` vrátí minuty (0-59),
`date.getSeconds()` vrátí sekundy (0-59),
`date.getMilliseconds()` vrátí milisekundy (0-999),
`date.toString()` vrátí formátovaný řetězec reprezentující datum a čas,
`date.toISOString()` vrátí ISO 8601 formátovaný řetězec,
`date.toLocaleString()` vrátí lokalizovaný řetězec reprezentující datum a čas,
`date.setFullYear(year)`, `date.setMonth(month)`, `date.setDate(day)`, `date.setHours(hours)`, `date.setMinutes(minutes)`, `date.setSeconds(seconds)`, `date.setMilliseconds(milliseconds)` nastaví jednotlivé komponenty data a času,
`date.toUTCString()` vrátí datum a čas ve formátu UTC,
`date.toDateString()` vrátí pouze datum jako řetězec,
`date.toTimeString()` vrátí pouze čas jako řetězec.

Důležité v rámci webových aplikací je, že uživatelskému datu a času se nedá věřit. Práce s datem a časem je velmi komplikovaná a proto doporučuji používat knihovny, které vám například pomohou s časovými pásmy, formátováním a parsováním.

4.2.8 Objektový návrh

Objektově orientované programování (OOP) je programovací paradigma, které využívá objekty a jejich interakce k návrhu softwaru.

V JavaScriptu můžeme vytvářet objekty pomocí tříd (classes), které byly zavedeny v ECMAScript 6 (ES6). Jedná se čistě teoreticky syntaktický cukr nad prototypovým dědičností, ale je to mnohem přehlednější a jednodušší na použití. Proto se jinými způsoby tvorby objektů nebudeme zabývat.

Třída je šablona pro vytváření objektů s předdefinovanými vlastnostmi a metodami. Třída popisuje tedy obecný, abstraktní koncept – například Uživatel. Každá instance třídy pak představuje konkrétní objekt – například konkrétní uživatel, například Pepa.

Uvnitř třídy definujeme:

Vlastnosti (properties) v podstatě proměnné, atributy, které popisují stav objektu,
Metody (methods) funkce, které definují chování objektu.

Instance je poté vlastně objekt. Novou instanci třídy tvoříme pomocí operátoru `new`. Poté následuje jméno třídy a závorky.

javascript

Základní definice třídy a vytvoření instance

```javascript
class User {
  name;
  roles;

  addRole(role) {
    if (!this.roles.includes(role)) {
      this.roles.push(role);
    }
  }
}
```

[10] Ano, opravdu 0 je leden.

```
11
12 const pepa = new User();
13 pepa.name = "Pepa";
14 pepa.roles = ["admin", "user"];
15
16 pepa.addRole("editor");
```

Ve výše uvedeném příkaldu máme třídu `User` s vlastnostmi `name` a `roles` a metodou `addRole`. Uvádět takto vlastnosti avšak není povinné. Děláme to pouze pro přehlednost a dokumentaci kódu.

Uvnitř třídy, respektive metod, přistupujeme k vlastnostem a metodám pomocí klíčového slova `this`, které odkazuje na aktuální instanci třídy. Metody jsou logicky hoistované a reprezentují klasicky definované funkce. Jako těmto funkcím můžeme předávat parametry a vracet hodnoty.

Znovu, přístupem k neexistující vlastnosti objektu získáme `undefined`.

4.2.8.1 Konstruktory

Konstruktory jsou speciální metody, které se volají při vytváření nové instance třídy. Každá třída má implicitní konstruktor, pokud není definován vlastní. Konstruktor se definuje pomocí klíčového slova `constructor`. Můžeme mu předat parametry, které nám dovolí inicializovat vlastnosti instance při jejím vytvoření.

Proč používáme konstruktory je jednoduchá otázka. Oni tam totiž zajistí, že stav instance je vždy konzistentní s tím, co daný objekt reprezentuje. Například v ukázce v minule Kapitole, jsme mohli vytvořit instanci `User` bez jména a rolí. To nedává smysl, protože uživatel by měl mít vždy jméno a role musí být alespoň definovány jako prázdné pole.

javascript

Použití konstruktoru pro inicializaci vlastností

```
1  class User {
2    name;
3    roles;
4
5    constructor(name, roles = []) {
6      this.name = name;
7      this.roles = roles;
8    }
9
10   addRole(role) {
11     if (!this.roles.includes(role)) {
12       this.roles.push(role);
13     }
14   }
15 }
16 const pepa = new User("Pepa", ["admin", "user"]);
```

4.2.8.2 Dědičnost

Dědičnost nám dovoluje vytvářet nové třídy na základě existujících tříd. Nová třída (podtřída) dědí vlastnosti a metody z existující třídy (nadtřídy) a může přidat nové vlastnosti a metody nebo přepsat (override) existující. Pro dědičnost používáme klíčové slovo `extends`.

Dědičnost nám dovoluje vytvářet hierarchie tříd a znovu používat kód. Navíce se nám to hodí u abstrakce a polymorfismu.

Pokud třída, ze které dědíme, má konstruktor, musíme v podtřídě zavolat konstruktor nadtřídy pomocí klíčového slova `super` jako první řádek konstruktoru podtřídy.

javascript

Příklad dědičnosti

```javascript
class Admin extends User {
  constructor(name, roles = []) {
    super(name, roles);

    this.addRole("admin");
  }

  addRole(role) {
    // Nemůžeme přidat roli
  }
}

const admin = new Admin("Admin", ["editor"]);
```

4.2.8.3 Abstrakce a polymorfismus

Abstrakce nám dovoluje skrýt složitost a vystavit pouze nezbytné části objektu. Polymorfismus nám dovoluje používat objekty různých tříd prostřednictvím společného rozhraní. To znamená, že můžeme volat stejné metody na objektech různých tříd, aniž bychom museli znát jejich konkrétní typ.

Například v poli si uložíme veškeré uživatele. O nich budeme vědět, že musí být instance třídy `User`, ale nemusíme vědět, zda se jedná o běžného uživatele nebo administrátora. Poté můžeme na tomto poli volat metodu `addRole` bez ohledu na konkrétní typ uživatele. Každá instance, resp. třída, si s tímto voláním poradí podle svého.

4.2.8.4 Použití

Objektově orientované programování je velmi široké téma a v této kapitole jsme si popsali pouze základy. Pro hlubší pochopení doporučuji prostudovat další zdroje a knihy věnované OOP v JavaScriptu.

Nejdůležitější jsou následující principy:

Zapouzdření (Encapsulation) skrývání vnitřního stavu objektu a vystavení pouze nezbytných částí, například pomocí getterů a setterů,

Privátní vlastnosti a metody v JavaScriptu můžeme definovat privátní vlastnosti a metody pomocí prefixu #, což znamená, že jsou přístupné pouze uvnitř třídy,

Statické vlastnosti a metody můžeme definovat vlastnosti a metody, které patří třídě jako celku, nikoli jednotlivým instancím, pomocí klíčového slova `static`.

Pro třídy máme k dispozici i operátor `instanceof`, který nám dovolí zjistit, zda je objekt instance dané třídy nebo její podtřídy.

javascript

Použití operátoru instanceof

```
1 console.log(pepa instanceof User); // true
2 console.log(admin instanceof User); // true
3 console.log(admin instanceof Admin); // true
4 console.log(pepa instanceof Admin); // false
```

4.3 Výjimky a zpracování chyb

Výjimky jsou speciální události, které naruší normální tok programu. Dojde k nim právě tehdy, když nastane chyba, ze které se program nedokáže dostat zpět do normálního toku. Často se to stává při různé práci s externími zdroji, jako je uživatel, soubory či cokoliv dalšího.

Proč se neumí program sám ošetřit je jednoduché – program nemůže vědět, jak se má zachovat v každé možné situaci. Kdyby se například četl vstup od uživatele a uživatel zadal něco nečekaného, program by si mohl výsledek interpretovat jak by chtěl a to by vedlo k nečekaným chybám.

Proto je vždy na uživateli aby určil, jak se má program v takové situaci zachovat. V JavaScriptu máme pro zpracování výjimek klíčová slova `try`, `catch`, `finally` a `throw`.

javascript

Základní struktura pro zpracování výjimek

```
1 try {
2   // Kód, který může vyvolat výjimku
3 } catch (error) {
4   // Kód pro zpracování výjimky
5 } finally {
6   // Kód, který se vždy provede, bez ohledu na to, zda došlo k výjimce
  nebo ne
7 }
```

V bloku `try` umístíme kód, který může vyvolat výjimku. Pokud dojde k výjimce, program přeruší vykonávání kódu v bloku `try` a přejde do bloku `catch`, kde můžeme výjimku zpracovat. Kód po vyvolání výjimky v bloku `try` se již neprovede. Blok `finally` je volitelný a obsahuje kód, který se vždy provede, bez ohledu na to, zda došlo k výjimce nebo ne.

Program vždy hledá nejbližší blok `try-catch`, který obaluje kód, ve kterém došlo k výjimce. Pokud žádný takový blok neexistuje, program skončí s chybou.

> 👮 **Hlavní blok try-catch**
>
> Pro praxi je běžné, že veškerý kód je obalen v hlavním bloku `try-catch`, který zachytí všechny neošetřené výjimky a například je zaloguje. Nechceme totiž aby program spadl a nedal uživateli žádnou zpětnou vazbu.

Pro vyvolání výjimky používáme klíčové slovo `throw`, za kterým následuje hodnota, která reprezentuje výjimku.

javascript

Vyvolání a zpracování výjimky

```javascript
function divide(a, b) {
  if (b === 0) {
    throw new Error("Division by zero");
  }
  return a / b;
}
try {
  console.log(divide(4, 2)); // 2
  console.log(divide(4, 0)); // Vyvolá výjimku
} catch (error) {
  console.error("Error:", error.message);
}
```

V JavaScriptu máme několik vestavěných typů výjimek, které můžeme použít nebo od nich dědit. Jedná se o následující typy:
`Error` základní typ výjimky,
`TypeError` vyvolá se, když dojde k chybě typu,
`RangeError` vyvolá se, když je hodnota mimo povolený rozsah,
`ReferenceError` vyvolá se, když je odkaz na neexistující proměnnou,
`SyntaxError` vyvolá se, když je chyba v syntaxi,
`EvalError` vyvolá se, když dojde k chybě v eval funkci,
`URIError` vyvolá se, když je chyba v URI funkci[11].

Můžeme také vytvářet vlastní typy výjimek děděním od vestavěných typů.

Obecně jsou výjimky ale považovány za špatný návrhový vzor a měly by se používat pouze pro neočekávané situace. Pro běžné chyby, které můžeme očekávat, je lepší používat návratové hodnoty nebo jiné mechanismy. Například je vhodné obalit každý výsledek funkce do objektu, který bude obsahovat informaci o tom, zda operace proběhla úspěšně nebo ne.

javascript

Použití návratové hodnoty pro zpracování chyby

```javascript
function safeDivide(a, b) {
  if (b === 0) {
    return { success: false, error: "Division by zero" };
  }
  return { success: true, result: a / b };
}
const result = safeDivide(4, 0);
if (result.success) {
  console.log("Result:", result.result);
} else {
  console.error("Error:", result.error);
}
```

[11] URL a URI se liší tím, že URI je obecnější pojem, který zahrnuje i URL.

V rámci TypeScriptu, se kterým se později setkáme, je běžné používat tzv. `Either` typ, který reprezentuje buď úspěšný výsledek nebo chybu.

4.4 Asynchronní JavaScript

Prvně si vysvětlíme rozdíl mezi synchronním, asynchronním a paralelním programem.

Synchronní program vykonává kód řádek po řádku, jeden po druhém, resp. jednotlivé operace čekají na dokončení před tím, než se vykoná další.

Asynchronní program umožňuje vykonávat kód, který může trvat delší dobu, aniž by blokoval hlavní tok programu. Tedy program může pokračovat v práci, zatímco čeká na dokončení asynchronní operace.

Paralelní program umožňuje vykonávat více operací současně, což může zlepšit výkon při práci s více jádry procesoru nebo při čekání na I/O operace.

Pro nás je důležité, že JS je jednovláknový jazyk. To znamená, že může vykonávat pouze jeden úkol najednou[12]. Zároveň ale máme k dispozici asynchronní operace, které nám dovolí vykonávat kód, který může trvat delší dobu, aniž by blokoval hlavní tok programu.

Bez asynchronního kódu by byly webové stránky v podstatě nepoužitelné. S celým tímto konceptem přišla technologie AJAX (Asynchronous JavaScript and XML), která nám dovolila načítat data ze serveru na pozadí bez nutnosti znovu načítat celou stránku. Od té doby se tento koncept rozšířil na to, že webové stránky můžeme používat i během toho, co dělají nějaký výpočet.

Asynchronnost v JS jste již viděli, jen jste si to možná neuvědomili. Například funkce `setTimeout` a `setInterval` jsou asynchronní. V praxi tyto funkce způsobí to, že funkci, kterou předáme jako první argument, se zpozdí či opakuje dle hodnoty druhého argumentu. To je taky důvod nepřesnosti těchto funkcí, protože nám garantují pouze to, že se funkce spustí nejdříve po uplynutí zadaného času. Horní hranice není garantována, protože JS může být zaneprázdněn jiným kódem.

Výše uvedené funkce a obecně starší JS kód používá tzv. callbacky, což jsou funkce předané jako argumenty do jiných funkcí. To ale vede k tzv. callback hell, což je situace, kdy máme mnoho vnořených callbacků, které vedou k nečitelnému a těžko udržovatelnému kódu.

4.4.1 Sliby

Moderní způsob, jak pracovat s asynchronním kódem v JS, jsou sliby (promises). Slib je objekt, který představuje budoucí hodnotu nebo chybu asynchronní operace. Slib může být ve třech stavech:
Pending (čekající) počáteční stav, operace ještě nebyla dokončena,
Fulfilled (splněný) operace byla úspěšně dokončena a slib má hodnotu,
Rejected (zamítnutý) operace selhala a slib má důvod chyby.

Sliby nám dovolí řetězit asynchronní operace pomocí metod `then` a `catch`. Takže můžeme jednoduše říct, že pokud se dokončí jedna akce, má na to navázat další akce.

javascript

```
Základní použití slibu
```

[12] Příští rok si ukážeme, je pomocí různých API je možné spouštět kód ve více vláknech.

```javascript
let promise = ...; // prozatím neznámý slib
promise
  .then((value) => {
    // Kód pro zpracování úspěšného výsledku
  })
  .catch((error) => {
    // Kód pro zpracování chyby
  });
```

Samotné funkce `then` a `catch` vrací nový slib, což nám dovolí řetězit více asynchronních operací.

javascript

Řetězení slibů

```javascript
promise.then((value) => {
  // Kód pro zpracování úspěšného výsledku
  return anotherPromise; // Může vrátit další slib
}).then((anotherValue) => {
  // Kód pro zpracování výsledku dalšího slibu
}).catch((error) => {
  // Kód pro zpracování chyby z jakéhokoli slibu v řetězci
});
```

Sliby většinou netvoříme ručně, ale zejména je využíváme při používání různých API prohlížeče[13]. Přesto ale můžeme tvořit vlastní sliby pomocí konstruktoru `Promise`. Konstruktor přijímá jako argument funkci, která má dva parametry: `resolve` a `reject`. Tyto parametry jsou funkce, které použijeme k označení slibu jako splněného nebo zamítnutého s příslušnou hodnotou nebo chybou tím, že je zavoláme.

javascript

Vytvoření vlastního slibu

```javascript
let myPromise = new Promise((resolve, reject) => {
  // Asynchronní operace
  let success = true; // nebo false v případě chyby
  if (success) {
    resolve("Operation successful");
  } else {
    reject("Operation failed");
  }
});
```

Běhové prostředí ukončí program až v době, kdy jsou všechny sliby vyřešeny.

4.4.2 Pomocná API

Pro práci s asynchronním kódem máme v JS několik vestavěných API, které nám to usnadní.

`Promise.all(iterable)` vrátí nový slib, který je splněn, když jsou všechny sliby v iterovatelné kolekci splněny, nebo zamítnut, pokud je některý z nich zamítnut,

[13] Protože čekáme na reakci prohlížeče, uživatele nebo další strany!

`Promise.race(iterable)` vrátí nový slib, který je splněn nebo zamítnut, jakmile je některý ze slibů v iterovatelné kolekci splněn nebo zamítnut,
`Promise.any(iterable)` vrátí nový slib, který je splněn, jakmile je některý ze slibů v iterovatelné kolekci splněn, nebo zamítnut, pokud jsou všechny zamítnuty,
`Promise.allSettled(iterable)` vrátí nový slib, který je splněn, když jsou všechny sliby v iterovatelné kolekci buď splněny nebo zamítnuty, s polem výsledků,
`Promise.resolve(value)` vrátí slib, který je splněn s danou hodnotou,
`Promise.reject(reason)` vrátí slib, který je zamítnut s daným důvodem.

Přesto toto používání slibů vede k nečitelnému kódu, protože máme mnoho vnořených funkcí – vážeme se na vázané a hodně to připomíná callbacky. Proto byl zaveden nový syntaktický cukr nad sliby – `async` a `await`. Tyto dvě nová klíčová slova nahrazují použití `then` a `catch` a dovolí nám psát asynchronní kód, který vypadá jako synchronní přímo v rámci funkce.

javascript

Použití async a await

```
async function fetchData() {
  try {
    let response = await waitFor(3000);
    console.log(response);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}
```

V ukázce je vidět, že místo `catch` používáme `try-catch`, což je běžný způsob zpracování výjimek v synchronním kódu. Funkce označené jako `async` vždy vrací slib. Klíčové slovo `await` můžeme použít pouze uvnitř `async` funkce a způsobí, že funkce počká na vyřešení slibu před tím, než pokračuje dál.

4.4.3 Fetch API

Nejpoužívanější API pro práci s asynchronním kódem je Fetch API, které nám dovolí načítat zdroje přes síť. Například to budeme používat až se setkáme s API. Tedy, že od jiného serveru budeme chtít získat nějaká data. Alternativou v JS DOM je `XMLHttpRequest`, které je ale složitější na použití a proto se dnes již moc nepoužívá[14].

Funkce `fetch` přijímá jako první argument URL, kterou chceme načíst, a volitelný druhý argument s možnostmi požadavku. Funkce vrací slib, který je splněn s objektem `Response`, když je požadavek úspěšný, nebo zamítnut s chybou, pokud požadavek selže.

javascript

Základní použití Fetch API

```
fetch("https://api.example.com/data")
  .then((response) => {
    if (!response.ok) {
      throw new Error("Network response was not ok");
    }
```

[14] Reálně používá, protože má nějaké výhody (postupné načítání), které Fetch API nenabízí ale naopak nejsou tak důležité pro každý web.

```
6     return response.json(); // nebo response.text(), response.blob(),
  atd.
7   })
8   .then((data) => {
9     console.log(data);
10  })
11  .catch((error) => {
12    console.error("Fetch error:", error);
13  });
```

Na prvotním výsledku `Response` vidíme, že musíme zkontrolovat, zda byl požadavek úspěšný pomocí vlastnosti `ok`. Poté můžeme zavolat metodu pro získání dat v požadovaném formátu, například `json()`, `text()`, `blob()` atd. Postupně jednotlivé metody vrací sliby, pro zpracování JSON, textu nebo binárních dat.

Do objektu nastavení požadavku můžeme uvést:
Metodu `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, atd. (výchozí je `GET`), která popisuje typ požadavku,
Hlavičky objekt s hlavičkami požadavku,
Tělo data, která chceme odeslat na server (používá se u metod jako `POST`, `PUT`, `PATCH`),
A další jako například `mode`, `credentials`, `cache` a další.

4.5 Balíčky a knihovny

Pro práci s JavaScriptem na serveru (Node.js) nebo pro správu závislostí ve webových projektech používáme balíčky a knihovny. Balíčky jsou sbírky kódu, které můžeme znovu použít v našich projektech. Knihovny jsou speciální typy balíčků, které poskytují specifickou funkcionalitu nebo API pro určité účely. V podstatě se jedná ale o to samé.

Pro správu balíčků a knihoven používáme nástroj npm (Node Package Manager), který je součástí Node.js. Pomocí npm můžeme instalovat, aktualizovat a odstraňovat balíčky a knihovny.

Každý projekt, který npm spravuje obsahuje soubor `package.json`, který obsahuje metadata o projektu, jako je název, verze, závislosti, skripty a další. Tento soubor je vytvořen pomocí příkazu `npm init` a můžeme ho upravovat ručně nebo pomocí dalších npm příkazů.

Ukázka souboru package.json

```
1 {
2   "name": "my-project",
3   "version": "1.0.0",
4   "description": "My first project",
5   "main": "index.js",
6   "scripts": {
7     "start": "node index.js",
8     "test": "echo \"Error: no test specified\" && exit 1"
9   },
10  "dependencies": {
11    "express": "^4.17.1"
12  }
13 }
```

Ve výše uvedeném příkladu máme základní strukturu souboru `package.json` s několika běžnými poli.
`name` název projektu,
`version` verze projektu,
`description` popis projektu,
`main` vstupní bod projektu,
`scripts` skripty, které můžeme spouštět pomocí npm,
`dependencies` závislosti projektu, které budou nainstalovány pomocí npm.

