Pedagogická fakulta

Univerzita Palackého v Olomouci

Katedra technické a informační výchovy

Bakalářská práce

Využití IT ke generování zadání písemných testů

Šimon Janča

Vedoucí práce: doc. RNDr. Petr Šaloun, Ph.D.

Olomouc 2024

Anotace / Annotation

Práce zdůrazňuje využití ***IT*** ve vzdělávání a zaměřuje se na jejich využití ke generování písemných testů. Zkoumá existující metody tvorby testů a jejich náležitosti a představuje novou webovou aplikaci, která umožňuje učitelům efektivně vytvářet testy z úloh s parametricky zadanými vlastnostmi. Aplikace podporuje uzavřené i otevřené úlohy a bude integrovatelná s jinými systémy pomocí *API*.

The thesis emphasizes the significance of technology in the educational process and contributes to modernizing the process of preparing exams. Introduces the web application that allows users to generate exams from prepared questions with parametrically defined properties. The application will support both questions with options and questions with long answers and will be integrable into other systems using *API*.

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně s využitím uvedených zdrojů.

Pro ucelenější pochopení kontextu problematiky a rozšíření znalostí jsem v rámci práce konzultoval některé otázky s umělou inteligencí (ChatGPT, Gemini). Texty v práci jsou výsledkem mé samostatné práce.

Šimon Janča, v Olomouci 14. června 2024

Poděkování

Chtěl bych touto cestou poděkovat všem, kteří mi byli nápomocni při návrhu a korektuře práce nebo upřesnění návrhu aplikace. Děkuji také rodině a blízkým za vlídné slovo a povzbuzení.

Veliký dík patří vedoucímu práce doc. RNDr. Petru Šalounovi, Ph.D. za trpělivost a ochotu pomoci při její tvorbě.

Děkuji Janu Sklepkovi za zhotovení loga aplikace za sponzorovanou cenu.

Děkuji také Pedagogické fakultě Univerzity Palackého v Olomouci za příspěvek na pilotní provoz aplikace ve formě stipendia.

Obsah

1 Úvod **3**

2 Testy, testové úlohy **4**
2.1 Uzavřené úlohy . . . 4
2.2 Otevřené úlohy . . . 4
2.3 Tvorba testů . . . 4
2.4 Nástroje k tvorbě testů . . . 5

3 Technologie a principy vývoje webových aplikací **6**
3.1 Základní statická HTML stránka . . . 6
3.2 Markdown . . . 6
3.3 HTML s dynamickým obsahem . . . 7
3.4 Single Page Application . . . 7
3.5 Kompilátor a interpret . . . 7
3.6 Kompilátor . . . 8
3.7 Web Assembly . . . 9
3.8 Procedurální a objektově orientované programování . . . 9
3.9 Javascript . . . 10
3.10 Typescript . . . 12
3.11 Document Object Model . . . 12
3.12 React . . . 12
3.13 Javascript XML . . . 13
3.14 Server Side Rendering a Static Site Generation . . . 14
3.15 Client-Server . . . 14
3.16 Webová služba . . . 15
3.17 Webový server . . . 15
3.18 Domain Name Server . . . 15
3.19 Model View Controller . . . 16
3.20 Frontend, Backend . . . 16
3.21 Javascript Object Notation . . . 17
3.22 Asynchronous Javascript Request . . . 17
3.23 Application Interface . . . 17
3.24 Representational state transfer . . . 17
3.25 GraphQL . . . 18
3.26 Zabezpečení . . . 18
3.26.1 Hashování . . . 18
3.26.2 Šifrování . . . 19
3.26.3 Autentizace . . . 19
3.26.4 Autorizace . . . 19
3.26.5 JSON Web Token . . . 19
3.26.6 Structured Query Language . . . 20
3.26.7 SQL Injection . . . 20
3.26.8 Cross Site Scripting . . . 21
3.27 42 . . . 21
3.28 Testování . . . 22
3.28.1 Testování API . . . 22
3.28.2 Testování frontendu . . . 22
3.29 Continous Integration a Continuous Delivery . . . 22
3.30 Verzovací systém GIT . . . 23
3.30.1 Git hooks . . . 23

3.30.2 Gitlab Actions . . . 23
3.31 Pravidla přístupnosti webu . . . 24
3.32 Optimalizace pro vyhledávače . . . 24
3.33 Údržba aplikace . . . 24
3.33.1 Životní cyklus vývoje software . . . 24
3.33.2 Sledování chyb nahlášených uživateli . . . 24
3.33.3 Běhové a kompilační chyby . . . 25
3.33.4 Sentry . . . 25
3.34 Linux a příkazová řádka, Windows Subsystem for Linux . . . 25
3.35 Integrated Development Environment a nástroje pro psaní kódu . . . 26
3.36 Mezinárodní lokalizace . . . 26
3.37 Unified Modeling Language . . . 27
3.37.1 Use Case Diagram . . . 28
3.37.2 Class Diagram . . . 29

4 Návrh aplikace **31**
4.1 Návrh fungování aplikace . . . 31
4.2 Návrh struktury databáze . . . 32
4.3 Výběr technologií . . . 33

5 Postup implementace **37**
5.1 Backend . . . 37
5.1.1 Funkce Init . . . 38
5.1.2 Router . . . 38
5.1.3 Generátor . . . 39
5.1.4 Autentizace uživatelů . . . 41
5.2 Frontend . . . 42
5.3 Testování . . . 43
5.4 Zálohy . . . 43

6 Závěr **44**

Seznam použitých zkratek **45**

Použitá literatura **48**

Kapitola 1

Úvod

Ověření výsledků výuky je důležitou součástí vzdělávání. Školní testy jsou nástrojem ke zhodnocení nabytých znalostí a dovedností a také důležitou součástí reflexe výuky, díky které učitelé vědí, na kterou látku se zaměřit při navazující výuce. Vznikají tak, že učitelé nebo vzdělávací instituce vytvářejí otázky a úkoly (problémy, podle [107]), které jsou zaměřeny na konkrétní téma nebo oblast učiva. Tyto *otázky* a úkoly jsou poté zařazeny do testu, který je žákům či studentům předložen k vyplnění.

Testy umožňují učitelům zjistit, jak dobře žáci porozuměli učivu a jaké jsou jejich silné a slabé stránky. Žákům poskytnou informace o tom, které znalosti je potřeba ještě zlepšovat. Součástí kvalitního zjišťování výsledků je tvorba více verzí zadání, např. pro spolužáky, sedící v jedné lavici ap. Kromě toho jsou výsledky různých testů použity jako parametr přijímání na vysoké školy nebo k udělování stipendií. Testy jsou využívány v mnoha případech i mimo školy k zajištění zpětné vazby nebo ověření znalostí kandidáta při pracovních pohovorech.

V této práci popíšu náležitosti testů a některé postupy, jakými je možné postupovat při tvorbě jejich zadání a definuji potřeby, které v této souvislosti mohou nastat. Zabývat se budu především písemnými testy, které jsou používány na základních a středních školách, nicméně většina z popisovaných pojmů bude použitelná i pro další typy testů. Testy vědomostí nejsou pouze záležitostí vzdělávání, využívají je firmy a další instituce a jsou důležitým prvkem při výběru zaměstnanců nebo při přijímacích zkouškách nebo v autoškolách. Od doby pandemie Covidu 19 se začaly používat i v online prostředí.

Ve druhé části popíšu a vytvořím aplikaci, kterou bude možné použít ke generování různých variant zadání školních testů. Popíšu základní prvky při vývoji aplikace a popíšu důvody k jednotlivým rozhodnutím od návrhu a výběru technologií, přes plánování, až k její implementaci, provozu a údržbě za použití vybraných technologií. Vyhotovená aplikace bude následně veřejně dostupná jako webová služba.

Kapitola 2

Testy, testové úlohy

V této kapitole se budu zabývat náležitostmi testových úloh a jaké možnosti nabízejí moderní technologie ke zjednodušení jejich tvorby. Zaměřuji se na nástroje, které jsou dostupné bezplatně nebo jsou dokonce dostupné s **otevřeným zdrojovým kódem** (**open–source**) [26].

2.1 Uzavřené úlohy

Uzavřené úlohy jsou úlohy, které mají předem danou množinu odpovědí. Respondent si při vyplňování testu vybere jednu nebo více správných odpovědí (podle zadání) a tím odpoví na otázku. Může jít o označení vyjmenovaných možností nebo z obecně předpokládaných odpovědí: např. na otázku „V jakém městě bydlíte?“ očekáváme v odpovědi existující město, nikoliv několik popisných vět. Hodnocení probíhá pomocí bodování.

Řešení uzavřených úloh může být oproti jiným typům méně náročné jak pro žáka, který může jednodušeji určit (nebo uhodnout) správnou odpověď, tak pro učitele, který nemusí při hodnocení brát v potaz různé možnosti odpovědí. Může jít také o doplnění písmene výběrem z možností za účelem procvičení správné gramatiky. Také je možné využít různé doplňovačky nebo výběr správného slova do textu, např. v cizích jazycích. V matematice může jít o výběr správné hodnoty (nebo množinu hodnot).

U uzavřených testů, obsahujících možnosti výběru, lze využít možnost zápisu do odpovědních archů a jejich vyhodnocení pomocí počítače. Bodování úloh může být jednoduché, kdy za každou správnou odpověď přičteme jeden bod, nebo může být složitější, kdy např. za každou správnou odpověď přičteme jeden bod, ale za každou špatnou odpověď bod (nebo část bodu) odečteme, za nevyplněnou odpověď většinou nepřičítáme ani neodečítáme žádný bod.

2.2 Otevřené úlohy

Otevřené úlohy naopak předem dané možnosti odpovědí nemají. Respondent musí odpověď napsat samostatně, odpovědí může být slovo, věta, nebo i celý text. Může jít o návodné *otázky*, kde je potřeba odpovědět vlastními slovy, nebo o *otázky*, kde je potřeba vypočítat nějakou hodnotu. Je možné takto zhodnotit odpověď nejen podle výsledné odpovědi, ale i podle postupů, které byly použity.

Řešení takových úloh je náročnější, jak pro žáka, který musí odpověď vymyslet, tak pro učitele, který musí při hodnocení brát v potaz různé možnosti odpovědí. Na druhou stranu to umožní ocenit i částečně správné odpovědi nebo vlastní invenci při řešení, což by u uzavřených úloh nebylo možné [41].

2.3 Tvorba testů

Na začátku tvorby testu je nutné uvažovat o jeho rozsahu. Jde o obecný test, ověřující schopnosti žáka, nebo pouze konkrétního výřezu učiva daného předmětu? Zároveň je ale potřeba při tvorbě myslet na žáky.

„Otázky by měly být pestré, pokrývající co nejširší spektrum prověřovaného učiva, a to z různých úhlů pohledu. Proto je nutné použít otázky, které zjišťují schopnost žáka pracovat s fakty (značky a jednotky fyzikálních veličin, matematické vyjádření fyzikálních zákonů atd.)“ [83].

Zároveň je potřeba myslet na to, že žáci mají různé schopnosti a znalosti a ne pro všechny je test stejně náročný. Proto je potřeba k dosažení spolehlivého ověření znalostí kombinovat testy s jinými metodami ověřování znalostí a dávat důraz na získané schopnosti a jejich aplikaci, nikoliv na to, co si žák dokázal doslovně zapamatovat. Důležitá je i práce s chybou a individuální přístup k žákům [10, 54].

Důležitý aspekt je tvorba správných a nesprávných odpovědí při tvorbě (*uzavřených*) otázek s výběrem z nabídnutých možností. Pokud je cílem dobře ověřit znalosti, pak je nutné tvořit *otázky* tak, aby nesprávné odpovědi vypadaly věrohodně. Také nesmíme zapomenout na to, aby odpovědi obsahovaly jedno či více správných řešení (podle povahy dané úlohy).

V příkladu se zaměřím na matematiku, zejména proto, že jde o jeden z mých studovaných oborů. Pokud je potřeba otestovat např. znalosti *lineárních rovnic*, bude pro zjištění, zda je testovaný schopný odpověď vypočítat a jak při tom postupuje, pravděpodobně použita *otevřená otázka*. Může ale nastat situace, že je potřeba ověřit rychlý úsudek testovaného na základě určité části učiva, a v tom případě jsou mu nabídnuty možnosti, ze kterých správné řešení vybíra. V tomto případě je také potřeba, aby nesprávné možnosti vypadaly věrohodně a aby testovaný nemohl jednoduše odhadnout správnou odpověď ale k odpovědi využil znalosti tématu.

Možnostmi pro odpovědi k řešení lineární rovnice budou výsledné hodnoty proměnné (např. x). Různé možnosti je možné definovat výpočtem z koeficientů. Nebo bude potřeba, aby výsledek odpovídal některému z číselných oborů. Pro žáky, kteří nemají znalost počítání se zlomky, je potřeba vytvořit rovnici s celočíselným výsledkem. Naopak pokud je potřeba ověřit znalost postupu pro výpočet kořenů kvadratické rovnice s komplexními čísly, budeme chtít v rovnici a v odpovědích obsáhnout i imaginární složku [107].

2.4 Nástroje k tvorbě testů

V současnosti je pro účely tvorby testů používáno několik nástrojů, které se liší svými možnostmi a způsobem použití.

Na internetu je možné nalézt připravené sady otázek ke stažení, které učitel jen vytiskne. Další možností jsou jednotlivé příklady, ze kterých učitel test sestaví, např. pomocí textového procesoru. Při tvorbě testů mohou učitelé také spolupracovat mezi sebou a otázky sdílet, osobně či přes internet např. pomocí sociálních sítí, fór, atp.

Pokud je potřeba procvičovat test k přijímacím zkouškám, materiál je možné nalézt na oficiálních webech společností *CERMAT* nebo *SCIO*. Existují také různě obsáhlé sbírky úloh a testů, které jsou vydávány jako samostatné knihy pro různé obory a obtížnost učiva.

Některé učebnice, které jsou používány ve školách, mohou v rámci verze pro učitele obsahovat didaktické přílohy, ve kterých jsou testy k použití připravené. Je možné použít i některá cvičení z pracovních listů.

Pokud si učitel chce test vytvořit svépomocí, může používat klasické *tabulkové procesory*, jako je např. *Microsoft Excel*, *LibreOffice Calc* nebo *Google Sheets*, anebo *textové procesory*, jako je např. *Microsoft Word*, *LibreOffice Writer* nebo *Google Docs*. Grafické zadání je potom možné vložit do těchto souborů ve formě obrázků.

V současnosti se přímo nabízí do tvorby otázek nebo i celých testů zapojit **umělou inteligenci** (***AI***). Pokud je tedy *ChatGPT*, *Google Gemini* (dříve **Bard**) nebo jinému generativnímu jazykovému modelu zadáno vytvoření testových otázek na konkrétní téma, ochotně je připraví. Nicméně pokud uživatel neumí správně vytvářet dotazy (**prompty** [23]), takto připravené otázky mohou být nepřesné a mohou obsahovat výrazné chyby. Proto se nebudu touto možností v této práci dále zabývat.

Je také zajímavé (nejen pro učitele) proces testování otočit a nechat děti úlohy vytvořit. To je přímo zapojí do tvorby testů a umožní jim učivo lépe pochopit a motivovat je ke zvládnutí učiva [39].

Je možné otázky zadat např. do systému *Moodle* a využít jeho možností. *Moodle* je *open–source* systém pro správu vzdělávacích kurzů. Obsahuje mnoho funkcí, např. možnost vytvářet různé typy testů, vygenerovaných z předpřipravených statických otázek [18].

Přestože je možné všechny tyto nástroje využívat, nejsou přímo určeny pro tvorbu testů nebo mají každý jinou množinu těchto funkcionalit, mohou vyžadovat speciální instalaci nebo být pro učitele příliš složité.

I proto tvořím aplikaci přímo zaměřenou na správu a generování otázek a testů, kterou bude možné jednoduše integrovat do dalších systémů pomocí *API*.

Kapitola 3

Technologie a principy vývoje webových aplikací

V dnešní době je webová aplikace první volbou pro většinu vývojářů, zejména protože webová aplikace je dostupná z jakéhokoliv zařízení, které má přístup k internetu přes webový prohlížeč. Webová aplikace je také snadno aktualizovatelná a není potřeba ji instalovat na každém zařízení zvlášť, stačí pouze do webového prohlížeče zadat adresu aplikace.

Vývojáři se také nemusí příliš starat o kompatibilitu, jelikož webový prohlížeč je dostupný na všech hlavních platformách a webové technologie jsou od roku 1994 ve většině hlavních prohlížečů standardizovány (viz `https://www.w3.org/standards/`). Webová aplikace je také snadno škálovatelná, protože je uložena na serveru a uživatelé se k ní připojují. Navíc je možné z jednoho zdrojového kódu (např. v *JS*) nechat jednoduše sestavit nativní[1] aplikaci ve verzích pro různé platformy (Desktop, Android, IOS, ...) [3].

Většina dnešních projektů je tedy vyvíjena jako *MVC* webová *aplikace*. Tím je možné oddělit logiku aplikace od jejího zobrazení, což umožňuje vývoj funkční části aplikace nezávisle na uživatelské části a je možné pracovat s různými technologiemi.

Obvykle není nutné tvořit aplikaci od začátku (tzv. **na zelené louce**). „Pro *FE* i *BE* aplikace existuje na webu velké množství použitelných materiálů. Ať už jsou to různé grafické prvky nebo serverové komponenty, to vše je už někde nejspíše hotové.“ [34].

3.1 Základní statická HTML stránka

V roce 1989 vytvořil Tim Berners-Lee první webový prohlížeč a jazyk *HTML*.

Hypertext Markup Language (HTML), česky **Hypertextový značkovací jazyk**, je strukturovaný jazyk, obsahující textový obsah a hiearchicky strukturované **značky** (**tagy**), které určují význam jednotlivých částí dokumentu.

Jednotlivé soubory na různých adresách v síti lze také propojit pomocí **hypertextových odkazů**, které umožňují uživateli kliknutím přecházet z jedné stránky na druhou.

Jednotlivé *HTML* stránky se skládají z *HTML* značek (*prvků*), ale i *CSS* stylů, které určují, jak má prohlížeč jednotlivé prvky stránky vykreslit uživateli. K tomu se přidává i skriptovací jazyk *JS*, který umožňuje přidat do stránky dynamický obsah a reagovat na uživatelské akce jako kliknutí myši nebo odeslání formuláře [9].

Jednotlivé prvky je také možné označit pomocí *identifikátorů* a *tříd*, které se používají pro jejich identifikaci ve stránce. Zatímco *tříd* může mít prvek více a jednu třídu je možné použít vícekrát, týž identifikátor může být použit v dokumentu pouze jednou – měl by být unikátní. *Třídy* se používají pro označení skupiny prvků, které mají stejný vzhled nebo funkcionalitu. Prvek může mít více *tříd* a kombinovat tak jejich vlastnosti [42].

3.2 Markdown

Markdown (MD) je textový formát, umožňující jednoduché formátování textu. Byl vytvořen v roce 2004 **Johnem Gruberem** ve spolupráci s **Aaronem Swartzem** s cílem být co nejčitelnější a snadno převoditelný do *HTML*.

MD používá formátování pomocí znaků, jako jsou hvězdičky (tučný text) nebo podtržítka pro kurzívu. Je oblíbený mezi vývojáři a dalšími uživateli webu pro jeho jednoduchost a čitelnost. *MD* je často používán na platformách jako Gitlab pro psaní dokumentace ke zdrojovým kódům nebo na webových fórech nebo blogovacích platformách, které tak umožňují uživatelům snadno formátovat jejich texty [11].

[1] Aplikace sestavená a spustitelná na konkrétní platformě

3.3 HTML s dynamickým obsahem

HTML stránky mohou mít i dynamický obsah. To znamená, že je obsah možné měnit podle toho, jak se stránkou uživatel pracuje. Na úrovni serveru je možné vygenerovat *HTML* stránku, která obsahuje dynamický obsah např. data z databáze.

V dnešní době je oblíbený přístup generovat výslednou stránku na serveru a pomocí *JS* si při změnách od *serveru* vyžádat pouze data, nicméně v minulosti se dynamický obsah generoval převážně na serveru pomocí technologií jako **Hypertext Preprocessor (PHP)** nebo **ASP.NET**, které na serveru generují kompletní *HTML* stránku i s načtenými daty [46, 13].

To samozřejmě znamená, že při každé změně obsahu je nutné znovu kontaktovat server a stránku znovu načíst do prohlížeče a celou vykreslit. Také je veškerá komunikace omezená na požadavek od klienta a odpověď serveru, který nemá možnost samostatně zaslat zprávu (data) klientovi. Tento problém adresuje *JS*. Dokáže zasílat požadavky na server **asynchronně** a na základě odpovědi mění obsah celé stránky nebo její části. Díky metodě **Asynchronous Javascript Requests (AJAX)** [56] je možné komunikovat se serverem i bez načítání celé stránky.

To znamená, že když např. uživatel klikne na tlačítko, nemusí se stránka znovu načítat celá, pouze jsou odeslána data z vyplněného formuláře a ve stránce je zobrazena odpověď ze serveru. To je rychlejší a příjemnější pro uživatele. Navíc to snižuje zátěž na vykreslování stránky na straně serveru.

Server může – samozřejmě po povolení ze strany uživatele – zasílat zprávy na klientské zařízení a vyžádat si reakci klienta díky **Server Sent Events (SSE)** [104, 59] nebo udržovat obousměrné spojení pomocí **Web Socket (WS)**, čehož využívají např. aplikace pro přímý přenos videa (livestream), online hry nebo platformy pro internetové hovory a chaty [63].

3.4 Single Page Application

Single Page Application (SPA) je aplikace, která je prohlížečem načtena pouze jednou a poté pouze aktualizuje její obsah pomocí technologie *AJAX*. Většinou k práci s daty využívá nějaké *API*.

Statické SPA jsou vygenerovány na serveru a obsahují všechny potřebné zdroje. Jde o *statickou webovou stránku*, která je vytvořena pomocí *HTML*, *CSS* a *JS*.

Serverless aplikace využívají k zajištění některých (nebo všech) svých funkcí již připravené *API* služeb třetích stran, např. **Google Firebase** nebo **Amazon Web Services**, což usnadňuje vývoj a umožňuje vývojářům soustředit se na vývoj aplikace, a neřešit tolik správu serveru a psaní *BE*.

3.5 Kompilátor a interpret

Pro lepší pochopení programování aplikací je potřeba ujasnit rozdíl mezi kompilovaným a interpretovaným jazykem. Souvisí to se způsobem, jakým se programy vykonávají.

Kompilovaný jazyk je jazyk, který se pomocí tzv. **překladače** (**kompilátoru**) překládá do binárního souboru, který obsahuje přímo instrukce pro procesor, ten nazýváme **program**. Při každém spuštění se potom do paměti načte program s instrukcemi a procesor je vykonává.

Při překladu (*kompilaci*) překladač navíc kontroluje správnost zápisu a případné zjistitelné chyby vypíše. To umožňuje odhalit chyby před spuštěním programu.

Kompilátor při překladu navíc provádí optimalizace, které zvyšují výkon a snižují nároky programu na *hardware*. Kompilovaný jazyk je většinou i **staticky typovaný**, což znamená, že při překladu navíc zkontroluje správnost typů a kompatibilitu proměnných a funkcí.

Kompilovaný program je tím rychlejší, nicméně při jeho tvorbě je potřeba řešit některé problémy, např. správu paměti. Jde např. o jazyky **C a C++**, **Rust** nebo **Go**. Tyto jazyky jsou blíže ke konkrétnímu **hardware** a nazýváme je **nízkoúrovňové**.

Interpretovaný jazyk je naopak vykonáván programem, kterému říkáme **interpret**. Ten čte příkazy a přímo je vykonává. To znamená, že při každém spuštění se příkazy překládají znovu. *Interpret* je program, který musí běžet, zatímco příkazy interpretuje, což je také jeden z důvodů, proč bývají interpretované jazyky pomalejší a náročnější na využití zdrojů, než ty kompilované. Kód v takovém jazyce nazýváme **skript**.

Takové jazyky jsou většinou také **dynamicky typované**, tedy jazyky provádí kontrolu typů proměnných a jejich použití až při vykonávání programu a často umožňují i změnu typu proměnné za běhu programu,

což zavádí nutnost dalších kontrol a zvyšuje nároky na procesor. Proto vznikl např. jazyk **Typescript (TS)**, který v *JS* umožňuje používat statické typování a následně se *transpiluje* zpět do *JS*.

Interpretování ale přináší některé výhody, zejména pro vývojáře. Například je možné příkazy vykonávat ihned, bez nutnosti překladu a není potřeba obstarávat správu paměti. Do této kategorie patří např. *JS*, *PHP*, **Python** nebo **Perl**. Tyto jazyky můžeme zařadit do **vysokoúrovňových jazyků** [105].

