Webové inženýrství

4. ročník
Skripta

Autor
Ing. Matěj Cajthaml

Verze
14. 12. 2025

Obsah

Seznam zkratek

2FA	Two-Factor Authentication
ABAC	Attribute-Based Access Control
ACL	Access Control List
AI	Artificial Intelligence
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
APK	Android Package
CD	Continuous Delivery / Continuous Deployment
CDN	Content Delivery Network
CI	Continuous Integration
CLI	Command Line Interface
CLS	Cumulative Layout Shift
CSS	Cascading Style Sheets
DOM	Document Object Model
DTO	Data Transfer Object
DTO	Data Transfer Object
E-E-A-T	Experience, Expertise, Authoritativeness, Trustworthiness
FCP	First Contentful Paint
FID	First Input Delay
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IDF	Inverse Document Frequency
INP	Interaction to Next Paint
IPA	iOS App Store Package
IPC	Inter-Process Communication
IaC	Infrastructure as Code
JS	JavaScript
JSON	JavaScript Object Notation
JSX	JavaScript XML
JWT	JSON Web Token
LCP	Largest Contentful Paint
MFA	Multi-Factor Authentication
OAuth	Open Authorization
OOP	Object-Oriented Programming
PBAC	Permission-Based Access Control
PBAC	Policy-Based Access Control
PWA	Progressive Web Application
RBAC	Role-Based Access Control
REST	Representational State Transfer
RPC	Remote Procedure Call
SAFe	Scaled Agile Framework
SDK	Software Development Kit
SEO	Search Engine Optimization
SGE	Search Generative Experience
SPA	Single Page Application
SSE	Server-Sent Events
SSL	Secure Sockets Layer
SSO	Single Sign-On
SSR	Server-Side Rendering
TCP	Transmission Control Protocol
TF	Term Frequency

TLS Transport Layer Security

TS TypeScript

UDP User Datagram Protocol

VDOM Virtual Document Object Model

WBA Webové applikace

WS WebSocket

WSS WebSocket Secure

WebGL Web Graphics Library

XML eXtensible Markup Language

XP Extreme Programming

Předmluva

1.1 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 a třetího ročníku, kdy jste se s nimi (a možná semnou) poprvé setkali.

Tento text je určen především studentům 4. 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 různorodou paletu témat, které vám třeba ani po prvním přečtení nebudou vzhledem k webovým stránkám dávat smysl. 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.

1.2 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.

1.3 Jak používat skripta

Skripta slouží jako podpůrný materiál k předmětu MVOP Webové inženýrství ve čtvrtém 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.

1.4 Návaznost

Celý předmět a i tato skripta zcela navazují na předchozí znalosti, které jste měli získat v předmětech Webové aplikace ve druhém ročníku a Webové inženýrství ve třetím ročníku. Skoro ve všech částech se tyto základy předpokládají a často se referuje na již zmíněné koncepty.

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

Projekty a jejich řízení

2.1 Motivace

Projekty jsou základní jednotkou práce ve vývoji softwaru. Každý projekt má svůj cíl, rozsah a časový rámec. Projekty mohou mít různé formy, od malých úkolů až po rozsáhlé programy. Projekty často zahrnují spolupráci mezi různými týmy a jednotlivci, což vyžaduje efektivní komunikaci a koordinaci. Proto vznikly různé metodiky a nástroje pro řízení projektů, které nám pomáhají dosáhnout našich cílů efektivněji a s menším rizikem.

Tato celá část se zabývá tím nejmenším rozumným celkem, kterým se opravdu každý vývojář setká. Pokročilejší a hlubší porozumění projektům je důležité, ale bohužel se to do tohoto učebního materiálu nevešlo. Doporučuji proto prostudovat další zdroje, které se zabývají řízením projektů a metodikami vývoje softwaru.

2.2 Projekty

Správa projektů zahrnuje mnohem více než jen samotné řízení. Při práci na projektech se věnujeme několika klíčovým oblastem:

- Analýze,
- Návrhu,
- Implementaci,
- Testování,
- Údržbě (a nasazení).

Kromě toho se zabýváme správou požadavků a problémů (issues) a správou souborů pomocí nástrojů jako je Git.

2.2.1 Analýza

Analýza je proces, který předchází samotnému vývoji. Cílem je zjistit, čeho má systém dosáhnout, jaké požadavky musí splňovat a které funkce jsou pro systém klíčové. Analytici často zjišťují tyto informace, přičemž vývojáři nejsou nutně zapojeni do této fáze. Často se jedná o speciální roli, která se zaměřuje na komunikaci s klienty a uživateli.

Důkladná analýza je nezbytná pro úspěch projektu, protože definuje základy pro všechny následující kroky. V tom nejzákladnějším případě konkrétně zjišťujeme:

- Kdo jsou uživatelé systému?
- Jaké jsou jejich potřeby a očekávání?
- Jaké funkce a vlastnosti by měl systém mít?
- Jaké jsou technické a obchodní požadavky?
- Jaké jsou možné omezení a rizika?
- Jaké jsou hlavní scénáře použití systému?
- S jakými dalšími systémy bude náš systém komunikovat?
- S jakými daty bude systém pracovat a jak budou tato data strukturována?
- A mnoho dalších.

Nejčastěji se setkáme s **funkčními požadavky** (co systém má dělat) a **nefunkčními požadavky** (jak dobře to má dělat, např. výkon, bezpečnost, použitelnost). Datům, se kterými pracuje systém, se definují v **datovém modelu** (data model). Pro lepší pochopení požadavků a scénářů použití se často vytvářejí **případy užití** (use cases) diagramy, které vizuálně znázorňují interakce mezi uživateli a systémem.

2.2.2 Návrh

Návrh zahrnuje definování architektury systému, tedy jak budou jednotlivé části systému komunikovat. V této fázi se také vybírají technologie, které budou použity. Rozhoduje se taktéž o konkrétní struktuře ukládání dat. Důležité je také detailně popsat komponenty a jejich integraci do celkového systému.

2.2.2.1 Architektura

Architektura může být rozdělena na logickou a fyzickou. Logická architektura určuje rozdělení aplikace a definování vrstev, jako jsou závislosti, MVC (Model-View-Controller), middleware a další. Fyzická architektura se pak zaměřuje na rozdělení systému na servery, například tenký klient, tlustý klient nebo cluster. Cílem architektury je vytvořit srozumitelný, rozšiřitelný a udržovatelný systém.

V rámci návrhu se často bavíme o tzv. **vrstvení** (layering), což je způsob, jak rozdělit aplikaci do různých vrstev podle jejich funkcí. Nejčastěji je řešíme v rámci aplikace na klientovi a i na serveru. Vrsty reprezentují různé úrovně abstrakce a zodpovědnosti, což usnadňuje vývoj, testování a údržbu aplikace. Například, pokud se v budoucnu rozhodneme změnit databázi, můžeme to udělat bez nutnosti měnit logiku aplikace, pokud je správně oddělena v různých vrstvách. Vrsty omezují přímou komunikaci mezi komponentami. Při změnách v jedné vrstvě by neměly být nutné změny v jiných vrstvách, pokud jsou správně navrženy.

2.2.3 Implementace

Implementace je samotná práce na vývoji systému. V této fázi se kód rozděluje na konkrétní třídy a metody, přičemž je důležité omezit opakování kódu a zajistit jeho udržitelnost. K tomu často používáme **návrhové vzory**, které pomáhají strukturovat kód efektivněji a přehledněji. Správná implementace může výrazně snížit riziko vzniku „špagetového kódu“, který je obtížně udržovatelný a rozšiřitelný.

2.2.3.1 Přenášení dat

V době implementace často pracujeme s přenášením dat mezi různými částmi systému nebo mezi různými systémy. Na to používáme různé protokoly a formáty, jako jsou JSON, XML nebo protokoly jako HTTP (př. REST), GraphQL a další, kterými se budeme zaobírat v Kapitola 5.1 a dalších. Použití těchto protokolů je však triviální – vždy bychom se měli snažit o co nejjednodušší a nejefektivnější přenos dat.

Důležité pro projekt je však správné určení obsahu dat, která se přenáší. Každá část systému může být napsána v různých jazycích a může mít různé datové struktury[1]. Skoro každý projekt proto potřebuje definice datových struktur, které se budou přenášet. Těm se říká **Data Transfer Objects** (DTO) a jsou to jednoduché objekty, které obsahují pouze data bez jakékoliv logiky.

DTO v sobě často obsahují pouze základní datové typy, jako jsou řetězce, čísla, pole a další DTO. Tedy není tam nic složitého, jako jsou složené objekty, reprezentace, funkce či jiné logiky.

[1]Představte si jen způsoby, kterými lze reprezentovat čas v jazycích.

DTO nám pomáhají zajistit, že data předány mezi různými částmi systému jsou konzistentní a správně strukturovaná. Při změně datové struktury v jedné části systému je pak jednodušší aktualizovat i ostatní části, protože změny jsou centralizovány v definici DTO.

typescript

Ukázka jednoduchého DTO v TypeScriptu.

```typescript
interface UserDTO {
  id: number;
  name: string;
  email: string;
  roles: string[];
  createdAt: string; // ISO 8601 date string
}
```

Důležité vlastnosti DTO:

- **Jednoduchost**: DTO obsahují pouze data, žádnou obchodní logiku ani metody (kromě základních getterů/setterů).
- **Serializovatelnost**: DTO musí být snadno převoditelná do formátů jako JSON, XML nebo binárních formátů pro přenos po síti.
- **Immutabilita**: Často jsou DTO navrženy jako neměnné objekty, což zvyšuje bezpečnost a předcházívá neočekávaným změnám dat.
- **Validace**: DTO často obsahují anotace nebo validační pravidla, která zajišťují správnost dat před jejich zpracováním.

DTO se často liší od interních datových modelů aplikace. Například databázový model může obsahovat citlivé informace (hesla, interní ID), které nechceme přenášet. DTO tak slouží jako „filtrovací vrstva“, která definuje, jaká data jsou bezpečná pro přenos.

typescript

Rozdíl mezi databázovým modelem a DTO.

```typescript
// Databázový model
class User {
  id: number;
  name: string;
  email: string;
  passwordHash: string; // Nesmí být v DTO!
  internalNotes: string; // Nesmí být v DTO!
  roles: Role[];
}

// DTO pro API
interface UserDTO {
  id: number;
  name: string;
  email: string;
  roles: string[]; // Zjednodušené oproti Role[]
}
```

Při návrhu DTO je důležité:

- **Verzování**: API evoluje a DTO se mohou měnit. Je potřeba podporovat starší verze pro zachování kompatibility.

- **Granularita**: Rozhodnout, zda vytvořit jedno univerzální DTO nebo více specifických pro různé use cases.
- **Transformace**: Definovat jasné způsoby převodu mezi interními modely a DTO (mapping).
- **Dokumentace**: DTO by měly být dobře zdokumentované, včetně formátů, omezení a příkladů použití.

2.2.4 Testování

Testování je klíčovou součástí vývoje softwaru, která zajišťuje kvalitu a spolehlivost aplikace. Často již máme vytvořenou aplikaci, ale je potřeba ověřit, že funguje správně a bez chyb. K tomu slouží různé typy testů, které pokrývají různé úrovně a aspekty aplikace.

Testování může být prováděno manuálně nebo automatizovaně. Automatizované testy jsou preferovány, protože umožňují rychlé a opakovatelné ověření funkčnosti aplikace při každé změně kódu. Ty ale nemohou zcela nahradit manuální testování, které je často potřeba pro ověření uživatelského rozhraní a uživatelského zážitku.

O testování se budeme více bavit v Kapitola 2.4.

2.2.5 Údržba

Aplikace je hotová. Nyní je potřeba ji nasadit. Nasazení je velmi individuální věc v rámci každého projektu. Někdy je to jen zkopírování souborů na server, jindy je to složitý proces zahrnující databázové migrace, konfigurace serverů a další kroky. Nebo to může být dokument, který říká někomu jinému, co pro projekt je potřeba udělat.

Aplikace se vprůběhu času mění a vyvíjí. Je potřeba ji aktualizovat, opravovat chyby a přidávat nové funkce. To vše spadá pod údržbu aplikace, která je často nejdelší fází životního cyklu softwaru. Dle typu projektu a smlouvy mezi klientem může být údržba různě náročná a časově omezená.

Při velkých změnách v aplikaci je potřeba dbát na znovutestování a ověření, že nové funkce neporušily stávající funkcionalitu.

2.3 Metodiky řízení projektů

Metodika vývoje je způsob, jakým přistupujeme k vývoji softwaru. Každý projekt je jedinečný a vyžaduje specifický přístup, ale je důležité mít stanovený standardní postup, i když se může měnit. Důraz na opakovatelnost procesů je klíčový, protože umožňuje lépe plánovat a předvídat vývojové kroky.

2.3.1 Vodopádový model

Klasické metodiky vývoje jsou detailně propracované a kladou velký důraz na dokumentaci. Tyto metodiky jsou často považovány za nadměrně náročné pro malé projekty. Mezi známé klasické metodiky patří Waterfall (vodopádový model), Unified Process a Modern Structures Analysis.

Waterfall funguje dle pevně daných fází, které na sebe navazují. Mezi fáze patří již zmíněná analýza, návrh, implementace, testování a údržba. Tyto fáze na sebe navazují a každá fáze musí být dokončena před začátkem další.

Pokud se v nějaké fázi zjistí, že je něco špatně navrhlé, nebo nevhodně analyzované, je nutné se vrátit do dané fáze a provést změny. To může být časově náročné a drahé, protože často znamená přepracování již hotové práce.

Obecně je nutné velmi dobře mít promyšlenou celou aplikaci z analýzy a návrhu.

Waterfall model se používá zejména v projektech, který nemají problém vložit hodně času do analýzy a návrhu. Typicky se jedná o projekty, které mají jasně dané požadavky a není potřeba je často měnit. Dobrou ukázkou jsou například státní zakázky, kde je potřeba mít vše velmi dobře zdokumentované a schválené.

2.3.2 Agilní metodiky

Agilní metodiky jsou flexibilní a zaměřují se na dosažení výsledků. Cílem je co nejrychleji dodat první verzi produktu a postupně ji vylepšovat. Mezi populární agilní metodiky patří SCRUM a Extrémní programování, například Test Driven Development (TDD).

Agilní metody kladou důraz na spolupráci mezi týmy, rychlou adaptaci na změny a pravidelné dodávání funkčního softwaru. Tento přístup umožňuje rychle reagovat na zpětnou vazbu od uživatelů a přizpůsobit se měnícím se požadavkům.

Jsou velmi populární v moderním vývoji softwaru. Jeden z dalších důvodů je, že i pro samotné vývojáře je mnohem příjemnější pracovat v agilním prostředí, kde mají větší svobodu a flexibilitu. Protože postup není tak striktní a rigidní jako u klasických metodik.

2.3.2.1 Scrum

Scrum je nejpopulárnější agilní metodika pro řízení projektů vývoje softwaru. Je založen na iterativním a inkrementálním přístupu, kde se práce rozděluje do krátkých časových úseků nazývaných **sprinty**. Scrum klade důraz na pravidelnou komunikaci, transparentnost a přizpůsobivost.

Základní principy Scrumu:

- **Iterativnost**: Práce je rozdělena do krátkých sprintů (obvykle 1-4 týdny).
- **Transparentnost**: Všichni členové týmu mají přehled o stavu projektu.
- **Inspekce**: Pravidelné kontroly pokroku a kvality práce.
- **Adaptace**: Schopnost rychle reagovat na změny a zpětnou vazbu.

Scrum definuje jasná pravidla a ceremonie (rituály), které pomáhají týmu fungovat efektivně:

- **Sprint Planning**: Plánování sprintu, kde tým definuje, co se bude v nadcházejícím sprintu realizovat.
- **Daily Scrum**: Krátké denní schůzky (obvykle 15 minut), kde členové týmu sdílejí pokrok a případné překážky.
- **Sprint Review**: Prezentace dokončené práce na konci sprintu pro stakeholdery.
- **Sprint Retrospective**: Reflexe týmu o tom, co fungovalo dobře a co lze zlepšit.

Klíčovým konceptem je **Product Backlog** – seznam všech požadavků a funkcí, které je potřeba implementovat, seřazených podle priority. Z tohoto seznamu se pro každý sprint vybírá **Sprint Backlog** – konkrétní úkoly, které se budou řešit.

Role ve Scrumu

Scrum definuje tři klíčové role, z nichž každá má specifické odpovědnosti:

Product Owner:
- Zodpovídá za definování požadavků a prioritizaci Product Backlogu.
- Komunikuje s klienty a stakeholdery.
- Rozhoduje o tom, jaké funkce budou implementovány a v jakém pořadí.
- Přijímá nebo odmítá výsledky sprintů.
- Musí být dostupný pro tým a schopen rychle odpovídat na otázky.

Scrum Master:
- Facilitátor a kouč týmu, ne nadřízený.
- Zajišťuje dodržování Scrum procesů a pravidel.
- Pomáhá odstranit překážky, které brání týmu v práci.
- Moderuje Scrum ceremonie.
- Chrání tým před vnějšími rušivými vlivy.
- Podporuje komunikaci mezi všemi zúčastněnými stranami.

Development Team (Vývojářský tým):
- Samoorganizující se tým odborníků, kteří vytváří produkt.
- Obvykle 3-9 členů s různými dovednostmi.
- Kolektivně zodpovídá za dodání funkčního produktu na konci každého sprintu.
- Odhad úsilí potřebného pro implementaci funkcí.
- Členové si mezi sebou rozdělují úkoly a spolupracují.

> ⚠ **Časté chyby ve Scrumu**
>
> Častou chybou při implementaci Scrumu je, že se Scrum Master chová jako projektový manažer nebo nadřízený. Scrum Master by měl být spíše služebník týmu, který pomáhá odstraňovat překážky a zlepšovat procesy. Podobně Development Team by měl být samoorganizující, což znamená, že si sám rozděluje práci a rozhoduje o technických řešeních.

Scrum artefakty

Scrum definuje tři hlavní artefakty:

Product Backlog:
- Dynamický seznam všech požadovaných funkcí, vylepšení a oprav.
- Spravuje ho Product Owner.
- Položky jsou seřazeny podle priority a obchodní hodnoty.
- Obsahuje uživatelské příběhy (user stories) a jejich odhady složitosti.

Sprint Backlog:
- Seznam úkolů vybraných z Product Backlogu pro aktuální sprint.
- Vytváří se během Sprint Planning.
- Vlastní ho Development Team.
- Může být během sprintu upravován týmem.

Increment:
- Funkční verze produktu vytvořená během sprintu.

- Musí být „hotová“ podle Definition of Done.
- Potenciálně nasaditelná do produkce.
- Každý sprint by měl přidat hodnotu k předchozímu Incrementu.

Varianty Scrumu

Scrum má často velký problém se škálováním na větší týmy a organizace. Rozdělení na týmy často není jednoduché a je potřeba zajistit koordinaci mezi nimi. Proto existuje několik variant a rozšíření Scrumu pro různé situace.

Scrum of Scrums:
- Používá se pro koordinaci více Scrum týmů na velkých projektech.
- Zástupci jednotlivých týmů se pravidelně setkávají.
- Řeší závislosti a synchronizaci mezi týmy.

SAFe (Scaled Agile Framework):
- Komplexní framework pro škálování agilních metodik ve velkých organizacích.
- Kombinuje Scrum s dalšími praktikami pro enterprise úroveň.
- Definuje role a procesy na několika úrovních organizace.

Scrumban:
- Kombinace Scrum a Kanban metodik.
- Zachovává Scrum ceremonie, ale používá Kanban board pro vizualizaci práce.
- Flexibilnější přístup k délce sprintů a plánování.

Large-Scale Scrum (LeSS):
- Framework pro škálování Scrumu na více týmů.
- Snaží se zachovat jednoduchost původního Scrumu.
- Definuje minimum dalších rolí a procesů.

V praxi se často Scrum přizpůsobuje specifickým potřebám týmu a organizace. Důležité je zachovat základní principy a hodnoty, i když se některé ceremonie nebo pravidla mírně upraví.

2.3.2.2 Kanban

Kanban je další populární agilní metodika, která se zaměřuje na vizualizaci práce a optimalizaci toku úkolů. Kanban board je základním nástrojem, který zobrazuje stav jednotlivých úkolů v různých fázích (např. To Do, In Progress, Done). Kanban klade důraz na omezení rozpracovanosti (WIP – Work In Progress), což pomáhá týmu soustředit se na dokončení úkolů předtím, než začnou nové. Kanban je flexibilní a nevyžaduje pevně dané role nebo ceremonie jako Scrum.

2.3.2.3 Extreme Programming

Extreme Programming (XP) je agilní metodika, která klade velký důraz na technické praktiky a kvalitu kódu. XP zahrnuje spoustu různých praktik, které se zaměřují na specifické aspekty vývoje softwaru.

Mezi klíčové praktiky XP patří:
- **Pair Programming**: Dva vývojáři pracují společně na jednom počítači, což zvyšuje kvalitu kódu a usnadňuje sdílení znalostí.

- **Test Driven Development**: Vývoj začíná psaním testů, které definují požadovanou funkčnost, a poté se píše kód, který tyto testy splňuje.
- **Refactoring**: Pravidelné zlepšování struktury kódu bez změny jeho chování, což zvyšuje čitelnost a udržovatelnost.
- **Continuous Integration**: Časté slučování kódu do hlavní větve, což umožňuje rychlé odhalení a opravu chyb.
- **Small Releases**: Časté vydávání malých, funkčních verzí softwaru, což umožňuje rychlou zpětnou vazbu od uživatelů.
- **Simple Design**: Snažení se o co nejjednodušší řešení, které splňuje aktuální požadavky, bez zbytečné složitosti.
- A další.

2.3.3 Další metodiky

Mezi další populární metodiky patří například Lean Software Development, Crystal, Dynamic Systems Development Method (DSDM) a další.

Důležité je si uvědomit, že žádná metodika není univerzální a vždy je potřeba přizpůsobit přístup konkrétním potřebám týmu a projektu. Proto jsou v různých firmách často tvořeny hybridní přístupy, které kombinují prvky různých metodik, a musíte se naučit pracovat s nimi.

2.4 Testování

Při testování se snažíme zjistit, zda aplikace funguje správně. Není možné však zjistit, zda je aplikace bezchybná, protože by to znamenalo otestovat všechny možné scénáře a kombinace vstupů, což je prakticky nemožné. Místo toho se snažíme najít chyby a problémy, které by mohly ovlivnit uživatele.

Vývojem softwaru, kde se snažíme elimitovat chyby, můžeme vybudovat důvěru uživatelů v náš produkt.

Úrovně testování:

- **Jednotkové testy** (Unit Tests): Testují jednotlivé funkce nebo metody v izolaci.
- **Integrační testy** (Integration Tests): Testují interakce mezi různými částmi systému.
- **Systémové testy** (System Tests): Testují celý systém jako celek.
- **Akceptační testy** (Acceptance Tests): Testují, zda systém splňuje požadavky a očekávání uživatelů.

Typy testů:

- **Funkční testy** (Functional Tests): Testují konkrétní funkce systému.
- **Nefunkční testy** (Non-Functional Tests): Testují aspekty jako výkon, bezpečnost, použitelnost.
- **Regresní testy** (Regression Tests): Testují, zda nové změny neporušily stávající funkčnost.
- **Exploratory Tests**: Manuální testování bez předem definovaných scénářů, zaměřené na objevování nečekaných chyb.

2.4.1 Automatické testování

Automatické testování je proces, při kterém jsou testy prováděny automaticky pomocí speciálních nástrojů a skriptů. Automatické testy jsou rychlé, opakovatelné a mohou být integrovány do vývojového procesu. Automatické testy by měly být spouštěny při každé změně kódu (resp. balíku změn).

Testy píšeme často vedle kódu, který testují. Při změně či přidání funkcionality otestujeme funkčnost a přidáme nové testy.

Do automatického testování taktéž vkládáme chyby, které se za vývoje staly tak, aby se zajistilo, že se nevrátí.

Mezi nejpopulárnější nástroje pro automatické testování patří JUnit, NUnit, pytest, Mocha, Jasmine, Selenium a další.

2.4.2 Uživatelské testování

Uživatelské testování je proces, při kterém jsou testy prováděny skutečnými uživateli aplikace. Tento typ testování je důležitý, protože nám umožňuje získat zpětnou vazbu od skutečných uživatelů a zjistit, jak aplikace funguje v reálném prostředí. Uživatele často bereme z různých skupin, aby pokryli různé scénáře použití.

Testování lze provádět různými způsoby, například pomocí A/B testování, uživatelských průzkumů, fokusních skupin nebo beta testování.

Často, pokud s uživateli jednáme prezenčně, tvoříme tzv. **testovací scénáře** (test cases), což jsou předem definované cíle, které má uživatel splnit. Tyto scénáře nám pomáhají strukturovat testování a zajistit, že pokryjeme všechny důležité funkce aplikace.

V rámci webových testů nejčastěji používáme A/B testování, což je metoda, při které jsou uživatelé náhodně rozděleni do dvou skupin. Jedna skupina vidí původní verzi aplikace (A) a druhá skupina vidí novou verzi (B). Tímto způsobem můžeme porovnat, jak různé verze aplikace fungují a která verze je pro uživatele lepší – měříme např. jejich interakce, konverze nebo jiné metriky.

Mimo klasické kódové testování v rámci webů používáme ještě i testování samotných vytvořených aplikací. Například knihovna Cypress umožňuje psát testy, které simulují chování uživatelů v prohlížeči – píšeme jednotlivé kroky (klik, vyplnění formuláře, navigace) a ověřujeme, že aplikace reaguje správně.

2.4.3 Analýza kódu

Mimo aktivní testování (jak automatické, tak uživatelské) je důležité také analyzovat kód. Během automatického sledování kódu totiž můžeme zjistit různé problémy, které mohou zamezit správnému fungování aplikace. Mimo to můžeme vyžadovat určité standardy psaní kódu, které nám pomohou udržet kód čistý a přehledný.

> ⚠ **Problém nekonzistentního stylu kódu**
>
> Častým problémem v projektech je, že si vývojáři nenastaví správné konvence psaní kódu. Poté se děje to, že každý píše svým stylem, což vede ve verzování k častým konfliktům, protože při úpravách jednoho řádku se změní například i odsazení dalších řádků. To může být řešeno pomocí nástrojů, které automaticky formátují kód podle předem definovaných pravidel. Těm se říká **lintery** (linters) a **formátovače** (formatters). Mezi populární lintery a formátovače patří například ESLint, Prettier, Stylelint, RuboCop, Pylint a další. Linter je nástroj, který analyzuje kód a hledá potenciální chyby, nekonzistence a problémy s kvalitou kódu. Formátovač je nástroj, který automaticky upravuje kód podle předem definovaných pravidel, jako je odsazení, mezery, řádkování a další.

Statická analýza kódu avšak umí odhalit i jiné problémy, jedná se například o:

- Mrtvý kódu (kód, který není nikdy volán).
- Potenciální chyby (například neuzavřené zdroje, nechycené výjimky).
- Bezpečnostní zranitelnosti (například SQL injection, XSS).
- Složité funkce (funkce, které jsou příliš dlouhé nebo mají příliš mnoho větvení).
- Duplicitní kód (opakující se bloky kódu).
- A další.

Statickou analýzu by měl provádět každý vývojář na svém kódu předtím, než kód odešle do společného repozitáře. Často to bývá součástí vývojových prostředí. V rámci projektů a verzování bývají nastaveny tzv. **pre-commit hooky**, což jsou skripty, které se spustí předtím, než je kód odeslán do repozitáře. Nebo je analýza kódu součástí **CI/CD** pipeline, což je proces, který automatizuje testování a nasazení kódu před jeho začleněním do ostatních větví.

2.5 DevOps

Developer Operations (DevOps) je přístup, který kombinuje vývoj (development) a provoz (operations) s cílem zlepšit spolupráci mezi těmito dvěma týmy. Bez správného DevOps přístupu může být nasazení a správa aplikací komplikovaná a náchylná k chybám. Můžeme si představit, že pokud tým vývojářů vytvoří aplikaci, ale pouze jeden z nich má přístup k nahrání na produkční server, může to vést k problémům, pokud tento vývojář není k dispozici. Podobně se do této produkce mohou dostat chyby, pokud není proces nasazení dobře definován a automatizován. Nehledě na bezpečnostní rizika spojená s tím, kdo má přístup k produkčnímu prostředí.

DevOps zahrnuje několik klíčových oblastí:

- Automatizace nasazení (deployment automation): Proces nasazení aplikace by měl být co nejvíce automatizovaný, aby se minimalizovaly chyby a zrychlil čas nasazení.
- Monitorování a logování (monitoring and logging): Je důležité sledovat výkon aplikace a zaznamenávat události, aby bylo možné rychle reagovat na problémy.
- Infrastruktura jako kód (Infrastructure as Code, IaC): Správa a provisioning infrastruktury pomocí kódu, což umožňuje verzování a opakovatelnost.
- Spolupráce a komunikace (collaboration and communication): DevOps klade důraz na spolupráci mezi vývojáři a provozními týmy, aby se zlepšila efektivita a kvalita softwaru.

Nejčastěji se setkáte s dvěma základními přístupy k DevOps:

- Continuous Integration (CI): Pravidelné integrace kódu do hlavní větve repozitáře, často s automatickým testováním.

- Continuous Delivery/Deployment (CD): Automatizované nasazení kódu do produkčního prostředí po úspěšném testování z určité větve.

Skoro každá Gitová služba (GitHub, GitLab, Bitbucket) má integrované nástroje pro CI/CD, které umožňují snadno nastavit automatizované procesy pro testování a nasazení kódu. Například na GitHubu existují GitHub Actions, které definujeme pomocí tzv. workflow souborů. Workflow soubory jsou psány ve formátu YAML a umožňují definovat různé kroky, které se mají provést při určitých událostech, jako je push do repozitáře nebo vytvoření pull requestu. Nalezneme je přímo v repozitáři v adresáři `.github/workflows/`.

Níže je jednoduchý příklad workflow souboru, který spouští testy při každém pushi do hlavní větve:

yaml

Ukázka CI workflow souboru pro GitHub.

```yaml
name: CI
on:
  push:
    branches:
      - main
jobs:
  - test:
      runs-on: ubuntu-latest
      steps:
        - uses: actions/checkout@v2
        - name: Set up Node.js
          uses: actions/setup-node@v2
          with:
            node-version: '14'
        - name: Install dependencies
          run: npm install
        - name: Run tests
          run: npm run test
```

Jednotlivých kroků může být samozřejmě více a mohou dělat v podstatě cokoliv. Komunitních akcí je k dispozici velké množství a můžeme si je i vytvářet sami. Do workflow lze přidávat i proměnné prostředí, které vidí jen spuštěný skript.

Obecně tyto skripty fungují na principu spouštěčů. Což je server, který je přiřazen k určité práci a ten spouští jednotlivé kroky. Nejčastěji je to Linuxový server, ale může to být i Windows nebo Mac. Tyto servery poskytnuté Gitovou službou jsou často sdílené mezi více uživateli a proto jsou velmi zabezpečené k tomu, aby nedošlo k úniku dat. Můžete si ale hostovat vlastní servery, které nejsou například ani omezené na počet minut, které můžete využít.

Autorizace a autentizace

3.1 Autentizace

Autentizace je proces ověření identity uživatele. Jinými slovy, jde o to zjistit, kdo uživatel je. Nejčastěji se autentizace provádí pomocí uživatelského jména a hesla.

Existuje nespočet různých metod autentizace, například:

- Jednoduchá autentizace pomocí uživatelského jména a hesla,
- Biometrická autentizace (otisk prstu, rozpoznání obličeje),
- Přístupové tokeny (JWT, OAuth),
- Použití certifikátů,
- Použití dalšího ověření, například e-mail nebo SMS,
- Použití hardwarových klíčů,
- Použití aplikací,
- A mnoho dalších.

V kontextu přihlašování se poté setkáme s pojmem vícefaktorová autentizace (Multi-Factor Authentication, MFA) či dvoufaktorová autentizace (Two-Factor Authentication, 2FA). Faktor je něco, co uživatel zná (heslo), něco, co uživatel má (mobilní telefon) nebo něco, čím uživatel je (biometrie). Je to tedy kombinace výše uvedených metod. Vícefaktorové přihlášení má za cíl zvýšit bezpečnost tím, že i v případě kompromitace jednoho faktoru (například hesla) zůstávají ostatní faktory nedotčeny.

Pro většinu aplikací je skoro nutností mít vícefaktorovou autentizaci, protože samotné heslo je často slabým článkem zabezpečení.

V kontextu autentizace se často setkáme s pojmem **Single Sign-On (SSO)**. SSO umožňuje uživatelům přihlásit se jednou a získat přístup k více aplikacím nebo službám bez nutnosti opakovaného přihlašování. Příkladem může být přihlášení pomocí Google účtu na různých webových stránkách.

3.2 Autorizace

Autorizace je proces, který určuje, co uživatel může a nemůže dělat po úspěšné autentizaci. Zatímco autentizace odpovídá na otázku „Kdo jsi?“, autorizace odpovídá na otázku „Co smíš dělat?“. Autorizace se implementuje různými způsoby, které odpovídají byznysovému záměru aplikace.

Nejčastější implementací je kombinace, níže popsaných, přístupových práv a rolí.

3.2.1 Přístupová práva

Přístupová práva (permissions) jsou specifické akce, které uživatel může provádět v rámci systému. Tomuto systému se říká **přístup na základě práv** (Permission-Based Access Control, PBAC). Můžeme si to představit jako jednotlivé klíče, které uživatel získá. Například pro vytvoření článku na blogu může uživatel potřebovat právo `create_article`, pro jeho úpravu právo `edit_article` a pro jeho smazání právo `delete_article`. Uživatel má poté pro svůj účet určenou sadu těchto práv.

Při provedení dané akce se poté kontroluje, zda uživatel má potřebné právo.

Výhoda tohoto přístupu je jeho flexibilita. Proto můžeme jednotlivě každému uživateli přiřadit přesně ta práva, která potřebuje. Nevýhodou je naopak složitost správy, zejména ve větších systémech, kdy nastavování práv může být hazardní a náchylné k chybám.

