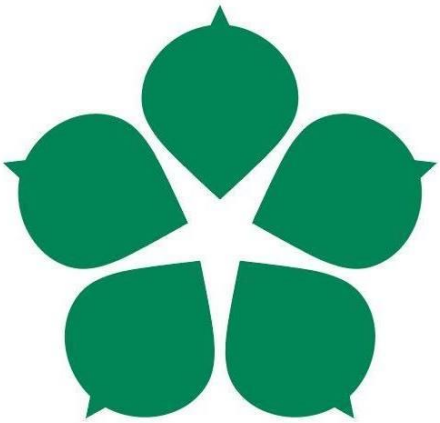
Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta



Webová aplikace pro zadávání, odevzdávání a automatickou kontrolu programátorských prací

Bakalářská práce

Kryštof Lískovec

Školitel: Fesl Jan, Ing. Ph.D.

České Budějovice

2024

Lískovec, K., 2024: Webová aplikace pro zadávání, odevzdávání a automatickou kontrolu programátorských prací. [Web application for assigning, handing in and automatically checking programming assignments. Bc. Thesis, In Czech.] – 43 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Tato bakalářská práce se zaměřuje na analýzu, návrh a implementaci systému používaného pro zadávání, odevzdávání a automatické kontrolování programovacích úloh. Analýza byla založena na funkčních a nefunkčních požadavcích a potřebách uživatelů. Je navržena tak, aby zefektivnila proces hodnocení programovacích úloh a poskytla lektorům efektivní a intuitivní nástroj pro zadávání programovacích úloh studentům. Studenti mají přístup k úlohám prostřednictvím stejné aplikace a mohou odevzdávat svá řešení. V rámci aplikace je sada nástrojů, které automatizují proces hodnocení zdrojového kódu od studentů. Po vyhodnocení je zdrojový kód ohodnocen na základě jeho vstupů a výsledky jsou zobrazeny uživateli, který jej odevzdal.

Anotation

This bachelor‘s thesis focuses on analysis, design and implementation of a system used for assigning, handing in and automatically checking programming assignments. Analysis was based on functional and non-functional requirements and user needs. It‘s designed to streamline the evaluation process of programming tasks and provide lecturers with an efficient and intuitive tool for assigning programming tasks to students. Students are able to access the assignments through the same application and submit their solutions. Within the application is set of tools that automate the process of evaluating source code from students. After the evalution the source code is graded based on it’s inputs and results are displayed to the user submitting it.

Prohlašuji, že jsem autorem této kvalifikační práce a že jsem ji vypracoval pouze s použitím pramenů a literatury uvedených v seznamu literatury.

České Budějovice, dne 16. 7. 2024

……

Kryštof Lískovec

Obsah

1 Úvod

V dnešní době neustále stoupají nároky na automatizaci procesů a scénářů provázející nás našimi životy. Hledají se nové způsoby, jak věci automatizovat a optimalizovat. Neustále je tento proces ovlivňován novými a komplexnějšími požadavky. Nicméně pokrok moderních technologií se nezastavil, nýbrž naopak byl donucen se přizpůsobit. Například v době pandemie v rámci omezení fyzického kontaktu jsme se stali závislí na moderních technologiích a bylo potřeba adaptovat technologie této situaci. I dnes stále používáme tato sofistikovaná řešení, které nám umožňují vykonávat naše povolání, vzdělávat se a setkávat se s našimi nejbližšími, když fyzický kontakt není adekvátní. Skutečnost omezit fyzický kontakt donutil vývojové týmy moderních webových systémů brát vyšší ohledy na stávající služby a aplikace dostupné online prostřednictvím internetu. Týmy se soustředily na jejich stabilitu, přístupnost, výkon, ale především začaly vznikat zcela nová řešení, určené pro tyto účely.

1.1 Problematika a motivace

Jedna z mnoha velice ovlivněných oblastí pandemií se stala školní sféra. Udržet úroveň vzdělání bez fyzického kontaktu na stejné úrovni jako předtím je velmi náročný úkol. Technologie nebyly ještě připraveny plně nahradit osobní výuku. Vznikla řada nesrovnalostí, nedorozumění a neefektivních přístupů, jež časově a psychicky zatěžují studenty i vyučující. Svět však byl schopen navrhnout taková řešení umožňující pokračování výuky, a dokonce k objevení zcela nových možností, jak výuku vést. S jedním takovým problémem, kterým se zabývá i tato práce, jsme se setkali konkrétně u výuky programování. Na první pohled se předmět může zdát jako naprosto perfektní kandidát pro distanční výuku. U studentů lze již předpokládat jistou technickou zdatnost a přístup k dostatečně výkonným počítačům umožňující vzdálenou komunikaci a interakci. Co se týče samostatných programovacích prací, jde často o programy napsané ve vyučovaném programovacím jazyce, které lze vyučujícímu odevzdat skrze internet. Problém však nastává po odevzdání. S růstem počtu odevzdaných prací stoupá časová náročnost ruční kontroly a hodnocení těchto prací. Dalším problém může vzniknout, pokud neexistuje podrobná specifikace od vyučujícího, jak patřičný úkol strukturovat. Může vzniknout nekonzistence formátu dále zvyšující časovou náročnost zpracovávání odevzdaného úkolu nebo nesoulad s verzemi kompilátorů, překladačů nebo knihoven, které odevzdané programy používají. S objemem a složitostí programovacích prací souvisí také bezpečnost těchto programů. Uvnitř odevzdané práce může být skryt potencionálně nebezpečný kód, který při nedůkladné kontrole může ohrozit počítač a data

vyučujícího. V neposlední řadě zde vzniká problém s poskytnutím zpětné vazby studentovi v případě, že program nešel vyučujícímu spustit.

Tato bakalářská práce se zaměřuje na návrh, vývoj a implementaci webové aplikace pro zadávání, odevzdávání a automatickou kontrolu programátorských prací. Aplikace má za cíl usnadnit proces hodnocení, snížit zátěž na vyučující a poskytnout studentům okamžitou zpětnou vazbu.

1.2 Cíle práce

Hlavním cílem této práce je navrhnout a vyvinout funkční webovou aplikaci, která umožní efektivní zadávání programátorských úkolů vyučujícími, jednoduché odevzdávání prací studenty a automatickou kontrolu a hodnocení odevzdaných prací. Aplikace musí být naprogramovaná pomocí moderních webových technologií, musí být patřičně zabezpečená proti vnějším i vnitřním útokům. Měla by automaticky brát data z aplikačního rozhraní systému IS Stag a umožnit autentizaci pouze pro studenty JČU. Služba také musí být jednoduchá a snadno použitelná studenty a vyučující. Zároveň musí umožnit vykonávání pokročilejších úkonů na základě přednastavených hodnot lektorem jako např. automatické vyhodnocení samostatné domácí práce na základě výsledků analýz provedeným systémem pro automatickou opravu programátorských prací a synchronizací dat s portálem IS Stag. Dalším cílem je uchovávání a archivování zadání a vypracování samostatných domácích úkolů do databáze a rovněž ukládání odevzdaných souborů na pevné úložiště.

2 Teoretické základy

Vývoj webové aplikace pro zadávání, odevzdávání a automatickou kontrolu programátorských prací vyžaduje důkladné pochopení stávajících technologií a metodik v této oblasti. Tato kapitola poskytuje přehled o současných řešeních, technologiích použitých při vývoji aplikace a základních konceptech automatické kontroly kódu. Je důležité porozumět těmto teoretickým základům, aby bylo možné efektivně navrhnout a implementovat systém, který splňuje požadavky uživatelů a přináší významné zlepšení v oblasti hodnocení programátorských prací.

2.1 Základní principy vývoje webové aplikace

Popíšeme si nejzákladnější principy pro vývoj webových aplikací, které následně obohatíme znalostmi v následující kapitole. Způsoby, kterými lze vytvořit webovou aplikaci je nespočet, ale existuje několik metodik a základních principů, které by měl webový vývojář znát nebo dodržovat.

2.1.1 Webová stránka

Webová stránka neboli také často interpretována jako klientská část aplikace nebo frontend je klíčovou částí, která zajišťuje interakci uživatele se systémem. Úplným základem jsou jazyky: HTML (Hypertext Markup Language) a CSS (Cascading Style Sheets). Dohromady jako základní stavební kameny tvoří statickou webovou stránku, kde HTML definuje jak a které prvky webové stránky se zobrazují, CSS popisuje stylování a vzhled stránky. Takto napsaná stránka může být následně sestavena a zobrazena přes webový prohlížeč. Logiku a určitý stupeň interaktivity může do těchto stránek promítnout samotný web server psaný v programovacích jazycích jako je např. PHP (Hypertext Preprocessor), který webovou stránku poskytuje uživatelům. Webserver také může dále obohatit dokument o data uložená na serveru. Tento stupeň interaktivity však vyžaduje neustálé obnovování stránky, aby server mohl obsah stránky modifikovat. Proto se k webovým stránkám standardně přidává interpretovaný skriptovací jazyk JavaScript, který umožňuje na straně klienta modifikovat dokument ať už ze strany HTML nebo CSS. Mezi časté použití na webových stránkách řadíme např. reagování a zpracovávání na uživatelského vstup, nasměrování uživatele na určité sekce webové stránky nebo dotazování se webového serveru na patřičná data, a to bez nutnosti nového načítání stránky V tomto smyslu mluvíme o asynchronním typu zpracovávání webových stránek označován jako AJAX (Asynchronous JavaScript and XML).

Operace v tomto případě probíhají „na pozadí“ na straně JavaScript jazyku bez vědomí uživatele.

Ke snazší manipulaci s HTML byl navrženo rozhraní pro objektový přístup k XML nebo HTML dokumentům pod názvem DOM (Document Object Model). DOM dnes obsahuje každý prohlížeč schopný spouštět JavaScript. Tento model reprezentuje dokument jako strom. [1]

2.1.2 Webový server

Webový server je klíčovou komponentou při vývoji webových aplikací, protože zajišťuje doručování webových stránek uživatelům přes internet. Webový sever má několik funkcí: přijímání požadavků od klientů, zpracovávání požadavků, odesílání odpovědi [2]. Server může odesílat statický obsah skládající se ze statického HTML a CSS, které může působit dynamicky po přidání patřičného JavaScript kódu. Obsah může být také dynamicky modifikován na základě požadavku klienta. K obsahu se mohou přidat informace ze souboru, databáze nebo jiného zařízení. K vytváření obsahu se dost často používají technologie jako PHP, ASP.NET, Node.JS nebo Python kombinací s relačními databázemi jako jsou MySQL, Microsoft SQL nebo MariaDB a ke komunikace mezi těmito moduly se nejčastěji používá rozhraní REST API. [3]

Velice zmiňovaný software pro tyto účely je Apache HTTP server. Apache web server vznikl v roce 1995 pod skupinou Apache Group tehdy skládající se z několika webmasterů s cílem rozšířit v minulosti velmi používaný web server software NCSA httpd vyvinutý Robem McCoolem. Software ale obsahoval řadu nedostatků, jenž byly opraveny jednotlivými webmastery v rámci rozšíření webového serveru. Skupina následně vydala v prosinci 1995 plno verzi 1.0 Apache a méně, než rok po vytvoření skupiny se stal Apache server nejpoužívanější web server na světě. Dnes se řadí mezi nejrozšířenější a nejstarší webové servery a je známý svou flexibilitou a širokou podporou modulů. [4]

Další známou technologií je web server Nginx. Nginx web server je brán jako jednodušší varianta Apache. Vyvinut Igorem Sysoevem v roce 2004 pro ruskou tehdy druhou největší webovou stránku Rambler. Cílem bylo vytvořit výkonný webový server, který dokáže rychlostí překonat Apache HTTP web server. Momentálně jsou však rychlosti obou web serverů si dost podobné a nelze s jistotou jednoznačně určit vítěze. [5]

Je nutné dodat, že tyto technologie nejsou jediné, které lze použít jako webový server. Pro tyto účely je možné použít i web server knihovny nebo moduly programovacích jazyků jako jsou například Node.JS, Python a Golang.

2.1.3 REST API

REST (Representational State Transfer) API je architektonický styl pro navrhování síťových aplikací. REST API umožňuje komunikaci mezi klientem a serverem pomocí standardních HTTP metod. Tento přístup je široce používán při vývoji webových aplikací, protože je jednoduchý, škálovatelný a flexibilní. Tento styl se řídí několika důležitými principy. Požadavek od klienta k serveru musí obsahovat veškeré potřebné informace k jeho zpracování. Server funguje v stateless režimu neboli neukládá žádný stav mezi požadavky. Oddělením uživatelského rozhraní od webového serveru zlepšuje přenositelnost uživatelského rozhraní a škálovatelnost aplikace. REST API nabízí jednotné rozhraní zjednodušující nezávislý vývoj jednotlivých částí systému. [6]

Rozhraní využívá standardní HTTP metody pro komunikaci mezi klientem a serverem jako jsou např. GET, POST, PUT a DELETE. Jednotlivé požadavky lze rozšířit o parametry poskytující serveru větší kontext a potřebná data pro zpracování požadavku. Prvním typem parametru jsou tzv. path parametry (viz obr. 1). Path parametry využíváme primárně k identifikaci zdroje dat. Hodnota parametru se vkládá jako proměnná část URL, která se následně extrahuje a zpracuje na straně serveru. Dalším možný parametrem je query parametr. HTTP klient query parametry zanáší do URL jako klíč a hodnotu na konci URL. Query parametrová část URL musí být od URL cesty oddělena znakem „?“ (viz obr. 1). Pokud je třeba přidat větší počet query parametrů, oddělíme je znakem „&“. Tento typ parametru používáme pro bližší specifikaci potřebných dat na straně klienta např. v případě stránkování obsahu potřebujeme na server dodat informace o minimálním počtu a čísle stránky záznamů. Pro podrobnější specifikaci požadavku můžeme použít parametry hlavičky požadavku, které jsou součást HTTP hlavičky (viz obr. 1). Parametry hlavičky se často používají pro identifikaci nebo autentizaci klienta. [6]

```
# Příklad path parametru jako identifikátor kurzu
/api/kurzy/{kurz_identifikator}/ukoly

# Příklad query parametru u stránkování obsahu
/api/clanky?strana=1&pocet=10

# Příklad header parametru jako autentizační token
Authorization Bearer token
```

Obr. 1 příklady REST API parametrů. Zdroj: vlastní tvorba

V rámci REST API používáme pro širší komunikaci také formáty dat, které jsou snadno čitelné a editovatelné jak lidmi, tak i stroji. Nejčastěji zahrnujeme formáty JSON, YAML nebo XML. Často jde o požadavky o vytvoření nového nebo editaci existujícího záznamu na straně webového serveru. [6]

Co se týče zabezpečení, pak nejjednodušší volbou je HTTP basic authentication, která k autentizaci používá uživatelské jméno a heslo zakódované v Base64. Při uniknutí nezašifrovaných hlaviček požadavku se však útočníkovi naskytne možnost trvalého přístupu, dokud se uživatelské údaje nezmění. Útočníka nelze ani nijak do té doby odhlásit a zamezit přístup k citlivým informacím. Navíc u tohoto přístupu se často zapomíná na implementaci brute force prolomení údajů. Pro bezpečnější zabezpečení se používá standard OAuth2.