Transpilací nazýváme proces podobný *kompilaci*, kde mezi sebou překládáme *vysokoúrovňové jazyky*. Např. z jazyka *TS* do *JS* [29].

„**Just In Time (JIT) kompilace** je způsob, jakým *interprety* urychlují běh skriptů tím, že během provádění programu překládají často prováděné části kódu (tzv. **hot paths**) do strojového kódu. Tento proces začíná interpretací skriptu, identifikací *hot paths* a jejich následným překladem do optimalizovaného strojového kódu, který je ukládán do *cache*. Díky tomu může v příštím běhu programu interpret spouštět tyto části kódu přímo, což významně urychluje jejich provádění a zvyšuje celkový výkon aplikace“ [72].

3.6 Kompilátor

Překladač je program, který čte zdrojový kód a překládá ho do podoby binárního souboru, který obsahuje instrukce vykonávané procesorem. Překladač také kontroluje správnost zápisu a případné zjistitelné chyby vypíše. To umožňuje odhalit některé chyby ještě před spuštěním výsledného programu. *Překladače* i *interprety* pracují v několika fázích:

Lexikální analýza *kompilátor* čte zdrojový kód znak po znaku a rozděluje jej na jednotlivé *tokeny* a určuje jejich typ (slova, čísla, operátory…). Při tom kontroluje správnost zápisu a případné chyby vypíše. Výstupem je seznam tokenů.

Syntaktická analýza kontroluje, jestli jsou jednotlivé *tokeny* správně strukturovány a v jakém vztahu k sobě stojí. Generuje **Abstraktní Syntaktický Strom (AST)** (**Abstract Syntax Tree**), který reprezentuje strukturu programu a umožňuje tak další zpracování a optimalizaci.

Sémantická analýza kontroluje správnost použití jednotlivých *tokenů* a jejich význam. Generuje tabulku symbolů, která obsahuje informace o proměnných a funkcích, dostupných v programu.

Optimalizace v této fázi probíhá **optimalizace** kódu, aby byl ve výsledku rychlejší a méně náročný na procesor. Optimalizace se děje v závislosti na použitém jazyce a překladači a může být prováděna několikrát v různých fázích překladu (a nemusí být prováděna vůbec).

Generování kódu Z *AST*, je možné generovat výsledný *program* nebo požadovanou formu dat [82, 92].

Překladač umožňuje k optimalizaci používat výrazy, které jsou vyhodnoceny během překladu a nezatěžují tolik aplikaci při běhu. Může jít o různá **makra**, která jsou provedena již během překladu, nebo použití konstantních výrazů, které jsou v kódu přímo nahrazeny. Některé jazyky (např. Zig pomocí `comptime`) umožňují i přímé vyhodnocení kódu během kompilace.

V kontextu **kompilátorů** a **optimalizace** kódu jsou tzv. **bitové masky** a další práce s bitovou reprezentací čísel na nejnižší úrovni výhodné. Použití instrukcí v **Jazyce Strojových Instrukcí (JSI)** (**Assembly (ASM)**) pro bitové maskování umožňuje vývojářům použít elementární instrukce procesoru pro práci s čísly (v tomto případě převážně instrukce `AND`), což vede ke zvýšení výkonu a efektivity programu.

V jazyce *Go* je možné využít koncept `iota konstant`, což je obdoba **výčtových typů** v jiných jazycích, viz 3.1.a. Díky němu je možné vytvořit řady číselných konstant, kde je každému prvku automaticky přiřazena následující číselná hodnota. Díky `iota` je možné také udržovat násobky čísel nebo **bitový posun** hodnot. Počítač navíc rychleji pracuje s čísly než s řetězci nebo jinými typy. Díky použití konstant *iota* je možné číselné identifikátory v kódu označit textově a při *kompilaci* budou nahrazeny příslušnými čísly.

To využívá např. systém Linux při určování práv k souborům. Ukázka použití v aplikaci pro generování testů je vidět v kódu 3.1.b.

Tato technika je často používaná pro definici **bitových masek**, které jsou poté využívány k testování, nastavování nebo odstranění specifických *bitů*. Tyto operace jsou navíc velmi rychlé, jelikož **bitové operace** jsou elementární instrukce procesoru. Použití *iota* v kombinaci s **operátorem** `<<` umožňuje posunovat číslo o bit, tedy navýší mocninu čísla 2.

Tento přístup usnadňuje čitelnost a údržbu kódu a také zlepšuje výkon aplikací, jelikož používá rychlé bitové operace. Pokud je navíc potřeba zkontrolovat více oprávnění, použijeme bitový součet, který je

```go
const (
    A = iota + 2 // 2 // začína číslem 2
    B // 3
    C // 4
)

const (
    D = iota * 2 // 0
    E // 2
    F // 4
)

const (
    G = iota * 3 + 1 // 1
    H // 4
    I // 7
)
```

(a) Různé možnosti použití číselníků *Iota*

```go
// Permission types
type Permissions uint8
const (
    ReadAllQuestions Permissions = 1 << iota // 2^0
    EditAllQuestions // 2^1 = 2
    EditAllCategories // 2^2 = 4
)

// Check if user has all of given permissions
//
// "user.UserCan(EditAllQuestions |
↪  EditAllCategories)"
// ...
func (u *User) UserCan(p Permissions) bool {
    // check if all relevant bits are present
    return u.Permission&p == p
}
```

(b) Bitová maska s použitím `<< iota`

Kód 3.1 Příklady využití *iota* číselníků a bitové masky

při překladu nahrazen konečným číslem, čímž se sníží počet operací potřebných ke kontrole. Význam *iota* a operací s bitovými maskami tak nachází uplatnění v široké škále aplikací, od nízkoúrovňového systémového programování po vysokoúrovňové aplikace, kde je potřeba efektivní práce s **konfiguračními příznaky** nebo **indikátory stavu**. Ty potom také potřebují méně prostoru v paměti [31, 92].

3.7 Web Assembly

Standard **Web Assembly (WASM)**, zavedený v roce 2017, mění způsob vývoje webových aplikací. Díky němu je možné spouštět programy napsané v jazycích jako **C**, **C++**, **Rust**, nebo i **Go** přímo v prohlížeči, nezávisle na platformě (Windows, MacOS, Linux ap.). Tím překonává omezení *JS* a otevírá dveře novým možnostem k výkonným, bezpečným a jednoduše přenositelným aplikacím [62].

To umožňuje pomocí *WASM* provozovat náročnější webové aplikací, jako jsou náročnější hry, 3D modelování, komplexní nástroje pro úpravu fotografií, nebo decentralizované aplikace (DApps) běžící na technologii **blockchain**. *WASM* je tak klíčovou technologií pro budoucnost webového vývoje [67].

3.8 Procedurální a objektově orientované programování

Procedurální programování je způsob programování, kdy se program skládá z jednotlivých **procedur**, které se volají po sobě (**sekvenčně**, **synchronně**). *Procedura* je část programu, která vykonává určitou činnost. *Procedury* mohou mít parametry, které jsou jim předány při volání a mění jejich funkci. *Procedury* mohou volat další *procedury*, což umožňuje znovupoužití kódu. *Funkce* jsou podobné *procedurám* s tím rozdílem, že funkce má tzv. **návratovou hodnotu**, kterou po svém dokončení vrací klíčovým slovem (`return`). *Procedura* je část kódu, která se vkládá do běhu programu a pouze vykoná své instrukce, nemá *návratovou hodnotu*, může mít pouze *vedlejší efekt.*

Procedury i *funkce* mohou mít tzv. **vedlejší efekty** (**side effects**), což znamená, že mohou ovlivnit chod programu změnou **stavu aplikace**, např. změnou hodnoty globální proměnné [78].

V dnešní době se častěji používá *OOP*, protože umožňuje vývojáři programovat v abstraktnější rovině a tím zjednodušit vývoj a údržbu programu. Jednotlivé objekty představují modely objektů z reálného světa, což je pro člověka přirozenější způsob myšlení. Při práci v týmu je také jednodušší rozdělit si práci na jednotlivé části programu, na kterých mohou vývojáři pracovat paralelně.

Objektově Orientované Programování (OOP) je programovací paradigma, které se snaží modelovat objekty z reálného světa. V *OOP* se program skládá z objektů, které mají své vlastnosti a metody (funkce). *Objekty* vznikají ze tříd. Třída je předpis, podle kterého je vytvořen (*inicializován*) konkrétní objekt.

OOP má 4 základní vlastnosti, které jsou při vývoji využívány [19].

Zapouzdření objekty mohou své vlastnosti a metody skrýt před zbytkem programu a jinými objekty. To umožňuje měnit vnitřní stav objektu bez nutnosti měnit ostatní části programu.

Dědičnost objekty mohou dědit vlastnosti a metody od jiných objektů. To umožňuje znovupoužití kódu a zjednodušení programu.

Polymorfismus objekty mohou mít stejné vlastnosti a metody, ale mohou se lišit v implementaci. To umožňuje vytvářet obecné třídy, které mohou být použity pro více konkrétních případů.

Abstrakce nemusí být implementována plná funkcionalita třídy a nelze ji samostatně *inicializovat.* To umožňuje vytvářet obecné (tzv. *bázové*) třídy, které mohou být použity jako vzor pro odvození jiných konkrétních tříd, využívajících jejich vlastnosti a metody.

Jazyk *Go* (Golang) podporuje *OOP* nepřímo a pouze částečně. Jde o jazyk *strukturovaný*, nikoliv *objektově orientovaný.* Některých vlastností *OOP* ale je možné dosáhnout pomocí použití *struktur* a **rozhraní**. Metody na strukturách je možné vytvořit za pomocí tzv. **funkcí na struktuře** (**receiver function**) a imitovat tím běžnou funkcionalitu *metod* [15, 78].

Díky kombinaci jednoduchosti v různých aspektech a rychlosti kompilovaného programu je *Go* jeden z jazyků, kterým vývojáři postupně nahrazují prokazatelně pomalý **Python** a preferují je pro nové projekty. Další takové jazyky jsou např. **Rust** nebo pro vědecké výpočty a strojové učení využívané jazyky **Julia** čí **R** [6].

Výraz *rozhraní* běžně definuje komunikaci mezi objekty, viz kapitola věnovaná *API.* V případě *OOP* definují vzor, podle kterého je potřeba implementovat budoucí třídy [15].

JS je často považován za *objektově orientovaný jazyk*, nicméně *OOP* implementuje formou tzv. prototypové dědičnosti, nikoliv tradičním použitím tříd. V praxi tak má každý objekt definovanou vlastnost označující mateřský objekt, ze kterého vychází, a dědí jeho vlastnosti [46, 78, 2.1.01].

V příkladě 3.2 je pro porovnání ta samá funkcionalita zapsána v jazyce *JS* pomocí *tříd* a v jazyce *Go* pomocí *struktur*, *rozhraní* a tzv. *receiver function*, což jsou funkce definované na dané struktuře (podobně jako *metody* ve třídě). **Rozhraní** (**interface**) zajišťují kontrolu, že na dané struktuře existují požadované *metody.* Je tak možné kontrolovat, že má vyšetřovaný objekt požadované vlastnosti a metody, a je možné jej používat zamýšleným způsobem, např. v parametru funkce. Je také možné vytvořit více typů, které odpovídají stejnému rozhraní [15].

V návaznosti na *OOP* se rozvíjí **Aspektové programování**, které umožňuje rozdělit program do *aspektů*, které se při vykonávání programu vkládají do kódu [51].

3.9 Javascript

Javascript (JS) je *skriptovací* jazyk, který se používá k tvorbě interaktivních webových stránek. Dokáže reagovat na uživatelské akce, jako je kliknutí, a tedy na rozdíl od statické podoby *HTML* stránky dokáže měnit obsah podle toho, jak uživatel se stránkou pracuje bez nutnosti nového požadavku na server.

Vznikl v roce 1995 a jeho tvůrci jsou *Brendan Eich* a společnost *Netscape.* *JS* (resp. *TS*) je podle – průzkumů síte **Stack Overflow** – v současnosti nejpoužívanějším programovacím jazykem na světě používaným nejen pro tvorbu webových aplikací, ale i pro tvorbu *nativních* a dokonce některých *embedded* aplikací, což jsou programy určené pro konkrétní elektronická zařízení s konkrétním účelem, např. rozhraní pro dotyková zařízení (GPS, televize, kontrolní panel v autě ap.).

Díky spouštěčům *JS* (tzv. *enginy*), je možné spouštět *JS* na různých platformách. V prohlížeči, ale i na serveru nebo i na mikrokontrolerech. Jeden jazyk tak zvládne pokrýt většinu potřeb, které může autor aplikace při vývoji pro různé platformy mít.

JS dokáže příkazy zpracovávat *synchronně* (sekvenčně) i *asynchronně* (paralelně). **Synchronní** zpracování znamená, že se příkazy vykonávají postupně, tak, jak jsou zapsány ve skriptu. **Asynchronní** zpracování naopak znamená, že se příkazy vykonávají nezávisle na sobě, tedy spuštění dalšího příkazu nečeká na dokončení předchozího. To je užitečné např. při načítání dat ze serveru, které může trvat delší dobu. V tomto případě je možné spustit *asynchronní akce* a pokračovat ve zpracování sekvence příkazů programu. Když dojde k dokončení *asynchronní akce*, vykoná se příkaz, který na její dokončení čeká. V rámci *asynchronní operace* je možné čekat na dokončení více akcí [56, 78].

Zpracování *JS* je rozděleno do dvou částí.

```javascript
class Animal {
    constructor(name) {
        this.name = name;
    }

    makeSound() {
        if (!this.sound) {
            const name = this.constructor.name;
            console.log(`What does the ${name}
            ↪  say?`);
            return;
        }
        console.log(`My name is ${this.name} and I
        ↪  go ${this.sound}`);
    }
}

class Dog extends Animal {
    constructor(name) {
        super(name);
        this.sound = "Haf";
    }
}

class Cat extends Animal {
    constructor(name) {
        super(name);
        this.sound = "Mňau";
    }
}

class Fox extends Animal {}

const a = new Dog("Alík");
a.makeSound();
const b = new Cat("Micka");
b.makeSound();
const c = new Fox("Foxy");
c.makeSound();
```

```go
// Animal is an abstract class (interface) in Go
type Animal interface {
    makeSound()
}

// AnimalBase is a base struct to store common
↪  properties of animals
type AnimalBase struct {
    sound string
    name  string
}

// Type (struct) representing a dog
type Dog struct {
    AnimalBase
}

// Cat
type Cat struct {
    AnimalBase
}

// Constructor-like function to create an instance
↪  of AnimalBase
func NewAnimalBase(name, sound string) *AnimalBase {
    return &AnimalBase{
        name: name,
        sound: sound,
    }
}

// Metoda zděděná všemi odvozenými objekty
func (a *AnimalBase) makeSound() {
    fmt.Printf("My name is %s and I go %s\n",
↪   a.name, a.sound)
}

// a musí implementovat rozhraní Animal
func animalMakeSound(a Animal) {
    a.makeSound()
}

func main() {
    a := &Dog{
        AnimalBase: *NewAnimalBase("Alík", "haf"),
    }
    animalMakeSound(a)

    b := &Cat{
        AnimalBase: *NewAnimalBase("Micka", "mňau"),
    }
    animalMakeSound(b)
}
```

Kód 3.2 Zápis OOP principů v jazycích Javascript a Go

První částí je ***JS* engine** (**běhové prostředí**), které zpracovává *JS* kód a podle toho spouští instrukce. Druhou částí jsou tzv. **Web APIs**, které umožňují využívat funkce prohlížeče, systému a další asynchronní operace. Jde o instrukce, poskytované nikoliv *JS enginem*, ale webovým prohlížečem nebo operačním systémem. Jde např. o přístup ke struktuře *DOM*, kameře, mikrofonu nebo asynchronním operacím, jako je načítání dat ze serveru. Jde převážně o asynchronní operace a služby systému, jako např. použití mikrofonu nebo přístup k souborovému systému [46].

3.10 Typescript

Jednou z nevýhod *JS* je, že není typově bezpečný. To znamená, že není možné při překladu zkontrolovat, zda jsou všechny proměnné a funkce použity správně, a jednotlivé typy je dokonce možné za běhu měnit. To může vést k chybám, které se projeví až při spuštění programu, což je pro větší projekty nežádoucí, a proto se používají nástroje, které typově bezpečné jsou a po kontrole jsou přeloženy (*transpilovány*) zpět do *JS*. Jedním z těchto nástrojů je jazyk *TS*. Kromě typové kontroly přináší *TS* i další výhody a optimalizace, které zjednodušují vývoj a zvyšují její výkon, a přidává další výhody oproti použití pouze *JS*.

TS přidává do *JS* statickou kontrolu typů, což znamená, že proměnné mají přiřazený datový typ, který se při běhu aplikace nemění, a jejich použití *TS* kontroluje během vývoje, což by v *JS* normálně neprobíhalo. Díky kontrole typů je možné provést při překladu kontrolu, zda jsou všechny proměnné a funkce použity správně, což výrazně redukuje množství chyb a výpadků během provozu aplikace. Navíc umožňuje bezpečně používat některé nové vlastnosti *JS*, které ještě nejsou podporovány ve všech prohlížečích tím, že novou funkcionalitu přepíše pomocí nástrojů dostupných ve starších standardech jazyka. Také umožňuje tvořit vlastní typy a struktury, které pomohou v rámci zpracování *AJAX* požadavků. Díky tomu je možné v *IDE* používat automatické doplňování vlastností objektů a mít tak lepší přehled o konkrétních položkách ve strukturách dat. Mmj. je eliminována možnost chyby v zápisu názvu položky.

TS je po kontrole přeložen (*transpilován*) zpět do vybrané verze *JS*. [55, Get started/TypeScript for the New Programmer]

3.11 Document Object Model

Document Object Model (DOM) je *objektový model dokumentu*, který vytváří konkrétní prohlížeč na základě *HTML kódu* stránky před samotným vykreslením stránky uživateli. Jednotlivé *HTML* prvky jsou reprezentovány objekty, tzv. *uzly*. *DOM* obsahuje *objekty* (uzly) vytvořené z jednotlivých prvků v *HTML* kódu. K nim pak prohlížeč přiřadí ***CSS*** styly a nakonec **prohlížečové jádro** vykreslí stránku na obrazovku uživatele [8, 58, 98].

K DOMu je potom možné přistupovat přes *JS* pomocí globálních objektů `document` a `window`, pomocí kterých je možné aktivně měnit obsah stránky včetně stylů nebo sledovat uživatelské akce a na jejich základě spouštět vlastní funkce [46].

3.12 React

React je oblíbená knihovna pro tvorbu *uživatelského rozhraní*. Je vyvíjena společností *Facebook* a je používána při vývoji mnoha webových aplikací. Je oblíbená pro svou jednoduchost a rychlost. Je také dobře dokumentovaná a má velkou komunitu, která ji podporuje.

Nejnáročnější operace na výpočetní výkon je vykreslování webu a změn v něm. Tedy změny v *DOMu*. *React* je díky využití tzv. *Virtuálního DOMu* velice rychlý. *Virtuální DOM* je kopie *DOMu*, která se používá pro porovnání změn. To umožňuje aplikaci rozpoznat, kdy je potřeba provést zásah do reálného *DOMu* a kdy ne. Zamezí se tak zbytečným operacím, což zvýší výkon aplikace [20].

React je postaven na tzv. *deklarativním* přístupu. To znamená, že kódu popisuje pouze, jak má aplikace vypadat, nikoliv jak se má chovat. To umožňuje *Reactu* optimalizovat vykreslování a zvýšit tím výkon aplikace.

Existuje mnoho nástrojů, které usnadňují vývoj aplikací v *Reactu*, jako např. **Create React App** (zkráceně **CRA**), nebo komplexnější nástroje jako **NextJS** a **Vite**. Externí moduly je možné v aplikacích používat pomocí balíčkovacích systémů, např. **yarn**.

Jednotlivé *komponenty* mohou mít svůj vlastní **stav**, který se může měnit, což umožňuje vytvářet interaktivní aplikace. Komponenty mohou být **funkční** nebo **třídní**. *Funkční* komponenty jsou jednodušší a práce

s nimi je rychlejší. *Třídní komponenty* jsou rozsáhlejší, ale nabízejí k použití všechny výhody Reactu. *Funkční* komponenty mohou k dosažení podobné funkcionality jako *třídní* komponenty využívat tzv. **hooks**. *Hooks* jsou stále ve vývoji a vývojáři používají *třídní* komponenty, pokud potřebují využít některé funkce, které *Hooks* zatím nepodporují.

Komponenty mohou obsahovat další (*vnořené*) komponenty. To se nazývá *zanořování* (*nesting*). To umožňuje vytvářet složitější komponenty s více vnořenými prvky. S použitím `React.createElement` (`https://react.dev/reference/react/createElement`) pro každý prvek by ale takový kód byl složitě čitelný. Proto je k dispozici zjednodušený zápis v souborech *JSX*.

Při psaní *stavové aplikace* (aplikace, která udržuje svůj vnitřní stav) v *Reactu* je také důležité vědět, že v něm funguje tzv. **jednosměrný tok dat** (**one directional data flow**). To znamená, že data se předávají směrem z **rodičovské komponenty potomkům**, nikoliv naopak. To umožňuje jednodušší kontrolu nad daty, zároveň to ale přináší výzvy při správě stavu celé aplikace.

Z toho vyplývá problém zvaný **prop drilling** (*rekurzivní předávání dat komponentami*), který spočívá v tom, že aby se data dostala k cílové komponentě, je nutné je předávat přes několik vnořených komponent, které data vůbec nepotřebují. To může vést k tomu, že se v *komponentách* objeví nepotřebné kódy, které způsobují zbytečnou komplexitu a snadněji se do kódu zavede nová chyba. Při změně jedné proměnné, která je předávána přes několik komponent, je kvůli tomu potřeba upravit kód všech *komponent*, které tuto proměnnou používají, a jde často o nepřehledný a zdlouhavý proces. Při změně dat dochází také ke zbytečnému překreslení všech komponent, které data předávají, i když se změna viditelně projeví pouze v jedné cílové komponentě, což je z hlediska výkonu nežádoucí. Proto je snaha vývojáře přesunout zdroj dat, se kterým pracuje, na nejvyšší možnou úroveň, aby bylo možné data předávat přímo *komponentám*, které je potřebují.

Prop drilling se řeší přesunem zdroje dat na nejvyšší možnou úroveň a na změny jednotlivých dat upozorňovat pouze ty komponenty, které to potřebují. Tím se zjednoduší kód a zvýší se přehlednost. Navíc se sníží počet překreslení komponent, což sníží počet operací a tím zvýší výkon aplikace.

K tomu je možné využít *React Context API* nebo podobné nástroje, které umožňují centrálně udržovat **stav aplikace** (**kontext**) a zpřístupnit jej více *komponentám*. *Kontext* je dostupný v komponentách, které jsou vnořené v komponentě (**potomci**), *kontext* poskytující. Obvykle se *kontext* nachází v samostatném souboru, kde je vytvořen pomocí `React.createContext` a poté obalí celou aplikaci pomocí komponenty **poskytovatele kontextu** (**provideru**).

V této práci pro správu stavu aplikace použiji knihovnu **Zustand**, která je moderní, rychlá a nabízí spoustu výhod oproti *Contextu* i knihovnám jako **Redux**, **Mobx** apod. [64].

3.13 Javascript XML

Javascript XML (JSX) je rozšíření jazyka *JS*, které umožňuje využívat zápisu *komponent* podobně jako v *HTML kódu*. Jednotlivé *prvky* jsou tzv. **komponenty**, které se při překladu přepisují na `React.createElement` a vykreslují se do HTML. To umožňuje jednodušší tvorbu komponent [53].

Rozdíl práce na tvorbě komponent za použití `React.createElement`, resp. `JSX` je vidět v kódu 3.3.

```
function ComponentA({ name }) {
    return React.createElement(
        // typ prvku
        'h1',
        // atributy
        { className: 'greeting' },
        // textový obsah
        'Hello World!'
        );
    }
}
```

```
const App = () => {
    return (
        <h1 className="greeting">Hello
        ↪  World!</h1>
    );
}
```

Kód 3.3 Rozdíl použití React vs. JSX

NextJS je *framework* pro *React*, který umožňuje vytvářet *SSR* aplikace. *NextJS* je vyvíjen společností *Vercel*, která je autorem *serverless* řešení pro aplikace běžící v prostředí **Node.js**.

V základním nastavení umožňuje využívat *SSR* i *SSG*, čímž jsou aplikace v *Nextu* velmi rychlé a přitom poskytují všechny výhody *Reactu* a je možné použít i většinu balíčků pro *React*.

Díky tomu je možné jednoduchým vytvořením *NextJS* projektu začít pracovat na *SSR* aplikaci, a to jak na *FE*, tak na *BE*, jelikož od verze 9.3 je možné využít také tzv.**API Routes**, které umožňují vytvořit *REST API* na jednom místě přímo v *NextJS* aplikaci [95, 94, 93].