3.2.2 Role

Role jsou předdefinované sady přístupových práv, které lze přiřadit uživatelům. Tomuto systému se říká **přístup na základě rolí** (Role-Based Access Control, RBAC). Role představují určitou funkci nebo pozici v rámci organizace nebo systému. Například role `admin` může mít práva na všechno, a naopak role `editor` může mít práva pouze na vytváření a úpravu obsahu, ale ne na jeho mazání. Uživatel má poté určenou roli (případně více rolí) a získává všechna práva, která jsou s touto rolí spojena.

Při provedení dané akce se kontroluje, zda uživatelova role stačí k provedení této akce. Často bývá role implementování pomocí práv, stejně jako výše, ale uživateli se dávají pouze role.

Výhodou tohoto přístupu je jeho jednoduchost a snadná správa. Nevýhodou je naopak menší flexibilita, protože všichni uživatelé s danou rolí mají stejná práva. A i dočasné odchylky vyžadují vytvoření nové role. A pokud jsou role tvořeny v kódu nedostatečně, je nutné upravovat kód.

3.2.3 Další modely autorizace

Kromě PBAC a RBAC existují i další modely autorizace, které mohou být vhodné pro specifické scénáře:

- **Přístup na základě atributů** (Attribute-Based Access Control, ABAC): Tento model využívá atributy uživatele, prostředí a zdroje k rozhodování o přístupu. Například uživatel může mít přístup k dokumentu pouze během pracovní doby nebo pouze pokud je členem určité skupiny.
- **Přístup na základě politik** (Policy-Based Access Control, PBAC[2]): Tento model umožňuje definovat komplexní politiky, které určují přístup na základě různých podmínek a pravidel.
- **Přístup na základě seznamu přístupů** (Access Control List, ACL): Tento model přiřazuje konkrétní práva jednotlivým uživatelům nebo skupinám k jednotlivým zdrojům. Například soubor může mít seznam uživatelů, kteří mají právo číst, zapisovat nebo spouštět tento soubor.

Každý z těchto modelů má své výhody a nevýhody a je důležité vybrat ten, který nejlépe vyhovuje potřebám konkrétní aplikace nebo systému. Jak jsem již zmínil, často se v praxi setkáme s kombinací těchto modelů, aby bylo dosaženo optimální rovnováhy mezi bezpečností, flexibilitou a snadnou správou.

3.3 Relace

Relace (sessions) jsou mechanismus, který umožňuje udržet stav mezi jednotlivými požadavky uživatele na server. Z definice HTTP se jedná o bezstavový protokol, což znamená, že každý požadavek je nezávislý a server si nepamatuje žádné informace o předchozích požadavcích. Proto musíme serveru předávat informace o tom, kdo jsme, při každém požadavku. Relace

[2] První kolize zkratky, jupí.

nám umožňují tento stav udržet, abychom na serveru věděli, že komunikujeme se stejným uživatelem.

Jedna z možností je použití právě relací. Relace fungují tak, že při prvním požadavku uživatele server vytvoří unikátní identifikátor relace (session ID) a tento identifikátor je uložen na straně klienta, obvykle v cookies nebo v lokálním úložišti[3] (local storage). Při každém dalším požadavku uživatel tento identifikátor posílá zpět na server, který na základě tohoto identifikátoru načte odpovídající data relace.

Při přihlášení (poslání přihlašovacích údajů, vícefázové autentizaci, ...) server přiřadí[4] k relaci informace o přihlášeném uživateli. Tyto informace mohou zahrnovat uživatelské ID, role, přístupová práva a další relevantní data.

Relace mají omezenou životnost, která může být nastavena na serveru. Po uplynutí této doby je relace automaticky ukončena a uživatel musí znovu projít autentizací. Pokud uživatel smaže uložiště, relace se z pohledu uživatele ztratí. Při dalším požadavku server vytvoří novou relaci.

3.3.1 Implementace

Níže je jednoduchá implementace session za pomoci knihovny Express.js a cookies.

javascript

Implementace relace v Express.js

```javascript
import express from 'express';

const app = express();
const sessions = new Map();

function generateUniqueSessionId() {
  return Math.random().toString(36).substring(2) +
Date.now().toString(36);
}

app.use((req, res, next) => {
  const sessionId = req.cookies.sessionId;

  if (!sessionId || !sessions.has(sessionId)) {
    const newSessionId = generateUniqueSessionId();

    res.cookie('sessionId', newSessionId, { httpOnly: true, secure:
true });
    const sessionData = { userId: null, roles: [], permissions: [] };
    sessions.set(newSessionId, sessionData);
  } else {
    req.session = sessions.get(sessionId);
  }

  next();
});
app.listen(3000);
```

[3] O tom jsme si povídali přechozí školní rok.

[4] Relace je tedy i pro nepřihlášené uživatele.

3.4 Přihlašovací tokeny

Dalším způsobem, jak udržet stav mezi požadavky, je použití přihlašovacích tokenů. Tokeny jsou obvykle krátké řetězce znaků, které buď nic neobsahují (náhodně generované) nebo obsahují zakódované informace.

Důležité je, že token obdrží uživatel po úspěšné autentizaci a tento token pak posílá při každém dalším požadavku na server. Klíčový rozdíl mezi tokeny a relacemi je, že tokeny jsou obvykle stateless, což znamená, že server nemusí ukládat žádné informace o stavu mezi požadavky. Server pouze ověří platnost tokenu a případně dekóduje informace, které token obsahuje.

To může zjednodušit škálování aplikace, protože není potřeba sdílet stav mezi samotnými servery[5].

Tokeny mohou mít také omezenou životnost, po jejímž uplynutí je potřeba token obnovit – často přihlášením. Další alternativa je použití obnovovacích tokenů (refresh tokens), které umožňují získat nový přihlašovací token bez nutnosti znovu zadávat přihlašovací údaje. Token má obvykle životnost 5 až 15 minut[6], zatímco obnovovací token může mít životnost několik dní nebo týdnů. Po použití obnovovacího tokenu je vhodné jej zneplatnit a vydat nový.

Náhodné tokeny:

- Jsou náhodně generované řetězce znaků, které neobsahují žádné informace,
- Při generaci je nutné zajistit dostatečnou entropii, aby bylo obtížné token uhodnout,
- Po generaci je musíme bezpečně uložit na serveru, aby bylo možné ověřit jejich platnost při každém požadavku,
- Musíme sdílet mezi servery, pokud máme více serverů.

Bezstavové tokeny:

- Obsahují zakódované informace, často např. pouze v Base64,
- Při generaci je podepisujeme klíčem, který znají pouze servery,
- Při ověřování tokenu na serveru dekódujeme a ověříme podpis, abychom zajistili, že token nebyl změněn,
- Není potřeba je ukládat na serveru, protože všechny potřebné informace jsou v tokenu,
- Není potřeba je sdílet mezi servery, protože každý server může ověřit token samostatně.

Mezi nejznámější bezstavové tokeny patří JSON Web Tokeny (JWT). JWT jsou standardizovaný formát pro bezpečné předávání informací mezi stranami jako JSON objekt. Skládá se ze tří částí:

- **Hlavička** (header): Obsahuje metadata o tokenu, jako je typ tokenu (obvykle „JWT“) a algoritmus použitý pro podepisování (např. HMAC SHA256 nebo RSA).
- **Tělo** (payload): Obsahuje samotná data, která chceme předat. Může obsahovat standardní nároky (claims) jako `iss` (vydavatel), `sub` (předmět), `aud` (publikum), `exp`[7] (expirace) a další vlastní data,

[5] U relací musíme jinému serveru říct, jaké informace o relaci máme. Load balancery ale často řeší tuto problematiku automaticky.

[6] Při přihlášení např. k Instagramu se skoro nikdy znovu nepřihlašujeme. Pokud bychom se ale na Instagram dlouhou dobu nepodívali, budeme muset znovu zadat heslo – protože nám expiroval refresh token.

[7] Udáváme dobu, po které již není token platný.

- **Podpis** (signature): Slouží k ověření integrity tokenu a jeho autenticity. Vytvoří se kombinací hlavičky, těla a tajného klíče pomocí zvoleného algoritmu.

Tělo a hlavička jsou zakódovány v Base64, a jejich data jsou v JSON formátu. Pokud změníme jakoukoli část tokenu, podpis již nebude platný, což zajišťuje, že data nebyla po cestě změněna.

Pro práci s JWT v JavaScriptu můžeme použít knihovnu `jsonwebtoken`.

3.5 OAuth

OAuth (Open Authorization) je otevřený standard pro autorizaci, který umožňuje aplikacím získat omezený přístup k uživatelským účtům na jiných službách bez nutnosti sdílet uživatelské jméno a heslo.

OAuth využíváme ve dvou hlavních scénářích:

- **Delegovaná autorizace**: Uživatel umožní jedné aplikaci přístup k určitým informacím nebo funkcím na jiné službě (např. umožní aplikaci přístup k jeho kontaktům na Google),
- **Single Sign-On (SSO)**: Uživatel se může přihlásit do jedné aplikace pomocí účtu z jiné služby (např. přihlášení na web pomocí Google nebo Facebook účtu) a nemusí si vytvářet nový účet.

Systém fungování je závislý na konkrétní verzi OAuth (dnes se používá OAuth 2.0), ale obecně zahrnuje následující kroky:

1. Uživatel se pokusí přihlásit do aplikace (klienta),
2. Aplikace přesměruje uživatele na autorizační server (např. Google, Facebook), kde uživatel zadá své přihlašovací údaje a udělí aplikaci potřebná oprávnění,
3. Po úspěšném přihlášení a udělení oprávnění autorizační server přesměruje uživatele zpět do aplikace se speciálním kódem.

Po tomto kroku již záleží na konkrétním způsobu přihlášení. V základu máme dvě možnosti, co dále nastane:

1. Aplikace kód použije tak, že pošle kód znovu na server služby, ze kterého dostane dva tokeny – přihlašovací a obnovovací. Pro toto je nutné mít backend, který s tímto serverem komunikuje (musíme mu totiž poskytnou náš tajný klíč).
2. Aplikace obrží v kódu přímo přihlašovací token, který může použít pro přístup k API služby. Tento způsob se používá hlavně u mobilních a desktopových aplikací. Nemáme však možnost získat obnovovací token.

K tomu, aby se přihlášení nemohlo zneužít, bude potřeba si pro aplikaci vytvořit účet na dané službě a získat tak klientské ID a tajný klíč (client secret). Pro daný projekt poté můžeme ve službě například povolit určité přístupy k API, nastavit přesměrovací URL[8], nastavit logo aplikace a další. Pro určité pravomoce vám daná služba může požadovat ověření aplikace – např. jim budete muset sdílet kód aplikace, aby ověřili, že s jejich API nebudete dělat něco špatného.

OAuth je široce používán pro přihlašování na webových stránkách a v aplikacích, protože umožňuje uživatelům snadno a bezpečně přistupovat k různým službám bez nutnosti vytvářet nové účty a hesla pro každou službu.

[8]Musíme přesně nastavit URL, na kterou se vůbec z aplikace můžeme dostat zpět po přihlášení.

3.5.1 Implementace

Nejjednodušší implementace OAuth2 je pomocí služeb Google. Níže je krátká implementace přihlášení pomocí Google účtu čistě v prohlížeči.

html

Implementace OAuth2 v čistém JavaScriptu

```
<script>
  const CLIENT_ID = 'YOUR_CLIENT_ID_HERE';
  const REDIRECT_URI = 'http://localhost:5500';
  const SCOPE = 'https://www.googleapis.com/auth/userinfo.profile';

  document.getElementById('login').addEventListener("click", () => {
    const authUrl = `https://accounts.google.com/o/oauth2/v2/auth?
response_type=token&client_id=${CLIENT_ID}
&redirect_uri=${encodeURIComponent(REDIRECT_URI)}
&scope=${encodeURIComponent(SCOPE)}&prompt=consent`;
    window.location = authUrl;
  });

  if (window.location.hash) {
    const hash = new URLSearchParams(window.location.hash.substring(1));
    const accessToken = hash.get('access_token');
    if (accessToken) {
      document.getElementById('token').innerText = `Access Token:
${accessToken}`;
      console.log('Access Token:', accessToken);
    }
  }
</script>
```

V ukázce si všimnete, že po kliknutí na tlačítko nás přesměruje prohlížeč na přihlašovací stránku Google. Po úspěšném přihlášení a udělení oprávnění nás Google přesměruje zpět na naši stránku (tuto stránku), kde v kotvě (viz `hash`) máme přihlašovací token. Token si poté můžeme uložit do cookies nebo do lokálního úložiště a posílat ho při každém požadavku na API Google. Tím můžeme získat například informace o uživateli – konkrétně v endpointu `https://www.googleapis.com/oauth2/v1/userinfo?alt=json`.

V ukázce ignorujeme nastavení v Google Cloud Console, kde je potřeba povolit OAuth2 a nastavit přesměrovací URL.

Animace a interaktivita na webu

4.1 Připomenutí klasických animací

Interaktivita na webu je velice důležitá. Z minulého roku víme, že je důležité, aby webová stránka reagovala na akce uživatele. Interaktivita z pouhé informační stránky dělá webovou aplikaci, které reálně může lidem sloužit.

V základním kontextu webů (tedy cca to, co jsme probrali v předmětu Webové aplikace) je interaktivita řešena pomocí následujících technologií:

- Formuláře,
- Odkazy,
- Přechodové animace v CSS,
- Základní animace v CSS,
- Základní animace pomocí JavaScriptu.

Zmíněné animace dovolují upravovat styly prvků na stránce, včetně například jejich pozice, velikosti, barvy, průhlednosti a dalších vlastností.

Přechodové animace tvoříme pomocí vlastnosti `transition`, ve které určujeme časování, vlastnosti a provedení přechodu. Provedení přechodu je funkce, která se používá pro interpolaci mezi počáteční a koncovou hodnotou animované vlastnosti.

Základní animace tvoříme pomocí vlastnosti `animation`, ve které určujeme název animace, časování, počet opakování a další vlastnosti. Animace definujeme pomocí klíčových snímků (`@keyframes`), kde určujeme, jaké hodnoty má animovaná vlastnost v různých časech animace.

Animace pomocí JavaScriptu dovolují i složitější interakce, například reagovat na pohyb myši, klávesnici, dotykové obrazovky a další vstupy uživatele. Změny stylů zařizuje DOM. Animace tvoříme pomocí funkce `animate` na prvku, která přijímá pole klíčových snímků a objekt s vlastnostmi animace.

Animace v JS dovoluje opravdu komplexní věci, protože můžeme reagovat na jednotlivé animace a různě je kombinovat.

> ⚠ **Doporučené materiály k animacím**
>
> Pro připomenutí doporučuji projít materiály pro předmět Webové aplikace. Všechny zmíněné funkcionality byste měli znát a umět je použít.

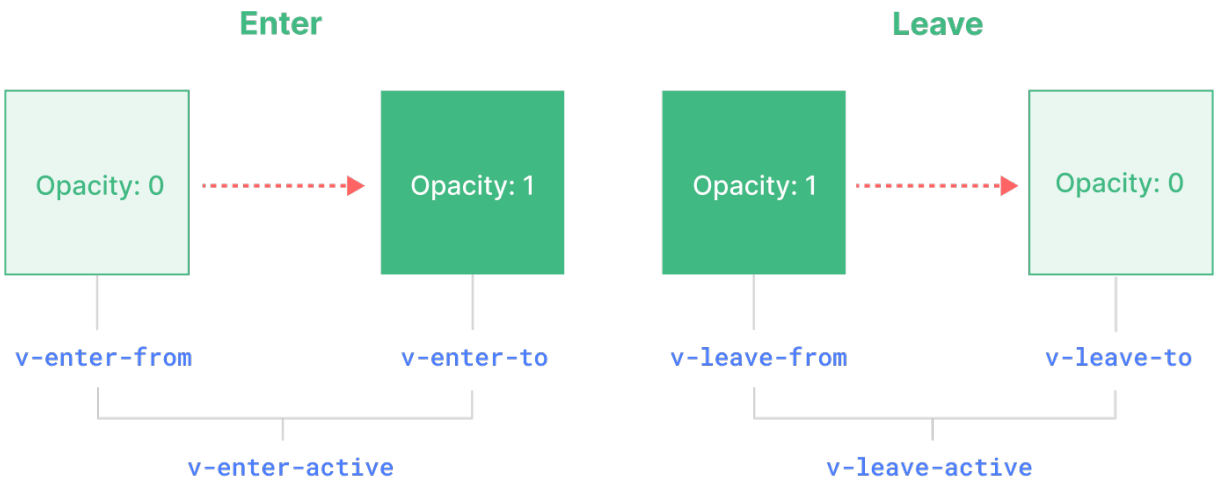
4.2 Animace ve Vue

Ve Vue potřebujeme řešit animace trochu jinak, protože Vue pracuje s virtuálním DOM (viz kapitola Kapitola 8.1). Například pokud používáme direktivy `v-if` tak změnění hodnoty (pro např. zobrazení) způsobí překreslení části DOM, což je okamžitá změna. Pokud chceme animaci, je potřeba použít speciální komponentu `transition`, která dovoluje animovat změny stavu. Uvnitř komponenty `transition` vkládáme jeden prvek, který chceme animovat. Danému přechodu při změně stavu můžeme definovat jméno „animace“, její délku a další vlastnosti.

Vue interně přidává třídy do prvku, které dovolují definovat animace pomocí CSS. Jedná se o třídy, viz níže a obrázek:

- `<name>-enter-from`: Počáteční stav který se předpokládá před animaci,
- `<name>-enter-active`: Stav během animace,

- `<name>-enter-to`: Konečný stav po animaci,
- `<name>-leave-from`: Počáteční stav který se předpokládá před animaci (při opuštění),
- `<name>-leave-active`: Stav během animace (při opuštění),
- `<name>-leave-to`: Konečný stav po animaci (při opuštění).



Kromě těchto tříd Vue přidává i třídu `v-enter` a `v-leave`, které se používají pro určení, že prvek je v procesu animace.

html

Animace ve Vue

```
<template>
  <button @click="show = !show">Toggle</button>
  <transition name="fade" mode="out-in">
    <p v-if="show">Hello Vue!</p>
  </transition>
</template>
<script>
// ...
</script>
<style>
.fade-enter-active, .fade-leave-active {
  transition: opacity 0.5s;
}
.fade-enter-from, .fade-leave-to {
  opacity: 0;
}
.fade-enter-to, .fade-leave-from {
  opacity: 1;
}
</style>
```

Animuje zobrazení a skrytí textu pomocí přechodu opacity.

Pomocí těchto tříd můžeme definovat skoro jakoukoliv animaci přechodu.

Komponenta `transition` zpracovává pouze jeden prvek. Pokud chceme animovat více prvků, je potřeba použít komponentu `transition-group`, která dovoluje animovat skupinu prvků a jejich změny závislé na sobě. Komponenta funguje podobně jako `transition`, ale uvnitř sebe dovoluje více prvků.

Nová třída, kterou Vue přidává pro `transition-group` je `<name>-move`, která se na prvek přidá, pokud se změní jeho pozice v rámci skupiny.

html

Animace ve Vue, více prvků

```
<template>
  <button @click="addItem">Add Item</button>
  <transition-group name="list" tag="ul">
    <li v-for="item in items" :key="item" class="list-item">{{ item }}</li>
  </transition-group>
  <button @click="removeItem">Remove Item</button>
</template>
<script setup>
import { ref } from 'vue';
const items = ref(['Item 1', 'Item 2', 'Item 3']);
function addItem() {
  items.value.push(`Item ${items.value.length + 1}`);
}
function removeItem() {
  items.value.pop();
}
</script>
<style>
.list-item {
  margin: 5px 0;
}
.list-enter-active, .list-leave-active {
  transition: all 0.5s;
}
.list-enter-from, .list-leave-to {
  opacity: 0;
  transform: translateY(-20px);
}
.list-enter-to, .list-leave-from {
  opacity: 1;
  transform: translateY(0);
}
</style>
```

Animuje přidání a odebrání položek ze seznamu.

4.3 Animace v DOM

Animace v DOM jsou trochu komplikovanější, protože musíme řešit vykreslování a aktualizaci hodnot. Funkce `animate` na prvcích je sice jednoduchá, ale má omezenou použitelnost – například není možné definovat délku jednotlivého snímku animace.

Pro složitější animace je potřeba použít JavaScript a manipulovat s vlastnostmi prvků přímo. To tedy znamená, že si sami budeme počítat hodnoty pro změnu vlastností a tyto hodnoty budeme aplikovat na prvky.

Velmi brzy zjistíme, že problém, který budeme řešit, je dopočítávání hodnot mezi jednotlivými snímky animace. To víme, že se dělá pomocí interpolace.

js

Jednoduchá animace v DOM

```
const element = document.getElementById("myElement");

let x = element.getComputedStyle().left.replace("px", "");
let targetX = x + 100; // Cílová pozice
let duration = 1000; // Délka animace v ms

const STEPS = 100; // Počet kroků animace
setInterval(() => {
  x += (targetX - x) / STEPS;
  > Použití interpolace, jako “čas” se používá počet kroků animace.
  element.style.left = `${x}px`;
}, duration / STEPS);
```

Posune prvek o 100px doprava během 1 sekundy.

Tento způsob animace má ale řadu problému. Nejklíčovějším problémem je to, že funkce `setInterval` (a `setTimeout`), které se používají pro časování, nejsou přesné. Jediné, co vám JS (resp. jeho runner) zaručí, je že se funkce zavolá „nejdříve“ po uplynutí daného času. To znamená, že pokud je CPU vytížené – a nebo okno není aktivní – tak se funkce zavolá později. To vede k tomu, že animace není plynulá a může „skákat“. Navíce obecně vůbec nerespektujete snímkovací frekvenci monitoru a ani žádost uživatele o omezení animací.

Pro plynulé animace by se měla používat funkce `requestAnimationFrame`, která je optimalizovaná pro animace a zaručuje, že se funkce zavolá před dalším vykreslením obrazovky. Tuto funkci musíme uvnitř volání původní funkce znovu zavolat, čímž vytvoříme smyčku.

Použitím request animation frame se ale vyskytne problém s tím, že nevíme, jak dlouho uplynulo od posledního snímku. To je možné řešit tím, že si budeme ukládat čas posledního snímku a při každém volání funkce si spočítáme rozdíl. Rozdíl je potom kolik milisekund uplynulo od posledního snímku a tedy můžeme dopočítat hodnoty podle skutečného času. Prvním argumentem funkce, kterou `requestAnimationFrame` volá, je čas v milisekundách od začátku načítání stránky a pro volání všech `requestAnimationFrame` je tento čas synchronizovaný.

js

Animace v DOM s `requestAnimationFrame`

```
const element = document.getElementById("myElement");
let x = element.getComputedStyle().left.replace("px", "");
let targetX = x + 100; // Cílová pozice
```

```
let duration = 1000; // Délka animace v ms
let startTime = null;

function animate(time) {
  if (!startTime) startTime = time;
  const elapsed = time - startTime;

  x += (targetX - x) * Math.min(elapsed / duration, 1);
  element.style.left = `${x}px`;

  element.style.left = `${targetX}px`;
  if (elapsed < duration) {
    requestAnimationFrame(animate);
  }
}
requestAnimationFrame(animate);
```

Posune prvek o 100px doprava během 1 sekundy.

Pokročilé aplikace často vůbec HTML prvky nepoužívají pro vykreslování. Jeden z důvodů je to, že prvky HTML jsou sice optimalizované, ale ne tak moc, aby v něm šlo tvořit pokročilé simulace či hry. HTML taktéž není navržené na simulaci 3D světa.

4.3.1 Interpolační knihovny

Na interpolační funkce existuje řada materiálů a knihoven. Postup při dopočítávání je však stejný, mění se však způsob (průběh) interpolační funkce. Na seznam známých interpolačních funkcí se můžete podívat například na easings.net včetně vizuálního znázornění.

Pro složitější animace je možné použít knihovny, které řeší časování a interpolační funkce za nás. Jedna ze známých knihoven je GSAP, která dovoluje tvořit složité animace. V něm existuje řada pluginů, které dovolují navzájem propojovat animace, tvořit časové osy a další pokročilé věci, včetně například zobrazení při scrollování.

Mezi další známé knihovny patří:

- Motion.dev,
- Anime.js,
- Velocity.js,
- Popmotion,
- Tween.js.

Všechny tyto knihovny doporučuji prozkoumat a vyzkoušet, protože každá má své výhody a nevýhody.

4.4 Vykreslování pomocí Canvas

V HTML často potřebujeme prvek, do kterého umí webová stránka zapisovat grafiku. To je možné pomocí elementu `canvas`, který je součástí HTML5.

Význam tohoto tagu je, že „poskytuje oblast pro dynamické, skriptované vykreslování grafiky (obvykle pomocí JavaScriptu)“.

Tento tag by se neměl používat jako nahrada semantiky. Není vhodné tedy například nadpis nahradit vykresleným textem v canvasu.

Uvnitř tagu `canvas` můžeme definovat alternativní obsah, který se zobrazí v případě, že prohlížeč nepodporuje tento tag – jedná se i o sémantiku tohoto obsahu.

Tag `canvas` má spoustu základních atributů, které ale nejsou příliš důležité pro běžné použití.

Canvas se ovládá pomocí JavaScriptu.

4.4.1 Kontext

Pro práci s canvasem je potřeba získat tzv. kontext. Kontext je objekt, který poskytuje metody a vlastnosti pro kreslení na canvas.

V základu rozlišujeme dva typy kontextu:

- 2D kontext (`2d`), který je velmi jednoduchý a netvoří příliš složité věci,
- WebGL kontext (`webgl`), který je složitější a dovoluje tvořit 3D grafiku.

Existují i další kontexty, například `webgl2`, `bitmaprenderer` a další, ale ty nejsou příliš rozšířené.

4.4.2 Základní struktura

Pro vytvoření canvasu v HTML použijeme tag `canvas`. Tento tag je nutné poté získat v JavaScriptu. Z toho je poté možné dostat kontext a s ním pracovat.

Pro správné zobrazení[9] je pak nutné, aby měl canvas nastavenou šířku a výšku.

V základu canvas funguje jako uložiště zapsaných pixelů. Při každém vykreslení se tedy přepíší pixely, které tam byly předtím. Ze základu tedy vykreslujeme na prázdný canvas a jednotlivé příkazy se vykreslují v pořadí, v jakém jsou napsány. Není potřeba přepisovat pixely aby zůstaly viditelné.

html

Základní použití canvasu

```html
<canvas id="myCanvas">
  Váš prohlížeč nepodporuje canvas.
</canvas>
<script>
  const canvas = document.getElementById("myCanvas");
  canvas.width = 500;
  canvas.height = 500;

  const ctx = canvas.getContext("2d");
</script>
```

[9] Později, až se budeme zaobírat responzivitou, je nutné canvas často přizpůsobovat velikosti okna a to vede k nutnosti překreslit obsah.

4.4.3 2D kontext

2D kontext je velmi jednoduchý a dovoluje tvořit základní grafiku, jako jsou čáry, kruhy, obdélníky, text a obrázky. Pro práci s 2D kontextem používáme řadu metod, které nám dovolují kreslit na canvas.

V základu se kreslí pomocí cest (paths). Cesty nám určují tvar, který chceme vykreslit. Skládají se z jednotlivých bodů, které jsou spojeny čarami. Většina metod pro kreslení tyto cesty automaticky skládá a vykresluje.

Tyto cesty pak můžeme vykreslit pomocí dvou základních metod:
- `stroke()`, která vykreslí obrys cesty,
- `fill()`, která vykreslí vyplněnou cestu.

Barvy a styly se nastavují pomocí vlastností kontextu, jako jsou `strokeStyle`, `fillStyle`, `lineWidth` a další.

Základní cesty z bodů tvoříme pomocí metod:

- `beginPath()`, která zahájí novou cestu,
- `moveTo(x, y)`, která přesune pero na souřadnice (x, y) bez kreslení,
- `lineTo(x, y)`, která nakreslí čáru z aktuální pozice na souřadnice (x, y),
- `arc(x, y, radius, startAngle, endAngle)`, která nakreslí oblouk s daným středem (x, y), poloměrem a úhly,
- `closePath()`, která uzavře cestu tím, že nakreslí čáru zpět na začátek cesty.

Vytvoření nové cesty

Pro vytvoření nové cesty je nutné vždy zavolat `beginPath()`. Funkce `closePath()` není povinná, ale je užitečná pro uzavření tvarů.

Po tvorbě cesty je nutné ji vykreslit, viz výše.

js

Základní kreslení cesty

```js
const canvas = document.getElementById("myCanvas");
canvas.width = 500;
canvas.height = 500;
const ctx = canvas.getContext("2d");
ctx.beginPath();
ctx.moveTo(50, 50);
ctx.lineTo(150, 50);
ctx.lineTo(100, 150);
ctx.closePath();
ctx.fillStyle = "blue";
ctx.fill();
ctx.strokeStyle = "red";
ctx.lineWidth = 5;
ctx.stroke();
```

Vykreslí modrý trojúhelník s červeným obrysem.

Cestu smažeme tím, že zavoláme `beginPath()` a vytvoříme novou cestu.

Pro zjednodušení práce existují následnující metody, které nám vytvoří cesty za nás:

- `rect(x, y, width, height)`, která vytvoří obdélník na souřadnicích (x, y) s danou šířkou a výškou,
- `fillRect(x, y, width, height)`, která vytvoří a vykreslí vyplněný obdélník,
- `strokeRect(x, y, width, height)`, která vytvoří a vykreslí obrys obdélníku.

Do canvasu můžeme taktéž vykreslovat text pomocí metod:

- `fillText(text, x, y)`, která vykreslí vyplněný text na souřadnicích (x, y),
- `strokeText(text, x, y)`, která vykreslí obrys textu na souřadnicích (x, y).

Pro nastavení fontu a stylu textu používáme vlastnost `font`. Dovnitř vkládáme CSS-like zápis fontu, například `20px Arial`.

js

Kreslení textu

```
ctx.font = "30px Arial";
ctx.fillStyle = "black";
ctx.fillText("Ahoj světe!", 200, 100);
ctx.strokeStyle = "green";
ctx.lineWidth = 2;
ctx.strokeText("Ahoj světe!", 200, 150);
```

Vykreslí text 'Ahoj světe!' dvakrát, jednou vyplněný černě a podruhé s obrysem zeleně.

Někdy je potřeba spočítat šířku textu, například pro zarovnání. To je možné pomocí metody `measureText(text)`, která vrací objekt s informacemi o šířce textu pro aktuální font.

Pro vykreslení obrázků používáme metodu `drawImage(image, x, y, width, height)`, která vykreslí obrázek na souřadnicích (x, y) s danou šířkou a výškou. Obrázek může být HTMLImageElement, HTMLCanvasElement nebo HTMLVideoElement.

js

Kreslení obrázku

```
const img = new Image();
img.src = "path/to/image.jpg";
img.addEventListener("load", () => {
  ctx.drawImage(img, 50, 200, 100, 100);
});
```

Vykreslí obrázek na canvas.

Pro animace je nutné smazat veškerý obsah canvasu aby nezůstávaly stopy po předchozích snímcích. To je možné pomocí metody `clearRect(x, y, width, height)`, která smaže obdélník na souřadnicích (x, y) s danou šířkou a výškou. Pro smazání celého canvasu tedy použijeme `clearRect(0, 0, canvas.width, canvas.height)`.

Pro animace je vhodné i používat transformace, které dovolují měnit pozici, velikost a rotaci vykreslovaných objektů. Transformace se aplikují na celý kontext a ovlivňují všechny následné kresby bez toho, abychom museli měnit souřadnice jednotlivých objektů počítáním. Transformace se aplikují pomocí metod:

- `translate(x, y)`, která posune kontext o (x, y),
- `rotate(angle)`, která otočí kontext o daný úhel v radiánech,
- `scale(x, y)`, která zvětší nebo zmenší kontext o (x, y).

To používáme například na:

- Vykreslení na střed,

- Kameru v 2D prostoru s pohybem,
- Otočení objektů,
- A další.

Transformace můžeme ukládat a obnovovat pomocí metod:
- `save()`, která uloží aktuální stav kontextu na zásobník,
- `restore()`, která obnoví stav kontextu ze zásobníku.

To je užitečné, když chceme aplikovat transformace pouze na určité části kreslení a poté se vrátit k původnímu stavu. Pokud bychom vracení nepoužívali, cyklili bychom transformace a každé vykreslení by bylo více a více deformované.

Transformace

```js
ctx.save();
ctx.translate(250, 250);
ctx.rotate(Math.PI / 4);
ctx.fillStyle = "purple";
ctx.fillRect(-50, -50, 100, 100);
ctx.restore();
```

Vykreslí fialový čtverec otočený o 45 stupňů na střed canvasu.

4.4.4 Kamera a její pohyb

Pro pohyb v 2D prostoru je potřeba simulovat kameru. Nikdy v kódu nechceme pro každý objekt řešit to, jak se má v rámci pohybu kamery vykreslit. Obyčejné systémy (a tedy i canvas) neřeší kameru jako objekt, ale je potřeba ji simulovat pomocí transformací, viz předchozí kapitola.

Typicky provedeme následující kroky při každém vykreslení snímku:
- Smažeme celý canvas,
- Uložíme stav kontextu,
- Aplikujeme transformace kamery (typicky `translate` a `scale`),
- Vykreslíme všechny objekty ve scéně – které mají své pozice v rámci světa,
- Obnovíme stav kontextu.

Pohyb kamery

```js
const camera = {
  x: 0,
  y: 0,
  zoom: 1,
};
const objects = [
  { x: -100, y: -100, width: 50, height: 50, color: "red" },
  { x: 100, y: -100, width: 50, height: 50, color: "green" },
  { x: -100, y: 100, width: 50, height: 50, color: "blue" },
  { x: 100, y: 100, width: 50, height: 50, color: "yellow" },
];
function render() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.save();
```

```
15
16   ctx.translate(canvas.width / 2, canvas.height / 2);
17   ctx.scale(camera.zoom, camera.zoom);
18   ctx.translate(-camera.x, -camera.y);
19
20   for(let object of objects) {
21     ctx.fillStyle = object.color;
       > Nejlepší by bylo, kdyby se každý objekt vykresloval sám, to si
   ukážeme později.
22     ctx.fillRect(object.x, object.y, object.width, object.height);
23   }
24
25   ctx.restore();
26   requestAnimationFrame(render);
27 }
28 requestAnimationFrame(render);
```

Simulace kamery v 2D prostoru.