[7] OAuth2 je otevřený standard pro autorizaci, který umožňuje aplikacím získat omezený přístup k uživatelským účtům na jiných službách. Tento přístup je implementován pomocí přístupových tokenů vydaných autorizací serveru třetí strany. Principem je přihlášení uživatele do bezpečného systému třetí strany, která vrací autorizační grant, kterým lze získat přístupový token, jenž může aplikace použít pro získání potřebných dat pro autorizaci uživatele. V rámci tohoto standardu může být použít standard Json Web Token (JWT) pro vytváření tokenů. [7]

JWT je standard pro vytváření tokenů, které lze použít pro autentizaci a výměnu informací mezi stranami. JWT tokeny jsou kompaktní, samostatné a přenositelné. JWT token se skládá se tří částí (viz obrázek 2): **header**, jenž obsahuje metadata tokenu, jako je použitý algoritmus pro podepisování a typ tokenu. Dále **payload** obsahuje informace o subjektu (uživateli) dalších údajích poskytnuté aplikací vytvářející token. Na konci JWT tokenu je **signature**, který slouží k ověření integrity tokenu a jeho autenticity. Vytváří se pomocí

zvoleného algoritmu (např. HMAC SHA256) z hlavičky, těla a tajného klíče sloužící k zašifrování celého tokenu. [8]

Obr. 2 části JWT tokenu. Zdroj jwt.io

2.1.4 Model-View-Controller (MVC)

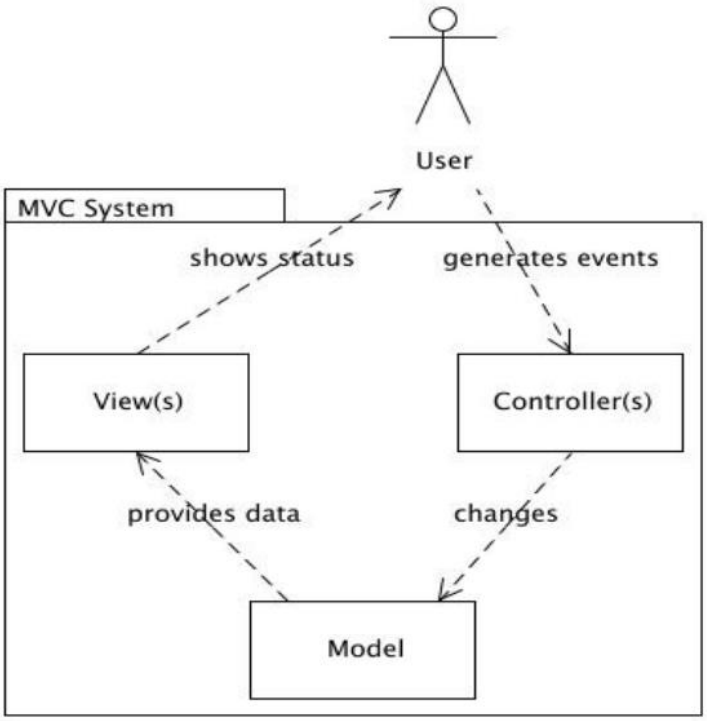
MVC je jedním z nejčastěji používaných návrhových vzorů pro vývoj webových aplikací. Dělí aplikaci do tří hlavních komponent: **model, view** a **controller**.

Model spravuje data aplikace a zajišťuje komunikaci s databází. Obsahuje definice logiky, specifikace funkcí a obchodní pravidla. Zpravidla nemá obsahovat logiku prezentace dat ani řídit uživatelské interakce. [9]

View je vrstva, která má na starosti uživatelské rozhraní aplikace. Zobrazuje výsledky dat obsažených v modelu a poskytuje uživateli požadovaná data, přičemž skrývá zbytečné hodnoty a atributy. Výhody zobrazení zahrnují enkapsulaci prezentace. Ve webové aplikaci zahrnujeme celou HTML, CSS a JS část jako view. View v této podobně používá skriptovací jazyk JS a snadno se integruje s technologií AJAX. View by nemělo obsahovat logiku pro vytváření, ukládání a manipulaci s modelem. [9]

Controller jako poslední vrstva reaguje na požadavky uživatelů a provádí akce na základě jejich požadavků. Spojuje uživatele se systémem a spravuje model i view. Řídí tok dat v modelu a aktualizuje view při změně dat. Controller přijímá data, pracuje s nimi a

provádí akce, které mění stav datového modelu. Po komunikaci s modelem vybírá správný view k zobrazení. [9]



Obr. 3 schématické znázornění MVC. Zdroj: [9]

2.1.5 Responzivní design

Responzivní design je přístup k tvorbě webových stránek, který zajišťuje optimální zobrazení a interakci napříč různými zařízeními a velikostmi obrazovek. Tento koncept je klíčový pro moderní webové aplikace, protože uživatelé přistupují k webu z různých zařízení, jako jsou stolní počítače, tablety a mobilní telefony. Hlavní principy pro správný responzivní vývoj řadíme **flexibilní mřížku**, **media queries**, **flexibilní obrázky a média** a **mobile-first přístup**. [10]

Flexibilní mřížka (Fluid Grid Layout) je první metodika, která umožňuje prvkům přizpůsobit se různým šířkám obrazovky. Velikosti prvků se často definují v procentech namísto pevných pixelů. Díky tomuto principu se z fixního rozložení komponent, kde komponenty jsou staticky na jenom místě, stává dynamická stránka, jenž modifikuje velikost těchto prvků dle používaného rozlišení obrazovky nebo okna prohlížeče. [10]

Media queries nám umožňují aplikovat různé styly pro různé velikosti obrazovky a zařízení. Tímto způsobem lze přizpůsobit rozvržení stránky definicí několika stylů pro různé rozlišení a zajistit optimálního rozložení na různých typech zařízení. [10]

Obrázky a další média by měly být škálovatelné a přizpůsobitelné velikosti obrazovky. Použití technik jako „max-width: 100%“ zajišťuje, že obrázky nebudou přesahovat ohraničení kontejnerů.

Mobile-first přístup klade důraz na tvorbu webových stránek a aplikací primárně pro mobilní zařízení. Tento přístup je založen na myšlence, že navrhování pro nejmenší obrazovky a zařízení s omezenými zdroji jako první poskytuje solidní základ, na kterém lze budovat komplexnější a bohatší verze pro větší obrazovky. Mobile-first design se stal populární díky rychlému růstu používání mobilních zařízení pro přístup k internetu. Tento design nezahrnuje pouze responzivní design, ale také přizpůsobení uživatelského rozhraní tak, aby bylo jednoduše použitelné a intuitivní i na malých zařízeních. Takto navržené webové stránky musí být výkonově správně optimalizované a musí být velikostně co nejmenší, aby se urychlilo načítání i při pomalejším připojení k internetu. [10]

2.2 Dnešní web development

V této podkapitole dále obohatíme znalosti z předchozí kapitoly o dnešní přístupy vývoje webových stránek. Dnešní web development zahrnuje širokou škálu technologií, které umožňují vývojářům vytvářet výkonné, škálovatelné a uživatelsky přívětivé webové aplikace, které poskytují stabilní architekturu pro vývoj.

2.2.1 Frontend

Frontend neboli klientská část se v dnešním web developmentu podstatně liší oproti klasickému vývoji webových stránek. Neustálý tlak na zvyšování interaktivity, dynamičnosti a rychlosti webový stránek vyvrcholil k tomu, že je potřeba vyvinout nástroje, které umožňují mnohem komplexnější zacházení s HTML a DOM. Nároky na interaktivitu a rychlost webových stránek rozšířily klientskou část o velmi důležité **stavy**, které ovlivňují svou hodnotou vzhled a obsah stránky a redukují počty dotazů na webový server, protože přesouvají prezentační logiku na stranu klienta, jenž je celá obsluhována skrze JavaScript. Při snížení počtu přesměrování a synchronních dotazů na webový server klesá nutnost znovu načítat veškerý obsah. Protože se po přesunutí prezentační logiky naskytla tendence generovat celé sekce a modifikovat je přes JavaScript, vznikl nápad shlukovat tyto sekce do komponent. Takto navržené sekce se mohou snadno znovu použít v jiné sekci stránky a existuje zde i možnost implementace parametrů, kterými lze dále ovlivňovat takto definované komponenty.

V tento moment se už nebavíme o prostých dokumentech obohacený o jednoduché interaktivní prvky, ale o plnohodnotných aplikacích. Kombinací přesunutí prezentační logiky

na klientskou část a snahu o to minimalizovat synchronní dotazy na web server vzniká také termín Single Page Application neboli SPA. [12]

SPA aplikace využívají tohoto potencionálu JavaScript do maximálních výšin, kdy tyto aplikace obsahují minimum čistého HTML a zbytek stránky je nadefinován v JS v rámci komponent. Obsah stránky se následně vygeneruje z těchto komponent přes JS po jednorázovém načtení. Další obsah stránky aplikace získává nejčastěji přes REST API webového serveru, a to asynchronním stylem, který nepřerušuje činnost a zobrazení uživatele a umožňuje velmi dynamickou navigaci skrze webovou stránku. Takto navržené aplikace dosahují vysoké rychlosti a plynulosti a svým chováním připomínají desktopové aplikace. Největší výhoda SPA aplikací je zároveň její slabina. Pokud je obsah aplikace celý řízen JavaScript, vzniká zde problém u optimalizace pro vyhledávače (SEO), kdy vyhledávače indexují a kategorizují webové stránky na základě prezentovaného HTML. V tomto případě se však HTML generuje až po spuštění JavaScript kódu, který obyčejný vyhledávací web crawler nedokáže spustit a tím pádem se pro tohoto web crawlera stránka jeví skoro prázdná. [12]

Tyto moderní principy daly vznik tzv. JavaScript frameworkům. Frameworky představují systémy vývoje webových aplikací, které dávají vývojáři podnět k tomu navrhovat dynamické a interaktivní aplikace a udržovat kód v jednotném stylu, který framework nastiňuje. Mezi tyto typy SPA frameworků patří **React.js**, **Vue.js**, **Angular** a **Svelte**. [12]

JavaScript frameworky disponují několika moderními přístupy týkající se ovlivňování jednotlivých komponent a stavů. Jedním takovým nástrojem je virtuální DOM. Jde o efektivní implementaci skutečného DOM, kterou frameworky používají pro minimalizaci změn ve skutečném DOM. Při každé změně stavu framework vytváří nový virtuální DOM a porovnává jej s předchozím, což umožňuje efektivnější aktualizaci pouze těch částí DOM, které se změnily.[12]

2.2.2 Frontend webové frameworky

React.js

React.js je populární JavaScript knihovna vyvinutá společností Facebook, která slouží k vytváření uživatelských rozhraní. Oproti ostatním podobným frontend nástrojům řadíme React.js spíš mezi **knihovny** nežli frameworky, ale díky své komplexnosti a podobným use-casem ho s ostatními frameworky porovnáváme. Nenutí vývojáře dodržovat určitou strukturu aplikace a spíše nabízí velkou sadu funkcí, které vývojář může kdykoli sám zavolat. React.js se zaměřuje na komponentně orientovaný vývoj. Komponenty jsou základními stavebními

bloky React aplikace. Každá komponenta představuje část uživatelského rozhraní, kterou lze znovu použít. Komponenty mohou být definované funkcí (React Hooks) nebo třídou. React používá syntaxi JSX, která se podobá HTML, ale zároveň umožňuje psát JavaScript kód přímo dovnitř HTML a použít JS výrazy k dynamické manipulaci obsahu stránky. Na komponenty se v React.js odkazuje skrze název funkce nebo třídy uzavřený v HTML tagu. Nabízí one-way binding a k aktualizaci jednotlivých komponent React využívá svojí implementaci virtuálního DOM. Největší výhodou React.js je jeho rozšířenost a obrovská komunitní podpora. [12]

Vue.js

Dalším, tentokrát už frameworkem, je Vue.js vyvíjené stejnojmennou firmou. Vue.js se stal populárním díky své jednoduchosti, flexibilitě a vysoké výkonnosti. Stejně jako u React.js jsou zde komponenty jako základní stavební bloky a dají se znovu používat. Vue oproti React.js umožňuje vytvářet single file komponenty. Jde o soubory s příponou „.vue“ obsahující všechny tři jazyky pro tvorbu webových stránek HTML, JavaScript a CSS. Místo JSX používá Vue.js vestavěné direktivy v HTML části, které umožňují různé typy chování a napojení na JavaScript část. Například při použití direktivy „v-if“ zobrazujeme takto označený element v případě, že podmínka za „v-if“ je pravdivá. Veškeré komponenty jsou na pozadí aktualizovány pomocí virtuálního DOM. Tento framework není tak komunitně rozšířený jako React.js, ale nabízí ucelené vývojové prostředí schopné navést vývojáře ke správnému vývoji rychlé a efektivní webové aplikace. [12]