Vite je nástroj pro rychlé nastartování projektu nejen v Reactu, ale i ve Vue a *Vanilla Javascriptu* nebo *Typescriptu*. Je vyvíjen společností `Vue.js`, která je známá díky svému frameworku *Vue.js*, který podobně jako *React* využívá *virtuální DOM*.

Vite (i *CRA*) zrychluje spouštění aplikace a zjednodušuje práci díky tzv. *hot reload*, což znamená, že se při každé změně kódu *FE* aplikace v prohlížeči automaticky načte znovu s novými změnami, bez nutnosti manipulace s prohlížečem [106].

3.14 Server Side Rendering a Static Site Generation

Render (**vykreslování**) se v kontextu počítačů používá k popisu procesu vykreslování obrazu z dat za účelem jejich zobrazení uživateli. Tato data mohou být např. ve formě 3D modelů, obrázků nebo třeba interaktivních webových prvků při vykreslování webu prohlížečovým jádrem [79].

V kontextu knihovny *React* se slovo *render* používá k popisu procesu vytváření *HTML* a *CSS* kódu ze struktury *JSX* komponent.

Přestože je *React* velmi rychlý oproti aplikaci v čistém *JS* (tzv. **Vanilla**), zejména díky použití *Virtuálního DOMu*, je při načítání aplikace v prohlížeči stále nutné počkat, než se stáhnou, načtou a provedou skripty. Teprve potom se vygeneruje výsledný *HTML* obsah a prohlížeč stránku vykreslí (**Client Side Rendering (CSR)**)[96]. To může být problém, pokud je aplikace rozsáhlá a obsahuje mnoho komponent, kde celý proces může trvat delší dobu a uživatel musí na vykreslení déle čekat, což je samozřejmě nežádoucí. To je možné řešit pomocí metod **Static Site Generation (SSG)** a **Server Side Rendering (SSR)**,

SSG (někdy *pre-rendering*) je způsob zrychlení webů, kde se generují **všechny** statické soubory z dat načtených z dynamického zdroje (např. *CMS – Content Management System (redakční systém)*). Data jsou načítána pluginem a z nich jsou – podle předdefinované šablony – generovány statické soubory. Díky tomu, že jde o čistě statický obsah, bez nutnosti *AJAX* požadavků nebo dynamického generování, je tento postup rychlostí na stejné úrovni jako poskytování statických souborů, které potom obsluhuje pouze *Webový Server*, a není nutné, aby běžela další *služba/aplikace*.

To samozřejmě nevylučuje použít *AJAX*, pouze je použit na uživatelské akce až po načtení aplikace, nikoliv k načítání prvotního obsahu stránky. Uživateli je poslán statický *HTML soubor*, který se vykreslí mnohem rychleji a před zpracováním *JS*, který by *HTML* nejdříve generoval a tím oddálil vykreslení prohlížečem. Následně se na vygenerované *komponenty* aplikuje jejich logika a akce. Tento proces se nazývá **hydratace** (**Hydration**).

SSR je koncept podobný *SSG*, pouze na straně serveru běží aplikace, která průběžně sleduje data a aktualizuje obsah (při změně v datech) za použití dané aplikační logiky a uživateli zasílá statické soubory. Při prvním požadavku na aplikaci server zašle *HTML* kód dané stránky a po provedení *hydratace* aplikace začne fungovat normálním způsobem. To umožňuje vytvořit aplikaci, která se načte rychleji a přitom neztrácí žádné výhody knihovny, kterou používáme (např. *React*). Při každém dalším požadavku *SSR* zašle uživateli kód pro požadovanou část aplikace. Často se používá v případě, že i *BE* aplikace používá *JS* [13].

Takto systém vygeneruje *HTML kód* (se skripty ap.) pro každou stránku pouze v případě změny obsahu. Při dalším spuštění potom načítá statický obsah, což ušetří výkon na straně *serveru* i *klienta*.

3.15 Client-Server

Vývoj webových aplikací je založen na modelu **klient-server**.

Server je počítač, který poskytuje služby klientovi. V případě webové aplikace je to *server*, který poskytuje *HTML stránku*, styly a skripty. Na **serveru** běží také další služby, které ukládají a poskytují data nebo zprostředkují další akce, které zajišťují funkčnost aplikace.

Klient je aplikace na zařízení uživatele (webový prohlížeč), který služby *serveru* využívá. V případě webové aplikace je to webový prohlížeč, který zobrazuje *HTML* stránku. V případě mobilní aplikace je to aplikace, která se spouští na mobilním zařízení a komunikuje se serverem. Např. messengery, sociální sítě [86].

3.16 Webová služba

Webová služba je program, který běží na *serveru* a poskytuje data *klientovi*. Na *serveru* může běžet více *webových služeb*, rozlišují se *portem*, na kterém běží a přijímají instrukce a data. Např. webová služba **Hypertext Transfer Protocol (HTTP)**, která poskytuje webová data a soubory, poběží na *portu 80*, resp. v dnešní době zejména *443* (***HTTP* Secure (HTTPS)**). Další služby mohou být např. emailová služba **Simple Mail Transfer Protocol (SMTP)**.

Služby **naslouchají** na svých *portech* a přijímají požadavky od klientů. K serverům na internetu se přistupuje pomocí *IP adresy* a ke službám pak pomocí portu, na kterém běží [61, 24].

3.17 Webový server

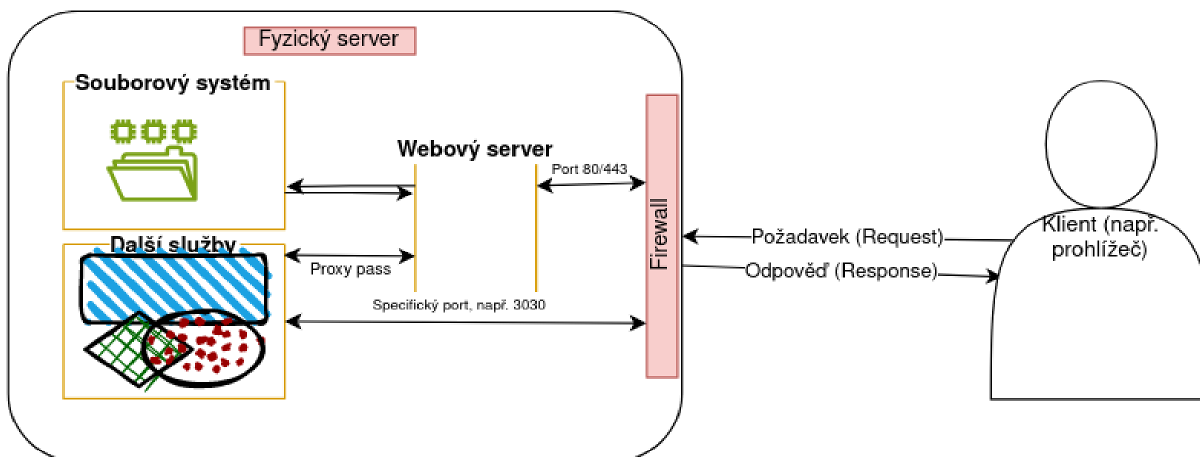
Webový server se stará o odpovědi na požadavky, přicházející většinou na port `443` nebo `80`. Server dostane požadavek ve formě **Uniform Resource Locator (URL)**, někdy **Uniform Resource Identifier (URI)**, obsahující cestu k požadovanému zdroji (souboru/službě).

Neslouží pouze k poskytování statických souborů, ale i k poskytování dynamických dat. Může zajistit, že se před odesláním soubor na serveru zpracuje (např. pomocí *PHP*) a vloží se do něj aktuální data, nebo před odesláním provede **kompresi dat** (snížení objemu dat matematickým zjednodušením souboru [78]), což může klientovi ušetřit čerpání mobilních dat. Na straně klienta se pak data *dekomprimují* a zobrazí.

Proxy v kontextu počítačových sítí jde o *server* nebo software, který působí jako prostředník pro požadavky mezi klienty a jinými servery nebo aplikacemi např. pomocí přesměrování portů nebo volání externího serveru.

Webový server je také zodpovědný za *směrování doménových jmen* ke konkrétním datům nebo aplikacím na daném serveru. To znamená, že na jednom serveru může běžet více webových služeb na různých doménách. Metoda **proxy pass** (viz *proxy*) je používána k přesměrování celého požadavku na jiný *port* (aplikaci) [61].

Webové servery mohou být např. **Apache**, **Nginx**, **Lighttpd**, **Caddy** nebo **Internet Information Service (IIS)**...



Obrázek 3.1 Schéma komunikace client-server

Hosting je služba, kterou poskytují *hostingové společnosti*, které mají *webové servery* a poskytují na nich prostor pro webové stránky (**Webhosting**), resp. pro další služby typu databáze, cloudové úložiště, vlastní aplikace nebo i vlastní fyzický server (**Serverhosting**) [48].

3.18 Domain Name Server

Domain Name Server (DNS), někdy také *Domain Name System*, je systém, který překládá doménová jména na **IP adresy**. To umožňuje uživatelům používat lidem čitelná *doménová jména* namísto *IP adres* cílového serveru, které jsou pro uživatele těžko zapamatovatelné.

Díky *DNS* např. oblíbený český vyhledávač namísto *IP adresy* `77.75.79.222` najdeme na čitelné adrese `Seznam.cz`.

Doménové jméno je složeno ze tří částí. První částí je *subdoména*, která je volitelná. Druhou částí je *doména* (2. řádu) a třetí část je *doména nejvyššího řádu* (**Top Level Domain (TLD)**) a může jít např. o národní koncovky (`.cz`, `.sk`, ...), některou ze specifických domén (`.edu`, `.org`, `.app`, ...). Doména 2. řádu a *TLD* jsou povinnou součástí adresy. Doménové jméno je po zadání do prohlížeče v procesu nazývaném ***DNS* resolve** přeloženo na *IP adresu*, která se dále použije k odeslání požadavku [60].

```
protokol  subdoména  doména          TLD   ?query
https://  robot      .killermachine  .com  / ?znicitlidstvo=false&parametr=42
```

Obrázek 3.2 Složky URI

DNS Resolve je proces překladu doménového jména na IP adresu. Když si uživatel vyžádá nějaké doménové jméno, prohlížeč se připojí k *DNS* a požádá jej o překlad doménového jména na IP adresu. *DNS* odpoví *IP adresou*, kterou prohlížeč použije pro navázání spojení se serverem.

Když je do prohlížeče zadáno doménové jméno, prohlížeč se pokusí najít *IP adresu* ve vlastní *DNS cache*, kam si ukládá již navštívené weby. Pokud adresu nenajde, připojí se k nejbližšímu DNS serveru a požádá o poskytnutí překladu jej a uloží si *IP adresu* do *cache*. Pokud *DNS server* adresu nemá, zeptá se nadřazeného *DNS serveru* atd.

Nejbližší *DNS* bývá poskytovatel připojení k internetu (**Internet Service Provider (ISP)**), který má uloženy *IP* adresy serverů, na které v minulosti zákazníci přistupovali. Pokud záznamy ještě nemá, zeptá se prohlížeč nadřazeného *DNS serveru*, resp. zašle požadavek na adresu nastaveného *DNS*. Postupně se tedy zeptá *ISP*, správce národní domény (*CZ.NIC*) a *kořenového DNS serveru* (*Root DNS server*), který má uloženy všechny *IP adresy* serverů, které jsou na internetu.

DNS Záznamy (**DNS Entries**) může upravovat vlastník domény prostřednictvím **doménového registrátora**. To umožňuje přesměrovat doménu na jiný server, nebo přidat další záznamy, jako např. *MX* záznamy, které používá e-mailová služba pro doručování e-mailů [57].

3.19 Model View Controller

Významným bodem vývoje aplikací je rozšíření používání architektury **Model View Controller (MVC)**. Tato architektura odděluje **logiku aplikace** od jejího **zobrazení**. To umožňuje vývoj funkční části aplikace nezávisle na jejím *zobrazení*.

Podle modelu *MVC* má aplikace 3 složky. **Model**, který se stará o úschovu a manipulaci s daty, **controller**, který zprostředkuje komunikaci mezi *modelem* a *zobrazením*, a **view**, který se stará o výsledné zobrazení dat.

Díky tomu je možné mít jednu aplikaci, která bude mít různá *zobrazení* (*view*), zatímco data čerpá pouze z jednoho zdroje. Např. webová stránka, mobilní i desktopová aplikace (*zobrazení*) mohou využívat týž *BE* (*model*).

Moderní aplikace díky tomu mohou snížit zátěž na server, jelikož část aplikace, která se stará o zobrazení, běží na straně klienta a mezi nimi probíhá pouze výměna dat [35].

3.20 Frontend, Backend

Aplikace lze díky architektuře *MVC* rozdělit na **frontend** a **backend**, které spolu komunikují a plní funkce aplikace, ale jinak zůstávají oddělené.

Díky *MVC* je takto možné vývoj aplikace rozdělit mezi tým vývojářů, kde každý může pracovat na jiné části aplikace a každá část aplikace má své rozhraní, takže mohou pracovat téměř nezávisle na sobě.

Backend (BE) je část aplikace, která běží na serveru a zpracovává nebo poskytuje data *FE*. Spolupracuje s databází (*Modelem*), souborovým systémem a dalšími službami, které jsou potřeba pro fungování aplikace. *BE* může být napsán v různých jazycích [16, 108], např. PHP, Java, C#, Go, *JS*, C++...

Frontend (FE) je část aplikace, kterou vidí uživatel, např. prostřednictvím prohlížeče. Stará se o zobrazení (*view*) a interakci s uživatelem (*controller*) a komunikuje s *BE*, zejména pomocí asynchronních požadavků na server. Je napsán většinou v *HTML*, *CSS* a *JS*.

3.21 Javascript Object Notation

Javascript Object Notation (JSON) je formát pro uložení strukturovaných dat. Je to textový formát čitelný pro člověka i pro stroj. Používá se ke strukturovanému ukládání a přenosu dat. Dříve byl hojně využíván formát **Extended Markup Language (XML)**, který vznikl v r. 1998 jako formát pro jednoduchou výměnu dat a který je podobný struktuře *HTML*. Ostatně *XML* stále používá spousta aplikací, zejména aplikace státní správy.

Formát *JSON* je podobný zápisu objektů v *JS*. Obsahuje dvojice *klíč-hodnota*, které jsou odděleny dvojtečkou. Klíčem je řetězec, který obvykle popisuje, jaká data jsou v něm uložena, a hodnota může být číselná, textová, logická hodnota, pole nebo další vnořená struktura dat.

Práce s formátem *JSON* je pohodlnější než s formátem *XML*, jelikož je jednodušší a čitelnější a vyžaduje méně kódu, jak na straně *serveru*, tak na straně klienta. Jde tedy celkově i o úspornější formát pro zápis strukturovaných dat a jejich přenos [16].

3.22 Asynchronous Javascript Request

Asynchronous Javascript Requests (AJAX) je technologie, která umožňuje *asynchronní* komunikaci, např. mezi *FE* a *BE*. To znamená, že při každém požadavku na *server* se stránka nemusí znovu načítat, ale použije se *AJAX*, jsou načtena požadovaná data ze serveru a změní se ve stránce.

Metody *JS* (resp. *WEB APIs*), umožňující *asynchronní* požadavky na server, jsou **Fetch** a **XMLHttpRequest (XHR)**. *AJAX* využívá různé *HTTP* metody, jako GET, POST, PUT, DELETE, PATCH. Tyto metody se používají pro odlišení typu požadavku. [56, 16].

Je také možné využít komunikaci přes *WS* nebo nedávno přidané *SSE* [104, 59].

3.23 Application Interface

Application Programming Interface (API) je rozhraní, které umožňuje komunikaci mezi aplikacemi. V případě webových aplikací se jedná o rozhraní mezi *FE* a *BE*. Některá *rozhraní* jsou určena ke komunikaci mezi aplikacemi nebo dokonce s *hardwarem*.

Někerá *API* jsou veřejná a umožňují komunikaci s aplikacemi třetích stran. Např. *API* Google Firebase, Microsoft Azure nebo Amazon Web Services umožňují tvorbu tzv. Serverless aplikací, kde vývojář tvoří pouze *FE* a o zbytek se stará externí služba. To umožňuje vývojářům se soustředit na vývoj aplikace a nemusí se starat o infrastrukturu, která je potřeba pro běh aplikace.

V dnešní době jsou nejrozšířenější formou tzv. *REST API* a *GraphQL*. Pro komunikaci mezi interními aplikacemi se často používá protokol *SOAP*. Různý hardware používá k ovládání také *UART* nebo *SPI*.

Velice oblíbená je také metoda *gRPC* (*Google Remote Procedure Call*), která je založena na protokolu *HTTP/2* a používá binární formát *Protocol Buffers*. *RPC* je zkratka pro *Remote Procedure Call* a funguje tak, že spouští vzdálené funkce, které jsou na serveru.

Výhodou *gRPC* je, že autor aplikace pouze definuje služby a jejich rozhraní a *gRPC* vygeneruje veškerý potřebný kód pro jazyk, který používáme. Také je díky efektivnější komunikaci rychlejší než jiné přístupy. Bohužel je k jeho použití nutné se naučit novému přístupu k přenosu dat a jazyku definice služeb *protobuf* [40].

3.24 Representational state transfer

Representational State Transfer (REST) je standard, který definuje způsob komunikace mezi klientem a serverem. V dnešní době je díky své jednoduchosti a rozšířenosti nejčastěji používanou architekturou pro tvorbu webových rozhraní. Každý zdroj, se kterým je potřeba pracovat, má vlastní adresu (*Endpoint*) a akce, kterou je s ním potřeba provést provést, je určena *HTTP metodou*, příp. zaslanými daty.

REST je bezstavový, což znamená, že každý požadavek je nezávislý na předchozím, server si neukládá data v závislosti na připojeném klientovi. To umožňuje jednodušší testování a ladění aplikace a trochu snižuje zátěž serveru. Také to ale přináší některé výzvy a omezení, které je potřeba řešit. Proto je sice při tvorbě *REST API* doporučeno omezit využití ukládání dat vázaných na konkrétní relaci, nicméně nástroje typu **Cookies** nebo **Session** je v případě potřeby možné využívat.

REST definuje, jaké *HTTP metody* se používají pro komunikaci, a každé přiděluje určitý význam. Mezi nejčastěji používané metody patří *GET*, *POST*, *PUT*, *DELETE* a *PATCH*. Metoda je uložena v hlavičce *HTTP* požadavku. Význam jednotlivých *HTTP metod* je následující:

GET slouží pro získání dat ze serveru,

POST pro odeslání dat na server,

PUT pro úpravu dat na serveru,

DELETE slouží pro smazání dat ze serveru,

PATCH potom pro úpravu části dat na serveru.

Souhrnně se těmto operacím říká ***CRUD*** operace, což je zkratka pro **Create**, **Read**, **Update** a **Delete**.

Kromě metod *GET* a *DELETE* se v těle *HTTP požadavku* mohou zasílat také data v textové formě, např. formátu JSON. Během přenosu je tělo požadavku zašifrováno. V případě požadavku *GET* se data vkládají do *URL* adresy, nejsou šifrována a mají velikost limitovanou maximální délkou *URL adresy*, kterou je server schopen zpracovat. Specifikace *HTTP* neuvádí žádnou maximální délku, ale v praxi je limitována prohlížeči a serverem [24, 3.2.1].

3.25 GraphQL

Naproti *REST*u má **GraphQL** pouze jeden *endpoint*. *GraphQL* je jazyk, který umožňuje klientovi specifikovat, jaká data chce klient získat. To umožňuje získat všechna potřebná data v jednom požadavku, což může být výhodné, pokud na zařízení je pomalé připojení nebo používá mobilní data, kdy je potřeba stahovat co nejméně dat a snížit počet požadavků na server.

Na druhou stranu je implementace *GraphQL* složitější než implementace *REST*, protože je potřeba implementovat *GraphQL* server, který zpracuje požadavky od klienta. Každá položka, kterou je potřeba zpřístupnit, navíc musí mít vlastní **resolver**, funkci, která zpracuje každou položku v dotazu a vrátí data. To zároveň vytváří větší zátěž na serveru, protože je potřeba pro každou položku spustit vlastní funkci (*resolver*), a to je v případě větších požadavků s rozsáhlými daty a přístupu do databáze, neefektivní [36, 16].

3.26 Zabezpečení

Vystavením aplikace na internetu se *API* vystavuje nebezpečí útoku a prolomení zabezpečení. Proto je potřeba v aplikaci na zabezpečení myslet a v žádném případě nespoléhat pouze na sílu zabezpečení knihoven třetích stran, používaných v aplikaci.

Tzn. je nutné zabezpečit *API*, databázi a další služby, které aplikace využívá. Je nutné také zvážit např. jaké služby je potřebné mít veřejně dostupné, a ke zbytku služeb zamezit přístup z internetu pomocí **firewallu**. K ostatním službám, ke kterým je potřeba mít soukromý přístup, potom přistupovat pomocí tzv. **SSH tunelu**, ideálně za použití **SSH klíče**, nikoliv uživatelského jména a hesla. Navíc je nutné udržovat systémy aktuální s nejnovějšími bezpečnostními aktualizacemi [32, 16].

3.26.1 Hashování

Hashování je proces, při kterém se ze zadaných dat použitím určitého algoritmu vytvoří řetězec jiný, tak, že z něj nelze získat původní data. To se používá např. při ukládání hesel, kdy heslo uživatele není uloženo v původní čitelné podobě, ale jako **hash**. Při přihlášení pak stejnou *hashovací* funkcí zadané heslo ověří s uloženým *hashem*, jelikož *hashovací funkce* pro dva stejné vstupy vrací totožné výsledné *hashe*. Pokud se shodují, uživatel byl úspěšně ověřen a tedy přihlášen.

Pokud se někomu podaří z *hashe* získat původní data (heslo), nebo dokáže najít způsob, jak z různých dat vygenerovat stejný *hash*, považuje se algoritmus za prolomený a měl by se přestat používat.

K *hashování* hesel se navíc používá tzv. **salt (sůl)**, což je náhodně vygenerovaný řetězec uložený s heslem (ve formě *hashe*) v databázi, který se přidává k heslu před *hashováním*. Protože pro každou kombinaci by měl algoritmus generovat jiný *hash*, je takto ztíženo zjištění původního hesla i v případě úniku *hashe* na veřejnost [32].

3.26.2 Šifrování

Při **šifrování** je naopak potřeba získat původní data z **šifrovaného textu** (**ciphertext**). K šifrovacímu algoritmu je potřeba vygenerovat klíč, kterým se šifrují i dešifrují data. Algoritmus za pomocí klíče data zašifruje. Při dešifrování týž algoritmus pomocí klíče *šifrovaný text* dešifruje do původní podoby.

Z hlediska zabezpečení webové aplikace pak existují dva základní procesy. *Autentizace* a *autorizace* [32].

3.26.3 Autentizace

Autentizace je proces ověření identity uživatele. To znamená, že uživatel musí dokázat, že je tím, za koho se vydává. To je možné udělat pomocí hesla, certifikátu, který uživatel má, nebo pomocí externí služby, která identitu ověří a potvrdí ji.

V případě *API* se jedná o ověření, že požadavek opravdu přišel z dané aplikace, a nikoliv z jiného nepovoleného zdroje, např. od útočníka. Do aplikace je přidán tajný klíč, který slouží k identifikaci aplikace a při každém požadavku na *BE* si ověřuje *autenticitu* požadavku. Na to existují různé metody, např. využití *JWT*.

Do *FE* aplikace je obvykle vložena dvojice tajných klíčů, které *BE* řeknou, že jde skutečně o autorizovanou aplikaci, která má k *BE* povolený přístup. Také je možné pro větší bezpečnost přístup omezit pouze na konkrétní IP adresy, ze kterých je možné se připojit a *URI*, ze které lze požadavek odeslat.

Při *autentizaci* aplikace je důležité kontrolovat také, zda byl požadavek vytvořen povolenou aplikací a další informace, které by mohly odhalit nepovolený přístup. Při každém požadavku je pak potřeba ověřit, že *token* nebyl změněn a přichází stále ze stejné aplikace a stejné *IP adresy*, jako při první *autentizaci* [32].

3.26.4 Autorizace

Autorizace je proces ověření oprávnění uživatele k manipulaci s konkrétním zdrojem. Ověřujeme, zda má uživatel oprávnění k provedení určité akce, např. úprava, resp. smazání příkladu. Je nutné vzít v potaz *uživatelskou roli* (tj. administrátor, uživatel, návštěvník...), kterou uživatel má ve vztahu k aplikaci a k danému zdroji.

Je potřeba zkontrolovat vlastnictví nebo přidělená oprávnění („může uživatel článek jen zobrazit nebo i upravovat?“). Např. pokud je uživatel autorem příkladu, může s ním libovolně manipulovat. Pokud příklad vytvořil někdo jiný, musí k úpravě dostat oprávnění [32].