Ve výše uvedném příkladu je kamera reprezentována objektem s pozicí a zoomem. Při pohybu (změny vlastností v objektu) se změní to, jak se objekty vykreslí. Pro pohyb kamery můžeme použít různé vstupy, například klávesnici, myš nebo dotykovou obrazovku. To již zvládneme s DOM, který umíme již od druhého ročníku.

Pro získání ukázané pozice z myši potřebujem získat pozici myši v rámci canvasu a poté ji přepočítat na pozici ve světě.

js

Pozice myši ve světě

```
1 canvas.addEventListener("mousemove", (event) => {
2   const rect = canvas.getBoundingClientRect();
3   const mouseX = event.clientX - rect.left;
4   const mouseY = event.clientY - rect.top;
5
6   let clickedX = (mouseX - canvas.width / 2) / camera.zoom + camera.x;
7   let clickedY = (mouseY - canvas.height / 2) / camera.zoom + camera.y;
8
9   console.log(`Mouse in world: (${clickedX}, ${clickedY})`);
10 });
```

Přepočítání pozice myši na pozici ve světě.

4.4.5 Reprezentace objektů

Dalším častým krokem je to, že počet objektů v naší scéně může být velký. Spravovat takový počet objektů přímo v hlavním kódu a řešit jejich vykreslování je nepraktické a neudržitelné. Proto je vhodné vytvořit si třídu nebo funkci, která bude reprezentovat jednotlivé objekty. Tato třída bude mít své vlastnosti, jako jsou pozice, velikost, barva a metody pro vykreslení a aktualizaci.

Níže je tedy ukázka vytvoření abstraktní třídy `SceneObject`, která má metody `update` a `draw`. Taktéž najdete ukázku konkrétní třídy `Rectangle`, která dědí z `SceneObject` a implementuje metody pro vykreslení obdélníku.

ts

Reprezentace objektů

```ts
abstract class SceneObject {
  x: number;
  y: number;
  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
  abstract update(deltaTime: number): void;
  abstract draw(ctx: CanvasRenderingContext2D): void;
}
class Rectangle extends SceneObject {
  width: number;
  height: number;
  color: string;
  constructor(x: number, y: number, width: number, height: number, color: string) {
    super(x, y);
    this.width = width;
    this.height = height;
    this.color = color;
  }

  update(sceneTime: number): void {}

  draw(ctx: CanvasRenderingContext2D): void {
    ctx.fillStyle = this.color;
    ctx.fillRect(this.x, this.y, this.width, this.height);
  }
}
```

Abstraktní třída pro objekty ve scéně a konkrétní implementace pro obdélník.

Tímto způsobem můžeme snadno přidávat nové typy objektů do naší scény a spravovat je pomocí společného rozhraní[10]. Mimo tento kód poté ukládáme odkazy na jednotlivé instance do pole a v hlavní smyčce je aktualizujeme a vykreslujeme.

ts

Správa objektů ve scéně

```ts
const objects: SceneObject[] = [
  new Rectangle(-100, -100, 50, 50, "red"),
  new Rectangle(100, -100, 50, 50, "green"),
  new Rectangle(-100, 100, 50, 50, "blue")
];

function render(time: number) {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  // Kamera

```

[10]Doporučuji Vám si připomenout význam polymorfismu v OOP.

```
  for(let object of objects) {
    object.update(time);
    object.draw(ctx);
  }

  requestAnimationFrame(render);
}
requestAnimationFrame(render);
```

Aktualizace a vykreslení všech objektů ve scéně.

)

Tímto způsobem máme čistý a udržitelný kód, který dovoluje snadno přidávat nové objekty a jejich chování.

4.5 Simulace

Každé vykreslení snímku (a aktualizace objektů) je možné považovat za simulaci. To znamená, že každý objekt může mít své vlastnosti, jako jsou pozice, rychlost, zrychlení a další. Tyto vlastnosti se pak aktualizují na základě času, který uplynul od posledního snímku. Problém je, že pokud je snímkovací frekvence nestabilní, tak se objekty mohou pohybovat různě rychle. To je možné řešit tím, že animace a aktualizace objektů bude závislá na čase, který uplynul od posledního snímku[11]. K tomu se často používá tzv. `delta time`, což je čas v sekundách nebo milisekundách, který uplynul od posledního snímku. Tento čas se pak používá pro výpočty pohybu a dalších vlastností objektů.

ts

Pohybující se obdélník

```ts
class MovingRectangle extends Rectangle {
  vx: number; // Rychlost v ose x
  vy: number; // Rychlost v ose y
  constructor(x: number, y: number, width: number, height: number, color: string, vx: number, vy: number) {
    super(x, y, width, height, color);
    this.vx = vx;
    this.vy = vy;
  }

  update(deltaTime: number): void {
    this.x += this.vx * deltaTime;
    this.y += this.vy * deltaTime;
  }
}
// ...
const lastTime = performance.now();
function render(time: number) {
  // ...
  const elapsed = time - lastTime;
```

[11] Poctivý čtenář si teď uvědomil, že jsme to v minulých kapitolách již implicitně používali.

```
20   movingRectangle.update(elapsed / 1000);
21   lastTime = time;
22   // ...
23 }
24 // ...
```

Obdélník, který se pohybuje na základě rychlosti a delta time.

4.5.1 Responzivita

Pro správné zobrazení canvasu je potřeba, aby měl správnou velikost. To znamená, že je potřeba nastavit jeho šířku a výšku na základě velikosti okna nebo kontejneru, ve kterém je umístěn. Často je tedy nutné reagovat na změnu velikosti okna a přizpůsobit velikost canvasu. To je možné řešit pomocí události `resize` na okně.

js

Responzivní canvas

```
1 function resizeCanvas() {
2   canvas.width = window.innerWidth;
3   canvas.height = window.innerHeight;
4   // Překreslení obsahu canvasu po změně velikosti
5   // Př. scale obsah podle nové velikosti
6 }
7 window.addEventListener("resize", resizeCanvas);
8 resizeCanvas();
```

Přizpůsobí velikost canvasu velikosti okna.

V ukázce vidíme, že po změnění velikosti musíme překreslit obsah canvasu. Pokud to neuděláme, tak se obsah může deformovat nebo být oříznutý. Pro správné přizpůsobení obsahu je často potřeba přepočítat pozice a velikosti objektů na základě nové velikosti canvasu. Taktéž je možné použít transformace, jako je `scale`, pro přizpůsobení obsahu pro danou velikost zařízení.

Uvnitř vykreslování bychom vždy měli používat aktuální velikost canvasu nebo případně kameru, viz Kapitola 4.7.1, pro správné umístění objektů.

4.5.2 Optimalizace

Problém moderního JS oproti jiným jazykům je to, že běží v jednom vlákně. To znamená, že pokud máme náročnou simulaci, tak může dojít k tomu, že se UI[12] zasekne a uživatel nebude moci interagovat s aplikací.

V JS běží na systému události (event loop). Můžeme si to představit jako frontu úkolů, které je potřeba vykonat. Každý úkol musí být vykonán celý, než se začne vykonávat další úkol. To znamená, že pokud máme náročný úkol, tak se může stát, že se jiné úkoly nedostanou na řadu včas.

[12]Každé okno má k sobě přiděleno skoro v každém prohlížeči své vlastní prostředky, proto se může rozbít maximálně jedna stránka.

JS umí asynchronní kód, tedy, že pokud úkol čeká (a tím by aktivně blokoval vlákno), tak se může vykonávat jiný úkol. Často se to používá pro síťové operace, které mohou trvat dlouho. Asynchronní kód ale nemůže řešit problém s dlouhým výpočtem v rámci lokálního kódu.

Mimo rozdělení kódu do menších částí (které jsou pozastavitelné a asynchronní) je možné použít tzv. web workery. Web workery jsou API prohlížeče, které si lze představit jako separátní vlákna, ve kterých může běžet JS kód. Jedna z nevýhod je ale to, že web worker nemá přístup k DOM. S těmito workery – které si lze představit jako zcela oddělené event loop – je možné komunikovat pomocí zpráv[13]. Zprávy jsou pouze textové a nebo binární data, která se posílají mezi hlavním vláknem a workerem.

js

Vytvoření web workeru

```
const worker = new Worker("worker.js");

worker.postMessage({ command: "start", data: {} });
worker.onmessage = (event) => {
  const message = event.data;
  if (message.command === "result") {
    console.log("Výsledek z workeru:", message.data);
  }
};
```

Vytvoří web worker a komunikuje s ním pomocí zpráv.

js

Kód web workeru

```
self.onmessage = (event) => {
  const message = event.data;
  if (message.command === "start") {
    // Proveď nějaký náročný výpočet
    const result = heavyComputation(message.data);
    self.postMessage({ command: "result", data: result });
  }
};
```

Worker přijímá zprávy a posílá zpět výsledky.

Jeden worker může být taktéž použit pro více webových stránek – tzv. sdílené workery. Ty se tvoří pomocí `SharedWorker`.

K workerům se ještě v těchto skriptech vrátíme, a to v Kapitola 7.3.4, kde se používá jejich odvozená verze – service worker.

4.6 Vykreslování pomocí WebGL

Jeden z kontextů canvasu je WebGL. WebGl je API pro vykreslování 2D a 3D grafiky v reálném čase pomocí GPU. WebGL je založený na OpenGL ES, což je zjednodušená verze OpenGL pro mobilní zařízení. WebGL dovoluje využít hardwarovou akceleraci pro vykreslování grafiky.

[13] Tyto zprávy se používají například i na komunikaci mezi okny nebo iframe.

Samotné použití WebGL je poměrně složitá záležitost, protože musíte zcela rozumět systému vykreslování pomocí GPU. Je třeba znát koncepty shaderů, bufferů, textur, matic a dalších.

Samotné WebGL nenabízí například vůbec žádné funkce pro práci s kamerou, světly, materiály a dalšími věcmi, které jsou běžné v 3D grafice. Máme k dispozici pouze nízkoúrovňové funkce pro vykreslování geometrie.

4.6.1 Shadery

Shader je malý program, který běží na GPU a určuje podle svého účelu, jak se má grafika vykreslit na obrazovce. Shadery (a jejich jazyky, resp. GPU) jsou optimalizované pro paralelní zpracování[14], což znamená, že mohou zpracovávat velké množství dat najednou.

Existují dva hlavní typy shaderů:

- Vertex shader, který zpracovává jednotlivé vrcholy geometrie a určuje jejich pozici na obrazovce,
- Fragment shader, který zpracovává jednotlivé pixely a určuje jejich barvu a další vlastnosti.

Vertex je vrcholem geometrie, což je bod v 3D prostoru, který má své souřadnice (x, y, z). Mezi vrcholy se pak tvoří trojúhelníky, které tvoří povrch 3D objektů.

Shadery se píší obecně v různých jazycích a kompilují se do strojového kódu, který je specifický pro daný GPU. Pro WebGL se používá jazyk GLSL (OpenGL Shading Language), který je podobný jazyku C.

Pro nás je důležité, že každý shader může přijímat vstupy (atributy, uniformy a varyings) a vracet výstupy (varyings). Zároveň shader má přístup k vestavěným proměnným, které poskytuje WebGL, například `gl_Position` pro vertex shader a `gl_FragColor` pro fragment shader, které používá pro výstupní hodnoty.

glsl

Jednoduchý vertex shader

```glsl
attribute vec3 aPosition;

void main() {
  gl_Position = vec4(aPosition, 1.0);
}
```

Vertex shader, který přijímá pozici vrcholu a nastaví ji jako výstupní pozici.

glsl

Jednoduchý fragment shader

```glsl
void main() {
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Fragment shader, který nastaví barvu pixelu na červenou.

Psaní shaderů je poměrně složitá záležitost a vyžaduje znalost grafiky a matematiky. Pro složitější věci je potřeba znát i další koncepty. Shadery ale umí vykreslit skoro cokoliv, co si dokážete představit.

[14] Proto mají GPU velmi velký počet jader.

4.6.1.1 Použití WebGL

Pro použití WebGL je potřeba získat kontext `webgl` z canvasu. Pro něj musíme inicializovat spoustu věcí, které nastaví WebGL do použitelného stavu.

Pro použití potřebujeme definovat jednotlivé shadery – potřebujeme zdrojový kód a je potřeba je zkompilovat.

Inicializace WebGL a shaderů

```js
const gl = canvas.getContext("webgl");

const vertexShaderSource = `...`;
const fragmentShaderSource = `...`;

const vs = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vs, vertexShaderSource);
gl.compileShader(vs);

const fs = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fs, fragmentShaderSource);
gl.compileShader(fs);
```

Vytvoří WebGL kontext a zkompiluje jednoduché shadery.

Poté musíme definovat tzv. program, který nám dovolí určit jeho shadery a jejich propojení.

Vytvoření a použití programu

```js
const prog = gl.createProgram();
gl.attachShader(prog, vs);
gl.attachShader(prog, fs);
gl.linkProgram(prog);
gl.useProgram(prog);
```

Vytvoří program, připojí shadery a použije ho.

Poté je nutné definovat geometrii, které se vykreslí na obrazovce. Nejjednoduší je vykreslit přes celé místo canvasu obdélník, který se vykreslí jako dva trojúhelníky. V něm poté můžeme vykreslovat cokoliv pomocí fragment shaderu.

Definice geometrie

```js
const buf = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buf);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([-1,-1, 3,-1, -1,3]),
  gl.STATIC_DRAW);

const loc = gl.getAttribLocation(prog, "pos");
gl.enableVertexAttribArray(loc);
gl.vertexAttribPointer(loc, 2, gl.FLOAT, false, 0, 0);
```

Vytvoří buffer s geometrií a nastaví atributy pro vertex shader.

Posledním krokem je nastavení viewportu a vykreslení.

js

Vykreslení

```
1 gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
2 gl.drawArrays(gl.TRIANGLES, 0, 3);
```

Nastaví viewport a vykreslí geometrii.

Tento kód (dle shaderu) vykreslí na celý canvas to, co shader vypočítá.

Pokud se při nastavení či kompilaci shaderů něco pokazí, tak se v konzoli zobrazí chyba s načtením programu. Podrobné chyby z kompilace shaderů je možné získat pomocí `gl.getShaderInfoLog(shader)` a chyby z linkování programu pomocí `gl.getProgramInfoLog(program)`.

4.6.2 Nepoužití WebGL

Použití WebGL je poměrně složité a vyžaduje hodně znalostí o grafice a programování. WebGL je skvělé pro herní enginy, knihovny, které mohou využít GPU a další pokročilé věci. A tím třeba vůbec dovolit hry v prohlížeči.

Pro běžné použití je ale WebGL zbytečně složitý a je lepší použít knihovny, které WebGL využívají a poskytují jednodušší API. Tedy, nemusíme nutně vědět, jak se načítá model, jak fungují světla nebo umět fyzikálně popsat jakýkoliv materiál.

4.6.3 Jak psát shadery

Pro psaní shaderů je potřeba znát základy grafiky a matematiky. Je potřeba znát vektorovou a maticovou algebru, protože shadery s nimi pracují. Je potřeba znát i základy 3D grafiky, jako jsou souřadnicové systémy, projekce, osvětlení a další.

Pro začátek je dobré si projít nějaký tutoriál, který vás provede základy psaní shaderů. Nejoblíbenější je knížka The Book of Shaders. Nebo WebGL Fundamentals, která vás provede základy WebGL a shaderů.

Shadery lze psát i v různých nástrojích, které vám pomohou s vizuálním návrhem a testováním. Mezi oblíbené nástroje patří ShaderToy, kde najdete i řadu hotových shaderů, které můžete upravovat a zkoušet.

4.7 Vykreslovací knihovna Three.js

Pro práci s WebGL je možné použít knihovnu Three.js, která poskytuje jednodušší API pro práci s 3D grafikou. Přímo ve WebGL se uvnitř knihovny Three.js moc často nepracuje.

Three.js poskytuje řadu funkcionalit:

- Snadné načítání 3D modelů z různých formátů včetně podpory animací,
- Práci s kamerou,
- Práci se světly a materiály,
- Práci s texturami,
- Práci s geometrií,
- A mnoho dalšího.

Materiál již byl zmíněn vícekrát, ale nebyl detailně rozebrán. Materiál určuje, jak se má povrch objektu vykreslit. Určujeme například barvu, texturu, lesk, průhlednost a další vlastnosti. Zejména se tyto vlastnosti získávají z reálného světa a jejich simulace je velmi složitá.

Pojem, který se v rámci Three.js používá je mesh. Mesh je objekt, který má geometrii a materiál. Geometrie jsou již dříve zmíněné vrcholy a jejich propojení do trojúhelníků.

Three.js musíme importovat do našeho projektu. Existuje spoustu CDN a jiných zdrojů. Osobně doporučuji si vždy stáhnout Three.js lokálně.

Three.js má velmi jednoduchý způsob jak vytvořit základní scénu a vykreslovat ji do webové stránky.

Three.js ze základu počítá s tím, že se budou věci na scéně měnit, a proto používá vlastní smyčku pro vykreslování.

html

Základní scéna v Three.js

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/r128/three.min.js"></script>
<script>
const width = window.innerWidth;
const height = window.innerHeight;

const scene = new THREE.Scene();

const camera = new THREE.PerspectiveCamera(75, width / height, 0.1, 1000);

const renderer = new THREE.WebGLRenderer();
renderer.setClearColor(0x000000, 0);
renderer.setPixelRatio(window.devicePixelRatio);
renderer.setSize(width, height);
document.body.appendChild(renderer.domElement);

function animate() {
    renderer.render(scene, camera);
}

renderer.setAnimationLoop(animate);
```

Vytvoří základní scénu s kamerou a vykreslováním.

Ihned v této ukázce můžeme vidět, že v Three.js ihned začínáme používat 3D prostor.

4.7.1 Kamery

Three.js poskytuje několik typů kamer. Nejběžnější je perspektivní kamera (`PerspectiveCamera`), která simuluje lidské vidění. Druhou kamerou je ortografická kamera (`OrthographicCamera`), která nemá perspektivu a objekty se vykreslují stejně velké bez ohledu na jejich vzdálenost od kamery. Kamera má několik vlastností, které určují její chování.

U perspektivní kamery jsou to:

- `fov` (field of view) – úhel zorného pole v stup

- `aspect` – poměr stran (šířka / výška)
- `near` – nejbližší vzdálenost, kterou kamera vidí
- `far` – nejvzdálenější vzdálenost, kterou kamera vidí

U ortografické kamery jsou to:

- `left`, `right`, `top`, `bottom` – hranice zorného pole
- `near` – nejbližší vzdálenost, kterou kamera vidí
- `far` – nejvzdálenější vzdálenost, kterou kamera vidí

Kamera má také pozici a rotaci, které určují její umístění a orientaci v 3D prostoru.

Nastavení pozice kamery

```js
camera.position.set(0, 0, 5);
camera.lookAt(0, 0, 0);
```

Nastaví pozici kamery na (0, 0, 5) a otočí ji směrem k bodu (0, 0, 0).

4.7.2 Responzivita

Three.js neřeší automaticky změnu velikosti okna. Je tedy potřeba při změně velikosti okna aktualizovat velikost vykreslovače a poměr stran kamery.

Responzivita

```js
window.addEventListener("resize", () => {
    width = window.innerWidth;
    height = window.innerHeight;
    renderer.setSize(width, height);
    camera.aspect = width / height;
    camera.updateProjectionMatrix();
});
```

Aktualizuje velikost vykreslovače a poměr stran kamery při změně velikosti okna.

4.7.3 Objekty a scéna

Scéna je kontejner, který obsahuje všechny objekty, světla a kamery. Objekty jsou instance třídy `Mesh`, která má geometrii a materiál. Jednotlivé objekty bychom měli upravovat uvnitř funkce `animate`, která se volá při každém snímku.

Přidání a animace objektu

```js
const geometry = new THREE.BoxGeometry(1, 1, 1);
const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);
function animate() {
  cube.rotation.x += 0.01;
  cube.rotation.y += 0.01;
  cube.rotation.z += 0.01;
  renderer.render(scene, camera);
```

```
10 }
11 renderer.setAnimationLoop(animate);
```

Vytvoří zelený krychlový objekt a animuje jeho rotaci.

Mezi nejznámější geometrie patří:
- `BoxGeometry` – krychle,
- `SphereGeometry` – koule,
- `PlaneGeometry` – rovina,
- `CylinderGeometry` – válec,
- `TorusGeometry` – torus (kruh s dírou),
- `ConeGeometry` – kužel,
- A další.

Mezi nejznámější materiály patří:
- `MeshBasicMaterial` – základní materiál bez osvětlení,
- `MeshStandardMaterial` – fyzikálně založený materiál s podporou osvětlení,
- `MeshPhongMaterial` – materiál s podporou lesku a osvětlení,
- `MeshMatcapMaterial` – materiál s podporou environment mapy,
- `MeshLambertMaterial` – materiál s podporou difuzního osvětlení (nejpoužívanější),
- A další.

Každý materiál má definované své vlastnosti, které určují zobrazení. Skoro u každé lze najít základní vlastnost `color`, která určuje barvu materiálu. Pomocí vlastnosti `map` lze nastavit texturu, což je obrázek, který se na materiál aplikuje.

Jednotlivým objektům můžeme nastavovat:
- Pozici (`position`),
- Rotaci (`rotation`),
- Měřítko (`scale`),
- Viditelnost (`visible`),
- A další.

4.7.4 Světla

Světla jsou objekty, které ovlivňují to, jak se materiály vykreslí. Existuje několik typů světel:
- `AmbientLight` – globální světlo, které osvětluje všechny objekty,
- `DirectionalLight` – světlo, které simuluje sluneční světlo, má směr,
- `PointLight` – bodové světlo, které vyzařuje světlo ve všech směrech z jednoho bodu,
- `SpotLight` – reflektor, který vyzařuje světlo ve tvaru kužele,
- `HemisphereLight` – světlo, které simuluje oblohu a zem, má dvě barvy (horní a dolní),
- `RectAreaLight` – světlo, které simuluje světlo z obdélníkového zdroje (například okno),
- A další.

4.7.5 Vstup

Pro práce se vstupem od uživatele nemá Three.js žádné speciální funkce. Vše se řeší pomocí standardního DOM API.

Důležité přesto je, že Three.js poskytuje třídu `Raycaster`, která dovoluje detekovat kolize mezi objekty a paprsky. To je užitečné pro detekci kliknutí na objekty, výběr objektů a další interakce.

js

Detekce kliknutí na objekty

```
const raycaster = new THREE.Raycaster();
const mouse = new THREE.Vector2();
canvas.addEventListener("click", (event) => {
  mouse.x = (event.clientX / canvas.width) * 2 - 1;
  mouse.y = -(event.clientY / canvas.height) * 2 + 1;

  raycaster.setFromCamera(mouse, camera);
  const intersects = raycaster.intersectObjects(scene.children);
  if (intersects.length > 0) {
    console.log("Kliknuto na objekt:", intersects[0].object);
  }
});
```

Použití Raycasteru pro detekci kliknutí na objekty ve scéně.

Pokud potřebujeme získat pozici myši v rámci canvasu, je důležité zajistit správnou „hloubku“ dané souřadnice. Na získání pak využíváme funkci `unproject`, která převede 2D souřadnici na 3D souřadnici v rámci scény.

js

Převedení 2D pozice myši na 3D

```
const vector = new THREE.Vector3();
vector.set((mouse.x / canvas.width) * 2 - 1, -(mouse.y / canvas.height) * 2 + 1, 0.5);
vector.unproject(camera);
console.log("3D pozice myši ve scéně:", vector);
```

Použití unproject pro získání 3D pozice myši ve scéně.

4.8 Tvorba her

Pro tvorbu her jsme si ukázali všechny základní nástroje pro vykreslení. Ukázali jsme si také, jak pracovat s rozdělením objektů pro abstrakci a udržitelnost kódu.

Pro tvorbu her je potřeba ještě řešit další věci, jako jsou:

- Fyzika (kolize, gravitace, pohyb),
- Zvuk (přehrávání zvuků, hudby),
- Uživatelské rozhraní (tlačítka, menu, HUD),
- Správa zdrojů (načítání obrázků, zvuků, modelů),
- A další.

Je toho příliš. Pro tvorbu her je možné použít různé herní enginy, které poskytují hotové řešení pro většinu těchto věcí.

Na webu jsou nejznámější herní enginy:

- Phaser – 2D herní engine,
- PixiJS – 2D vykreslovací knihovna (není herní engine, ale často se používá jako základ pro herní enginy),
- PlayCanvas – 3D herní engine,
- Babylon.js – 3D herní engine a vykreslovací knihovna,
- A spousta dalších.

Přesto moc her na webu nevidíte, a pokud ano, tak nejsou ve výše uvedených enginech. Spousta nejznámější enginů (Godot, Unity) dovoluje export do webového prostředí. Vykreslování pak probíhá pomocí WebGL.

V Kapitola 5.1 se setkáme jestě s protokolem WebSocket, který lze použít pro tvorbu her s více hráči. V dané kapitole si ukážeme, jak na to.

4.9 Výběr vykreslení

Kdy použít vykreslení pomocí DOM:

Potřebujeme vykreslovat jednoduchou 2D grafiku, která se nemění často. Lze pro to použít vlastnosti CSS. Prvků není příliš mnoho. Jsme schopni je spravovat pomocí funkcí DOM API. Vykreslované prvky nejsou příliš komplexní. Je nám jedno jak rychle se vykreslí.

Kdy použít vykreslení pomocí canvas, kontextu 2D:

Potřebujeme vykreslovat 2D grafiku v reálném čase. Animace na sebe navazují a je potřeba je synchronizovat. Prvků je více a je potřeba je spravovat pomocí kódu. Chceme mít plnou kontrolu nad vykreslováním a animacemi. Chceme mít možnost vykreslovat složitější grafiku, než co dovoluje DOM a CSS.

Kdy použít vykreslení pomocí WebGL:

Potřebujeme vykreslovat 3D grafiku v reálném čase. Chceme využít hardwarovou akceleraci pro vykreslování grafiky. Chceme mít plnou kontrolu nad vykreslováním a animacemi. Chceme mít možnost vykreslovat složitější grafiku, než co dovoluje canvas 2D a DOM. Prvků je hodně a je potřeba je spravovat pomocí kódu.

Alternativní protokoly

5.1 WebSocket

WebSocket (WS) je jeden z nejmodernějších protokolů na webových stránkách. Podporuje ho skoro každý moderní prohlížeč a dovoluje obousměrnou komunikaci mezi klientem a serverem.

Staví na protokolu TCP. TCP je spolehlivý protokol, který zajišťuje doručení dat v pořádku a ve správném pořadí. Opakem je UDP, který je rychlejší, ale nezaručuje doručení ani pořadí. UDP se používá například pro streamování videa nebo online hry, kde je důležitější rychlost než spolehlivost.

TCP dovoluje jako jeden z mála protokolů na internetu udržovat otevřené spojení mezi klientem a serverem. To znamená, že jakmile je spojení navázáno, může klient i server posílat data kdykoliv bez nutnosti opětovného navazování spojení. To je ideální pro aplikace, které potřebují rychlou a obousměrnou komunikaci, jako jsou chatovací aplikace, online hry nebo real-time notifikace.

WebSockety přesto používají HTTP pro navázání spojení. Při požadavku na připojení se zasílá společně v HTTP hlavičkách hlavička `Upgrade`, která říká serveru, že klient chce přejít na protokol WebSocket. Pokud server podporuje WebSockety, odpoví také s hlavičkou `Upgrade` a spojení je navázáno.

Od tohoto momentu už komunikace probíhá mimo HTTP, i když stále přes stejné TCP spojení.

5.1.1 Použití na straně klienta

WebSockety musí implementovat přímo prohlížeč (protože to implementuje jiný protokol). Proto máme v prohlížeči (který to podporuje) přímo k dispozici objekt `WebSocket`, tedy API pro práci s WebSockety.

WebSockety si definují vlastní protokol, tedy používáme URL s protokolem `ws://` nebo `wss://` (pro zabezpečené spojení přes TLS). Pomocí API se můžeme připojit, posílat na server zprávy a přijímat zprávy ze serveru. Více věcí moc WebSockety nenabízí.

js

Příklad použití WebSocketů na straně klienta

```
1 const socket = new WebSocket('wss://example.com/socket');
2 socket.addEventListener('open', (event) => {
3   console.log('Připojeno k WebSocket serveru');
4   socket.send('Ahoj server!');
5 });
6 socket.addEventListener('message', (event) => {
7   console.log('Zpráva od serveru:', event.data);
8 });
9 socket.addEventListener('close', (event) => {
10  console.log('Spojení uzavřeno');
11 });
12 socket.addEventListener('error', (error) => {
13  console.error('Chyba WebSocket:', error);
14 });
```

Připojí se k WebSocket serveru, odešle zprávu a zpracovává příchozí zprávy.

WebSockety umožnují předávat následující typy dat:

- Text (řetězce),
- Binární data (ArrayBuffer, Blob),
- Pole (TypedArray, DataView).

Nejčastěji používáme textové zprávy a obsah formatujeme jako JSON.

5.1.2 Použití na straně serveru

Na straně serveru WebSockety nejsou součástí standardní knihovny Node.js. Proto je potřeba použít nějakou knihovnu, která WebSockety implementuje. Nejznámější knihovny jsou `ws`. Jedná se o velim podobné API jako na straně klienta. Co je jiné, je to, že na serveru můžeme mít více klientů připojených najednou. Často si tedy musíme ukládat tzv. Socket, což je objekt reprezentující jedno připojení klienta. To si můžeme uložit například do pole nebo mapy. A odesílat zprávy jen některým klientům.

js

Příklad použití WebSocketů na straně serveru

```
import http from 'http';
import WebSocket, { WebSocketServer } from 'ws';
const server = http.createServer();
const wss = new WebSocketServer({ server });
wss.on('connection', (ws) => {
  console.log('Nový klient připojen');
  ws.on('message', (message) => {
    console.log('Zpráva od klienta:', message);
    ws.send(`Server obdržel: ${message}`);
  });
  ws.on('close', () => {
    console.log('Klient odpojen');
  });
});
server.listen(8080, () => {
  console.log('Server naslouchá na portu 8080');
});
```

Vytvoří WebSocket server, který přijímá zprávy od klientů a odesílá jim odpovědi.

5.1.3 Knihovny pro WebSockety

Základní implementace v prohlížeči a na serveru jsou obecně velmi nízkoúrovňové. Proto často používáme knihovny, které nám usnadní práci s WebSockety. Největší problém je to, že WebSockety nedovolují:

- Automatické znovupřipojení při výpadku spojení,
- Zjištění, zda je spojení stále aktivní (ping/pong),
- Práci s více kanály (rooms, namespaces),
- Fallback na jiné technologie, pokud WebSockety nejsou dostupné (například v některých firemních sítích).

Nejznámější knihovnou, která řeší tyto problémy, je například Socket.IO. Socket.IO je knihovna, která poskytuje vyšší úroveň abstrakce nad WebSocket protokolem. Socket.IO

automaticky řeší vše, co je zmíněno výše, a navíc přidává další funkce, jako je například broadcast zpráv všem připojeným klientům.

Další knihovny, které stojí za zmínku, jsou například:

- `SockJS`,
- `SignalR`,
- `Primus`.

5.1.4 Práce s WebSockety

WebSockety lze používat pro různé účely. Jsou velmi praktické pro skoro každou komunikaci. Obecně by ale na nich neměla stát celá aplikace. Měly by být používány jako doplněk k HTTP API.

Vhodné účely pro WebSockety jsou například:

- Chatovací aplikace,
- Online hry,
- Real-time notifikace,
- Spolupráce na dokumentech (například Google Docs),
- Sledování polohy v reálném čase (například Uber),
- Streamování dat (například finanční data, senzory),
- Aplikace pro obchodování (například burzy).
- Spolupráce na kódu (například VSCode Live Share).

Tvorba dat, které se posílají přes WebSockety, je na nás. Obecně je dobré si vytvořit tzv. packety, které se dají jasně identifikovat a mají jasnou strukturu. Například můžeme mít packet pro chatovací zprávu, který vypadá takto:

- Každý packet má svůj typ (například `chat_message`, `user_joined`, `user_left`, nebo čísla),
- Každý packet má svůj payload (data), která jsou specifická pro daný typ.
- Packet může mít i metadata (například čas odeslání, ID uživatele, atd.).

json

Příklad struktury packetu pro WebSocket zprávu

```
{
  "type": "chat_message",
  "payload": {
    "message": "Ahoj všichni!",
    "user": "Jan",
    "timestamp": "2023-10-01T12:00:00Z"
  }
}
```

Packet obsahuje typ zprávy a její payload.

Uvnitř her, například, kdybychom chtěli měnit pozice hráče (objekt, který se nějak vykresluje), můžeme při každém pohybu odeslat packet s novou pozicí.

json

Příklad packetu pro pohyb hráče ve hře

```
{
  "type": "player_move",
  "payload": {
    "playerId": "player1",
```

```
5     "position": { "x": 100, "y": 200 },
6     "direction": "north",
7     "timestamp": "2023-10-01T12:00:05Z"
8   }
9 }
```

Packet obsahuje ID hráče, jeho novou pozici, směr a čas.

5.2 Oboustránná komunikace v HTTP

HTTP je protokol, který je navržen pro komunikaci typu požadavek-odpověď. Vždy tedy inciativa vychází ze strany klienta, který odešle požadavek na server a ten odpoví.

V průběhu let s tímto protokolem však vznikli požadavky na to, aby mohl server odesílat data i po své iniciativě. Nyní si představíme několik technik, jak toho dosáhnout. Celkově se pořád jedná o jednosměrnou komunikaci, ale obousměrnou simulujeme různými triky.

5.2.1 Polling

Polling je nejjednodušší technika, jak dosáhnout něčeho jako obousměrné komunikace. Klient pravidelně (například každých 5 sekund) odesílá požadavek na server, zda nemá nějaká nová data. Pokud ano, server je pošle v odpovědi. Pokud ne, odpoví prázdnou odpovědí.

Nevýhodou tohoto přístupu je zbytečné zatěžování serveru a sítě, pokud klient často posílá požadavky, ale server nemá žádná nová data. Tento problém je amplifikován tím, že tyto požadavky mohou chodit od více klientů současně. V případě velkých aplikací s mnoha uživateli může tento přístup vést k výraznému zatížení serveru.