Angular

Angular je robustní frontend JavaScript framework vyvinutý společností Google, který je určen pro vývoj dynamických a **komplexních** webových aplikací. Angular poskytuje vývojářům komplexní sadu nástrojů a funkcí, které usnadňují vytváření, údržbu a rozšiřování aplikací. V porovnání s ostatními frameworky Angular organizuje aplikace do logických celků nazývané moduly psané v nativní JS souborech, ale spíše v TypeScript souborech. Moduly obsahují deklarace komponent a služeb a definují, které další moduly importují. Služby uvnitř modulů jsou třídy, které obsahují sdílenou logiku a mohou být snadno injektovány do komponent a jiných služeb pomocí mechanismu dependency injection. HTML je zde rozšířeno o direktivy stejně jako u Vue.js. Angular nepoužívá virtuální DOM pro aktualizaci komponent, ale používá vlastní mechanismus detekce změny stavu. Zahrnuje se zde zachycení všech asynchronních operací pomocí knihovny Zone.js a OnPush strategie, kterou vývojáři mohou sami používat pro optimalizaci změny stavu. Sám o sobě Angular obsahuje spoustu modulů a direktiv, které pokrývají většinu potřeb webového vývojáře. Díky své modularitě, silné typové

kontrole a obsáhlosti z nástrojové stránky je Angular vhodným kandidátem na komplexní frontend aplikace. [12]

Svelte

Svelte je moderní frontend framework pro vývoj uživatelských rozhraní, který se liší od tradičních frameworků jako React nebo Vue tím, že většinu práce provádí během kompilace. To znamená, že výsledné aplikace jsou vysoce výkonné a mají menší velikost, protože nepotřebují runtime knihovnu. Dalším rozdílem od tradičních frameworků je jeho jednoduchost. Svelte nabízí jednoduché deklarování reaktivních proměnných, které automaticky aktualizují uživatelské rozhraní při změně hodnot a možnost tyto proměnné propojit skrze two-way binding na elementy pomocí speciálních HTML direktiv. Svelte komponenty jsou psané ve speciálních souborech s příponou „.svelte“, které podporují single file komponenty. Framework nabízí několik dalších nástrojů jako je efektivní způsob správy stavu sdílený napříč komponentami pomocí „store“. Stejně jako Angular Svelte nepoužívá virtuální DOM a místo něj se zaměřuje na optimalizaci kódu během kompilace a přímé aktualizace DOM, kde Svelte je schopné přesně rozpoznat, kterou část aktualizovat. Největší výhodou Svelte je jeho jednoduchost a intuitivní chování. Nabízí také vysoký výkon a velmi malou velikost balíčku, do kterého se aplikace kompiluje. Komunita Svelte není příliš velká oproti konkurenčním frameworkům, ale díky svému modernímu charakteru stále roste. [14]

2.2.3 TypeScript

V poslední frontend podkapitole probereme typy v JavaScript. JavaScript sám o sobě nemá statickou typovou kontrolu. Tento fakt umožňuje velmi rychle v JavaScript vyvíjet a lehce se učit jeho principy, ale postupně zde nastává problém, kdy přiřazení hodnoty nesprávného typu nebo volaní metody na objektu, který není definován se často projeví až během chodu programu. Bez jasně daných typů je těžké zjistit, jaké argumenty přijímá definovaná funkce a jaký typ naopak vrací přinášející nedorozumění mezi vývojáři a klesá tak udržitelnost kódu zvláště u komplexních projektů. Nástroje a funkce běžně dostupné v editorech kódu jako refaktorovací nástroje mají omezené schopnosti.

TypeScript je open-source programovací jazyk vyvinutý společností Microsoft, který rozšiřuje JavaScript o statickou typovou kontrolu uživatelského kódu, ale i vestavěných nástrojů. Kód napsaný v TypeScript jazyce je následně transpilován do čistého JavaScript a při transpilaci se patřičně provádí typová kontrola, kterou lze dle nastavení zpřísnit nebo zmírnit. Díky tomu, že tato nadstavba jazyku JavaScript zlepšuje kvalitu kódu a zmírňuje či

úplně odstraňuje problémy chybějící typové kontroly, je TypeScript považován za standard, který dnes nalezneme u většiny knihoven a frameworků.

2.2.4 Backend

V následující podkapitole se budeme věnovat dnešní backend neboli serverové části webové aplikace. Stejně jako u FE se dnešní způsob vývoje serverové části liší od způsobu používaném v minulosti. Klade se zde důraz především na architekturu, rychlost, bezpečnost a škálovatelnost aplikace s vysokou integritou dat. Staré aplikace, většinou nativně v jazyce PHP psané ve funkcionálním stylu, jsou svojí architekturou velmi jednoduché a zpočátku vývoje velmi jednoduché na rozvinutí. Principem těchto aplikací je odchycení požadavku, jeho zpracování, získání dat z relační databáze dle specifikací požadavku a odbavení požadavku odesláním modifikovaného HTML dokumentu.

V rámci malých webových služeb je tento přístup zdánlivě dostačující, pokud se věnuje dostatečně času bezpečnosti při zpracování požadavků. Spousta aplikací vyvíjených tímto stylem se stalo terčem nejčastěji SQL injection útoků. Šlo o jednoduchý typ útoku, který využíval neošetřené vstupy např. přes query parametry a umožňoval spouštět jakýkoli SQL kód na straně serveru. Je tedy nutné vstupy řádně ošetřit sanitací např. využitím „prepared statements“ a parametrů v SQL. [15]

Jakmile ale malá webová aplikace začíná růst ve větší aplikaci obsahující mnoho komplexních funkcionalit a logiky, začíná být aplikace náročnější na údržbu a vývoj nové funkcionality. Dost často se tento problém u těchto aplikací řeší neustálým vytvářením nových modulů a servisních komponent, které poté většinou nikdy nejsou zpětně použity znovu. Narůstá tak redundance kódu a při změně chování funkcionality v globálním měřítku aplikace, se musí všechny tyto moduly a servisy zpětně procházet. Dalším problémem při růstu aplikace nastává u rychlosti. Protože je zde preferován způsob odbavování celé stránky přes jediný požadavek, musí se veškerá potřebná data získat v rámci tohoto zpracování požadavku, nejčastěji na jednom vlákně procesu, kde zvláště u komplexních SQL dotazů dochází ke drastickému zpomalení načítání stránky.

Objevil se zde prostor na zlepšení situace. Architektonickému problému se dá z části vyhnout použitím jiného programovacího jazyku jako např. Java nebo C# nebo jiné objektově orientované prostředí a problému s jedním komplexním požadavkem na jednom vlákně se dá zamezit použitím vláken nebo AJAX požadavků a správně připraveného REST API. Nicméně vývoj byl v tomto ohledu komplikovanější a dražší. Hledalo se řešení, které by usnadnily

vývoj bezpečných, rychlých a škálovatelných aplikací. S řešením přišly samostatné knihovny a nástroje, které byly následně vestavěné do stabilních backend frameworků.

Velice populárními frameworky se staly ORM (Object-relational mapping) frameworky. ORM frameworky jsou nástroje, které mapují data mezi relační databází a objektově orientovaným prostředím. Umožňuje s tabulkami v relační databázi pracovat, jako kdyby šlo o klasické objekty. Vývojáři jsou tedy do jisté míry odstíněni od klasického SQL. Frameworky tímto způsobem velmi usnadňují vývojáři provádět čtení a zápis dat bez nutnosti se starat, jak se jednotlivé SQL operace provedou. Avšak úplně odstínění od SQL zde nelze dosáhnout, protože u určitých komplexnějších operací je nutné analyzovat generované SQL dotazy frameworkem, aby se dosáhlo optimálních a správných výsledků. ORM frameworky mimo jiné umožňují registrovat event listener pro různé operace frameworku a reagovat na nebo ovlivnit jejich provedení, čímž se přesouvá logika většinou provozovaná na straně databáze přes trigger funkce na stranu webové aplikace. Mezi tyto frameworky se řadí např. **Hibernate**, **Eloquent**, **Prisma**, **Gorm** nebo **Entity Framework**. Existují i takové frameworky použitelné pro nerelační databáze pod názvem ODM frameworky. Velice známým takovým frameworkem je např. **Mongoose** pro MongoDB databázi. [16]

U aplikací, kde se používá server side rendering stránky kvůli jednoduchosti nebo zlepšení SEO se hojně využívají templating jazyky. Jde o speciální jazyky používané k vytváření dynamického obsahu v HTML. Umožňuje oddělit logiku aplikace od prezentační vrstvy už na serverové části aplikace a mnohem snazší práci s prezentací dat. Na základě používané platformy se můžeme setkat s **Blade**, **EJS**, **JSP** nebo **Razor**.

2.2.5 Backendové webové frameworky

Spring

Spring je rozsáhlým webovým frameworkem ideální pro vývoj enterprise aplikací v Javě vyvíjen společností SpringSource. Byl vyvinut za účelem zjednodušení vývoje a poskytuje rozsáhlou sadu funkcí a nástrojů pro vývoj a testování. Spring využívá principu Inversion of Control neboli přenášení kontroly nad vytvářením a spravováním objektů z vývojáře na framework. Spring tohoto dosahuje přes Dependency Injection. K přístupu datům využívá další framework soustředěný speciálně na správu dat v relačních databázích s názvem Hibernate. V případě vytváření dynamického obsahu se nejčastěji Spring kombinuje s templating jazykem JSP. Dále Spring obsahuje nástroje pro zajištění bezpečnosti, autentizace a autorizace v aplikacích skrze Spring Security modul. Rozsáhlý systém modulů je podporován bohatou dokumentací a komunitou. [17]

Laravel

Ve světě PHP je velice populární framework Laravel vyvíjen stejnojmennou společností. Je zaměřen na jednoduchost, robustnost a udržitelnost kódu. I když PHP svou pověstí je objektivně bráno jako zastaralá technologie, Laravel poskytuje čistou a intuitivní nejnovější PHP syntaxi s moderními funkcemi, anotacemi a idiomy přinášející lepší čitelnost a udržovatelnost kódu. Prostředí Laravel lze dále rozšiřovat o další moduly umožňující např. autentizaci, autorizaci, administraci dat, monitoring nebo debugging díky balíčkovacímu správci Composer. Díky progresivnímu přístupu k vývoji webových aplikací, bohatému ekosystému a aktivní komunitě je Laravel velice populární volbou pro vývoj webových aplikací. Jako ORM Laravel používá Eloquent a jako templating jazyk zde nalezneme obvykle Blade. [18]

ASP.NET

Dalším frekventovaně používaným stabilním frameworkem je ASP.NET od korporace Microsoft. Tento ekosystém využívá primárně jazyky C# nebo Visual Basic. Je postaven na .NET platformě. Framework je rozdělen do několika knihoven, kde ASP.NET Web API obsluhuje vytváření HTTP služeb, Razor je používán jako templating framework, SignalR umožňuje přidání obousměrné real-time komunikace mezi serverem a klientem a Entity Framework, který zastupuje roli ORM frameworku. Stejně jako Spring se hodně zde uplatňuje princip Dependency Injection ke snadné správě objektů a jejich závislostí. V minulosti bylo možné ASP.NET webové aplikace vyvíjet a nasadit pouze ve Windows operačním systému. Dnes ale existuje varianta frameworku ASP.NET Core, která je open-source a lze provozovat na většině nejznámějších operačních systémů. Hlavní výhodou je velká podpora od velká společnosti Microsoft a integrace v rámci .NET ekosystému nabízející širokou škálu knihoven a nástrojů. ASP.NET je výborným kandidátem pro střední nebo velké webové aplikace díky své škálovatelnosti a výkonu. [19]

Express.js

Express.js je minimalistický a flexibilní framework pro vývoj webových aplikací a API v runtime serverovém prostředí Node.js pro JavaScript. Projekt je veden společností OpenJS Foundation. Jeho návrh se soustředí na jednoduchost a urychlení vývoje serverových aplikacích. Poskytuje širokou sadu funkcí pro správu požadavků a middlewares. Na rozdíl od předešlých zmíněných frameworků je Express.js velmi minimalistický a neobsahuje vazby na ORM framework ani na templating jazyk. Je možné však potřebné funkcionality přidávat v rámci JavaScript modulů. Tento princip je založený na modulárním JavaScript ekosystému

postavený na správci JavaScript balíčků Npm, do kterého velmi hojně přispívá bohatá komunita různými balíčky. Pro vývojáře je Express.js velmi atraktivní volbou, protože s pomocí runtime prostředí Node.js používá stejný jazyk jako klientská část. Tento fakt umožňuje sdílení codebase mezi backend a frontend např. interfaces, funkce, modely nebo celé npm knihovny. Velice populární ORM frameworkem pro Express.js je Sequelize nebo Prisma. Mezi hojně používané templating jazyky jsou EJS nebo Pug. Express.js jde mimo jiné zmíněné vlastnosti jako validaci požadavků, autentizaci, autorizaci a monitoring rozšířit o typovou kontrolu pomocí jazyku TypeScript. Express.js nabízí velice jednoduché rychle rozšiřitelné prostředí rapidně usnadňující vývoj s obrovskou komunitou sdílenou s klientským „světem“ webové aplikace. [20]