3.26.5 JSON Web Token

Hojně používaná metoda zabezpečení je **JSON Web Token (JWT)**. Jedná se o formát **textového řetězce** (tzv. **tokenu**), který obsahuje informace o uživateli a aktivním stavu aplikace ve vztahu k němu. *Token* se při každém požadavku posílá v hlavičce požadavku a umožňuje tak požadavek *autorizovat*.

Při *autentizaci* je vygenerován *token* (řetězec znaků), který se při každém požadavku na server posílá v hlavičce požadavku. Data v tokenu jsou kódována ve formátu **base64** a podepsána šifrovacím klíčem, takže je nelze, bez znalosti klíče, změnit, protože jakákoliv změna v datech znamená změnu *podpisu*. Do tokenu se přidávají data, která je potřeba uchovat nebo jsou využita k dodatečné kontrole. Je důležité do *tokenu* neukládat žádná citlivá data, jelikož formát *base64* není šifrovaný.

V případě změny nebo nepřesnosti v datech *tokenu* se díky podpisu *token* stává neplatným, což umožňuje ověřit, že byl token původně vytvořen *serverem*, a ne někým jiným. Zároveň je možné mít v datech *tokenu* uloženy informace o aktuálně přihlášeném uživateli a další kontrolní informace, jako délka platnosti *tokenu*, *IP adresa požadavku*, *typ zařízení* a *doména*, ze které byl požadavek na jeho vytvoření prvně odeslán.

JWT sestává ze tří částí, které jsou odděleny tečkou.

Hlavička (header) obsahuje informace o typu tokenu a typu šifrování podpisu.

Tělo tokenu (body) obsahuje data, která je potřeba uchovat (*payload*).

Podpis (signature) zajišťuje kontrolu, že token nebyl během cesty na server změněn.

Všechny tři části jsou zakódovány pomocí metody *Base64* a odděleny tečkou. Výsledný token je také textový řetězec, který je možné přenášet v hlavičce *HTTP požadavku.*

Funkci *JWT* v praxi lze vidět na schématu 3.3.

Obrázek 3.3 Použití JWT v praxi

Jednou z výhod *JWT* je možnost přenositelnosti. Token může být vytvořen na jednom serveru a ověřen na jiném, který má k dispozici daný klíč ([4]). Je možné tak např. umožnit jednotné přihlášení v rámci několika služeb. *JWT* není vázán na symetrické šifrování a kvůli přenositelnosti se často využívá *asymetrické šifrování*, kde hlavní server má tzv. **privátní klíč** (**private key**) a další účastnící mají svůj vlastní **veřejný klíč** (**public key**), odvozený od privátního, kterým ověřují, že byl token vytvořen na správném serveru [33].

Díky *asymetrickému šifrování* je možné vytvořit více *veřejných klíčů* a přidělit je různým službám, které je pak mohou používat pro ověření *tokenu* z původního *serveru*.

Formát *Base64* je kódování, které se používá pro přenos dat, nikoliv pro zabezpečení. Bezpečnost *JWT* je zajištěna pouze podpisem, který zaručuje integritu dat. Data ale může přečíst kdokoliv, kdo získá přístup k tokenu. Proto je důležité do tokenu ukládat pouze data, která jsou nezbytně nutná a neobsahují citlivé informace. K šifrování dat a ostatních částí *JWT* vznikly navazující standardy – **JWE**, **JWS**, **JWA** nebo **JWK** [25, 32].

3.26.6 Structured Query Language

Structured Query Language (SQL) je jazyk, používaný pro manipulaci s daty uloženými v rámci relačních databází. *SQL* je deklarativní jazyk, což znamená, že dotaz popisuje, jaká data je potřeba získat, a ne jakým způsobem. *SQL* je standardizovaný jazyk, který je podporován většinou relačních databází. V jazyce *SQL* jsou zapisovány tzv. **SQL dotazy** (**SQL Query**), které mohou ukládat, načítat, upravovat, nebo mazat data v databázi [50].

3.26.7 SQL Injection

SQL Injection je zřejmě nejčastější typ útoku, při kterém útočník vloží kus *SQL dotazu* do neošetřeného uživatelského vstupu (např. vstupní pole pro email) a pokud jej autor aplikace neošetří, je místo zpracování vykonán společně se zbytkem dotazu. To může vést k získání citlivých dat, jejich změně nebo smazání. Může např. dojít ke změně hesla administrátorského účtu a tím získání přístupu ke chráněným datům.

Proto je nutné data od uživatele vždy ošetřit a v ideálním případě použít navíc tzv. *Parametrizované dotazy* (*Prepared statements*). Při použití *parametrizovaných dotazů* se do dotazu nevkládají přímo data, ale na místo, kam patří data je vložený tzv. *Zástupný symbol* (*placeholder*), který je poté nahrazen ošetřenými (*escapovanými*) daty tak, aby nebylo možné případný vložený příkaz provést. Zástupný symbol v *SQL* představuje otazník „`?`“ nebo v případě pojmenovaných hodnot název proměnné za dvojtečkou „`:nazev`“ [32].

Např. dotaz na změnu emailové adresy, který by v případě neošetření umožnil útočníkovi získat administrátorský přístup do aplikace, by mohl vypadat, jako v ukázce 3.4.:

```
id := 42
// nebezpečný vstup od uživatele
email := "mujmejl@seznam.cz', role='admin"
// přímé spuštění dotazu
DB.Exec("UPDATE users SET email = '" + email + "'
↪  WHERE id = " + id)

-- výsledný spuštěný dotaz, který by umožnil
↪  útočníkovi získat administrátorský přístup
UPDATE users SET email = 'mujmejl@seznam.cz',
↪  role='admin' WHERE id = 42
```

(a) Neošetřený řetězec vložený do *SQL*

```
id := 42
// nebezpečný vstup od uživatele
email := "mujmejl@seznam.cz', role='admin"
// příprava dotazu pomocí "prepared statements"
DB.Exec("UPDATE users SET email = ? WHERE id = ?",
↪  email, id)

-- ošetřený (escapovaný) dotaz, který by pouze
↪  vložil do databáze nevalidní emailovou adresu
UPDATE users SET email = 'mujmejl@seznam.cz\',
↪  role=\'admin', WHERE id = 42'
```

(b) Ošetření SQL injection pomocí prepared statement

Kód 3.4 *SQL* injection a jeho ošetření

Samozřejmě v příkladu není vstup nijak ošetřen a k přijetí takové hodnoty do pole `email` by nemělo vůbec dojít, jelikož nejde o validní emailovou adresu. Vstupní data je potřeba v aplikaci ošetřovat nejméně na dvou místech. Všechna vstupní data je tedy nutné ošetřit v rámci *FE* i *BE* a při *SQL* dotazech využívat parametrizované dotazy. Je tedy potřeba kontrolu provést před odesláním na straně uživatele, nicméně na to nelze spoléhat, jelikož to lze jednoduše obejít. Kontrola na straně uživatele je především kvůli informaci pro uživatele o správnosti a pokynech pro zadání správných dat uživatelem. Podrobná kontrola probíhá po přijetí dat na serveru.

Další kontroly vstupních dat mohou probíhat i na úrovni databáze, např. při indexaci. To už ale je na uvážení autora konkrétní aplikace [75, 38].

3.26.8 Cross Site Scripting

Cross Site Scripting (XSS) je typ útoku, při kterém útočník vloží do uživatelského vstupu kód, který se při načtení stránky spustí. To může vést k získání citlivých dat, přesměrování na jinou stránku nebo k jiným nežádoucím akcím.

Možnou ochranou před tímto typem útoku je nahrazení znaků, které mohou způsobit problémy, za jejich *HTML entity*, kde je potřeba např. znaky jako `<` nahradit za `<`, `>` za `>`, `&` za `&` apod. Takové *HTML entity* prohlížeč vykreslí uživateli a nezpracovává je jako funkční kód.

Možností je také využít specializované knihovny, které toto ošetření provedou samy, a není nutné řešit otázky, které znaky je nebo není potřeba ošetřovat, např. *DOMPurify* nebo *sanitize-html* [102].

Jelikož v projektu používám *React*, který využívá *JSX*, je tento typ útoku téměř vyloučen. *JSX* totiž neumožňuje vkládat přímo *HTML* kód, ale pouze textové řetězce, které se při načtení stránky zobrazí.

Existují ale výjimky, a to použití funkce `eval`, která umožňuje spustit *JS* kód z řetězce (textové proměnné), a přímé vkládání neošetřeného vstupu pomocí vlastnosti `dangerouslySetInnerHTML` nebo jí podobných, které se přímo přepíšou do kódu, např. `href`, `OnClick` apod. Tyto vlastnosti ale vývojáři používají s vědomím rizika [53].

3.27 42

Číslo „**42**“ ([2]) někteří vývojáři používají jako nenápadný odkaz na zábavní rádiový pořad a knihu Stopařův průvodce po galaxii britského autora Douglase Adamse, vydávaný od r. 1979. Jde o způsob, jak lidé z *IT* a jiných oborů vyjadřují svůj smysl pro humor [49].

42 je číslo, které v příběhu dal velký počítač **Hlubina myšlení** (**Deep thought**) po dlouhém počítání na otázku "života, vesmíru a vůbec", myšleno na jakoukoliv otázku ve vesmíru. Vtipně se tak vyskytuje v různých podobách, jako vševříkající poznámka k dané problematice. V *IT* např. v komentářích, dokumentacích, nebo je přímo použito v kódu.

V různých oborech má však *42* reálný význam. Existují studie, zamýšlející se nad významem a důkazy reálného významu tohoto čísla. Jde např. o jedno z nejdéle řešených parametrů v rozsahu od 0 do 100 **kubické diofantovské rovnice** $x^3 + y^3 + z^3 = k$ [85], který počítače po celém světě řešily přes milion

hodin. To je v kontextu knihy zajímavé, jelikož v knize superpočítač *Hlubina myšlení* odpovídá na otázku neúměrně dlouho, konkrétně `7,5 milionu let` , a odpoví číslem *42* [2].

Další zajímavou souvislostí je, že v ***ASCII*** tabulce (viz `https://www.ascii-code.com/`), je znak s kódem *42* hvězdička (asterisk). V různých implementacích nahrazuje jeden libovolný znak (tzv. (wildcard)) nebo v **regulárních výrazech** jde o tzv. **kvantifikátor** označující libovolné množství z určené množiny znaků.

Číslo *42* je také součástí několika významných algoritmů, např. **SHA-1** nebo **MD5**, které se používají pro *hashování* hesel nebo *kontrolní součet* souborů. V *ASCII* tabulce je *42* znak **asterisk** (*), který se používá jako *wildcard* v *regulárních výrazech* nebo ve *fulltextovém* vyhledávání [28, 14, 76].

V aplikaci jsem se rozhodl věnovat číslu *42* samostatný *endpoint* `/theanswer`, jehož účelem je v *JSON* odpovědi vrátit odpověď na otázku "života, vesmíru a vůbec".

3.28 Testování

Testování je důležitou součástí vývoje aplikace a černé svědomí velké části vývojářů. Testování je časově náročné a vývojáři se mu často vyhýbají, protože je potřeba vytvořit sadu testů, které pokryjí co nejvíce možných scénářů, které mohou nastat.

Zejména automatickým testováním je možné ušetřit spoustu času, který bychom jinak strávili ručním testováním aplikace nebo při hledání chyb, které se objevily po změnách v kódu aplikace. Automatické testování umožňuje vytvořit sadu testů, automaticky spouštěných při každé změně v kódu a ověřit tak, že aplikace stále funguje správně. Pokud je daný jazyk navíc *dynamicky typovaný*, je dobré mít nástroj na kontrolu správnosti použití funkcí a proměnných [92, 34].

Např. při změnách základní funkcionality je dobré otestovat, že i po změnách aplikace funguje stále stejně. Zejména jsou testovány tzv. **čisté funkce** (**pure function**), což jsou funkce, které nemají žádné **vedlejší efekty** (přímo neovlivňují výstupní data a stav programu) a pro stejný vstup vždy poskytnou (vrací) stejný výsledek. Takové testování se nazývá *unit testing*.

V každém programovacím jazyce jsou potřeba nástroje pro automatické testování. V případě *JS* to jsou např. **Jest** nebo **Mocha**. Jazyk *Go* má vestavěný nástroj pro automatické testování, který se nazývá **go test**.

Pro testování React aplikace budu používat *Jest*. Pro testování *BE* potom zmíněný *Go test* [7].

3.28.1 Testování API

Pro testování *API* existují nástroje, které umožní vytvářet požadavky bez nutnosti mít dokončenu *FE* část. Mezi nejznámější patří **Postman** nebo **Insomnia**.

Tyto nástroje navíc většinou nabízejí možnost vytvořit si sadu automatických testů, které je možné spouštět při sestavování aplikace v *Dockeru* a automaticky si tak před kompilací celého programu ověřit, že jednotlivé součásti *API* stále funguje správně, např. jsou dostupné všechny *endpointy*.

K tomuto účelu použiji nástroj *Postman*, jelikož v době psaní této práce nástroj *Insomnia* neumožňoval psaní a automatické spouštění testovacích `post-request` skriptů.

S nastaveným testovacím skriptem si ukládám nový *token* a při následujícím požadavku jej posílám v hlavičce požadavku. Tímto způsobem mohu testovat *BE* bez omezení bezpečnostních opatření.

3.28.2 Testování frontendu

Pro testování *FE* budu používat knihovnu *Jest*, kterou na testování používá např. *Facebook*. Testovat bude potřeba *čisté funkce*, ale také *komponenty*. *Komponenty* se testují pomocí tzv. **snapshotů**, které porovnají vykreslenou komponentu s předchozí vykreslenou verzí. Pokud se liší, test selže a je potřeba zkontrolovat, zda je změna v pořádku nebo *snapshot* obnovit v nové podobě *komponenty* [7].

3.29 Continous Integration a Continuous Delivery

CI/CD je souhrnný pojem, který označuje průběžnou integraci **Continuous Integration (CI)** a průběžné doručení nebo nasazování **Continuous Deployment/Delivery (CD)**, dvě klíčové praktiky v automatizaci procesů vývoje softwaru.

CI zahrnuje automatické spouštění sady testů po každé změně kódu v **GIT** repozitáři, aby byla ověřena jeho funkčnost a kompatibilita. Cílem je rychle identifikovat a opravit chyby, což vede k udržení kvality

softwaru. Po úspěšném provedení testů dochází k automatickému sestavení aplikace. Pokud testy selžou, proces informuje vývojáře o potřebě oprav a zastaví další postup.

CD rozšiřuje *CI* tím, že po úspěšném sestavení aplikace automaticky provede její nasazení na server. To umožňuje rychlou dostupnost aktualizované aplikace uživatelům bez manuálního zásahu, což zvyšuje efektivitu vývojového procesu a umožňuje rychlejší reakci na požadavky trhu [32].

3.30 Verzovací systém GIT

GIT vytvořil autor Linuxu *Linus Torvalds* v roce 2005. Je to **distribuovaný verzovací systém**. Každý vývojář má lokální kopii celého projektu a může pracovat nezávisle na ostatních i bez připojení k internetu. Po dokončení práce své změny nahraje na server, kde proběhne kontrola a změny jsou zahrnuty (**merge**) do hlavní větve (**branch**) projektu.

Základní funkcí *GITu* je ukládání změn. Neukládá celé soubory, pouze změny v kódu. Díky tomu je možné procházet historií změn a vracet se k různým verzím kódu. To umožňuje spravovat větší projekty, na kterých paralelně spolupracuje více vývojářů zároveň, a po dokončení části práce a její kontrole výsledek připojit ke zbytku kódu. Potom je možné označit schválenou a otestovanou verzi kódu jako finální verzi (tzv. **release**) nebo je na kódu možné dále pracovat.

Je také možné pracovat na několika různých problémech zároveň díky **větvím** nebo na různé verze možné v jakémkoliv odkazovat pomocí tagů a tím umožnit používat označené místo ve vývoji např. číslem verze software.

Existují služby, které poskytují prostor pro ukládání *GIT* projektů online, tak aby byly zálohované a přístupné odkudkoliv, jako např. **GitHub**, **GitLab** nebo **Bitbucket**. Mimo zmíněné funkce jsou zde tzv. **Issues**, které umožňují sledovat stav jednotlivých právě probíhající úkoly a hlásit chyby k opravě. Tímto způsobem může (pokud je *repozitář* veřejný) např. uživatel sdělit svůj nápad nebo ohlásit chybu v aplikaci a případně se k ní vrátit později. *Issues* je také možné přiřadit konkrétnímu vývojáři, který chybu nebo návrh na vylepšení dostane za úkol, a automaticky vytvořit **větev** (**branch**), na které bude práce probíhat. Na konci práce vývojář vytvoří tzv. **merge request**, ve kterém okomentuje, jak problém řešil, a požádá o sloučení nového kódu do hlavní *větve* projektu. Je také možné jednoduše zobrazit rozdíly mezi hlavní *větví* a tou, na které vývojář pracuje [88].

GIT je pro vývoj dnešních aplikací nepostradatelný nástroj.

Gitlab je open–source nástroj pro správu a udržování *GIT repozitářů* na serveru. Podobně jako *Github* nabízí možnosti, jak kompletně spravovat projekty. Kromě toho je často používán k automatizaci testování a publikaci aplikace na produkční server (*CI/CD pipeline*, viz sekce o *testování aplikací*). Je možné jej nainstalovat na vlastní server a mít tak plnou kontrolu nad svými daty, čehož využívá spousta společností, pokud např. mají více projektů nebo nechtějí svůj kód ukládat na cizích serverech [70, 12, 30].

3.30.1 Git hooks

Git hooks jsou skripty, které se spouští při určité události v *Gitu*. Je možné např. automaticky spustit testy před každým nahráním změn v kódu do repozitáře [88].

3.30.2 Gitlab Actions

Gitlab Actions jsou součástí *Gitlabu* a umožňují vytvořit sadu procesů (tzv. **workflow**), které se spustí při určité události, např. při nahrání nového kódu do repozitáře.

Workflow se spouští ve virtualizačním prostředí (*kontejneru*), který je vytvořený na základě *Dockerfile* a obsahuje prostředí, potřebné pro sestavení a spuštění aplikace a testů. Celý proces se automaticky spouští na *virtuálním serveru*, např. po nahrání projektu na *Gitlab*, a nastavuje se pomocí konfiguračního souboru *.yml*.

Během procesu je sestaven *kontejner*, ve kterém se spustí přednastavené testy a poté je aplikace sestavena. Pokud vše proběhne v pořádku, aplikace je automaticky nasazena na produkční server. Použitím *CI/CD* procesů je zjednodušeno nasazování aplikace a zrychluje se tak proces vývoje [30].

3.31 Pravidla přístupnosti webu

Webové aplikace je nutné vytvářet přístupné pro všechny uživatele. Tato pravidla jsou definována v **Web Content Accessibility Guidelines (WCAG)**, které jsou vydávány organizací **World Wide Web Consortium (W3C)**.

V ČR je přístupnost webu upravena zákonem č. 99/2019 Sb., o přístupnosti internetových stránek a mobilních aplikací veřejného sektoru a o změně zákona č. 365/2000 Sb., o informačních systémech veřejné správy a o změně některých dalších zákonů (zákon o přístupnosti).

Pravidla přístupnosti zahrnují optimalizaci (přizpůsobení) aplikace pro hladké použití aplikace osobami se zrakovými nebo sluchovými vadami, poruchami pohybu ap. Mají např. zjednodušit navigaci ve stránce pomocí klávesnice, či s použitím programů na čtení displeje, zmiňují kontrast prvků ve stránce pro lepší čitelnost a správné použití **semantických** (**významových**) prvků *HTML* jako `<nav>`, `<header>`, `<article>`, správně použité úrovně nadpisů a další [100].

3.32 Optimalizace pro vyhledávače

Důležitou součástí, která souvisí i s přístupností, je **optimalizace webu pro vyhledávače (SEO)**. Při **indexaci** (načtení strukturovaných dat ze stránky a jejich zařazení do vyhledávání). Vyhledávač musí – stejně jako např. software pro čtení obrazovky – načítat surová data, která si ukládá do své databáze. V případě, že jsou data špatně strukturovaná, nebo nejsou dostatečně popsaná, může dojít k jejich špatnému zařazení, nebo k jejich nezařazení do vyhledávání. To může mít za následek, že stránka nebude v hledání nalezena, nebo bude nalezena až na pozdějších pozicích [17].

Důležitými faktory jsou také zabezpečení webu (*HTTPS*) a optimalizace pro mobilní zařízení, nebo správná struktura webu, např. oddělení navigace od obsahu (`<nav><article>` ap.), nebo hiearchie nadpisů. Vyhledávače (i sociální sítě ap.), např. *Google* bodují splnění různých kategorií a podle vlastního algoritmu potom řadí výsledky ve vyhledávání. Viz aktualizace algoritmu [37].

3.33 Údržba aplikace

Údržba aplikace je důležitá část vývoje. Je potřeba aplikaci pravidelně aktualizovat a opravovat chyby, které se vyskytnou. Tedy jde o nejdelší část životního cyklu aplikace. V této části je potřeba aplikaci testovat a opravovat chyby, které se vyskytnou.

3.33.1 Životní cyklus vývoje software

Software Development Lifecycle (SDLC) je označení pro životní cyklus software. Jedná se o soubor procesů, které se používají při vývoji softwaru. Tyto procesy se opakují v kolech, které se nazývají iterace. V každé iteraci se vyvíjí část aplikace, která je následně testována. Vývojáři se také mohou v každé iteraci vrátit k předchozí části a upravit ji podle potřeby, případně do vývoje zapojit zákazníka.

Ve většině případů jde o opakující se proces, který se opakuje až do dokončení aplikace. Různé týmy mohou mít různé požadavky na vývoj aplikace a používat jiné modely (metodiky) vývoje, kterých existuje několik.

Nejčastěji je používán **Agilní** (**Agile**) model, který je založen na iteracích a pravidelné komunikaci se zákazníkem. V každé iteraci se vyvíjí část aplikace, která je následně testována.

Existují ale i jiné modely vývoje, jako např. Vodopádový, W-model a další [44].

3.33.2 Sledování chyb nahlášených uživateli

Pokud uživatel narazí na chyby v aplikaci, měl by je mít možnost nahlásit. Vývojář má potom lepší možnost tyto chyby sledovat a opravit je.

Podobný je i případ požadavků na nové funkce. Uživatel by měl mít možnost požádat o novou funkci nebo zlepšení stávajících, aby vývojář mohl aplikaci dále rozvíjet.

3.33.3 Běhové a kompilační chyby

Kompilace (**sestavení**) je proces překladu zdrojového kódu do binárního souboru, který je přímo spustitelný na procesoru. Při kompilaci se kontroluje správnost kódu, datových typů ap. Případné chyby překladač oznámí vývojáři. Některé chyby může překladač opravit automaticky a podle závažnosti chyby a zavedených parametrů překlad proběhne v pořádku nebo se překlad zastaví a je potřeba chyby předem opravit.

Chyby se ale mohou projevit až při běhu programu. To jsou chyby, které např. závisí na datech, která se do programu načítají nebo je zadává uživatel. Příkladem takové chyby může být špatná práce s pamětí, která může vést k přetečení paměti a následnému pádu programu.

Na běhových chybách se docela často zakládají útoky na aplikace, proto je potřeba je hlídat. Útočník může záměrně zasílat data, která způsobí chybu v aplikaci, čímž může získat přístup k systému nebo získat data, ke kterým by jinak přístup neměl.

Důležité pro opravní chyb je uvědomit si jejich příčinu. To je často složité, protože chyba se může projevit až po několika krocích nebo pouze za specifického stavu aplikace [32].

Proto je nutné aplikaci testovat, chyby zaznamenávat i se vstupními daty a opravovat je. To je důležitá část vývoje aplikace. Jeden z nástrojů, které efektivní sledování chyb umožňují, je *Sentry*.

3.33.4 Sentry

Sentry je nástroj, který umožňuje zachytávat chyby za běhu aplikace. Pomocí webového rozhraní potom může vývojář (nebo jeho tým) sledovat chyby, které se vyskytly za chodu aplikace. *Sentry* umožňuje sledovat chyby v různých jazycích, jako v tomto případě *JS* a *Go*. Ale funguje i v dalších jazycích. Existuje také modul pro integraci s frameworkem *Gin*, což umožňuje získat i informace o *HTTP* požadavku a stavu serveru, který chybě předcházel.

V každém z těchto jazyků umožní *Sentry* sledovat chyby specifické pro daný jazyk a tím zjednodušit opravu chyb. Např. v případě *JS* umožní sledovat akce, které uživatel provedl před tím, než došlo k chybě. Chyby z *BE* i *FE* aplikace se uloží na jedno místo a je tak pro celý tým jednodušší s nimi pracovat.