5.2.2 Long Polling

Long Polling je vylepšená verze pollingu. V tomto případě klient odešle požadavek na server, ale pokud server nemá žádná nová data, neodpoví hned. Místo toho drží spojení otevřené, dokud nemá nějaká data k odeslání, nebo dokud nevyprší časový limit. Servery tento časový limit nastavují na několik desítek sekund. Když server má nová data, odpoví na požadavek a klient je zpracuje. Poté klient okamžitě odešle nový požadavek a celý proces se opakuje.

Tento přístup snižuje počet zbytečných požadavků na server, protože klient neodesílá nové požadavky, dokud nedostane odpověď. Přesto se požadavky stále volají opakovaně, což může být nevýhodné pro velmi aktivní aplikace.

5.2.3 Server-Sent Events

Server-Sent Events (SSE) je technologie, která umožňuje serveru posílat data klientovi přes jedno směrové spojení. Klíčovým rozdílem oproti long pollingu je, že spojení zůstává otevřené a server může posílat data kdykoliv, aniž by klient musel opakovaně odesílat požadavky. Klient obdrží vždy část (jednu zprávu), když server něco pošle.

Klient pošle požadavek na speciální endpoint, který vrátí odpověď s hlavičkou `Content-Type: text/event-stream`. Navíce se posílají různá nastavení, které například dovolují aby vůbec

zůstalo spojení otevřené. To se dělá hlavičkou `Cache-Control: no-cache` a `Connection: keep-alive`.

Server při žádosti o odeslání dat pošle data ve speciálním formátu, který začíná prefixem `data:` a končí dvěma novými řádky. Každá zpráva má své číslo, které se posílá prefixem `id:`. Tento formát umožňuje klientovi snadno zpracovat přijatá data.

Výhodou SSE je, že posílají opravdu jen data, když je to potřeba, což snižuje zátěž na server a síť. SSE dovoluje klientovi serveru poslat žádost o připojení od nějaké zprávy a tedy může přijímat zprávy i po výpadku spojení.

Nevýhodou je, že SSE je jednosměrný protokol, což znamená, že klient nemůže posílat data serveru přes toto spojení. Pro obousměrnou komunikaci je tedy potřeba použít jiný kanál, například klasické HTTP požadavky.

js

Příklad SSE endpointu na straně serveru

```
// ...
app.get('/events', (req, res) => {
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');
  setInterval(() => {
    res.write(`data: ${JSON.stringify({ time: new Date() })}\n\n`);
  }, 1000);
  res.on('close', () => {
    clearInterval(intervalId);
    res.end();
  });
});
```

Vytvoří endpoint, který každou sekundu posílá aktuální čas.

js

Příklad použití SSE na straně klienta

```
// ...
const eventSource = new EventSource('/events');
eventSource.addEventListener('message', (event) => {
  const data = JSON.parse(event.data);
  console.log('Nová data ze serveru:', data);
});
```

Připojí se k SSE endpointu a zpracovává přijatá data.

5.2.4 Rámec prohlížeče

Prohlížeč omezuje webové stránky v tom, co mohou dělat. To dělají z bezpečnostních důvodů, aby zabránily škodlivým stránkám v přístupu k citlivým datům nebo funkcím.

Jedním z těchto omezení je i to, jak mohou stránky komunikovat se servery na internetu. Jedno z největších omezení je to, že stránky mají omezené kolik požadavků najednou mohou poslat na jeden server. Respektive kolik otevřených TCP spojení může mít na jeden server.

Toto omezení je obvykle kolem 6-8 současných spojení na jednu doménu[15].

Tedy například použitím SSE nebo pollingů zablokujeme jedno celé připojení na webovou stránku. To může viditelně zpomalit načítání dalších zdrojů ze stejné domény, protože prohlížeč nemůže otevřít nové spojení, dokud některé z těchto spojení nezavře.

Dalším problémem long pollingu a SSE je, že pokud se dlouhou dobu žádné data neposílají, může server (nebo například proxy) uzavřít spojení kvůli nečinnosti. Obecně stejný problém bývá s tím, když servery zpracovávají požadavky příliš dlouho.

Spousta webových stránek se chrání pomocí různých proxy serverů, nejznámejší je Cloudflare, a ty aktivně omezují délku spojení i přes nastavení hlavičky `keep-alive`.

5.3 gRPC

gRPC je moderní, výkonný a univerzální rámec pro vzdálené volání procedur (RPC). Je vyvinutý společností Google a je založený na protokolu HTTP/2. gRPC umožnuje klientům a serverům komunikovat efektivně. Zařizuje serializaci a deserializaci dat pomocí protokolu Protocol Buffers (Protobuf), což je efektivní binární formát. gRPC podporuje různé typy komunikace, včetně jednosměrných a obousměrných streamů.

Na webu se gRPC používá málo, protože neexistuje moc podpory na straně prohlížeče. Používáme ho tedy často pro komunikaci mezi servery a tam, kde je možné používat vlastní protokoly.

Zmínka o gRPC je zde hlavně pro úplnost a pro představu, že existují i jiné možnosti než čisté HTTP.

proto

Příklad definice služby v gRPC

```proto
syntax = "proto3";
package example;
service UserService {
  rpc GetUser (GetUserRequest) returns (GetUserResponse);
}
message GetUserRequest {
  string user_id = 1;
}
message GetUserResponse {
  string name = 1;
  int32 age = 2;
}
```

Definuje službu UserService s metodou GetUser, která přijímá GetUserRequest a vrací GetUserResponse.

5.4 Zásady komunikace

Již z lekce backendu ze třetího ročníku víme, že komunikace mezi klientem a server by měla dodržovat určité zásady.

[15] Připomeňte si význam AJAX a proč existuje. Podobně je vhodné si pripomenout asynchronní JavaScript.

Nejdůležitější lekcí je to, že datům, které obdržíme zvenčí, nemůžeme věřit. To znamená, že musíme vždy ověřit a validovat data, která přicházejí od uživatelů nebo z jiných nedůvěryhodných zdrojů. To platí pro všechny typy komunikace, ať už používáme HTTP, WebSockety, SSE nebo jiné protokoly. Důležité je taktéž omezovat velikost přijatých dat, aby nedošlo k přetížení serveru.

Pro dobrý návrh komunikace je taktéž dobré mít jasně definované rozhraní (API), které popisuje, jaké požadavky může klient posílat a jaké odpovědi může očekávat.

Celkově bychom neměli zapomínat na základní principy bezpečnosti a ochrany dat při návrhu jakékoliv komunikace.

Webové vyhledávače a optimalizace

6.1 Problém vyhledávání

Web je obrovský a neustále roste. Existuje na něm obrovské množství informací, které jsou často neuspořádané a nekvalitní. Najít na webu konkrétní informaci je velmi těžké. Proto vznikly webové vyhledávače, které nám pomáhají najít to, co hledáme.

Problém vyhledávání informací však není vázán pouze na webové stránky. I mimo web je potřeba najít konkrétní informaci v různých dokumentech, databázích a dalších zdrojích. Vyhledávání je tedy obecný problém, který se řeší v mnoha oblastech informatiky.

Jakožto webový inženýři se setkáte v praxi s tím, že budete potřebovat v textu něco najít. Mimo kontext webových vyhledávačů budete hledat v textu buď ručně, nebo pomocí nějakého nástroje. Nejznámější nástroj na validaci a hledání v textu je tzv. regulární výraz (regular expression, regex). Ten umí hledat v textu podle nějakého vzoru. Je vhodné se naučit s regulárními výrazy pracovat.

6.2 Funkcionality vyhledávačů

Webové vyhledávače jsou složitý software, který musí umět mnoho věcí. Z WBA ze druhého ročníku již víme, že vyhledávač je aktivní program, který neustále prochází webové stránky a indexuje je. Webové stránky prochází pomocí svých crawlerů (pavouci, boti). Ti mají seznam stránek, který mají procházet. Při každé návštěvě si ze stránky vezmou další odkazy, které mají navštívit. Většina vyhledávačů však do této fronty vkládá i stránky, které již navštívila dříve pro aktualizaci obsahu. Vyhledávačům můžeme říct, jak často chceme, aby stránky navštěvovaly[16], viz kapitola Kapitola 6.6.

Vyhledávače často získávají i domény od registrů domén, aby mohly navštěvovat i domény, na které ještě nikdo neukazuje. Podobně můžeme často vyhledávačům podstrčit informace, aby naší stránku poprvé navštívili.

Logicky si umíme představit, že stránky, které jsou populární, bude chtít vyhledávač navštěvovat častěji. Populární stránky jsou ty, na které vede hodně odkazů z jiných stránek a chodí na ně hodně uživatelů.

Při návštěvě stránky si vyhledávač stáhne její obsah a pokusí se z něj získat informace, které mu pomohou stránku zařadit do svého indexu. Index je databáze, ve které se ukládají potřebná data a odkaz, na kterém se stránka nachází. Ze stránky si umí vyhledávač vzít například její název, metatagy a obsah. Problém je, že obsahuje je hodně a často neví, co z toho je důležité. Tady přichází na řadu sémantika[17], která se snaží pochopit význam obsahu. Obecně je těžké přesně říct jaká data si vyhledávač z obsahu vezme.

Když uživatel zadá do vyhledávače dotaz, ten se pokusí najít v indexu stránky, které nejlépe odpovídají zadanému dotazu. Po nalazení stránek, které odpovídají dotazu je nutné, aby dané stránky seřadil podle relevance. Relevančních faktorů je mnoho, často se jedná o:

- Popularitu stránky,
- Osobní preference uživatele (historie vyhledávání, lokalita, zařízení),
- Obsah stránky (slova v názvu, metatagy, obsah),
- Čerstvost obsahu (novější obsah může být relevantnější pro aktuální události),
- Kvalitu obsahu (délka, struktura, gramatika),

[16] Jak se rozhodnou, je na nich.

[17] A proto byla vždy důležitá.

- Uživatelské chování (míra prokliku, doba strávená na stránce, míra okamžitého opuštění),
- Odkazy na stránku (kvalita a počet zpětných odkazů z jiných stránek),
- Sociální signály (sdílení na sociálních sítích, komentáře),
- Technické aspekty (rychlost načítání, mobilní přívětivost, bezpečnost),
- Strukturovaná data (schema.org, Open Graph, JSON-LD),
- Uživatelské recenze a hodnocení,
- Lokalizace (geografická poloha uživatele a lokalizace obsahu),
- Kontext dotazu (historie vyhledávání, aktuální trendy),
- Personalizace (přizpůsobení výsledků na základě uživatelských preferencí),
- A mnoho dalších.

Prohlížeče tedy pro posouzení stránky nejen berou obsah, ale aktivně stránku analyzují a snaží se pochopit její význam. To je u některých stránkách těžké a vyhledávače se často mýlí – musíme jim tedy pomoci. To se dělá pomocí optimalizace pro vyhledávače, o které se budeme bavit později, viz Kapitola 6.6.

Samotné vyhledání v daném indexu je pak otázka rychlého algoritmu, který umí najít relevantní stránky. Tyto algoritmy mohou používat řadu datových struktur a triků, aby bylo vyhledávání co nejrychlejší. Těmto algoritmům či systémum se říká model vyhledávání. Do modelu vyhledávání se poslední léta přidávají i umělé inteligence, o kterých se ale bavit nebudeme.

Vytvořit dobrý vyhledávač je velmi složitý úkol, který vyžaduje hodně znalostí a zkušeností. Je potřeba hodně ladění a testování, aby vyhledávač fungoval dobře. Vašim cílem, jako tvůrce webového vyhledávače je, aby uživatel přišel na váš vyhledávač, a vždy našel to, co hledá.

Proč se o tomto vůbec bavíme? Protože jako tvůrci webových stránek chceme, aby na naše stránky uživatelé chodili. Přirozený přístup na web zadáním URL do prohlížeče je omezený, celkově odhady říkají, že na web přijde 10-20 % uživatelů tímto způsobem. Většina uživatelů přijde na web pomocí vyhledávače nebo odkazu z jiného webu, např. sociálních sítí. Je důležité vědět, jak tyto vyhledávače fungují a jak jim můžeme pomoci, aby na naše stránky uživatele dostaly.

Webové vyhledávače mají blízko samotnému vyhledávání, které probíhá často na každému webu. Když už někdo na web přijde, chceme, aby na stránce zůstal a mohl si případně najít, co hledá a nepoužil k tomu vyhledávač. Tedy stránku odešel a našel si informace jinde.

Takže postupně vyhledávače:
1. Procházejí webové stránky (crawler),
2. Analyzují jejich obsah, sémantiku a další data, které ukládají (indexer[18]),
3. Umožňují uživatelům vyhledávat v jejich indexu, kdy se vyhledává (searching) a výsledky se hodnotí (ranking).

6.2.1 Získávání dat

Pokud již máme webovou stránku, a její HTML obsah[19], musíme z něj získat důležitá data.

Způsobů, jak získat data, je spousta. To nejjednodušší co existuje, je, že vyhledávač z HTML získá veškerý textový obsah. Nejjednoduší způsob jak to udělat, je odstranit veškeré HTML tagy a získat pouze text. To ale zanechá velmi mnoho šumu, který je potřeba nějakým

[18] Pokud stránka není indexována, tak na ni vyhledávač uživatele nedostane i kdyby měla úžasné SEO.

[19] Připomeňte si, jaký význam na CSS – pouze dekorační a to vyhledávače neřeší. Řeší tedy často to, že obsah není vidět, viz Kapitola 6.5

způsobem odstranit. Jedná se třeba o záznamy, které se často opakují na stránce, jako jsou patičky, hlavičky, menu, reklamy a podobně. To stejné nejspíše zanechají tagy i mimo tělo stránky, jako jsou komentáře, skripty a styly.

Proto je lepší HTML zparsovat, to znamená, že k němu, respektive k jeho hierarchické struktuře, budeme přistupovat programově. Zjednodušeně si můžeme vytvořit vlastní DOM, který nám umožní přistupovat k jednotlivým částem stránky. Na to existuje mimo prohlížeč dobrá knihovna `jsdom`.

Použití jsdom

```js
import { JSDOM } from 'jsdom';
const dom = new JSDOM(htmlString);
const document = dom.window.document;
const bodyText = document.body.textContent;
console.log(bodyText);
```

Získá se textový obsah z těla stránky.

Vlastnost `textContent` nám vrátí čistý text bez HTML tagů s takovým obsahem, který lze vykreslit. To je ale znovu poměrně hrubý způsob, jak data získat. Lépe je, když si z HTML vybereme konkrétní části, které nás zajímají. To můžeme udělat pomocí selektorů, a třeba si vybrat pouze věci uvnitř tagu `main`, a potom třeba získat data z konkrétních tagů, jako jsou `h1`, `h2`, `p` a podobně.

Výběr konkrétních částí

```js
const main = document.querySelector('main');
const headings = main.querySelectorAll('h1, h2, h3, h4, h5, h6');
const paragraphs = main.querySelectorAll('p');
let content = '';
for(let element of [...headings, ...paragraphs]) {
  content += element.textContent + '\n';
}
console.log(content);
```

Získá se textový obsah z tagů v hlavní části stránky.

Další data, která musíme získat, jsou metatagy. Přesněji nás zajímají metatagy `title`, `description` a `keywords`.

6.2.2 Zpracování dat

Ze získaného obsahu je nutné získat jednotlivá slova. Nám je totiž jedno, kolikrát se na stránce slovo vyskytuje, ale jaká slova se na stránce vyskytují a v jaké frekvenci.

Kdybychom počítali pouze absolutní četnost slov, tak se někomu velmi jednoduše povede podstrčit stránku, která bude obsahovat všechna možná slova s velko četností. Proto se používá relativní četnost, která bere v úvahu i celkový počet slov na stránce. Relativní četnost slova se vypočítá jako podíl absolutní četnosti a celkového počtu slov na stránce.

Jednotlivá slova můžeme získat tím, že obsah rozdělíme podle mezer, interpunkce a obecně dalších rozdělovačů a tzv. bílých znaků.

```js
```

Počítání slov

```js
const text = "Toto je nějaký text, který obsahuje slova. Některá slova se opakují, například slovo 'slovo'.";
const words = text.toLowerCase().split(/\W+/).filter(Boolean);
const wordCount = {};
for (let word of words) {
  wordCount[word] = (wordCount[word] || 0) + 1;
}
console.log(wordCount);
```

Získá se objekt s počtem výskytů jednotlivých slov.

Počítání relativní četnosti

```js
const relativeWordCount = {};
const totalWords = words.length;
for (let word in wordCount) {
  relativeWordCount[word] = wordCount[word] / totalWords;
}
console.log(relativeWordCount);
```

Získá se objekt s relativní četností jednotlivých slov.

Ve výše uvedených příkladech si můžeme všimnou velkého problému v některých jazycích, jako je čeština. Slova se v češtině často ohýbají, což znamená, že stejné slovo může mít mnoho různých tvarů. Například slovo „kočka“ může mít tvary „kočky“, „kočce“, „kočku“ a podobně. Uživateli je ale často jedno, v jakém tvaru je slovo použito, hlavně když hledá. Kdybychom tento problém neřešili, tak uživatel hledající „kočka“ by na stránce s textem „kočky“ nic nenašel. Tento problém se řeší pomocí stemmingu nebo lemmatizace. Stemming je proces, při kterém se slovo zkrátí na svůj základní tvar, tzv. kmen. Lemmatizace je proces, při kterém se slovo převede na svůj základní tvar, tzv. lemma, pomocí slovníku a gramatických pravidel.

Například pomocí stemmingu se slovo „kočky“ zkrátí na „kočk“. U lemmatizace se slovo „kočičího“ převede na „kočičí“.

Na stemming a lemmatizaci existuje mnoho knihoven, například `snowball-stemmer` pro stemming a `wink-lemmatizer` pro lemmatizaci.

6.3 Modely vyhledávání

Pokud již máme zpracovaný obsah stránky, tak tyto slova můžeme uložit do indexu.

Každý model vyhledávání používá svůj vlastní způsob na uložení dat do indexu – či databáze.

My si ukážeme jen dva (resp. tři, protože třetí je úprava jednoho z nich) nejzákladnějších modelů vyhledávání.

6.3.1 Booleovský model

Nejjednodušší model vyhledávání je Booleovský model. Jak víte, tak Boolean je datový typ, který může nabývat pouze dvou hodnot – pravda a nepravda. Na tom funguje i tento celý model.

V tomto modelu si do databáze ukládáme pouze stránku a to, jaká slova obsahuje a jaká ne. Reálně si to můžeme představit jako velkou tabulku, kde buď řádky nebo sloupce jsou jednotlivá slova a na průsečíku je hodnota true nebo false, podle toho, jestli stránka dané slovo obsahuje[20].

Pro každou stránku tedy určujeme i to, že dané slovo neobsahuje. To je ale velký problém, protože slov je velké množství, a to ještě díky tomu, že jazyků je mnoho. Obecně v praxi říkáme, že tato tabulka je řídká[21], protože většina slov na stránce není. To je ale v pořádku, protože se s tím dá pracovat, viz níže.

Stránka	**pes**	**kočka**	**auto**
Stránka A	true	false	true
Stránka B	false	true	false
Stránka C	true	true	false
Stránka D	false	false	true

Dotazování při hledání je pak velmi jednoduché. Uživatel totiž musí zadávat dotazy pomocí Booleovských operátorů AND, OR a NOT[22]. Například dotaz „pes AND kočka“ vrátí všechny stránky, které obsahují slova „pes“ i „kočka“. Dotaz „pes OR kočka“ vrátí všechny stránky, které obsahují slovo „pes“ nebo „kočka“. Dotaz „pes AND NOT kočka“ vrátí všechny stránky, které obsahují slovo „pes“, ale neobsahují slovo „kočka“. Operátory ale často chceme více kombinovat, například „pes AND (kočka OR auto)“.

Tyto dotazy se pak jednoduše vyhodnotí nad tabulkou. Stačí totiž projet každý řádek a zjistit, jestli splňuje podmínky dotazu. Výsledkem hledání jsou stránky, které vyhovují dotazu.

Tento model je velmi jednoduchý a rychlý, ale má mnoho nevýhod. Největší nevýhodou je, že uživatel musí znát Booleovské operátory a musí umět s nimi pracovat. Případné je nutné implementovat rozhraní, které dovoluje dotazy pouze zadávat pomocí těchto operátorů. K tomu tyto operátory musíme umět zparsovat a vyhodnotit. Druhou nevýhodou je, že tento model neřeší relevanci. To znamená, že všechny stránky, které vyhovují dotazu, jsou považovány za stejně relevantní. Frekvence výskytu slov na stránce se neřeší, stejně jako další faktory, které by mohly ovlivnit relevanci. Tento model je tedy vhodný pouze pro velmi jednoduché vyhledávání, kde uživatel přesně ví, co hledá, ale neví kde data jsou[23].

Implementace Booleovského modelu

```js
const database = {
  "Stránka A": { pes: true, kočka: false, auto: true },
  "Stránka B": { pes: false, kočka: true, auto: false },
  "Stránka C": { pes: true, kočka: true, auto: false },
  "Stránka D": { pes: false, kočka: false, auto: true },
};
```

[20] Pro pokročilé si můžeme představit, že se neukládá true/false, ale pouze 0/1, a tedy celé to může být matice.
[21] Řídká matice, angl. sparse matrix, je matice.
[22] Případně dalších, ale to záleží na implementaci.
[23] Častá ukázka bývají knihovní katalogy a právnické vyhledávání v zákonech a dalších právních dokumentech.

```
 7 const splitters = [' AND NOT ', ' AND ', ' OR '];
 8 function evaluateQuery(pageData, query) {
 9   let operator = null;
10   for (let splitter of splitters) {
11     if (query.includes(splitter)) {
12       operator = splitter.trim();
13       break;
14     }
15   }
16
17   const [left, right] = query.split(operator).map(s => s.trim());
18   switch (operator) {
19     case 'AND':
20       return pageData[left] && pageData[right];
21     case 'OR':
22       return pageData[left] || pageData[right];
23     case 'AND NOT':
24       return pageData[left] && !pageData[right];
25     default:
26       throw new Error('Neznámý operátor ' + operator);
27   }
28 }
29 function search(database, query) {
30   const results = [];
31   for (let page in database) {
32     if (evaluateQuery(database[page], query)) {
33       results.push(page);
34     }
35   }
36   return results;
37 }
38 console.log(search(database, 'pes AND kočka')); // ["Stránka C"]
39 console.log(search(database, 'pes OR kočka')); // ["Stránka A", "Stránka
   B", "Stránka C"]
40 console.log(search(database, 'pes AND NOT kočka')); // ["Stránka A"]
```

Implementace Booleovského modelu vyhledávání, který dovoluje dotaz pomocí jednoho operátoru.

6.3.1.1 Uchování dat

Tento model je velmi jednoduchý, ale jak jsme si řekli, tak tabulka je velmi řídká. Pro ještě lepší optimalizaci můžeme místo ukládání tabulky ukládat pouze ta slova, která mají ke stránce hodnotu true.

Tomuto se říká inverzní index. To si můžeme představit jako objekt, kde klíče jsou slova a hodnoty jsou pole stránek, které dané slovo obsahují.

Slovo	Stránky
pes	Stránka A, Stránka C
kočka	Stránka B, Stránka C
auto	Stránka A, Stránka D

To velmi zmenší množství ukládaných dat a navíce nám to zjednoduší vyhledávání. Při zpracování dotazu se nám stačí podívat dle slova, které najdeme, do indexu a získat pole stránek, které dané slovo obsahují. S druhým slovem jenom dle operátoru spojíme výsledky. To je velmi rychlé a efektivní.

Implementace inverzního indexu

```js
const invertedIndex = {
  pes: ["Stránka A", "Stránka C"],
  kočka: ["Stránka B", "Stránka C"],
  auto: ["Stránka A", "Stránka D"],
};
const splitters = [' AND NOT ', ' AND ', ' OR '];
function search(index, query) {
  let operator = null;
  for (let splitter of splitters) {
    if (query.includes(splitter)) {
      operator = splitter.trim();
      break;
    }
  }

  const [left, right] = query.split(operator).map(s => s.trim());
  const leftPages = index[left] || [];
  const rightPages = index[right] || [];
  switch (operator) {
    case 'AND':
      return leftPages.filter(page => rightPages.includes(page));
    case 'OR':
      return Array.from(new Set([...leftPages, ...rightPages]));
    case 'AND NOT':
      return leftPages.filter(page => !rightPages.includes(page));
    default:
      throw new Error('Neznámý operátor ' + operator);
  }
}
console.log(search(invertedIndex, 'pes AND kočka')); // ["Stránka C"]
console.log(search(invertedIndex, 'pes OR kočka')); // ["Stránka A",
  "Stránka B", "Stránka C"]
console.log(search(invertedIndex, 'pes AND NOT kočka')); // ["Stránka A"]
```

Implementace vyhledávání pomocí inverzního indexu, který dovoluje dotaz pomocí jednoho operátoru.

6.3.2 Rozšíření Booleovského modelu

Problémy Booleovského modelu jsou jasné. To vedlo k tomu, že byl popsán rozšířený model, který řeší největší problém a to je relevance.

Rozšířený Booleovský model totiž místo true/false používá relativní četnost slova na stránce. Dříve jsme si ukázali, jak se relativní četnost počítá. Pro moderní vyhledávače se ale používá ještě modifikovaná relativní četnost, která bere v úvahu i to, jak často se slovo vyskytuje na všech stránkách.

Tento model se nazývá TF-IDF (Term Frequency-Inverse Document Frequency).

$$\text{TF} = \frac{\text{absolutní četnost slova na stránce}}{\text{celkový počet slov na stránce}}$$

$$\text{IDF} = \log\left(\frac{\text{celkový počet stránek}}{\text{počet stránek, které obsahují slovo}}\right)$$

$$\text{TF-IDF} = \text{TF} * \text{IDF}$$

Tento model tedy bere v úvahu jak často se slovo vyskytuje na stránce (TF), tak i to, jak často se slovo vyskytuje na všech stránkách (IDF). Pro každou buňku v tabulce tedy místo true/false máme číslo, které vypočítáme pomocí TF-IDF.

Stránka	**pes**	**kočka**	**auto**
Stránka A	0.462	0.000	0.231
Stránka B	0.000	0.693	0.000
Stránka C	0.347	0.347	0.000
Stránka D	0.000	0.000	0.693

Dotazování je pak podobné jako u Booleovského modelu. Pořád se používají Booleovské operátory, ale místo true/false se používají čísla. Při vyhodnocení dotazu se pak místo filtrování stránek podle true/false, se stránky sečtou podle operátoru. Například dotaz „pes AND kočka“ vrátí všechny stránky, které obsahují slova „pes“ i „kočka“, ale místo toho, aby vrátil pouze stránky, které obsahují obě slova, tak vrátí součet TF-IDF hodnot pro obě slova. Dotaz „pes OR kočka“ vrátí všechny stránky, které obsahují slovo „pes“ nebo „kočka“, ale místo toho, aby vrátil pouze stránky, které obsahují alespoň jedno slovo, tak vrátí součet TF-IDF hodnot pro obě slova. Dotaz „pes AND NOT kočka“ vrátí všechny stránky, které obsahují slovo „pes“, ale neobsahují slovo „kočka“, ale místo toho, aby vrátil pouze stránky, které obsahují slovo „pes“ a neobsahují slovo „kočka“, tak vrátí TF-IDF hodnotu pro slovo „pes“. Výsledkem hledání jsou stránky seřazené podle součtu TF-IDF hodnot takových slov, které se nacházejí v dotazu.

Implementace rozšířeného Booleovského modelu

```js
const tfidfIndex = {
  "Stránka A": { pes: 0.462, kočka: 0.000, auto: 0.231 },
  "Stránka B": { pes: 0.000, kočka: 0.693, auto: 0.000 },
  "Stránka C": { pes: 0.347, kočka: 0.347, auto: 0.000 },
  "Stránka D": { pes: 0.000, kočka: 0.000, auto: 0.693 },
};
const splitters = [' AND NOT ', ' AND ', ' OR '];
```

```
 8 function search(index, query) {
 9   let operator = null;
10   for (let splitter of splitters) {
11     if (query.includes(splitter)) {
12       operator = splitter.trim();
13       break;
14     }
15   }
16
17   const [left, right] = query.split(operator).map(s => s.trim());
18   const scores = {};
19   for (let page in index) {
20     const leftScore = index[page][left] || 0;
21     const rightScore = index[page][right] || 0;
22     let score = 0;
23     switch (operator) {
24       case 'AND':
25         score = leftScore && rightScore ? leftScore + rightScore : 0;
26         break;
27       case 'OR':
28         score = leftScore + rightScore;
29         break;
30       case 'AND NOT':
31         score = leftScore && !rightScore ? leftScore : 0;
32         break;
33       default:
34         throw new Error('Neznámý operátor ' + operator);
35     }
36     if (score > 0) {
37       scores[page] = score;
38     }
39   }
40   return Object.entries(scores).sort((a, b) => b[1] -- a[1]).map(entry =>
   entry[0]);
41 }
42 console.log(search(tfidfIndex, 'pes AND kočka')); // ["Stránka C"]
43 console.log(search(tfidfIndex, 'pes OR kočka')); // ["Stránka C, "Stránka
   B", "Stránka A"]
44 console.log(search(tfidfIndex, 'pes AND NOT kočka')); // ["Stránka A"]
```

Implementace rozšířeného Booleovského modelu vyhledávání pomocí TF-IDF, který dovoluje dotaz pomocí jednoho operátoru.

Výsledky nemusíme rovnou řadit, ale použit je pro další řazení, viz kapitola o tom, jak vyhledávače řadí stránky, viz Kapitola 6.2.

6.3.3 Vektorový model

Druhý model, který si ukážeme, je vektorový model. Tento model je podobný rozšířenému Booleovskému modelu, ale místo toho, aby používal Booleovské operátory, používá matema-

tické operace s vektory. Používá řadu matematických konceptů, jako jsou vektory, kosinusová podobnost a další.

V tomto modelu si každou stránku představíme jako vektor v n-rozměrném prostoru, kde n je počet unikátních slov[24] ve všech dokumentech. Každá složka vektoru pak odpovídá relativní četnosti daného slova na stránce, znovu to pravděpodobně vždy bude TF-IDF.

Dotaz uživatele v tomto modelu představíme také jako vektor ve stejném prostoru. Ten se ale získá jinak. Dotaz je často už pouze několik slov, žádné operátory. Pro každý dotaz si tedy spočítáme relativní četnost slov v dotazu a vytvoříme z toho vektor.

Samotné dotazování potom probíhá tak, že spočítáme vzdálenost mezi vektorem dotazu a vektorem každé stránky. Napadne vás použít Eukleidovskou vzdálenost ($d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - -p_i)^2}$), ta ale není vhodná. Bere totiž v úvahu i velikost vektoru, což není to, co chceme, protože i velmi podobná stránka dotazu může být vzdálená velmi daleko. Co nás zajímá je spíše směr vektoru, tedy jak moc jsou si vektory podobné. K tomu se používá tzv. kosínova podobnost.

$$\text{sim} = \frac{p \cdot q}{\|p\| \cdot \|q\|} = \frac{\sum_{i=0}^{n}(p_i \cdot q_i)}{\sqrt{\sum_{i=0}^{n}(p_i^2)} \cdot \sqrt{\sum_{i=0}^{n}(q_i^2)}}$$

Kde p a q jsou vektory, p_i jsou složky vektorů, resp. hodnota v dané dimenzi, $\|p\|$ jsou velikosti vektorů. V zápisu se používá násobení $\cdot$ pro obyčejné násobení tak i pro skalární součin vektorů.

Po vyhledání máme seznam stránek a jejich skóre podobnosti s dotazem. Ten můžeme použít pro seřazení stránek podle relevance.

Toto je nejspíše nejpoužívanější model vyhledávání pro většinu moderních vyhledávačů. Dovoluje totiž velmi dobře určit relevanci stránky k dotazu a nevyžaduje od uživatele, aby uměl pracovat s Booleovskými operátory.

V ukládání vektorů bychom jsme se zase setkali s tím, že jednotlivé složky vektoru budou často nulové. Proto můžeme znovu používat inverzní index.

Implementace vektorového modelu

```js
function dotProduct(vecA, vecB) {
  let product = 0;
  for (let key in vecA) {
    if (vecB.hasOwnProperty(key)) {
      product += vecA[key] * vecB[key];
    }
  }
  return product;
}
function magnitude(vec) {
  let sum = 0;
  for (let key in vec) {
    sum += vec[key] * vec[key];
  }
  return Math.sqrt(sum);
```

[24] Tedy stejně, jako počet sloupců v Booleovském modelu.

```
}
function cosineSimilarity(vecA, vecB) {
  const magA = magnitude(vecA);
  const magB = magnitude(vecB);
  return magA && magB ? dotProduct(vecA, vecB) / (magA * magB) : 0;
}
const tfidfIndex = {
  "Stránka A": { pes: 0.462, kočka: 0.000, auto: 0.231 },
  "Stránka B": { pes: 0.000, kočka: 0.693, auto: 0.000 },
  "Stránka C": { pes: 0.347, kočka: 0.347, auto: 0.000 },
  "Stránka D": { pes: 0.000, kočka: 0.000, auto: 0.693 },
};
function createQueryVector(query) {
  const words = query.toLowerCase().split(/\W+/).filter(Boolean);
  const wordCount = {};
  for (let word of words) {
    wordCount[word] = (wordCount[word] || 0) + 1;
  }
  const totalWords = words.length;
  const queryVector = {};
  for (let word in wordCount) {
    queryVector[word] = wordCount[word] / totalWords;
  }
  return queryVector;
}
function search(index, query) {
  const queryVector = createQueryVector(query);
  const scores = {};
  for (let page in index) {
    const pageVector = index[page];
    const score = cosineSimilarity(pageVector, queryVector);
    if (score > 0) {
      scores[page] = score;
    }
  }
  return Object.entries(scores).sort((a, b) => b[1] -- a[1]).map(entry =>
entry[0]);
}
console.log(search(tfidfIndex, 'pes kočka')); // ["Stránka A", "Stránka
C"]
console.log(search(tfidfIndex, 'pes auto')); // ["Stránka A", "Stránka
D", "Stránka C"]
console.log(search(tfidfIndex, 'kočka auto')); // ["Stránka D", "Stránka
A"]
```

Implementace vektorového modelu vyhledávání pomocí kosínové podobnosti.