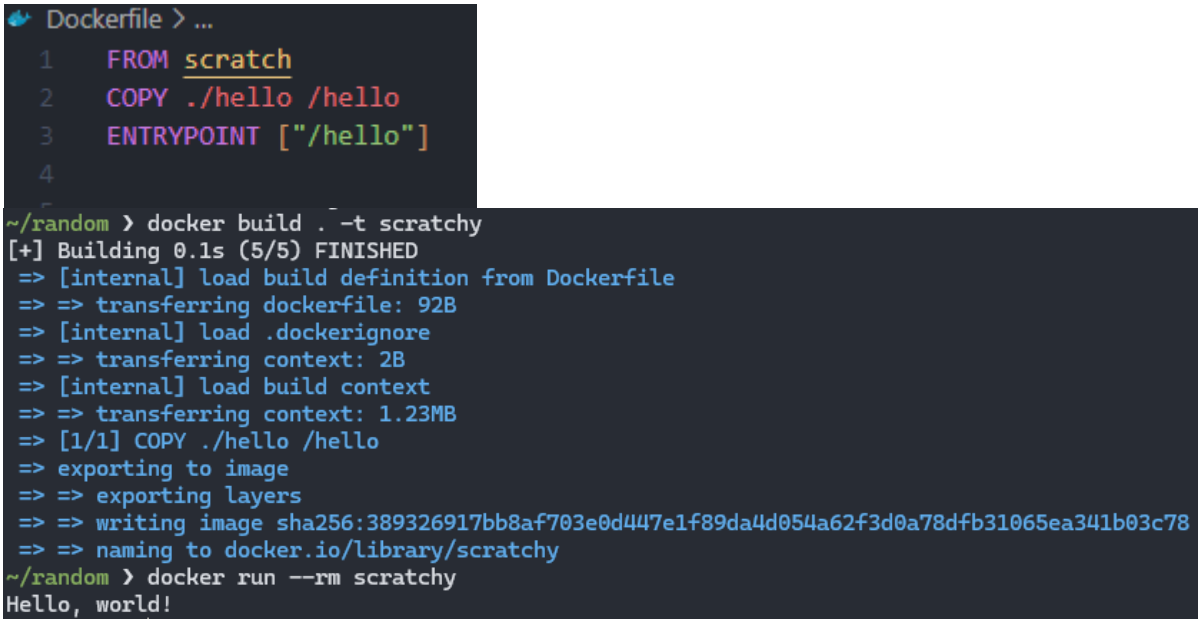
2.3 Kontejnerizace aplikací

V poslední teoreticky zaměřené podkapitole se zaměříme na téma kontejnerizaci aplikací. Kontejner je minimalistické, izolované, spustitelné, virtualizované prostředí připomínající lehký virtuální stroj. Odlišují se však právě svojí lehkostí. Obsahují jen nezbytné komponenty a nástroje pro běh specifické aplikace nebo systému. Jsou méně náročné na zdroje běžící na hostitelském stroji, se kterým sdílí jádro operačního systému, což dále snižuje režii a zvyšuje efektivitu využití systémových zdrojů počítače. Při stejné konfiguraci a shodné verzi kontejnerové platformy se kontejnery chovají na všech zařízeních totožně zajišťující konzistenci, předvídatelnost chování aplikace a odstranění konfiguračních prerekvizit dané platformy. V naprosté většině případech jsou kontejnerizované aplikace mnohem lépe spravovatelné a udržovatelné. Tato snadná správa usnadňuje aktualizaci jednotlivých nástrojů a prostředí uvnitř kontejneru nebo omezení dostupných zdrojů pro kontejner. Umožňuje drasticky snížit downtime aplikací a případně velice rychle nasadit nebo škálovat. [21]

I přes svou relativní izolovanost od systému existují způsoby, kterými útočník po získání root oprávnění uvnitř kontejneru, může získat root oprávnění i nad celým hostitelským systémem. Převážně je tento stav způsoben tím, že uvnitř kontejneru jsou aplikace spuštěné pod root uživatelem kontejneru. Existují však možnosti, které limitují možnosti útočníka jako je např. užití jiného uživatele uvnitř kontejneru pro spuštění aplikace, než je root. Je stále však dobré mít na paměti, že útočník může stále ovlivnit jiné aplikace, které s kontejnerem komunikují. [21]

Nejpopulárnější platformou pro správu a běh kontejnerů je platforma **Docker**. Docker je vyvíjen společností Docker, Inc. Docker mimo správu kontejneru poskytuje šablony pod názvem Docker image. Jde o binární soubory, které zahrnují specifickou aplikaci, její

závislosti, systémové knihovny a konfigurační soubory. Tento binární soubor vždy až na jednu výjimku musí vycházet ze specifikace jiného Docker image. Výjimkou je základní specifikace image s názvem „scratch“ ze kterého vychází naprosto všechny Docker image. Po sestavení image vycházejícího ze „scratch“ kontejner neobsahuje v souborovém systému vůbec nic. Tento stav umožňuje podrobně optimalizovat kontejner pro konkrétní potřeby patřičné aplikace (viz obr. 4). Ke specifikaci celého image se používá formát Dockerfile, pomocí které se sestaví následně Docker image na hostitelském počítači. Po sestavení kontejner obsahuje většinou hierarchii a strukturu souborového systému podobné plnohodnotným Linuxovým prostředí (až na výjimky jako jsou některé image sestavené ze „scratch“ specifikace). [21]



Obr. 4 Sestavení scratch image s „Hello, world!“ Golang programem a spustění kontejneru. Zdroj: vlastní tvorba

3 Analýza

Kapitola o analýze se soustředí na dokumentaci a vyhodnocení funkčních a nefunkčních požadavků, porovnání stávajících možných řešení, stanovení metodiky vývoje softwaru a popis uživatelských scénářů. Tato kapitola je stěžejní pro vývoj a přehled celé aplikace. Ze stanovených požadavků lze následně odvodit použitelné metody a technologie pro splnění požadavků. Při správné analýze a vytyčení požadavků na software dosáhneme možnosti efektivního vývoje, jehož výstupem by měla být funkční webová aplikace dosahující stanovených cílů.

3.1 Funkční požadavky

3.1.1 Uživatelské role a jejich potřeby

Studenti

Student je základní role uvnitř systému. Chování role by měla připomínat chování běžného uživatele. Uživatel si potřebuje po přihlášení zobrazit předměty, k nímž je přiřazen a ve kterých má uživatel zadané úkoly. Ze seznamu úkolů je nutné, aby student mohl vyčíst název úkolu, podrobnosti o času odevzdávání a stav úkolu. Po přechodu na patřičný si může student přečíst zadání úkolů, stáhnout si přílohy, pokud u úkolu existují. Student v této sekci musí mít možnost nahrát vypracované zadání a zobrazit si předešlé pokusy.

Vyučující

Role vyučujícího musí mít charakter role administrátora. Na rozdíl od studentské role, musí vyučující mít možnost vidět všechny předměty, ke kterým je přiřazen jeho IS Stag účet. Vyučující by neměl mít možnost vidět naprosto všechny existující předměty. V části předmětu vyučující má možnost editovat informace o předmětu a zadávat, mazat a editovat úkoly.

3.1.2 Správa předmětu

V systému by měl být seznam předmětů, který je modifikován v rámci přiřazené role uživatele. Obsah jednotlivých předmětů by měl být schopen měnit pouze vyučující nebo administrátor. Obsahem by měly být především informace o průběhu předmětu a zadávání úkolů.

3.1.3 Zadávání a správa úkolů

Se správou předmětu souvisí správa úkolů, která je součástí každého předmětu. V sekci úkolů by měl vyučující mít možnosti zakládat, mazat a editovat jednotlivá zadání. V rámci

podrobnější specifikace úkolu by měl být vyučující schopen dodat některé parametry, které jsou klíčové pro automatickou kontrolu programátorských prací. Pro lepší správu úkolů si vyučující navíc potřebuje zobrazit studenty přiřazené k úkolu, stav jejich odevzdání, výsledek, datum a jednotlivé pokusy s dostatečnými informacemi a zdrojovými soubory. Jednotlivá přiřazení studentů k úkolům musí vyučující mít také pod kontrolou.

3.1.4 Odevzdávání úkolů

Tento proces se týká pouze uživatelů s rolí student. V rámci sekce předmětu si student může přečíst informace o předmětu a zobrazit si momentálně přiřazené úkoly. Při zobrazení úkolů má student k dispozici přečtení zadání úkolu. V této části musí mít možnost nahrát řešení úkolu a zobrazit si předešlé pokusy. Student naopak nesmí mít volbu editovat obsah jednotlivých předmětů a úkolů a nesmí být schopen si zobrazit úkoly, které nejsou uživateli přiřazeny. V rámci odevzdaných pokusů úkolů může student nahlédnout do některých informací pokusů, ale nesmí mít možnost je editovat.

3.1.5 Automatická kontrola a hodnocení

Po odevzdání úkolu bude systém automaticky spouštět a kontrolovat odevzdané řešení úkolu. Odevzdaný zdrojový kód projde validací a v bezpečném izolovaném prostředí se musí zkompilovat a spustit na základě zadaných parametrů a specifikací na úrovni zadání úkolu. Po automatickém spuštění se musí vyhodnotit výsledky spuštění programu a řádně tyto výsledky uložit a zobrazit uživateli.

3.1.6 Využití dat momentálního systému

Vzniklá služba musí být napojená na API stávajícího systému IS Stag a brát zde potřebná data o uživatelích. Komunikace musí být jednosměrná a nesmí nijak ovlivňovat data na straně IS Stag. Abychom zamezili příliš velké redundanci dat, musíme během návrhu zohlednit množství strukturu tohoto API rozhraní a stanovit data, která je součástí databázového modelu aplikace a data, jež jsou externě dostupné přes vzdálené API rozhraní IS Stag. Toto rozdělení nesmí však příliš zpomalit běh a plynulost aplikace.

3.2 Nefunkční požadavky

3.2.1 Výkon a dostupnost

Systém musí být schopný rychle reagovat na požadavky uživatelů různého rozsahu. Během automatické opravy programátorských prací musí být systém schopen ohodnotit

poměrně velké množství prací v rozumném čase. Systém musí být stále dostupný celému internetu a musí být stabilní s minimálními výpadky.

3.2.2 Bezpečnost

Platforma musí být navržená tak, aby splňovala bezpečnostní praktiky a nedocházelo k ohrožení citlivých dat uživatelů. Autentizace uživatelů musí být řádně chráněna proti útočníkům s cílem získat přístup k těmto uživatelským účtům. Všechny vyčleněné komunikační metody musí být odolné proti běžným bezpečnostním hrozbám. V rámci automatické opravy programátorských prací musí být před kompilací a spuštěním odevzdaného programu nastaveno prostředí, jenž je izolováno od hostitelského systému a při spuštění nebezpečného kódu nedojde k žádnému narušení nebo prolomení systému nebo zavést dostatečnou validaci a kontrolu odevzdaného zdrojového kódu.

3.2.3 Použitelnost

Uživatelské rozhraní musí být jednoduché a snadno použitelné pro všechny typy uživatelů. Rozhraní musí být intuitivní a v rámci malých obrazovek stačí, aby bylo použitelné, vzhledem k tomu, že vypracovávání a odevzdávání úkolů se primárně provádí na stolních počítačích.

3.2.4 Integrita

Systém musí vyhovět praktikám k zachování integrity dat. Při neočekávaných chybách musí být schopen se z chybového stavu zotavit a neohrozit tak ovlivněná data a v případě transakcí úspěšně provést patřičný rollback operací.

3.3 Analýza stávajícího řešení

Stávajícím řešením je momentálně v době psaní této práce platforma Moodle. Tato platforma vznikla v roce 2002 za účelem vytvořit službu pro správu online předmětů na internetu s kladením důrazu na interakci. Systém v základu disponuje automaticky vyhodnocovanými úlohami, které ale neumožňují spouštění a hodnocení zdrojového kódu. Je zde tedy velký limit v interpretaci úkolu pro programátory.

Existuje však stažitelný plugin s názvem CodeRunner udržován Richardem Lobbem a Timem Huntem. Plugin využívá Moodle kvizů u předmětu ke zprostředkování úkolu. K vytvoření úkolu vyučující využívá administraci otázek Moodle. Vyučující při vytváření úkolu stanoví základní nastavení pro otázku jako programovací jazyk, zadání, a hlavně testovací sekci s očekávanými výstupy. Tato sekce nejčastěji otestuje vložený kód od studenta

a zformátuje jeho výsledek. Výsledek porovná s očekávanými výsledky na základě podobnosti odpověď studenta ohodnotí. Veškerý cizí kód je spouštěný v sandbox prostředí Jobe ideálně nainstalovaný na separátním zařízení, aby se docílilo nejvyšší bezpečnosti. V základu podporuje programovací jazyky Python, Javu, C, C++, PHP a JavaScript, Octave a Matlab. Lze však plugin rozšířit o podporu dalších programovacích jazyků. Nejjednodušeji se tím docílí přes Python script, který zavolá patřičné programy pro kompilaci a spuštění zdrojového kódu. CodeRunner nabízí velký prostor pro definici vlastní logiky spouštění a hodnocení programátorských prací.