U každého balíčku je poté uvedena verze, kterou chceme použít. Můžeme úvést přesnou verzi, rozsah verzí nebo použít speciální znaky jako `^` nebo `~`, které nám dovolí automaticky aktualizovat na novější verze s určitými omezeními. Například `^1.2.3` nám dovolí aktualizovat na jakoukoliv verzi `1.x.x`, ale ne na `2.x.x`. Pro `~1.2.3` je to naopak, dovolí nám aktualizovat na jakoukoliv verzi `1.2.x`, ale ne na `1.3.x`.

Pro instalaci balíčků používáme příkaz `npm install <package-name>`, který nainstaluje balíček a přidá ho do souboru `package.json`. Jednotlivé balíčky se poté ukládají do složky `node_modules`, kterou byste neměli verzovat v systému pro správu verzí (například Git). Místo toho byste měli verzovat pouze soubor `package.json` a `package-lock.json`, který je automaticky generován npm a obsahuje přesné verze nainstalovaných balíčků.

Pro odstranění balíčků používáme příkaz `npm uninstall <package-name>`, který odstraní balíček a aktualizuje soubor `package.json`. Pro aktualizaci balíčků používáme příkaz `npm update`, který aktualizuje všechny balíčky na nejnovější verze podle zadaných verzí v souboru `package.json`. Pro spuštění skriptů definovaných v souboru `package.json` používáme příkaz `npm run <script-name>`.

Během tohoto roku (a v těchto skriptech) se s npm setkáme zejména při práci s různými knihovnami a nástroji, které nám usnadní vývoj webových aplikací. Prozatím nejpoužívanější knihovnou může být například `node-fetch`, což je implementace Fetch API pro Node.js.

4.5.1 Propojení souborů

V rámci větších projektů je běžné, že kód rozdělíme do více souborů. To nám dovolí lépe organizovat kód, znovu používat části kódu a spolupracovat s dalšími vývojáři. Pro propojení souborů v JavaScriptu máme dva hlavní způsoby: CommonJS a ES Modules. Pro moderní webové stránky se doporučují používat ES Modules, které jsou nativně podporovány v prohlížečích i v Node.js. Proto si ukážeme pouze tento způsob.

Pro použití ES Modules musíme v Node.js nastavit typ modulu na `module` v souboru `package.json` nebo použít příponu `.mjs` pro soubory.

Nastavení typu modulu v package.json[15]

```json
{
  // ...
  "type": "module"
  // ...
}
```

Pro exportování funkcí, objektů nebo hodnot z modulu používáme klíčové slovo `export`. Jednotlivé položky poté můžeme importovat v jiném souboru pomocí klíčového slova `import`.

[15] Mimochodem JSON nedovoluje komentáře, ale pro přehlednost je zde uvádíme.

javascript

Exportování funkcí a hodnot z modulu

```
// math.js
export function add(a, b) {
  return a + b;
}
export const PI = 3.14159;
```

javascript

Importování funkcí a hodnot z modulu

```
// main.js
import { add, PI } from "./math.js";
console.log(add(2, 3)); // 5
console.log(PI); // 3.14159
```

Jednotlivé importované položky poté můžeme přejmenovat pro případné kolize jmen pomocí klíčového slova `as`.

javascript

Přejmenování importovaných položek

```
import { add as sum, PI as piValue } from "./math.js";
console.log(sum(2, 3)); // 5
console.log(piValue); // 3.14159
```

Můžeme také importovat všechny exportované položky jako jeden objekt pomocí hvězdičky `*` a klíčového slova `as`.

javascript

Importování všech položek jako objekt

```
import * as math from "./math.js";
console.log(math.add(2, 3)); // 5
console.log(math.PI); // 3.14159
```

Další možností je exportovat jednu konkrétní položku jako výchozí export pomocí klíčového slova `default`.

javascript

Výchozí export funkce

```
// math.js
export default function multiply(a, b) {
  return a * b;
}
```

javascript

Importování výchozího exportu

```
// main.js
import multiply from "./math.js";
console.log(multiply(2, 3)); // 6
```

Výchozí export

Výchozí export osobně nedoporučuji používat, protože může vést k nejasnostem a kolizím jmen. Je lepší vždy používat pojmenované exporty, které jsou jasnější a lépe se s nimi pracuje. Navíce to do budoucna usnadní rozšíření modulu o další exporty.

4.5.2 Knihovny běhových prostředí

Knihovny běhových prostředí nám dovolí pracovat s různými funkcemi, které nejsou součástí samotného jazyka JavaScript. Tyto knihovny však musíme importovat a musí je mít k dispozici dané běhové prostředí. V Node.js bez knihovny máme k dispozici globální proměnné jako `console`, `setTimeout`, `process` a další.

Pro práci se soubory, cestami a operačním systémem máme v Node.js k dispozici několik vestavěných modulů. Budeme používat knihovnu `fs` (file system) pro práci se soubory, `path` pro práci s cestami a `os` pro získání informací o operačním systému.

Pro použití těchto modulů je musíme nejdříve importovat.

javascript

Importování vestavěných modulů v Node.js

```
1 import * as fs from "fs";
2 import * as path from "path";
3 import * as os from "os";
```

V jednotlivých modulech máme k dispozici jednotlivé funkce a objekty pro práci s danou oblastí.

Pro modul `fs`:
`fs.readFileSync(path, options)` synchronně čte obsah souboru,
`fs.writeFileSync(path, data, options)` synchronně zapisuje data do souboru,
`fs.readdirSync(path, options)` synchronně čte obsah adresáře
`fs.statSync(path)` synchronně získává informace o souboru nebo adresáři,
A další `fs.appendFileSync`, `fs.unlinkSync`, `fs.mkdirSync` a další.

Pro modul `path`:
`path.join(...paths)` spojuje více částí cesty do jedné,
`path.resolve(...paths)` řeší absolutní cestu,
`path.basename(path, ext)` získává název souboru z cesty,
`path.dirname(path)` získává adresářovou část cesty,
`path.extname(path)` získává příponu souboru z cesty,
A další `path.isAbsolute`, `path.relative`, `path.parse` a další.

Pro modul `os`:
`os.platform()` získává platformu operačního systému,
`os.arch()` získává architekturu procesoru,
`os.cpus()` získává informace o procesorech,
`os.totalmem()` získává celkovou paměť systému,
`os.freemem()` získává volnou paměť systému,
A další `os.uptime`, `os.userInfo`, `os.networkInterfaces` a další.

4.6 Debugging

Poslední kapitolou základního JavaScriptu je debugging, což je proces hledání a odstraňování chyb v kódu. Debugging je důležitá dovednost pro každého vývojáře, protože chyby jsou nevyhnutelnou součástí vývoje softwaru. Pro debugging máme k dispozici několik nástrojů a technik, které nám mohou pomoci najít a opravit chyby.

Nejdůležitější je používat konzoli[16] pro výpis informací o stavu programu. To lze taktéž používat pro ten nejzákladnější způsob debuggingu – přidávání výpisů na různá místa v kódu, abychom viděli, co se děje.

javascript

Základní použití console.log pro debugging

```javascript
console.log("Debug info:", variable);
```

Pro pokročilejší debugging máme k dispozici nástroje jako jsou DevTools v prohlížečích nebo vestavěný debugger v Node.js. Tyto nástroje nám dovolí nastavovat breakpointy, krokovat kódem, prohlížet hodnoty proměnných a další.

javascript

Nastavení breakpointu pomocí klíčového slova debugger

```javascript
debugger; // Nastaví breakpoint
```

Breakpoint je místo v kódu, kde se program zastaví a dovolí nám prohlížet stav programu. To nám dovolí krokovat kódem, prohlížet hodnoty proměnných a další. Pro použití debuggeru v Node.js můžeme spustit program s příznakem `--inspect` nebo `--inspect-brk`, který nám dovolí připojit se k debuggeru pomocí DevTools v prohlížeči.

bash

Spuštění Node.js s debuggerem

```bash
node --inspect index.js
```

Poté můžeme otevříit DevTools v prohlížeči a připojit se k běžícímu procesu Node.js. Například pomocí URL `chrome://inspect` v Google Chrome.

Podobný problém, který řešíme, je profilování výkonu, což je proces měření a optimalizace výkonu kódu. Při spuštění můžeme uvést přepínač `--prof`, který nám vygeneruje profil výkonu, který můžeme analyzovat pomocí nástrojů jako je `node --prof-process`.

[16] A samozřejmě jiné způsoby.

TypeScript

5.1 Co je TypeScript

TypeScript je nadmnožina JavaScriptu, která přidává statické typování a další funkce pro vývojáře.

TypeScript je metajazyk, což znamená, že kód napsaný v TypeScriptu je přeložen (transpilován) do čistého JavaScriptu, který může běžet v jakémkoli prostředí podporujícím JavaScript, včetně webových prohlížečů a Node.js. Zařizuje nám tedy typovou kontrolu během vývoje, což pomáhá odhalit chyby dříve než při běhu aplikace. Což je největší nevýhoda obyčejného JavaScriptu – stále musíme kontrolovat, zda proměnné mají očekávaný typ.

TypeScript je vyvíjen jako open-source projekt a má velkou komunitu vývojářů, kteří přispívají k jeho vylepšení a rozšiřování. Hlavní správce projektu je Microsoft, ale mnoho dalších přispěvatelů z celého světa se podílí na jeho vývoji.

V smyslu našeho kurzu je TypeScript důležitý, protože nám umožňuje psát robustnější a lépe strukturovaný kód pro webové aplikace. Pokud budeme dělat jakýkoliv větší projekt, je očekáváno, že budeme používat TypeScript.

5.1.1 Kompilace TypeScriptu

Kompilace TypeScriptu se provádí pomocí TypeScript kompilátoru (tsc), který převede TypeScript kód na JavaScript. Kompilátor může být spuštěn z příkazového řádku nebo integrován do vývojového prostředí (IDE).

```bash
1 tsc soubor.ts
```

Tento příkaz vytvoří soubor.js, který obsahuje přeložený JavaScript kód. Pro zavolání je nutné mít stažený Node.js a TypeScript balíček, který lze nainstalovat pomocí příkazu `npm install -g typescript`. Globální instalace umožňuje používat příkaz `tsc`[17] z jakéhokoliv místa v systému.

Pro větší projekty je vhodné vytvořit konfigurační soubor `tsconfig.json`, který určuje různé možnosti kompilace, jako jsou cílová verze JavaScriptu, zahrnuté soubory a další nastavení.

Nejdůležitější možnosti nastavení pro nás jsou:

- `target`: Určuje verzi JavaScriptu, do které bude TypeScript přeložen (např. ES5, ES6/ES2015, ES2020), což ovlivňuje kompatibilitu s různými prohlížeči. Obecně chceme cílet spíše na novější verze, protože starší prohlížeče už dneska skoro nikdo nepoužívá,
- `module`: Určuje formát modulů, který bude použit v přeloženém JavaScriptu (např. CommonJS, ES6 modules). Pro webové aplikace je běžné používat ES6 modules, viz Kapitola 4.5.1,
- `outDir`: Určuje výstupní adresář pro přeložené soubory JavaScriptu,
- `rootDir`: Určuje kořenový adresář pro zdrojové soubory TypeScriptu,
- A dále spousta dalších možností, které mohou být užitečné v závislosti na konkrétním projektu.

Více možností a podrobností najdete v oficiální dokumentaci TypeScriptu na adrese stránkách jazykové specifikace.

[17] Pokud vám nepůjde tento příkaz, zkuste zavolat `npx tsc`

Při použití `tsc` bez dalších parametrů se kompilátor pokusí najít `tsconfig.json` v aktuálním adresáři a použít jeho nastavení pro kompilaci.

Pokud nechceme stále při vývoji pouštět `tsc` ručně, můžeme použít příkaz `tsc --watch`, který bude sledovat změny v souborech a automaticky je překompiluje.

Pro vytvoření základního `tsconfig.json` můžeme použít příkaz `tsc --init`, který vygeneruje výchozí konfigurační soubor, který můžeme upravit podle našich potřeb.

Pokud v nastavení neurčíme `outDir`, budou přeložené soubory vytvořeny ve stejném adresáři jako zdrojové soubory, což může být nepřehledné. Proto je dobré vždy určit `outDir`, například na `dist` nebo `build`, aby byly přeložené soubory odděleny od zdrojových. Poté pouštíme jako obyčejný JavaScript soubor z tohoto adresáře, například `node dist/soubor.js`.

5.2 Typování

Typ je vlastnost, která určuje, jaký druh hodnoty může být přiřazen k proměnné nebo jaký druh hodnoty může být vrácen z funkce. Typy nám pomáhají zajistit, že kód je správně strukturovaný a že proměnné a funkce jsou používány způsobem, který odpovídá jejich určení.

V TS (a jiných jazycích) existují typy primitivní, složené a uživatelsky definované. Mezi primitivní typy patří například:

- `number`: Čísla (celá i desetinná),
- `string`: Textové řetězce,
- `boolean`: Pravdivostní hodnoty (`true` nebo `false`),
- `null` a `undefined`: Speciální hodnoty reprezentující „žádnou hodnotu“ nebo „hodnotu, která nebyla definována“,
- `symbol`: Unikátní identifikátory,
- `bigint`: Velká celá čísla (větší než `number` může reprezentovat).

Mezi složené typy patří například:

- `array`: Pole hodnot stejného typu,
- `tuple`: Pole hodnot různých typů s pevně danou délkou,
- `object`: Objekty s vlastnostmi a metodami,
- `function`: Funkce s definovanými vstupními a výstupními typy.

Mezi uživatelsky definované typy patří například:

- `enum`: Výčtové typy s předdefinovanými hodnotami,
- `interface`: Rozhraní definující strukturu objektů,
- `type alias`: Alias pro existující typ nebo kombinaci typů,
- `class`: Třídy definující objekty s vlastnostmi a metodami.

Některé typy existují pouze v době vývoje a při překladu jsou odstraněny, například `interface` a `type alias`. Některé poté existují i v přeloženém JavaScriptu, například `enum` či `class`.

Typy můžeme přiřadit k proměnným, funkcím a dalším konstrukcím pomocí anotací. Například:

typescript

Příklad anotací typů

```typescript
let cislo: number = 42;
let text: string = "Ahoj";
let pravda: boolean = true;
function secti(a: number, b: number): number {
```

```
5     return a + b;
6 }
```

Pokud typ přiřadíme, TypeScript zkontroluje, zda jsou hodnoty přiřazené k proměnným nebo vrácené z funkcí kompatibilní s deklarovanými typy. Pokud nejsou, kompilátor vygeneruje chybu.

Mimo přímo definované typy je možné v staticky typovaných jazycích používat implicitní typování, kdy kompilátor odvodí typ na základě přiřazené hodnoty. Například:

typescript

Příklad implicitního typování

```
1 let cislo = 42; // TS říká, že cislo je typu number
2 let text = "Ahoj"; // TS říká, že text je typu string
3 let pravda = true; // TS říká, že pravda je typu boolean
```

Implicitní typování ale může vést k budoucím problémům, protože pokud se změní přiřazovaná hodnota, může se změnit i typ proměnné a tím se problém bude propagovat dále v kódu. Proto je dobré používat explicitní anotace typů, zejména u veřejných API, funkcí a složitějších struktur.

TS si přidává taktéž následující speciální typy:

- `any`: Speciální typ, který může reprezentovat jakýkoliv typ hodnoty. Použití `any` vypíná typovou kontrolu pro danou proměnnou, což může být užitečné při práci s dynamickými daty nebo při migraci z JavaScriptu na TypeScript. Nicméně nadměrné používání `any` může vést k chybám, které by jinak byly zachyceny kompilátorem,
- `unknown`: Podobný jako `any`, ale bezpečnější, protože vyžaduje, aby byla provedena kontrola typu před použitím hodnoty. To znamená, že před přiřazením nebo voláním metod na proměnné typu `unknown` musíme nejprve ověřit její skutečný typ,
- `void`: Speciální typ, který se používá jako návratový typ funkcí, které nevracejí žádnou hodnotu. V praxi to znamená, že funkce s návratovým typem `void` může buď nic nevracet, nebo vracet `undefined`,
- `never`: Speciální typ, který reprezentuje hodnoty, které nikdy nenastanou. Používá se například jako návratový typ funkcí, které vždy vyhodí výjimku nebo nekonečně běží (např. funkce s nekonečnou smyčkou). Proměnná typu `never` nemůže nikdy obsahovat žádnou hodnotu.

5.2.1 Definice vlastních typů

Kromě vestavěných typů můžeme v TypeScriptu definovat i vlastní typy pomocí `interface`, `type alias`, `enum` a dalších.

Začneme s `type`, který slouží na definování nového typu jako alias pro existující typ nebo kombinaci typů. V TS typy kombinujeme pomocí operátorů `|` (nebo) a `&` (a).

typescript

Příklad type aliasů

```
1 type ID = number | string; // ID může být číslo nebo řetězec
2 type Point = { id: ID, x: number; y: number }; // Point je objekt s
  vlastnostmi id, x a y
3 type ColoredPoint = Point & { color: string }; // ColoredPoint je Point s
  přidanou vlastností color
```

Dále máme `interface`, který slouží k definování struktury objektů.

typescript

Příklad interface

```typescript
interface Person {
    name: string;
    age: number;
    greet(): void;
}
```

V ukázce výše můžeme vidět, že `Person` definuje objekt s vlastnostmi `name` a `age` a metodou `greet`. Jasně definuje i příjímací typy a návratový typ metody (v tomto případě `void`, což znamená, že metoda nic nevrací a taktéž nic nepřijímá).

U definice interface, objektů a tříd (viz později) můžeme definovat i volitelné vlastnosti pomocí `?`. Taková proměnná se poté rovná zápisu `<puvodní_typ> | undefined`[18].

typescript

Příklad volitelné vlastnosti

```typescript
interface Project {
    name: string;
    description?: string;
}
```

Tyto typy pak můžeme použít k anotaci proměnných, funkcí a dalších konstrukcí.

typescript

Příklad použití interface

```typescript
let osoba: Person = {
    name: "Jan",
    age: 30,
    greet() {
        console.log(`Ahoj, jmenuji se ${this.name} a je mi ${this.age} let.`);
    }
};
```

Pokud by v ukázce výše neseděla struktura objektu `osoba` s definicí `Person`, TypeScript by vygeneroval chybu.

Další možností je `enum`, který slouží k definování výčtových typů s předdefinovanými hodnotami.

typescript

Příklad enum

```typescript
enum Direction {
    Up,
    Down,
    Left,
    Right
}

```

[18] Tohle třeba ale neplatí ve funkcích!

```
8 let smer: Direction = Direction.Up;
```

Enum se interně překládá na objekt s číselnými hodnotami, kde první hodnota začíná na 0 a každá další se zvyšuje o 1. Každé hodnotě ale můžeme přiřadit i vlastní číselnou nebo řetězcovou hodnotu. Výčtové typy se používají například pro reprezentaci stavů, směrů nebo jiných předdefinovaných sad hodnot. Zpřehledňují kód a zabraňují použití neplatných hodnot.

Dále můžeme definovat třídy pomocí `class`, které kombinují vlastnosti a metody do jedné struktury. Je to stejné jako v JS, jen udáváme typy – vlastnostem, parametrům a návratovým hodnotám metod.

typescript

Příklad class

```
1 class Animal {
2     name: string;
3     constructor(name: string) {
4         this.name = name;
5     }
6     speak(): void {
7         console.log(`${this.name} dělá zvuk.`);
8     }
9 }
```

Interface a class se liší tím, že interface je pouze typová definice a neexistuje v přeloženém JavaScriptu, zatímco class je skutečná konstrukce, která existuje i v přeloženém kódu. Se třídami se dá pracovat v TS mnohem více a ukážeme si to později, viz Kapitola 5.3.

5.2.2 Pomocné typy

Pro práci s typy TypeScript nabízí několik vestavěných pomocných typů, které nám usnadňují práci s typy a strukturami. Mezi nejčastěji používané patří:

- `Partial<T>`: Vytvoří nový typ, kde všechny vlastnosti typu `T` jsou volitelné,
- `Required<T>`: Vytvoří nový typ, kde všechny vlastnosti typu `T` jsou povinné,
- `Readonly<T>`: Vytvoří nový typ, kde všechny vlastnosti typu `T` jsou pouze pro čtení (nemohou být měněny),
- `Record<K, T>`: Vytvoří typ objektu s klíči typu `K` a hodnotami typu `T`,
- `Pick<T, K>`: Vytvoří nový typ výběrem vlastností `K` z typu `T`,
- `Omit<T, K>`: Vytvoří nový typ odstraněním vlastností `K` z typu `T`,
- `ReturnType<T>`: Získá návratový typ funkce `T`,
- `InstanceType<T>`: Získá typ instance třídy `T`.