Díky tomu má vývojář k dispozici spoustu informací, které potřebuje k nalezení, vyvolání a opravě chyby. *Sentry* také umožňuje přímo vytvářet *úkoly*, na kterých mohou vývojáři pracovat. To umožňuje vývojářům efektivněji rozdělovat práci na opravách chyb, které *Sentry* zachytí.

Informace navíc zůstanou uloženy v *Sentry* a vývojáři se k nim mohou kdykoliv vrátit. *Sentry* je možno integrovat do různých komunikačních nástrojů, jako např. **Slack**, **Telegram**, **Whatsapp** nebo **Discord**, které se běžně používají ke komunikaci ve vývojových týmech, a upomínka v případě chyby se objeví přímo v komunikačním kanálu týmu.

3.34 Linux a příkazová řádka, Windows Subsystem for Linux

Linux je operační systém, který vytvořil Linus Torvalds v roce 1991. Jde o nejrozšířenější *open–source* operační systém, používaný zejména na serverech. Je možné jej ale jednoduše použít také na *desktopových zařízeních* v podobě mnoha distribucí (např. Ubuntu, Linux Mint, . . .)

Je to také základ pro mnoho dalších operačních systémů, jako např. *Android open–source project (*AOSP*)* od společnosti *Google*, který na *jádro* **OS** *Linux* přidává funkce pro použití na mobilních zařízeních [101].

Velikou výhodou je otevřenost a možnost upravovat zdrojový kód. To umožňuje vývojářům nahlížet a navrhovat změny ke zlepšení do jádra *Linuxu.* Výhodou je také jeho bezpečnost a svoboda. V porovnání s jinými operačními systémy je bezpečnější a méně náchylný ke škodlivým programům, zejména díky pravidlům pro přístupy k souborům. V Linuxu je navíc za soubor považováno všechno (konfigurace, procesy, síť i periférie atd.), což zabezpečení zjednodušuje - pokud jsou správně nastavena přístupová oprávnění a vlastnictví [52].

Nabízí mnoho grafických prostředí, takže si uživatel může vybrat, jakým způsobem bude systém používat, a je možné téměř všechno nastavit tak, jak v práci potřebuje. Navíc většina nástrojů pro vývoj aplikací je vyvíjena pro Linux, takže je možné je používat bez problémů a automaticky vše nastavit jedním skriptem z příkazové řádky. Není také nutné instalovat a mít zároveň spuštěno několik různých nástrojových oken, vše je na jednom místě.

Kvůli zvyšujícímu se počtu vývojářů pracujících na Linuxu přidala firma Microsoft do svého operačního systému Windows možnost spouštět aplikace a příkazy Linuxu pomocí nástroje **WSL2 (Windows Subsystem for Linux)**. Jedná se o virtuální stroj, který běží na pozadí a umožní uživateli Windows používat

výhody, které nabízí Linux, např. má vývojové nástroje na jednom místě, místo několika oken, které musí mít otevřené při normálním vývoji na Windows [97].

3.35 Integrated Development Environment a nástroje pro psaní kódu

Integrated Development Environment (IDE) (**integrované vývojové prostředí**) jsou programy, které umožňují vývojářům psát spouštět a ladit (tzv. **debugovat**) ji v jednom prostředí. To umožňuje vývojářům pracovat efektivněji a rychleji. V jednom editoru mají všechny nástroje, které potřebují k vývoji aplikace. V dnešní době existuje spousta *IDE*, které se liší podporovanými jazyky, funkcemi a vzhledem. Funkcionalitu lze většinou doplňovat i o nové funkce pomocí pluginů.

Mezi nejznámější *IDE* patří *Visual Studio Code*, *IntelliJ IDEA*, *Google Atom*, *Netbeans* nebo *Sublime Text*. Jako *IDE* je možné při vhodném nastavení označit také konzolový editor *Vim* [84].

S pokrokem *umělé inteligence* (*AI*) se objevují nástroje, které dokáží automaticky opravovat chyby v kódu, nebo dokonce přímo na základě formálního popisu běžným jazykem psát kód za vývojáře, viz [71]. To ale zatím není možné v každém jazyce a je nutné kód osobně zkontrolovat, protože generativní nástroje mohou vytvořit kód, který je sice funkční, ale neefektivní nebo nečitelný.

Příkladem nástroje generování kódu může být např. *Github Copilot*, který dokáže na základě komentářů nebo kontextu v kódu vygenerovat funkční pokračování kódu. Další nástroje jsou např. *Tabnine*, *Kite* a další [47].

V průzkumu „The State Of AI Tools 2023“ ([65]) většina vývojářů odpověděla, že se nebojí toho, že by je *AI* v blízké budoucnosti plně nahradila. Programování je komplexní proces, který vyžaduje kreativitu a schopnost řešit problémy různého typu. Využití kódu od *AI* je nutné brát s rezervou – učí se z kódu, který vytvořili lidi, takže může být ovlivněný jejich nedostatky a *AI* prozatím není schopná samostatně přemýšlet nad řešením problému, ale pouze generovat data na základě předchozích řešení, která mohou být chybná či neefektivní.

Jazykové modely *AI*, jako *ChatGPT* či *Google Bard* potom pomáhají zejména ke konzultaci problémů a získání nápadů, jak je vyřešit. Svým způsobem tím suplují síť webových fór *Stack Overflow*, kde si vývojáři navzájem radí s řešením problémů, zejména z oboru *IT*. Tyto *AI* nástroje jsou skvělé na opakované úkoly, jako přepisování struktur dat nebo hromadných příkazů.

AI nástroje typu *ChatGPT* ale poslední dobou ztrácí na přesnosti. Je to díky tomu, že k dalšímu učení využívají data přístupná z internetu, a to často i data generovaná jinými *AI*. Výsledky je tedy nutné posuzovat kriticky [81].

3.36 Mezinárodní lokalizace

Internationalization (I18n) je proces přizpůsobení aplikace pro různé jazyky a kultury. To zahrnuje mimo překlad např. formát data, měny nebo jednotky. Standard *I18n* zahrnuje i různé formy slov, např. pro různá množství [99].

Každý znak je v počítači reprezentován číslem. V minulosti používaly počítače k ukládání znaků kódování *ASCII*, které (z historických důvodů) obsahovalo pouze 128 znaků (7 bitů v rozsahu 0 až $2^7 - 1$). To znamená, že bylo možné zobrazit pouze anglické texty a některá interpunkční znaménka.

V dnešní době se většinově používá kódování **Unicode (UTF)**, které k původním 7 bitům (*ASCII*) přidává další, značící, zda daný znak pokračuje dalším **bajtem**. To umožňuje zobrazit mnohem více znaků, vč. např. čínských znaků nebo smajlíků, a tedy překlad aplikace do různých jazyků a znakových sad. Na druhou stranu každý další byte znamená více použité paměti, kterou text zabírá a počítač také pracuje déle na zpracování každého znaku.

Proto je vhodné všechny zdrojové kódy a aplikace psát v Anglickém jazyce, který využívá základní *ASCII* znaky, a aplikaci pro uživatele do jeho jazyka překládat (**lokalizovat**). Anglický jazyk je navíc považován za mezinárodní jazyk, který je známý většině uživatelů, a aplikace je tak jednodušeji přístupná širší skupině uživatelů.

Webový prohlížeč si pamatuje preferenci jazyka uživatele a při požadavku na aplikaci zašle i informaci o preferovaném jazyce. Webový server pak zpřístupní uživateli aplikaci ve vyžádaném jazyce.

Pro překlad aplikace se používají *lokalizační soubory*, které obsahují překlady jednotlivých textů aplikace. Každý jazyk má svůj soubor, který se načítá při spouštění aplikace. Pokud překlad není nalezen, použije se původní text.

V rámci *BE* aplikace je potřeba přeložit zejména e-mailové zprávy a hlášky, které budou odesílány uživatelům. Na *FE* pak celé *uživatelské rozhraní (*UI*)*, které bude uživatel používat.

3.37 Unified Modeling Language

Unified Modeling Language (UML) je standardizovaný grafický jazyk pro modelování softwarových systémů. První verze byla vydána v roce 1997 a od té doby se stala standardem pro modelování softwarových systémů. Základem *UML* je *UML diagram*, který slouží k vizualizaci návrhu aplikace. Výhoda použití *UML* je mmj. v tom, že k modelování není použitý konkrétní jazyk, takže jej může vytvořit *softwarový návrhář* a *vývojář* následně aplikaci implementuje podle požadavků z návrhu.

Specifikace definuje základní dva typy diagramů:

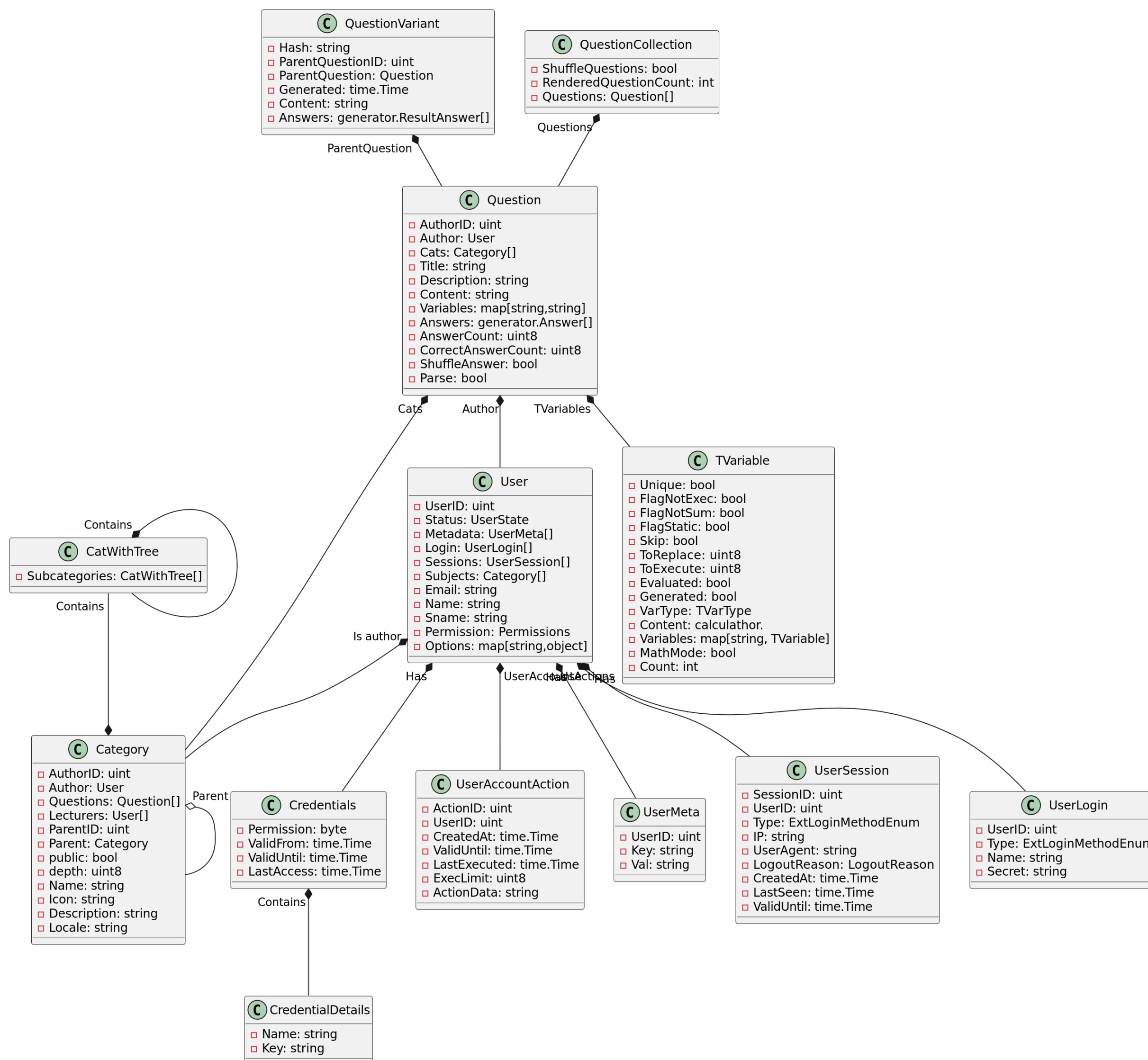
Strukturální diagramy popisují strukturu systému, tedy jeho části a vztahy mezi nimi. Může jít např. o diagramy tříd, objektů, komponent, nebo komunikace mezi komponentami ap.

Diagramy chování popisují chování systému, tedy jak se jeho části chovají a jak spolu komunikují. Např. diagramy aktivit, stavový diagram ap.

Aplikace je možné modelovat pomocí *UML* diagramů, ale není to nutné. Taktéž každá aplikace může mít více *UML* reprezentací, podle toho, jakým způsobem se vývojář rozhodne aplikaci navrhnout.

K naplánování této práce jsem vybral **diagram případů užití** (**Use Case Diagram**) a **diagram tříd** (**Class Diagram**) ze standardu *UML*.

Je důležité zmínit, že neexistují předem dané postupy, jak diagramy tvořit a co vše do nich zahrnout, pouze standardizovaný způsob jejich struktury a významů. Způsob, jakým vývojář funkcionalitu modeluje, je čistě na jeho osobním zvážení, dokud se drží definice *UML*. Je ale důležité, aby svá rozhodnutí zdůvodnil a průběžně dokumentoval [27, 91].



Obrázek 3.4 Diagram tříd aplikace

3.37.1 Use Case Diagram

Use Case Diagram (UCD) je diagram ze standardu *UML*, který popisuje chování systému z pohledu uživatele, tedy interakci systému s uživatelem. Jednotlivým aktérům jsou přiřazeny případy užití, které popisují, jaké akce mohou jednotliví aktéři používat.

To umožní lépe si představit, jak bude aplikace fungovat, jaké funkce bude mít a jaké *endpointy* bude potřeba vytvořit, aby bylo možné akce provést. Také je možné ověřit správnost návrhu a zjistit, zda nebyla některá funkcionalita vynechána. Na diagramu jsou případné nedostatky jednodušeji vidět. Tvorba *UCD* vychází ze specifikace funkčních požadavků aplikace [22].

Při modelování pomocí *UCD* jsou používány symboly, které znázorňují jednotlivé části diagramu.

Aktér je osoba nebo systém, který používá aplikaci. Aktérem může být např. uživatel, čas nebo jiná aplikace. K označení se používají ikony lidí a textový popisek.

Případ užití je akce, kterou aplikace *aktérovi* nabízí. Případ užití je zobrazen jako elipsa, která je připojena k aktérovi čarou.

Vztahy mezi aktéry a případy užití znázorňují, jestli a jakým způsobem může aktér případ užití používat. Vztahy se značí šipkami, které vycházejí z aktéra a končí u případu užití.

Rozšíření (extend) – je případ užití, který přejímá vlastnosti předchozího případu užití a rozšiřuje je o další možnosti. Znázorňuje se jako šipka, která míří směrem z případu užití a končí u případu užití, který rozšiřuje.

Zahrnutí (include) – zahrnutí je případ užití, který je součástí jiného případu užití. Znázorňuje se jako šipka, která vychází z případu užití, který zahrnuje a končí u případu užití, který je zahrnut.

Zde jsou základní aktéři a některé jejich akce, které v aplikaci modeluji:

Návštěvník je osoba, která se do aplikace nepřihlásí, ale může zobrazit zadání podle odkazu a také procházet veřejnou část aplikace.

Uživatel je přihlášený návštěvník. Má možnost vytvářet, upravovat a mazat příklady a testy, jak pomocí aplikace, tak pomocí *API*. Může také sdílet příklady a vygenerovat na ně veřejný odkaz.

Administrátor může dělat vše, co *uživatel*. Navíc může provádět privilegované zásahy do aplikace a spravovat data jiných uživatelů za účelem údržby aplikace.

Systém je *pomocný aktér*, který provádí pravidelné automatické kontroly a manipulace s aplikací.

3.37.2 Class Diagram

Class diagram (*Diagram tříd*) je také součástí standardu *UML*. Zobrazuje třídy, jejich atributy a metody a vztahy mezi nimi. Souvisí s návrhem struktur uložených v databázi a modelů dat, se kterými pracuji v aplikaci.

Vlastnost (atribut) třídy je proměnná, která je přiřazena k třídě. Nese vlastnost, která dává v kontextu třídy smysl. Např. třída `Auto` může mít vlastnost `barva` s hodnotou `červená` a `maxSpeed` s hodnotou `200 km/h`.

Metoda třídy je funkce, která je přiřazena k třídě. Může mít parametry a výstupní hodnotu. Např. třída `Auto` může mít metodu `drive`, která bude mít parametr `speed` k určení, jakou rychlostí má auto jet.

Třída je v diagramu tříd zobrazena jako obdélník rozdělený na tři části. V první části je název třídy, ve druhé jsou vlastnosti a ve třetí metody. **Veřejné** vlastnosti a metody mají na začátku názvu `+`, **chráněné** `#`, **privátní** `-` a *statické* _.

Veřejné (public) vlastnosti a metody jsou přístupné ostatním třídám a komukoliv, kdo má přístup k objektu a může je zobrazit či změnit. Vlastnosti a metody mají na začátku názvu `+`.

Chráněné (protected) vlastnosti a metody jsou přístupné jen třídám, které dědí vlastnosti z třídy, ze které dědí. Na začátku názvu je `#`.

Privátní (private) vlastnosti a metody jsou přístupné jen třídě, ve které jsou definovány, a nikdo jiný s nimi nemůže pracovat. Na začátku názvu je znak `-`.

Statické (static) jsou vlastnosti a metody, které jsou společné pro všechny instance třídy a tedy nejsou přidružené žádnému objektu a lze je používat i bez inicializace objektu. Je označován pomocí znaku _.

Vztahy *include* a *extend* jsou v diagramu tříd, podobně jako u *use case diagramu*, zobrazeny spojnicemi a šipkami.

Diagram tříd přímo souvisí s návrhem struktur a dat uložených v databázi. Lépe si uvědomíme, jaká data bude nutné uchovávat v databázi a jaká jsou dočasná, nebo tyto závěry je možné z diagramu jednodušeji vyvodit. Tzv. vypočítaná (**závislá**) data se do databáze nezapisují.

Je dobré udržovat *diagram tříd* aktuální, protože se může stát, že se v průběhu vývoje změní návrh a diagram tříd už nebude odpovídat aktuálnímu stavu aplikace. Aktuální diagram může pomoci novému vývojáři k rychlejšímu zorientování v projektu [68].

V této aplikaci modeluji tyto třídy:

Uživatel je fyzická osoba, která používá aplikaci. Slouží k identifikaci uživatele, přidělení a ohlídání oprávnění.

Způsob přihlášení znamená dvojici údaje, který identifikuje daného uživatele, a tajného řetězce, která slouží k přihlášení uživatele.

Záznamy o přihlášení je záznam, který umožní ověření uživatele při každém přístupu k aplikaci. Obsahuje časy přihlášení a odhlášení, *IP adresu*, typ zařízení a další údaje.

Uživatelské akce jsou záznamy o aktivitách uživatele v aplikaci, vázané na konkrétní přihlášení. Např. vytvoření testu, úprava nebo smazání příkladu nebo testu, kategorie, přidělení práv jinému uživateli. Akce je možné vyvolat pomocí jejich hashe, např. ověření e–mailu při registraci.

Kategorie pro příklady nebo testy, které jsou tematicky nebo jinak spojeny. Kategoriemi nejvyšší úrovně je typ školy, potom předmět a konkrétní zařazení látky. Obsahuje název, popis a odkaz na nadřazenou kategorii.

Tagy jsou klíčová slova, která jsou přiřazena k příkladům nebo testům. Označují jejich obsah nebo jakékoli jiné vlastnosti, jako například stupeň obtížnosti. Tagy nemohou mít strukturu (jsou **nestrukturované**).

Příklad značí konkrétní úlohu, vkládanou do testů. Obsahuje název, text, typ (otevřený, uzavřený), odpovědi, proměnné, odkaz na kategorii a pokyny pro generování výsledku.

Proměnná reprezentuje proměnnou, která může být použita v příkladu. Obsahuje název, typ, hodnotu a příznaky sloužící k jejímu zpracování.

Test reprezentuje soubor příkladů, kategorií a pokyny pro generování.

Kapitola 4

Návrh aplikace

Před samotným vývojem aplikace je potřeba se zamyslet, jak bude aplikace fungovat, jaké bude mít funkce a jak bude vypadat. Tvoříme tzv. *funkční požadavky* na výslednou aplikaci. Potom je možné vytvořit *UML modely*, které mohou pomoci blíže si promyslet strukturu a uvědomit si některé další aspekty vývoje, které nás mohou potkat.

Na začátku plánování projektu je dobré se zamyslet i nad tím, jaké technologie budou používány, jak budou jednotlivé součásti komunikovat a jakým způsobem je aplikace vyvíjena. V profesionálním prostředí je potom důležité přemýšlet i nad tím, jestli aplikaci vyvíjí autor sám, nebo v týmu, a zda je na trhu práce dostatek lidí, kteří dané technologie znají ap. [90].

4.1 Návrh fungování aplikace

Aplikace umožní registrovaným uživatelům vytvořit si příklady, umístěné do volitelně strukturovaných kategorií – podle předmětu, ročníku, tématu atp. Tyto příklady bude následně možné vložit do testů při jejich tvorbě, kde bude uživatel mít možnost automatického výběru z kategorií nebo konkrétních příkladů, a ty potom do testu vložit. Uživatelé budou pro přihlášení moci využít kromě hesla také svůj Google účet. Pro požadavky na *API* bude *endpoint* zabezpečen *API* klíčem. Klíč bude závislý na konkrétním účtu, což umožní manipulace a použití příkladů daného účtu.

Řekněme, že uživatel bude chtít pro své žáky vytvořit několik verzí testu s rozdílným zadáním příkladů. Zaregistruje se do aplikace, ověří svoji mailovou adresu kliknutím na odkaz a přihlásí se do svého účtu. Jde o učitele matematiky, takže si vytvoří 5 příkladů z tohoto předmětu a vloží je do kategorie *Matematika* a podkategorie *Lineární rovnice*.

Následně si vytvoří test a může si vybrat, zda chce, aby se v jeho testu objevily všechny příklady z dané kategorie, nebo bude chtít zkombinovat příklady z růzých kategorií. Nakonec otázky v testu bude chtít nechat náhodně seřadit. Příklady jsou navíc zapsány obecně pomocí proměnných, takže v každé verzi budou moci být náhodně dosazené hodnoty. Jelikož některé hodnoty nemohou být zcela náhodné, jinak by to mohlo vést ke složitě řešitelným nebo pro danou úroveň zcela neřešitelným příkladům, bude možné vytvářet mezivýpočty v rámci proměnných. Pomocí proměnných bude možné dosadit náhodnou hodnotu z určeného intervalu nebo vypočtenou z ostatních proměnných. Uživatel si také bude moci vytvořit vlastní proměnné, které bude moci využít v dalších příkladech.

Jako součást příkladu si uživatel vytvoří proměnné A, B, C a D, kde A, B definuje jako náhodná *lichá* čísla od 1 do 10, $C = 4/3 \cdot A$ a $D = C + B$. Řešený příklad se vygeneruje podle předlohy, kterou si uživatel vytvořil. Do proměnných A, B se při konečném generování příkladů uloží náhodné číslo z intervalu, který si uživatel nastavil, a pomocí zápisu proměnné {{`.A`}} se vloží do příkladu či proměnných na zadané místo. Proměnné budou vyčísleny, nebo zachovány jako původní nezpracovaný text. Funkční proměnné budou moci kromě generování čísel obsahovat také vstupní pole, jako např. výběr z možností, nebo textové pole, které bude mít za úkol žák vyplnit. Při generování se proměnné doplní do zadání, případně do možností výběru, pokud je uživatel nastaví. V případě, že zadá vzorec, ze kterého bude možné možnosti vygenerovat, bude možné určit, kolik odpovědí chce generovat. Aplikace pohlídá, že uživatel označí alespoň jedno správné řešení. Příklady bude možné komentovat a hodnotit, aby bylo jednodušší najít chyby v zadání.

Předpokládejme, že budou vygenerována náhodná čísla $A = 3$ a $B = 5$ a hodnoty proměnných C, D jsou zadány jako obecné výpočty takto:

$$
\begin{aligned}
\$A &= 3 \\
\$B &= 5 \\
\$C &= 4/3 \cdot \$A = 4/3 \cdot 3 = 4 \\
\$D &= \$C + \$B = 4 + 5 = 9
\end{aligned}
$$

Proměnné se při generování výsledné podoby příkladu nahradí vygenerovanými hodnotami (čísly, výrazy

nebo funkčními prvky). Po vygenerování příkladu bude potřeba uložit do databáze konkrétní vygenerované hodnoty proměnných a možnosti odpovědí, aby bylo možné identifikovat přesnou verzi příkladu.