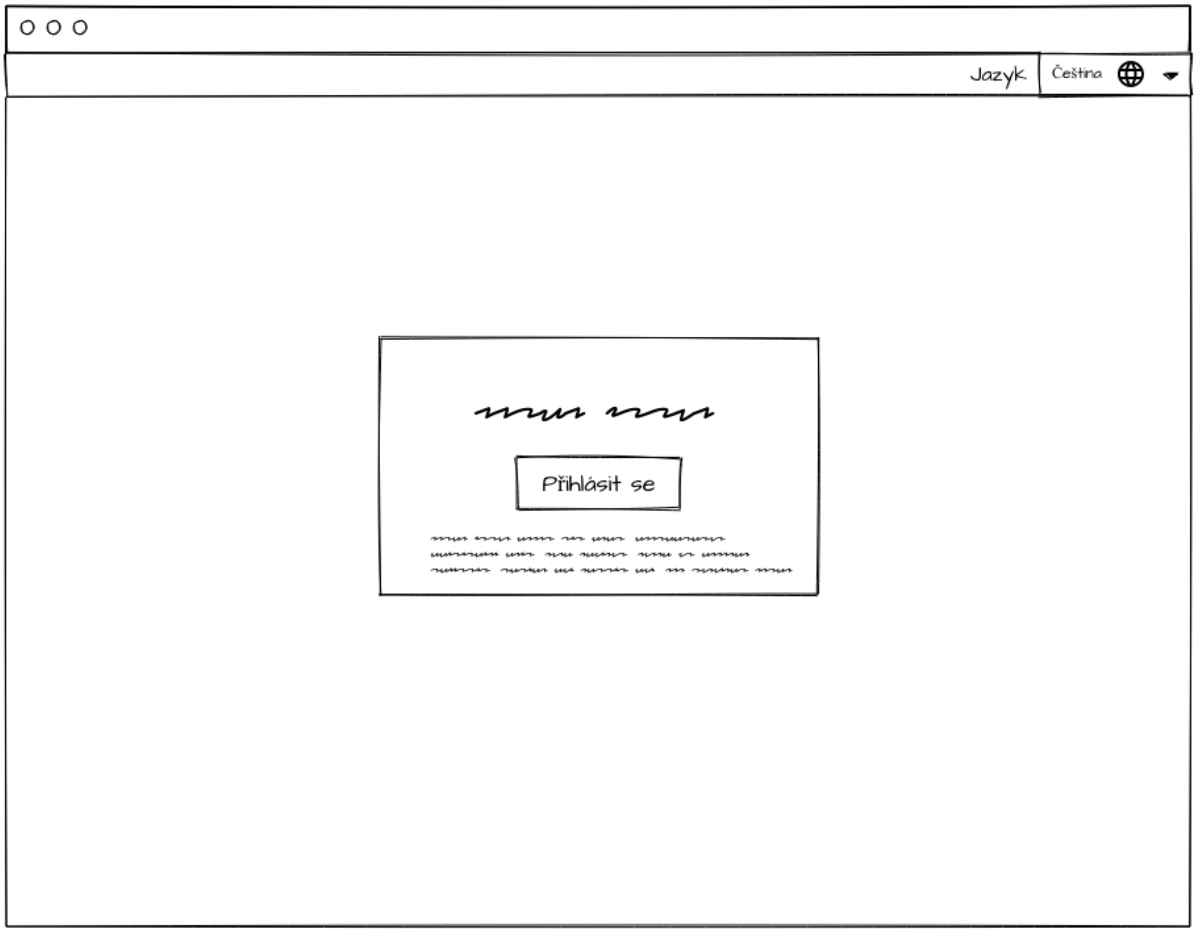
4 Návrh řešení

Návrh řešení pro systém automatické opravy programátorských prací zahrnuje plánování architektury systému, výběr technologií, návrh databáze, uživatelského rozhraní a implementace klíčových komponent. Návrh zajišťuje, že systém je efektivní, škálovatelný, bezpečný, intuitivní a snadno použitelný.

4.1 Návrh uživatelského rozhraní

Následující podkapitola rozebírá návrh uživatelského rozhraní a poskytuje k popisu jednoduché wireframes.

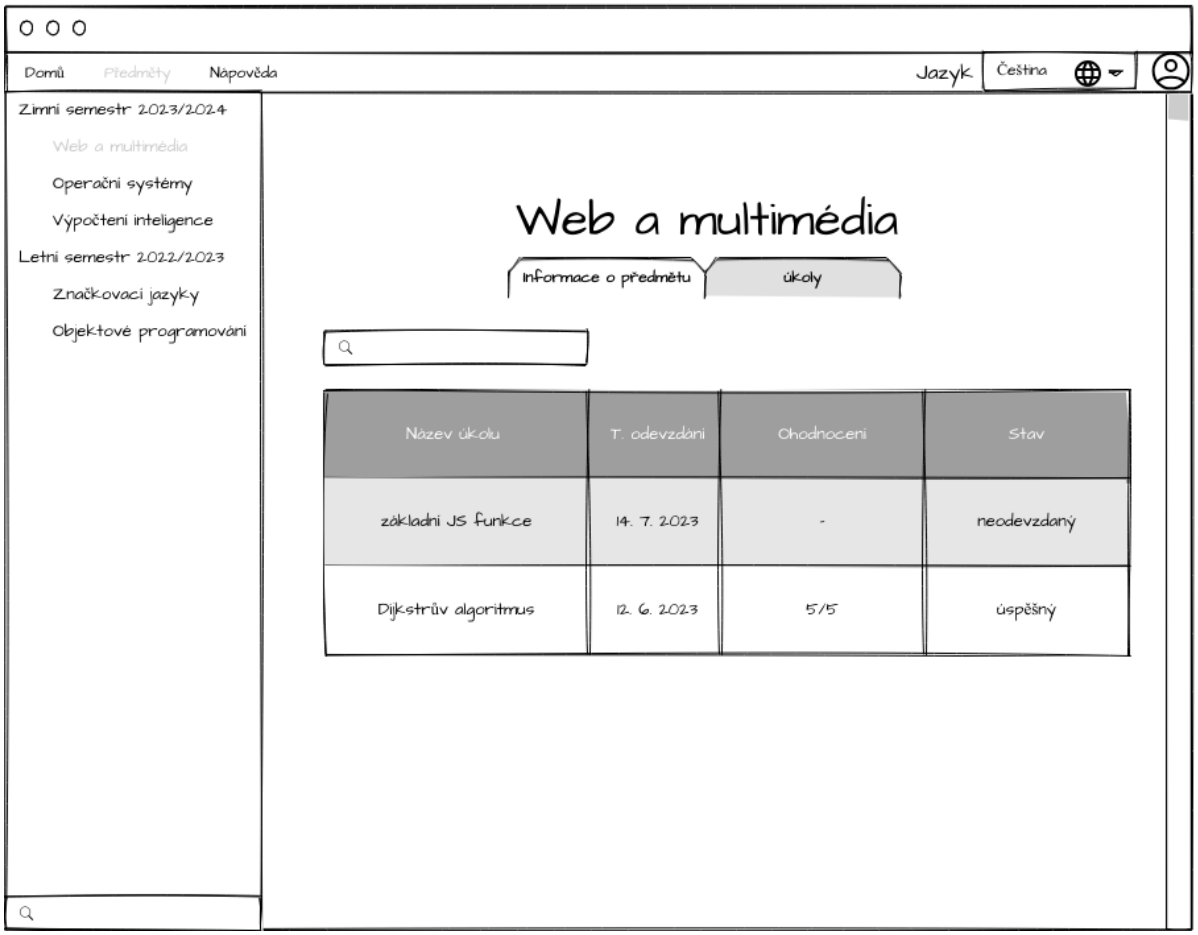
4.1.1 Přihlašovací obrazovka



Obr. 5 Wireframe přihlašovací obrazovky. Zdroj: vlastní tvorba

Přihlašovací obrazovka aplikace je poměrně jednoduchá. Jako je na wireframe v obr. 5 zaznamenáno, jde o malé okno s tlačítkem přihlásit se. Toto tlačítko uživatele přesměruje na přihlašovací obrazovku portálu Stag, kde následně vyplnění údaje a při úspěšném přihlašování dojde k přesměrování do hlavní části aplikace. V pravé horní části obrazovky je už na výběr jazyk aplikace.

4.1.2 Obrazovka předmětů a zobrazení předmětu

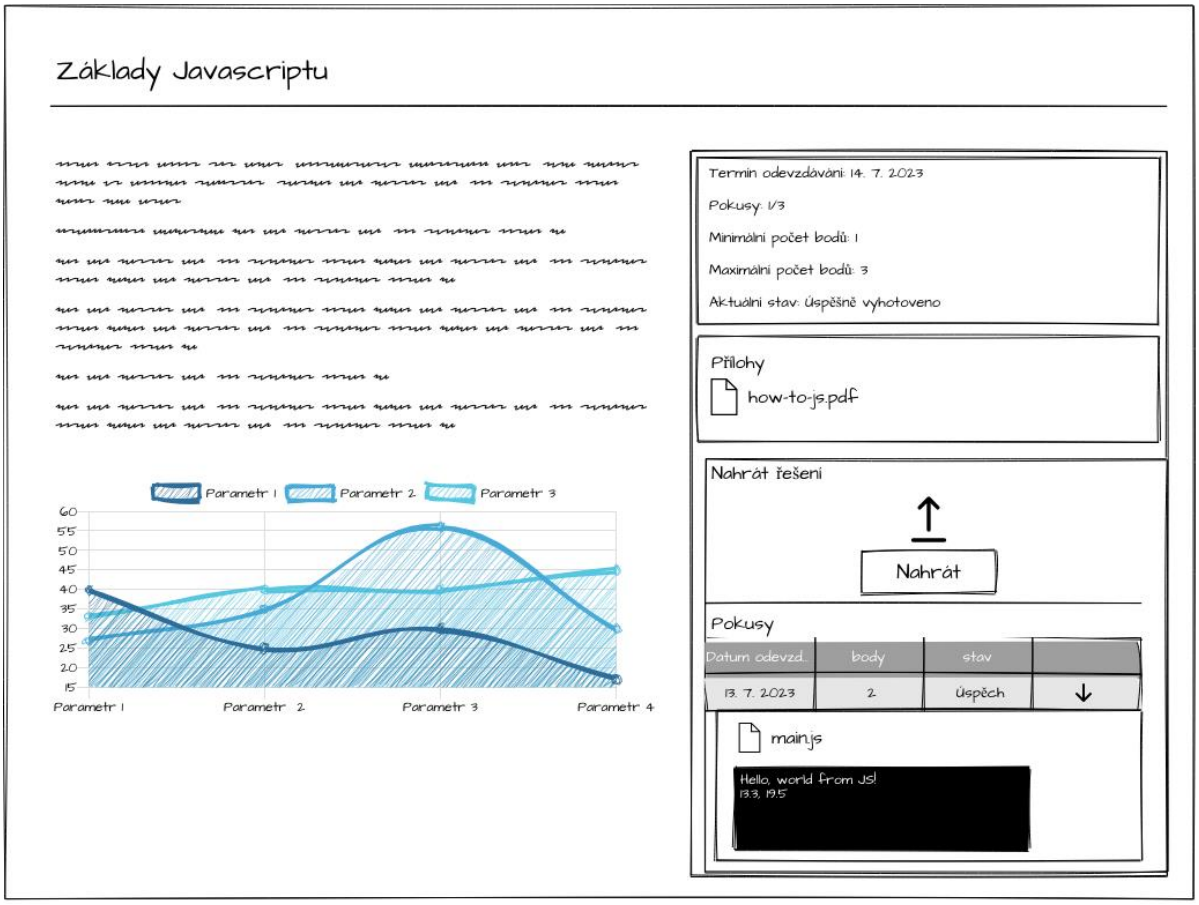


Obr. 6 Wireframe obrazovky předmětu a zobrazení předmětů. Zdroj: vlastní tvorba

Po přihlášení je uživatel přivítán domovskou stránkou a má na výběr z jednoduchého menu. Nejvíc důležitá položka, je položka „Předměty“. Po rozkliknutí dochází k zobrazení obrazovky předmětů (obr. 6). Tato obrazovka po prvotním načtení obsahuje pouze levou sekci se seznamem předmětů, které má student zapsané, rozdělený na semestry. V seznamu předmětů je vyhledávací pole pro textové filtrování předmětů.

Jakmile uživatel rozklikne některý z předmětů, objeví se obrazovka s informacemi. Je zde i malé menu umožňující přístup na sekce s informacemi a úkoly předmětu. V sekci úkolů se nachází tabulka zobrazující jednotlivé zadané úkoly v tomto předmětu (viz obr. 6). Patrný je zde název, termín odevzdání, hodnocení a stav úkolu. Úkol jde dál rozkliknout na detail úkolu s formulářem pro odevzdávání úkolu.