Více pomocných typů a jejich použití najdete v oficiální dokumentaci TypeScriptu na adrese Utility Types. Tyto pomocné typy nám umožňují lépe pracovat s typy a strukturami v našem kódu, což vede k robustnějším a lépe udržovatelným aplikacím.

ts

Příklad použití pomocného typu Omit

```
1 interface User {
2     id: number;
3     name: string;
4     email: string;
5 }
```

```
6 type NamelessUser = Omit<User, "name" | "email">; // { id: number }
```

5.2.3 Propojení souborů

Stejně jako v JS zejména v TS používáme ES modules pro rozdělení kódu do více souborů a jeho organizaci. Importování je tu trošku jiné než v JS, protože musíme správně uvádět přípony souborů a cesty. Také musíme mít správně nastavený `tsconfig.json`, aby kompilátor věděl, jak s moduly pracovat.

V moderním TS se k souborům neuvádí žádná přípona při importování.

typescript

Příklad importu bez přípony

```
1 import { User } from "./models/User";
```

TS přidává taktéž možnost importovat pouze typy pomocí klíčového slova `type`. To zajistí, že importovaný typ nebude zahrnut do přeloženého JavaScriptu, což může být užitečné pro optimalizaci velikosti kódu.

typescript

Příklad importu typu

```
1 import type { User } from "./models/User";
```

5.2.4 Typování funkcí

Jak se typuje základní funkce jsme si ukázali už výše. TS nám ale nutí typovat i parametry, a tam je to již trochu složitější.

typescript

Příklad dvou funkcí s různým typováním

```
1 const a = (x: number, y?: number): number => {
2     return x + (y ?? 0);
3 }
4
5 const b = (x: number, y: number | undefined): number => {
6     return x + (y ?? 0);
7 }
```

V ukázce výše vidíme dvě funkce, které dělají to samé, ale mají jinak typovaný druhý parametr. V první funkci `y` je volitelný parametr, což znamená, že může být buď číslo, nebo `undefined`, pokud není předán. Ve druhé funkci je `y` povinný parametr, který může být buď číslo, nebo `undefined`. Rozdíl je v tom, jak se funkce volají:

typescript

Příklad volání funkcí

```
1 a(5); // Platné
2 b(5); // Chyba
3 b(5, undefined); // Platné
```

Funkce můžeme používat i jako typy.

typescript

Příklad použití funkce jako typu

```
type Operation = (x: number, y: number) => number;
const add: Operation = (x, y) => x + y;
const multiply: Operation = (x, y) => x * y;
```

5.2.5 Kontrola typů a přetypování

TypeScript provádí statickou kontrolu typů během kompilace, což znamená, že zkontroluje, zda jsou všechny přiřazení a operace s hodnotami kompatibilní s jejich deklarovanými typy. Pokud nejsou, kompilátor vygeneruje chybu a upozorní nás na problém. Někdy se ale může stát, že potřebujeme přetypovat hodnotu z jednoho typu na jiný. To můžeme udělat pomocí tzv. „type assertions“, což je způsob, jak říct kompilátoru, že víme lépe než on, jaký typ má daná hodnota.

typescript

Příklad přetypování pomocí 'as'

```
let someValue: unknown = "Toto je řetězec";
let strLength: number = (someValue as string).length;
```

V ukázce výše máme proměnnou `someValue` typu `unknown`, což znamená, že nevíme, jaký typ má. Použitím `as string` říkáme kompilátoru, že víme, že `someValue` je ve skutečnosti řetězec, a můžeme tedy bezpečně získat jeho délku.

Podobně tedy kontrolujeme typy za běhu pomocí `typeof` nebo `instanceof`.

typescript

Příklad kontroly typů za běhu

```
function printValue(value: number | string) {
    if (typeof value === "string") {
        console.log("Řetězec: " + value);
    } else {
        console.log("Číslo: " + value);
    }
}
```

5.3 Objektově orientované programování

Mimo základní třídy a dědičnost, které jsou stejné jako v JavaScriptu, TypeScript přidává několik dalších funkcí pro podporu objektově orientovaného programování (OOP).

5.3.1 Modifikátory

Mimo klasické modifikátory z JS (veřejné vs. soukromé a statické) TypeScript přidává i další modifikátory pro vlastnosti a metody tříd:

- `public`: Výchozí modifikátor, znamená, že vlastnost nebo metoda je přístupná odkudkoliv,
- `private`: Znamená, že vlastnost nebo metoda je přístupná pouze uvnitř třídy,

- `protected`: Znamená, že vlastnost nebo metoda je přístupná uvnitř třídy a jejích podtříd,
- `readonly`: Znamená, že vlastnost může být nastavena pouze při deklaraci nebo v konstruktoru třídy a nemůže být změněna později.

Tyto modifikátory používáme uvnitř tříd v rámci jejich definice.

typescript

Příklad použití modifikátorů

```typescript
class Person {
    private name: string;
    protected age: number;
    public readonly id: number;

    constructor(name: string, age: number, id: number) {
        this.name = name;
        this.age = age;
        this.id = id;
    }

    public greet(): void {
        console.log(`Ahoj, jmenuji se ${this.name} a je mi ${this.age} let.`);
    }
}
```

5.3.2 Rozhraní a třídy

TypeScript umožňuje třídám implementovat rozhraní pomocí klíčového slova `implements`. To znamená, že třída musí implementovat všechny vlastnosti a metody definované v rozhraní. Zároveň třída umožňuje implementaci více rozhraní najednou.

typescript

Příklad implementace rozhraní třídou

```typescript
interface Drivable {
    drive(): void;
}
interface Flyable {
    fly(): void;
}
class FlyingCar implements Drivable, Flyable {
    drive(): void {
        console.log("Jedu po silnici.");
    }
    fly(): void {
        console.log("Letím vzduchem.");
    }
}
```

5.3.3 Abstraktní třídy

Abstraktní třídy jsou třídy, které nemohou být přímo instanciovány. Slouží jako základ pro jiné třídy a mohou obsahovat abstraktní metody, které musí být implementovány v odvozených třídách. Abstraktní metoda je metoda, která je deklarována, ale nemá žádnou implementaci v abstraktní třídě.

typescript

Příklad abstraktní třídy

```typescript
abstract class Shape {
    abstract area(): number;
    describe(): void {
        console.log(`Toto je tvar s plochou: ${this.area()}`);
    }
}
```

Abstraktní třídu `Shape` nemůžeme přímo instanciovat pomocí konstruktoru a operátoru `new`. Můžeme ji ale používat jako typ pro proměnné a parametry funkcí. To vede k jejímu nejdůležitějšímu použití – polymorfismu.

Polymorfismus (často pojmenováno jako abstrakce) je schopnost objektů různých tříd reagovat na stejnou metodu různými způsoby. Často to používáme v programech pro jednoduché zpracování různých typů objektů stejným způsobem.

> 📚 **Příklad polymorfismu**
>
> Představme si například hru typu Tower Defense[19]. V takové hře můžeme mít různé typy věží, například střeleckou věž, raketomet nebo laserovou věž. Každá věž může mít metodu `attack()`, která provádí útok na nepřátele.
>
> Místo toho, abychom psali speciální kód pro každý typ věže, můžeme vytvořit abstraktní třídu `Tower` s abstraktní metodou `attack()`. Pak můžeme jednoduše vytvořit různé třídy věží, které dědí z `Tower` a implementují svou vlastní verzi metody `attack()`. Když pak budeme chtít provést útok všech věží, můžeme jednoduše iterovat přes pole věží a volat metodu `attack()` na každé z nich, aniž bychom museli vědět, jaký konkrétní typ věže to je.

Abstrakce a polymorfismus právě to z ukázky výše umožňují. Abstrakce nám dovoluje v poli ukládat nadřazený typ, a poté každá implementace (odvozená třída) se postará o to, jak přesně se metoda provede.

Z abstraktní tříd můžeme dědit. V děděné třídě poté musíme implementovat všechny abstraktní metody. Nebo můžeme třídu také označit jako abstraktní, čímž ji opět znemožníme instanciovat a můžeme v ní mít další abstraktní metody. Jednotlivé metody pak můžeme implementovat nebo je zanechat na podděděných třídách.

5.3.4 Generika

Generika je jedním z nejmocnějších nástrojů skoro ve všech staticky typovaných jazycích. Dovolují totiž psát komponenty, které fungují s různými datovými typy, aniž bychom museli

[19] Hry, kde stavíte věže, které brání vlnám nepřátel postupujícím po cestě.

obětovat typovou bezpečnost. Generika nám umožňují definovat třídy, rozhraní a funkce s parametry typů. Tyto parametry typů pak můžeme použít uvnitř definice pro označení typů vlastností, návratových hodnot a parametrů.

typescript

Příklad generické funkce

```typescript
function identity<T>(arg: T): T {
    return arg;
}
```

V ukázce výše máme generickou funkci `identity`, která přijímá jeden parametr `arg` typu `T` a vrací hodnotu stejného typu `T`. Když voláme tuto funkci, můžeme buď explicitně určit typový argument, nebo nechat TypeScript, aby ho odvodil na základě předaného argumentu.

typescript

Příklad volání generické funkce

```typescript
let output1 = identity<string>("Ahoj světe"); // Explicitní určení typu
let output2 = identity(42); // Implicitní odvození typu
```

Generika jsou užitečná pro vytváření opakovaně použitelných komponent, které mohou pracovat s různými datovými typy, aniž bychom museli psát více verzí stejné funkce nebo třídy pro každý typ zvlášť.

Například uvnitř třídy můžeme vytvořit generickou třídu `Queue`, která může ukládat prvky libovolného typu.

typescript

Příklad generické třídy

```typescript
class Queue<T> {
    private items: T[] = [];
    enqueue(item: T): void {
        this.items.push(item);
    }
    dequeue(): T | undefined {
        return this.items.shift();
    }
}
```

Na daném typu avšak nemůžeme provádět žádné operace, protože TypeScript neví, jaký typ to je. My můžeme ale omezit typový parametr na určitou množinu typů pomocí tzv. „constraints“.

typescript

Příklad omezení typového parametru

```typescript
function loggingIdentity<T extends { length: number }>(arg: T): T {
    console.log(arg.length);
    return arg;
}
```

V ukázce výše jsme omezili typový parametr `T` na typy, které mají vlastnost `length`. To znamená, že můžeme předat například řetězec nebo pole, ale ne číslo nebo boolean.

5.4 Použití na webu

Pro použití TypeScriptu na webu je běžné použít nástroj pro sestavení (build tool) nebo balíčkovač modulů (module bundler), jako je například Webpack, Rollup nebo Vite. Tyto nástroje nám umožňují automatizovat proces kompilace TypeScriptu do JavaScriptu, spravovat závislosti a optimalizovat výsledný kód pro nasazení na webový server.

Ruční kompilace pomocí `tsc` je možná, ale v praxi se téměř nepoužívá, protože je to nepraktické pro větší projekty s mnoha soubory a závislostmi. Pro nastavení projektu s TypeScriptem na webu obvykle postupujeme následovně:

1. Inicializujeme nový projekt pomocí `npm init` a nainstalujeme TypeScript a další potřebné balíčky jako vývojové závislosti.
2. Vytvoříme `tsconfig.json` pro konfiguraci TypeScript kompilátoru.
3. Nastavíme nástroj pro sestavení nebo balíčkovač modulů, který bude používat TypeScript kompilátor jako součást svého procesu sestavení.
4. Vytvoříme strukturu projektu s adresáři pro zdrojové soubory TypeScriptu, statické soubory a výstupní adresář pro přeložený JavaScript.
5. Spustíme nástroj pro sestavení nebo balíčkovač modulů, který přeloží TypeScript kód do JavaScriptu a vytvoří výsledný balíček pro nasazení na webový server.

My budeme v tomto předmětu[20] používat Vite, který je moderní a rychlý nástroj pro vývoj webových aplikací s podporou TypeScriptu. Nyní si ho ukážeme pouze stručně. Vite je jednoduchý na nastavení a nabízí rychlý vývojový server s podporou Hot Module Replacement pro rychlé iterace během vývoje.

Pro založení Vite projektu s TypeScriptem můžeme použít následující příkaz:

bash

Příklad příkazu pro vytvoření Vite projektu

```bash
npm create vite@latest
```

Poté se nás interaktivní průvodce zeptá na název projektu a typ šablony, kde vybereme `vanilla` a vyberem TypeScript jako jazyk. To vytvoří tzv. scaffold projektu s předpřipravenou strukturou a konfigurací pro použití TypeScriptu s Vite. Po vytvoření projektu můžeme nainstalovat závislosti pomocí `npm install` a spustit vývojový server pomocí `npm run dev`. To nám zapne stránku na konkrétní adrese (obvykle `http://localhost:5173`), kde můžeme vidět naši aplikaci běžící s podporou TypeScriptu.

Celá Vite aplikace ale již počítá s tím, že se jedná o Single Page Application (SPA), což znamená, že veškerá navigace a interakce probíhá na jediné HTML stránce pomocí JavaScriptu. Pokud chcete Multi Page Application (MPA), kde každá stránka je samostatný HTML dokument, je potřeba Vite nakonfigurovat jinak nebo použít jiný nástroj. Je to ale poměrně složité téma, které přesahuje rámec tohoto úvodu do TypeScriptu.

Ve vytvořeném projektu pak píšeme TS kód do `.ts` souborů uvnitř `src` adresáře.

Uvnitř projektu si všimneme dvou nejdůležitějších souborů pro TypeScript:

- `index.html`: Hlavní HTML soubor, který načítá náš TypeScript kód (přeložený do JavaScriptu) a slouží jako vstupní bod pro naši aplikaci,
- `main.ts`: Hlavní TypeScript soubor, kde začíná náš kód a odkud můžeme importovat další moduly a komponenty.

[20] Zejména až se dostaneme k Vue.

Pro nasazení aplikace na webový server použijeme příkaz `npm run build`, který vytvoří optimalizovaný balíček v `dist` adresáři, připravený k nasazení. Stačí pak zkopírovat obsah `dist` na náš webový server a aplikace bude dostupná online.

Design

6.1 Dekompozice webových stránek

Webové stránky se skládají z různých částí, které spolu tvoří funkční celek. Nejčastěji rozdělujeme stránky na dvě důležité části a to frontend a backend.

Frontend je část, kterou vidí a se kterou interaguje uživatel. Na této části webu se zaměřujeme na její vzhled a použitelnost. Pokud nebude frontend dobře navržený, uživatelé budou mít problém se na stránce orientovat a mohou ji opustit. Taktéž se může stát, že frontend bude příliš pomalý, což uživatele odradí od další návštěvy.

Backend je část, která zajišťuje funkčnost webu. Zde se staráme o správu dat, komunikaci s databází a zpracování požadavků od uživatelů. I když uživatel backend nevidí, je klíčový pro správné fungování webové stránky. Bez dobře navrženého backendu by webová stránka nemohla správně fungovat. Může se taktéž stát, že backend bude příliš pomalý nebo nespolehlivý, což negativně ovlivní uživatelský zážitek.

Obecně chceme, aby veškeré části webové stránky byly dobře navržené a optimalizované. Pokud jedna z částí selže, může to mít negativní dopad na celkový dojem z webu.

V této celé kapitole se budeme zaobírat především návrhem frontendu, jelikož je to část, kterou uživatelé vidí a se kterou interagují. Často se ale setkáme s pojmem fullstack vývojář, což je vývojář, který se zabývá jak frontendem, tak backendem.

6.1.1 Uživatelské rozhraní

Uživatelské rozhraní (UI) je součástí webové stránky, se kterou uživatel interaguje. Zahrnuje všechny prvky, které uživatel vidí a používá, jako jsou tlačítka, formuláře, menu a další ovládací prvky. Cílem dobrého uživatelského rozhraní je usnadnit uživatelům interakci s webovou stránkou a zajistit, aby byla intuitivní a snadno ovladatelná.

Mimo samotných prvků uživatelského rozhraní je důležitá i jejich vizuální stránka. Ta zahrnuje barvy, písma, rozvržení a další designové prvky, které ovlivňují celkový dojem z webové stránky.

6.1.2 Uživatelská zkušenost

Uživatelská zkušenost (UX) či uživatelský zážitek se zaměřuje na celkový dojem a spokojenost uživatele při používání webové stránky. Nejde jen o samotné uživatelské rozhraní, ale i o to, jak snadno a efektivně může uživatel dosáhnout svých cílů na webové stránce. To zahrnuje faktory jako rychlost načítání stránek, navigace, obsah a celková použitelnost.

Často je nutné přemýšlet o tom, jak různí uživatelé budou webovou stránku používat. Vžít se tedy do role uživatele a navrhnout webovou stránku tak, aby byla co nejpřívětivější pro co nejširší spektrum uživatelů.

6.2 Zásady dobrého designu

Dobrý design by měl řešit obě strany – jak uživatelské rozhraní tak i uživatelskou zkušenost. Je důležité, aby byly obě části navrženy s ohledem na potřeby a očekávání uživatelů.

Existuje mnoho zásad a pravidel, které mohou pomoci při navrhování dobrého designu. Níže jsou uvedeny některé z nejdůležitějších. Obecně neplatí, že každá stránka musí dodržovat všechny tyto zásady, ale je dobré je mít na paměti při návrhu webové stránky.

Můžete totiž mít dobrý důvod pro to, proč některé zásady nedodržet. Například pokud máte specifické požadavky od klienta nebo pokud pracujete na projektu s omezeným rozpočtem a časem. Ale může se jednat i o kreativní rozhodnutí, kdy chcete vytvořit něco unikátního a odlišného od běžných standardů[21].

Vy si jako vývojáři a designéři může vytvořit i zcela vlastní nové zásady, které budou lépe vyhovovat vašim potřebám. Ale tam je nutné si ověřit, jestli to opravdu funguje a je to užitečné pro uživatele.

Pokud nevíte, jestli něco funguje, je nutné se řídit instinktem a případně si to otestovat na reálných uživatelích.

6.2.1 Konzistence

Zásada konzistence říká, že by měly být podobné prvky navrženy podobně. To znamená, že stejné typy tlačítek by měly mít stejný vzhled a chování na celé webové stránce.

Tady je důležité si uvědomit, že konzistence neznamená, že každé tlačítko musí vypadat úplně stejně. Říká to, že když už jednou uživatele seznámíte s určitým tlačítkem s konkrétní akci – například odeslání formuláře – tak by měl toto tlačítko vypadat stejně na všech místech, kde jsou formuláře k odeslání.

Pokud porušíte konzistenci, může to vést k zmatku a frustraci uživatelů, protože si přestanou být jistí, co daný prvek vlastně dělá. To povede k tomu, že uživatel bude experimentovat, méně si jistý a nakonec může stránku opustit.

6.2.2 Hierarchie

Hierarchie je důležitá pro to, aby uživatelé mohli snadno najít to, co hledají. To znamená, že nejdůležitější prvky by měly být nejvíce viditelné a snadno dostupné. Toho lze dosáhnout pomocí velikosti, barvy, umístění a dalších vizuálních prvků.

Například nadpisy by měly být větší a výraznější než běžný text, aby uživatelé mohli rychle pochopit strukturu obsahu.

Pokud je hierarchie špatně navržena, uživatelé mohou mít problém najít důležité informace nebo funkce, což povede k frustraci a odchodu ze stránky.

6.2.3 Standard a konvence

Při navrhování webových stránek je důležité dodržovat standardy a konvence, které jsou běžně používané na webu. To znamená, že byste měli používat běžné prvky a vzory, které uživatelé očekávají. Například logo webové stránky je obvykle umístěno v levém horním rohu a slouží jako odkaz na domovskou stránku. Navigační menu je často umístěno v horní části stránky nebo na levé straně. Podobně to funguje například s ikonami – například ikona lupy obvykle značí vyhledávání, ikona obálky značí e-mail a tak dále.

[21] Pochopíte později…

Dodržováním těchto standardů a konvencí usnadníte uživatelům orientaci na vaší webové stránce, protože budou vědět, kde hledat určité prvky a jak s nimi interagovat. To se hodí zejména pro uživatele s menší zkušeností, kteří nemusí být obeznámeni s vaším konkrétním designem.

Pokud porušíte tyto standardy a konvence, může to vést k zmatku a frustraci uživatelů, protože nebudou vědět, jak dané prvky fungují nebo kde je najít. To není nutně ale špatné – někdy může být užitečné tyto konvence porušit, pokud máte dobrý důvod, například chcete vytvořit něco unikátního nebo inovativního. Pro nové věci je nutné experimentovat.

6.2.4 Přívětivost

Přívětivost se zaměřuje na to, aby uživatelé měli pozitivní zkušenost při používání webové stránky. Nejdůležitější je, aby stránka říkala, co uživatel dělá špatně, a nebo ji opravit za něj. To zahrnuje například jasné chybové zprávy, které uživateli vysvětlí, co se stalo a jak to může opravit.

Nestačí jen označit chybu např. červeným rámečkem kolem pole formuláře. Je nutné uživateli říct, co je špatně a jak to může opravit.

6.2.5 Stručnost

Rozhraní stránky by mělo být na první pohled jednoduché a přehledné. Uživatel by ihned neměl vidět příliš mnoho informací a možností najednou.