6.3.4 Další modely a rozšíření

Mezi další populární modely vyhledávání patří:

- Latentní sémantické indexování (LSI), které se snaží najít skryté vztahy mezi slovy a dokumenty pomocí matematických metod,
- Pravděpodobnostní modely, které se snaží odhadnout pravděpodobnost, že daný dokument je relevantní pro daný dotaz,
- Modely založené na umělé inteligenci, které využívají strojové učení a hluboké učení k lepšímu pochopení dotazů a dokumentů, viz později.

Tyto modely jsou složitější a vyžadují více výpočetního výkonu, ale mohou nabídnout lepší výsledky. Často se také kombinují různé modely dohromady, aby se dosáhlo co nejlepších výsledků.

Webové vyhledávače se snaží taktéž poskytnout co nejvíce nástrojů těm, co hledají. Nabízejí třeba filtrace různých typů obsahu, časové omezení, hledání v konkrétních doménách, kategoriích a podobně.

Například vyhledávací engine Google nabízí možnost do dotazu zadat různé speciální klíčová slova, která způsobí omezení vyhledávání. Nejčastější zápisy:

- Označení fráze uvozovkami, např. „nějaká fráze“, což způsobí, že se hledá přesná shoda této fráze,
- Vyloučení slova pomocí mínus, např. jablko -zelené, což způsobí, že se hledá slovo jablko, ale nesmí se vyskytovat slovo zelené,
- Omezení na konkrétní webovou stránku pomocí site:, např. site:example.com jablko, což způsobí, že se hledá slovo jablko pouze na stránkách example.com,
- Hledání podle typu souboru pomocí file:, např. jablko file:pdf, což způsobí, že se hledá slovo jablko pouze v PDF dokumentech,
- Hledání podle URL pomocí inurl:, např. inurl:jablko, což způsobí, že se hledá slovo jablko pouze v URL adresách stránek.

6.4 Hodnocení stránek

Z dřívějších kapitol víme, že samotné vyhledání relevantních stránek je pouze část úkolu. Poté potřebujeme stránky nějak více seřadit, aby odpovídali zájmu uživatele.

To, že stránka obsahuje dané slovo, ještě neznamená, že je pro uživatele relevantní. Stránka totiž může obsahovat slovo náhodou, může se záměrně jednat o stránku, která se snaží uživatele oklamat, nebo může být stránka prostě špatná a uživatel na ní nic nenajde. Proto je potřeba stránky nějak ohodnotit a seřadit podle relevance.

Nejznámější vyhledávače se snaží sledovat mnoho faktorů, mezi které patří:

- Stav stránky předchozí období, například jak často se obsah na stránce mění,
- Kolik lidí na stránku přijde a na vyhledávač se již nevrátí,
- Zda je stránka online a dostupná, jak často padá,
- Rychlost načtení a lokace stránky vůči lokaci uživatele,
- Zda je stránka přátelská k mobilním zařízením (např. přístup z telefonu),
- Zda stránka nedělá podvodné techniky,
- Zda je text na stránce opravdu viditelný uživateli,
- A další.

Dříve byl nejdůležtějším faktorem to, jak je stránka oblíbená mezi ostatními stránkami. Obecný konsensus je, že pokud na stránku odkazuje mnoho jiných stránek, tak je pravděpodobně relevantní. Při průchodu webem tedy vyhledávače sledují i odkazy na stránkách a vytváří si tak síť odkazů mezi stránkami. To se dá opět modelovat jako graf, kde uzly jsou stránky a hrany

jsou odkazy mezi stránkami. Tento graf se pak dá analyzovat různými algoritmy, například PageRank, který hodnotí stránky podle toho, kolik a jak kvalitních stránek na ně odkazuje.

PageRank je založen na myšlence, že důležitější stránky jsou ty, na které odkazuje mnoho jiných důležitých stránek. A důležitá stránka je taková stránka, na kterou odkazuje mnoho jiných důležitých stránek. Je to tedy rekurzivní definice, kterou se dá vyřešit pomocí iterativního algoritmu. Iterativní algoritmus běží po dobu, než se hodnoty PageRank stránek ustálí – což může trvat několik iterací a nebo se někdy zcela neustálí.

V každé iteraci se hodnota PageRank stránky vypočítá jako součet hodnot PageRank všech stránek, které na ni odkazují, děleno počtem odkazů na těchto stránkách. Tento proces se opakuje, dokud se hodnoty PageRank stránek nezmění o méně než určitou prahovou hodnotu nebo dokud se nedosáhne maximálního počtu iterací. Všechny stránky začínají s počáteční hodnotou PageRank $R_0(p)$, která je obvykle nastavena na $\frac{1}{n}$, kde n je počet stránek.

$$R_{i(p)} = \sum_{q \in Q} \frac{R_{i-1}(q)}{C(q)}$$

Kde $R_{i(p)}$ je hodnota PageRank stránky p v iteraci i a $i > 0$, Q jsou stránky, které odkazují na stránku p, a $C(q)$ je počet odkazů na stránce q.

Tento postup si můžeme představit jako přelévání důležitosti mezi stránkami. Pokud na nás ukazuje stránka, tak se její důležitost přeleje do všech stránek na které odkazuje ve stejném poměru. A protože důležitost stránky závisí na důležitosti stránek, které na ni odkazují, tak se tento proces opakuje.

Problém, který může nastat je to, že některé stránky zcela ztrácí svojí relevanci. Proto se přidává tzv. damping factor, což je pravděpodobnost, že uživatel přestane sledovat odkazy a začne hledat znovu. To se obvykle nastavuje na hodnotu kolem 0.85. Tuto hodnotu ale dále rozdělujeme mezi všechny stránky rovnoměrně. Tím pádem modifikujeme vzorec na:

$$R_{i(p)} = \frac{1--d}{n} + d \sum_{q \in Q} \frac{R_{i-1}(q)}{C(q)}$$

Kde d je damping factor a n je počet stránek.

Implementace PageRank

```js
const links = {
  "A": ["B", "C"],
  "B": ["C"],
  "C": ["A"],
  "D": ["C"],
};
function computePageRank(links, dampingFactor = 0.85, maxIterations =
100, tolerance = 1.0e-6) {
  const pages = Object.keys(links);
  const numPages = pages.length;
  let pageRank = {};
  for (let page of pages) {
    pageRank[page] = 1 / numPages;
  }

  for (let i = 0; i < maxIterations; i++) {
    let newPageRank = {};
    let delta = 0;
    for (let page of pages) {
```

```
19        let rankSum = 0;
20        for (let otherPage of pages) {
21          if (links[otherPage].includes(page)) {
22            rankSum += pageRank[otherPage] / links[otherPage].length;
23          }
24        }
25        newPageRank[page] = (1 -- dampingFactor) / numPages + dampingFactor
    * rankSum;
26        delta += Math.abs(newPageRank[page] -- pageRank[page]);
27      }
28      pageRank = newPageRank;
29      if (delta < tolerance) {
30        break;
31      }
32    }
33    return pageRank;
34  }
35  console.log(computePageRank(links));
```

Implementace algoritmu PageRank.

Tento způsob není moc optimální pro spoustu stránek, protože je potřeba projít všechny stránky a jejich odkazy v každé iteraci. V praxi se používá pro výpočet iterace matice, což je rychlejší a efektivnější.

Ustálení algoritmu je závislé na tom, jestli existují v grafu cykly. Ty si poté předávají důležitost stále dokola a nikdy se neustálí na určitou hodnotu.

Nejznámější implementace PageRanku je v rámci vyhledávače Google. Ten používá různé modifikace a vylepšení tohoto algoritmu, aby dosáhl co nejlepších výsledků.

6.4.1 Moderní hodnocení

Vědci a i samotní vývojáři vyhledávačů si poslední léta uvědomili, že samotný PageRank nestačí. To vedlo k tomu, že vznikla sada metrik zvaná **Core Web Vitals**. Jedná se o tři klíčové faktory, které Google považuje za nejdůležitější pro uživatelskou zkušenost a přímo ovlivňují pozici ve vyhledávání:

- **LCP** (Largest Contentful Paint): Měří rychlost načtení hlavního obsahu,
- **CLS** (Cumulative Layout Shift): Měří vizuální stabilitu stránky,
- **INP** (Interaction to Next Paint): Měří odezvu stránky na interakce uživatele.

Kromě těchto tří hlavních „Core“ metrik existují i další diagnostické metriky, které nám pomáhají odhalit problémy, například **FCP** (First Contentful Paint) nebo TTFB (Time to First Byte).

Toto jsou přímo měřitelné metriky. Obecně chceme dosáhnout následujících hodnot:

- **LCP**: pod 2.5 sekundy,
- **CLS**: pod 0.1,
- **INP**: pod 200 milisekund,
- **FCP**: pod 1.8 sekundy.

Google tyto metriky měří u reálných uživatelů (Chrome User Experience Report) a používá je pro hodnocení. My si je můžeme změřit pomocí nástrojů jako Lighthouse (v Chrome DevTools) nebo PageSpeed Insights.

Nyní se zaměříme na to, jak tyto metriky optimalizovat.

CLS (Cumulative Layout Shift) měří vizuální stabilitu. Přímo lze termín přeložit jako „kumulativní posun rozvržení stránky". Znamená to, jak moc se obsah na stránce posouvá během načítání. Pokud se obsah posouvá[25] během načítání, způsobuje to špatnou uživatelskou zkušenost – uživatel může omylem kliknout na něco jiného, než chtěl.

Tipy pro snížení CLS:
- Všem obrázkům a videím nastavte atributy `width` a `height` (nebo poměr stran v CSS), aby prohlížeč věděl, kolik místa rezervovat ještě před stažením obrázku,
- U fontů používejte `font-display: swap;` nebo `optional`,
- Neanimujte vlastnosti měnící rozvržení (`width`, `margin`, `top`), ale raději pouze a jenom `transform` a `opacity`,
- Vyhněte se vkládání obsahu (reklam, bannerů) nad již načtený text bez rezervace místa (např. pomocí skeleton loadingu).

LCP (Largest Contentful Paint) měří čas vykreslení největšího viditelného prvku (obrázek, video, blok textu). Toto je moment, kdy má uživatel pocit, že „stránka je načtená". Pro zlepšení LCP je nutné, aby se tento hlavní prvek začal stahovat co nejdříve.

Tipy pro zlepšení LCP:
- **Pozor na lazy-loading:** Nikdy nepoužívejte `loading="lazy"` na obrázek, který je LCP (ten hlavní nahoře). To by prohlížeči řeklo, ať s načítáním počká, což LCP zhorší. Naopak použijte `fetchpriority="high"`,
- Použijte `<link rel="preload">` pro kritické zdroje (hlavní obrázek, font),
- Optimalizujte velikost obrázků (formáty WebP/AVIF, správné rozlišení),
- Zrychlete odpověď serveru (TTFB) pomocí cachování nebo CDN.

INP (Interaction to Next Paint) nahradilo v roce 2024 starší metriku First Input Delay (FID). Zatímco LCP řeší načítání, INP řeší interaktivitu. Měří, jak dlouho trvá, než stránka zareaguje na kliknutí myší nebo stisk klávesy. Pokud kliknete na tlačítko a nic se nestane (protože prohlížeč je „zasekaný" vykonáváním JavaScriptu), INP bude vysoké.

Tipy pro zlepšení INP:
- Rozdělte dlouhé JavaScriptové úlohy na menší části (aby se hlavní vlákno uvolnilo pro vykreslení změny),
- Minimalizujte JavaScript, který se spouští při načítání stránky,
- Pozor na složité event listenery, které blokují vykreslování.

FCP (First Contentful Paint) je diagnostická metrika. Měří čas od začátku načítání do doby, kdy je vykreslen **první** jakýkoliv obsah (text, logo). Pro zlepšení FCP musíme odstranit tzv. **blokující zdroje** (render-blocking resources). Prohlížeč totiž musí stáhnout a zpracovat CSS a JS v `<head>`, než může cokoli vykreslit.

Tipy pro FCP:
- Kritické CSS vložte přímo do hlavičky (inline styles),
- Skripty, které nejsou nutné pro vykreslení, odložte pomocí atributů `defer` nebo `async`.

[25] Vzpomeňte si, jak jsme si povídali na WBA o tom, že nechceme, aby fonty způsobili probliknutí textu.

6.4.2 Co přesně vyhledávače chtějí

Vyhledávače poslední dobou začínají klást důraz i na další věci, které je vhodné mít na paměti. Jedna z nejdůležitějších věcí je responzivita vašich stránek. Dříve Google hodnotil stav zejména podle toho, jak se stránka chová na desktopových zařízeních. Nyní ale přechází na **mobile-first indexing**, což znamená, že Google indexuje a hodnotí stránky primárně podle jejich mobilní verze. To znamená, že vaše stránky musí být plně funkční a dobře použitelné na mobilních zařízeních.

Nejčastější problém bývá s tím, že na handheldech často chybí nějaké části obsahu – to tedy pro dobré SEO nesmíte dělat. Často se například i skrývá navigační menu za „hamburger" ikonu, což je v pořádku, ale pokud je v menu nějaký důležitý odkaz, tak by měl být dostupný i bez rozkliknutí menu. Proto byste měli mít na stránce odkazy na všechny důležité části stránky v samotném obsahu. Co vyhledávač nevidí, to nehodnotí.

Google konkrétně představil i jejích cíle při hodnocení stránek, které nazval **E-E-A-T**:

- **Experience** (Zkušenost): Má stránka obsah vytvořený někým, kdo má s daným tématem zkušenosti?
- **Expertise** (Odbornost): Je obsah vytvořen odborníkem na dané téma?
- **Authoritativeness** (Autorita): Je stránka nebo autor považován za autoritu v daném oboru?
- **Trustworthiness** (Důvěryhodnost): Je stránka důvěryhodná a bezpečná pro uživatele?

Samozřejmě byste si mohli říct – „Tohle je přeci strašně subjektivní. Já dělám stránku kavárně, jak mám dokázat, že jsem odborník na kávu?". Právě proto je důležité budovat si na webu svou značku a autoritu. Tyhle faktory se totiž často hodnotí na základě zpětných odkazů, recenzí, hodnocení a dalších signálů z webu. Primárně jsou určeny zejména ale na obory jako jsou zdravotnictví, finance a další oblasti, kde je důvěryhodnost klíčová.

Důležité pro vás je, že když budete tvořit stránku pro právní poradenství, tak obyčejné taktiky, které jsme si dříve popsali nestačí. Musíte aktivně budovat svoji autoritu a důvěryhodnost, například tím, že budete mít na stránce informace o autorovi, jeho kvalifikaci, odkazy na recenze a podobně. Zejména je ale musíte předávat tzv. stuktorovanými daty, aby je vyhledávače správně pochopily.

6.5 Podvodné techniky

Cílem lidstva je vždy najít nejrychlejší a nejefektivnější způsob, jak dosáhnout cíle. To platí i pro vyhledávače, respektive lidi, kteří chtějí být zadarmo na prvních místech ve vyhledávačích. Proto se objevují různé podvodné techniky, které se snaží oklamat vyhledávače a dostat se na přední místa ve výsledcích vyhledávání.

Farma odkazů je technika, kdy se vytváří velké množství stránek, které obsahují odkazy na cílovou stránku. Tyto stránky jsou často nízké kvality a obsahují málo nebo žádný relevantní obsah. Cílem je zvýšit počet odkazů na cílovou stránku a tím zvýšit její PageRank. Vyhledávače se snaží tyto farmy detekovat a dané odkazy nepočítají do hodnocení.

Skrytý text je technika, kdy se na stránce nachází text, který je viditelný pouze pro vyhledávače, ale ne pro uživatele. To se často dělá pomocí bílého textu na bílém pozadí nebo pomocí CSS, které skrývá text před uživatelem. Vyhledávače umí detekovat nevykreslený text a penalizují stránky, které tuto techniku používají.

Přesměrování je technika, kdy se crawler vyhledávače přesměruje na jinou stránku, než kterou vidí uživatel. Často webové servery detekují hlavičku `User-Agent` a podle toho rozhodují, jaký obsah zobrazit. Vyhledávače někdy zkouší přístup i bez této hlavičky, aby zjistily, jestli je obsah stejný. Pokud není, tak stránka může být penalizována.

6.6 Optimalizace pro vyhledávače

Optimalizace pro vyhledávače, známá jako SEO (Search Engine Optimization), je proces, při kterém se snažíme zlepšit viditelnost a pozici webové stránky ve výsledcích vyhledávačů. Cílem SEO je podstrčit eticky a legálně webovým vyhledávačům co nejlepší informace o obsahu stránky, aby ji mohly správně zařadit do svých indexů a nabídnout ji uživatelům při relevantních dotazech. Klíčem SEO je opravdu pár základních věcí.

Často se stává, a určitě jste to viděli, že na SEO existují celé firmy a agentury. Tyto agentury se často snaží vnutit to, že SEO je něco složitého a že je potřeba dělat mnoho věcí. Problémem SEO je, že aby se projevilo, je potřeba čekat a tvořit kvalitní obsah. Ne každá firma se snaží majitelům stránek prodat to, že kvůli SEO je nutné předělat celý web a výsledky se dostaví ihned. SEO se taktéž nedá opravit tím, že se opraví aktuální obsah. Obsah se totiž stále přidává a ten musí být upraven tak, aby stránka obecně neztrácela na relevantnosti mezi obsahy.

Důležité je si uvědomit, že SEO není o podvádění vyhledávačů, ale o tom, jak jim správně předat informace o obsahu stránky.

Mezi základní techniky SEO patří:

- Použití relevantních klíčových slov v názvech, nadpisech a textu stránky,
- Vytváření kvalitního a relevantního obsahu, který odpovídá na otázky uživatelů,
- Optimalizace struktury URL, aby byly srozumitelné a obsahovaly klíčová slova,
- Použití meta tagů, jako je title a description, které popisují obsah stránky,
- Vytváření interních a externích odkazů, které zvyšují autoritu stránky,
- Zajištění, že stránka je rychlá a responzivní, tedy dobře funguje na mobilních zařízeních,
- Použití strukturovaných dat, které pomáhají vyhledávačům lépe pochopit obsah stránky.

Pro moderní stránky je důležité, aby se stránka dobře zobrazovala zejména ve vyhledávači Google. Google pro majitele webu vyvinul nástroj Google Search Console, který pomáhá sledovat výkon stránky ve vyhledávači, zjišťovat problémy a optimalizovat obsah. Po zaregistrování webu se majitelům ukáže například:

- Jaké dotazy uživatelé zadávají, aby našli jejich stránku,
- Jaké stránky jsou nejvíce navštěvované,
- Jaké chyby vyhledávač při procházení stránky našel,
- Jaké odkazy jsou uloženy v indexu vyhledávače,
- Na jaké průměrné pozici se jejich stránka zobrazuje ve výsledcích vyhledávání.

V tomto nástroji můžeme taktéž požádat o znovuprocházení stránky, pokud jsme na ní provedli nějaké změny, nebo jsme přidali nový obsah.

Klíčové pro vyhledávače je, že pokud příjdou na hlavní stránku, a na ni neexistuje odkaz na jinou stránku na našem webu, tak na ni logicky nebudou pokračovat. V celém průchodu webu a všech stránek pouze klikáním na odkazy by mělo být možné dojít na všechny stránky. Odkazy by taktéž měli být viditelné ihned při navštívení stránky a nebýt schované například ve vyskakovacích menu.

Pokud je stránka moc hluboko v hierarchii odkazů, vyhledávač se může rozhodnout, že již hlouběji nepůjde. To dělá kvůli problémům nekonečného procházení. Jako ukázku si lze

představit stránku, která uchovává kalendář. Aktuální měsíc kalendáře je uložen v URL adrese, např. v parametru `month`. Na každé stránce kalendáře se nacházejí dvě tlačítka – předchozí měsíc a další měsíc. Pokud by vyhledávač začal procházet kalendář, dostane se do nekonečného procházení měsíců (pokud samozřejmě měsíc neomezíme).

6.6.1 Sitemap

Pokud nějaké stránky nemáme šanci dostat do struktury odkazů, můžeme webovým vyhledávačům pomoci pomocí souboru sitemap. Sitemap je soubor ve formátu XML, který obsahuje seznam všech stránek na našem webu. U každé stránky určujeme její URL, jak často se má navštevovat a jak je důležitá vůči ostatním stránkám.

Sitemap se umisťuje do kořenového adresáře webu a její umístění se zadává do souboru `robots.txt`, který si ještě ukážeme, viz Kapitola 6.6.2.

xml

Ukázka souboru sitemap

```xml
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://www.example.com/kalendář?month=1</loc>
    <lastmod>2023-01-01</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.8</priority>
  </url>
  <url>
    <loc>https://www.example.com/o-nás</loc>
    <lastmod>20243-02-01</lastmod>
    <changefreq>daily</changefreq>
    <priority>0.4</priority>
  </url>
</urlset>
```

Jednoduchý soubor sitemap, který obsahuje dvě stránky s jejich URL, datem poslední změny, frekvencí změn a prioritou.

Soubor sitemap se často generuje automaticky z databáze stránek. Častým problémem je, že stránek máme mnoho, a do jednoho souboru se jich nevejde tolik. Resp. pokud vyhledávač narazí na sitemap, který se stahuje příliš dlouho (nebo je moc velký), tak ho nemusí zpracovat. V takovém případě se používá sitemap index, což je soubor, který obsahuje odkazy na další soubory sitemap.

xml

Ukázka souboru sitemap index

```xml
<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex xmlns="http://www.sitemaps.org/schemas/sitemap/0.
9">
  <sitemap>
    <loc>https://www.example.com/sitemap1.xml</loc>
    <lastmod>2023-01-01</lastmod>
  </sitemap>
  <sitemap>
```

```
 9     <loc>https://www.example.com/sitemap2.xml</loc>
10     <lastmod>2023-01-01</lastmod>
11   </sitemap>
12 </sitemapindex>
```

Jednoduchý soubor sitemap index, který obsahuje odkazy na dva další soubory sitemap.

6.6.2 Ovládání přístupu

Na webovou stránku chodí různí roboti (nejen vyhledávače), kteří si stránku procházejí a stahují. Etické je, že pokud nechceme aby robot s naší stránkou pracoval, tak mu to dáme najevo v speciálním souboru `robots.txt` v kořenovém adresáři webu. Tento soubor je ve formátu prostého textu a obsahuje pravidla pro různé roboty. Můžeme jim určit, jaké stránky mohou a nemohou navštěvovat. Můžeme omezit i pro jaké uživatelské agenty (User-Agent) jsou pravidla platná.

Dále uvádíme umístění souboru sitemap, aby ho roboti mohli najít a použít.

plaintext

Ukázka souboru robots.txt

```
1 User-Agent: *
2 Disallow: /admin/
3 Disallow: /private/
4 Allow: /public/
5
6 User-Agent: Cajthaml
7 Allow: /
8
9 Sitemap: https://www.example.com/sitemap.xml
```

Jednoduchý soubor robots.txt, který zakazuje přístup do adresářů /admin/ a /private/, povoluje přístup do /public/ a určuje umístění souboru sitemap.

Mezi nejznámější `User-Agent` robotů patří:

- `Googlebot`,
- `Bingbot`,
- `Slurp` (Yahoo),
- `DuckDuckBot`,
- `Baiduspider`,
- `YandexBot`.

6.6.3 Obsah

Obsah stránek někdy nestačí pro to, abychom předali vyhledávačům správné informace o tom, co se na stránce nachází. Proto se používají tzv. strukturovaná data, což je speciální formát, který se vkládá do HTML stránky a obsahuje informace o obsahu stránky.

Mezi nejznámější formáty strukturovaných dat patří Schema.org, JSON-LD a Microdata. Tyto formáty umožňují popsat různé typy obsahu, jako jsou články, produkty, události, recenze a další. Vyhledávače poté tyto informace používají pro lepší pochopení obsahu stránky a pro zobrazení ve výsledcích vyhledávání.

html

Ukázka strukturovaných dat v JSON-LD

```html
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Article",
  "headline": "Nadpis článku",
  "author": {
    "@type": "Person",
    "name": "Jméno autora"
  },
  "datePublished": "2023-01-01",
  "image": "https://www.example.com/obrazek.jpg",
  "publisher": {
    "@type": "Organization",
    "name": "Název vydavatele",
    "logo": {
      "@type": "ImageObject",
      "url": "https://www.example.com/logo.jpg"
    }
  },
  "description": "Krátký popis článku."
}
</script>
```

Ukázka strukturovaných dat v JSON-LD formátu, která popisuje článek s jeho nadpisem, autorem, datem publikace, obrázkem, vydavatelem a popisem.

Dále jsou důležité metatagy pro tzv. Open Graph, které používají sociální sítě, jako je Facebook, Twitter a další.

html

Ukázka Open Graph metatagů

```html
<meta property="og:title" content="Nadpis stránky">
<meta property="og:description" content="Krátký popis stránky.">
<meta property="og:image" content="https://www.example.com/obrazek.jpg">
<meta property="og:url" content="https://www.example.com/stranka">
<meta property="og:type" content="website">
```

Ukázka základních Open Graph metatagů pro lepší sdílení na sociálních sítích.

Pro samotný obsah se dále doporučje na jedné stránce (v článku, ...) používat různá slova v různých tvarech a synonymech. Tím pádem se zvyšuje šance, že uživatel najde stránku, i když použije jiné slovo, než které je na stránce. Tedy například místo slova „auto“ použít i „automobil“, „vozidlo“, „dopravní prostředek“ a podobně.

Důležité je určit i základní meta tagy, které popisují stránku.

html

Ukázka základních meta tagů

```html
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
4   <meta name="description" content="Krátký popis stránky pro
  vyhledávače.">
5   <meta name="keywords" content="klíčové, slova, oddělená, čárkami">
6   <meta name="author" content="Jméno autora">
7   <title>Nadpis stránky</title>
8 </head>
```

Ukázka základních meta tagů, které by měly být na každé stránce pro lepší SEO.

6.6.4 Kanonické URL

Kanonické URL je technika, která pomáhá vyhledávačům pochopit, která verze stránky je ta „správná“ nebo „hlavní“, pokud existuje více verzí stejného obsahu. Pokud totiž vyhledávač vidí, že existuje více URL, které vedou na stejný obsah, může to způsobit problémy s duplicitním obsahem a snížit hodnocení stránky.

Můžeme si to jednoduše představit třeba v e-shopu, který obsahuje produkty. Produkt může být dostupný v různých kategoriích, například „elektronika“ a „domácí spotřebiče“. Tedy stejný produkt může být dostupný na více URL, například:

- `https://www.example.com/elektronika/produkt-123`
- `https://www.example.com/domaci-spotrebice/produkt-123`

Pokud toto uvidí vyhledávač, může si myslet, že se jedná o duplicitní obsah a snížit hodnocení stránky, protože neví, která verze je ta hlavní – myslí si, že se ho snažíte oklamat. Resp. on se bude snažit zobrazit obě stránky v jeden moment, takže jejich „síla“ se rozloží na dvě URL a tím pádem se sníží jejich hodnocení. Abychom tomu předešli, můžeme použít kanonické URL.

Kanonické URL je technika, kde přímo v HTML určíme, že „tato stránka není hlavní, ale hlavní je tato jiná stránka“. Děláme to pomocí tagu `<link rel="canonical" ...>` v hlavičce HTML stránky.

html

Ukázka kanonické URL

```
1 <head>
2   <link rel="canonical" href="https://www.example.com/elektronika/
  produkt-123">
3 </head>
```

Ukázka kanonické URL, která určuje hlavní verzi stránky.

6.7 Single Page Application a SEO

Single Page Application (SPA) je typ webové aplikace, která načítá všechny potřebné zdroje najednou a poté dynamicky mění obsah stránky bez nutnosti znovu načítat celou stránku. Výsledkem zkompilované SPA je jeden HTML soubor a spoustu pomocných JavaScript a CSS souborů.

Pro načtení této stránky je ale nutné spustit JavaScript. Zdrojový kód stránky je totiž až v JavaScriptu, který se spustí až po načtení stránky. To může být problém pro vyhledávače, protože některé z nich nemusí umět spustit JavaScript a tím pádem nevidí obsah stránky. Moderní vyhledávače, jako je Google, umí JavaScript spustit a obsah stránky vidí. Často ale

nechceme nutit vyhledávače spouštět JavaScript, protože to může být pomalé a náročné na zdroje, a obecně ne všichni to dělají.

Naštěstí máme dvě různé možnosti jak tento problém vyřešit.

6.7.1 Server-side rendering

První možností je použít server-side rendering (SSR). To znamená, že místo toho, aby se SPA aplikace vykreslovala[26] na straně klienta (v prohlížeči), tak se HTML vykreslí podle logiky aplikace na straně serveru a pošle se již hotové HTML uživateli. Tím pádem vyhledávače uvidí již hotové HTML a nemusí spouštět JavaScript. To pomáhá i uživatelům se staršími prohlížeči. Navíce na serveru můžeme volat jednodušeji různé služby, například přistupovat k databázi, aniž bychom museli řešit CORS a podobně.

Na fungování SSR je nutné myslet již při vývoji aplikace. Potřebujeme k tomu speciální frameworky, které toto podporují, například Next.js pro React, Nuxt.js pro Vue a podobně.

6.7.1.1 Vsuvka historického kontextu

Webový vývoj je v cyklu již od svých počátků.

Nejdříve se vše dělalo na straně serveru, kde se generovalo HTML a posílalo uživateli. Webový prohlížeč pouze zobrazoval HTML a CSS. To je velmi nákladné pro majitele stránek, které platí za přenesená data a výpočetní výkon serveru.

To nestačilo, protože stránky chceme mít interaktivní a proto se začal používat JavaScript. Ten se ale spouštěl až na straně klienta, což znamenalo, že stránka musela být nejdříve načtena a poté se spustil JavaScript, který stránku upravil.

To velmi později vedlo k tomu, že se začaly tvořit frameworky pro webové stránky, které umožňují vytvářet složité aplikace na straně klienta. Servery často tedy řeší jen to nejdůležitější – bezpečnost a poskytují data aplikacím. To pomáhá majitelům webových stránkám, protože nemusí platit tolik za výpočetní výkon serveru a přenesená data. Klient ale musí mít výkonný počítač a rychlé připojení k internetu, aby stránka fungovala dobře.

To ale zase přineslo problémy a to s tím, že vyhledávače nemusí umět spustit JavaScript a tím pádem nevidí obsah stránky. Tedy stránky vytvořené pomocí frameworků nebyly vidět ve vyhledávačích.

To vedlo k tomu, že se začaly používat techniky jako server-side rendering a pre-rendering, které pomáhají vyhledávačům vidět obsah stránek tím, že jim poskytneme již hotové HTML. HTML se generuje na serveru, takže vyhledávače nemusí spouštět JavaScript.

Celý tento cyklus se stále opakuje a vznikají nové technologie, které již dříve existovali a přestali se používat, protože přinášeli nějaké nevýhody.

6.7.2 Pre-rendering

Pre-rendering je technika, kdy se HTML stránky vygenerují předem. To znamená, že když vývojář vytvoří novou stránku nebo aktualizuje stávající, tak se tato stránka vygeneruje na serveru a uloží se jako statický HTML soubor. Tím pádem, když uživatel nebo vyhledávač požádá o stránku, server jednoduše pošle již hotový HTML soubor.

[26] Generovala HTML. CSS zde vůbec neřešíme.

Tedy existuje služba, která sleduje, zda na stránku přichází vyhledávač. Pokud ano, tak se podíváme do cache a vrátíme již vygenerovaný HTML soubor. Pokud není vygenerovaný, tak jej pošleme do virutálního prohlížeče, který stránku načte a vygeneruje HTML, které se uloží do cache a pošle uživateli. Tím pádem vyhledávače vidí již hotové HTML a nemusí spouštět JavaScript.

Většinou se ale tento přístup používá pouze pro vyhledávače a běžní uživatelé dostávají klasickou SPA aplikaci.

Mezi nejznámější služby pro pre-rendering patří Prerender.io, Rendertron a podobně.

6.8 Budoucnost vyhledávání

Již při psaní tohoto textu jsem si i já osobně musel nastudovat spoustu dalších informací, protože se způsob vyhledávaní masivně mění (zejména díky úmělé inteligenci). Moje zkušenosti z toho jsem ale přímo z praxe a z dřívějších let. Za poslední 3 roky se situace opavdu velmi změnila.

Pokud budeme abstrahovat tento trend, tak můžeme říct, že i v budoucnu se bude nadále velmi měnit způsob, jakým lidé vyhledávají informace na internetu. Proto je důležité, abyste se neustále učili ale hlavně sledovali trendy a kriticky přemýšleli o tom, proč se něco děje.

Jako poslední věc, kterou v těchto skriptech zmíním, je to, jak AI mění vyhledávání.

Tradiční model „zadej dotaz -> dostaneš 10 modrých odkazů“ ustupuje. Nastupuje totiž tzv. SGE (Search Generative Experience) a AI chatboti (ChatGPT, Bing Copilot).

V tomto modelu uživatel položí otázku a vyhledávač mu vygeneruje přímou odpověď. Uživatelé tak často vůbec nekliknou na žádný odkaz (tzv. se jedná o Zero-click searches).

Pro nás jako tvůrce webů to znamená nové výzvy:
1. Musíme tvořit obsah tak strukturovaný, aby ho AI pochopila a použila jako zdroj pro svou odpověď,
2. Strukturovaná data (např. Schema.org) jsou ještě důležitější, protože AI z nich čerpá fakta,
3. Optimalizujeme spíše pro „dlouhé konverzační dotazy“ (Long-tail keywords) než pro heslovitá klíčová slova.

Aplikace tvořené webovými technologiemi

7.1 Popularita webu

Webové aplikace jsou dnes velmi populární. Poskytují totiž jednoduchý způsob, jak uživatelům nabídnout funkce a služby bez nutnosti instalace[27] speciálního softwaru. Prohlížeče jsou totiž v každém zařízení, ať už jde o počítač, tablet nebo mobilní telefon. Uživateli stačí pouze zadat adresu webové aplikace (nebo použít webový vyhledávač[28]) a může ji okamžitě začít používat.