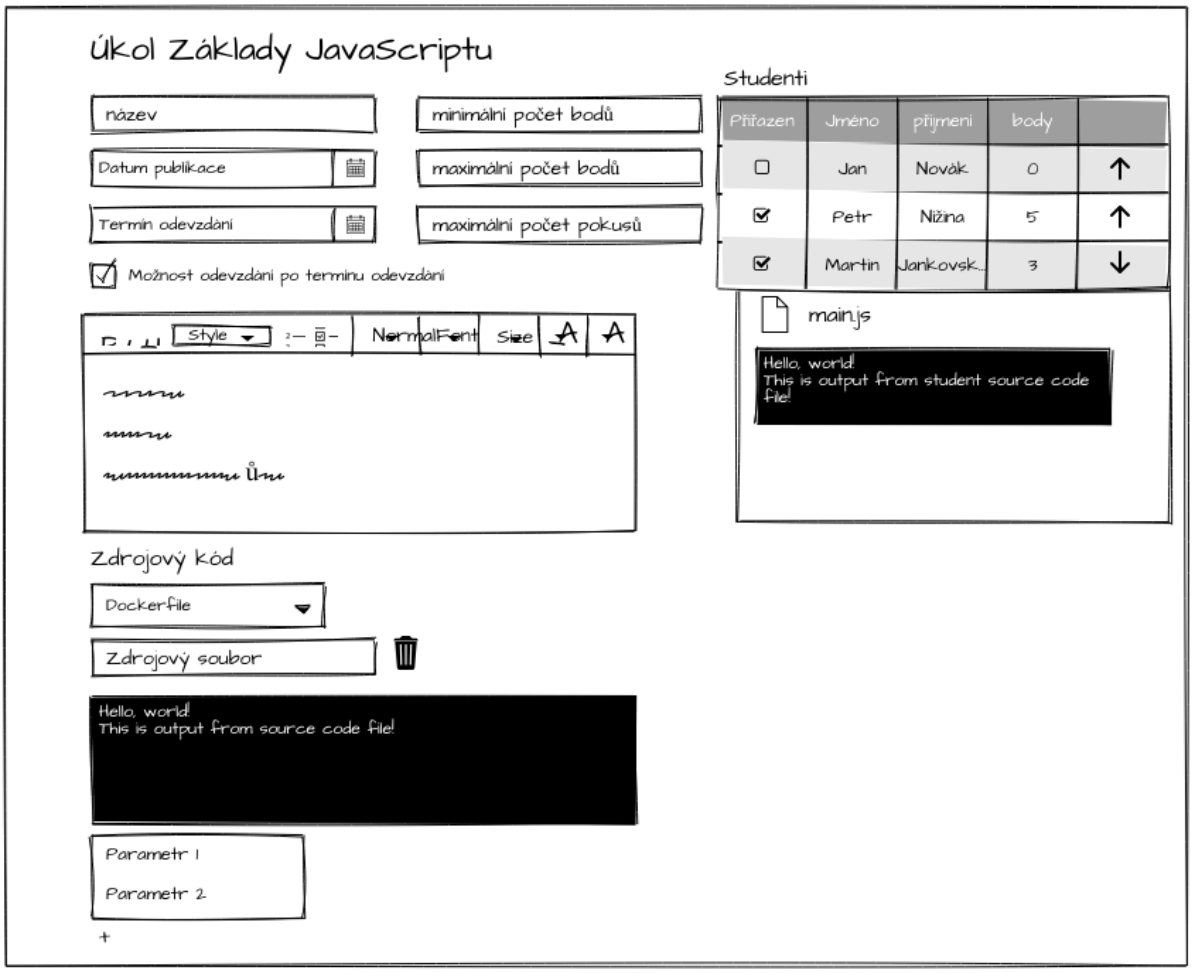
4.1.3 Detail a formulář odevzdávání úkolu



Obr. 7 Wireframe detailu a formuláře odevzdávání úkolu. Zdroj: vlastní tvorba

Po rozkliknutí úkolu se uživatel dostává na konkrétní detail úkolu v rámci obrazovky předmětu. V detailu se nachází název úkolu a pod ním textové zadání, které je možné obohatit obrázky. Vedle zadání má uživatel specifické informace o úkolu a termínech spolu s přílohami, které si student může stáhnout. Pod informacemi obsahuje sekce formulář pro odevzdání úkolu. Formulář s nahrávací ikonou a tlačítkem slouží k odeslání nahraného souboru. Pod touto částí se vytváří seznam všech pokusů, kde si uživatel může zobrazit důležité informace včetně výstupu vloženého zdrojového kódu.

4.1.4 Formulář zadávání úkolu



Obr. 8 Wireframe formuláře zadávání úkolu. Zdroj: vlastní tvorba

Poslední wireframe popisuje návrh formuláře zadávání úkolu (obr. 8). K tomuto formuláři má přístup pouze účet vyučujících oproti předešlým obrazovkám určený pro studenty. Jde o jednotlivá pole, která jsou určena k podrobné specifikaci úkolu. Dělí se na: základní sekci určenou pro specifikaci textových vlastností a termínů úkolu, sekci vkládání zdrojového kódu potřebný pro automatickou opravu odevzdaného úkolu a sekci se studenty, kde vyučující může manuálně přiřadit nebo odebrat studenta a zobrazit si pokusy jednotlivých studentů se vstupním souborem a výstupem zdrojového kódu.

4.2 Architektura systému

Architektura systému je založena na vícevrstvé architektuře s oddělením prezentační logiky, aplikační logiky a datové vrstvy. Tento přístup zajišťuje modularitu řešení, škálovatelnost a snadnou údržbu a hojně se využívá pro vývoj webových aplikací. Klientská a serverová logika je od sebe oddělena dle architektury klient-server.

4.2.1 Prezentační vrstva

Prezentační vrstvu aplikace tvoří moderní frontend JavaScript framework. Vrstva se spouští v prohlížeči na počítači klienta a pomocí REST API komunikuje se serverovou částí.

Cesta JavaScript frameworku byla zvolena, protože na klientské části potřebujeme mít hodně logiky související se zadávacími a odevzdávacími formuláři. Framework velice usnadňuje vývoj a nastoluje prostředí pro možnou budoucí expanzi projektu. Pro tyto účely využíváme konkrétně SPA framework, protože nemusíme brát ohled na web crawlers. Stránka je primárně pro autorizované uživatele. Při rozhodování bylo nastoleno, že framework musí být rychlý, mít rozvinutou velkou komunitu pro budoucí podporu, rozsáhlou a srozumitelnou dokumentaci. Musí podporovat vývoj pomocí komponent pro snadné znovupoužití sekcí a nesmí být příliš náročný na údržbu. REST API je osvědčený standard využívaný velkým počtem webových aplikací. Svojí jednoduchostí a robustností se stává vhodným kandidátem pro komunikaci se serverem.

Dále pro ošetření chyb týkající se špatného typování v dynamickém jazyce JavaScript se používá nadstavba **TypeScript**, která obohacuje vývoj o typovou kontrolu a zamezuje tak některým chybám už při vývoji.

Pro tyto všechny tyto účely byla vybrána knihovna **React.js**. I když se nejedná o framework, je svojí flexibilitou, „arsenálem“ možností a s jednou z největších frontend komunit brán jako vhodný kandidát pro středně velké aplikace.

V rámci stylování a implementace určitého stupně responzivity je vhodné k JavaScript frameworku přidat také CSS framework, jenž nabízí předem připravenou množinu stylů usnadňující stylování komponent a následnou optimalizaci pro různá zařízení. V některých případech obsahuje už i hotové komponenty připravené k použití nebo personalizaci a vyhovění potřeb webové stránky. Knihovna React.js jsme skombinovali s CSS nadstavbou **Bootstrap**, který je mezi CSS frameworks velmi uznávaný. Obsahuje sadu předpřipravených stylů a komponent a pro účely tohoto projektu velmi pomáhá při stylování a urychluje vývoj webové aplikace.

Mimo jiné pro optimalizaci a sestavení výsledného JavaScript kódu z projektu se používá balíčkovací nástroj **Vite**.

4.2.2 Aplikační vrstva

U aplikační vrstvy si rozebereme backend funkcionalitu projektu běžící na serveru a určíme technologie pro vývoj a údržbu backend aplikace. Backend aplikace je svou velikostí

a náročností poměrně náročná pro běžnou implementaci. Proto volba jít opět cestou frameworku velice rapidně urychluje vývoj a uspořádá jednotlivé komponenty aplikace do řádné projektové struktury. Není však třeba přemýšlet o platformě schopné udržovat a vyvíjet aplikace na enterprise úrovni. Zároveň se nebavíme o použití pouze jednoho frameworku. Konkrétně je zapotřebí integrovat minimálně webový a ORM/ODM framework.

Webový framework nám pomáhá rychle vyvinout REST API aplikace s rychlým zpracováním a validací API požadavků a jejich odbavením adekvátní odpovědí. Mimo to nám webový framework vede ke správnému zabezpečení jednotlivých sekcí aplikace, aby v budoucnu nebyly náchylné na nejběžnější bezpečnostní hrozby.

ORM/ODM framework nám hodně zlehčuje práci s databází. V případě použití relační databáze nám umožňuje mapovat tabulky na objekty urychlující vývoj a zlepšení kontroly nad daty. Framework většinou také disponuje migrační funkcionalitou, která při vývoji aplikace, zvláště budoucím, pomáhá s migrací dat do nebo z databázového schématu.

Oba frameworky, a také programovací jazyk využívající framework musí být opět rychlé, dobře udržované, s bohatou komunitou a musí splňovat dostatečnou flexibilitu při vývoji.

V souvislosti se zmíněnými požadavky byl jako webový framework vybrán **Express.js** a tím i platforma **Node.js** postavená na programovacím jazyku JavaScript. Tento framework vyniká svojí flexibilitou a díky zvolení frontend JavaScript frameworku v předešlé podkapitole, využíváme sdílení patřičných částí codebase mezi aplikacemi. JavaScript platforma má také díky používání stejného balíčkovacího systému Npm přístup k nespočet knihovnám dále usnadňující vývoj aplikace jako je například nadstavba **Typescript**, kterou využíváme i na serverové straně.

V případě ORM frameworku je využit framework **Prisma**. Tento framework pracuje na základě schématu definovaném pomocí souboru „schema.prisma“, který popisuje databázové modely a jejich vztahy. Po vytvoření schématu generuje typovaného klienta v Typescript jazyce přesně na míru schématu, který se chová jako objektové rozhraní pro komunikaci s databází. Framework je použitelný pro většinu relačních databází, má rozsáhlou dokumentaci a rozvinutou komunitu a řadí se mezi nejmodernější JavaScript ORM frameworky.

V rámci autentizace je implementována autentizace skrze autentizační formulář JČU, která následně poskytuje přístupový ticket systému pro opravu programátorských prací. Samotná aplikace využívá tento ticket pro přístup k potřebným informacím. Navíc na straně

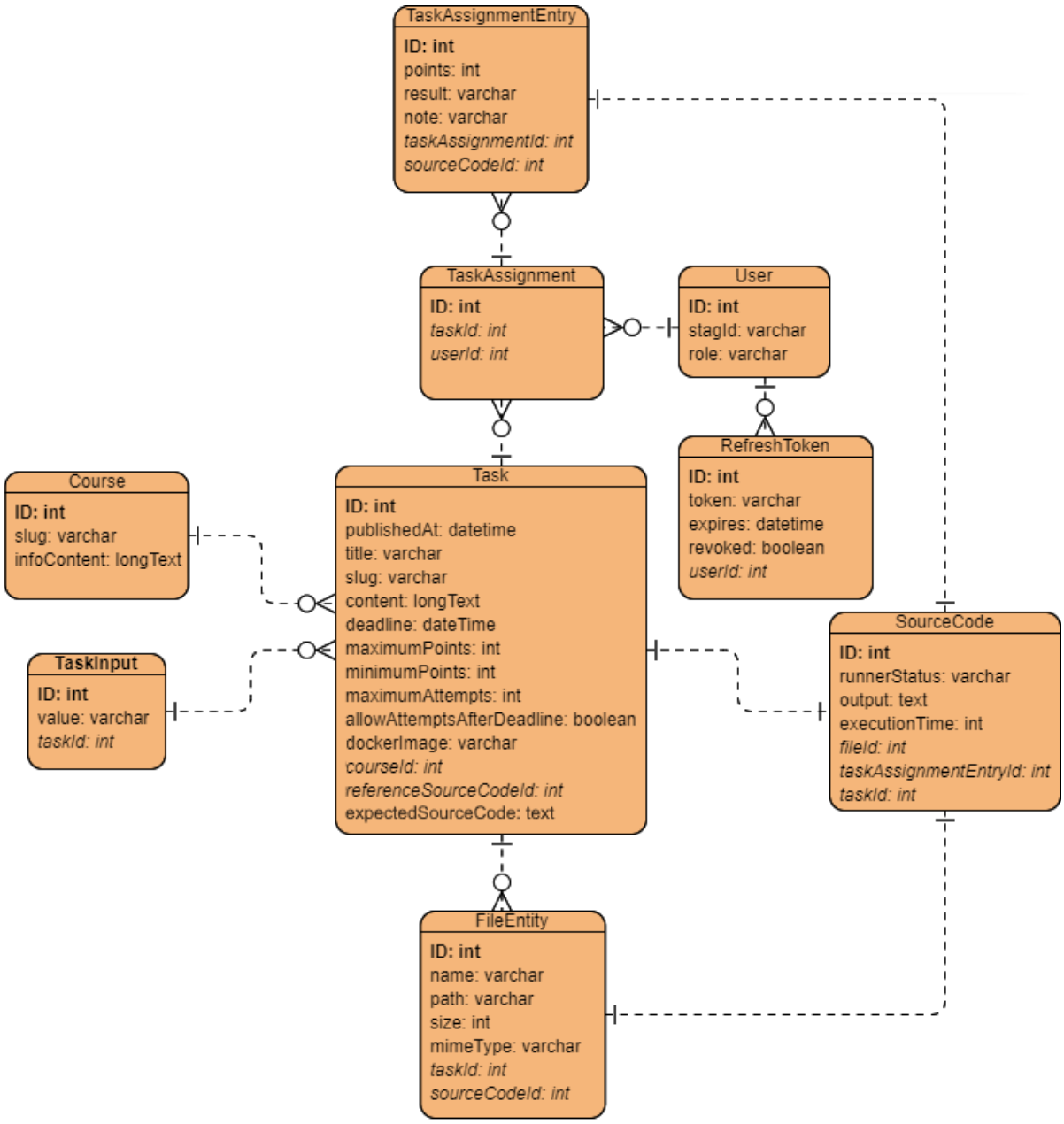
aplikace existuje vlastní autentizace implementována ve formátu **JWT access** a **refresh** tokenů. Tato vlastní autentizace nabízí větší kontrolu nad přihlášenými uživateli. Např. v případě donucení uživatele k odhlášení se nastaví jeho refresh token jako vypršelý a aplikace tohoto uživatele automaticky odhlásí při validaci tohoto tokenu.