Toho lze dosáhnout tím, že se zaměříte na nejdůležitější prvky a funkce a skryjete ty méně důležité. Skrytí neznamená, že funkce aplikace nebude mít, jen, že budou dostupné například v menu nebo na jiné stránce.

Příliš mnoho informací najednou může uživatele zahlcovat a vést k frustraci.

6.2.6 Zpětná vazba

Rozhraní říká, že uživatelé by měli dostávat okamžitou zpětnou vazbu na své akce. To znamená i to, že když aplikace něco dělá, tak by to uživatel měl vědět. Například když uživatel odešle formulář, měl by vidět potvrzení, že formulář byl úspěšně odeslán. Ale i to, že se formulář odesílá – například tím, že uvidí načítací ikonu nebo zprávu „Odesílání…“.

6.2.7 Příjemnost

Aplikace a rozhraní by mělo být navrženo tak, že uživatel nemusí dělat komplikované akce pro dosažení svých klasických cílů. To znamená, že by mělo být snadné najít a použít nejdůležitější funkce aplikace.

Jako ukázku lze ukázat seznam kontaktů v telefonu. Když chci napsat SMS několik z nich, tak by mělo být snadné je vybrat z kontaktů – například více kliknutím na jednotlivé kontakty – místo toho, abych SMS psal opravdu jednotlivým kontaktům zvlášť.

Čas strávený na nejdůležitějších stránkách by měl být co nejpříjemnější a nejrychlejší.

6.2.8 Očekávatelnost

Rozhraní by mělo být rozloženo tak, že je na první pohled jasné, jak je stránka hierarchicky uspořádána. Dále by měl být jasný tok stránky – tedy jaké kroky je nutné udělat pro dosažení určitého cíle. Toho lze dosáhnout pomocí vizuálních prvků, jako jsou šipky, čísla kroků, nebo barevné odlišení.

S očekávatelností souvisí i princip minimálního překvapení. Princip říká dvě důležité věci:

- Uživatelé jsou schopni sledovat pouze jednu věc najednou, a té říkáme fokus,
- Při používání aplikace by mělo být uživateli jasné, co se stane, když něco udělá.

Princip minimálního překvapení zasahuje i do jiných různých zásad. Před tvorbou aplikace by jsme si měli zjistit, kdo nejspíše aplikaci bude používat, jaké může mít zkušenosti, co očekává od aplikace a s jakým přístupem přistupuje k aplikaci.

Nejčastějším problémem webových stránek jsou různé popup okna, které uživatel nemá šanci očekávat a uživatele to demotivuje aplikaci dále používat. Další problémy jsou různé tlačítka s názvy, od kterým uživatel čeká kompletně něco jiného.

Problémům mezi aplikací a uživatelem se nelze vyhnout, ale je důležité je co nejvíce minimalizovat. Do aplikace může vkročit uživatel, se kterým se nepočítalo, nebo novým updatem se přidá něco, co tyto zásady poruší.

6.2.9 Hodnota webu

Hodnota webové stránky je důležitá pro to, aby uživatelé měli důvod stránku navštěvovat a používat. To znamená, že webová stránka by měla nabízet něco, co uživatelé potřebují nebo chtějí. To může být užitečný obsah, služby, produkty nebo zábava.

Podle modelu UX honey comb existuje šest hlavních aspektů, které přispívají k hodnotě webové stránky:

- **Užitečnost**: Webová stránka by měla být užitečná a splnit potřeby uživatelů,
- **Použitelnost**: Webová stránka by měla být snadno použitelná a intuitivní,
- **Důvěryhodnost**: Webová stránka by měla být důvěryhodná a spolehlivá,
- **Přístupnost**: Webová stránka by měla být přístupná pro všechny uživatele, včetně těch s různými schopnostmi a hendikepy,
- **Žádoucnost**: Webová stránka by měla být vizuálně atraktivní a příjemná pro uživatele,
- **Dohledatelnost**: Webová stránka by měla být snadno nalezitelná prostřednictvím vyhledávačů a dalších kanálů.

Tento model nám říká, že pokud stránka nesplňuje některý z těchto aspektů, může to negativně ovlivnit její hodnotu pro uživatele.

6.2.10 Časté problémy při designu

V teorii je možné aplikovat a řešit skoro každý problém, na který narazíme při návrhu webové stránky. V praxi ale narazíme na problém toho, že uživatelé mají různé potřeby, očekávání a zkušenosti.

Často se stává, že navrhujeme určitý způsob například toho, jak se konkrétní akce na stránce provádí. Uživatelé si však mohou najít jiný způsob, jak danou akci provést, který jsme nečekali. To povede k tomu, že musíte hlídat různé způsoby použití a podporovat jich případně co nejvíce.

Pro nás je tedy důležité případné alternativní způsoby použití sledovat a analyzovat. Čímž se budeme zabývat v další kapitole.

6.3 Analýza uživatelů

Při tvorbě webových aplikací často potřebujeme sledovat uživatele – ale ne v tom špatném slova smyslu. Jednoduše potřebujeme vědět, jak se uživatel na stránce chová a nebo na jaké stránky vůbec chodí. Nejčastější sledování je sledování požadavku, kde z různých trendů a grafů můžeme vidět, jak uživatele přicházejí a odcházejí. Pokročilejší sledování cílí na celé monitorování stránky při aktivitě uživatele.

Důležité je, si uvědomit, že sledování požadavků je zcela normální a dělá ho v podstatě každá stránka a to i bez Vašeho souhlasu. Sledování pokročilé, kde sledujeme např. i kam uživatel kliká, musí uživatel vědět a v moderní době je nutné mu povolit to vypnout.

Sledování může však přicházet i na jiné úrovni, a to, že zaplatíte lidem, aby Vaší stránku použili a buď je budete sledovat u Vás v kanceláři (nebo mimo ni) a nebo např. procházení nahraje. Jedná se o tzv. testery.

6.3.1 Sledování požadavků

Jedná se o nejjednodušší způsob monitoringu a je (jak jsem již zmínil) používán v podstatě všude. Nenarušujeme práva uživatelů, protože nesledujeme to, co uživatel dělá, ale jaké požadavky na server (resp. na web) zasílá pomocí HTTP protokolu. Sledujeme tedy např. cestu, kterou navštěvuje, jeho IP (můžeme vědět geolokaci, zemi, ...) a další. Z tohoto sledování můžeme zjistit nejnavštěvovanější stránku, země uživatelů, počet crawlerů a další. Implementace je pomocí jakéhokoliv web serveru či např. pomocí Cloudflare Analytics.

6.3.2 Sledování uživatelů

Sledování uživatelů je více agresivní sledování, které bere informace o přechodu stránek (jak dlouho byl na stránce, kdy odešel, ...), informace o uživatelovi, typ prohlížeče, zdroj přístupu, různé interakce a další.

O tomto sledování je nutné informovat uživatele a dát mu možnost funkcionalitu vypnout. Nejčastěji se používá Google Analytics. Do sledování uživatelů patří taktéž sledování kurzoru, které se v praxi dělá kvůli soukromí méně často, ale při velkém množství záznamů můžeme vytvořit tzv. heatmapu, které ukazuje např. nejčastější pozici myši, nejčastější místo kliku a další.

6.4 Psychologie designu

Při tvorbě webových stránek je důležité si uvědomit, že design ovlivňuje chování uživatele – a to jak přímo, tak nepřímo. Nepřímý vliv nastává tehdy, když uživatele ovlivňujeme neúmyslně. Přímý vliv je naopak cílený – záměrně navrhujeme design tak, aby vedl uživatele k určitému jednání.

Design může působit pozitivně i negativně a může vyvolávat příjemný i nepříjemný dojem. Bohužel se často setkáváme s nekalými praktikami, kdy jsou uživatelé ovlivňováni příliš a

neeticky. Mezi nástroje ovlivňování patří animace, barvy, fotografie, umístění klikacích prvků a mnoho dalšího.

Ovlivňovat můžeme prostřednictvím UI ale i UX. Tato problematika se netýká pouze webových stránek, ale také mobilních a desktopových aplikací, tisku a dalších marketingových materiálů.

Důležité je si uvědomit, že není možné neovlivňovat. Každý design působí na každého člověka jinak – podle jeho zkušeností, kultury a osobních preferencí. Proto je nutné při tvorbě designu přemýšlet o tom, jak může působit na různé skupiny uživatelů.

6.4.1 Barvy

Barvy mají na uživatele velmi silný vliv, protože každá barva vzbuzuje různé emoce a asociace. Každý člověk vnímá barvy odlišně – a to jak na základě osobních preferencí, tak kvůli kulturním rozdílům. V některých kulturách například červená symbolizuje štěstí a prosperitu, zatímco v jiných může znamenat nebezpečí nebo varování.

Každou použitou barvu je proto nutné pečlivě rozmyslet a zvážit její dopad na cílové uživatele. Teorie barev nám pomáhá pochopit, jaké barvy k sobě pasují a jak vytvářet harmonické barevné schéma.

Při výběru barev je důležité myslet na následující faktory:

- **Psychologický dopad**: Různé barvy vyvolávají různé emoce (modrá klid, červená energii, zelená přírodní pocit),
- **Kontrast a čitelnost**: Barvy by měly zajistit dostatečný kontrast pro snadné čtení textu,
- **Přístupnost**: Při výběru barev je nutné myslet na uživatele s barvoslepostí,
- **Konzistence**: Barevné schéma by mělo být jednotné v rámci celé webové stránky,
- **Kultura a cílové publikum**: Barvy by měly odpovídat očekáváním a kulturnímu kontextu uživatelů.

6.4.1.1 Barevné modely

6.4.2 Typografie

Typografie neboli nauka o písmu hraje klíčovou roli v designu webových stránek. Správný výběr písma neovlivňuje pouze čitelnost textu, ale také celkový dojem a emocionální působení stránky.

Do typografie je možné zařadit následující aspekty:

- **Výběr písma**: Různé typy písem vyvolávají různé pocity a asociace,
- **Velikost písma**: Velikost písma ovlivňuje čitelnost a hierarchii informací,
- **Řádkování a mezery**: Správné řádkování a mezery mezi písmeny zlepšují čitelnost,
- **Barva písma**: Barva by měla zajistit dostatečný kontrast s pozadím,
- **Konzistence**: Použití jednotného stylu písma napříč celou webovou stránkou.

Mimo to do typografie často řadíme i gramatiku a pravopis, protože špatně napsaný text může negativně ovlivnit dojem z webové stránky.

6.4.2.1 Typy písem

Obecně existuje několik základních typů písem, které se hodí na různé účely a vyvolávají odlišné myšlenky a pocity:

Serif je typ písem, které využívají patky – různé dekorace na koncích čar jednotlivých znaků. Tato písma se hojně používají pro tisk, protože vyvolávají důvěryhodnost, autoritu, formálnost a respekt. Na webových stránkách se používají především pro delší texty nebo když chceme vytvořit dojem tradice a seriózności.

Sans-serif je typ písem, které patky nevyužívají. Tato písma jsou moderní a čistá, proto se nejvíce používají pro online služby a webové aplikace. Jsou snadno čitelná i na obrazovkách a působí současně a přístupně.

Cursive je typ písem, které vypadají jako ručně psané – jednotlivé znaky se často napojují. Vyvolávají pocity elegantnosti, kreativity a tradičnosti. Na webu se používají spíše omezeně, například pro loga nebo nadpisy, kde chceme vytvořit osobitější dojem.

Monospace je typ písem, kde všechny znaky zabírají stejné množství místa. Používají se zejména pro zobrazení programového kódu, protože zlepšují viditelnost syntaxe a zarovnání. Vyvolávají pocity techničnosti, vzdělanosti a úhlednosti.

Při výběru písma je důležité myslet na účel, čitelnost a celkový dojem, který chceme vytvořit. Neměli bychom na jedné stránce používat příliš mnoho různých písem – obvykle stačí jedno až tři pečlivě vybraná písma.

6.4.3 Obrázky a ikony

Obrázky jsou pro webové stránky mimořádně důležité, protože platí známé pravidlo, že jeden obrázek vydá za tisíc slov. Vizuální obsah dokáže předat informace mnohem rychleji a efektivněji než samotný text.

Proto je vhodné dlouhé texty vždy obohacovat příhodnými obrázky – ať už se jedná o ukázky z tématu, ilustrace, diagramy nebo fotografie. Obrázky nám pomáhají:

- **Rychle upoutat pozornost**: Máme jen několik vteřin na to, abychom oslovili návštěvníka stránky,
- **Snadněji vysvětlit**: Komplexní koncepty lze často vysvětlit lépe obrázkem než dlouhým textem,
- **Zpříjemnit čtení**: Obrázky rozbíjejí dlouhé bloky textu a činí stránku vizuálně přitažlivější,
- **Podpořit sdělení**: Správně zvolený obrázek může posílit hlavní myšlenku textu.

Ne každý návštěvník má čas nebo chuť číst paragrafy textu. Pomocí obrázků a stručného textu můžeme oslovit i tyto uživatele a předat jim klíčové informace rychleji.

6.4.4 Bílé místo

Pro správnou tvorbu uživatelského rozhraní je důležité používat tzv. bílé místo (whitespace nebo negative space). Jedná se o prázdný prostor mezi jednotlivými texty a obecně všemi prvky na stránce.

Bílé místo plní několik důležitých funkcí:

- **Oddělení obsahu**: Pomáhá vizuálně oddělit různé sekce a prvky na stránce,
- **Zlepšení čitelnosti**: Dostatečný prostor kolem textu usnadňuje jeho čtení,

- **Snížení kognitivní zátěže**: Méně přeplněná stránka je pro uživatele příjemnější,
- **Zvýraznění důležitých prvků**: Prostor kolem prvku ho činí významnějším.

Místo by mělo být na stránce dostatečně velké, aby se text a jednotlivé sekce lépe rozdělily. Pokud však budou místa příliš velká, mohli bychom uživatele zmást – může to působit dojmem chybějícího obsahu nebo nedokončené stránky.

Je důležité najít správnou rovnováhu mezi přeplněným designem a přílišnou prázdnotou. Obecně platí, že bílé místo by mělo být úměrné důležitosti prvků a mělo by přirozeně vést oko uživatele přes stránku.

Je důležité si uvědomit, že místo na webové stránce je nekonečné (naopak od jiných médií, jako je tisk) a je vhodné ho využít k vytvoření přehledného a příjemného designu.

6.4.5 Další

Mezi další důležité aspekty designu patří:

- **Přístupnost**: Zajištění, že webové stránky jsou použitelné pro všechny uživatele, včetně těch se zdravotním postižením,
- **Testování použitelnosti**: Pravidelné testování webových stránek s reálnými uživateli pro identifikaci problémů a zlepšení uživatelského zážitku,
- **Trendy v designu**: Sledování aktuálních trendů v designu a jejich vhodná aplikace na webové stránky,
- A další.

6.5 Multimédia na webu

Multimédia zahrnují různé formy obsahu, jako jsou obrázky, videa, zvuky a animace. S obrázky a animacemi jsme se již částečně seznámili v předchozích kapitolách a v předmětu WBA.

V této části se zaměříme především na práce s videem a zvukem na webových stránkách. Na webových stránkách definujeme video i zvuk jako sémantické prvky HTML5.

6.5.1 Video

Videa vytváříme v HTML pomocí značky `<video>`. Nejdůležitější atributy jsou:

- `src`: URL adresa videa,
- `width` a `height`: rozměry videa,
- `autoplay`: automatické přehrávání videa po načtení stránky,
- `loop`: opakování videa po jeho skončení,
- `muted`: ztlumení zvuku videa,
- `controls`: zobrazení ovládacích prvků pro přehrávání videa (přehrát, pauza, hlasitost, atd.).

html

Ukázka videa v HTML

```
1 <video src="https://www.example.com/video.mp4" width="640" height="360"
  controls>
2   Váš prohlížeč nepodporuje přehrávání videa.
3 </video>
```

Videa můžeme ovládat i pomocí JS DOM API. Nejdůležitější metody a vlastnosti jsou:

- `play()`: spustí přehrávání videa,
- `pause()`: pozastaví přehrávání videa,
- `currentTime`: nastaví nebo získá aktuální čas přehrávání videa,
- `duration`: získá celkovou délku videa,
- `volume`: nastaví nebo získá hlasitost videa (hodnoty od 0.0 do 1.0),
- `playbackRate`: nastaví nebo získá rychlost přehrávání videa (např. 1.0 je normální rychlost, 0.5 je poloviční rychlost, 2.0 je dvojnásobná rychlost).

6.5.2 Zvuk

Zvuk na webových stránkách vytváříme pomocí značky `<audio>`. Nejdůležitější atributy jsou:

- `src`: URL adresa zvukového souboru,
- `autoplay`: automatické přehrávání zvuku po načtení stránky,
- `loop`: opakování zvuku po jeho skončení,
- `muted`: ztlumení zvuku,
- `controls`: zobrazení ovládacích prvků pro přehrávání zvuku (přehrát, pauza, hlasitost, atd.).

Ukázka zvuku v HTML (html)

```html
<audio src="https://www.example.com/audio.mp3" controls>
  Váš prohlížeč nepodporuje přehrávání zvuku.
</audio>
```

Zvuk můžeme ovládat i pomocí JS DOM API.

Nejdůležitější metody a vlastnosti jsou skoro ekvivalentní s videem:

6.5.3 Omezení

Při práci s multimédii na webových stránkách je nutné brát v úvahu několik omezení:

- **Formáty souborů**: Různé prohlížeče podporují různé formáty videa a zvuku. Je důležité zajistit kompatibilitu napříč prohlížeči například tím, že poskytneme více formátů pro jeden soubor,
- **Velikost souborů**: Velké multimediální soubory mohou zpomalit načítání stránky a negativně ovlivnit uživatelský zážitek. Je důležité optimalizovat velikost souborů a používat vhodné kompresní metody,
- **Přístupnost**: Je důležité zajistit, aby multimediální obsah byl přístupný pro všechny uživatele, včetně těch se zdravotním postižením. To zahrnuje například poskytování titulků pro videa a alternativního textu pro zvukové soubory,
- **Autoplay omezení**: Mnoho prohlížečů omezuje automatické přehrávání multimédií, zejména pokud obsahuje zvuk. Je důležité respektovat tato omezení a poskytovat uživatelům kontrolu nad přehráváním,
- **Výkon zařízení**: Některá zařízení mohou mít omezený výkon a nemusí být schopna plynule přehrávat vysokokvalitní multimediální obsah. Je důležité optimalizovat multimédia pro různé typy zařízení a připojení k internetu.

CSS preprocesory

7.1 Moderní CSS

CSS je nejvíce aktualizovaná část webových technologií. HTML je prakticky beze změn již od roku 2017. JavaScript sice každý rok dostává změny od standardu ECMA, ale ty jsou od posledních let spíše menší. Největší změny v JS nastaly v letech 2015-2019, kdy přidali je asynchronní programování, třídy a mnoho dalšího. CSS se naopak mění velmi rychle.

V CSS je totiž zcela možné přidávat nové vlastnosti a funkce, aniž by to mělo zásadní dopad na zpětnou kompatibilitu a tudíž není nutné nad tím tolik přemýšlet jako u jiných jazyků.

Poslední roky se CSS snaží přidat spoustu funkcionalit, o kterých se budeme v této kapitole bavit.

7.1.1 Proměnné

Uvnitř CSS je možné definovat proměnné pomocí vlastností začínajících na `--`. Mohou být jakéhokoliv jména. Používají se tím, že se zavolá funkce `var()`.

Daná hodnota se jako některé vlastnosti předává dědičností. Stejně je můžeme přepsat tím, že je v daném prvku přepíšeme jiným pravidlem.

css

Použití proměnných v CSS

```css
body {
  --border-color: #fafe0e;
}
article.technology {
  color: var(--border-color);
}
```

Při použití `var` můžeme jako druhý parametr předat uvést základní hodnotu, která se použije, pokud žádná proměnná daného jména u daného prvku (či rodičů pomocí předání) neexistuje.