Navíc je možné webové aplikace aktualizovat na straně serveru, takže uživatelé mají vždy přístup k nejnovější verzi. Weby jsou tak velmi dostupné, flexibilní a snadno použitelné.

Přesto že webové aplikace mají mnoho výhod, existují i určité nevýhody. Mezi ně patří například omezený přístup k hardwarovým funkcím zařízení, závislost na internetovém připojení a potenciální problémy s výkonem ve srovnání s nativními aplikacemi[29].

A navíce nativní aplikace jsou ještě více příjemné než přístupné webové aplikace. Uživatel nemusí totiž nic hledat, a ví, že danou aplikaci najde na svém zařízení – ať už na ploše, v nabídce aplikací nebo v docku. To může být použité jako taktika, aby uživatelé používali nativní aplikaci místo webové aplikace, kde se může stát, že najde konkurenci.

Proto se mnoho vývojářů snaží kombinovat výhody webových technologií s výhodami nativních aplikací. Tyto hybridní aplikace se nazývají progresivní webové aplikace o kterých bude řeč dále.

I přes tuto snahu však jsou potřeba nativní aplikace. Proto se vývojáři rozhodli použít webové technologie k tvorbě nativních aplikací. O nativních aplikacích bude řeč také dále.

7.2 Webové API

Pro potřeby komunikace mezi webovou aplikací a serverem, jak již víme, se používají API. API umožňují webovým aplikacím komunikovat se serverem a využívat jeho funkce a data.

Ve webovém prohlížeči máme k dispozici různá API, které slouží k tomu obdobnému. Komunikaci s zařízením. Stejně jako na serveru zařizuje API jednu klíčovou funkci, a to je bezpečnost. Aby se nestalo to, že webová stránka dostane přístup k něčemu, k čemu by neměla. Pro Webové API bezpečnost zajišťuje prohlížeč. K API přistupujeme pomocí globálních objektů, které jsou dostupné v JavaScriptu.

Webová API nám pomáhají používat funkcionality zařízrení a prohlížeče. Bez většího přemýšlení jste již mnoho z nich použili – například se jedná o paměť (localStorage), posílání požadavků na server (Fetch API) a mnoho dalších.

Ne každý prohlížeč však podporuje všechna API. Proto je potřeba vždy zkontrolovat, zda je dané API podporováno v prohlížeči uživatele. Podobně mají API různá omezení na použití. Asi si umíme představit, že kdybychom povolili každému webu přístup k fotoaparátu, nebylo by to moc bezpečné.

Proto každé API má své konkrétní povolení na každé stránce. V základu může být přístup v následuujících stavech:

- Povolený (Granted) – web má přístup k API,

[27] Uvádí se, že 40 % uživatelů není ani schopno potvrdit aktualizaci již nainstalovaného softwaru.

[28] Už je to tu zase.

[29] Nativní aplikace jsou aplikace, které jsou nainstalované na zařízení.

- Zamítnutý (Denied) – web nemá přístup k API,
- Neurčený (Prompt) – web musí požádat uživatele o povolení přístupu k API.

Prohlížeče mají základní nastavení pro každé API. Každá stránka z těchto základních nastavení (které může uživatel změnit) může vybočit dle nastavení uživatele. API hojně používají sliby pro asynchronní práci s povoleními.

Interně API mají další interní stavy, které např. omezují dobu platnosti povolení. Některá API fungují pouze na zabezpečených stránkách (https) a proti zneužití jsou implementovány různé bezpečnostní mechanismy – například zamezení použití API, pokud uživatel neinteragoval se stránkou (např. kliknutím nebo stisknutím klávesy)[30].

Některá API jsou automaticky povolená, protože je chcete používat každý web – například API pro práci s pamětí (localStorage), samotný JavaScript (ECMAScript), nebo Fetch API pro komunikaci se serverem.

Mezi používaná API patří například:

- **Geolocation API** – umožňuje získat polohu uživatele,
- **Camera API** – umožňuje přístup k fotoaparátu zařízení,
- **Notifications API** – umožňuje zobrazit notifikace uživateli,
- **Web Storage API** – umožňuje ukládat data v prohlížeči (localStorage, sessionStorage),
- **Clipboard API** – umožňuje přístup ke schránce zařízení,
- **WebRTC API** – umožňuje komunikaci v reálném čase (video, audio),
- **Service Workers API** – umožňuje práci na pozadí a offline režim, o kterém bude řeč dále,
- **Push API** – umožňuje přijímat push notifikace,
- **File API** – umožňuje práci se soubory na zařízení,
- A mnoho dalších.

Klíčové je vědět, že skoro každá funkcionalita na webu má své API. Každé API používá kompletně jiný způsob použití a má různá omezení.

Získání povolení k použití API vypadá například takto:

javascript

Povolení k použití API

```
1 navigator.permissions.query({ name: 'geolocation' })
2   .then((permissionStatus) => {
3     console.log('Permission status:', permissionStatus.state);
4     permissionStatus.addEventListener('change', () => {
5       console.log('Permission status changed to:',
  permissionStatus.state);
6     });
7   });
```

Získá povolení k použití Geolocation API.

7.2.1 Notifikace

Nejzajímavější API, které ještě neznám a je často používané, je Notifications API. Notifikace dovolují webových aplikacím zobrazovat uživateli zprávy, které přijdou uživateli mimo aktuálně otevřenou stránku – například pomocí nativních notifikací v operačním systému.

[30] Dříve bylo možné při načtení stránky získat informace a uživatele hned přesměrovat jinam, takže si ničeho ani nevšiml.

Notifikace jsou často používané pro různé zprávy, jako jsou nové zprávy, upozornění na události nebo jiné důležité informace.

Pro použití notifikací je potřeba získat povolení od uživatele. Prozatím si ukážeme jak se notifikace tvoří a zobrazují pokud je stránka otevřená. Později se dostaneme k tomu, jak udělat, že notifikace přijde i když stránka není otevřená, viz Kapitola 7.3.4.1.

javascript

Zobrazení notifikace

```
if('Notification' in window) {
  Notification.requestPermission().then((permission) => {
    if(permission === 'granted') {
      const notification = new Notification('Ahoj!', {
        body: 'Toto je notifikace z webové aplikace.',
        icon: '/path/to/icon.png'
      });
      notification.addEventListener('click', () => {
        window.focus();
        notification.close();
      });
    }
  });
} else {
  console.log('Tento prohlížeč nepodporuje notifikace.');
}
```

Zobrazí notifikaci pokud uživatel povolil notifikace.

Mezi další vlastnosti, které notifikacím můžeme nastavit, patří například:

- **body** – text notifikace,
- **icon** – ikona notifikace,
- **image** – obrázek notifikace,
- **badge** – malá ikona notifikace,
- **vibrate** – vibrace zařízení při zobrazení notifikace,
- **actions** – akce, které může uživatel provést přímo z notifikace,
- **tag** – identifikátor notifikace pro seskupení,
- **data** – vlastní data, která můžeme připojit k notifikaci,
- A mnoho dalších.

7.3 Progresivní webové aplikace

Progresivní webové aplikace (PWA) jsou webové aplikace, které využívají moderní webové technologie a API k tomu, aby poskytly uživatelům zážitek podobný nativním aplikacím.

Dříve bylo běžné, že webové aplikace měly omezené možnosti a neměli skoro vůbec žádný přístup k funkcím zařízení. Proto přišli vývojáři s konceptem progresivních webových aplikací, které tento problém řeší. PWA jsou stránky, které mohou být nainstalované na zařízení uživatele a mohou fungovat i offline. Uživatelovi se často přidá odkaz na „aplikaci“ na plochu nebo do nabídky aplikací, takže k ní má rychlý přístup. A to i na desktopu. PWA využívají různé webové API, jako jsou Service Workers pro práci na pozadí a offline režim, Push API pro push notifikace a další.

PWA je možné aktualizovat na pozadí, takže mohou získat nejnovější funkce a opravy chyb bez nutnosti zásahu uživatele.

> ⚠ **Podpora prohlížečů**
>
> PWA jsou zejména určené pro webové prohlížeče založené na Chromiu (Google Chrome, Microsoft Edge, Opera, atd.). Některé funkce PWA nemusí být dostupné v jiných prohlížečích (např. Safari, Firefox). Firefox například nepodporuje instalaci PWA.

7.3.1 PWA v prohlížeči

PWA aplikace je zcela závislá na podpoře prohlížeče, pomocí kterého je spuštěná.

Interně je PWA pouze unikátní odkaz, který si prohlížeč pamatuje. Při instalaci (a při aktualizaci) si v cache[31] uloží všechny potřebné soubory, aby mohla fungovat i offline.

Při zapnutí aplikace se zapne prohlížeč, který načte uložené soubory z cache a spustí je. Mimo to samotná aplikace může mít nastavené různé vlastnosti, jako je ikona, název, barva záhlaví a další. PWA může být spuštěná v samostatném okně bez viditelných prvků prohlížeče, jako je adresní řádek a tlačítka pro navigaci.

7.3.2 Tvorba PWA

Pro vytvoření PWA je potřeba splnit několik požadavků:

- **HTTPS** – PWA musí být servírovaná přes zabezpečené připojení (HTTPS), nebo lokálně (localhost),
- **Manifest** – PWA musí mít manifest soubor (manifest.json), který obsahuje metadata o aplikaci, jako je název, ikona, barva a další,
- **Service Worker** – PWA musí používat Service Worker pro práci na pozadí a offline režim.

Po splnění těchto požadavků může být webová aplikace nainstalována jako PWA. Prohlížeč často nabídne speciální ikonu, př. pop-up okno, které uživateli umožní aplikaci nainstalovat.

Pokud uživatel alespoň jednou interagoval s webovou aplikací, může mu webová stránka samotná nabídnout instalaci PWA. Ukázka požadavku o registraci PWA vypadá například takto:

javascript

Požadavek o instalaci PWA

```javascript
let deferredPrompt;
window.addEventListener('beforeinstallprompt', (e) => {
  e.preventDefault();
  deferredPrompt = e;
  showInstallButton();
});
// ...
button.addEventListener('click', (e) => {
  hideInstallButton();
  deferredPrompt.prompt();
```

[31] Je to stejná cache, jako ta, kterou používá například prohlížeč na ukládání obrázků a jiných statických souborů.

```
11   deferredPrompt.userChoice.then((choiceResult) => {
12     if (choiceResult.outcome === 'accepted') {
13       console.log('Uživatel přijal instalaci PWA');
14     } else {
15       console.log('Uživatel odmítl instalaci PWA');
16     }
17     deferredPrompt = null;
18   });
19 });
```

Zachytí událost před instalací a umožní vlastní zobrazení dialogu.

7.3.3 Manifest

Manifest je JSON soubor, který obsahuje metadata o PWA. Manifest definuje, jak se má aplikace zobrazit na zařízení uživatele, když je nainstalovaná. Mimo to obsahuje i řadu dalších informací, které se používají i pro jiné účely.

Manifest musí být pojmenovaný jako `manifest.json` a musí být dostupný na kořenové adrese webu (např. `https://example.com/manifest.json`). Nebo může být umístěný jinde, ale musí být odkazovaný v HTML dokumentu pomocí `<link>` tagu.

html

Odkaz na manifest

```
1 <link rel="manifest" href="/manifest32.json">
```

Odkazuje na manifest soubor.

Manifest může obsahovat následující klíčová pole:

- **name** – Název aplikace, který se zobrazí uživateli,
- **short_name** – Krátký název aplikace, který se zobrazí, když není dostatek místa,
- **icons** – Pole ikon aplikace v různých velikostech a formátech,
- **start_url** – URL, na kterou se aplikace načte při spuštění,
- **display** – Určuje, jak se má aplikace zobrazit (např. `standalone`, `fullscreen`, `minimal-ui`, `browser`),
- **background_color** – Barva pozadí aplikace při spuštění,
- **theme_color** – Barva tématu aplikace, která se použije v prohlížeči,
- **orientation** – Určuje orientaci aplikace (např. `portrait`, `landscape`),
- **scope** – Určuje rozsah URL, které aplikace může používat,
- A mnoho dalších.

Pro to, aby PWA mohlo být nainstalováno, musí obsahovat alespoň pole `name`, `icons` a `start_url`.

Příklad jednoduchého manifestu může vypadat takto:

json

Příklad manifestu

```
1 {
2   "name": "Moje PWA Aplikace",
3   "short_name": "MojePWA",
4   "icons": [
5     {
```

```
      "src": "/icons/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ],
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#0000ff",
  "orientation": "portrait"
}
```

Jednoduchý příklad manifestu pro PWA.

Stav instalace PWA můžeme zkontrolovat pomocí DevTools v prohlížeči. V záložce Application najdeme sekci Manifest, kde můžeme vidět informace o manifestu a stav instalace.

7.3.4 Service Worker

S termínem Worker jsme se již setkali v Kapitola 4.5.2. Service Worker je speciální typ webového workeru, který funguje na pozadí, stejně jako běžný worker. Navíce ale běží i když uživatel zavře stránku (a často i když zavře celý prohlížeč). To umožňuje webovým aplikacím provádět různé úkoly na pozadí, jako je synchronizace dat, přijímání notifikací a další.

Service Worker mimo to umí definovat i řadu dalších chování:

- **Offline režim** – Service Worker může zachytit požadavky na síť a vrátit uložené odpovědi z cache, pokud není dostupné internetové připojení,
- **Cache** – Service Worker může ukládat soubory do cache pro rychlejší načítání a snížení zátěže serveru.

Service Worker je nutné zaregistrovat v JavaScriptu na stránce. Jeho soubor se může jmenovat jakkoliv. Uvnitř JS poté voláme metodu `navigator.serviceWorker.register()`, která zaregistruje Service Worker.

javascript

Registrace Service Worker

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/service-worker.js')
      .then((registration) => {
        console.log('Service Worker registrován s dosahem:',
  registration.scope);
      })
      .catch((error) => {
        console.log('Registrace Service Worker selhala:', error);
      });
  });
}
```

Zaregistruje Service Worker při načtení stránky.

Pokud je Service Worker úspěšně zaregistrován, začne běžet na pozadí a může zachytávat požadavky na síť a provádět další úkoly. Pokud je již zaregistrován, další pokusy o registraci jsou ignorovány.

Service Worker má přístup k různým událostem, jako jsou `install`, `activate`, `fetch` a další. Jedná se o omezené prostředí (stejně jako běžný worker), takže nemá přístup k DOM a některým dalším funkcím.

javascript

Service Worker události

```
self.addEventListener('install', (event) => {
  console.log('Service Worker nainstalován');
  event.waitUntil(
    caches.open('my-cache').then((cache) => {
      return cache.addAll([
        '/', // Vloží do cache všechny tyto soubory
        '/index.html',
        '/styles.css',
        '/script.js',
        '/image.png'
      ]);
    })
  );
});
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      if (response) {
        return response; // Vrátí odpověď z cache
      }
      return fetch(event.request); // Jinak provede síťový požadavek
    })
  );
});
```

Ukázka použití událostí install a fetch.

Pro instalaci PWA je potřeba mít zaregistrovaný Service Worker, který může být klidně prázdný. Jak vidíme, tak Service worker nepoužíváme jen pro PWA, ale může být použit i na běžných stránkách.

Seznam nainstalovaných Service Workerů můžeme zkontrolovat v DevTools v prohlížeči. V záložce Application najdeme sekci Service Workers, kde můžeme vidět informace o zaregistrovaných Service Workerech a jejich stavu. Můžeme jim tam i ručně poslat data, nebo je například smazat.

7.3.4.1 Push API

Push API umožňuje webovým aplikacím přijímat push notifikace, i když uživatel nemá otevřenou stránku. Push notifikace nejsou nutně notifikace zobrazené pomocí Notifications API (ale pro to se zejména používají). Push notifikace jsou zasílány ze serveru na Service Worker, který s nimi potom může naložit dle svého. Často se třeba použijí pro zobrazení notifikace uživateli nebo pro synchronizaci dat na pozadí – například ještě než se stránka načte, uživatel

bude mít nejnovější data. Pro použití Push API je potřeba mít zaregistrovaný Service Worker a získat povolení od uživatele pro přijímání push notifikací.

Od prohlížeče pak dostaneme na webové stránce „registraci“, která obsahuje informace potřebné pro zasílání push notifikací. Jedná se o:

- **endpoint** – URL, na kterou se posílají push notifikace,
- **klíče** – různé klíče, které specifikují uživatele a zabezpečení.

javascript

Registrace pro push notifikace

```javascript
navigator.serviceWorker.ready.then((registration) => {
  return registration.pushManager.subscribe({
    userVisibleOnly: true,
    applicationServerKey: urlBase64ToUint8Array('YOUR_PUBLIC_VAPID_KEY')
  });
}).then((subscription) => {
  console.log('Push registrace:', subscription);
}).catch((error) => {
  console.log('Registrace pro push selhala:', error);
});
```

Zaregistruje se pro push notifikace a získá registraci.

V ukázceč můžeme vidět, že potřebujeme získat tzv. VAPID klíče, které slouží k zabezpečení push notifikací. Tyto klíče si můžeme vygenerovat pomocí různých nástrojů a knihoven. Klíče reprezentují veřejný a soukromý klíč, které se používají pro šifrování a ověřování push notifikací. Vždy se váží ke konkrétní e-mailové adrese. Nejjednodušší je použít webovou službu, která klíče vygeneruje za nás – například https://vapidkeys.com/. Z WBA si vzpomenete na asymetrické šifrování, kde máme veřejný a soukromý klíč. Na klientovi vždy ukládáme a používáme pouze veřejný klíč!

Na serveru poté můžeme tuto registraci použít k zasílání push notifikací pomocí různých knihoven a služeb. Nejznámější je knihovna `web-push` pro Node.js. Nejčastěji ukládáme registraci někde do databáze pro konkrétního uživatele.

javascript

Odeslání push notifikace

```javascript
import webPush from 'web-push';

const publicVapidKey = 'YOUR_PUBLIC_VAPID_KEY';
const privateVapidKey = 'YOUR_PRIVATE_VAPID_KEY';

webPush.setVapidDetails('mailto:YOUR_EMAIL@example.com', publicVapidKey, privateVapidKey);

const pushSubscription = { /* získaná registrace z klienta */ };
const payload = JSON.stringify({ title: 'Nová zpráva!', body: 'Máte novou zprávu.' });
const options = {
  TTL: 60
};
webPush.sendNotification(pushSubscription, payload, options)
  .then(() => {
```

```
15      console.log('Push notifikace odeslána');
16    })
17    .catch((error) => {
18      console.log('Chyba při odesílání push notifikace:', error);
19    });
```

Odesílá push notifikaci pomocí web-push knihovny.

Uvnitř service workeru potom můžeme tuto notifikaci zachytit a zobrazit zprávu například pomocí notifikací.

javascript

Zachycení push notifikace

```
1  self.addEventListener('push', (event) => {
2    const data = event.data ? event.data.json() : {};
3    const title = data.title || 'Push notifikace';
4    const options = {
5      body: data.body || 'Máte novou zprávu.',
6      icon: '/icons/icon-192x192.png'
7    };
8    event.waitUntil(
9      self.registration.showNotification(title, options)
10   );
11 });
```

Zachytí push notifikaci a zobrazí ji uživateli uvnitř service workeru.

Uvnitř service workeru notifikace zobrazujeme pomocí `self.registration.showNotification()`, která je dostupná pouze uvnitř service workeru. Je to podobné jako Notifications API, ale je to jiné API, které je dostupné pouze uvnitř service workeru.

7.3.5 Fungování Push API

Interně Push API funguje na základě komunikace serveru směrem ke klientovi. Pokud pošleme data na endpoint v registraci, vzdálený server prohlížeče zprávu uloží a pokusí se ji doručit klientovi.

Prohlížeč po zapnutí zařízení (nebo spuštění prohlížeče) naváže spojení se serverem a čeká na nové zprávy. Jedná se o dlouhodobé spojení, které je udržované na pozadí. Můžeme si to představit jako kanál SSE nebo WebSocket, ale je to implementované na úrovni prohlížeče, viz Kapitola 5.2.3.

7.3.6 Použití PWA

spise omezene pouziti, nepopularni ale zpristupnil API, service workery a dalsi veci, ktere se daji vyuzit i mimo PWA!

PWA se dají používat zejména na chromium prohlížečích. Mozilla Firefox nemá ráda PWA a nechce je příliš podporovat. Proto se spousta stránek rozhodne PWA nedělat. Mimo to, použití PWA je hodně specifické. Ne každá stránka může být použita bez sítě. Obecně vývojáři nevěří v ukládání dat (které by například vznikla během offline režimu) na straně klienta. Navíce jsme v době, kdy máme všude internetové připojení.

PWA se jednoduše neuchytili a mnoho lidí nenadchli.

PWA ale zajistilo, že se webová API hodně rozšířila a dnes máme přístup k mnoha funkcím zařízení, které dříve nebyly dostupné. Podobně je to s myšlenkou práce na pozadí – service workery, které se dají použít i na běžných stránkách.

Mimo PWA se dnes spíše používají nativní aplikace, které jsou vytvořené pomocí webových technologií.

7.4 Tvorba mobilních aplikací

Tvorba mobilních aplikací je vždy závislá na platformě (operacním systému), pro kterou aplikaci tvoříme. Nejznámější platformy jsou Android a iOS. Každá platforma má svůj vlastní způsob tvorby aplikací, své vlastní nástroje a jazyky.

Pro Android se aplikace tvoří nejčastěji v jazyce Java (resp. Kotlin), je potřeba mít naistalované Android SDK. Nejčastěji používáme Android Studio, což je oficiální IDE pro vývoj Android aplikací.

Pro iOS se aplikace tvoří nejčastěji v jazyce Swift, je potřeba mít nainstalované Xcode.

Nejčastější formáty pro distribuci aplikací jsou APK pro Android a IPA pro iOS.

Tvorba aplikací je neplacená. Po zkompilování aplikace je potřeba ji sdílet. V rámci platformy Android je možné instalovat aplikace typu APK. Přesto to není ideální způsob distribuce aplikací a proto se používají obchody s aplikacemi. Pro Android je nejznámější obchod Google Play. Pro zveřejnění aplikace v obchodě je potřeba získat vývojářský účet, který stojí jednorázový poplatek.

V rámci platformy iOS je instalace aplikací mimo App Store velmi omezená. Pro distribuci aplikací je potřeba použít App Store. Pro zveřejnění aplikace v App Store je potřeba získat vývojářský účet, který se platí ročně.

Pro vývoj aplikací potřebujou vývojáři potřebuji znát specifika dané platformy. To není vhodné pro vývojáře webových aplikací, kteří neznají nativní vývoj. Proto vznikly různé nástroje a frameworky, které umožňují vývoj mobilních aplikací pomocí webových technologií. Těmi se od teď budeme zabývat.

7.4.1 Webové zobrazení

Všechny tyto technologie fungují na podobném principu. Uvnitř všech typů platforem existuje komponenta, která lze vložit do rozhraní aplikace, která umí zobrazit webovou stránku. Tato komponenta se nazývá WebView – webové zobrazení.

WebView je v podstatě zabudovaný webový prohlížeč. Vykresluje tedy celé stránky a umožňuje interakci s nimi.

Uvnitř nativní aplikace tedy máme často pouze jednu komponentu – WebView, která zobrazuje webovou stránku.

To z aplikace ale nedělá nic zajímavého – je to stejné jako otevřená stránka v prohlížeči. Proto v nativní aplikaci vytvoříme propojení – API, které umožňuje komunikaci mezi nativní aplikací a webovou stránkou uvnitř WebView. Tímto způsobem můžeme využít funkce zařízení, které nejsou dostupné na webu – například přístup k souborovému systému, k fotoaparátu, k notifikacím a další.

Nativní aplikace tedy slouží jako obal, který umožňuje webové stránce přístup k funkcím zařízení a zobrazuje webovou stránku. Toto propojení, vytvoření aplikace s WebView a API je ale velmi náročné a vyžaduje znalosti nativního vývoje. Proto existují frameworky, které vám tyto základni včetně API vytvořili, a vy pouze tvoříte webovou stránku.

7.4.2 Capacitor

Capacitor je moderní framework pro tvorbu nativních aplikací pomocí webových technologií. Je vytvořený společností Ionic, která je známá svým komponentovým frameworkem pro tvorbu webových aplikací.

Capacitor umožňuje vývojářům vytvářet aplikace pomocí HTML, CSS a JavaScriptu, které mohou být nasazeny na různé platformy, jako jsou iOS, Android a web. Tedy dovoluje „export“ i do PWA.

Capacitor je možné přidat do jakéhokoliv webového projektu. Sestavení aplikace přímo Capacitor neprovádí, ale tvoří pro nás potřebné soubory pro nativní platformy. Například, pokud chceme vytvořit aplikaci pro Android, Capacitor vytvoří potřebné soubory a konfigurace. Poté můžeme otevřít projekt v Android Studiu a aplikaci sestavit a nasadit.

> **i Proč Capacitor nesestavuje aplikaci**
>
> Dobrý důvod pro to, proč Capacitor nepodporuje sestavení aplikace, je to, že i přes to, že chceme stejný základ, tak každá aplikace může vyžadovat různé změny v nativním projektu. Jedná se například o různé nastavení pravomocí, různé knihovny a třeba i vlastní nativní kód.

Po přidání capacitoru můžeme vytvořit soubory pro danou platformu. Poté při každé změně provádíme tzv. synchronizaci, která zkopíruje naše změny do nativního projektu. Kopírují se ale sestavné soubory. Pokud tedy chceme používat Vue, tak musíme nejdříve sestavit projekt a poté synchronizujeme nativní projekt se složkou s výsledným sestavením – například `dist`.

7.4.2.1 Použití Capacitoru

Capacitor využívá balíčkovacího nástroje npm. Pro přidání do existujícího projektu je nutné se ujistit, že:

- Existuje v kořenové složce soubor `package.json`,
- Projekt obsahuje v kořenové složce soubor `index.html`, který je vstupním bodem aplikace,
- Soubor `index.html` obsahuje alespoň tagy `<html>` a `<head>`,
- Existuje složka sestavení – soubory pro webovou stránku `dist` nebo `build`.

Poté do projektu můžeme přidat Capacitor pomocí příkazu:

bash

Instalace Capacitoru

```
1 npm install @capacitor/core
2 npm install @capacitor/cli --save-dev
```

Nainstaluje Capacitor do projektu.

Poté je nutné inicializovat nastavení Capacitoru pomocí příkazu:

bash

Inicializace Capacitoru

```
1 npx cap init
```

Inicializuje Capacitor v projektu.

Výše uvedený příkaz vytvoří soubor `capacitor.config.` (přípona `json`, `js` či `ts`) z informací, které zadáváme pomocí průvodce uvnitř terminálu. Tento soubor může vypadat například takto:

typescript

Příklad konfigurace Capacitoru

```
1 import { CapacitorConfig } from '@capacitor/cli';
2
3 const config: CapacitorConfig = {
4   appId: 'eu.cajthaml.ssps',
5   appName: 'SSPŠ Cajthaml',
6   webDir: 'dist',
7   bundledWebRuntime: false
8 };
9
10 export default config;
```

Příklad konfigurace Capacitoru v TypeScriptu.

Poté můžeme již přidat konkrétní platformu, například Android:

bash

Přidání platformy

```
1 npx cap add android
```

Přidá platformu Android do projektu.

Po tom, až budeme chtít aplikaci sestavit jako nativní aplikaci, je potřeba nejdříve sestavit webovou stránku. Poté můžeme provést synchronizaci.

bash

Sestavení a synchronizace

```
1 npm run build
2 npx cap sync
```

Sestaví webovou stránku a synchronizuje nativní projekt.

V dané složce, např. `android`, poté můžeme otevřít nativní projekt v Android Studiu a aplikaci sestavit a nasadit. To často děláme mimo aktuální počítač, například v rámci DevOps pipeline, viz Kapitola 2.5.

Android Studio (a obdobně ostatní IDE) můžeme otevřít pomocí příkazu `npx cap open android`. Sestavení potom provedeme přímo v IDE, nebo pomocí systému Gradle. Pro sestavení lze použít příkaz `gradlew assembleRelease`[32] uvnitř složky `android`.

Sestavenou aplikaci poté najdeme ve složce `android/app/build/outputs/apk/release`.

[32] Případně `gradlew.exe` na Windows.

7.4.2.2 Rozšíření

Přístup k funkcím zařízení je možný pomocí tzv. pluginů. Pro použití rozšíření si musíme zjistit, jestli naše platforma dané rozšíření podporuje. Pokud ano, je možné rozšíření nainstalovat.

Níže si ukážeme, jak se používá plugin pro přístup k souborům na zařízení.

Nainstalujeme plugin pomocí příkazu `npm install @capacitor/filesystem`. Poté musíme synchronizovat nativní projekt pomocí `npx cap sync` – to vytvoří potřebné soubory pro nativní projekt.

Poté uvnitř stránky, v jakémkoliv souboru můžeme plugin použít.

javascript

Použití pluginu

```
1 import { Filesystem, Directory, Encoding } from "@capacitor/filesystem";
2
3 await Filesystem.writeFile({
4     path: "project/ssps.json",
5     data: JSON.stringify({ name: "SSPŠ Cajthaml" }),
6     directory: Directory.Documents,
7     encoding: Encoding.UTF8,
8 });
```

Ukázka použití pluginu pro přístup k souborovému systému.

Pro některé zápisy a platformy potřebuje aplikace různé specifické pravomoce. Tyto úskalí nalezneme v dokumentaci k daném pluginu.

Pro fungování souborového systému je potřeba mít pro celou aplikaci povolený přístup k souborovému systému. Například v rámci Android musíme do souboru `android/app/src/main/AndroidManifest.xml` přidat následující řádek:

xml

Povolení přístupu k souborovému systému

```
1 <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/
  >
2 <uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Přidá povolení pro přístup k souborovému systému v Android.

Po změnách musíme aplikaci sestavit a znovu synchronizovat.

Mezi další metody v rámci tohoto pluginu patří například:

- **readFile** – čte soubor ze zařízení,
- **deleteFile** – smaže soubor ze zařízení,
- **readdir** – čte obsah adresáře,
- **mkdir** – vytvoří nový adresář,
- A další.

Mezi další často používané pluginy patří například:

- Clipboard – přístup ke schránce zařízení,
- Camera – přístup k fotoaparátu zařízení,
- Dialog – zobrazení nativních dialogů,
- Local Notifications – zobrazení nativních notifikací,

- Privacy Screen – ochrana obsahu aplikace v multitaskingu,
- A další.

Mimo oficiální pluginy existují i pluginy třetích stran, které poskytují další funkce.

7.4.2.3 Další práce

Kreativitě s Capacitorem se meze nekladou. Obyčejně tvoříte webové stránky a používáte různé pluginy pro přístup k funkcím zařízení.

I projekt, který používá Capacitor může být používán jako běžná webová stránka. Mimo to je možné použít různé frameworky pro tvorbu webových stránek, jako je Vue, React, Angular a další.

Mezi další známe alternativy k Capacitoru patří například:
- Cordova – starší framework pro tvorbu nativních aplikací pomocí webových technologií,
- React Native – framework pro tvorbu nativních aplikací pomocí Reactu,
- Flutter – framework pro tvorbu nativních aplikací pomocí jazyka Dart,
- A další.

7.5 Desktopové aplikace

Dalším odvětvím tvorby nativních aplikací jsou desktopové aplikace. Klasické desktopové aplikace jsou vývoje prakticky totožné jako již zmíněné mobilní aplikace, viz Kapitola 7.4. Místo toho, abychom řešili konkrétní jazyky, tak řešíme zejména cílové platformy. Nejznámější platformy jsou Windows, macOS a Linux.

Pro tvorbu aplikací lze použít skoro cokoliv, protože samotné UI aplikace je vždy specifické pro danou platformu, resp. její systémové knihovny. Klasickými aplikacemi se tu již více nebudeme zabývat, protože byste je měli zcela znát z předmětu Programování a vývoj aplikací z prvního a druhého ročníku.

My se znovu zaměříme na tvorbu nativní desktopových aplikací za pomoci webových technologií. A nebude pro vás žádné překvapení to, že je to znovu skoro totožné jako u mobilních aplikací. Máme nativní aplikaci, která obsahuje komponentu WebView, která zobrazuje webovou stránku. A máme API, které umožňuje přístup k funkcím zařízení.

Frameworky, které se používají pro tvorbu desktopových aplikací pomocí webových technologií, jsou například:
- Electron – nejznámější framework pro tvorbu desktopových aplikací pomocí webových technologií, používá Node.js,
- Tauri – moderní framework pro tvorbu desktopových aplikací pomocí webových technologií, je nutné ale znát Rust,
- A další.

My se seznámíme s frameworkem Electron, abychom si nemuseli vysvětlovat nový jazyk.

7.5.1 Electron

Electron[33] je nejznámější framework pro tvorbu desktopových aplikací pomocí webových technologií. Je známý jak pozitivně, tak i negativně. Určitě jste někdy slyšeli, že někdo řekl,

[33] Electron, Capacitor, to jsou jména.

že Electron aplikace jsou příliš velké, pomalé a žerou hodně paměti. Ona to je pravda, ale je to daň za to, že můžeme tvořit aplikace pomocí webových technologií. Každá Electron aplikace totiž obsahuje celý prohlížeč, který dokáže vykreslit webovou stránku. Prohlížeč je navíc Chromium[34], což je jeden z největších a nejtěžších prohlížečů.

Electron funguje v rámci procesů, kde každý zařizuje něco jiného. Komunikace mezi těmito procesy je možná, což si ukážeme později.

Existuje hlavní proces (main process), který je zodpovědný za správu aplikace, vytváření oken a komunikaci s operačním systémem. Ten tedy vytvoří okno aplikace, řekne mu, co se má vykreslit, nabídne mu informace pro to, co se má zobrazit v nabídce a další. Tady se taktéž implementuje komunikace s nativními funkcemi zařízení – můžeme sem přidat různé knihovny nebo psát vlastní kód, například s knihovnou `fs` či `child_process` z Node.js.

Dále existuje renderer proces (renderer process), který je zodpovědný za vykreslování webové stránky uvnitř okna aplikace. Ten je v podstatě webový prohlížeč, který umí vykreslit HTML, CSS a JavaScript. Může používat různé webové API, jako je DOM, Fetch API, WebSocket a další. Každé okno aplikace má svůj vlastní renderer proces.