Na závěr bylo potřeba si specifikovat automatický proces spouštění a hodnocení odevzdaných programátorských prací. Tento proces musí probíhat ve virtualizovaném prostředí. Virtualizované prostředí nám dovoluje oddělit běh programů od hostitelského stroje a minimalizovat tak potencionální vnitřní hrozby. Velice přívětivou volbou je kontejnerizace těchto programů. Kontejnery jsou velmi lehké a jednoduché výpočetní instance, které lze snadno nakonfigurovat. Pro tyto účely byla vybrána platforma **Docker**, která disponuje specifikaci kontejnerů přes **Dockerfiles** a nabízí CLI pro ovládání kontejnerů, které využíváme přímo v naší aplikaci a necháváme tak veškerou správu těchto kontejnerů na aplikaci. Abychom správně hodnotili vložený kód i v případě např. náhodně generovaných vstupů, musíme místo použití pouze textové specifikace očekávaného výstupu nadefinovat také referenční zdrojový kód. Tento kód vkládá vyučující při specifikaci úkolu. Je spouštěn zároveň se zdrojovým kódem nahrávaný studentem a následně porovnán s výsledky obou programů, kde referenční kód produkuje očekávaný výstup. Pokud používáme náhodně generované vstupy, poskytneme je oběma implementacím a tím docílíme správného hodnocení v tomto případě.

4.2.3 Databáze

Databáze hraje klíčovou roli pro zaznamenávání záznamů produkované studenty a vyučujícími. Musí být tedy také zvolena. Protože data jsou svojí strukturou velmi jednoduchá a struktura je pevná s danými datovými typy, zvolili jsme pro chod aplikace databázi **relační** oproti dokumentové. Souvisí to i s faktem, že projekt nevyžaduje nejvyšší stupeň dostupnosti na úkor konzistence a pokročilé horizontální škálování jako je např. sharding, je často dostupné u dokumentových databází. Pro naše účely byl zvolen systém řízení báze dat **MariaDB**.

4.3 Databázový model



Obr. 9 ER diagram databáze systému. Zdroj: vlastní tvorba

Po stanovení funkčních a nefunkčních požadavků jsme schopni navrhnout databázový model. Budeme se věnovat definicí entit a vztahů. Během návrhu databázového modelu zohledníme stanovenou API strukturu rozhraní IS Stag a vyhneme se redundanci dat. Úkolem bylo tedy navrhnout stabilní, funkční a škálovatelný databázový model, který obsahuje dostatečné informace pro správný a plynulý běh aplikace.

Postupně si popíšeme navržené schéma (viz obr. 9). Jako základní entita, jež je stěžejní pro chod aplikace, je jednoznačně **uživatel** (viz obr. 9 entita User). Uživatel musí obsahovat roli, abychom každý uživatel měl správná oprávnění. U této entity předpokládáme, že

uživatelé už v rámci systému JČU IS Stag **existují**. Tudíž přidávání uživatelů do výsledné tabulky je provedeno v rámci prvního přihlášení uživatele do systému automatických oprav po úspěšné autentizaci na straně systému IS Stag a po úspěšném poskytnutí informací o tomto uživateli. V rámci prvního přihlášení se ukládají informace, se kterými zpětně uživatele párujeme na uživatele v IS Stag systému. Pro tyto účely nám stačí identifikační číslo IS STAG. Pro podporu JWT refresh tokenů, jsme k uživateli napojili entitu **refresh tokenů** (viz obr. 9 entita RefreshToken).

Další velmi důležitou entitou je entita **zadání** (na obr. 9 jako entita Task). Tato entita obsahuje veškeré informace o zadání vytvořeném vyučujícím. Jde o viditelné informace pro studenty jako název, zadání, datum nejpozdějšího odevzdání a hodnotící kritéria. Dále zde nalezneme technické specifikace úkolu jako používaný Docker image.

Entita zadání jsme napojili na entitu **předmětů** (obr. 9 entita Course). Entita předmětů toho neobsahuje příliš. Většinu informací nalezneme v API IS Stag. Stačí zde tedy tzv. „slug“, který má úlohu jako jedinečný identifikátor předmětu. Skládá se z parametrů, které nám následně dovolují napárovat předmět na systém IS Stag. Posledním sloupcem je sloupec pro uchování textu popisující daný předmět.

Další malou entitou jsou **vstupní argumenty** zadání. Jde o jednoduchou entitu, kde ukládáme hodnotu vstupu jako řetězec umožňující vytvářet komplexnější zadání.

Když se vrátíme k entitě uživatel a chceme určitě tuto entitu propojit s entitou úkolů. Dostáváme tím dvě entity: Entitu **zadání** (obr. 9 v diagramu jako TaskAssignment) a entitu **pokusů** (v obr. 9 jako TaskAssignmentEntry). Entita zadání používáme k propojení uživatele (studenta) s úkolem. Každý uživatel pod určitým předmětem nemusí v reálném světě plnit všechny úkoly a takto stanovené schéma umožňuje zadat úkol pouze některým uživatelům. Navíc k této tabulce potřebujeme znát informace o předešlých pokusech a ty shromažďujeme v rámci entity pokusů. Pokusy obsahují stav vyhodnocení a bodové ocenění odevzdaného programu.

Entita pokusů dále vlastní referenci na konkrétní vložený zdrojový kód. K tomu slouží entita **zdrojového kódu** (viz obr. 9 SourceCode entita). Každý pokus je napojen na instanci entity pro zdrojový kód. Zde jsou především informace o stavu kompilace a spuštění programu, čas vyhodnocení a výstup programu. Tato entita je také napojená na úkol. V rámci zadávání úkolů potřebujeme napojit i referenční zdrojový kód.

Všechny zdrojové kódy jsou napojeny na **souborovou entitu** (viz obr. 9 entita FileEntity). Tato entita zprostředkovává informace o nahraném souboru jako cesta k souboru

na serveru, velikost, název a typ souboru. Mimo jiné je tato entita napojená na entitu úkolu z důvodu vkládání příloh a obrázků ke specifikaci konkrétního úkolu.

5 Implementace

Pátá kapitola této práce zahrnuje popis vývoje a implementace systému dle stanovených požadavku a návrhu. V souladu s vývojem bude představeno použité vývojové prostředí s použitými nástroji usnadňující vývoj. Bude popsána struktura projektu a integrace na testovací server.

5.1 Vývojové prostředí a nástroje

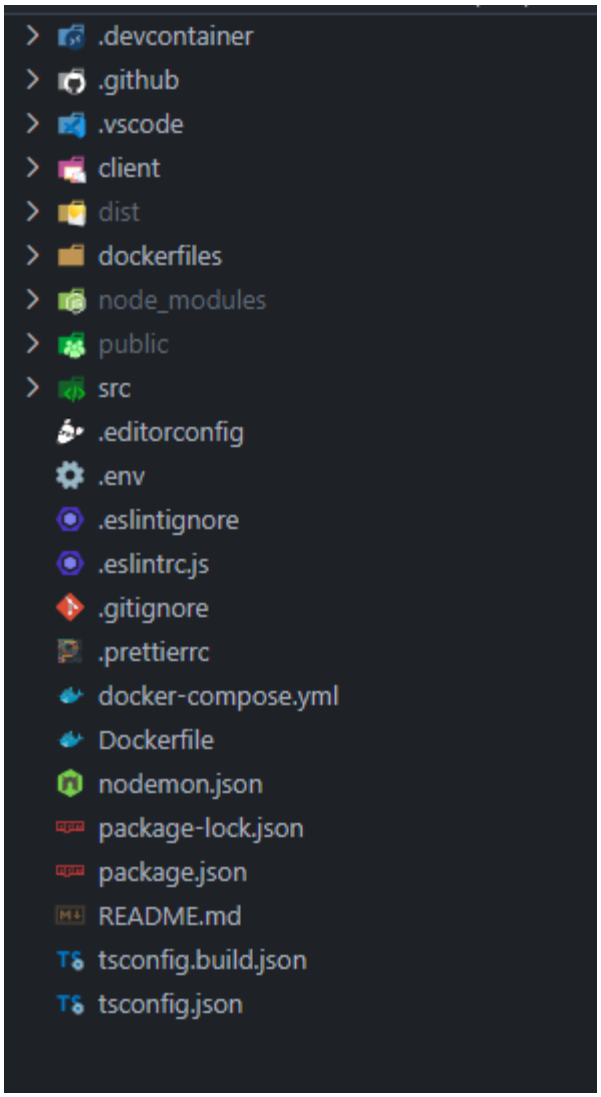
Pro vývoj bylo využito vývojové prostředí VS Code. Toto prostředí je velmi populární pro vývoj JavaScript aplikací a má širokou podporu pro různé programovací jazyky a frameworků nejen v jazyce JavaScript skrze pluginy. Pro spuštění a vyvíjení aplikace byla zvolena platforma Docker pro správu kontejnerů. Kontejnery usnadňují vývoj aplikace v prostředí, které si můžeme přesně nadefinovat a pomohly nám aplikaci integrovat. K rozšíření a usnadnění konfigurace Docker kontejnerů jsme použili nadstavbu Docker compose.

Vývojové prostředí VS Code nabízí velmi pokročilou interakci s Docker kontejnery a nabízí plugin balíčky, které zavádí možnost vyvíjet přímo uvnitř kontejneru. Tento způsob vývoje přesouvá veškeré systémové a programové požadavky na kontejner odstraňující problémy s kompatibilitou a nutností instalace různých verzí stejného softwarového vybavení pro vývoj. Mimo kontejnery budou v rámci VS Code prostředí použity pluginy specifické pro projekt využívané frameworky.

5.2 Struktura projektu

Projekt je rozdělen na dvě části (dvě aplikace): serverovou, jenž se nachází v kořenu adresáře a klientskou, kterou jsme umístili do adresáře „client“ (znázorněno na obr. 10). Ke každému tomuto adresáři jsou přiřazeny patřičné Docker kontejnery, které se po v rámci načtení konfigurace určené pro vývoj spouští a jsou startovány aplikace uvnitř kontejnerů. Kontejnery (přesněji aplikace v kontejnerech) posléze spolu komunikují v rámci jedné Docker sítě přes REST API rozhraní na jednom hostitelském stroji. Vyjma zdrojového kódu struktura

projektu disponuje konfiguračními soubory, určené pro nastavení Docker kontejnerů a vývojového prostředí včetně kontroly zdrojového kódu.



Obr. 10 Struktura projektu v IDE VS Code. Zdroj: vlastní tvorba

5.3 Backend aplikace

V následujících podkapitolách si popíšeme backend aplikaci dělící se na controller část, servisovou část a modelovou/datovou část. V podkapitolách nebude popsán veškerý kód, ale pouze klíčové části aplikace důležité pro tento projekt.

5.3.1 Routers

Jednotlivé kontroléry jsou reprezentovány v podobě routerů. Tyto routery slouží ke specifikací koncových bodů (endpoints), zachytávající a odbavující REST API požadavky. Jednotlivé routery jsou definovány dle přiřazené sekci aplikace a každý je identifikován jiným koncovým bodem. Některé routery mohou být závislé na jiných routerech (jako např. v případě task routeru). Routery využívají systém middlewares. Jde o funkce, které se si

navzájem předávají odchycený požadavek a na základě vlastní implementace mohou požadavek modifikovat nebo celý proces předávání požadavku přerušit zpracováním požadavku přímo v middlewaru. V tomto případě se požadavek odešle zpátky směrem ke klientovi nejčastěji s chybovou hláškou nebo validační chybou. Tento systém využívá aplikace např. při validaci POST požadavku ve formě „validateBody" middleware funkce (viz obr. 11). Jde o funkci, která vrací implementaci další funkce, v tomto případě náš konkrétní middleware. Funkce si přetransformuje pomocí knihovny „class-transformer" tělo požadavku a napáruje na patřičnou validační třídu obsahující definice validačních testů nad atributy pomocí TypeScript decorator deklarace. Tato instance nově napárované třídy je následně zvalidována na základě zmíněných validačních testů. Pokud validace obsahuje validační chyby, jsou chyby zpracovány do čitelné podoby a vráceny zpátky klientovi. Pokud ne, volá se funkce „next" předávající požadavek další middleware funkci.

```
export function validateBody<T>(dtoClass: ClassConstructor<T>) {
    return async (req: Request, res: Response, next: NextFunction) => {
        const dto = transformToDto(dtoClass, req.body);
        const errors = await validate(<object>dto);
        if (errors.length > 0) {
            const errorResponse = mapValidationErrorsToErrorResponse(errors);
            return res.status(400).json(errorResponse);
        }

        next();
    }
}
```

Obr. 11 validateBody middleware. Zdroj: vlastní tvorba

5.3.2 Proces registrace a přihlášení

Registrace uživatelů probíhá v rámci přihlášení provedené skrze IS Stag. Po tomto přihlášení dojde k odeslání přihlašovacího požadavku s IS Stag tokenem na webový server. Webový server odchytí tento požadavek v uživatelském routeru. Pro tento požadavek jsou na základě poskytnutého Stag tokenu vyžádána data o uživateli od IS Stag, kterému patří přístupový token. Pokud IS Stag odpoví kladně, webový server zkontroluje, jestli uživatel existuje. Pokud neexistuje, vytváří nového uživatele s poskytnutými údaji. Po získání uživatele se vytváří patřičné JWT tokeny (podrobněji popsané v následující podkapitole) a posílají se klientské aplikaci v rámci odpovědi na požadavek.