Nejen pro používání proměnných můžeme v CSS použít nový selektor pseudotřídy `:root`. Tato pseudotřída reprezentuje nejvyšší prvek pro daný dokument, což je v případě HTML element `<html>`. V případě např. SVG dokumentu je to kořenový `<svg>` element. Navíce je `root` pseudotřída a tedy se její specifičnost počítá jako `0-1-0`, což je více než u běžného elementu (0-0-1). Nejčastěji je používáme pro definování základní hodnoty pro `rem` a právě proměnné pro celou stránku.

css

Použití selektoru `:root` a základní hodnoty.

```css
:root {
  --logo: "./logo.png";
}
header div.container .logo {
  background-image: url(var(--logo, "/img/logo.png"))
}
```

Ukázka výše **nebude** fungovat, protože funkce `url()` v CSS nepodporuje vnořené funkce. Obdobně nefungují další různá volání (např. použití proměnné jako názvu vlastnosti, použití proměnné v `@media`, `:nth-child()` a další). Správné použití výše by bylo například:

css

Správné použití proměnné v CSS se selektorem `:root` a funkcí `url()`.

```css
:root {
  --logo: url("./logo.png");
}
header div.container .logo {
  background-image: var(--logo, url("/img/logo.png"));
}
```

7.1.2 Vnořování

Psaní selektorů v CSS je náročné, protože píšeme spoustu věcí, které jsou spojeny pod jedním společným selektorem. Proto můžeme použít vnořování, které nám dovolí do selektorů vložit jiné selektory a tím pádem se zkombinují.

css

Vnořování selektorů v CSS

```css
.article {
  border: 1px solid black;

  .title {
    font-size: 2rem;
  }
}
```

Výše uvedená ukázka je ekvivalentní k následujícímu kódu:

css

Ekvivalentní kód bez vnořování

```css
.article {
  border: 1px solid black;
}
.article .title {
  font-size: 2rem;
}
```

V CSS existuje selektor vnořování (&), který reprezentuje explicitně rodičovský selektor. Používáme ho tehdy, když chceme vnořit něco, co není podřízené danému selektoru, ale je s ním nějak spojeno.

css

Použití selektoru vnořování

```css
.button {
  background-color: blue;

  &:hover {
    background-color: red;
  }
}
```

Tento kód je ekvivalentní k následujícímu:

css

Ekvivalentní kód bez vnořování s použitím selektoru vnořování

```
1 .button {
2   background-color: blue;
3 }
4 .button:hover {
5   background-color: red;
6 }
```

Bez operátoru & by totiž vznikl selektor `.button :hover`, což není to, co chceme, protože znamená, že uvnitř `.button` je nějaký prvek, který je v hover stavu.

Ve vnořování lze používat i at-pravidla jako `@media`. To nám dovolí psát media query přímo uvnitř selektoru, na který se vztahují.

css

Použití media query ve vnořování

```
1 .container {
2   width: 100%;
3
4   @media (min-width: 768px) {
5     width: 750px;
6   }
7 }
```

Tento kód je ekvivalentní k následujícímu:

css

Ekvivalentní kód bez vnořování s použitím media query

```
1 .container {
2   width: 100%;
3 }
4 @media (min-width: 768px) {
5   .container {
6     width: 750px;
7   }
8 }
```

Selektor vnořování & lze použít i v jiných místech selektoru. Například:

css

Použití selektoru vnořování v jiných částech selektoru

```
1 img {
2   article & {
3     border: 2px solid black;
4   }
5 }
```

Výše uvedený kód způsobí, že všechny obrázky uvnitř článků (`<article>`) budou mít černý rámeček.

7.1.3 Budoucnost CSS

Jak již bylo řečeno, CSS se neustále vyvíjí a přidávají se do něj nové funkce. Snaží se totiž nahradit potřebu preprocesorů, které byly dříve nutné pro psaní pokročilých stylů. O nich se budeme bavit v následující části kapitoly.

Použití novinek v CSS totiž vede k jednoduší práci i pochopení celého stylování. Vzhledem k tomu, že jsou stále nové, je potřeba sledovat kolik moderních prohlížečů je podporuje. Toho lze docílit pomocí nástrojů jako je Can I use. Výše uvedené ukázky (proměnné a vnořování) jsou již dnes podporovány ve všech moderních prohlížečích a je možné je bez obav používat.

Další novinky, které se do CSS chystají jsou například:

- Container Queries – Media query, které se vztahují na velikost rodičovského kontejneru, nikoliv na velikost okna,
- Uživatelsky definované funkce,
- Uživatelsky definované tzv. mixiny,
- Vylepšené operátory pro barvy a další.

> 🔎 **Využití `@container` queries**
>
> Container queries jsem již začal vyučovat v rámci předmětu WBA ve školním roce 2025/2026. Od vás se neočekává, že je budete plně znát (ale jsou skvělé!), od dalších ročníků je již plánuji v rámci MVOP vyžadovat.
>
> Každopádně, at-pravidlo `@container` je popsáno ve skriptech WBA.

7.2 Preprocesory

Preprocesor je obecný název pro nástroj, který nám dovolí psát kód v nějakém jazyce, který se následně přeloží do jiného jazyka. V případě CSS preprocesorů nám dovolí psát kód v jazyce, který má více funkcí než samotné CSS a následně se tento kód přeloží do standardního CSS, které prohlížeče umí zpracovat. Podobně jsme se s preprocesory setkali u jazyka TypeScript, viz Kapitola 5.1, kde TypeScript překládáme do JavaScriptu. Často se zaměňuje pojem preprocesor a metajazyk, v podstatě se jedná o to samé.

Pro CSS je nutné si uvědomit, proč preprocesory vůbec vznikly. Psaní CSS bylo dříve velmi omezené, protože CSS neumělo základní věci jako proměnné, funkce, vnořování selektorů a další. To vedlo k tomu, že vývojáři začali vytvářet nástroje, které tyto funkce přidávaly.

Bez preprocesorů by bylo velmi obtížné psát velké a složité styly, protože by bylo nutné opakovat spoustu kódu a nebylo by možné využít základní programovací koncepty jako proměnné a funkce. Proto se preprocesory staly velmi populárními a dodnes jsou používány v mnoha projektech právě pro ušetření času a zjednodušení vývoje. V moderních dobách, viz předchozí část kapitol, se však mnoho funkcí preprocesorů dostalo přímo do CSS, což vede k tomu, že jejich používání není již tak nutné jako dříve. Přesto zůstává řada výhod, proč je používat. Nyní si představíme nejznámější CSS preprocesor a jeho funkce.

Mezi nejznámější CSS preprocesory patří:

- Sass (Syntactically Awesome Stylesheets),
- Less (Leaner Style Sheets),
- Stylus.

7.3 Sass

Jeden z nejpoužívanějších CSS preprocesorů je Sass. Proč je nejpoužívanější je těžké říct, ale pravděpodobně to souvisí s jeho dlouhou historií a širokou podporou v různých nástrojích a frameworkách. Navíc navazuje na syntax CSS, takže je pro vývojáře snadné se ho naučit.

Sass nabízí dva základní formáty psaní kódu:

- **SCSS** (Sassy CSS): Tento formát je rozšířením standardního CSS, což znamená, že každý platný CSS kód je také platný SCSS kód. SCSS používá složené závorky `{}` pro bloky a středníky `;` pro oddělení vlastností, stejně jako CSS,
- **Sass**[22] (indentační syntaxe): Tento formát nepoužívá složené závorky ani středníky. Místo toho používá odsazení pro označení bloků a nové řádky pro oddělení vlastností. Tato syntaxe je více kompaktní, ale může být méně čitelná pro ty, kteří jsou zvyklí na tradiční CSS.

V této kapitole se budeme věnovat formátu SCSS, protože je více podobný standardnímu CSS a je pravděpodobně častěji používaný[23].

scss

Ukázka kódu v SCSS

```scss
.button {
  background-color: blue;
  color: white;

  &:hover {
    background-color: darkblue;
  }
}
```

sass

Ukázka kódu v indentační syntaxi Sass

```sass
.button
  background-color: blue
  color: white

  &:hover
    background-color: darkblue
```

Přípony soubory pro SCSS jsou `.scss` a pro indentační syntaxi `.sass`.

Uvnitř souborů můžeme psát komentáře dvěma způsoby a to pomocí `//` pro jednořádkové komentáře a `/* ... */` pro víceřádkové komentáře.

7.3.1 Kompilace

Pro použití Sass je nutné mít nainstalovaný nástroj, který nám dovolí přeložit kód z jazyka Sass do CSS. Nejčastěji se používá oficiální nástroj `sass`, který je dostupný jako balíček pro Node.js. Instalace probíhá pomocí příkazu:

[22] Ano, hrozný nápad pojmenovat syntaxi podle jména knihovny.

[23] Za celou dobu výuky MVOP jsem viděl pouze jednoho studenta, který se dobrovolně rozhodl používat indentační syntaxi Sass.

bash

```
Instalace nástroje Sass pomocí npm
1 npm install -g sass
```

Poté můžeme v našem projektu spustit příkaz pro kompilaci:

bash

```
Kompilace souboru SCSS do CSS
1 sass input.scss output.css
```

Můžeme taktéž přidat přepínač `--watch`, který bude sledovat změny v souboru a automaticky překládávat při každé změně:

bash

```
Sledování změn a automatická kompilace
1 sass --watch input.scss:output.css
```

Mimo jiné nám kompilace vytvoří soubor `.css.map`, který slouží pro mapování mezi původním SCSS kódem a vygenerovaným CSS kódem. To využívá například debugger (DevTools) v prohlížeči, který nám dovolí vidět původní SCSS kód místo vygenerovaného CSS kódu při ladění stylů.

7.3.2 Proměnné

Sass nám dovoluje definovat proměnné pomocí `$` následovaného jménem proměnné.

scss

```
Použití proměnných v Sass
1 $primary-color: #3498db;
2 .button {
3   background-color: $primary-color;
4 }
```

> **CSS vs. Sass proměnné**
>
> Proměnné v Sass jsou zpracovávány během kompilace do CSS, což znamená, že jejich hodnoty jsou pevně stanoveny v době překladu. Na druhou stranu, CSS proměnné jsou zpracovávány prohlížečem za běhu a mohou být dynamicky měněny pomocí JavaScriptu nebo dědičnosti v rámci DOM stromu.
>
> CSS proměnné ale nelze uvnitř Sass používat pro řízení kompilace – například podmínek nebo smyček.

7.3.3 Základní konstrukce

Mezi základní konstrukce v Sass patří podmínky a smyčky.

scss

```
Podmínky v Sass
1 $theme: dark;
2 .button {
```

```
  @if $theme == dark {
    background-color: black;
    color: white;
  } @else {
    background-color: white;
    color: black;
  }
}
```

scss

Smyčka `@for` v Sass

```
@for $i from 1 through 3 {
  .col-#{$i} {
    width: (100% / 3) * $i;
  }
}
```

Kód výše způsobí v rámci kompilace chybu, protože v CSS existuje operátor `/`, který má jiný význam než dělení[24].. Pro dělení v Sass je nutné použít funkci `math.div()` z modulu `math`.

scss

Dělení v Sass pomocí funkce `math.div()`

```
@use "sass:math";
@for $i from 1 through 3 {
  .col-#{$i} {
    width: math.div(100%, 3) * $i;
  }
}
```

Výše uvedená ukázka vytvoří následující CSS kód:

css

Vygenerovaný CSS kód

```
.col-1 {
  width: 33.3333%;
}
.col-2 {
  width: 66.6666%;
}
.col-3 {
  width: 100%;
}
```

Pokud zrovna nedělíme, je možné s proměnnými dělat běžné matematické operace jako sčítání, odčítání a násobení.

scss

Matematické operace s proměnnými v Sass

```
$base-padding: 10px;
```

[24]V CSS je `/` používán například v rámci vlastnosti `font`, kde odděluje velikost písma od řádkové výšky, nebo například v rámci vlastnosti `grid-row`, kde odděluje začátek a konec řádku.

```
2 .container {
3   padding: $base-padding * 2;
4 }
```

Dovnitř proměnných můžeme ukládat různé datové typy jako čísla, řetězce, barvy, seznamy a mapy.

scss

Různé datové typy v proměnných v Sass

```
1 $font-stack:    Helvetica, sans-serif;
2 $primary-color: #333;
3 $base-margin: 16px;
4 $is-enabled: true;
5 $theme-colors: (
6   "primary": #3498db,
7   "secondary": #2ecc71,
8   "danger": #e74c3c
9 );
10 body {
11   font: 100% $font-stack;
12   color: $primary-color;
13   margin: $base-margin;
14 }
15 .button {
16   @if $is-enabled {
17     background-color: map.get($theme-colors, "primary");
18   } @else {
19     background-color: gray;
20   }
21 }
```

Dalším typem cyklu je smyčka `@each`, která nám dovolí iterovat přes seznamy nebo mapy.

scss

Smyčka `@each` v Sass

```
1 $colors: ("red", "green", "blue");
2 @each $color in $colors {
3   .text-#{$color} {
4     color: $color;
5   }
6 }
```

Dále existuje smyčka typu `@while`, která funguje podobně jako v běžných programovacích jazycích.

7.3.4 Vnořování

Vnořování v SASS funguje zcela stejně jako v moderním CSS, viz předchozí část kapitoly.

scss

Vnořování v Sass

```
.navbar {
  background-color: #333;

  .nav-item {
    color: white;

    &:hover {
      color: yellow;
    }
  }
}
```

Není to náhoda – vnořování v CSS bylo inspirováno právě preprocesory jako je Sass. Obecně je ale více použitelný selektor vnořování & právě v Sass, protože dovoluje jejich použití např. uvnitř `@media` a dalších at-pravidel.

7.3.5 Moduly

Sass nám dovoluje rozdělit kód do více souborů pomocí modulů. Moduly se importují pomocí `@use` nebo `@import`. Doporučuje se používat `@use`, protože `@import` je zastaralé a bude v budoucnu odstraněno.

scss

Definice modulu s proměnnými

```
// _variables.scss
$primary-color: #3498db;
```

scss

Import modulu a použití proměnné

```
// styles.scss
@use 'variables';
.button {
  background-color: variables.$primary-color;
}
```

Modul můžeme případně pojmenovat pomocí klíčového slova `as`.

scss

Import modulu s aliasem

```
@use 'variables' as vars;
.button {
  background-color: vars.$primary-color;
}
```

7.3.6 Mixiny a funkce

Funkce jsou v Sass podobné jako v běžných programovacích jazycích. Dostávají parametry a vrací hodnotu. V SASS je používáme pro opakující se výpočty nebo transformace hodnot.

scss

Definice a použití funkce v Sass

```scss
@function calculate-rem($size-px) {
  @return $size-px / 16px * 1rem;
}
.button {
  font-size: calculate-rem(24px);
}
```

Mixiny jsou podobné funkcím, ale místo vracení hodnoty vkládají blok kódu přímo do místa, kde jsou volány. Blok kódu jsou buď jednotlivé vlastnosti nebo celé bloky pravidel.

scss

Definice a použití mixinu v Sass

```scss
@mixin flex-center {
  display: flex;
  justify-content: center;
  align-items: center;
}
.container {
  @include flex-center;
  height: 100vh;
}
```

Uvnitř mixinů můžeme používat proměnné a podmínky stejně jako v běžném kódu Sass.

scss

Mixiny s parametry a podmínkami

```scss
@mixin button-style($bg-color, $text-color) {
  background-color: $bg-color;
  color: $text-color;

  @if $bg-color == black {
    border: 1px solid white;
  } @else {
    border: 1px solid black;
  }

  @media(max-width: 600px) {
    font-size: 14px;
  }
}
.button {
  @include button-style(blue, white);
}
```

7.4 Alternativní preprocesory

Dalším zmíněným preprocesorem je Less. Less má podobné funkce jako Sass, ale jeho syntaxe a některé koncepty se liší. Less používá například `@` pro definici proměnných místo `$`. Další rozdíly jsou v tom, jak se definují mixiny a funkce. Obecně ale není tak rozšířený jako Sass.

Stylus je další preprocesor, který nabízí velmi flexibilní a minimalistickou syntaxi. Stylus dovoluje psát kód bez složených závorek a středníků, podobně jako indentační syntaxe Sass. Stylus také nabízí mnoho pokročilých funkcí, ale není tak populární jako Sass a ani jako Less.

7.5 Kdy používat preprocesory

I přes to, že moderní CSS nabízí mnoho funkcí, které dříve vyžadovaly preprocesory, stále existují situace, kdy může být použití preprocesoru výhodné.

Mezi aktuální největší výhody používání preprocesorů patří:

- Psaní funkcí a mixinů pro opakující se vzory,
- Pokročilé řízení toku pomocí podmínek a smyček,
- Generace kódu na základě datových struktur jako jsou seznamy a mapy,
- Lepší organizace kódu pomocí modulů a importů,
- Psaní jednořádkových komentářů[25].

V moderních projektech není nutné začít psát Sass nebo jiný preprocesor od začátku. Můžeme je používat například pouze pro konkrétní části, základní styly a podobně. Protože SASS se řeší již v době sestavení projektu, není problém kombinovat jej s běžným CSS. Nezpomalí nám to načítání stránek, protože výsledkem je vždy standardní CSS. Tato myšlenka se nám bude hodit v dalších kapitolách, zejména o frameworcích, viz Kapitola 9.1.

7.6 Budoucnost preprocesorů

S příchodem moderního CSS se role preprocesorů mění.

V budoucnosti se očekává, že preprocesory budou více zaměřeny na specifické potřeby vývojářů, jako je generování kódu a pokročilé řízení toku, spíše než na základní funkce, které již jsou dostupné v CSS. Postupně se věří a může se stát, že preprocesory se přestanou úplně používat, protože moderní CSS bude dostatečně schopné pro většinu potřeb vývojářů. Pokud příjdou zmíněné funkce a mixiny, tak jedinou výhodou preprocesorů zůstane generování kódu na základě datových struktur a lepší organizace kódu pomocí modulů. A to řada vývojářů vůbec ke svým projektům nepotřebuje.

[25] Nemohl jsem si pomoci…

Verzování

8.1 Verzovací systémy

Verzovací systémy jsou nějaký systém či software, který je určen pro správu změn jakýkoliv projektů a prací. Projekt se často v průběhu času mění a to nejvíce ve vývoji, kdy jsou klíčové části vytvořeny.

Verzovací systémy často mohou být i jen nějaké postupy a doporučení, kterými se jednotlivci i týmy řídi. Verzovací systémy se nejčastěji používají právě pro práci mezi více lidmi, aby byl ve změnách přehled a mohlo pracovat nezávisle na sobě. Klasické kopírování složek (např. s jménem datumu změny) je do jisté míry taktéž verzovací systém, ten nás ale nechrání vůbec proti ztrátě dat.

Název revize se ve verzovacích systémech používá pro označení balíčků změn. Například revize je změnění několika souborů v projektu. Často se zaměňuje s názvem verze.

Verzovací systémy jsou často i v jistých programech a webech. Jako příklad můžeme uvést Google Docs, které ukládají historii verzí dokumentu, ve kterých můžeme hledat a dokonce se i k verzi vrátit.

Verzovací systémy se používají na řadu typů projektů, nejčastěji však na ty spojené s IT (kódy, nastavení, weby, ...) ale lze je používat i na grafické či vlastně jakékoliv jiné. Práce s binárními soubory (např. obrázky) je však komplikovanější. Verzování nám pomůže všude, je totiž jednoduché se dívat na historii a případně se k ni vracet. Sjednocení změn z různých zdrojů (při práci více lidí najednou) se taktéž zjednoduší.

8.1.1 Práce se soubory

Verzovací systémy často používají různé styly práce se soubory. Jeden z nich je zamykání souborů a spojování změn. Zamykání souborů je způsob, kdy se celý projekt nebo soubor před prací jedince zamkne. V tomto stavu nemůže nikdo jiný na souboru pracovat, resp. jej uložit (max. číst). Tím zcela zmizí problém sjednocování změn (více lidí pracovalo, teď je musíme dát dohromady), ale zase může pracovat pouze jeden člověk na souboru (v případě projektu jen jeden zároveň). To je pro rychlé vytváření projektů velmi neefektivní. Často se totiž stává, že člověk může nechat soubor nechtěně zamknutý. Na tento styl se hodí právě grafické soubory a další binární soubory.

Slučování změn je mírnější. Na projektu může pracovat kdokoliv a kdykoliv, jen když se soubor uloží a někdo jej předtím ještě změnil, musí se určit, jak ten soubor vlastně má vypadat — sloučit se do pravého stavu. Na tento typ se hodí jakýkoliv typ souboru, který je textové povahy (kódy, texty, nastavení, ...).

8.1.2 Umístění dat

V základním režimu rozlišujeme tři různé přístupy k umístění dat.

Centralizované:

Existuje pouze jeden autoritativní server (umístění), kde jsou všechny změny uchovány nehledě na uživatele.