Proměnné bude možné shlukovat do variant. To umožní vytvořit více variant příkladů. Např. varianta, kde se budou generovat čísla celá, a varianta, kde se budou vyskytovat i zlomky. Uživatel si pak při tvorbě testu jen vybere, kterou variantu chce použít.

Uživatel bude mít možnost zvolit si, zda chce vytvořit otevřenou nebo uzavřenou formu příkladu s možnostmi výběru. V tomto případě může také využít proměnné, které si vytvořil dříve. V obou případech bude moci zapsat vzorec, ze kterého může být vygenerováno několik možností výběru, vč. správné odpovědi. Pokud uživatel pro tvorbu možnosti využije vzorec, bude možné jich z jedné možnosti vytvořit několik. Ve výsledném příkladu budou v možnostech výběru proměnné nahrazeny a vyčísleny, pokud nebude přepínači určeno jinak.

Vzorový příklad na výpočet kořenů jednoduché rovnice může vypadat např. takto:

$$\text{Řeš v oboru } \mathbb{N} \text{ kořeny rovnice } \{\{.A\}\}x + \{\{.B\}\} = \{\{.C\}\} + \{\{.D\}\}$$

$$\begin{aligned} a) &\quad \text{Nemá řešení v oboru } \mathbb{N} \\ b) &\quad x = \{\{.A\}\} \\ c) &\quad x = \{\{.B\}\} \\ d) &\quad x = \{\{.A\}\} \cdot \{\{.B\}\} - 2 \end{aligned}$$

Po vygenerování a dosazení hodnot:

$$\text{Řeš v oboru } \mathbb{N} \text{ kořeny rovnice} \quad 3x + 5 = 4 + 9$$

$$\begin{aligned} a) &\quad \text{Nemá řešení v oboru } \mathbb{N} \\ b) &\quad x = 3 \\ c) &\quad x = 5 \\ d) &\quad x = 13 \end{aligned}$$

Vstupní políčka se ve výsledné práci vykreslí jako prázdné nebo podtržené místo o velikosti určené při zadávání proměnné. V případě otázky s možnostmi výběru se na místě vykreslí všechny možnosti nebo pole pro výběr ve webové aplikaci, aby žák mohl vybrat správnou možnost kroužkováním nebo jinak, jak učitel určí.

To umožní systém používat např. pro jazykové předměty, kde testovaný doplňuje slova do textu nebo vybírá vhodné písmeno. Umožní to také použití jak pro tisk, tak pro webový dotazník.

Některá nastavení bude možné ovlivnit (měnit velikosti, počty možností, vybrat šablonu, zapnout/vypnout náhodné řazení otázek nebo odpovědí) i na úrovni celých testů.

Po nasdílení trvalého odkazu na daný příklad/test bude umožněno zobrazit konkrétní nebo náhodně vygenerovanou variantu příkladu, případně si příklad přidat a upravit ve vlastním účtu.

4.2 Návrh struktury databáze

Pro ukládání dat v aplikaci použiji relační databázový systém *MySQL*. Databáze je soubor dat, která jsou uspořádána tak, aby byla snadno vyhledatelná a měla nějaký význam.

Při plánování databáze využiji *diagram tříd* k odvození konkrétních modelů a vztahů mezi nimi. *Use case diagram* zase pomůže zjistit, jaké **SQL dotazy** (**SQL Query**) budou používány, což umožní lepší promyšlení indexaci jednotlivých sloupců v ***DB***.

O problematiku tvorby databázových tabulek se postará **ORM** (**Object Relational Mapper**), který sám vytvoří tabulky z modelů dat, které bude aplikace používat. Autor aplikace se tak nemusí starat o tvorbu tabulek a jejich vztahů, ale může se soustředit na tvorbu aplikace, nicméně by měl pamatovat na zásady indexování a optimalizace databáze.

Tyto zásady jsou popsány jako 1 – 3. normální formy a jsou součástí zásad nazývaných *ACID* (zkratka pro *Atomicity, Consistency, Isolation, Durability*) [80, 103].

Potom následuje vytvoření tabulek, resp. *modelů dat.* Používá se k tomu jazyk *SQL*, resp. jeho součást **Data Definition Language (DDL)**. Veškerou komunikaci prostřednictvím *SQL* obstará *ORM*, nicméně i při použití *ORM* je dobré jazyk *SQL* alespoň na základní úrovni příkazů `INSERT` a `SELECT` ovládat.

Indexace sloupců znamená optimalizaci dat v daném sloupci pro rychlejší třídění, spojování tabulek a vyhledávání v nich. *Indexy* se vytváří na sloupcích představujících *primární klíč* nebo *unikátní* hodnoty,

které se v tabulce nesmí vyskytnout opakovaně a slouží k jednoznačné identifikaci záznamu, nebo sloupců, které jsou v *SQL* dotazech často využívány k třídění a vyhledávání. *Indexy* je možné připodobnit k lékařské kartotéce, kde jsou karty pacientů seřazeny abecedně, zdravotní sestra pak při vyhledávání pacientů aplikuje metodu **binárního vyhledávání**, čímž najde kartu dříve než kdyby karty nebyly seřazeny [80, 50].

4.3 Výběr technologií

Výběr technologií je důležitý krok. Je nutné vybírat efektivní technologie, které jsou dostatečně výkonné a zároveň přívětivé pro vývojáře, aby vývoj nebyl příliš časově náročný. Myslet je potřeba také na budoucí údržbu a rozšiřitelnost aplikace a zda jsou vybrané nástroje pravidelně opravovány a aktalizovány. Také je potřeba zvážit aktivní komunitu a uživatelskou podporu daných technologií v případě problémů.

Pro tvorbu *FE* použiji knihovnu **React**. Umožní mi efektivní tvorbu *UI*.

Při vývoji aplikace v Reactu použiji i *TS*. *Facebook* s použitím *TS* počítá a vydává mimo *React* také balíček s příslušnými datovými typy. *TS* podporuje i *JSX* pomocí souborů s příponou `*.TSX` [55].

LaTeX je profesionální nástroj pro sazbu textů s možnostmi matematických vzorců, který je používán zejména v akademickém prostředí [77]. V aplikaci využiji formát zápisu matematických výrazů z LaTeXu. LaTeX má mnoho dalších využití a byl použit při tvorbě písemné části této práce.

Mathjax (`https://www.mathjax.org`), je knihovna, která umožňuje zobrazit na webu matematické zápisy mmj. ve formátu LaTeXzápisu. Použiji jej tedy pro zobrazení matematických výrazů v textu příkladů.

Pro formátování textu uživatelům umožním využít formát *MD*. Výsledek generování zobrazím jako webovou stránku a vyvolám **tisk do souboru**, kde je umožněn i tisk do souboru ***PDF***.

Go je *staticky typovaný* kompilovaný jazyk vyvinutý společností Google v roce 2007. *Go* je kompilovaný jazyk. Zároveň je pro vývojáře přívětivý na psaní. Výsledný *program* je rychlý a efektivní. Pro správu paměti využívá tzv. **Garbage Collector (GC)**, který automaticky přiděluje paměť programu a uvolňuje ji pro použití jinými procesy, pokud již není potřeba. To samozřejmě během chodu programu nepatrně zatěžuje zdroje, nicméně díky tomu nemusím správu paměti řešit, což je velká výhoda [92].

Pro **časově kritické aplikace** operující v reálném čase, např. řízení auta, kde by **režijní zátěž** z provozu *GC* mohla způsobit opožděné spuštění jiných důležitých procesů, je vhodné zvolit jazyk s nižší *režijní zátěží*, např. **Rust** nebo **Zig**.

Pro aplikace jako je webový server, je ale *Go* dobrá volba. *Go* navíc automaticky formátuje kód, což zjednodušuje práci s ním a umožňuje lepší spolupráci v týmu, jelikož není nutné hledat jednotný **codestyle**, který by nejlépe vyhovoval potřebám vývoje.

V *Go* jsou také k dispozici tzv. **metadata položek struktur** (**struct tags**), díky kterým je možné ke každému členu (údaji) dané struktury přidat doplňující parametry, sloužící např. k definici přesného datového typu v *SQL* tabulce vytvořené podle *modelu* nebo k automatické kontrole dat při přijetí požadavku ve formátu *JSON*. To vývojáři výrazně zjednoduší práci a zpřehlední kód. Krátká ukázka, viz 4.1.

Během vývoje v *Go* není potřeba celý program kompilovat, stačí jej spustit příkazem `go run`. Při publikaci programu je potřeba program *zkompilovat* pomocí `go build`, čímž *Go* vytvoří optimalizovaný spustitelný binární soubor. Nástroj `cosmtrek/Air` navíc automaticky restartuje server, pokud se změní některý ze zdrojových souborů. To zjednoduší a zrychlí vývoj *BE*.

Gin gonic je webový framework pro jazyk *Go*. Umožňuje vytvářet webové aplikace a *API*. Základní součást webových frameworků je tzv. *Router*, který umožňuje definovat *koncové body* (*endpointy*) a typy požadavků, které je možné využít pro práci s nimi.

Po příchodu požadavku *Router* kontroluje, zda existuje *koncový bod*, který odpovídá požadavku a pokud ano, předá řízení jeho *obslužné rutině* (*handler*u), což je funkce, která zpracuje požadavek a vrátí odpověď. Každý *koncový bod* má svou vlastní *obslužnou rutinu*.

Gin umožňuje také vytvářet skupiny *koncových bodů*, které mají společný základ *URL* adresy a které mohou mít společný *handler*. To umožňuje vytvářet přehlednější kód aplikace.

MySQL je relační **databáze**, která je dnes jednou z nejpoužívanějších databází. Je dostatečně rychlá a spolehlivá. Největší jednotkou je *databáze*, která obsahuje *tabulky*. Jednotlivé sloupce *tabulek* určují *typ dat* a jejich název. Každý *řádek* je jeden záznam v dané tabulce [45].

Jednotlivé *databáze* – někdy používáme název *schéma* – jsou na disku fyzicky reprezentovány adresáři s názvem konkrétní *databáze*. Jednotlivé *tabulky* jsou tvořeny binárními soubory s příponou podle databázového *enginu*, který používáme, umístěné v daném adresáři.

V mém případě jde o *MySQL engine* **InnoDB**, který ukládá strukturu a data v souborech s příponou `*.ibd` [69].

```go
r.POST("/age", func(c *gin.Context) {
    var data struct{
        Age int `json:"age" min:"0" required`
    }

    err := c.BindJSON(&data)
    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "error": err.Error(),
        })
        return
    }

    c.JSON(http.StatusOK, gin.H{
        "age": age.Age,
    })
})
```

(a) Použití struktur v Go k validaci dat

```php
<?php
//...
class dataLoader extends Endpoints{
    //...
    public function actionEndpoint{
        if (!isset($_POST['age'])){
            throw new Exception('Age must be sent');
        }
        if (!is_numeric($_POST['age'])){
            throw new Exception('Age must be a
            ↪  number');
        }

        $age = intval($_POST['age']);

        if ($age < 0){
            throw new Exception('Age must be
            ↪  positive number');
        }

        return JSONResponse({
            "age": $age
        });
    }
}
```

(b) Např. v PHP je potřeba další kontroly

Kód 4.1 Kontrola dat v PHP a použití Bind v Go

GORM je **Object Relational Mapper (ORM)** pro jazyk *Go*. *ORM* systémy umožňují jednoduše vytvářet databázové modely z definovaných struktur. Z nich *ORM* automaticky vytvoří všechny databázové tabulky, včetně těch relačních, a při jakékoliv změně struktur také automaticky změní tabulky v databázi, v rámci *migrace modelů*. Pomocí struktur *GORM* zpřístupní i rozhraní k jednoduššímu vytváření a úpravě záznamů tabulky. Struktury zajišťují jednodušší kontrolu nad daty, která se do databáze ukládají, a zjednodušují načítání konkrétních dat.

Příklad použití modelu tabulky s otázkami a její *SQL* interpretace v *GORM* je vidět v kódu 4.2.

Pokud je potřeba použít složitější operace s *DB*, je samozřejmě možné využít samostatně formulované *SQL dotazy* [43, 1].

Next.js mi jedoduše umožní využít výhody *SSR* aplikace. *Next.js* řeší několik problémů, což usnadňuje vývoj aplikací. Jsou to např.:

Automatický adresářový router *Next.js* automaticky obsluhuje adresy *URI* v aplikaci na základě adresářové struktury.

Server-side rendering *Next.js* umožňuje *renderovat* aplikace na serveru, čímž se zlepšuje jejich výkon a *SEO*.

Statické stránky *Next.js* umožňuje generovat statické stránky, které se rychle načítají a snižují zátěž serveru.

Optimalizace pro SEO Next.js automaticky generuje meta tagy a další informace důležité pro SEO.

Docker, docker-compose jsou nástroje, které umožňují vytvářet **kontejnery**, někdy jim říkáme i **instance**, ve kterých může běžet nějaká aplikace. Použití *kontejnerů* při vývoji je výhodné zejména při spolupráci v týmu, jelikož ihned po stažení a spuštění máme jistotu, že aplikace běží v přizpůsobeném prostředí. To umožňuje jednodušší zapojení dalších lidí do vývoje a nasazení aplikace na různých platformách (např . *Windows*, *Linux* i *MacOS*).

Kontejnery jsou miniaturní verze operačního systému, který je přizpůsobený pro běh jedné aplikace nebo služby. Pomocí souborů `Dockerfile` a `docker-compose.yml` je možné definovat *kontejner* s potřebnými službami, jako jsou *databáze* nebo *emailový server*.

```go
type Question struct {
	gorm.Model
	// User ID who created question
	AuthorID uint
	// User who created question
	Author User
	// Categories
	Cats []Category `gorm:"many2many:cat_question;DELETE"`
	// Question name (for search)
	Title string `gorm:"type:varchar(127);index"`
	// Description of question (for search)
	Description string `gorm:"type:varchar(255)"`
	// The task itself
	Content string `gorm:"type:text"`
	// variables
	Variables map[string]string `gorm:"type:json;serializer:json"`
	// Answers with parameters to generate
	Answers []generator.Answer `gorm:"type:json;serializer:json"`
	// Number of answers to choose from list of answers (both generated and stringified)
	AnswerCount uint8 `gorm:"type:tinyint unsigned"`
	// Number of correct answers
	CorrectAnswerCount uint8 `gorm:"type:tinyint unsigned"`
	// Randomize answer order
	ShuffleAnswer bool
	// If we should parse and generate
	Parse bool
}
```

(a) *GORM* model (Go struktura)

```sql
CREATE TABLE `questions` (
    `id` bigint unsigned NOT NULL AUTO_INCREMENT,
    `created_at` datetime(3) DEFAULT NULL,
    `updated_at` datetime(3) DEFAULT NULL,
    `deleted_at` datetime(3) DEFAULT NULL,
    `author_id` bigint unsigned DEFAULT NULL,
    `title` varchar(127) DEFAULT NULL,
    `description` varchar(255) DEFAULT NULL,
    `content` text,
    `variables` json DEFAULT NULL,
    `answers` json DEFAULT NULL,
    `answer_count` tinyint unsigned DEFAULT NULL,
    `correct_answer_count` tinyint unsigned DEFAULT NULL,
    `shuffle_answer` tinyint(1) DEFAULT NULL,
    `parse` tinyint(1) DEFAULT NULL,
    PRIMARY KEY (`id`),
    KEY `idx_questions_title` (`title`),
    KEY `idx_questions_deleted_at` (`deleted_at`),
    KEY `fk_questions_author` (`author_id`),
    CONSTRAINT `fk_questions_author` FOREIGN KEY (`author_id`) REFERENCES `users` (`user_id`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 COLLATE = utf8mb4_0900_ai_ci
```

(b) Vytvořené tabulky s relacemi, omezeními a indexy

Kód 4.2 Ukázka použití *GORM* modelu

Kontejnery umožní přesměrovat specifický port, čímž je možné se vyhnout kolizi portů a snižuje se riziko útoku na server, na kterém aplikace běží. Na jednom serveru je tím také umožněno provozovat více tzv. **kontejnerizovaných** aplikací [87, 48].

Nginx sám sebe definuje jako *webový server*, *reverzní proxy server* a *load balancer*. **Reverzní server** umožňuje přesměrovat požadavky na různé servery, resp. aplikace (*porty*), podle zadaných pravidel. Např. podle *URL adresy*, typu požadovaného souboru nebo typu požadavku může vyvolat akci aplikace, běžící na určitém portu. **Load balancer** zajišťuje rozložení zátěže mezi servery (resp. více instancemi aplikace), které má k dispozici. To se využívá např. u větších aplikací, které mají vysokou návštěvnost, nebo v případě, že jednotlivé součásti aplikace běží na více serverech. Také je možné nastavení záložních serverů pro případ, že nastane výpadek nebo přetížení hlavního serveru.

Nginx je ve srovnání s jinými servery, jako např. Apache, považován za velmi rychlý. Zejména pak při obsluze požadavků na statické soubory. Je napsán v jazyce C a je dostupný pro většinu operačních systémů [21].

Alpine Linux je odlehčená distribuce *Linuxu* zaměřená na jednoduchost, zabezpečení a efektivní využití prostoru na disku, určená zejména pro nasazení v kontejnerech, kde očekáváme malou zátěž na úložiště a zdroje. Jde nicméně o plnohodnotný systém, který je možné použít i na serveru nebo jiném stroji. Celková velikost distribuce v kontejneru je jen kolem 8 MB (cca. 130 MB v plné verzi pro počítač s grafickým rozhraním), což je velice málo [5].

Na **Virtual Private Server (VPS)** se systémem **Debian** poběží *Nginx* a *Firewall*, který nasměruje požadavky na správné porty, a *Firewall* (`UFW`), který zablokuje přístup k portům, které je potřeba skrýt před veřejností.

Kombinací těchto technologií bych rád dosáhnul co možná nejlepšího poměru výkonu a bezpečnosti.

Kapitola 5

Postup implementace

Jako první jsem vytvořil základní adresářovou strukturu aplikace. Pro lepší automatizaci a konzistenci prostředí, kdekoliv aplikaci spustím, budou *vývojové* i *produkční* prostředí v *Docker kontejnerech.*

V produkčním prostředí dojde ke změně z vývojového na produkční prostředí, čímž dojde k překladu a optimalizaci kódu. Před tím dojde také ke spuštění testů. Spuštěná výsledná služba tak bude lépe optimalizována a automatickými testy bude zaručeno, že je plně funkční.

Při vývojí *FE* využiji nastavení *proxy*, které *NextJS* nabízí. Požadavky, jejichž *URI query* začíná `/api/...` nasměruji na kontejner, obsahující *BE* aplikace.

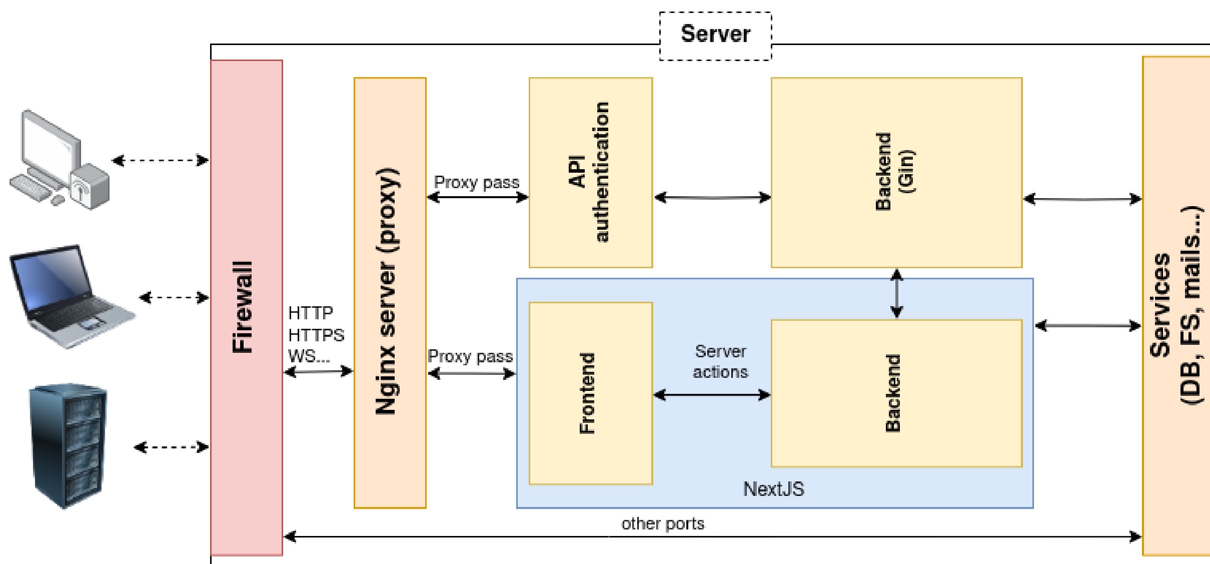
Vývojová verze budou kontejnery pro *React*, *Go* a *databázi*. *Go* kontejner obsahuje nástroj *Air*. *FE* bude spuštěný ve vývojové verzi *NextJS*, která již má funkci (**Hot reload**), která při změně automaticky přenačte stránku v prohlížeči.

Produkční verze, tedy ta výsledná, se kterou bude pracovat *koncový uživatel*, bude obsahovat stejné kontejnery bez nástrojů pro vývoj. Při tvorbě kontejnerů se *BE* zkompiluje a spustí se pouze optimalizovaný binární soubor. *FE* (*React*) bude optimalizován podobným způsobem a poběží s použitím *NextJS*. *NextJS* bude obsluhovat webové požadavky a požadavky na *API*, přesměruje na kontejner s *BE*. Přesměrování ze serveru na kontejnery zajistí *Nginx*, který běží v prostředí *VPS* serveru a obsluhuje i další aplikace.

Kód pro *BE* bude v adresáři `goapp` a kód pro *FE* v adresáři `frontend-next`. Všechny kontejnery budou mít své úložiště v adresáři `data`, kam budou ukládat trvalá data, která se zachovají i po restartu kontejneru. To zjednoduší zálohování dat a případný přesun na jiný server.

Konfigurace pro vytvoření kontejnerů budou v adresáři `docker` a v souborech `docker-compose.yml` v kořenovém adresáři. K dispozici budou dvě verze, *vývojová* bude připravena pro vývoj na lokálním počítači a optimalizovaná, produkční verze pro nasazení aplikace na server.

Přibližné schéma je možné vidět na diagramu 5.1



Obrázek 5.1 Schéma služeb aplikace (AJ)

5.1 Backend

Na *BE* bude program v jazyce *Go*, který zpracovává požadavky od klienta. Je dobrou praxí rozdělit kód na více funkčních celků, tj. *moduly*. *Moduly* umožňují mmj. využít veřejně dostupný kód jiných projektů, pokud to dovoluje zadání od zákazníka i licence daného *modulu*. Tím je možné ušetřit čas, který správa takového modulu zahrnuje [108]. *Moduly* v *Go* je možné používat přímo použitím veřejného odkazu na zdrojový kód (např. kód z *Gitlabu*), čímž výrazně zjednodušuje práci s nimi.

Pro obsluhu požadavků využiji *gin-gonic/gin* s vlastním dříve vytvořeným rozšířením o autentizaci požadavku a kontrolu a správu *JWT sjiamnocna/goethe*.

Při vývoji využiji nástroj `cosmtrek/Air`, který automaticky restartuje aplikaci při každé změně kódu, čímž vývoj aplikace zrychlí, jelikož nebude nutné manuální spouštění aplikace.

BE tvoří dvě části, aby byl lépe testovatelný a rozšiřitelný. Jsou to *webový server*, který bude přijímat požadavky a zpracovávat uživatelská data, a *generátor*, který přijme požadavky a vygeneruje části příkladu podle zadání. Ty předá ve strukturované podobě výslednému zobrazení (ve smyslu *MVC*). V případě nalezených chyb (např. nekompatibilní nastavení, závislost na neexistující proměnné...), vedoucí k nevytvoření příkladu, sdělí chyby uživateli, případně se pokusí, pokud je to možné, samostatně chyby odstranit.

Základním prvkem aplikací je registrace a přihlašování uživatelů. Při registraci je potřeba ověřit uživatele pomocí e-mailové adresy. Také je potřeba zamezit duplicitním účtům kontrolou existence uživatele před registrací a uživateli sdělit, že se pokouší registrovat podruhé, ještě před odesláním formuláře. Uživatel se přihlašuje pomocí jména a hesla nebo některého *externího poskytovatele ověření identity*.

V případě, že uživatel zapomene heslo, je potřeba mu poslat e-mail s odkazem na stránku, kde si může heslo změnit. Tento odkaz musí být unikátní a platný pouze po omezenou dobu, aby nebylo možné jej zneužít. K tomu využije jednoznačný *hash*, platný po omezenou dobu a obsahující informace o daném požadavku. Zároveň informaci o požadavku uložím do *databáze*, abych mohl po přístupu na stránku pomocí odkazu zkontrolovat platnost požadavku na změnu hesla.