Pro práci s Electron ho musíme nainstalovat. Samozřejmé používáme npm.

bash

Instalace Electronu

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

Nainstaluje Electron do projektu.

Poté můžeme vytvořit hlavní soubor aplikace, který bude obsahovat kód pro hlavní proces.

javascript

Hlavní proces Electronu

```javascript
const { app, BrowserWindow } = require('electron');
const path = require('path');
function createWindow() {
  const mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js'),
      contextIsolation: true,
      enableRemoteModule: false
    }
  });
  mainWindow.loadFile('index.html');
}
app.whenReady().then(() => {
  createWindow();
  app.on('activate', () => {
    if (BrowserWindow.getAllWindows().length === 0) {
      createWindow();
    }
  });
  app.on('window-all-closed', () => {
```

[34] Vzpomeňte si na jeho význam z WBA.

```
23     app.quit();
24   });
25 });
```

Vytvoří okno aplikace a načte webovou stránku.

V ukázce můžeme vidět, že až aplikace bude připravena, vytváříme okno aplikace pomocí třídy `BrowserWindow`, která obsahuje různé možnosti, jako je velikost okna a další. Dále načítáme soubor `index.html`, který bude vykreslen uvnitř okna aplikace. Poté se zachytíme na události aktivace, která se spustí když se aplikace načte poprvé. A poté události zavření všech oken, která ukončí aplikaci.

V rámci možností `webPreferences` můžeme nastavit různé možnosti pro renderer proces. Jedna z těch důležitějších je `preload`, což je cesta k souboru, který se načte před tím, než se načte webová stránka. Electron totiž po zapnutí okna vytvoří základní kontext pro vykreslení stránky. Poté načte soubor `preload.js`, který může upravit tento kontext – můžeme přidat propojení mezi hlavním procesem a renderer procesem. Nakonec se načte samotná webová stránka.

Do preload dáváme právě samotnou komunikaci s nativní částí aplikace. Soubor může vypadat například takto:

javascript

Preload skript

```javascript
const { contextBridge } = require('electron')
const fs = require('fs')

contextBridge.exposeInMainWorld('api', {
  node: () => process.versions.node,
  getFileData: () => fs.readFileSync('data.json', 'utf-8')
})
```

Přidá do globálního objektu `window` objekt `api` s funkcemi pro přístup k nativním funkcím.

V ukázce můžeme vidět, že používáme `contextBridge`, který umožňuje bezpečně vystavit API z hlavního procesu do renderer procesu.

> **Bezpečnost v Electronu**
>
> Je důležité, aby renderer proces neměl přímý přístup k Node.js API, protože by to mohlo vést k bezpečnostním problémům. Omezte jakýmkoliv způsobem možný neoprávněný přístup k nativním funkcím. Vždy je nutné validovat vstup a výstup z těchto funkcí.

V rámci webové stránky potom můžeme použít toto API.

html

Použití API v renderer procesu

```html
<!-- ... -->
<script>
  console.log('Node.js verze:', window.api.node());
  console.log('Data ze souboru:', window.api.getFileData());
</script>
```

Používá API vystavené v preload skriptu.

K další výměně dat mezi hlavním procesem a renderer procesem můžeme použít IPC (Inter-Process Communication). To je způsob pro posílání zpráv mezi těmito procesy. Velmi jednoduše můžeme pomocí tohoto taktéž navázat dvoustrannou komunikaci, případně komunikaci z aplikace do vykreslené stránky.

Uvnitř preload skriptu můžeme definovat takové funkce, které zašlou zprávu do hlavního procesu. V hlavní procesu můžeme například pracovat se zařízením, ale taktéž třeba otevřít či zavřít okno aplikace. Uvnitř hlavního procesu poté definujeme jak se na danou zprávu zareaguje.

Role preload skriptu

Preload si je nutné představit jako most mezi hlavním procesem a renderer procesem. Při různém nastavení nemusí být použit, ale v základu slouží jako bezpečností složka, která rozděluje chování mezi těmito procesy.

javascript

Preload skript s IPC

```javascript
const { contextBridge, ipcRenderer } = require('electron');
contextBridge.exposeInMainWorld('api', {
  closeApp: () => ipcRenderer.send('close-app')
});
```

Přidá do globálního objektu `window` funkci pro zavření aplikace.

javascript

Hlavní proces s IPC

```javascript
const { app, BrowserWindow, ipcMain } = require('electron');
ipcMain.on('close-app', () => {
  app.quit();
});
```

Zachytí zprávu pro zavření aplikace a ukončí ji.

7.5.2 Možnosti Electronu

Electron nabízí řadu možnosti a API, které umožňují přístup k různým funkcím aplikace. Dle operačního systému můžeme například nastavovat funkcionality při najetí myší na ikonu aplikace, přidávat položky do systémové nabídky, přidávat zkratky a další. Mimo to můžeme používat různé pluginy a knihovny, které rozšiřují možnosti Electronu.

7.5.3 Sestavení aplikace

Základní knihovna Electron neobsahuje nástroje pro sestavení aplikace do spustitelného formátu. Pro sestavení aplikace můžeme použít různé nástroje, neznámější z nich je `electronforge`.

Pro sestavení musíme nainstalovat balíček `@electron-forge/cli`. Poté je v již existujícím projektu nutné inicializovat nastavení pomocí:

bash

Instalace a inicializace Electron Forge

```
1 npm install --save-dev @electron-forge/cli
2 npm exec --package=@electron-forge/cli -c "electron-forge import"
```

Nainstaluje a inicializuje Electron Forge v projektu.

Poté je možné v projektu zavolat tři nové příkazy:

- `npx electron-forge start` – spustí aplikaci v režimu vývoje,
- `npx electron-forge package` – sestaví aplikaci do složky `out`,
- `npx electron-forge make` – sestaví aplikaci do instalačního formátu pro danou platformu, například `.exe` pro Windows, `.dmg` pro macOS nebo `.deb` pro Linux.
- `npx electron-forge publish` – zveřejní aplikaci na zvoleném místě, například GitHub Releases.

Tyto příkazy můžeme přidat do souboru `package.json` do sekce `scripts`, aby bylo možné je spouštět jednoduše pomocí `npm run <příkaz>`. Ze základu tyto příkazy staví aplikaci pro aktuální platformu. Je možné pomocí přepínačů sestavit aplikaci pro jiné platformy, například `--platform=win32` pro Windows, `--platform=darwin` pro macOS nebo `--platform=linux` pro Linux.

7.5.4 Pokročilé koncepty

Mezi pokročilé koncepty pro tvorbu nativních desktopových aplikací patří například:

- **Automatické aktualizace** – umožňují aplikaci automaticky se aktualizovat na nejnovější verzi bez nutnosti uživatelského zásahu,
- **Vlastní ikony** – umožňují aplikaci mít vlastní ikony pro různé platformy a velikosti,
- **Podepsání aplikace** – umožňují aplikaci být podepsána, což zvyšuje důvěryhodnost a bezpečnost aplikace, nutné pro nějaké platformy,
- A další.

7.6 Budoucnost webových aplikací

Webové aplikace se stále vyvíjejí a získávají nové možnosti. S příchodem nových webových API a technologií se webové aplikace stávají stále více schopnými a konkurenceschopnými s nativními aplikacemi. Jen čas ukáže, jakým směrem se budou webové aplikace ubírat.

Mimo ostatní věci se očekává, že webové aplikace budou mít stále lepší přístup k funkcím zařízení, jako je například přístup k souborovému systému, k fotoaparátu, k notifikacím a další. Proč to tak není už teď je jednoduché – může to být zneužito k útokům na uživatele. Web je aktuálně v zajímavém stavu, kdy nikdo moc neví, jestli je dobrý nápad více zvýšit možnosti webových aplikací.

V poslední době se hodně mluví o bezpečnosti a ochraně soukromí uživatelů. Doporučuji se vám podívat například na to, co „nedávno“ navrhli developeři z Google k ověřování uživatelů na webu, viz Web Environment Integrity.

Populární novinka je také WebAssembly, což je technologie, která umožňuje spouštět kód napsaný v různých jazycích (například C, C++, Rust) přímo v prohlížeči. A to v podstatě nativním výkonem. To otevírá nové možnosti pro webové aplikace, které mohou být nyní ještě výkonnější a schopnější. WebAssembly je ale samostatné téma, které je velmi rozsáhle a proto se jím zde nebudeme zabývat.

Webové frameworky

8.1 Frameworky

Ihned na začátek si představíme, resp. připomenem, co je to framework. Framework lze klasifikovat různými způsoby.

Framework pro nás je sada knihoven, které nám pomáhají vytvářet nějakou aplikaci. V rámci webů jsme zvyklí na různé typy frameworků – backendové, frontendové a CSS. Backendové frameworky nám pomáhají vytvářet webové aplikace na serveru – jedná se například o Django, Express.js nebo více high-level, treba Nest.js. CSS frameworky nám pomáhají vytvářet vzhled webových stránek tím, že mají předpřipravené styly a komponenty – jedná se například o Bootstrap, Tailwind CSS nebo Bulma. Frontendové frameworky nám pomáhají vytvářet aplikace, které běží v prohlížeči – jedná se například o React, Vue, Svelte, Ember nebo Angular.

V této kapitole se budeme věnovat frontendovým frameworkům – ukážeme si nejznámější alternativy a jejich principy. Na začátku si ukážeme Vue a odhalíme si princip, o kterém jsme se zatím nezmínili.

8.2 Vue

Vue je framework, který jsme se skoro půlku minulého roku učili. Je to moderní framework a patří mezi ty nejpoužívanější. Pro výuku jsem musel nějaký z frameworků zvolit a Vue je dobrá volba. Ukazuje totiž principy, které jsou společné i pro jiné frameworky a přesto je poměrně jednoduchý na naučení.

Pro funkci Vue potřebujeme správce projektu – bundler. Minulý rok jsme využívali Vite, který má velmi jednoduchou konfiguraci a je rychlý. Vite jde používat i s jinými frameworky.

Důležitá věc, kterou jsme si o Vue neřekli je to, že Vue využívá tzv. virtuální DOM (VDOM). My již víme, že DOM je objektový model dokumentu, který reprezentuje HTML dokument jako strom uzlů. Tyto uzly potom můžeme v rámci prohlížeče měnit pomocí JavaScriptu.

8.2.1 Virtuální DOM

Vue, a další frameworky, vytváří virtuální reprezentaci DOMu v paměti. Jedná se tedy o abstrakci abstrakce HTML dokumentu. Pro pochopení proč je někdy vhodné implementovat VDOM je nutné pochopit jak vůbec frameworky fungují a jak změny DOM řeší prohlížeče.

Uvnitř frameworků je tzv. reaktivní systém[35], který sleduje změny dat a podle toho aktualizuje uživatelské rozhraní. Pokud tedy změníme nějaká data, které se vykreslují uvnitř šablony, tak framework zjistí, že se data změnila a aktualizuje se stránka. Komplexní šablony ale často změní mnoho částí DOMu.

Problém je, že změny v DOMu jsou pomalé. Každá změna v DOMu způsobí, že prohlížeč musí znovu vykreslit stránku. A to může způsobit drahé operace, jako je přepočítání stylů, rozložení a malování. Uvnitř aplikací se ale prvky mění pořad a často. DOM ale musí změny zkontrolovat a aktualizovat stránku.

Proto frameworky využívají VDOM. Místo toho, aby se změny hned předali DOMu, který začne drahé operace, tak se změny nejdříve provedou ve VDOMu. VDOM je v podstatě to stejné jako DOM, obsahuje objekty, které reprezentují jednotlivé elementy na stránce. Při

[35] To je to nejdůležitější na frameworcích.

změně se změní tento abstrahovaný model. Mezi změnami se poté vypočítá rozdíl (diffing) mezi starým a novým VDOMem. Nakonec se tyto změny aplikují na skutečný DOM najednou. Tím se minimalizuje počet změn v DOMu a tím i drahých operací, které musí prohlížeč provést.

Jako ukázku si můžeme uvést změny v DOM za pomocí `style` a za pomocí `innerHTML`. Pokud změníme `style` nějakého elementu, tak prohlížeč musí přepočítat styly, rozložení a malování. Pokud ale změníme `innerHTML`, tak prohlížeč musí přepočítat celý DOM pod tímto elementem, vytvořit možné nové elementy a potom je vykreslit. Změnou `innerHTML` tedy může dojít k tomu, že ztratíme závislosti na elementech a proto je budeme moci znovu vytvořit.

VDOM vyřeší tento problém, ale některé frameworky ho vůbec nepoužívají. Použití VDOM má totiž i své nevýhody. Jedna z nich je to, že sice ušetříme drahé operace v DOMu, ale na druhou stranu přidáme režii navíc, protože musíme udržovat VDOM a počítat rozdíly mezi starým a novým VDOMem. Navíce to reálně (při hrůzných implementacích) může vést k tomu, že se aplikace zpomalí. Některé frameworky používají při kompilaci šablon různé optimalizace, aby se minimalizovalo množství změn v DOMu a tím i potřeba VDOM – například používají tzv. optimální DOM operace.

8.3 React

React je nejpopulárnější[36] framework. React je vyvíjen Facebookem a je to knihovna pro vytváření uživatelských rozhraní. Je určen na vytváření malých stránek (dokonce pouze komponent) ale i gigantických aplikací.

React je založen na komponentovém přístupu, kde každá část uživatelského rozhraní je reprezentována jako samostatná komponenta. Komponenty mohou být znovu použity a složeny dohromady, aby vytvořily složitější uživatelská rozhraní. Stejně jako Vue, i React používá VDOM pro efektivní aktualizaci uživatelského rozhraní.

Největší rozdíly mezi Vue a Reactem jsou v syntaxi a přístupu k datům. Data v Reactu jsou předávány jako vlastnosti (props) a stav (state) je spravován pomocí tzv. hooks. React také používá JSX, což je rozšíření syntaxe JavaScriptu, které umožňuje psát HTML podobný kód uvnitř JavaScriptu.

Všechny komponenty v Reactu jsou tedy funkce, které si uchovávají svůj stav a vrací kód, který definuje, jak má komponenta vypadat.

tsx

Hlavní soubor React aplikace

```tsx
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import "./styles.css";

import App from "./App";

const root = createRoot(document.getElementById("root"));
root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
```

[36] Při psaní ukápla nejedna slza.

tsx

Počítací komponenta v Reactu

```tsx
import React, { useState } from 'react';
export default function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Počet kliknutí: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Klikni na mě
      </button>
    </div>
  );
}
```

Ve výše uvedeném příkladu máme komponentu `Counter`, která si uchovává stav `count` pomocí hooku `useState`. Když uživatel klikne na tlačítko, zavolá se funkce `setCount`, která aktualizuje stav a způsobí, že se komponenta znovu vykreslí s novým počtem kliknutí.

Komponenty poté můžeme skládat dohromady a vytvářet tak složitější uživatelská rozhraní.

tsx

Složená aplikace v Reactu

```tsx
import React from 'react';
import Counter from './Counter';
function App() {
  return (
    <div>
      <h1>Moje aplikace</h1>
      <Counter />
      <Counter />
    </div>
  );
}
```

Mezi komponenty můžeme předávat data pomocí vlastností (props).

tsx

Předávání dat pomocí vlastností v Reactu

```tsx
import React from 'react';
function Greeting(props) {
  return <h1>Ahoj, {props.name}!</h1>;
}
function App() {
  return (
    <div>
      <Greeting name="Matěj" />
      <Greeting name="Pepík" />
    </div>
  );
```

```
12 }
```

Podobně můžeme oznámit změnu rodičovské komponentě.

tsx

Oznámení rodiči v Reactu

```tsx
import React, { useState } from 'react';
function Child(props) {
  return (
    <button onClick={() => props.onNotify('Ahoj z dítěte!')}>
      Oznám rodiči
    </button>
  );
}
function Parent() {
  const [message, setMessage] = useState('');
  return (
    <div>
      <Child onNotify={(msg) => setMessage(msg)} />
      <p>Zpráva od dítěte: {message}</p>
    </div>
  );
}
```

React má také rozsáhlý ekosystém knihoven a nástrojů, které usnadňují vývoj aplikací. Mezi nejpopulárnější patří:

- **React Router** pro správu routování v aplikacích,
- **Redux** pro správu stavu aplikace.

Obě knihovny fungují skoro ekvivalentně jako alternativy ve Vue.

8.4 Svelte

Svelte je moderní framework, který se poměrně hodně liší od klasických frameworků. Svelte cílí na minimalismus, jednoduchost a výkon.

Svelte je kompilátor, který překládá komponenty do vysoce optimalizovaného JavaScriptu, který přímo manipuluje s DOMem. To znamená, že Svelte nevyužívá VDOM, ale místo toho generuje kód, který přímo aktualizuje DOM, když se změní stav aplikace.

Stejně jako jinde zde máme koncept komponent, které si uchovávají svůj stav a vykreslují uživatelské rozhraní.

svlt

Počítací komponenta v Svelte

```svlt
<script lang="ts">
  let count: number = $state(0)
  const increment = () => {
    count += 1
  }
</script>

```

```
 8 <button onclick={increment}>
 9   count is {count}
10 </button>
```

Ve výše uvedeném příkladu máme komponentu, která si uchovává stav `count` a funkci `increment`, která zvyšuje počet kliknutí. Stav je reaktivní, takže když se změní, komponenta se znovu vykreslí s novým počtem kliknutí.

svlt

Složená aplikace ve Svelte

```
1 <script lang="ts">
2   import Counter from './Counter.svelte'
3 </script>
4 <main>
5   <h1>Moje aplikace</h1>
6   <Counter />
7   <Counter />
8 </main>
```

Do komponent můžeme předávat data pomocí vlastností (props).

svlt

Předávání dat pomocí vlastností ve Svelte

```
1 <script lang="ts">
2   let { name } = $props();
3 </script>
4 <h1>Ahoj, {name}!</h1>
```

svlt

Použití komponenty s vlastností ve Svelte

```
1 <Greeting name="Matěj" />
```

svlt

Oznámení rodiči ve Svelte

```
1 <script lang="ts">
2   let { notify } = $props();
3 </script>
4
5 <button onclick={() => notify('Ahoj z dítěte!')}>
6   Oznám rodiči
7 </button>
```

Podobně jako ve Vue má i Svelte koncept přímého navázání hodnoty rodiče a dítěte.

Svelte má svůj vlastní ekosystém knihoven a nástrojů, které usnadňují vývoj aplikací.

Stejně jako ostatní má alternativu pro správu stavu přímo ve frameworku – Svelte Store. Pro routování neexistuje oficiální knihovna, ale je zde několik populárních alternativ. Svelte se ale nejčastěji používá s SvelteKit, o čemž si povíme více, viz Kapitola 8.7.

8.5 Ember

Ember je framework, který je určen pro vytváření ambiciózních webových aplikací. Ember je založen na komponentovém přístupu a používá VDOM pro efektivní aktualizaci uživatelského rozhraní. Zakládá si na konvencích před konfigurací, což znamená, že má předdefinované struktury a způsoby, jak dělat věci. Můžeme si to představit jako to, že máme pro každou věc „správný“ způsob, jak to udělat. To může být výhodné pro větší týmy, protože to zajišťuje konzistenci v kódu. Pro menší projekty to ale může být omezující. Ember má také svůj vlastní ekosystém knihoven a nástrojů, které usnadňují vývoj aplikací. Velmi dobře například řeší i testování aplikace.

Ember má velkou křivku učení, protože má mnoho konceptů a nástrojů, které je třeba pochopit. To může být výzva pro nové vývojáře, ale na druhou stranu to může vést k lepšímu kódu a struktuře aplikace.

Více si o Emberu neřekneme, protože je to poměrně specifický framework, který se používá zejména ve velkých projektech. Nejznámější projekt postaven na Emberu je pravděpodobně webová aplikace LinkedIn.

8.6 Angular

Poslední framework, který si představíme je Angular. Angular je poměrně starý framework, který je vyvíjen Googlem. Angular je kompletní framework, který nabízí vše, co potřebujeme pro vytváření webových aplikací. Angular je založen na komponentovém přístupu a používá VDOM pro efektivní aktualizaci uživatelského rozhraní.

Angular považuje mnoho lidí jako první framework, který přinesl řadu konceptů, ze kterých se ostatní frameworky inspirovaly. Mezi tyto koncepty patří například:

- **Two-way Data Binding** což je způsob, jakým se data synchronizují mezi modelem a zobrazením,
- **Directives** což jsou speciální atributy, které umožňují přidávat chování k HTML elementům.

Obecný konsensus mezi webovými vývojáři je, že Angular je poměrně těžký na naučení a používání ale nemá skoro žádné výhody oproti ostatním frameworkům. Proto se dnes již moc nepoužívá a většina nových projektů volí jiné frameworky. Navíce za ním již není tak velká komunita a ekosystém knihoven a nástrojů.

8.7 Frameworky s Server-Side Renderingem

Všechny frameworky, které jsme si představili, jsou tzv. client-side frameworky. To znamená, že aplikace běží v prohlížeči a veškerý kód je stažen do prohlížeče.

To má své výhody i nevýhody. Již dříve, viz Kapitola 6.6, jsem si ukázali, že SPA aplikace mají obecně horší SEO. Je to tím, že vyhledávače mají problém s indexací stránek, které jsou generovány na straně klienta. Jedno z řešení, které jsme si ukázali, je SSR.

SSR znamená, že aplikace je vykreslena na serveru a poté je odeslána do prohlížeče jako hotová HTML stránka. Na serveru navíce můžeme udělat i další věci, jako je například přednačtení dat z API, které aplikace potřebuje.

Některé frameworky mají oficiální nástroje[37] pro SSR.

- **Next.js** je nástroj pro React,
- **Nuxt.js** je nástroj pro Vue,
- **SvelteKit** je nástroj pro Svelte.

Všechny tyto nástroje dělají v podstatě to samé – umožňují nám vytvářet aplikace, které jsou vykresleny na serveru. Tyto nástroje mají navíce i další funkce, jako je například:

- **Routing** což je způsob, jakým se naviguje mezi různými stránkami aplikace,
- **Static Site Generation (SSG)** což je způsob, jakým se generují statické stránky z aplikace,
- **API Routes** což je způsob, jakým můžeme vytvářet API přímo v rámci aplikace.

Tyto nástroje jsou velmi populární a jsou často používány pro vytváření moderních webových aplikací.

[37] Ne nutně je dělají stejní lidé, ale mají vzájemnou podporu.

Závěr

9.1 Shrnutí

Během našich dvou (tří) let společného studia jsme se společně vydali na cestu poznávání webových technologií od základů až po nejmodernější trendy a pokročilé koncepty.

V této poslední publikaci pro 4. ročník jsme probrali témata, která vás připravují na skutečnou profesionální praxi:

- **Projekty a jejich řízení**: Naučili jste se, jak fungují metodiky jako Scrum, Kanban a Extreme Programming. Pochopili jste důležitost testování, CI/CD a DevOps. Tyto znalosti jsou klíčové pro práci ve vývojářských týmech.
- **Animace a interaktivita**: Prohloubili jsme znalosti CSS animací, WebGL, Canvas API a moderních přístupů k vytváření interaktivních webových aplikací.
- **Alternativní protokoly**: Seznámili jste se s dvoucestnou komunikací, WebSockety a dalšími moderními způsoby komunikace, které jsou nezbytné pro aplikace.
- **SEO a vyhledávače**: Pochopili jste, jak fungují webové vyhledávače a jak optimalizovat stránky pro lepší viditelnost. Znáte různé faktory ovlivňující úspěch webu.
- **Moderní webové aplikace**: Naučili jste se o PWA, desktopových a webových aplikacích tvořených webovými technologiemi a hybridních přístupech.
- **Webové frameworky**: Prošli jsme si Vue, React, Svelte, Angular a další nástroje, které dominují modernímu webovému vývoji. Pochopili jste jejich výhody, nevýhody a vhodné scénáře použití.

Všechna tato témata mají jeden společný cíl – **naučit vás vytvářet kvalitní, rychlé, dostupné a uživatelsky přívětivé webové aplikace**.

9.2 Co je opravdu důležité pro weby

Po letech studia a praxe mohu s jistotou říci, že nejdůležitějšími aspekty úspěšného webu jsou:

1. **Výkon a rychlost**: Uživatelé nečekají. Stránka se musí načíst rychle a reagovat okamžitě.
2. **Dostupnost (accessibility)**: Web musí být použitelný pro všechny uživatele, včetně těch se zdravotním hendikepem.
3. **Responzivní design**: Aplikace musí fungovat na všech zařízeních – od mobilu po desktop až k ledničkám.
4. **SEO optimalizace**: Bez viditelnosti ve vyhledávačích váš web ledakdo nenajde.
5. **Bezpečnost**: Chraňte data uživatelů a zajistěte bezpečnou komunikaci.
6. **Udržovatelnost kódu**: Pište čistý, dokumentovaný kód s testy. Budoucí já vám poděkuje.
7. **Uživatelský zážitek (UX)**: Technická dokonalost je zbytečná, pokud aplikace není intuitivní a příjemná na používání.

Pamatujte si: **Technologie jsou jen nástroje. Důležité je řešit skutečné problémy uživatelů.**

9.3 Motivace do budoucna

Doufám, že jsem vám za tyto roky dokázal předat nejen technické znalosti, ale také nadšení pro webové technologie a chuť neustále se učit nové věci.

Webový vývoj je obor, který se neustále mění a vyvíjí. To, co jste se naučili, jsou pevné základy, ale nikdy se nepřestávejte vzdělávat. Nové frameworky, knihovny a přístupy vznikají každý den.

Věřím, že máte nyní dostatečné znalosti a dovednosti pro to, abyste se mohli vydat vlastní cestou – ať už na vysokou školu, do praxe nebo k vlastnímu podnikání.

Těším se na naše ústní zkoušky u maturity, kde si budeme moci povídat o vašich zkušenostech, projektech, a ukázat, jak daleko jste se dostali. Bude to příležitost předvést, co všechno umíte a jak dokážete aplikovat získané znalosti nejen v praxi.

Přeji vám hodně štěstí na dalších cestách a doufám, že webové technologie zůstanou součástí vašeho života – ať už jako koníček nebo jako profese.

9.4 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.

JavaScript a frameworky:
- The Modern JavaScript Tutorial – komplexní guide pro moderní JavaScript,
- Vue.js dokumentace – oficiální dokumentace Vue,
- React dokumentace – oficiální dokumentace React,
- Svelte dokumentace – oficiální dokumentace Svelte.

Performance a optimalizace:
- Core Web Vitals – Google metriky pro výkon webu,
- PageSpeed Insights – nástroj pro testování rychlosti stránek.

SEO a přístupnost:
- Google Search Central – oficiální dokumentace pro SEO,
- WCAG Guidelines – standardy pro přístupnost webu.

Vývojářské nástroje a DevOps:
- GitHub Actions – CI/CD na GitHubu,
- Git dokumentace – kompletní návod pro verzování kódu pomocí Gitu.

Komunity a učení:
- Stack Overflow – největší komunita vývojářů,
- Dev.to – články a tutoriály od vývojářů,
- CSS-Tricks – pokročilé techniky a tutoriály pro CSS.

Podcasty a video obsah:
- Traversy Media – praktické tutoriály,
- Web Dev Simplified – jednoduché vysvětlení složitých konceptů.

Nezapomeňte, že nejlepší způsob učení je **praxe**. Vytvářejte vlastní projekty, experimentujte s novými technologiemi a nebojte se dělat chyby – z těch se člověk naučí nejvíce.

—

Hodně štěstí na vaší další cestě webovým vývojem! Pevně věřím, že se v životě znovu potkáme.

Díky za skvělou spolupráci a přeji vám vše nejlepší do budoucna.

Příloha A.

Ověření znalostí

A.1 Projekty a jejich řízení

- ☐ Rozumím pojmu „projekt“ v kontextu vývoje softwaru.
- ☐ Vím, jaké jsou hlavní fáze projektu (analýza, návrh, implementace, testování, údržba).
- ☐ Umím vysvětlit rozdíl mezi funkčními a nefunkčními požadavky.
- ☐ Rozumím pojmu „datový model“ a „use case“.

- ☐ Rozumím principům Waterfall metodiky.
- ☐ Vím, jaké jsou výhody a nevýhody vodopádového modelu.
- ☐ Umím vysvětlit, proč se Waterfall hodí pro státní zakázky.

- ☐ Rozumím základním principům agilního vývoje.
- ☐ Vím, v čem se agilní metodiky liší od Waterfallu.

- ☐ Rozumím pojmům: sprint, Product Backlog, Sprint Backlog.
- ☐ Vím, jaké jsou hlavní Scrum ceremonie (Planning, Daily, Review, Retrospective).
- ☐ Umím vysvětlit role: Product Owner, Scrum Master, Development Team.
- ☐ Rozumím rozdílu mezi těmito rolemi.

- ☐ Rozumím principům Kanbanu.
- ☐ Vím, co je WIP (Work In Progress) limit a proč je důležitý.
- ☐ Umím vysvětlit rozdíl mezi Scrum a Kanban.

- ☐ Rozumím pojmům: Pair Programming, TDD, Refactoring, Continuous Integration.
- ☐ Vím, jaké jsou výhody těchto praktik.

- ☐ Rozumím účelu DTO.
- ☐ Vím, proč oddělujeme databázové modely od DTO.
- ☐ Rozumím vlastnostem DTO (jednoduchost, serializovatelnost, immutabilita).

- ☐ Rozumím rozdílu mezi unit, integration, system a acceptance testy.
- ☐ Vím, co jsou funkční vs. nefunkční testy.
- ☐ Rozumím pojmu regresní testování.
- ☐ Vím, co je TDD (Test Driven Development).
- ☐ Rozumím výhodám a nevýhodám automatického testování.
- ☐ Vím, kdy je vhodné použít uživatelské testování.
- ☐ Znám nástroj jako Cypress pro end-to-end testování.
- ☐ Rozumím účelu linterů a formátovačů (ESLint, Prettier).
- ☐ Vím, co je statická analýza kódu.
- ☐ Umím vysvětlit problémy, které statická analýza odhaluje.

- ☐ Rozumím pojmu DevOps a jeho cíli.
- ☐ Vím, co je Continuous Integration (CI).

- [ ] Vím, co je Continuous Delivery/Deployment (CD).
- [ ] Rozumím pojmu Infrastructure as Code (IaC).

A.1.1 Praktická část

- [ ] Umím vytvořit základní analýzu požadavků pro jednoduchý projekt.
- [ ] Dokážu identifikovat hlavní scénáře použití (use cases) aplikace.

- [ ] Dokážu rozpoznat, kdy je vhodné použít Waterfall a kdy agilní metodiky.
- [ ] Dokážu vysvětlit, jak probíhá typický sprint.
- [ ] Umím popsat, co dělá každá role ve Scrumu.
- [ ] Dokážu rozpoznat běžné chyby při implementaci Scrumu (např. Scrum Master jako manažer).

- [ ] Dokážu vytvořit jednoduchý Kanban board.

- [ ] Umím vytvořit DTO pro jednoduchý případ použití.
- [ ] Dokážu převést databázový model na DTO (odstranit citlivá data).
- [ ] Umím implementovat validaci DTO.

- [ ] Umím napsat základní unit test v JavaScriptu.
- [ ] Dokážu vytvořit testovací scénář pro uživatelské testování.
- [ ] Umím vysvětlit, co je A/B testování a kdy se používá.

- [ ] Dokážu přečíst a pochopit jednoduchý GitHub Actions workflow.

A.1.2 Bonusové části

- [ ] Znám varianty Scrumu (Scrum of Scrums, SAFe, Scrumban, LeSS)
- [ ] Vím, kdy a proč se používají tyto varianty.

- [ ] Umím vysvětlit základní workflow CI/CD.
- [ ] Umím vysvětlit, co je runner/spouštěč v kontextu CI/CD.

- [ ] Vím, kde se umísťují workflow soubory (.github/workflows/)
- [ ] Rozumím základní struktuře YAML workflow souboru.
- [ ] Umím vysvětlit koncept „jobs“ a „steps“.
- [ ] Vím, jak fungují spouštěče (triggers) – např. on push, on pull_request.

A.2 Autorizace a autentizace

- ☐ Rozumím rozdílu mezi autentizací a autorizací.
- ☐ Vím, co je Multi-Factor Authentication (MFA) a Two-Factor Authentication (2FA).
- ☐ Umím vysvětlit tři faktory autentizace (co znám, co mám, čím jsem).
- ☐ Rozumím pojmu Single Sign-On (SSO).

- ☐ Rozumím principům Permission-Based Access Control (PBAC).
- ☐ Rozumím principům Role-Based Access Control (RBAC).
- ☐ Vím, kdy použít práva a kdy role.
- ☐ Umím vysvětlit rozdíl mezi RBAC a PBAC.

- ☐ Znám další modely autorizace (ABAC, ACL, Policy-Based).
- ☐ Vím, v jakých scénářích se tyto modely používají.

- ☐ Rozumím pojmu „relace“ (session) na webu.
- ☐ Vím, proč HTTP potřebuje relace (bezstavovost protokolu).
- ☐ Rozumím, jak se relace ukládají a identifikují (session ID, cookies).
- ☐ Vím, jaká je životnost relace a jak se spravuje.

- ☐ Rozumím rozdílu mezi náhodnými a bezstavovými tokeny.
- ☐ Vím, co je JWT (JSON Web Token) a z čeho se skládá.
- ☐ Rozumím struktuře JWT (header, payload, signature).
- ☐ Vím, k čemu slouží refresh token.
- ☐ Umím vysvětlit, proč je JWT podepsaný a jak se ověřuje.

- ☐ Rozumím principům OAuth 2.0.
- ☐ Vím, k čemu OAuth slouží (delegovaná autorizace, SSO).
- ☐ Rozumím základnímu OAuth flow (authorization code).
- ☐ Vím, co je client ID a client secret.
- ☐ Umím vysvětlit, proč OAuth vyžaduje redirect URI.

A.2.1 Praktická část

- ☐ Umím implementovat základní relační systém v Express.js.
- ☐ Dokážu vytvořit a ověřit JWT token.
- ☐ Umím bezpečně ukládat session ID do cookies.

- ☐ Umím implementovat kontrolu práv v aplikaci.
- ☐ Dokážu vytvořit systém rolí pro uživatele.
- ☐ Umím kombinovat role a práva v jednom systému.