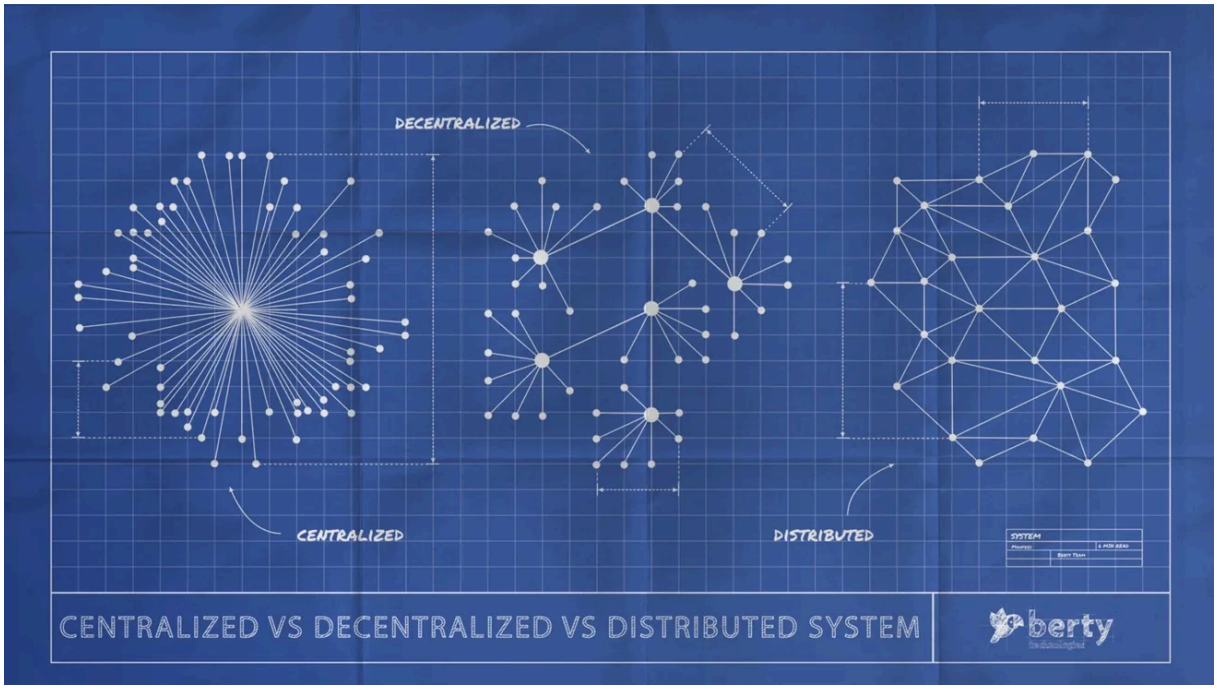
Decentralizované:

Mají různé majitele, na které se pojí další uživatelé, nemusí vědět o sobě navzájem.

Distribuované:

Všichni vědi o ostatních a všichni mají ke všem verzím přístup.

Toto umístění nemusí automaticky znamenat, že daný typ automaticky komunikuje se serverem. Velmi často se komunikuje jen tehdy, když je to potřeba.



Obrázek 1: Rozdíly umístění dat ve verzovacích systémech

8.1.3 Historie verzí

Obecně existují pouze dva typy historií verzí ve verzovacích systémech. Jedná se o lineární a grafovou historii. Lineární je jednoduchá — jedná se o list, ve kterém každá verze ukazuje na tu předchozí a první neukazuje na nic. S tímhle typem se moc nesetkáme u týmových projektů.

Grafová historie — jedná se o orientovaný (tj. ukazuje na něco) acyklický (tj. nemá cyklus, nelze se postupem vpřed dostat do nějaké předchozí verze) graf, kde z jedné verze může existovat více verzí, které se mohou různě spojovat.

8.1.4 Známé verzovací systémy

Mezi nejznámější verzovací systémy patří:

- Git
- Subversion (SVN)
- Mercurial
- Perforce
- Bazaar

8.2 Git

Git je opensource (tj. můžete se podívat na kód) verzovací systém, který je velmi populární. Používá ho řada služeb a programátorů a v podstatě nemá v moderním světě žádnou velkou

konkurenci. Byl vytvořen v roce 2005 Linusem Torvaldsem a Juniem Hamanem. Používá řadu protokolů, jako jsou HTTP, SSH[26] a i FTP.

Cíle Gitu jsou:

- Být jednoduchý (pro naučení stačí pár hodin samostudia),
- Podporovat distribuované použití (celá historie je u všech zařízení, mohou se stahovat a upravovat bez nutnosti serveru),
- Být zdarma,
- Být rychlý (proto vlastní Git vzniknul, protože dřívější systémy pro vývoj jádra Linux byly velmi pomalé a některé akce trvaly i hodiny),
- Ochránit proti ztrátě dat (dost aktivit v Gitu lze vzít zpět (resp. získat data zpátky) a již z principu je možné se vrátit do předchozích verzí).

Charakteristiky Gitu jsou:

- Data se po odeslání jen tak neztratí (existují lokální kopie u klientů),
- Změny se na sebe váží a jsou propojeny,
- Veškeré objekty v Gitu jsou reprezentovány jednoznačným identifikátorem pomocí hashovací funkce SHA-1[27],
- Podporuje různé OS, systémy a jazyky,
- Rychlost Gitu závisí především na hardware počítače a se sítí komunikuje pouze pokud to uživatel chce.

Velmi důležitou vlastností Gitu je to, že neukládá změny jednotlivých souborů, ale jejích snímky. Tedy, jak v určité verzi celý soubor vypadal. A pokud se soubor nezměnil, ukazuje to na nějakou předchozí verzi.

Git se rozděluje do tří částí, se kterými se pracuje:

- Working directory, neboli pracovní adresář, což je složka ve které pracujete na projektu,
- Staging area, neboli připravené změny, což je speciální místo, do kterého dáváte soubory, které půjdou do revize,
- Git directory, neboli Git složka, ve které se nacházejí informace i Gitu a my je můžeme měnit.

Pro práci s Gitem je nutné mít Git jako příkaz v příkazové řádce (tj. CLI). Prozatím budeme využívat právě příkazovou řádku (např. na Windows Powershell) a později si ukážeme Git klienty, které mají GUI.

Git má skvělou knížku, která celou práci s Gitem a jeho vnitřnosti popisuje a vysvětluje. Během či po přečtení této kapitoly ji doporučuji číst na oficiální stránce — je skoro celá v českém jazyku!

8.2.1 Nastavení Gitu

Jako první příkaz si ukážeme příkaz k nastavení gitu. Tento příkaz slouží především na nastavení repozitáře (více později) ale i globálních nastavení. Nastavení používá páry klíče a hodnoty. Seznam všech klíčů naleznete v nápovědě, viz dále.

Jako základní nastavení po nainstalování by se hodilo nastavit správné jméno a e-mail, který se používá skoro všude jako reference na to, kdo změny tvoří.

bash

[26] Podle nastavení je nutné dané službě (serveru, uživateli) předat svůj veřejný klíč.

[27] SHA-1 je již zastaralá a nedoporučuje se pro bezpečnostní účely, ale pro Git je stále dostačující.

Příkazy k nastavení jména a e-mailu

```
1 git config --global user.name "Pepa Kopecky"
2 git config --global user.email kopecky.pe.2020@skola.ssps.cz
```

Výše uvedený příkaz nastaví globálně (dle přepínače `--global`) klíče `user.name` a `user.email` na dané hodnoty. Je-li hodnota víceslovná, je nutné celou hodnotu obalit uvozovkami.

Můžeme dokonce vypsat veškerá nastavení a určité nastavení dle klíče.

bash

Příkazy k výpisu nastavení

```
1 git config --list
2 git config user.name
```

Je důležité si uvědomit, že nastavení může být vázáno na repozitář (tj. bez `--global`). Vždy má prioritu nastavení repozitáře a až potom globální.

Všechny příkazy lze vyvolávat v jakékoliv složce repozitáře. Příkazy vždy najdou nejbližší rodičovskou složku, která obsahuje složku `.git`.

Pokud si nebudete s nějakým příkazem vědět rady, je možné si pro něj zobrazit přímo pomocí příkazu nápovědu.

bash

Příkazy k zobrazení nápovědy

```
1 git help <příkaz>
2 git <příkaz> --help
3 git help config
```

8.2.2 Repozitáře

Repozitář je název pro projekt, který je verzovaný pomocí Gitu. Jedná se tedy o složku, ve které je projekt, který je verzován. Tento repozitář musí být vytvořen (např. již v založeném projektu) a nebo to děláme ještě předtím. Často repozitář taktéž tzv. klonujeme, tedy stáhneme z internetu, ale to si povíme až v dalších kapitolách.

Repozitář má velmi důležitý úkol, a to je, sledovat veškeré změny projektu. Uvnitř této složky nalezneme taktéž složku `.git`, která uchovává veškerou historii, nastavení a cokoliv co Git potřebuje.

8.2.2.1 Založení repozitáře

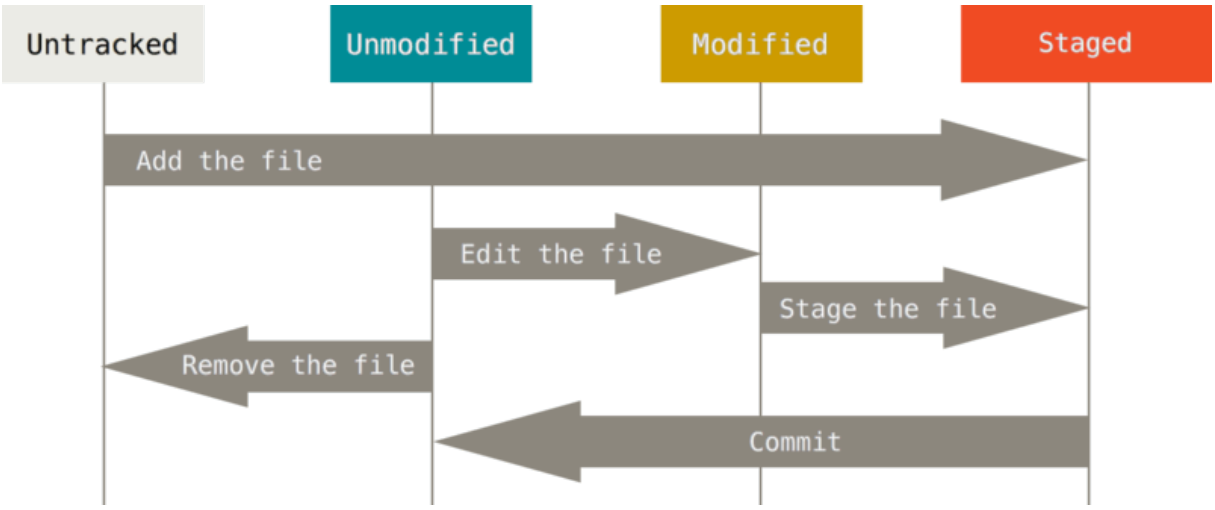
Založit repozitář nyní můžeme lokálně pomocí příkazu `git init`. Tento příkaz vytvoří složku `.git`, veškeré základní nastavení a ihned začne sledovat soubory. Když tuto složku smažeme, přijdete o veškerou historii projektu (prozatím lokální kopii).

8.2.3 Commit

Commit je balíček změn (již víme, že se jedná o snímky), které jsou již zaznamenané a ukončené. Jedná se tedy už o nějakou revizi (verzi) projektu, ve které se něco změnilo. Celý repozitář se z těchto commitů skládá (některé repozitáře jich mají miliony). Commit je vždy

identifikován SHA-1 hashem, kterým na něj referujeme. Commit má taktéž svojí zprávu, časovou značku a autora. Většina commitů má alespoň jednoho rodiče (ze kterého své změny putují). Jen jeden commit nemá žádného rodiče a to je první commit v linii historie.

Git používá stavy souborů:
`untracked` soubor není sledován, jedná se o nový soubor, který v historii není,
`unmodified` soubor již byl sledován, ale svůj obsah nezměnil,
`modified` soubor je sledován a změnil se od poslední revize,
`staged` soubor je sledován a je připraven být zabalen do commitu.



Obrázek 2: Stavy souborů v Gitu

Pro zjištění, v jakém stavu je aktuálně repozitář, lze použít příkaz `git status`. Tento příkaz nám vypíše, stavy jednotlivých souborů projektu. Vynechává soubory, které se nezměnili.

8.2.3.1 Přidání souborů do commitu

Pokud chceme označit soubory jako staged, můžeme to udělat (jak nám říká nápověda výpisu statusu) pomocí příkazu `git add`. Lze například udělat:

bash

Příkazy k přidání souborů do commitu

```
1 git add <soubor>
2 git add README.md
3 git add path/to/another/file.txt
```

V případě, že označíme soubor jako staged, tak se označí na odeslání aktuální stav soubor (snímek) a při další úpravě tyto úpravy nebudou automaticky jako staged označeny.

Existují taktéž různé přepínače a zápisy, které označí více souborů najednou:

`git add -A` označí všechny soubor v celém repozitáři,
`git add .` označí všechny soubor v aktuální složce a v podsložkách (a rekurzivně),
`git add -u` označí soubory, které byly modifikované a smazané, ale neoznačí nové soubory.

8.2.3.2 Přesné změny revize

Příkaz `git status` ale není moc vhodný na to, abyste zjistili co se přesně změnilo (řádky kódu, ...). Proto existuje příkaz `git diff` a `git diff --staged`, kde první zobrazí ve formátu patch

změny všech souborů ve working directory s minulou verzí. Druhý příkaz toto porovnání vytvoří pouze se soubory, které jsou staged.

Patch je formát, ve kterém se vám zobrazí jaké řádky byly přidány a smazány společně s nějakými metadaty (jaké commity, jaké SHA-1 hashe, ...). Plus na začátku řádky značí přidaný řádek a mínus smazaný řádek.

8.2.3.3 Vytvoření commitu

Nyní víme co je to commit, jak označit soubory jako staged ale nevíme, jak vůbec takový commit vytvořit. Po tom, co přidáme všechny námi chtěné soubory (resp. změny a jejich snímky), můžeme zavolat příkaz:

bash

Příkazy k vytvoření commitu

```
1 git commit -m "Zpráva commitu"
2 git commit
```

První řádek okamžitě vytvoří commit s uvedenou zprávou (o tom, jak by zpráva měla vypadat, si povíme v jiné kapitole). Druhý příkaz je komplikovanější. Ten vám otevře vámi nastavený editor[28] (nebo základní pro úpravu souborů), ve kterém uvidíte všechny změny v commitu a do něj napíšete danou zprávu. Po uzavření souboru se commit vytvoří a zapíše.

Po zapsání změn se staged soubory odznačí a celý cyklus může začít znovu.

8.2.3.4 Ignorování souborů

Mimo složku .git umí Git pracovat i se soubory, které více specifikují práci. Jeden z těchto souborů je soubor .gitignore, který lze vložit do jakékoliv složky, ale nejčastěji se vkládá do kořenového adresáře. Tento soubor ukládá informace o souborech, které má Git ignorovat (nebude je sledovat a přidávat).

Soubor obsahuje různá pravidla, lze v něm používat různé zástupné znaky a negace. Níže je ukázka s pravidly a vysvětlením toho, co ignorují.

Ukázka souboru `.gitignore`

```
1 # Ignoruj všechny soubory (v jakékoliv složce), které končí na .log.
2 *.log
3
4 # Negace pravidla -- přestože výše zakazuje tento soubor, sleduj ho.
5 !dev.log
6
7 # Ignoruj složku build v hlavním adresáři.
8 /build/
9
10 # Ignoruj soubor build v hlavním adresáři.
11 /build
12
13 # Ignoruj všechny složky lib (v jakékoliv složce).
```

[28] Pozor na otevření editor Vim, který se uzavírá zapsáním `:wq`. ;)

```
14 lib/
15
16 # Ignoruj soubor config.xml v jakékoliv podsložce složky config.
17 /config/**/config.xml
18
19 # Ignoruj všechny soubory, které začínají config.
20 config*
```

8.2.4 Historie commitů

Jeden z cílů verzovacích systémů je dovolit uživatelům sledovat, co se v daném projektu stalo. Nyní si ukážeme příkaz, který právě to dělá. Tento příkaz má spoustu přepínačů a každý je důležitější než ten předchozí. Neukážeme si taktéž všechny, ale ukážeme si ty nejdůležitější. Přepínače lze kombinovat a tím zobrazení vylepšovat.

V každém případě je výpis commitů v historii vždy chronologický a velmi často platí, že první záznam, který uvidíte, je nejnovější commit. Zobrazení v tomto příkazu je často zobrazováno v menším okně, ve kterém se lze pohybovat pomocí šipek a tlačítek PAGE UP a PAGE DOWN. Z tohoto okna odejdete pomocí zápisu `q`.

V základu je příkaz `git log` velmi strohý. Vypíše nám seznam commitů a neukáže nám žádné vazby (resp. větve, o kterých si povíme později). Bez přepínačů se nám ke každému commitu ukáže identifikátor, autor, časová značka a jeho zpráva.

Pomocí přepínače `-x`, kde `x` (např. `-1`, `-30`) je celé kladné číslo můžeme určit počet commitů, které se nám v tomto zobrazení ukáže.

Pomocí přepínače `--stat` můžeme ke každému commitu zobrazit jednoduchou statistiku, která říká kolik bytů, řádků a změn bylo vytvořeno pro daný soubor a commit. Ukázkový výpis:

bash

Ukázka výpisu s přepínačem `--stat`

```
1 commit 9fceb02... (HEAD -> main, origin/main, origin/HEAD)
2 Author: Yoko <yoko@cajthaml.eu>
3 Date:   Mon Mar 7 22:23:45 2022 +0100
4
5     Update README.md
6
7  README.md                                 | 2 +-
8  1 file changed, 1 insertion(+), 1 deletion(-)
```

Přepínačem `--graph` se zobrazí celá historie jako graf, na kterém můžeme vidět různé větve a jejich spojování. Větve jsou vždy po levé straně. O větvích si povíme později.

Přepínačem `--pretty=oneline` se každý commit vypíše na jeden řádek, což je velmi užitečné pro rychlý přehled.

Pomocí přepínačů `--since`, `--before`, `--after` a dalších můžeme omezit, jaké commity se nám zobrazí. Pro datum používáme formát `YYYY-MM-DD` nebo zápisy jako `x.weeks` a obdoby. Ukázky:

bash

Ukázky příkazů s přepínači pro časové omezení

```bash
git log --since="2023-01-01"
git log --after="2.weeks"
git log --before="2023-06-01"
git log --since="2023-01-01" --before="2023-06-01"
```

Pomocí přepínačů `--author` a `--grep` můžeme vyhledávat v autorech commitů a jejich zpráv. Ukázky:

Ukázky příkazů s přepínači pro vyhledávání v historii

```bash
git log --author="Pepa"
git log --grep="bugfix"
git log --author="Pepa" --grep="bugfix"
```

8.2.5 Změny stavu souborů

Často se nám stává, že se musíme vrátit do předchozího stavu souboru či commitu. Nyní si ukážeme nejčastější akce.

Resetování aktuálně změněného souboru:

Používáme tehdy, když chceme smazat veškeré změny, které se v daném souboru staly od posledního commitu.

Příkazy k resetování změněného souboru

```bash
git checkout -- <soubor>
git checkout -- README.md
git restore <soubor>
git restore README.md
```

Odstranění souboru z připravených změn:

Používáme tehdy, když jsme omylem přidali soubor do připravených změn (staged) a chceme jej odtamtud odebrat.

Příkazy k odebrání souboru z připravených změn

```bash
git reset <soubor>
git reset README.md
git restore --staged <soubor>
git restore --staged README.md
```

Připojení změny k poslednímu commitu/změna zprávy posledního commitu:

Používáme tehdy, když jsme zapomněli přidat nějaký soubor do posledního commitu a chceme jej tam přidat. Nebo když chceme změnit zprávu posledního commitu.

Tehdy musíme změnit staged soubory (přidat/odebrat) a poté zavolat tento příkaz.

Příkazy k připojení změny k poslednímu commitu nebo změně zprávy posledního commitu

```bash
git commit --amend
```

```
2 git commit --amend -m "Nová zpráva"
```

Návrat ke konkrétnímu commitu:

Používáme tehdy, když chceme vrátit celý projekt do určitého commitu. Checkout používáme tehdy, když chceme pouze prohlížet soubory v daném commitu (v read-only režimu). Restore používáme tehdy, když chceme změnit aktuální stav projektu na daný commit (v read-write režimu).

bash

```
Příkazy k návratu ke konkrétnímu commitu
1 git checkout <hash>
2 git checkout 9fceb02
3 git restore --source <hash> .
4 git restore --source 9fceb02 .
```

Vytvoření nového commitu, který vrací změny z určitého commitu:

Používáme tehdy, když chceme vytvořit nový commit, který vrací změny z určitého commitu.

bash

```
Příkazy k vytvoření nového commitu, který vrací změny z určitého commitu
1 git revert <hash>
2 git revert 9fceb02
```

8.2.6 Větve

Větve jsou způsob Gitu na rozdělení jedné práce na projektu do více (kupředu pokračující) verzí. Větve existují v odděleném prostředí a při vytvoření commitu neovlivňujeme další větve.

Interně jsou větve velmi jednoduché — když si představíme, jak vypadá graf commitů, tak větve (angl. branches) jsou takové ukazatele na nějaký commit, ve kterém se nacházejí. Na jednom commitu může být tímto způsobem nalepeno neomezeně větví. Postupem času se stane, že jsou větve velmi daleko v historii od sebe.

V týmových projektech se větve často používají pro různé stavy projektu (produkce, development, beta, ...). Jak se využívají podrobně, si povíme později.

Při vytvoření repozitáře jsme si mohli všimnout, že základní větev se jmenuje `master` či `main`. Reálně tato větev nemá žádné speciální funkčnosti a při vytvoření repozitáře ji můžete název změnit.

8.2.6.1 Vytvoření větve

Lokální větev můžeme vytvořit pomocí příkazu `git branch <jméno>`. Větev můžeme kdykoliv taktéž smazat a to přidáním přepínače `-d`. Pokud nebyla větev do žádné jiné větve spojena (viz. níže), musíme použít přepínač `-D`, který všechny změny zahodí. Vytvořením větve se však do ní nepřesuneme.