Šablony s různými překlady zpráv budou v adresáři `/goapp/templates`. Podle jazykových preferencí uživatele se vybere příslušná šablona. V případě, že uživatel nemá nastavený jazyk, použije se výchozí jazyk, tedy angličtina.

5.1.1 Funkce Init

Go má dvě vstupní funkce *main* a *init*. Funkce *main* je hlavní vstupní bod aplikace, který se spustí jako první. Funkce *init* se spustí jako první, ještě před funkcí *main* a slouží ke konfiguraci programu nebo připojení do databáze. Funkce *main* je hlavní funkcí aplikace, kde se nachází logika aplikace volající další akce.

Ve chvíli, kdy spustíme aplikaci, se nejdříve spustí funkce *init* a poté hlavní část aplikace, funkce *main*.

Funkce *init* použiji k nastavení připojení k databázi prostřednictvím *ORM*(`go-gorm/gorm`). V rámci *initu* spouštím migraci databáze, aby *GORM* vytvořil databázové tabulky nebo jejich podobu aktualizoval. Jednotlivé modely dat jsou vidět ve zdrojovém kódu aplikace.

K připojení do databáze je používán tzv. **Data Source Name (DSN)**. Je to řetezec znaků, který obsahuje údaje k připojení do databáze. Obsahuje *adresu* databáze, *port*, na kterém běží, *název databáze*, *jméno uživatele* a *heslo*. Ty si načítám z prostředí systému, tedy *Dockeru*, který si je načte ze souboru `.env`.

Ve funkci *init* také připravuji šablony různých zpráv. Při použití modulu `html/template` pro zpracování šablon s proměnnými, je lepší si šablony připravit do paměti a v místě použití jen dodat příslušná data. To zajistí rychlejší odezvu aplikace. Šablony slouží zejména pro vytváření e-mailů, které budou odesílány uživatelům a k tvorbě případných chybových hlášek.

5.1.2 Router

Ve funkci *main* je hlavní logika aplikace. Spouštím zde *HTTP server*, který čeká na požadavky od *klienta*. K tomu slouží funkce `Run` z modulu `gin-gonic/gin`, prostřednictvím mého dříve vytvořeného modulu `sjiamnocna/goethe`, který zjednodušuje udržování *JWT tokenu* – ověří, načte a aplikaci zpřístupní informace, které jsou v tokenu uloženy. Při každé změně dat změní a v odpovědi zašle nový *token*.

Při každém volání zkontroluji a do požadavku načtu informace o aktuálním uživateli. S těmito daty pak mohou pracovat další *handlery*. Spuštění *Gin* serveru (routeru), nastavení *callbacků* a vytvoření skupin *endpointů* je vidět v kódu 5.1.

Důležitou součástí aplikace *GINu* je tzv. **Router**, který z příchozího požadavku zjistí, který *koncový bod* je volán a předá volání příslušnému *handleru*. *Router* také umožňuje vytvářet skupiny *koncových bodů*, které mají společný základ *URL* adresy a které mohou sdílet část logiky. To umožňuje vytvářet aplikace, které mají jednotnou strukturu a jsou přehledné. *Router* funguje podobně i pro obsluhování požadavků na *FE*.

Obsluha jednotlivých požadavků díky *GINu* probíhá paralelně (souběžně) v **Gorutinách** (**Goroutines**), což znamená, že může být obslouženo více požadavků najednou. To umožňuje obsloužit více požadavků najednou [31].

```go
func main() {
	// create gin engine
	goethe.Callbacks = &goethe.GCallbacks{
		// issueJWT with custom claims
		BeforeIssueJWT: callbacks.BeforeIssueJWT,
		// verify Service Name and Key
		OnVerifyCredentials: callbacks.OnVerifyAPICredentials,
		// verify custom JWT claims (e.g. API session)
		OnVerifyClaims: callbacks.OnVerifyCustomClaims,
	}

	engine := goethe.GoetheEntry()

	// attach gin-sentry middleware
	engine.Use(sentrygin.New(sentrygin.Options{}))

	{
		// define endpoint groups
		categoriesGroup := engine.Group("/Cat",
↪ goethe.RequireAuthLevel(goethe.AUTH_USER_LOGGED_IN))
		eapp.RegisterCatsGroup(categoriesGroup)

	//...
	}

	engine.Run() // listen and serve on 8080
}
```

Kód 5.1 Spuštění serveru a definice skupin endpointů

Endpointy jsou umístěny v adresáři `Endpoints`, kde má každá skupina *Endpointů* svůj vlastní adresář. V těchto souborech jsou jejich *handlery*. *Handler* dostává při obsluze *požadavku* jako parametr *gin.Context*, který obsahuje kontext aktuálního požadavku, a umožňuje s ním *handleru* pracovat. Komunikace *FE* a *BE* bude probíhat pomocí *JSON*u, případně *HTTP stavovými kódy*. *Endpointy* budou sdružovány podle zdroje se kterým manipulují.

/api/user – *CRUD* akce přihlášení, registrace a správy uživatelských účtů, bez kterých se neobejde většina moderních aplikací.

/api/cat – *CRUD* akce spojené s kategoriemi. Kategorie jsou určeny pro zařazení úkolů a testů nebo specifikaci konkrétního oboru uživatelem.

Při implementaci načtení dat z požadavků bylo nutné použít jiný způsob zpracování některých vlastních struktur, jelikož pracuji se *staticky typovaným* jazykem a *Gin* nedokázal automaticky zpracovat všechny struktury. Proto načítám danou vlastnost jako textový řetězec a dále zpracovávám.

Ukázka funkce pro zpracování struktury dat odpovědí z *JSON* řetězce na otázku je vidět v kódu 5.2.

5.1.3 Generátor

Zásadní funkci při tvorbě výsledné podoby zadání bude mít *generátor*.

Generátor bude postupně nahrazovat hodnoty použitých proměnných a funkcí ke generování čísel a dalších prvků. K tomuto účelu slouží modul správce proměnných, který nahrazuje proměnné a funkce jejich hodnotami. Mimo to zajistí náhodné seřazení a splnění dalších bodů, které si uživatel vyžádá. Konečné nahrazení hodnotami proměnných v zadání proběhne pomocí modulu `text/template`, který umožňuje jednoduché využití šablon s proměnnými.

Proměnné budou při definici i v zadání používat stejný zápis, jak definuje balík `template/html`, tedy {{`.jméno`}}.

K identifikaci proměnných budou sloužit jejich jména. Jména pak poslouží jako klíče, které na sebe **namapují** (přiřadí) vlastnosti a obsah proměnných. K určení obsahu, spuštění funkcí, které vygenerují jejich hodnoty, a případnému vyčíslení (redukci) *mnohočlenů* bude nutné vytvořit **parser**, což je program, který bude schopný rozdělit řetězec a načíst jednotlivé části, kterým přiřadí konkrétní význam. Podle celkové podoby pak určí, zda jde o *proměnnou*, *funkci*, *polynom* nebo *operátor/závorku*. U každé části pak určí, zda je platná, a případně ji nahradí její nalezenou/vypočítanou hodnotou.

```go
// unmarshall data from JSON request byte string
func UnmarshallAnswerSlice(raw []byte) ([]generator.Answer, error) {
	// single string representation of answer
	var answers []struct {
		Content  string `json:"Content" binding:"required"`
		Correct  bool   `json:"Correct,omitempty"`
		MathMode bool   `json:"MathMode,omitempty"`
		Generate uint8  `json:"Generate,omitempty"`
		Final    bool   `json:"Final,omitempty"`
	}

	err := json.Unmarshal(raw, &answers)

	if err != nil {
		return nil, errors.New("wrong format of answers")
	}

	var res []generator.Answer

	// map answers from JSON
	for _, ans := range answers {
		res = append(res, generator.Answer{
			ResultAnswer: generator.ResultAnswer{
				Content: ans.Content,
				Correct: ans.Correct,
			},
			Final:    ans.Final,
			Generate: ans.Generate,
		})
	}

	return res, nil
}
```

Kód 5.2 Načtení dat odpovědí a jejich převod před uložením do databáze

Parser musí rozložit zápisy hodnot proměnných na jednotlivé **dílčí řetězce** (**tokeny**) a identifikovat jejich typ. K tomu použiji modul `bzick/tokenizer`. Ten ve vstupním textu identifikuje základní typy jako – čísla, identifikátory a vyfiltruje **bílé znaky** (mezery, ukončení řádku ap.). Výsledkem je seznam *tokenů*, které byly v textu rozpoznány. Podle kontextu (typů okolních *tokenů*) potom určím jejich význam v aplikaci a jak je s nimi potřeba pracovat.

Např. pokud za znakem { bude tečka, vím, že jde o proměnnou. Pokud je tam identifikátor (řetězec znaků) a závorky s argumenty pro funkci, jde o funkci. Po zpracování proměnných a funkcí bude možné určit zbytek jako mnohočleny, zlomky apod.

Při řešení proměnných musí nejdříve proběhnout spuštění funkcí ve všech proměnných, na kterých je právě řešená proměnná závislá (odkazuje se na ně). Tím jsou vygenerovány náhodné nebo vypočítané hodnoty a pokud je proměnná použita na více místech, bude mít všude stejnou hodnotu. Pro snížení počtu operací toto provedu při prvním použití proměnné v jiné proměnné nebo před vytvořením konečného zadání. Použití proměnných budu zaznamenávat v rámci *FE* a upozorňovat na nepoužité proměnné.

Potom je možné provést nahrazení proměnných, vložením *tokenů* z proměnné na místo určení. Pokud je závislá proměnná definována dříve než ta, kterou je potřeba nahradit, jde o chybu. Pokud závislá proměnná není zpracovaná, zpracuji ji voláním téže funkce nad danou funkcí (tzv. **rekurzivní volání**). Do seznamu vložím název proměnné a pokud zpracovávaná proměnná již v seznamu je, jde o chybu, která by způsobila nekonečné cyklické zpracování týchž proměnných (**deadlock**).

Proměnné tak budou nahrazeny ve správném pořadí a pokud k nahrazení nedode, znamená to, že neexistuje, nebo má *cyklickou závislost*.

V matematickém režimu je potřeba vyčíslit hodnotu výrazu zapsanou v *tokenech*, např. `randInt(0,42) + 1` pro náhodné liché číslo. To realizuji pomocí algoritmu podobnému tzv. **Postfixové notaci** (**Reverzní polská notace**), která nevyužívá závorky ani prioritu operátorů, a operátory jsou zapsány za operandy. Výraz je zapsán jako posloupnost **operandů** (**hodnot**) a *operátorů*, např. klasický infixový zápis $19 + (26 + 1)$ je převeden do *postfixového* jako $19\ 26\ 1\ +\ +$. Dvojice *operandů* jsou zpracovávány operátorem a tedy je jednodušší zpracování pomocí funkcí (např. `sum(a, b)`) v programovacích jazycích. Pro názornost algoritmu převodu a vyčíslení pomocí struktury *zásobníku* (struktura typu **Last In First Out (LIFO)** [78]) je možné využít vizualizaci www.free-online-calculator-use.com/postfix-evaluator.html [74].

Protože pracuji s jazykem *Go*, obdrží *generátor* data ve formě *struktury* definující příklad a výsledkem bude struktura obsahující zpracované zadání a možnosti s nahrazenými a vyčíslenými hodnotami, aby bylo možné hodnoty předat *zobrazení* (ve smyslu *MVC*). Výsledná data budou obsahovat zadání otázky, možnosti odpovědí a informaci o tom, která možnost je správné řešení. Na místě zobrazení uživatelského vstupu bude vložen *placeholder*, který bude zpracován až podle volby zobrazení jako vstupní pole nebo prostor pro odpověď s případným výčtem možností.

Generátor samotný nemá přístup k databázi, proto manipulaci se zadáním a uložení vygenerovaných hodnot zajistí aplikace v rámci konkrétního *endpointu*, hodnoty si načte ze struktury, kterou obdrží z generátoru. Konkrétní hodnoty uloží do databáze vygenerovaných příkladů s jednoznačným identifikátorem, který předá uživateli, který zároveň bude mít k dispozici seznam dříve vygenerovaných verzí příkladu.

5.1.4 Autentizace uživatelů

Registrace a přihlášení do aplikace budou možné více způsoby. Pro všechny způsoby přihlášení bude v databázi existovat jedna tabulka s přihlašovacími údaji. Uživatelé budou moci k používání aplikace použít kombinaci *e-mailové adresy a hesla*, ale také *Google* účet, díky použití *Firebase*. V budoucnu tak bude jednodušší přidat i další služby k ověření identity uživatele.

Na *FE* použiji nástroj *Firebase* od společnosti *Google*, který umožňuje mmj. jednoduchou *autentizaci* uživatelů pomocí účtu u společnosti Google. Přihlášení pomocí poskytovatele identity proběhne takto:

1. Uživatel se přihlásí k *FE* pomocí vybraného poskytovatele.
2. *FE* vygeneruje *JWT* pomocí uživatelských údajů získaných z *Firebase* a odešle jej na *BE*.
3. *BE* (*Go*) ověří *JWT* pomocí údaju poskytovatele.
4. *BE* provede ověření přihlášení požadavkem u poskytovatele
5. *BE* najde uživatele v databázi podle *e-mailové adresy* a pokud neexistuje, zahájí proces registrace nového uživatele.

6. *BE* vytvoří nový *JWT* token a zašle jej na *FE*.
7. *FE* si uloží nový *JWT* a bude jej nadále používat pro autorizaci požadavků na *BE*.

Pokud bude ověření úspěšné, uživatel je přihlášen a unikátní identifikátor aktuální relace vložím do *JWT* a databáze. Při každém přístupu bude zkontrolována platnost relace. To umožní mít přehled o přihlášených uživatelích a zabezpečí přístup k datům. Umožní to načíst data uživatele do paměti programu a vzdálené odhlášení v případě podezřelého přístupu k účtu.

5.2 Frontend

FE bude vytvořen pomocí nástroje *NextJS* v jazyce *TS*. *NextJS* je založen na knihovně *React* a umožňuje využívat většinu jeho výhod. Pro lepší přístupnost aplikace a *SEO*, využiji k tvorbě komponent některý z *CSS* nástrojů nebo hotových šablon, které jsou již většinou připraveny s myšlenkou přístupnosti pro slabozraké či využívající čtecí zařízení, což zjednoduší práci při tvorbě *UI*.

Ke kontrole přístupnosti využiji rozšíření **Wave** (`https://wave.webaim.org/extension/`).

Pro komunikaci s *BE* budu používat dříve napsaný *Typescript modul sjiamnocna/renette-api*, který je napsaný k udržování *JWT* a tedy zjednoduší komunikaci *FE* s *BE*. Při každé důležité změně dat o uživateli, jako je přihlášení, odhlášení a změna hesla, se změní data a s nimi i *JWT*, čímž se redukuje riziko jeho zneužití.

Důležitým bodem při vývoji je neukládat citlivé údaje, jako např. klíče nebo soukromé údaje v kódu (tzv. **hardcoded**), kde k nim bude mít přístup téměř kdokoliv, kdo má přístup k používanému verzovacímu systému, a vzniká tak riziko zneužití údajů. Proto jsou takové údaje načítány do systému jako **proměnné prostředí** (**environment variable**). *Docker* je schopný načíst a zpřístupnit kontejneru hodnoty, které jsou uloženy v souboru `.env`. *BE* napsaný v *Go* umí konstanty z prostředí číst pomocí `os.Getenv("nazevpromenne")`. Nicméně *NextJS* díky svému návrhu neumožňuje načtení *env proměnných* do veřejné části aplikace, budu je při spuštění *NextJS* aplikace pomocí skriptu z prostředí přepisovat do *JS* souboru jako konstanty a dále s nimi jako s běžnými z *JS* exportovanými *konstantami* budu pracovat. Přepis proměnných do *JS* souboru bude prováděn pomocí *Node.js* skriptu, viz 5.3.

```
// node.js environment
const fs = require('fs');
const os = require('os');

// truncate the file and write new env vars into it
const f = fs.open('env_vars.js', 'w', function(err, fd) {
    if (err) {
        throw 'could not open file: ' + err;
    }

    // now one by one write the env vars into the file
    for (const i in process.env){
        if (!i.startsWith("NEXT_PUBLIC")){
            continue;
        }

        const content = `export const ${i} = '${process.env[i]}'${os.EOL}`
        fs.appendFile(fd, content, function(err) {
            if (err) throw 'error appending file: ' + err;
        });
    }

    // close the file
    fs.close(fd, function() {
        console.log('wrote the file successfully');
    });
});
```

Kód 5.3 Přepis **proměnných z prostředí** na *JS* konstanty pomocí *Node.js* kódu

Aplikace bude napsaná v anglickém jazyce, do ostatních jazyků (vč. Češtiny) budu aplikaci překládat.

K tomu použiji modul `next-intl`, na jehož webu je uvedeno, že má nativní podporu přímo od **Vercelu**. Překlady budou uloženy v adresáři `locales` a budou se načítat dynamicky podle aktuálního jazyka. V případě, že překlad chybí, zobrazí se anglický text. Překlady budou vytvořeny pomocí *Google Translate* a budou přeloženy ručně.

Router v případě *FE* určuje, která stránka (resp. *komponenta*) bude na základě *URI* (adresy), *query* parametrů a stavu okna prohlížeče zobrazena. Veřejná prezentace aplikace bude definována v hlavní části *routeru*, stránky pro přihlášené uživatele použijí tzv. *lazy loading*.

React po spuštění aplikace za normálních okolností načítá celou aplikaci najednou, a to včetně obrázků, obsahu ap. To může vyústit ve zbytečné načítání velkého objemu dat a ke snížení celkové rychlosti aplikace, nehledě na zátěž na připojení a výkon na straně *klienta*.

Lazy loading je technika, umožňující načíst pouze *komponenty*, které jsou potřeba, a zbytek načítat dodatečně pouze v případě, že jsou zapotřebí. Typicky rozdělujeme funkční komponenty tak, jak jsou v aplikaci použity, a podle toho jsou postupně načítány. V této aplikaci budu používat *lazy loading* pro načítání stránek, které jsou dostupné pouze přihlášeným uživatelům. V případě, že uživatel není přihlášen, bude přesměrován na stránku s přihlašovacím formulářem. V případě úspěšného přihlášení uživatele pak zobrazím příslušnou část aplikace s kontrolními prvky [89].

Router je zodpovědný také za zobrazení chybové hlášky v případě, že uživatel zadá neexistující adresu (*HTTP chyba 404*) nebo se snaží k přístupu ke stránce, ke které nemá oprávnění (*HTTP chyba 403*). V takovém případě je nutné uživatele informovat, že došlo k chybě, a případně mu doporučit, co může dělat dál, a zaznamenat chybu do logů (tzv. *zalogovat*).

User Interface (UI) (**uživatelské rozhraní**) bude vytvořeno s použitím knihovny **Tailwind CSS (TW)**. *TW* je **utility-first** knihovna. Soustředí se na poskytnutí jednoduchých prvků prostřednictvím *CSS* tříd, které je možné ke komponentám přidávat. Skládáním těchto tříd je možné vytvořit potřebný vzhled komponent. *TW* je také možné rozšířit o vlastní třídy nebo další knihovny, které rozšiřují jeho funkčnost a mohou být použity v rámci celé aplikace. Pro výslednou podobu aplikace potom *TW* vytvoří *CSS* kód pouze z použitých tříd, čímž ještě omezí množství dat, které jsou přenášeny v podobě stylů [73].

Konkrétně využiji knihovnu komponent **Shadcn**, která poskytuje jednoduše použitelné a upravitelné komponenty, které jsou optimalizovány pro přístupnost i *SEO*, a využívá *TW* [66]. K tvorbě *UI* je ale samozřejmě možné využít i jiné knihovny stylů (např. **Bootstrap**) nebo další různé přístupy ke stylování.

5.3 Testování

Nemá význam testovat celý *BE*. Balíčky *GIN* i *GORM* jsou aktivně testovány při jejich vývoji. Při chybě, která by nastala při překladu, aplikace vůbec nenastartuje a selže spuštění *Docker* kontejneru. Aplikace v tom případě zůstane v původní spuštěné verzi. Chyby externích služeb (*SMTP*) jsou zachyceny v *Sentry*.

Testovací scénáře budu postupně rozšiřovat, nicméně ze začátku se budu věnovat testování pouze u *generátoru* a *čistých funkcí*.

Testovány budou všechny scénáře, které jsou od *generátoru* požadovány – např. nahrazení proměnných a spuštění funkcí, vyčíslení jednoduchých výrazů. Náhodné řazení nelze testovat, protože výsledek je náhodný a nelze vyloučit, že nebudou vygenerovány několikrát stejné hodnoty. Totéž platí pro náhodné pořadí odpovědí.

Na *FE* budu testovat funkčnost *komponent*. V případě, že se chyba vyskytne za běhu aplikace, zaznamená ji *Sentry* a upozorní na nutnost opravy. Díky *Sentry* budu mít informace o chybě z *FE* i *BE* na jednom místě a budu na ni včas upozorněn.

5.4 Zálohy

Zálohování je jednodušší díky požití kontejnerů s **trvalým úložištěm** (**volume**), kde jsou uložena veškerá data, která se zachovávají i po zastavení či restartu kontejneru.

Zbytek dat tvoří zdrojový kód, který má již zálohu v podobě *GIT repozitáře*.

Proto bude na *serveru* pomocí **CRONu** pravidelně spouštěn skript s příkazem `rsync`, který umožňuje zkopírovat pouze změněné soubory. Při změně souborů se změní také jejich kontrolní součty (*hashe*). V tom případě se soubory zkomprimují a archivují do archivu s názvem tvořeným hashem aktuální podoby dat. Pokud se obsah souborů nezmění, zálohování na externí *server* neproběhne.

Kapitola 6

Závěr

Cíl práce byl dosažen, podařilo se mi v souladu se zadáním vytvořit aplikaci pro generování školních testů. Koncept aplikace jsem diskutoval s vybranými učiteli i vývojáři. Aplikace je v rané fázi vývoje a některé funkcionality tak bude nutné ještě dokončit.

Kód generátoru je přiložen k práci formou `*.ZIP` archivu.

Uživatelé mohou sdílet příklady mezi sebou a přenechat vymýšlení úloh počítači. Může tak vzniknout databáze příkladů, která bude moci být používána k tvorbě testů. Pokud se aplikace dostatečně rozšíří, vytvoří databázi generovaných příkladů, kterou budou moci učitelé používat pro ověření znalostí svých žáků a studentů.

V práci jsem použil aktuální technologie a postupy a snažil se držet základních pouček pro vývoj aplikací a psát čitelný, dokumentovaný a rozšiřitelný kód s důrazem na rychlost a optimalizaci kódu, aby v budoucnu dokázala obsloužit co největší počet požadavků.

Je potřeba vzít v potaz, že popsané postupy v praxi nemusejí být ideálním řešením a může se lišit v závislosti na potřebách konkrétní aplikace. V této implementaci jsem se soustředil na vysoký výkon a ukázku funkce *MVC* s programem v jazyce *Go*, nicméně bylo by možné celou aplikaci napsat jednodušeji např. v *NextJS*.

Informační technologie se také neustále vyvíjí a je nutné sledovat a vyhodnocovat nové trendy a technologie, jelikož v době publikace této práce již mohou být některé postupy překonané. Většina principů ale zůstane platná.

Kód některých částí aplikace je možné najít na Gitlabu (`https://gitlab.com/pisemkomat`).

Přes všechny snahy s pluginy a konfiguracemi pro *I18n* v *NextJS* zjistil, že je v *NextJS* již od r. 2021 nahlášena chyba (viz `https://github.com/vercel/next.js/issues/21565`), která znemožňuje použití *I18n* zároveň s konfigurací *proxy*. Z tohoto důvodu bude řešení problému překladů a spuštění veřejné aplikace realizováno mimo rámec této práce.

Aplikace je dostupná také pomocí *API* a je možné ji integrovat do dalších prostředí formou pluginů. Uživatelé tak budou moci používat generované příklady např. v prostředí **Moodlu** nebo **Content Management System (CMS)** jako např. **Wordpress**. Dokumentace k *API* je k dispozici ve veřejné části webu.

Základ pro tvorbu aplikace pomocí jazyka *Go* a knihovny *NextJS* je k dispozici na *Gitlabu* `gitlab.com/gormrest`. Dokumentace ke kódu je realizována pomocí komentářů v kódu, dokumentačních řetězcích a souborech **README** v popisu repozitářů.