- ☐ Umím implementovat základní OAuth 2.0 přihlášení (např. přes Google).

- [ ] Dokážu zpracovat OAuth callback a získat access token.
- [ ] Umím bezpečně uložit OAuth tokeny.

- [ ] Vím, jak bezpečně ukládat hesla (hashing, ne plaintext).
- [ ] Rozumím důležitosti HTTPS pro autentizaci.

A.2.2 Bonusové části

- [ ] Rozumím rozdílu mezi stateful a stateless autentizací.
- [ ] Vím, jaké jsou výhody a nevýhody JWT oproti session-based auth.
- [ ] Znám bezpečnostní rizika spojená s JWT (token expiration, token storage).

- [ ] Vím, co je PKCE (Proof Key for Code Exchange) v OAuth.
- [ ] Rozumím rozdílu mezi Authorization Code Flow a Implicit Flow.
- [ ] Znám bezpečnostní best practices pro OAuth implementaci.

- [ ] Umím vysvětlit, co je token rotation a proč se používá.
- [ ] Znám strategie pro revokaci tokenů.
- [ ] Vím, jak funguje biometrická autentizace na webu.

A.3 Animace a interaktivita na webu

- [ ] Rozumím základním CSS přechodům (transition).
- [ ] Rozumím základním CSS animacím (@keyframes).
- [ ] Vím, jak animovat pomocí DOM API (element.animate()).

- [ ] Rozumím komponentě `<transition>` ve Vue.
- [ ] Vím, jaké třídy Vue přidává při animacích (enter-from, enter-active, enter-to, leave-*).
- [ ] Rozumím komponentě `<transition-group>` pro animaci seznamů.
- [ ] Vím, co dělá třída `*-move` u transition-group.

- [ ] Rozumím problému s setInterval/setTimeout pro animace.
- [ ] Vím, proč je requestAnimationFrame lepší pro plynulé animace.
- [ ] Rozumím pojmu delta time a proč je důležitý.
- [ ] Umím vysvětlit, proč změny DOM jsou drahé operace.

- [ ] Rozumím interpolačním funkcím (easing functions).
- [ ] Znám knihovny pro animace (GSAP, Anime.js, Motion.dev).

- [ ] Rozumím pojmu Web Worker a k čemu slouží.
- [ ] Vím, jak Web Workers komunikují (postMessage).
- [ ] Rozumím omezením Web Workers (nemají přístup k DOM).

- [ ] Rozumím HTML elementu `<canvas>`.
- [ ] Vím, co je kontext canvasu (2d, webgl).
- [ ] Rozumím pojmu „cesta“ (path) v 2D canvasu.
- [ ] Umím vysvětlit rozdíl mezi stroke() a fill().

- [ ] Vím, jak kreslit základní tvary (obdélník, kruh, čáry).
- [ ] Rozumím vykreslování textu na canvas.
- [ ] Rozumím vykreslování obrázků na canvas (drawImage).
- [ ] Vím, jak smazat obsah canvasu (clearRect).

- [ ] Rozumím transformacím v canvasu (translate, rotate, scale).
- [ ] Vím, k čemu slouží save() a restore().
- [ ] Umím vysvětlit, jak simulovat kameru v 2D prostoru.
- [ ] Rozumím přepočtu pozice myši na souřadnice ve světě.

- [ ] Rozumím objektově orientovanému přístupu k canvas objektům.
- [ ] Vím, proč je vhodné vytvářet třídy pro objekty ve scéně.
- [ ] Rozumím metodám update() a draw() u objektů.

- [ ] Rozumím pojmu WebGL a k čemu slouží.

- [ ] Vím, co jsou shadery (vertex shader, fragment shader).
- [ ] Rozumím jazyka GLSL na základní úrovni.
- [ ] Vím, že WebGL je nízkoúrovňové API pro 3D grafiku.

- [ ] Znám knihovnu Three.js pro práci s 3D grafikou.
- [ ] Rozumím základním konceptům Three.js (Scene, Camera, Renderer).
- [ ] Vím, co je Mesh (geometrie + materiál).
- [ ] Rozumím rozdílu mezi různými typy kamer (Perspective, Orthographic).

- [ ] Znám základní geometrie v Three.js (Box, Sphere, Plane).
- [ ] Znám základní materiály v Three.js (Basic, Standard, Lambert).
- [ ] Rozumím typům světel v Three.js (Ambient, Directional, Point).
- [ ] Vím, jak funguje Raycaster pro detekci kliknutí na objekty.

- [ ] Rozumím rozdílům mezi 2D Canvas, WebGL a DOM animacemi.
- [ ] Vím, kdy použít kterou technologii vykreslování.

A.3.1 Praktická část

- [ ] Umím vytvořit animaci ve Vue pomocí transition komponenty.
- [ ] Dokážu animovat seznam položek pomocí transition-group.

- [ ] Umím použít requestAnimationFrame pro vlastní animaci.
- [ ] Dokážu implementovat delta time v animační smyčce.
- [ ] Umím vytvořit jednoduchou interpolaci mezi hodnotami.

- [ ] Umím vytvořit canvas element a získat 2D kontext.
- [ ] Dokážu nakreslit základní tvary (obdélník, kruh, čáru).
- [ ] Umím vykreslit text a obrázek na canvas.
- [ ] Dokážu vytvořit animační smyčku pro canvas.

- [ ] Umím implementovat jednoduchý pohyb kamery v 2D.
- [ ] Dokážu přepočítat pozici myši na souřadnice ve světě.
- [ ] Umím vytvořit třídu pro objekty ve scéně s update() a draw().

- [ ] Umím vytvořit základní 3D scénu v Three.js.
- [ ] Dokážu přidat objekty, světla a kameru do scény.
- [ ] Umím animovat objekty v Three.js (rotace, pozice).
- [ ] Dokážu zpracovat responzivitu Three.js scény.

A.3.2 Bonusové části

- [ ] Znám knihovny pro interpolaci (GSAP, Anime.js) a umím je použít.
- [ ] Rozumím optimalizacím pro canvas (off-screen canvas, layering).

- [ ] Vím, co je Shared Worker a k čemu slouží.
- [ ] Rozumím rozdílu mezi Worker, Shared Worker a Service Worker.

- [ ] Umím psát jednoduché shadery v GLSL.
- [ ] Rozumím pipeline WebGL (vertex → fragment).
- [ ] Vím, jak fungují textury v WebGL/Three.js.

- [ ] Znám pokročilé koncepty Three.js (shadows, post-processing).
- [ ] Umím načíst 3D model do Three.js (GLTF, FBX).
- [ ] Rozumím optimalizaci 3D scén (LOD, frustum culling).

- [ ] Znám herní enginy pro web (Phaser, PixiJS, Babylon.js, PlayCanvas).
- [ ] Rozumím základům herní fyziky (kolize, gravitace).
- [ ] Vím, jak implementovat základní herní smyčku (game loop).

A.4 Alternativní protokoly

- [ ] Rozumím, že WebSocket staví na protokolu TCP.
- [ ] Vím, jaký je rozdíl mezi TCP a UDP.
- [ ] Rozumím, proč WebSocket umožňuje obousměrnou komunikaci.
- [ ] Vím, že WebSocket používá HTTP pro navázání spojení (Upgrade header).
- [ ] Rozumím protokolu ws:// a wss:// (zabezpečené spojení).

- [ ] Vím, jak vytvořit WebSocket spojení v JavaScriptu.
- [ ] Rozumím událostem WebSocket (open, message, close, error).
- [ ] Vím, jaké typy dat lze posílat přes WebSocket (text, binární data).
- [ ] Umím vysvětlit, proč je výhodné používat JSON formát pro zprávy.

- [ ] Vím, jak vytvořit WebSocket server v Node.js (knihovna ws).
- [ ] Rozumím pojmu „socket“ jako reprezentace jednoho připojení.
- [ ] Umím vysvětlit, proč server musí spravovat více klientů najednou.

- [ ] Znám knihovnu Socket.IO a její výhody.
- [ ] Vím, co Socket.IO řeší (automatické znovupřipojení, ping/pong, rooms).
- [ ] Rozumím pojmu „fallback“ na jiné technologie.
- [ ] Znám další knihovny pro WebSocket (SockJS, SignalR, Primus).

- [ ] Rozumím struktuře packetů pro WebSocket komunikaci.
- [ ] Vím, proč je dobré mít typ a payload u každé zprávy.
- [ ] Umím navrhnout strukturu dat pro chat aplikaci.
- [ ] Umím navrhnout strukturu dat pro online hru (např. pozice hráče).

- [ ] Rozumím vhodným účelům pro WebSocket (chat, hry, notifikace).
- [ ] Vím, že WebSocket by neměl být základ celé aplikace.
- [ ] Rozumím, proč kombinovat WebSocket s HTTP API.

- [ ] Rozumím principu Polling.
- [ ] Vím, jaké jsou nevýhody pollingu (zbytečné požadavky).

- [ ] Rozumím principu Long Polling.
- [ ] Vím, v čem je Long Polling lepší než běžný polling.
- [ ] Rozumím, že server drží spojení otevřené, dokud nemá data.

- [ ] Rozumím principu Server-Sent Events (SSE).
- [ ] Vím, že SSE je jednosměrný protokol (server → klient).
- [ ] Rozumím Content-Type: text/event-stream.
- [ ] Vím, jak SSE zprávy vypadají (data:, id:, dvě nové řádky).
- [ ] Rozumím EventSource API v JavaScriptu.

- [ ] Vím o omezení počtu současných TCP spojení v prohlížeči (6-8 na doménu).
- [ ] Rozumím problémům s long-running requesty (proxy, timeouty).
- [ ] Vím, proč polling a SSE mohou blokovat další požadavky.

- [ ] Vím, co je gRPC a že staví na HTTP/2.
- [ ] Rozumím, že gRPC používá Protocol Buffers.
- [ ] Vím, proč se gRPC na webu moc nepoužívá (omezená podpora v prohlížečích).
- [ ] Rozumím, že gRPC se hodí pro server-to-server komunikaci.

- [ ] Rozumím, že datům zvenčí nikdy nelze věřit.
- [ ] Vím o důležitosti validace a sanitizace dat.
- [ ] Rozumím potřebě omezovat velikost přijatých dat.
- [ ] Vím o důležitosti jasně definovaného API rozhraní.

A.4.1 Praktická část

- [ ] Umím vytvořit WebSocket spojení na straně klienta.
- [ ] Dokážu poslat a přijmout zprávu přes WebSocket.
- [ ] Umím zpracovat události (open, message, close, error).

- [ ] Umím vytvořit základní WebSocket server v Node.js.
- [ ] Dokážu broadcast zprávu všem připojeným klientům.
- [ ] Umím spravovat seznam připojených klientů.

- [ ] Umím navrhnout a implementovat strukturu packetů.
- [ ] Dokážu vytvořit jednoduchou chat aplikaci s WebSocket.
- [ ] Umím implementovat odesílání pozice v real-time aplikaci.

- [ ] Umím vytvořit SSE endpoint na serveru.
- [ ] Dokážu použít EventSource API na klientovi.
- [ ] Umím implementovat posílání dat ze serveru pomocí SSE.

- [ ] Dokážu implementovat validaci dat na serveru.
- [ ] Umím ošetřit chybové stavy při komunikaci.

A.4.2 Bonusové části

- [ ] Umím použít Socket.IO pro pokročilou WebSocket komunikaci.
- [ ] Rozumím rooms a namespaces v Socket.IO.
- [ ] Vím, jak implementovat autentizaci u WebSocket spojení.

- [ ] Rozumím rozdílu mezi unicast, broadcast a multicast.
- [ ] Vím, jak implementovat soukromé zprávy (1-on-1) přes WebSocket.
- [ ] Umím vytvořit systém „rooms“ pro skupinovou komunikaci.

- [ ] Rozumím strategiím pro reconnect (exponential backoff).
- [ ] Vím, jak řešit message queuing při výpadku spojení.
- [ ] Rozumím heartbeat/ping-pong mechanismu pro detekci živých spojení.

- [ ] Vím o bezpečnostních rizicích WebSocket (CSRF, XSS).
- [ ] Rozumím důležitosti použití wss:// (zabezpečené spojení).
- [ ] Vím, jak implementovat rate limiting pro WebSocket.

- [ ] Rozumím Protocol Buffers na základní úrovni.
- [ ] Vím, v jakých scénářích má smysl použít gRPC.

A.5 Webové vyhledávače a optimalizace

- [ ] Rozumím, proč je vyhledávání na webu složitý problém.
- [ ] Vím, co jsou crawlery (pavouci, boti) a jak fungují.
- [ ] Rozumím pojmu „index“ vyhledávače.
- [ ] Vím, že vyhledávače neustále procházejí a aktualizují stránky.

- [ ] Rozumím, jak crawlery získávají nové URL (odkazy, registr domén, sitemap).
- [ ] Vím, proč jsou populární stránky navštěvovány častěji.
- [ ] Rozumím důležitosti sémantiky HTML pro vyhledávače.

- [ ] Vím, jak vyhledávače získávají data ze stránek (parsing HTML).
- [ ] Rozumím pojmům title, meta description, keywords.
- [ ] Vím, proč je důležité mít obsah viditelný (ne skrytý v CSS).

- [ ] Rozumím pojmu stemming a lemmatizace.
- [ ] Vím, proč je důležité normalizovat slova (různé tvary).
- [ ] Umím vysvětlit rozdíl mezi absolutní a relativní četností slov.

- [ ] Rozumím Booleovskému modelu vyhledávání.
- [ ] Vím, jak fungují operátory AND, OR, NOT.
- [ ] Rozumím pojmu inverzní index.
- [ ] Vím, jaké jsou výhody a nevýhody Booleovského modelu.

- [ ] Rozumím TF-IDF (Term Frequency-Inverse Document Frequency).
- [ ] Vím, co měří TF (jak často slovo na stránce).
- [ ] Vím, co měří IDF (jak vzácné je slovo napříč stránkami).
- [ ] Rozumím rozšířenému Booleovskému modelu s TF-IDF.

- [ ] Rozumím vektorovému modelu vyhledávání.
- [ ] Vím, jak se stránky reprezentují jako vektory.
- [ ] Rozumím kosínové podobnosti.
- [ ] Vím, proč je kosínová podobnost lepší než Euklidovská vzdálenost.

- [ ] Znám další modely (LSI, pravděpodobnostní modely, AI modely).

- [ ] Rozumím algoritmu PageRank a jeho principu.
- [ ] Vím, že důležitost stránky závisí na důležitosti odkazujících stránek.
- [ ] Rozumím pojmu damping factor.
- [ ] Vím, že PageRank není jediný faktor hodnocení dnes.

- [ ] Znám další faktory hodnocení (rychlost, mobilní přívětivost, čerstvost).
- [ ] Vím o důležitosti uživatelského chování (bounce rate, time on site).

- [ ] Rozumím významu zpětných odkazů (backlinks).

- [ ] Vím, co je farma odkazů (link farm) a proč je špatná.
- [ ] Rozumím pojmu skrytý text (hidden text) a jeho penalizaci.
- [ ] Vím o problému cloakingu (jiný obsah pro crawler vs. uživatele).
- [ ] Rozumím, proč jsou podvodné techniky penalizovány.

- [ ] Rozumím základům SEO (Search Engine Optimization).
- [ ] Vím, že SEO je o pomoci vyhledávačům, ne podvádění.
- [ ] Rozumím důležitosti kvalitního obsahu.
- [ ] Vím o důležitosti relevantních klíčových slov.

- [ ] Rozumím významu meta tagů (title, description).
- [ ] Vím o důležitosti struktury URL.
- [ ] Rozumím významu interních a externích odkazů.
- [ ] Vím o důležitosti rychlosti a responzivity stránky.

- [ ] Rozumím pojmu sitemap a jeho účelu.
- [ ] Vím, jak vypadá sitemap.xml soubor.
- [ ] Rozumím pojmům changefreq, priority v sitemap.
- [ ] Vím o sitemap indexu pro velké weby.

- [ ] Rozumím souboru robots.txt a jeho účelu.
- [ ] Vím, jak v robots.txt povolit/zakázat přístup.
- [ ] Znám User-Agent direktivu.
- [ ] Vím, jak v robots.txt odkázat na sitemap.

- [ ] Rozumím strukturovaným datům (structured data).
- [ ] Vím, co je Schema.org a JSON-LD.
- [ ] Rozumím Open Graph meta tagům pro sociální sítě.
- [ ] Vím o důležitosti používání synonym a variant slov.

- [ ] Rozumím problému SPA aplikací s SEO.
- [ ] Vím, že některé vyhledávače neumí spustit JavaScript.
- [ ] Rozumím řešením: Server-Side Rendering (SSR) a Pre-rendering.

- [ ] Vím, co je Google Search Console.
- [ ] Rozumím, že v Search Console vidíme výkon stránky.
- [ ] Vím, že můžeme požádat o znovuprocházení stránky.

A.5.1 Praktická část

- [ ] Umím vytvořit správné meta tagy (title, description, keywords).

- [ ] Dokážu strukturovat URL srozumitelně a s klíčovými slovy.
- [ ] Umím používat sémantické HTML tagy správně.

- [ ] Umím vytvořit sitemap.xml soubor.
- [ ] Dokážu vytvořit sitemap index pro větší weby.
- [ ] Umím správně strukturovat sitemap (loc, lastmod, changefreq, priority).

- [ ] Umím vytvořit robots.txt soubor.
- [ ] Dokážu povolit/zakázat přístup crawlerům.
- [ ] Umím v robots.txt odkázat na sitemap.

- [ ] Umím přidat strukturovaná data pomocí JSON-LD.
- [ ] Dokážu implementovat Open Graph meta tagy.
- [ ] Umím použít Schema.org pro různé typy obsahu (Article, Product, Event).

- [ ] Dokážu optimalizovat rychlost načítání stránky.
- [ ] Umím zajistit mobilní přívětivost (responsive design).
- [ ] Dokážu optimalizovat obrázky pro web.

- [ ] Umím implementovat interní odkazy logicky.
- [ ] Dokážu vytvořit breadcrumbs navigaci.

A.5.2 Bonusové části

- [ ] Rozumím algoritmu na hlubší úrovni (jak funguje iterace PageRank).
- [ ] Vím o problémech s cykly v grafu odkazů.
- [ ] Rozumím maticové reprezentaci PageRank.

- [ ] Umím implementovat jednoduchý Booleovský vyhledávač.
- [ ] Dokážu implementovat TF-IDF výpočet.
- [ ] Umím implementovat kosínovou podobnost.

- [ ] Znám pokročilé SEO techniky (canonical URLs, hreflang).
- [ ] Rozumím Core Web Vitals (LCP, FID, CLS).
- [ ] Vím o důležitosti HTTPS pro SEO.
- [ ] Rozumím konceptu E-A-T (Expertise, Authoritativeness, Trustworthiness).

- [ ] Umím implementovat SSR pro Vue/React aplikaci.
- [ ] Dokážu nastavit pre-rendering pro SPA.
- [ ] Vím, jak testovat, jak vyhledávače vidí moji stránku.

- [ ] Znám nástroje pro SEO analýzu (Lighthouse, PageSpeed Insights).
- [ ] Umím používat Google Search Console efektivně.

- [ ] Rozumím Google Analytics pro měření návštěvnosti.

- [ ] Vím o lokálním SEO (Google My Business).
- [ ] Rozumím důležitosti backlink profilu.
- [ ] Znám strategie pro získávání kvalitních zpětných odkazů.

A.6 Aplikace tvořené webovými technologiemi

- [ ] Rozumím, proč jsou webové aplikace populární.
- [ ] Vím o výhodách webových aplikací (bez instalace, dostupnost, aktualizace).
- [ ] Rozumím nevýhodám webových aplikací (omezený přístup k HW, výkon).
- [ ] Vím, proč uživatelé preferují nativní aplikace (snadná dostupnost na zařízení).

- [ ] Rozumím pojmu Webové API.
- [ ] Vím, že Webové API poskytují přístup k funkcím zařízení.
- [ ] Rozumím, že prohlížeč kontroluje bezpečnost přístupu k API.
- [ ] Vím o stavech povolení API (Granted, Denied, Prompt).

- [ ] Znám běžná Webová API (Geolocation, Camera, Notifications, Storage).
- [ ] Vím o WebRTC, Service Workers, Push API, File API.
- [ ] Rozumím, že ne všechny API jsou dostupné ve všech prohlížečích.
- [ ] Vím o bezpečnostních omezeních API (nutnost HTTPS, user interaction).

- [ ] Rozumím Notifications API.
- [ ] Vím, jak požádat o povolení k notifikacím.
- [ ] Umím vysvětlit strukturu notifikace (title, body, icon, actions).
- [ ] Rozumím rozdílu mezi notifikacemi na aktivní vs. zavřené stránce.

- [ ] Rozumím pojmu Progresivní webová aplikace (PWA).
- [ ] Vím, že PWA lze nainstalovat na zařízení.
- [ ] Rozumím, že PWA může fungovat offline.
- [ ] Vím, že PWA využívá webová API jako Service Workers.

- [ ] Vím, že PWA je závislé na podpoře prohlížeče (zejména Chromium).
- [ ] Rozumím, že PWA interně používá cache pro offline režim.
- [ ] Vím, že PWA může běžet v samostatném okně bez UI prohlížeče.

- [ ] Vím o požadavcích pro PWA (HTTPS, manifest, Service Worker).
- [ ] Rozumím události beforeinstallprompt.
- [ ] Vím, jak nabídnout vlastní instalační dialog.

- [ ] Rozumím souboru manifest.json a jeho účelu.
- [ ] Vím o klíčových polích manifestu (name, icons, start_url, display).
- [ ] Rozumím režimům zobrazení (standalone, fullscreen, minimal-ui, browser).
- [ ] Vím, jak odkazovat na manifest v HTML (<link rel=„manifest“>).

- [ ] Rozumím pojmu Service Worker.

- [ ] Vím, že Service Worker běží na pozadí i po zavření stránky.
- [ ] Rozumím životnímu cyklu Service Worker (install, activate, fetch).
- [ ] Vím, že Service Worker nemá přístup k DOM.

- [ ] Rozumím registraci Service Workeru (navigator.serviceWorker.register).
- [ ] Vím o událostech Service Workeru (install, activate, fetch, push).
- [ ] Rozumím práci s cache v Service Workeru.
- [ ] Vím, jak implementovat offline režim pomocí cache.

- [ ] Rozumím Push API.
- [ ] Vím, že Push API umožňuje přijímat notifikace i když stránka není otevřená.
- [ ] Rozumím pojmu VAPID klíče (veřejný a soukromý).
- [ ] Vím o struktuře push subscription (endpoint, klíče).

- [ ] Rozumím, jak Push API funguje interně (server prohlížeče, dlouhodobé spojení).
- [ ] Vím, že push zprávy přicházejí do Service Workeru.
- [ ] Rozumím zobrazení notifikace z Service Workeru (self.registration.showNotification).

- [ ] Vím, že PWA není velmi populární (omezená podpora, specifické použití).
- [ ] Rozumím, že PWA přineslo důležitá API (Service Workers, Push).

- [ ] Rozumím principu WebView v nativních aplikacích.
- [ ] Vím, že WebView je zabudovaný prohlížeč v aplikaci.
- [ ] Rozumím, že nativní aplikace vytváří API pro komunikaci s webovou stránkou.

- [ ] Znám framework Capacitor pro tvorbu hybridních aplikací.
- [ ] Vím, že Capacitor umožňuje export na iOS, Android a web (PWA).
- [ ] Rozumím, že Capacitor vytváří nativní projekty, které je nutné sestavit.

- [ ] Vím o požadavcích pro použití Capacitoru (package.json, index.html, build folder).
- [ ] Rozumím souboru capacitor.config (appId, appName, webDir).
- [ ] Vím o příkazech: cap init, cap add, cap sync, cap open.

- [ ] Rozumím pluginům v Capacitoru.
- [ ] Znám běžné pluginy (Filesystem, Camera, Clipboard, Dialog).
- [ ] Vím, že pluginy vyžadují pravomoci v nativním projektu (AndroidManifest.xml).

- [ ] Znám alternativy k Capacitoru (Cordova, React Native, Flutter).

- [ ] Rozumím frameworku Electron pro desktopové aplikace.
- [ ] Vím, že Electron obsahuje celý Chromium prohlížeč.
- [ ] Rozumím problémům Electronu (velikost, paměť, výkon).

- [ ] Rozumím procesům v Electronu (main process, renderer process).
- [ ] Vím, že main process spravuje okna a komunikaci s OS.
- [ ] Vím, že renderer process vykresluje webovou stránku.
- [ ] Rozumím pojmu preload script a jeho účelu.

- [ ] Rozumím BrowserWindow v Electronu.
- [ ] Vím o webPreferences (preload, contextIsolation).
- [ ] Rozumím důležitosti contextBridge pro bezpečnost.

- [ ] Rozumím IPC (Inter-Process Communication) v Electronu.
- [ ] Vím, jak posílat zprávy mezi main a renderer procesem.
- [ ] Rozumím ipcRenderer.send() a ipcMain.on().

- [ ] Vím o nástroji Electron Forge pro sestavení aplikace.
- [ ] Znám příkazy: electron-forge start, package, make, publish.

- [ ] Vím o budoucnosti webových aplikací (WebAssembly, nová API).
- [ ] Rozumím obavám o bezpečnost a soukromí na webu.

A.6.1 Praktická část

- [ ] Umím požádat o povolení k Webovému API.
- [ ] Dokážu zobrazit notifikaci pomocí Notifications API.
- [ ] Umím zpracovat kliknutí na notifikaci.

- [ ] Umím vytvořit manifest.json soubor pro PWA.
- [ ] Dokážu přidat všechna potřebná pole do manifestu.
- [ ] Umím odkazovat na manifest v HTML.

- [ ] Umím zaregistrovat Service Worker.
- [ ] Dokážu implementovat základní cache strategii.
- [ ] Umím implementovat offline režim pomocí Service Workeru.
- [ ] Dokážu zpracovat událost fetch v Service Workeru.

- [ ] Umím vygenerovat VAPID klíče.
- [ ] Dokážu zaregistrovat push subscription na klientovi.
- [ ] Umím poslat push notifikaci ze serveru (knihovna web-push).
- [ ] Dokážu zpracovat push událost v Service Workeru.

- [ ] Umím inicializovat Capacitor v projektu.
- [ ] Dokážu přidat platformu (Android, iOS).
- [ ] Umím synchronizovat a sestavit projekt.
- [ ] Dokážu použít plugin v Capacitoru (např. Filesystem).

- [ ] Umím přidat potřebné pravomoci v AndroidManifest.xml.

- [ ] Umím vytvořit základní Electron aplikaci.
- [ ] Dokážu vytvořit okno pomocí BrowserWindow.
- [ ] Umím vytvořit preload script s contextBridge.
- [ ] Dokážu implementovat IPC komunikaci mezi procesy.
- [ ] Umím sestavit Electron aplikaci pomocí Electron Forge.

A.6.2 Bonusové části

- [ ] Umím implementovat vlastní instalační prompt pro PWA.
- [ ] Rozumím strategiím cache (Cache First, Network First, Stale While Revalidate).
- [ ] Dokážu implementovat synchronizaci na pozadí (Background Sync API).

- [ ] Vím o WebAssembly a jeho použití.
- [ ] Rozumím výhodám a nevýhodám WebAssembly.

- [ ] Umím vytvořit vlastní Capacitor plugin.
- [ ] Rozumím bridge mezi webovou a nativní částí v Capacitoru.

- [ ] Umím nastavit automatické aktualizace v Electronu.
- [ ] Rozumím podepsání Electron aplikace (code signing).
- [ ] Vím o distribuci Electron aplikací (instalátory pro různé OS).
- [ ] Dokážu optimalizovat velikost Electron aplikace.

- [ ] Znám Tauri jako alternativu k Electronu.
- [ ] Rozumím výhodám Tauri (menší velikost, bezpečnost).

A.7 Webové frameworky

- [ ] Rozumím pojmu „framework“ v kontextu webového vývoje.
- [ ] Vím o různých typech frameworků (backend, frontend, CSS).
- [ ] Rozumím rozdílu mezi knihovnou a frameworkem.
- [ ] Vím, že frontendové frameworky pomáhají vytvářet aplikace v prohlížeči.

- [ ] Rozumím pojmu reaktivní systém.
- [ ] Vím, že frameworky sledují změny dat a aktualizují UI.
- [ ] Rozumím komponentovému přístupu (každá část UI je komponenta).

- [ ] Rozumím pojmu Virtuální DOM (VDOM).
- [ ] Vím, proč změny v DOM jsou drahé operace.
- [ ] Rozumím, že VDOM minimalizuje počet změn v reálném DOM.
- [ ] Vím o procesu diffing (porovnání starého a nového VDOM).
- [ ] Vím o výhodách VDOM (optimalizace změn).
- [ ] Rozumím nevýhodám VDOM (režie navíc).
- [ ] Vím, že ne všechny frameworky používají VDOM.

- [ ] Znám framework Vue.
- [ ] Vím, že Vue používá VDOM.
- [ ] Rozumím Single File Components (.vue soubory).
- [ ] Vím o Vue direktivách (v-if, v-for, v-model, v-bind, v-on).
- [ ] Rozumím reaktivitě ve Vue (ref, reactive, computed).

- [ ] Znám framework React.
- [ ] Vím, že React používá VDOM.
- [ ] Rozumím JSX syntaxi.
- [ ] Vím, že komponenty v Reactu jsou funkce.
- [ ] Rozumím React Hooks (useState, useEffect, useCallback).
- [ ] Rozumím předávání dat pomocí props v Reactu.
- [ ] Vím, jak oznámit změnu rodičovské komponentě v Reactu.
- [ ] Znám ekosystém React (React Router, Redux).

- [ ] Znám framework Svelte.
- [ ] Vím, že Svelte je kompilátor, ne runtime framework.
- [ ] Rozumím, že Svelte nepoužívá VDOM.
- [ ] Vím, že Svelte generuje optimalizovaný kód přímo manipulující DOM.
- [ ] Rozumím reaktivitě ve Svelte ($state, $derived).
- [ ] Vím, jak fungují props ve Svelte ($props).
- [ ] Znám SvelteKit jako nástroj pro Svelte.

- [ ] Znám framework Ember.
- [ ] Vím, že Ember se zakládá na konvencích před konfigurací.

- [ ] Rozumím, že Ember je vhodný pro velké projekty a týmy.
- [ ] Vím o vysoké křivce učení Emberu.

- [ ] Znám framework Angular.
- [ ] Vím, že Angular je komplexní framework od Googlu.
- [ ] Rozumím konceptům: Two-way Data Binding, Directives.
- [ ] Vím, že Angular má dnes menší popularitu než dříve.

- [ ] Rozumím problému SPA aplikací s SEO.
- [ ] Vím, že vyhledávače mohou mít problém s indexací SPA.
- [ ] Vím, co je Server-Side Rendering (SSR).
- [ ] Vím, co je pre-rendering.
- [ ] Rozumím, že při SSR se HTML generuje na serveru.
- [ ] Vím o výhodách SSR (lepší SEO, rychlejší první zobrazení).

- [ ] Znám nástroj Next.js pro React s SSR.
- [ ] Znám nástroj Nuxt.js pro Vue s SSR.
- [ ] Znám nástroj SvelteKit pro Svelte s SSR.

- [ ] Rozumím Static Site Generation (SSG).
- [ ] Vím, že SSG generuje statické HTML stránky předem.
- [ ] Rozumím rozdílu mezi SSR a SSG.
- [ ] Rozumím historickému cyklu webového vývoje (server → client → server).
- [ ] Vím, proč se trendy opakují (náklady, výkon, SEO).

A.7.1 Praktická část

- [ ] Umím vytvořit základní Vue komponentu.
- [ ] Dokážu používat Vue direktivy (v-if, v-for, v-model).
- [ ] Umím předávat data pomocí props a emits ve Vue.
- [ ] Dokážu používat Vue Composition API (ref, reactive, computed).

- [ ] Umím vytvořit základní React komponentu.
- [ ] Dokážu používat JSX syntaxi.
- [ ] Umím používat useState pro správu stavu.
- [ ] Dokážu předávat data pomocí props v Reactu.
- [ ] Umím používat useEffect pro side effects.

- [ ] Umím vytvořit základní Svelte komponentu.
- [ ] Dokážu používat reaktivitu ve Svelte ($state).
- [ ] Umím předávat data pomocí props ve Svelte.
- [ ] Dokážu vytvořit jednoduchou aplikaci ve Svelte.

- [ ] Dokážu porovnat stejnou aplikaci ve Vue, React a Svelte.

- [ ] Umím vybrat vhodný framework pro konkrétní projekt.

- [ ] Umím nastavit Vue projekt s Vite.
- [ ] Umím nastavit React projekt s Vite.
- [ ] Umím nastavit Svelte projekt s Vite.

A.7.2 Bonusové části

- [ ] Rozumím pokročilým Vue konceptům (Teleport, Suspense, provide/inject).

- [ ] Rozumím pokročilým React Hooks (useMemo, useCallback, useRef).
- [ ] Vím o React Context API pro sdílení dat.
- [ ] Znám Redux pro komplexní správu stavu.
- [ ] Rozumím React Router pro routing.

- [ ] Rozumím Svelte Stores pro správu stavu.
- [ ] Vím o Svelte transitions a animations.
- [ ] Znám SvelteKit routing a data loading.

- [ ] Umím nastavit Next.js projekt.
- [ ] Rozumím Next.js routingu (App Router, Pages Router).
- [ ] Vím o getServerSideProps a getStaticProps.
- [ ] Umím vytvořit API routes v Next.js.

- [ ] Umím nastavit Nuxt.js projekt.
- [ ] Rozumím Nuxt routingu a layouts.
- [ ] Vím o asyncData a fetch v Nuxtu.

- [ ] Umím nastavit SvelteKit projekt.
- [ ] Rozumím SvelteKit load functions.
- [ ] Vím o form actions v SvelteKitu.

- [ ] Rozumím hydrataci (hydration) v SSR aplikacích.
- [ ] Znám další frameworky (Solid.js, Qwik, Alpine.js).
- [ ] Rozumím meta-frameworkům (Astro, Remix).
- [ ] Vím o Islands Architecture.
- [ ] Umím optimalizovat bundle size (code splitting, lazy loading).

Příloha B.

Seznamy

B.1 Seznam kódových bloků