Do větve se přesuneme pomocí příkazu `git checkout <branch>`. Přesunutí není možné, pokud máte nějaké soubory rozpracované a nebo dokonce označené jako staged. Po přesunutí budou všechny soubory ze staré větve upraveny / smazány / přidány do podoby, kterou nalezneme v nové větvi.

Při vytvoření commitu v nové větvi se rozběhne historie projektu do více části. Toto rozdělení umí hezky reprezentovat příkaz `git log --graph`.

Všechny větve můžeme zobrazit pomocí příkazů `git branch -v`, které vypíše i poslední commit, a nebo `git branch` jen se seznamem.

Aby Git věděl, v jaké větvi jsme, používá ukazatel HEAD, který ukazuje na větev (a my víme, že vlastně ukazuje i commit), ve kterém jsme. V příkazu `git log` a `git status` můžeme vidět, v jaké větvi jsme.

8.2.6.2 Slučování větví

Příkaz `git merge <jméno>` právě sloučení větví způsobí. Příkaz se v aktuální větvi bude snažit přenést změny série snímků ze zadané větve. Referovaná větev zůstane nepozměněna, aktuální větev bude změněna.

Toto slučování může být jednoduchého rázu (a to tehdy, když změny nejsou komplexní a mají například stejného rodiče) a commity se jen různě posunou. Může dojit taktéž k vytvoření nového commitu (tzv. troj-spojení).

Při slučování mohou nastat také problémy se sloučením. Když ve větvích proběhli velké změny ve stejných souborech a řádcích, Git nebude vědět, co je pravý stav, ve kterém chce po sloučení být. Tento incident se jmenuje merge conflict. Merge conflict se musí vyřešit ručně.

8.2.6.3 Merge conflict

Interně se vytvoří v aktuální větvi všechny změny z větve, ze které spojujeme. Do staged area se vloží již vyřešené sloučení a v modified souborech, ve kterých merge conflicty nastali se vytvoří speciální formule, které vám konflikt popíší. Tuto areu poznáte pomocí šipek a očividného textu. Abyste konflikt vyřešili, musíte text u šípek a rovná se smazat. Po opravení může zůstat první, druhá verze a nebo ruční kombinace dvou zároveň. U šipek se Vám taktéž popisuje, v jaké větvi se změny nacházejí.

plaintext

Ukázka merge conflictu v souboru `index.html`

```
<<<<<<< HEAD:index.html
<div id="footer">
    <p>Matěj Cajthaml &copy; 1823</p>
</div>
=======
<footer>
    <p>Ing. Matěj Cajthaml &copy; 2025</p>
</footer>
>>>>>>> dev:index.html
```

Všechny conflicty naleznete v příkazu `git status`. Až soubory opravíte, můžete se sloučením větví pokračovat pomocí `git commit`.

8.2.7 Vzdálené repozitáře

Protože se Git využívá jako týmový nástroj, často je potřeba, aby existoval vzdálený repozitář, do kterého má každý developer přístup a může v něm dělat změny a stav i stahovat.

Důležité je pochopit, že máme lokální repozitář, který s tím vzdáleným nemusí mít leccos společného. Do lokálního repozitáře si můžeme nalinkovat řadu vzdálených repozitářu, ze kterých můžeme číst a pracovat s nimi. Vzdálený repozitář musí mít jméno, pro hlavní repozitář pro týmový vývoj se často používá `origin`.

Všechny vzdálené repozitáře zobrazíme příkazem `git remote -v`, kde uvidíme každý repozitář dvakrát, jednou pro čtení (fetch) a pro nahrání (push). Vzdálený repozitář přidáme příkazem `git remote add <nazev> <url>`, například tedy `git remote add origin https://github.com/ssps-cajthaml/issues`. Tyto repozitáře lze logicky i smazat pomocí příkazu `git remote rm <nazev>`.

Vzdálené repozitáře obsahují větve, těm říkáme vzdálené větve. Tyto větve lze taktéž mazat. Pokud chceme s větví komunikovat, musíme je stáhnout a implementovat do našich lokálních větví. Na větve odkazujeme dle názvu vzdáleného repozitáře a jejího názvu, tedy `<jméno vzdáleného repozitáře>/<jméno větve>`.

Vzdálené repozitáře můžeme ovlivnit jen různými příkazy a nelze je ovlivňovat jen tak např. z working directory!

8.2.7.1 Práce se vzdálenými repozitáři

Stáhnutí změn ze vzdáleného repozitáře:

Příkaz `git fetch <nazev>` stáhne ze vzdáleného serveru stav repozitáře. Příkaz nic jiného nedělá a neovlivňuje přímo lokální repozitář (ani working directory či cokoliv podobného).

Implementace změn ze vzdáleného repozitáře do lokální větve:

Příkaz `git pull <nazev> <vetve>` stáhne a implementuje změny ze vzdáleného repozitáře do aktuální větve. Příkaz je v podstatě zkratka na `git fetch` a `git merge`. Pokud je potřeba, může dojít i k merge conflictům.

Nastavení upstream větve:

Upstream větve je nastavení, které říká, ze které vzdálené větve se mají stahovat změny a do které se mají nahrávat. Příkaz `git branch --set-upstream-to=<nazev>/<vetve>` toto nastavení provede.

Odeslání změn do vzdáleného repozitáře:

Příkaz `git push <nazev> <vetve>` odešle změny z aktuální větve do dané větve ve vzdáleném repozitáři. Pokud upstream větev není nastavena, je nutné ji při prvním odeslání nastavit přepínačem `-u`.

Klonování vzdáleného repozitáře:

Příkaz `git clone <url>` stáhne celý repozitář z dané URL a vytvoří lokální kopii. Při klonování se automaticky nastaví vzdálený repozitář s názvem `origin` a nastaví se i upstream větev na výchozí větev vzdáleného repozitáře.

8.2.8 Gitové služby

Zatímco Git je verzovací systém (nástroj), tak existují víceméně známější Gitové služby, které poskytují nad Gitem další kontrolu, ale vnitřně Git využívají a dávají Vám k němu přístup.

GitLab

Jedná se o opensource Gitovou službu. Můžeme si jí nainstalovat na vlastní servery. Je velmi jednoduchá, hezká a má spoustu funkcionalit. Je zdarma v omezeném prostředí.

GitHub

Jedná se o nejznámější Gitovou službu, je však bohužel closedsource. Je zdarma v omezeném prostředí. Pro další části textu budeme uvažovat zejména možnosti, které nabízí ona (avšak např. pro GitLab existují skoro 1:1 alternativy).

Uvnitř Gitových služeb nalezneme několik důležitých částí, které si nyní popíšeme. Ne v každé Gitové službě se tyto části jmenují stejně, ale jejich funkce je velmi podobná.

8.2.8.1 Funkcionality

Issues:

Je to vlastně seznam problémů, nápadů a bugů v daném projektu. Slouží pro lepší orientaci toho, co je potřeba udělat. Když se Issue dokončí, tak se vyřeší a v hlavním seznamu se již nebude zobrazovat. Issue můžeme například v commitech referovat. V Issues lze diskutovat, přiřadit lidi a např. přiřadit značky.

Každá Issue má svoje číslo (čísla se sdílí ještě s pull requesty). Pro Issues se doporučje být konstruktivní a jasně psát své požadavky a problémy. Je důležité např. uvést i jaký operační systém používat a další podrobnosti, které vedou k reprodukci problému.

GitHub dovoluje vytvářet vlastní šablony, které donutí uživatele vyplnit různá pole, které zlepšují přehlednost Issues.

Projects:

Jedná se o tzv. Kaban board a nebo tabulky, kdy jsou jednotlivé Issues (a dokonce i textové poznámky) rozdělené do několika kategorií. Používá se pro zjednodušení čtení issues a určení priorit. GitHub dovoluje tyto boardy automatizován různými akcemi při přesunu atp.

Pull Request:

Dovoluje Vám požádat repozitář (resp. jeho majitele) o to, aby z Vašeho repozitáře (resp. větve) stáhli změny a implementovali je do jejich repozitáře.

I když jsou repozitáře veřejné, neznamená to, že do nich (logicky) můžete zapisovat. Proto existuje možnost jak je spojit. Jako v Issues, můžeme do nich psát diskuze, mají různé vlastnosti a přiřazení. Taktéž mají číslo, které můžeme různě referovat. Jen administrátoři (resp. uživatelé s právy) mohou pullrequest schválit a kód implementovat.

Pokud chcete do veřejných repozitářů přidávat kód, často při vytvoření prvního pullrequestu budete požádání o podepsání smlouvy (SLA agreement), ve kterém se vzdáte části svých práv na kód. Například souhlasíte s tím, že kód bude vždy veřejný, může být dále upravován, používán pro komerční účely a že tento krok nelze vzít zpět. Každá společnost má tuto smlouvu jinou.

I když je repozitář veřejně, neznamená to, že ke kódu máme jakékoliv práva. Vždy je důležité zkontrolovat licenci, kterou často naleznete v souboru LICENSE.

Releases:

Jedná se o způsob vydání vašeho programu či projektu. Automaticky se přikládá kód a ukazují se na hlavní stránce. Při vydání můžeme vložit i vlastní soubory, jako např. zkompilované

exe či obdoby. Pro číslování se často používá sémantické verzování, které naleznete v další kapitole.

Milestone:

Určení nějaké cíle projektu. Do tohoto cíle, které má často nějakou časovou značku, můžeme určit issues, které se do té doby musí splnit.

Actions:

Jedná se o velmi silnou funkcionalitu, která dovoluje buď ručně nebo automaticky spouštět různé skripty a kódy v omezeném prostředí. Nejčastěji se používá např. na otestování kódu, nahrání aplikace na server, nebo třeba pro uzavíraní starých (resp. neaktivních) issues. Pro každé spuštění můžeme vidět jaké práce se konají, logy a podobně. GitHub má speciální nastavovací YAML soubor, ve kterém můžeme využívat již předpřipravené části od komunity.

8.2.8.2 Dokumentace

Pro projekt je důležité, abyste psali dokumentaci. Dokumentaci lze psát přímo v kódu (která ale slouží spíše pro vývoj aplikace), je však nutné přemýšlet i na externí dokumentaci, jako jsou návody na spuštění aplikace a podobně. Z kódu lze vytvořit webovou dokumentaci například pomoci Doxygenu a obdob.

Gitové nástroje dovolují vytvářet tzv. Wiki, ve kterých můžete tvořit různé stránky ve formátu Markdown. Gitové nástroje mají taktéž speciální soubor `README.*`, který se zobrazí při načtení repozitáře (resp. jakékoliv složky). Můžeme zde taktéž používat markdown. V souboru README v kořenovém adresáři repozitáře by měly být ty nejpodstatnější informace projektu. Například jak se projekt používá, k čemu slouží a poskytnout další odkazy na zdroje.

8.2.8.3 Sémantické verzování

Jedná se o doporučení, jak správně pojmenovávat (číslovat) verze projektů. Přesné doporučení naleznete na semver.org, kde je vše podrobně vysvětleno. Zde si ukážeme pouze základy.

Sémantické verzování používá tři čísla, kde se každé z nich v různých chvílích zvětšuje, ve formátu `MAJOR.MINOR.PATCH`. Jednotlivé čísla znamenají:

MAJOR: zvýšení právě tehdy, když nastala změna, která není zpětně kompatibilní

MINOR: zvýšení právě tehdy, když se přidala nová funkcionalita, ale kompatibilita zůstala zachována

PATCH: zvýšení právě tehdy, když se opravila chyba, ale kompatibilita zůstala zachována

Při zvýšení čísel se ostatní čísla, které následují (např. při změně MAJOR se změní MINOR a PATCH), vyresetují na nulovou hodnotu. Za těmito čísly se může nacházet ještě předběžné verze a metadata a to ve formátu `-*+*`. Za obě hvězdy lze napsat jakýkoliv text, ale obecně je první např. alfa, beta, gamma a druhá třeba číslo identifikátoru.

Toto doporučení je často různě přeformátováno, kde např. MAJOR znamená velké přepsání aplikace, MINOR velký posun aplikace, a PATCH malý posun či oprava chyb. Nejdůležitější vlastností tohoto verzování je jeho možnost seřazení, mělo by tedy platit, že verze `1.1.1` je zdaleka starší verze, než `12.3.1`.

8.2.9 Zásady práce s Gitem

Nyní si představíme nějaké klíčové zásady práce s Gitem, které v praxi určitě potkáte. Pro každé tvrzení bude v této stránce jeden odstavec, kde na začátku je tučně právě dané tvrzení.

Každý commit by měl být funkční (resp. spustitelný). Má-li Git poskytovat možnost verzování, musí být taktéž schopný návratu do commitu. Pokud v historii existuje rozbitý (nefunkční) commit, tak návratem do něj by se projekt mohl rozbít. Proto je důležité, aby commit reprezentovat jednu celistvou změnu a nebyl rozdělený do více commitů.

Zpráva commitu by měla reprezentovat co se v commitu stalo. Historii v Gitu lze jednoduše zobrazovat a v případě kryptických či nelogických zpráv commitů se může stát, že si daný čtenář bude muset celý commit otevřít a podívat se na dané změny. To by mohlo být v nějakých projektech časově náročné.

Je důležité znát firemní (nebo personální) styl práce, syntaxe a sémantiky. Názvy commitů by měli být konzistentní (např. v přítomném čase, anglicky, ...). Budeme-li konzistentní, kdokoliv, kdo projekt zná, se ihned v naší práci vyzná. Znovu budeme časově efektivní.

Historie repozitáře by se měla číst jako kniha a *nezveřejňovat nedokončené části.* Tuto zásadu nám již naše spojování větví rozbilo. V případě komplikovaných sjednocení se vytváří commit, který je vlastně pozůstatkem spojování a při čtení již přestává dávat smysl. To můžeme vyřešit tzv. rebase — přeskládáním, o kterém jsme si nepovídali.

8.2.9.1 Typy větví

Projekty obvykle používají nějaké typy větví, aby byli co nejvíce konzistentní. Nejčastěji se rozdělují do dvou typů a to stálé (master, develop, next), které zůstávají v projektu napořád a reprezentují různé hlavní verze projektu, které jsou různě zveřejňované. Druhý typ jsou krátkodobé, které po svém využití umírají. Jedná se např. o issues, experimenty či nápady. Pro issues se často vytvářejí různé speciální názvy větví, například dle jejich identifikátoru a krátkého popisu. Pro issue 23: consistent formatting by se vytvořila např. větev `23-formatting`.

8.2.10 Git klienti

Git klient je software (program) a nebo webová stránka, které zjednodušuje práci s repozitáři. Pro její funkčnost musíme mít Git v CLI. Hlavním důvodem, proč se Git klienti používají je to, že uživatel se nerad dívá do příkazové řádky, kde vidí pouze nějaký text. Díky klientovi můžeme velmi dobře vizualizovat, jak historie Gitu vypadá a klikat jen na tlačítka. Git klienti taky skrývají spoustu věcí, které uživatel nemusí chápat, protože příkazy za něj volají oni. To vede k tomu, že tyto nástroje používají i úplní začátečníci.

Známí Gitoví klienti:

- GitKraken
- GitHub Desktop
- Sourcetree
- **v IDE**

Výhody:

- Je vizuálně vidět, v jakém stavu repozitář i projekt je,
- Lze jednoduše vidět jednotlivé soubory a změny v commitech,

- Pro základní práci intuitivní,
- Často jednoduchý způsob řešení merge conflictů,
- Vše to píše za Vás.

Nevýhody:

- Píše to všechno za Vás,
- Často klikáte na něco, čemu vlastně nerozumíte,
- Není dobré na pochopení (nechápete, co dané tlačítko vlastně přesně dělá),
- Často není zdarma, různá podpora.

8.2.11 Závěr

Git je velmi silný verzovací systém, který Vám může pomoci s vývojem Vašich projektů. Je důležité si uvědomit, že Git je pouze nástroj a je na Vás, jak s ním budete pracovat.

Za tak málo hodin (a ještě na střední škole) nemůžeme stihnout všechnu látku Gitu. Počítám s tím, že jestli máte zájem o další věci Gitu, že se je naučíte sami buď z dříve referované knížky, nebo obecně z internetu při práci. Níže je seznam věcí, které jsme neprobrali a nebo jednoduše řekli zjednodušeně:

- **Vnitřnosti Gitu**, Git je velmi pokročilý nástroj a vnitřně je důležité znát další věci, jako je například to, jak funguje resetování, reference, protokoly, obnova dat, a spousta dalších,
- **Hooks**, což jsou speciální skripty, které se na serveru / počítači spouští při commitu, pushi a nebo při jakékoliv jiné akci. Tímto můžete velmi hezky modifikovat přímo Váš development cycle,
- **Co vlastně .git složka obsahuje za věci**, protože to velmi závisí na tom, jak rozumíme vnitřnostem Gitu,
- Různé další příkazy, jako: – `stash` – uchování aktuálních změn do speciálního balíčku, který můžeme otevřít, – `blame` – zjištění, co se s nějakým řádkem v průběhu verze stalo, – přidání více souborů jako staged najednou,
- Různé další způsoby spojování změn — místo sjednocení commitů můžeme použít přeskládání, neboli rebase, či například squash,
- Submoduly, které nám dovolí vložit do repozitáře jiný repozitář,
- A různé další pokročilé věci.

Frameworky

9.1 Co je framework

9.2 Existence frameworků

9.2.1 Reaktivita

9.2.2 Komponenty v DOM

9.3 Vue

9.3.1 Proč zrovna Vue?

9.3.2 Použití

9.3.2.1 Instalace

9.3.2.2 Projekt

9.3.3 Struktura projektu

9.3.3.1 Základní komponenta

9.4 Typy zápisů

9.4.1 Options API

9.4.2 Composition API

9.5 Vue instance

9.5.1 Životní cyklus

9.5.2 Definice komponenty

9.6 Skript

9.6.1 Proměnné

Backend

10.1 Backend

10.1.1 Webové služby

10.1.2 Webový server

10.2 Hypertext Transfer Protocol

10.2.1 Používání na webu

10.2.2 Zabezpečení

10.3 API

10.3.1 Webové API

10.3.2 Verzování API

10.3.3 Representational State Transfer

10.4 Express.js

10.4.1 Middleware

10.4.2 Vstup

10.5 Databáze

10.6 Správa serveru

10.6.1 Nasazení aplikace

10.6.2 Platformy jako služba

10.6.3 Docker

10.6.4 Kontinuální integrace a nasazení

10.6.5 Reverzní proxy

Závěr

11.1 Shrnutí

11.2 Motivace do budoucna

11.3 Další zdroje informací

Pro další studium a prohlubování znalostí doporučuji tyto zdroje:

Obecné webové technologie:

- MDN Web Docs – nejlepší dokumentace pro HTML, CSS a JavaScript,
- Web.dev – Google's platforma pro moderní webový vývoj,
- Can I Use – kompatibilita webových technologií napříč prohlížeči.

Příloha A.

Ověření znalostí

A.1 Opakování WBA

Pro ověření znalostí z WBA si můžete projít skripta daného předmětu.

A.2 Rozšíření znalostí

- ☐ Vím, co je to specificita selektoru.
- ☐ Chápu, jak funguje priorita stylů v CSS.
- ☐ Umí vysvětlit, jak fungují tři úrovně specifikace stylů.
- ☐ Vím, jak funguje dědičnost stylů v CSS.
- ☐ Chápu, kdy využití !important v CSS.

- ☐ Vím, co jsou to pseudo-elementy a jak se liší od pseudo-tříd.
- ☐ Vím rozdíl mezi selektorem `::before` a `::after`.
- ☐ Vím, proč je nutné při `::before` a `::after` používat vlastnost `content`.
- ☐ Umím vysvětlit, proč je někdy vhodné využít vlastnost `content` i pro jiné účely než pseudo-elementy.
- ☐ Umím vysvětlit, co dělá pseudo-element `::first-line`.
- ☐ Umím vysvětlit, co dělá pseudo-element `::first-letter`.
- ☐ Umím vysvětlit, co dělá pseudo-element `::selection`.
- ☐ Umím vysvětlit, co dělá pseudo-element `::placeholder`.
- ☐ Umím vysvětlit, co dělá pseudo-element `::marker`.

- ☐ Umím vysvětlit, jak fungují selektory podle vztahů mezi elementy (sousední).
- ☐ Umím vysvětlit rozdíl mezi selektorem `A + B` a `A ~ B`.
- ☐ Umím vysvětlit, jak fungují selektory podle atributů.
- ☐ Pro výběr dle atributů znám různé možnosti zápisu (přesnou shodu a zda atribut obsahuje).

- ☐ Umím vysvětlit jak funguje pseudo-třída `:not`.
- ☐ Umím vysvětlit jak funguje pseudo-třída `:has`.