Seznam použitých zkratek

AI Artificial Intelligence. 5

AJAX Asynchronous Javascript Requests. 7, 12, 14, 17

API Application Programming Interface. 1, 5, 7, 10, 14, 17–19, 22, 29, 31, 33, 37, 44

ASCII American Standard Code for Information Interchange. 22

ASM Assembly. 8

AST Abstraktní Syntaktický Strom. 8

BE Backend. 6, 7, 14, 16, 17, 19, 21, 22, 25, 27, 33, 37–39, 41–43

CD Continuous Deployment/Delivery. 22, 23

CI Continuous Integration. 22

CMS Content Management System. 44

CRUD Create Read Update Delete. 18

CSR Client Side Rendering. 14

CSS Cascading Style Sheet. 7, 12, 14, 16, 42, 43

DB Databáze. 32, 34

DDL Data Definition Language. 32

DNS Domain Name Server. 15, 16

DOM Document Object Model. 12

DSN Data Source Name. 38

FE Frontend. 6, 14, 16, 17, 19, 21, 22, 25, 27, 33, 37–39, 41–43

GC Garbage Collector. 33

HTML Hypertext Markup Language. 6, 7, 10, 12, 14, 16, 17, 21, 24

HTTP Hypertext Transfer Protocol. 15, 17, 18, 25, 45

HTTPS *HTTP* Secure. 15

I18n Internationalization. 26, 44

IDE Integrated Development Environment. 12, 26

IIS Internet Information Service. 15

ISP Internet Service Provider. 16

IT Informační Technologie. 1, 21, 26

JIT Just In Time. 8

JS Javascript. 6–10, 12–14, 16, 17, 21, 22, 25, 42

JSI Jazyce Strojových Instrukcí. 8

JSON Javascript Object Notation. 17, 22, 33

JSX Javascript XML. 13, 14, 33

JWT JSON Web Token. 19, 20, 38, 41, 42

LIFO Last In First Out. 41

MD Markdown. 6, 33

MVC Model View Controller. 16, 41

OOP Objektově Orientované Programování. 9

ORM Object Relational Mapper. 32, 34, 38

OS Operační Systém. 25

PDF Portable Document Format. 33

PHP Hypertext Preprocessor. 7, 8, 15

REST Representational State Transfer. 14, 17, 18

SDLC Software Development Lifecycle. 24

SEO optimalizace webu pro vyhledávače. 24, 34, 42, 43

SMTP Simple Mail Transfer Protocol. 15

SPA Single Page Application. 7

SQL Structured Query Language. 20, 21, 32–34

SSE Server Sent Events. 7, 17

SSG Static Site Generation. 13, 14

SSR Server Side Rendering. 14, 34

TLD Top Level Domain. 16

TS Typescript. 8, 10, 12, 33, 42

TW Tailwind CSS. 43

UCD Use Case Diagram. 28

UI User Interface. 27, 33, 42, 43

UML Unified Modeling Language. 27

URI Uniform Resource Identifier. 15, 19, 34, 43

URL Uniform Resource Locator. 15, 18

UTF Unicode. 26

VPS Virtual Private Server. 36

W3C World Wide Web Consortium. 24

WASM Web Assembly. 9

WCAG Web Content Accessibility Guidelines. 24

WS Web Socket. 7, 17

XHR XMLHttpRequest. 17

XML Extended Markup Language. 17

XSS Cross Site Scripting. 21

Použitá literatura

[1] ABBA, I. V.: What is an ORM – The Meaning of Object Relational Mapping Database Tools — freecodecamp.org. 2022, [Zob. 15-Čvc-2023].
URL `https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mappi`

[2] ADAMS, D.: *The Hitchhiker's Guide to the Galaxy*. Pan Books, první vydání, 1979, ISBN 978-0-330-25864-3.

[3] ADOBE: Web Applications. *Adobe Help Center*, 2023.
URL `https://helpx.adobe.com/cz/dreamweaver/using/web-applications.html`

[4] AKANKSHA; Chaturvedi, A.: Comparison of Different Authentication Techniques and Steps to Implement Robust JWT Authentication. In *2022 7th International Conference on Communication and Electronics Systems (ICCES)*, 2022, s. 772–779, doi:10.1109/ICCES54183.2022.9835796.

[5] ALPINE: about — Alpine Linux — alpinelinux.org. 2024, [Zob. 28. 02. 2024].
URL `https://www.alpinelinux.org/about/`

[6] AVA: The Year Python Begins to Fade in the Tech World. 2024, [Accessed 25-05-2024].
URL `https://python.plainenglish.io/2024-the-year-python-begins-to-fade-in-the-tech-world-0`

[7] BEKKHUS, S.: Testing React Apps · Jest — jestjs.io. 2023, [Zob. 21-Čvc-2023].
URL `https://jestjs.io/docs/tutorial-react`

[8] BERJON, R.: *DOM Enlightenment*. O'Reilly Media, Inc., 2014.
URL `https://domenlightenment.com/`

[9] BERNERS-LEE, T.: Information Management: A Proposal. 1989.
URL `http://www.w3.org/History/1989/proposal.html`

[10] CHRÁSKA, M.: *Didaktické testy: příručka pro učitele a studenty učitelství*. Edice pedagogické literatury, Brno: Paido, 1999, ISBN 8085931680.

[11] CONE, M.: Markdown Guide. 2024, [Accessed 28. 02. 2024].
URL `https://www.markdownguide.org/`

[12] COWELL, C.; LOTZ, N.; TIMBERLAKE, C.: *Automating DevOps with GitLab CI/CD Pipelines: Build efficient CI/CD pipelines to verify, secure, and deploy your code using real-life examples*. Packt Publishing, 2023.
URL `https://ieeexplore.ieee.org/document/10162814`

[13] CRUTCHLEY, C.: What is Server Side Rendering (SSR) and Static Site Generation (SSG)? — unicorn-utterances.com. 2020, [Zob. 13. 08. 2023].
URL `https://unicorn-utterances.com/posts/what-is-ssr-and-ssg`

[14] DANG, Q.: Secure Hash Standard. 2015-08-04 2015, doi:https://doi.org/10.6028/NIST.FIPS.180-4.

[15] Debnath, M.: Introduction to Object-oriented Programming in Go — Developer.com — developer.com. 2022, [Zob. 05. 08. 2023].
URL `https://www.developer.com/languages/oop-go/`

[16] DORMAN, M.: *Introduction to Web Mapping*. Boca Raton: CRC Press, Taylor & Francis Group, 2020, ISBN 978-0-367-86118-6, xix, 346 s., obsahuje bibliografii, bibliografické odkazy a rejstřík.

[17] DOVER, D.; DAFFORN, E.: *SEO: optimalizace pro vyhledávače profesionálně*. Encyklopedie webdesignera, Brno: Zoner Press, první vydání, 2012, ISBN 978-80-7413-172-1, 400 s., encyklopedie webdesignera.

[18] DRLÍK, M.: *Moodle*. Brno: Computer Press, a. s., 2013, ISBN 978-80-251-3759-8.

[19] EDWARD, J. K.; GIANNINI, M.: *OOP bez předchozích znalostí : průvodce pro samouky*. Brno: Computer Press, vyd. 1. vydání, 2006, ISBN 80-251-0973-9, 222 s., přeloženo z angličtiny.
URL https://www.knihovny.cz/Record/svkpk.PNA01-000361667

[20] ELROM, E.: React and libraries: your complete guide to the react ecosystem. 2021.
URL https://www.knihovny.cz/Record/svkhk.HKA01-001027121

[21] F5: What Is NGINX? 2004, [Zob. 15-Čvc-2023].
URL https://www.nginx.com/resources/glossary/nginx/

[22] FAKHROUTDINOV, K.: Use Case Diagrams. 2023, [Zob. 18-Čvc-2023].
URL https://www.uml-diagrams.org/use-case-diagrams.html

[23] FAPI: Jak na prompty v chatGPT? — fapi.cz. https://fapi.cz/blog/jak-na-prompty-v-chatgpt/, 2023, [Accessed 11-03-2024].

[24] FIELDING, R. T.; NOTTINGHAM, M.; RESCHKE, J.: HTTP/1.1 Messaging. Internet-Draft draft-ietf-httpbis-messaging-03, Internet Engineering Task Force, Øíjen 2018, work in Progress.
URL https://datatracker.ietf.org/doc/draft-ietf-httpbis-messaging/03/

[25] FIRDAUS, B.: Introduction to JWT (Also JWS, JWE, JWA, JWK) — codecurated.com. 2022, [Zob. 25. 07. 2023].
URL https://codecurated.com/blog/introduction-to-jwt-jws-jwe-jwa-jwk/

[26] FOGEL, K.: *Tvorba open source softwaru*. Edice CZ.NIC, 2012, ISBN 978-80-88168-03-4.

[27] FOWLER, M.: *Destilované UML*. Myslíme v–; Knihovna programátora, Praha: Grada, první vydání, 2009, 173 s., iSBN 978-80-247-2062-3 (brož.) : 299.00 Kč.

[28] FRIEDL, J. E.: *Mastering Regular Expressions*. O'Reilly Media, tøetí vydání, 2015, ISBN 978-1491948769.

[29] FUERTES, A. B.; MARIA, P.; HORMAZA, M.: Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry. *Applied Sciences*, roèník 13, è. 6, 2023: str. 3667, doi: 10.3390/app13063667.
URL https://www.mdpi.com/2076-3417/13/6/3667

[30] GitLab: GitLab Actions.
URL https://docs.gitlab.com/ee/ci/pipelines/actions/

[31] Google: Effective Go. 2009, zob. 2024-02-17.
URL https://tip.golang.org/doc/effective_go

[32] GRAHAM, D.: Ethical hacking: a hands-on introduction to breaking in. 2021.
URL https://aleph.nkp.cz/F/?func=direct&doc_number=001875895&local_base=STK

[33] GRINBERG, M.: JSON Web Tokens with Public Key Signatures — blog.miguelgrinberg.com. [Zob. 21-Čvc-2023].
URL https://blog.miguelgrinberg.com/post/json-web-tokens-with-public-key-signatures

[34] HANÁK, D.: Lekce 11 - Best practices pro vývoj softwaru - Vývoj webových aplikací — itnetwork.cz. [Zob. 17-Čvc-2023].
URL https://www.itnetwork.cz/navrh/best-practices-pro-vyvoj-softwaru/jak-rychle-a-kvalitne-vytvorit-webovou-aplikaci

[35] HERNANDEZ, R. D.: The Model View Controller Pattern – MVC Architecture and Frameworks Explained — freecodecamp.org. 2021, [Zob. 18-Čvc-2023].
URL https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-ar

[36] HERRERA, E.: GraphQL vs. REST APIs: Why you shouldn't use GraphQL. *LogRocket Blog*, 2023, zob. 26. 7. 2023.
URL https://blog.logrocket.com/graphql-vs-rest-api-why-you-shouldnt-use-graphql/

[37] HLADIŠ, K.: Aktualizace algoritmu Google Penguin 4: Google událost roku 2016? *IT Systems*, roèník 18, è. 11, 2016: s. 24–25, ISSN 1802-002X, roč. 18, č. 11 (2016), s. 24-25.
URL https://www.systemonline.cz/sprava-it/aktualizace-algoritmu-google-penguin-4.htm

[38] HRANICKÝ, J.: Lekce 3 - Jak se bránit proti SQL injection — itnetwork.cz. [Zob. 04. 08. 2023].
URL https://www.itnetwork.cz/php/bezpecnost/jak-se-branit-proti-sql-injection

[39] HÉDLOVÁ, L.: *Chyba v řešení matematické úlohy a její reflexe učitelem primární školy*. Diplomová práce, UNIVERZITA PALACKÉHO V OLOMOUCI, Pedagogická fakulta, 2015, přístupné online z https://theses.cz/id/o8gusp/.

[40] INDRASIRI, K.: Build Real-World Microservices with gRPC — thenewstack.io. 2018, [Zob. 18-Čvc-2023].
URL https://thenewstack.io/build-real-world-microservices-with-grpc/

[41] JAGELKA, M.: Co znamenají pojmy uzavřená a otevřená otázka? 2022, [Zob. 24-Čvc-2023].
URL https://dvojka.rozhlas.cz/co-znamenaji-pojmy-uzavrena-a-otevrena-otazka-8750731

[42] JANOVSKÝ, D.: CSS třídy (class). [Zob. 15. července 2023].
URL https://www.jakpsatweb.cz/css/css-tridy-class.html

[43] JINZHU: GORM — gorm.io. 2013, [Zob. 15-Čvc-2023].
URL https://gorm.io/

[44] KADLEC, V.: *Agilní programování: metodiky efektivního vývoje softwaru*. –, Brno: Computer Press, první vydání, 2004, ISBN 8025103420, 278 s., obsahuje bibliografii, bibliografické odkazy a rejstřík.

[45] KAMARUZZAMAN, M.: Top 10 Databases to Use in 2021 — towardsdatascience.com. [Zob. 15-Čvc-2023].
URL https://towardsdatascience.com/top-10-databases-to-use-in-2021-d7e6a85402ba

[46] KANTOR, I.: JavaScript Tutorial. Zob. 14. července 2023.
URL https://github.com/javascript-tutorial/en.javascript.info

[47] KASÍK, P.: Ušetří vám práci a vyrazí dech. 10 tipů, jak zapřáhnout umělou inteligenci. 2023, [Zob. 17. 08. 2023].
URL https://www.seznamzpravy.cz/clanek/tech-technologie-navody-usetri-vam-praci-a-vyrazi-de

[48] KITHULWATTA, W. M. C. J. T.; JAYASENA, K. P. N.; KUMARA, B. T. G. S.; aj.: Performance Evaluation of Docker-based Apache and Nginx Web Server. 2022, doi:10.1109/INCET54531.2022.9824303.

[49] KNUTH, D.: *The Art of Computer Programming*. Addison-Wesley Professional, tøetí vydání, 1992, ISBN 978-0-201-89683-1.

[50] LAURENČÍK, M.: *SQL: podrobný průvodce uživatele*. Praha: Grada Publishing, 2018, ISBN 978-80-271-0774-2, 211 s., doi:10.1590/ph118067.

[51] MAČUROVÁ, E.: *Aspektové programovanie*. Diplomová práca, Ekonomická univerzita v Bratislave, Fakulta hospodárskej informatiky, 2012.

[52] MEMEL, T.: 5 Reasons Why Linux is More Secure Than Windows — medium.com. 2021, [Zob. 17. 08. 2023].
URL https://medium.com/codex/5-reasons-why-linux-is-more-secure-than-windows-1d036c3d3324

[53] META: Writing Markup with JSX – React — react.dev. 2023, [Zob. 17-Čvc-2023].
URL https://react.dev/learn/writing-markup-with-jsx

[54] METCALFE, J.: Why We Should Embrace Mistakes in School. *Greater Good Magazine*, 2017, [Zob. 23. 08. 2023].
URL https://greatergood.berkeley.edu/article/item/why_we_should_embrace_mistakes_in_school

[55] MICROSOFT: TypeScript. Zob. 14. července 2023.
URL https://www.typescriptlang.org/

[56] MOZILLA: Ajax. 2023, [Zob. 18. 7. 2023].
URL https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

[57] MOZILLA: DNS. 2023, [Zob. 18-Čvc-2023].
URL https://developer.mozilla.org/en-US/docs/Glossary/DNS

[58] MOZILLA: The Document Object Model (DOM). *Mozilla Developer Network*, 2023.
URL https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

[59] MOZILLA: Server-sent events. 2023, [Zob. 18-Čvc-2023].
URL https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events

[60] MOZILLA: Top-level domain. 2023, [Zob. 18-Čvc-2023].
URL https://developer.mozilla.org/en-US/docs/Glossary/TLD

[61] MOZILLA: Web server. 2023, [Zob. 18-Čvc-2023].
URL https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server

[62] MOZILLA: WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly, 2023, [Zob. 25. 5. 2024].

[63] MOZILLA: WebSocket. 2023, [Zob. 18-Čvc-2023].
URL https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

[64] NAWO: Top 6 React state management libraries for 2022. 2022, [Zob. 05. 10. 2023].
URL https://blog.openreplay.com/top-6-react-state-management-libraries-for-2022/

[65] NEAGOIE, A.: The State Of AI Tools And Coding: 2023 Edition — Zero To Mastery — zerotomastery.io. 2023, [Zob. 25. 07. 2023].
URL https://zerotomastery.io/blog/the-state-of-ai-tools-and-coding-2023-edition/

[66] NEFE, J.: Introducing Shadcn UI: A reusable UI component collection. 2023, [Zob. 4. 4. 2024].
URL https://blog.logrocket.com/shadcn-ui-reusable-ui-component-collection/

[67] OSMANI, A.: Photoshop is now on the web! https://medium.com/@addyosmani/photoshop-is-now-on-the-web-38d70954365a, 2023, [Accessed 25-05-2024].

[68] PARADIGM, V.: UML Class Diagram Tutorial — visual-paradigm.com. [Zob. 21-Čvc-2023].
URL https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

[69] PRASAD, M.: MySQL :: InnoDB : Tablespace Space Management — dev.mysql.com. 2019, [Zob. 17-Čvc-2023].
URL https://dev.mysql.com/blog-archive/innodb-tablespace-space-management/

[70] PÁNEK, P.: *Optimalizace prostředí GitLab pro výuku softwarového inženýrství*. Bakalářská práce, Vysoká škola ekonomická v Praze, 2019.
URL https://insis.vse.cz/zp/67071/podrobnosti

[71] PÁSEK, J.: *Generování zdrojových kódů na základě popisu v přirozeném jazyce*. Bakalářská práce, Fakulta informačních technologií, Vysoké učení technické v Brně, 2023.

[72] RAMANAN, N.: Evolution of just-in-time compilation from theoretical performance improvements to smartphone runtime and browser optimizations. *Journal of Information Technology*, 12 2012.

[73] RAPPIN, N.; DVORAK, K.: *Modern CSS with Tailwind: flexible styling without the fuss*. Raleigh, North Carolina: The Pragmatic Bookshelf, second edition vydání, 2022, ISBN 978-1-68050-940-3, xiv, 82 s.

[74] RASTOGI, R.; MONDAL, P.; AGARWAL, K.: An exhaustive review for infix to postfix conversion with applications and benefits. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 2015, str. 6.

[75] REFSNES, D.: SQL Injection — w3schools.com. 2023, [Zob. 04. 08. 2023].
URL `https://www.w3schools.com/sql/sql_injection.asp`

[76] RIVEST, R. L.: MD5 Message-Digest Algorithm. 4 1992, doi:10.17487/RFC1321.
URL `https://www.rfc-editor.org/info/rfc1321`

[77] RYBIČKA, J.: *LATEX pro začátečníky*. Brno: Konvoj: CSTUG, tøetí vydání, 2003, ISBN 80-7302-049-1.
URL `http://www.digitalniknihovna.cz/mzk/uuid/uuid:f1ad47a0-b2eb-11e2-8b87-005056827e51`

[78] SCOTT, M. L.: *Programming Language Pragmatics*. Boston: Morgan Kaufmann, Ètvrté vydání, 2019, ISBN 978-0124104099.

[79] SHIRLEY, P.; MARSCHNER, S.: *Fundamentals of Computer Graphics*. CRC Press, Ètvrté vydání, 2015, ISBN 978-1-4987-6521-3.
URL `https://www.crcpress.com/Fundamentals-of-Computer-Graphics-4th-Edition/Shirley/p/book/9781498765213`

[80] SKŘIVAN, J.: Databáze a jazyk SQL. 2000, [Zob. 08. 09. 2023].
URL `https://www.interval.cz/clanky/databaze-a-jazyk-sql/`

[81] SLOUKA, D.: AI vs ML: co je umělá inteligence a co strojové učení? - Computerworld — computerworld.cz. 2019, [Zob. 04. 08. 2023].
URL `https://www.computerworld.cz/clanky/ai-vs-ml-co-je-umela-inteligence-a-co-strojove-ucen`

[82] STEC, A.: How Compilers Work — Baeldung on Computer Science — baeldung.com. 2023, [Zob. 25. 07. 2023].
URL `https://www.baeldung.com/cs/how-compilers-work`

[83] SUCHORADSKÝ, O.: Testy a jejich užití při hodnocení žáků. *Metodický portál: Články*, 10 2008, ISSN 1802-4785.
URL `https://clanky.rvp.cz/clanek/2666/TESTY-A-JEJICH-UZITI-PRI-HODNOCENI-ZAKU.html`

[84] SUNIL: The Most Popular IDEs for Developers in 2021 - EPIC Software Development — epicsoftwaredev.com. 2021, [Zob. 17. 08. 2023].
URL `https://www.epicsoftwaredev.com/blog/the-most-popular-ides-for-developers-in-2021/`

[85] SUTHERLAND, A.; BOOKER, A.: The answer to life, the universe, and everything? Sum of three cubes, say MIT mathematicians. *MIT News*, 2019.
URL `https://news.mit.edu/2019/answer-life-universe-and-everything-sum-three-cubes-mathemati`

[86] TechTarget: What is the Client-Server Model? - Definition from WhatIs.com — techtarget.com. 2023, [Zob. 17. 7. 2023].
URL `https://www.techtarget.com/searchnetworking/definition/client-server`

[87] THOMAS, G.: A beginner's guide to Docker – how to create your first Docker application — freecodecamp.org. 2019, [Zob. 17-Čvc-2023].
URL `https://www.freecodecamp.org/news/a-beginners-guide-to-docker-how-to-create-your-first-`

[88] TORVALDS, L.; community, G.: Git - Book — git-scm.com. 2023, [Zob. 19-Čvc-2023].
URL `https://www.git-scm.com/book/cs/v2`

[89] TUAYCHAREON, N.; Prodpran, V.; Srithong, B.: ClassSchedule: A web-based application for school class scheduling with real-time lazy loading. In *2018 5th International Conference on Business and Industrial Research (ICBIR)*, 2018, s. 210–214, doi:10.1109/ICBIR.2018.8391194.

[90] TÝNOVSKÝ, L.: *Specifikace požadavků na software ve formátu případů užití*. Bakalářská práce, Západočeská univerzita v Plzni Fakulta aplikovaných věd Katedra informatiky a výpočetní techniky, 2014.

[91] UML: UML Diagrams. 2023, [Zob. 18-Čvc-2023].
URL `https://www.uml-diagrams.org/`

[92] V., A. A.; S., M. L.; SETHI, R.; aj.: *Compilers: Principles, Techniques and Tools*. Pearson Education, Inc., druhé vydání, 2007.

[93] VERCEL: Next.js 9.3 Release Notes. 2022, [Zob. 20. 7. 2023].
URL `https://nextjs.org/blog/next-9-3`

[94] VERCEL: API Routes in Next.js. 2023.
URL `https://nextjs.org/docs/api-routes/introduction`

[95] VERCEL: Next.js. 2023.
URL `https://nextjs.org/`

[96] VERMA, A.: What is CSR/SSR in React? — theamanverma.medium.com. 2019, [Zob. 13. 08. 2023].
URL `https://theamanverma.medium.com/what-is-csr-ssr-in-react-14bc3ea2ca32`

[97] VIROUX, G.: The Great Operating System Battle: Why WSL2 is Winning. 2023, zob. 6. 1. 2024.
URL `https://dev.to/glennviroux/the-great-operating-system-battle-why-wsl2-is-winning-2e89`

[98] W3C: Document Object Model (DOM). 2023.
URL `https://www.w3.org/TR/dom/`

[99] W3C: Internationalization. 2023, [Zob. 26. 07. 2023].
URL `https://www.w3.org/mission/internationalization/`

[100] (WAI), W. W. A. I.: WCAG 2 Overview — w3.org. 2023, [Zob. 2. 10. 2023].
URL `https://www.w3.org/WAI/standards-guidelines/wcag/`

[101] WANKHEDE, C.: Is Android really just Linux? — androidauthority.com. 2023, [Zob. 17. 08. 2023].
URL `https://www.androidauthority.com/android-linux-784964/`

[102] WATANABE, L.: Cross Site Scripting (XSS) Attacks in React — medium.com. 2020, [Zob. 04. 08. 2023].
URL `https://medium.com/dont-leave-me-out-in-the-code/basic-cross-site-scripting-xss-attacks`

[103] WATTS, S.: ACID Explained: Atomic, Consistent, Isolated & Durable. 2020, [Zob. 08. 09. 2023].
URL `https://www.bmc.com/blogs/acid-atomic-consistent-isolated-durable/`

[104] WHATWG: Server-sent events. 2023.
URL `https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events`

[105] YOHEI, U.; OHARA, M.: Performance competitiveness of a statically compiled language for server-side Web applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, s. 13–22.
URL `https://ieeexplore.ieee.org/document/7975266`

[106] YOU, E.: Vite — vitejs.dev. 2019, [Zob. 15-Čvc-2023].
URL `https://vitejs.dev/`

[107] ZHOUF, J.: *Tvorba matematických problémů pro talentované žáky*. Praha: Univerzita Karlova v Praze, Pedagogická fakulta, první vydání, 2010, ISBN 978-80-7290-432-7 (brož.), 299 s., obsahuje bibliografii, bibliografické odkazy a rejstřík, Anglické resumé.

[108] ZIMMERMAN, C.: *The rules of programming: how to write better code*. O'REILLY, first edition vydání, 2023, ISBN 978-1-098-13311-5.