A.2.1 Praktická část

- ☐ Umím spočítat specifickou váhu CSS selektoru.
- ☐ Umím ověřit, zda je nějaká vlastnost dědičná.
- ☐ Umím určit, který styl bude použitý v případě konfliktu více pravidel.

- ☐ Umím vytvořit CSS selektor, který vybere všechny odstavce, které následují po nadpisu h2.
- ☐ Umím vytvořit CSS selektor, který vybere všechny odkazy, které jsou uvnitř elementu s třídou „menu“.
- ☐ Umím vytvořit CSS selektor, který vybere všechny inputy s atributem type nastaveným na „text“.

- [ ] Umím vytvořit CSS selektor, který vybere všechny obrázky, jejichž alt atribut obsahuje slovo „logo“.
- [ ] Umím vytvořit CSS selektor, který vybere všechny odstavce, které nemají třídu „highlight“.
- [ ] Umím vytvořit CSS selektor, který vybere všechny položky seznamu, které jsou prvními dětmi svého rodiče.

- [ ] Umím vytvořit CSS pravidlo, která za každý odkaz vloží za text ikonu šipky pomocí pseudo-elementu `::after`.

A.2.2 Bonusové části

- [ ] Pro výběr dle atributu znám i pokročilé možnosti zápisu (začíná na, končí na, obsahuje slovo, obsahuje hodnotu oddělenou mezerou).
- [ ] Umím vysvětlit, jak fungují další pseudo-elementy jako `::backdrop`, `::cue`, `::part`, `::slotted`.

A.3 JavaScript

- [ ] Vím, že existují různé typy jazyků.
- [ ] Chápu, co znamená, že jazyk je interpretovaný.
- [ ] Chápu, co znamená, že jazyk je kompilovaný.
- [ ] Chápu, co znamená, že jazyk je dynamicky typovaný.
- [ ] Chápu, co znamená, že jazyk je staticky typovaný.
- [ ] Chápu rozdíl mezi skriptovacím a programovacím jazykem.
- [ ] Chápu, co je to programovací paradigma.
- [ ] Vím, co je to objektově orientované programování.
- [ ] Vím, co je to funkcionální programování.
- [ ] Vím, co je to datově orientované programování.
- [ ] Vím, že nějaké jazyky jsou více paradigmové.

- [ ] Vím, co je to JavaScript a k čemu se používá.
- [ ] Vím, že JavaScript je interpretovaný jazyk.
- [ ] Vím, že JavaScript je dynamicky typovaný jazyk.
- [ ] Vím, že JavaScript je více paradigmový jazyk.
- [ ] Vím, že JavaScript je skriptovací jazyk.
- [ ] Vím, že JavaScript může být programovací jazyk.

- [ ] Vím, co je to ECMAScript.
- [ ] Vím, že JavaScript je implementace standardu ECMAScript.
- [ ] Vím, že nejdůležitější verze ECMAScript jsou ES5 a ES6.

- [ ] Vím, co je to runtime prostředí.
- [ ] Vím, jak spustit JavaScript v prohlížeči.
- [ ] Vím, že JavaScript má přístup od prohlížeče k různým API (DOM, Fetch a další).

- [ ] Vím, k čemu slouží Document Object Model.
- [ ] Vím, jak se v DOM vybírají elementy.
- [ ] Vím, jak se v DOM mění obsah a vlastnosti elementů.
- [ ] Vím, jak se v DOM přidávají a odebírají elementy.
- [ ] Vím, jak se v DOM přidávají události na elementy.
- [ ] Vím, co na události způsobí zavolání `preventDefault()`.

- [ ] Vím, že jde JavaScript spustit i mimo prohlížeč v jiném runtime prostředí (např. Node.js).
- [ ] Vím, co je to Node.js a k čemu se používá.
- [ ] Chápu, že v různých runtime prostředích mohou být dostupná různá API.
- [ ] Vím proč v Node.js není k dispozici např. API pro práci s DOM.

- [ ] Vím, jaké datové typy JavaScript má.
- [ ] Vím, jakými způsoby se dají vytvářet proměnné v JavaScriptu.
- [ ] Vím, jaké jsou rozdíly mezi `var`, `let` a `const`.
- [ ] Chápu rozdíl mezi hodnotami primitivních a referenčních datových typů.

- [ ] Vím, jaké operátory JavaScript má.
- [ ] Umím vysvětlit, na co se hodí binární operátory.
- [ ] Umím vysvětlit, na co se hodí logické operátory.
- [ ] Umím vysvětlit, na co se hodí přiřazovací operátory.
- [ ] Umím vysvětlit, na co se hodí aritmetické operátory.
- [ ] Umím vysvětlit, na co se hodí porovnávací operátory.
- [ ] Vím, jaké jsou rozdíly mezi operátory == a ===.
- [ ] Vím, jaký je rozdíl mezi operátory prefixovými a postfixovými (např. `++a` vs `a++`).

- [ ] Vím, jaké jsou základní konstrukce pro řízení toku programu v JavaScriptu.
- [ ] Vím, jak fungují podmínky (`if`, `else if`, `else`, `switch`).
- [ ] Vím, jak fungují cykly (`for`, `while`, `do while`, `for...in`, `for...of`).
- [ ] Chápu, k čemu slouží přerušení toku programu (`break`, `continue`, `return`).

- [ ] Vím, co je to funkce, jak se deklaruje a jak se volá.
- [ ] Vím, co jsou to parametry a argumenty funkce.
- [ ] Vím, co je to návratová hodnota funkce.
- [ ] Vím, že funkce nemusí přijímat parametry ani vracet hodnotu.
- [ ] Vím, jakými způsoby se dají funkce v JavaScriptu deklarovat.
- [ ] Vím, co jsou to anonymní funkce a kdy se používají.
- [ ] Vím, co jsou to šipkové funkce (arrow functions) a jaký je mezi nimi rozdíl oproti běžným funkcím.
- [ ] Vím, jak se dá použít rest opertár u funkcí a k čemu slouží.
- [ ] Vím, co je to rekurze a kdy se používá.

- [ ] Vím, jak se používá konzole pro výpis informací během vývoje.
- [ ] Vím, jaké funkce jsou připravené v knihovně Math pro práci s čísly.

- [ ] Umím na řetězci volat připravené metody pro práci s textem.
- [ ] Vím, k čemu slouží funkce `startWith()` a `endsWith()` u řetězců.
- [ ] Vím, k čemu slouží funkce `substring()` a `slice()` u řetězců.
- [ ] Vím, k čemu slouží funkce `replace()` a `replaceAll()` u řetězců.
- [ ] Vím, k čemu slouží funkce `toUpperCase()` a `toLowerCase()` u řetězců.
- [ ] Vím, k čemu slouží funkce `trim()`, `trimStart()` a `trimEnd()` u řetězců.
- [ ] Vím, k čemu slouží funkce `split()` u řetězců.
- [ ] Vím, k čemu slouží funkce `Number()`, `parseInt()` a `parseFloat()` pro převod na číslo.

- [ ] Vím, co je to struktura (či kolekce) v programování.
- [ ] Vím, jaké struktury jsou v JavaScriptu dostupné (pole, objekt, mapa, množina).
- [ ] Vím, jak se vytváří a používá pole v JavaScriptu.
- [ ] Vím, jak se vytváří a používá objekt v JavaScriptu.
- [ ] Vím, jak se vytváří a používá mapa (Map) v JavaScriptu.
- [ ] Vím, jak se vytváří a používá množina (Set) v JavaScriptu.
- [ ] Vím, k čemu slouží funkce `push`, `pop`, `shift`, `unshift` u polí.
- [ ] Vím, k čemu slouží funkce `slice`, `splice`, `indexOf`, `includes` u polí.
- [ ] Vím, k čemu slouží funkce `map`, `filter`, `reduce`, `find` u polí.
- [ ] Chápu rozdíl mezi použití funkce `forEach` a cyklů (`for`, `for...of`) pro iteraci přes pole.
- [ ] Vím, k čemu slouží funkce `sort`, `reverse` u polí.
- [ ] Vím jaké funkce nabízí množina (Set) pro práci s unikátními hodnotami.
- [ ] Vím jaké funkce nabízí mapa (Map) pro práci s klíč-hodnota páry.
- [ ] Vím, jak se v objektu nastavuje a získává vlastnost.
- [ ] Vím, že když vlastnost v objektu neexistuje, vrátí se `undefined`.
- [ ] Vím, jaké funkce nabízí objekt `Object` pro práci s objekty.
- [ ] Vím, že v objektu můžu definovat i funkce.

- [ ] Vím, jakými způsoby se může datum a čas ukládat v počítači.
- [ ] Vím, jak se v JS pracuje s datumem a časem.
- [ ] Vím, jak se vytváří instance `Date`.
- [ ] Vím, jaké metody nabízí objekt `Date` pro práci s datem a časem.

- [ ] Vím, co je to třída v objektově orientovaném programování.
- [ ] Vím, jak se v JavaScriptu deklaruje třída.
- [ ] Vím, jak se v JavaScriptu vytváří instance třídy.
- [ ] Vím, jak se v JavaScriptu deklaruje konstruktor třídy.
- [ ] Vím, jak se v JavaScriptu deklarují metody třídy.
- [ ] Vím, jak se v JavaScriptu deklarují vlastnosti třídy.
- [ ] Chápu, že vlastnosti v třídě nemusí být předem deklarované.
- [ ] Chápu, proč jsou konstruktory tříd důležité.
- [ ] Vím, jak funguje dědičnost tříd v JavaScriptu.
- [ ] Vím, k čemu slouží operátor `super` v kontextu tříd.

- [ ] Vím, co je to abstrakce v objektově orientovaném programování.

- [ ] Vím, co je to zapouzdření v objektově orientovaném programování.
- [ ] Vím, co je to polymorfismus v objektově orientovaném programování.
- [ ] Vím, jak v JS zjistit, že nějaká hodnota je instance určité třídy.

- [ ] Vím, co je to chyba (exception) v programování.
- [ ] Vím, jak v JavaScriptu zachytit chyby pomocí `try`, `catch`, `finally`.
- [ ] Vím, jak v JavaScriptu vyhodit vlastní chybu pomocí `throw`.
- [ ] Vím, proč se výjimky používají a kdy není vhodné je používat.
- [ ] Vím, jaké jsou běžné vestavěné typy chyb v JavaScriptu (např. `TypeError`, `ReferenceError`).
- [ ] Vím, proč v programu je často hlavní try-catch blok.

- [ ] Vím, co je to asynchronní programování.
- [ ] Vím, jaký je rozdíl mezi asynchronním a synchronním programováním.
- [ ] Vím, jaký je rozdíl mezi asynchronním a vícevláknovým programováním.
- [ ] Vím, co je to AJAX a s čím přišel.
- [ ] Vím, jak funguje `setTimeout` a `setInterval`.
- [ ] Vím, co jsou to callbacky a jak se používají.
- [ ] Vím, co je to slib (Promise) a jak se používá.
- [ ] Vím v jakých stavech může být slib.
- [ ] Vím, co jsou to funkce `then`, `catch` u slibů.
- [ ] Vím, že si můžu vytvořit vlastní slib pomocí konstruktoru `Promise`.
- [ ] Vím, jako pomocné funkce pro práci se sliby existují u `Promise`.
- [ ] Vím, co je to async/await a jak se používá.
- [ ] Vím, co to je tzv. syntax sugar (syntaktický cukr).

- [ ] Vím, jak se používá Fetch API pro práci s HTTP požadavky.
- [ ] Vím, jaké k čemu slouží na odpovědi `.json()`, `text()`, `blob()` a další metody.
- [ ] Vím, jaké informace můžu zadat v objektu s nastavením požadavku (metoda, hlavičky, tělo, …).

- [ ] Vím, co je to npm a k čemu slouží.
- [ ] Vím, jak se inicializuje nový npm projekt.
- [ ] Vím, jak se instalují balíčky pomocí npm.
- [ ] Vím, jak co je to soubor `package.json` a k čemu slouží.
- [ ] Vím, jak se spouští skripty definované v `package.json`.
- [ ] Vím, jak se instalují balíčky jako vývojové závislosti.
- [ ] Vím, co se nachází v souboru `package-lock.json`.
- [ ] Vím, jak se liší zapsání verze balíčku s `^` a `~`.

- [ ] Vím, jak se v JS propojují soubory.
- [ ] Vím, jak se používá `import` a `export` pro práci s moduly.
- [ ] Vím, co je to výchozí export (default export).
- [ ] Vím, jak můžu importovaný modul přejmenovat.

- [ ] Vím, jaké API jsou k dispozici v Node.js.
- [ ] Vím, jak se v Node.js pracuje se souborovým systémem pomocí modulu `fs`.
- [ ] Vím, jak se v Node.js pracuje s cestami k souborům pomocí modulu `path`.
- [ ] Vím, jak se v Node.js pracuje s procesy pomocí modulu `process`.

A.3.1 Praktická část

- [ ] Umím spustit JavaScript kód v rámci prohlížeče.
- [ ] Umím spustit JavaScript kód v rámci běhového prostředí Node.js.
- [ ] Umím použít základní datové typy v JavaScriptu.
- [ ] Umím použít operátory v JavaScriptu.
- [ ] Umím vytvořit a používat proměnné v JavaScriptu.
- [ ] Umím vytvořit a používat pole a objekty v JavaScriptu.
- [ ] Umím vytvořit a používat mapy a množiny v JavaScriptu.
- [ ] Umím vytvořit a používat funkce v JavaScriptu.
- [ ] Umím rozhodnout, zda použít primitivní nebo referenční datový typ.
- [ ] Vím, kdy je vhodné použít asynchronní programování.

- [ ] Umím převést text na číslo a naopak.
- [ ] Umím použít cykly pro detekci, zda je číslo prvočíslo.
- [ ] Umím vytvořit funkci pro výpočet faktoriálu čísla pomocí rekurze.
- [ ] Umím vytvořit třídu pro reprezentaci osoby s vlastnostmi jméno a věk a metodou pro představení se.
- [ ] Umím vytvořit instanci třídy a použít její metody pro příklad výše uvedené třídy osoba.
- [ ] Umím zachytit chybu při dělení nulou pomocí try-catch.
- [ ] Umím vytvořit a použít slib při zavolání HTTP požadavku pomocí Fetch API.

- [ ] Umím založit nový npm projekt a nainstalovat do něj balíček.
- [ ] Umím vytvořit více souborů s kódem a propojit je pomocí modulů (import/export).
- [ ] Umím vytvořit jednoduchý skript v Node.js, který čte obsah souboru a vypíše ho do konzole.

A.3.2 Bonusové části

- [ ] Chápu, že není možné každý jazyk rozdělit do jedné kategorie a že některé jazyky mají vlastnosti více kategorií.
- [ ] Chápu rozdíl mezi vysokoúrovňovým a nízkoúrovňovým jazykem.
- [ ] Chápu, co znamená, že jazyk je imperativní.
- [ ] Chápu, co znamená, že jazyk je deklarativní.

- [ ] Vím o historii JavaScriptu a jeho vývoji.
- [ ] Vím, co je to zpětná kompatibilita.
- [ ] Vím, proč je zpětná kompatibilita důležitá pro webové technologie.
- [ ] Vím, jaké jsou hlavní rozdíly mezi ES5 a ES6.
- [ ] Vím, že existují i jiné implementace ECMAScript (např. JScript, ActionScript).

- [ ] Vím rozdíl mezi bubble a capture fází událostí v DOM.
- [ ] Vím, jak můžu zastavit šíření události v DOM.
- [ ] Vím, jak zachytiti událost v capture fázi.

- [ ] Vím, co je to hoistování (hoisting).

- [ ] Vím, jak se dá optimalizovat rekurze pomocí tail call optimization.
- [ ] Chápu, kdy je vhodné použít rekurzi místo iterace.
- [ ] Chápu rizika spojená s příliš hlubokou rekurzí (stack overflow).

- [ ] Umím destrukturing u polí.
- [ ] Umím destrukturing u objektů.

- [ ] Vím, jak funguje spojování souborů pomocí způsobu CommonJS (require, module.exports).
- [ ] Vím, proč jsou default export problematické.

- [ ] Vím, jak se debuguje kód v prohlížeči pomocí vývojářských nástrojů.
- [ ] Vím, jak nastavit breakpointy v kódu.
- [ ] Vím, jak spustit pomocí Node profilování výkonu skriptu.

A.4 TypeScript

- [ ] Vím, co je to TypeScript a k čemu se používá.
- [ ] Vím, že TypeScript je nadmnožina JavaScriptu.
- [ ] Vím, že TypeScript přidává do JavaScriptu statické typování.
- [ ] Vím, že TS se musí před použitím přeložit do JavaScriptu.
- [ ] Vím, že jeden z kompileru TypeScriptu je `tsc`.
- [ ] Vím, jak se inicializuje nový TypeScript projekt.
- [ ] Vím, k čemu slouží soubor `tsconfig.json`.
- [ ] Vím, jaké základní datové typy TypeScript má.
- [ ] Chápu, proč TS vznikl a jaké problémy řeší.
- [ ] Vím, jaké výhody TS přináší k vývoji v IDE.

- [ ] Chápu rozdíl mezi implicitním a explicitním typováním.
- [ ] Vím, jak se v TS deklarují proměnné s typem.
- [ ] Chápu význam operátoru `as`.
- [ ] Chápu, že TS podporuje stejné typy jako JS.
- [ ] Chápu, že nějaké TS typy se po zkompilování do JS ztratí a nějaké zůstanou.

- [ ] Chápu, co je to typ `any` a kdy se používá.
- [ ] Chápu, co je to typ `unknown` a kdy se používá.
- [ ] Chápu, co je to typ `void` a kdy se používá.

- [ ] Chápu, co je to typ `never` a kdy se používá.

- [ ] Vím, že je možné definovat vlastní typy v TS.
- [ ] Vím, jak se v TS deklaruje interface.
- [ ] Vím, jak se v TS deklaruje typ pomocí `type`.
- [ ] Vím, jak se v TS deklaruje enum.

- [ ] Vím, jak se v TS deklaruje třída s typovanými vlastnostmi a metodami.
- [ ] Vím, že TS přidává spoustu OOP funkcionalit, které v JS nejsou.
- [ ] Vím, jaký je rozdíl mezi modifikátorem `public`, `private` a `protected`.
- [ ] Vím, jak se v TS deklarují abstraktní třídy a metody.
- [ ] Vím, co platí pro abstraktní třídy.
- [ ] Vím, co dělá modifikátor `readonly`.
- [ ] Vím, jak funguje dědičnost tříd v TS.
- [ ] Vím, že třída v TS může implementovat více rozhraní.

- [ ] Vím, k čemu slouží pomocné typy jako `Partial`, `Readonly`, `Record`, `Pick`, `Omit`.
- [ ] Vím, jaký je rozdíl mezi nepovinným parametrem a parametrem s možnou hodnotou `undefined`.

- [ ] Vím, co je to generika a proč se používají.
- [ ] Vím, jak se v TS deklaruje generická funkce.
- [ ] Vím, jak se v TS deklaruje generická třída.
- [ ] Vím, jak se v TS deklaruje generické rozhraní.

- [ ] Vím, jak se s TS pracuje na straně klienta (v prohlížeči).
- [ ] Vím, co je to module bundler a proč se používá.
- [ ] Vím, co je to Vite a jak se používá s TS.
- [ ] Vím, co je to hot reloading / hot module replacement.
- [ ] Vím, co je to single page application.

A.4.1 Praktická část

- [ ] Umím vytvořit nový TypeScript projekt.
- [ ] Umím zkompilovat TypeScript kód do JavaScriptu.
- [ ] Umím deklarovat proměnné s typy.
- [ ] Umím vytvořit funkci s typovanými parametry a návratovou hodnotou.
- [ ] Umím vytvořit vlastní typ pomocí `interface` a `type`.
- [ ] Umím vytvořit výčtový typ (enum) a použít ho.
- [ ] Umím vytvořit třídu s typovanými vlastnostmi a metodami.
- [ ] Umím použít modifikátory přístupu (`public`, `private`, `protected`) v třídě.
- [ ] Umím vytvořit generickou funkci.
- [ ] Umím vytvořit generickou třídu.
- [ ] Umím nastavit TypeScript projekt pro použití ve Vite.

- [ ] Dokážu vytvořit jednoduchou aplikaci v TypeScriptu, která běží v prohlížeči pomocí Vite.

A.4.2 Bonusové části

- [ ] Vím, co dělá `target` v `tsconfig.json` a jaké jsou běžné hodnoty.
- [ ] Vím, jak interně fungují výčtové typy a na co se zkompilují.
- [ ] Vím, jak můžeme typově definovat funkci.
- [ ] Vím jak přesně funguje spojování souborů/modulů v TypeScriptu.
- [ ] Umím založit Vite s multi page application podporou v TypeScriptu.
- [ ] Dokážu vytvořit v TS typování pro existující JavaScript knihovnu.

A.5 Design

A.6 CSS preprocesory

A.7 Verzování

A.8 Frameworky

A.9 Backend

Příloha B.

Seznamy

B.1 Seznam kódových bloků

B.2 Seznam vlastností CSS

B.3 Seznam CSS selektorů