5.3.3 Autentizace

Autentizace požadavků je v aplikaci řešená skrze knihovnu Passport s nadefinovanou vlastní JWT strategií. Naše vlastní JWT strategie se opírá o uložení Stag identifikátoru do JWT

tokenu, spolu s rolí uživatele a Stag ověřovacího tokenu. K vytváření tokenů se používá na serveru uložené šifrovací heslo, které se používá k šifrování a dešifrování tokenu. Tento přístup zajišťuje, že se např. Stag ověřovací token nedá z JWT tokenu získat bez šifrovacího hesla. Passport knihovna má na starosti extrahování JWT tokenu z požadavku a následného dešifrování. V rámci JWT tokenů považujeme token za validní, pokud elektronický podpis se

```
public initializePassport(app: express.Express) {
    passport.use(
        new JwtStrategy({
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
            secretOrKey: process.env.JWT_SECRET
        }, (jwtPayload: JwtPayload, done) => {
            prisma.user.findUnique({
                where: {
                    stagId: jwtPayload.sub
                }
            }).then(async (user) => {
                if (!jwtPayload.stagTicket) return done(null, false);
                if (!user) {
                    user = await this.userService.createOrGetUser(jwtPayload.stagTicket);
                }

                return done(null, { ...user, stagTicket: jwtPayload.stagTicket });
            }).catch(err => {
                return done(err, false);
            });
        })
    );

    app.use(passport.initialize());
}
```

Obr. 12 část auth.service souboru s definicí Passport autentizační strategie. Zdroj: vlastní tvorba

shoduje s podpisem, který se generuje z datové části (jde o stejný proces jako u vytváření podpisové části JWT tokenu). V rámci knihovny Passport definujeme vlastní strategii pro napárování uživatele na specifický token. Z dešifrovaného tokenu se extrahuje identifikátor uživatele, podle kterého se uživatel najde v databázi. Tento nalezený uživatel se následně využívá v „check-auth“ middleware funkci, sloužící k autorizaci a k následnému přidání uživatele na API požadavek pro snazší práci v následujících middleware funkcí pracující s tímto požadavkem. Veškerá tato funkcionalita je součástí auth servisní třídy projektu (až na „check-auth“ middleware, který je definovaný mezi ostatními middlewares).

5.3.4 Automatická kontrola a hodnocení kódu

V rozboru návrhu řešení jsme si ujasnili, že potřebujeme, aby automatická kontrola kódu probíhala v Docker kontejnerech z důvodu bezpečnosti a jednoduchosti konfigurace. Aplikace sama o sobě už však v kontejneru pracuje (minimálně ve vývojovém prostředí). Na první

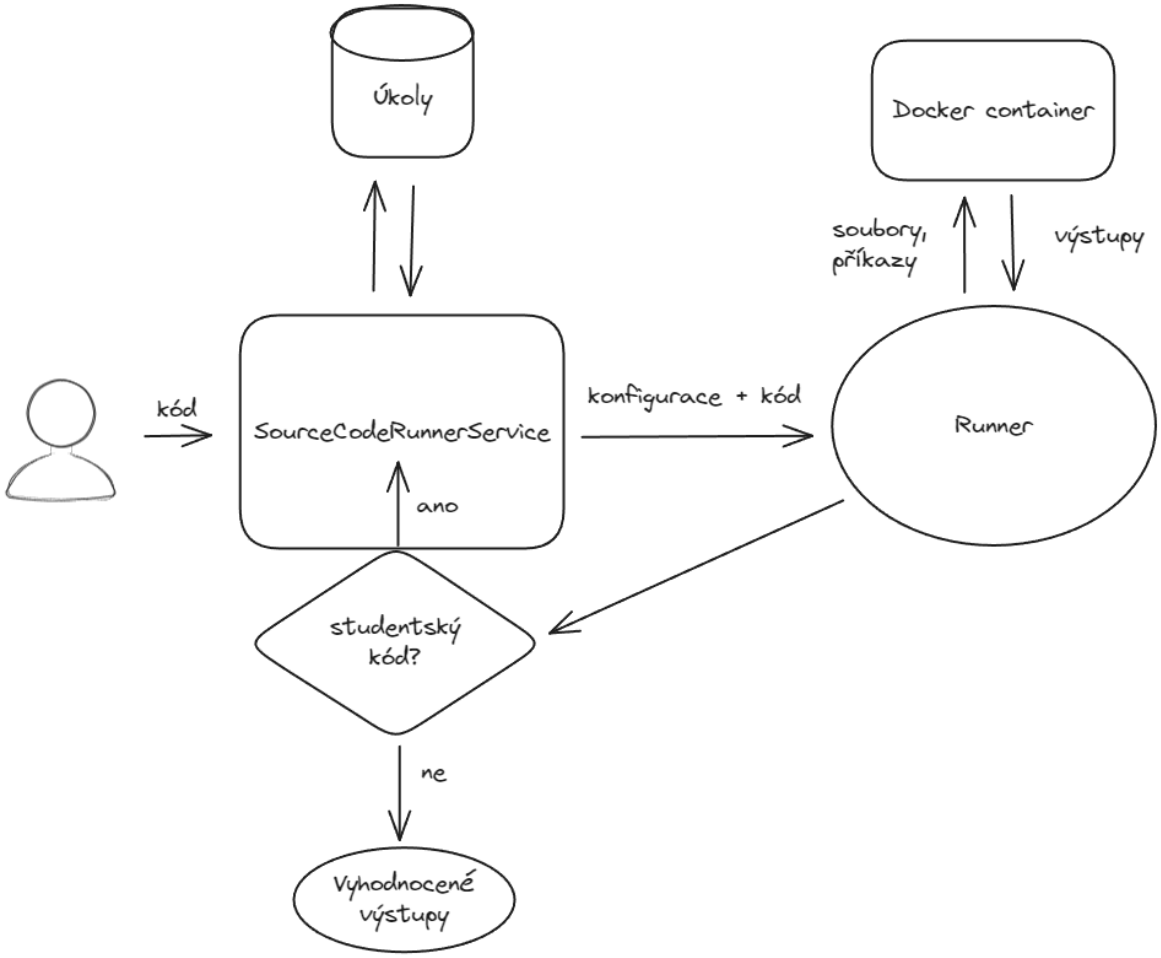
pohled se tento fakt jeví jako závažná překážka. Avšak pokud Docker kontejneru udělíme privilegovaný přístup, konkrétně přes udělení přístupu k Docker komunikačnímu soketu, kontejner dokáže samotnou Docker instanci ovládat a využít Docker CLI. Ke konfiguraci je nutné připravit vlastní Dockerfile, jenž stáhne Docker CLI a potřebné knihovny, přidat uživatele a namapovat socket pomocí Docker volume: „/var/run/docker.sock:/var/run/docker.sock“. Pro následné ovládání docker kontejnerů vznikly funkce v docker části aplikace. Node.js umožňuje spouštění příkazů mimo aplikaci skrze interface, který startuje příkaz v nové instanci shell.

Celý tento proces (viz obr. 13) začíná nahráním souboru určeného ke spuštění a kontrole. Hlavní třídou implementující funkcionalitu automatické kontroly a hodnocení kódu je servisní třída SourceCodeRunnerService. Z instance této třídy se volá patřičná třídy podle typu zdrojového kódu. Pokud je zdrojový kód určený jako referenční, tedy nahrávaný vyučujícím z formuláře zadávání úkolu, předloží se tento zdrojový kód nové instanci třídy s názvem Runner. Runner je třída, která má na starosti pouze běh kódu v kontejneru. V jeden moment může vložený kód zpracovávat pouze jedna instance třídy Runner. Tato třída používá nadefinované Docker funkce k tomu, aby sestavila a spustila patřičný kontejner dle Dockerfile, který byl vybrán vyučujícím. Tyto Dockerfiles jsou uloženy v projektu aplikace a nabízí možnost specifikovat nejen prostředí pro kontrolu a hodnocení zdrojového kódu. Vytvořený kontejner má omezené výpočetní zdroje, nemá přístup k internetu, kód uvnitř se spouští pod omezeným uživatelem a s životnostní 30 sekund. Následně Runner instance zkopíruje zdrojový kód do kontejneru a kód spustí. V případě, že je vložen kód běžící déle jak 30 sekund, Runner automaticky kontejner ukončí.

V rámci možnosti vkládání více souborů systém umožňuje vkládat i archivní soubory. Ke zpracování tohoto typu souboru je uvnitř kontejneru nadefinován pomocný skript s názvem „process_input_file.sh“. Jde o skript napsaný v jazyce Bash, jenž se uvnitř kontejneru používá k předzpracování zkopírovaného souboru. Proces spočívá v určení typu souboru. Pokud vložený soubor je zdrojový kód, skript soubor nemodifikuje. V případě archivního souboru dojde k extrahování souborů z archivního souboru a skript nalezne patřičný entrypoint soubor aplikace.

Po spuštění skriptu se kompiluje a spouští samotný zdrojový kód. Po spuštění se vrací výstupní hodnoty zpátky k instanci Runner, která kontejner správně ukončí a zavolá metodu, kterou instanci třídy Runner předala instance třídy SourceCodeRunnerService. Zavoláním této metody se provede patřičné vyhodnocení a zapíšou se hodnoty do databáze.

Při vložení zdrojového kódu ze strany studenta přes formulář odevzdávání zdrojového kódu, se zmíněný proces spouští dvakrát. Jednou pro studentský kód a následně pro referenční kód přiřazený k úkolu. Poté se provádí vyhodnocení na základě podobnosti výstupu referenčního a studentského zdrojového kódu.



Obr. 13 Zjednodušený diagram procesu kontroly a spouštění kódu. Zdroj: vlastní tvorba

5.4 Frontend aplikace

Tato kapitola popisuje hlavní strukturu a implementaci klientské části webové aplikace. Jednotlivé komponenty a vzhled aplikace byl implenetován a nastylován v rámci wireframů a návrhu. Podobně jako u předešlé kapitoly nebude popsán veškerý kód aplikace.

5.4.1 Autentizace požadavků

K autentizaci požadavků na webový server je nutné se nejprve autentizovat v IS Stag systému. Po úspěšné autentizace dojde k přesměrování zpátky na aplikaci s IS Stag tokenem. Po přihlášení přes systém IS Stag a přesměrování je potřeba provést několik důležitých operací, které umožní získání oprávnění k použití patřičných API rozhraní webového serveru.

První operací je předání IS Stag tokenu webovému serveru přes přihlašovací API. Při úspěšné autentizaci server vrací dva JWT tokeny: access a refresh token. Access token slouží k běžné autentizaci požadavků na webovém serveru. Mají však krátkou životnost a po vypršení životnosti se musí použít refresh token k získání nového access tokenu. Po přihlášení už klientská aplikace s IS Stag systémem přímo nekomunikuje a hned po získání těchto tokenů si aplikace uloží tyto tokeny do local storage prohlížeče. Následně aplikace využije uloženého access tokenu, aby si vyžádala informace o uživateli přes webový server.

Aby JWT tokeny byly používány v rámci dotazování se webového serveru a byla zautomatizována obnova access tokenu přes refresh token, využila se knihovna Axios (a nadstavba této knihovny Axios JWT) a její možnost přidat interceptors. Axios je moderní knihovna pro posílání REST API požadavků. Nabízí možnost definice interceptors. Jde o speciální funkce, které v našem případě mohou reagovat na odpověď ze serveru. [22]

Náš interceptor (obr. 14) reaguje na status kódy, které se objeví v rámci autentizační chyby. Pokud jsme použili access token a server nám vrátil tuto chybu, jde pravděpodobně o vypršelý access token. V rámci funce interceptor se vyžádá nový access token přes refresh token. Pokud i tato operace vrátí autentizační chybu (refresh token je neplatný nebo také vypršelý), uživatele odhlásí.

```
const requestRefresh: TokenRefreshRequest = async (refreshToken: string): Promise<string> => {
    const response: AxiosResponse<JwtAccesssTokenResponse> | void = await ax.post(
        backendBaseUrl + API.REFRESH,
        {
            refreshToken
        }
    ).catch(err => {
        logout();
    });

    if(response) {
        return response.data.accessToken;
    }
    return "";
}

applyAuthTokenInterceptor(axios, { requestRefresh, headerPrefix: "Bearer "});
```

Obr. 14 Definice Axios interceptor pro autentizační chyby. Zdroj: vlastní tvorba

5.4.2 Router klientské aplikace

K plynulému přechodu mezi jednotlivými sekcemi webové aplikace se používá knihovna React Router. Tato knihovna kontroluje momentální URL uživatele a na základě této URL poskytne uživateli správnou React komponentu K vytvoření React Router je třeba nadefinovat jednotlivé sekce aplikace a k nim patřičné komponenty (viz obr. 15).

```
const router = createBrowserRouter(
    createRoutesFromElements(
        <Route>
            <Route element={<App/>}>
                <Route path={LOCAL_ROUTES.ROOT} element={<Home/>} />
                <Route path={LOCAL_ROUTES.COURSES} element={<Course/>}>
                    <Route path=":courseSlug">
                        <Route path={LOCAL_ROUTES.TASKS} element={<Restricted navigateTo={
                            `/${LOCAL_ROUTES.COURSES}/:courseSlug/${LOCAL_ROUTES.TASK_ASSIGNMENTS}`} disallowedRoles={[Role.ST, Role.UN]}/>}>
                            <Route path="" element={<Tasks/>}/>
                            <Route path={LOCAL_ROUTES.ADDTASK} element={<TaskForm/>}/>
                            <Route path=":taskSlug" element={<TaskForm/>}/>
                        </Route>
                        <Route path={LOCAL_ROUTES.TASK_ASSIGNMENTS} >
                            <Route path="" element={<TaskAssignments/>}/>
                            <Route path=":taskSlug" element={<TaskAssignment/>}/>
                        </Route>
                        <Route path={LOCAL_ROUTES.INFO_EDIT} element={<Restricted navigateTo={
                            `/${LOCAL_ROUTES.COURSES}/:courseSlug/${LOCAL_ROUTES.INFO}`} disallowedRoles={[Role.ST, Role.UN]}/>}>
                            <Route path="" element={<CourseInfoForm/>}/>
                        </Route>
                        <Route path={LOCAL_ROUTES.INFO} element={<CourseInfo/>}/>
                        <Route path={LOCAL_ROUTES.STUDENTS}>
                            <Route path="" element={<SubjectStudents/>}/>
                            <Route path=":studentId" element={<CourseStudent/>}/>
                        </Route>
                    </Route>
                </Route>
                <Route path={LOCAL_ROUTES.NOTFOUND} element={<NotFound/>} />
            </Route>
            <Route path={LOCAL_ROUTES.LOGIN} element={<Login/>} />
            <Route path="*" element={<NotFound/>} />
        </Route>
    )
);
```

Obr. 15 Struktura aplikace v React Router specifikaci. Zdroj: vlastní tvorba

5.4.3 Využití JavaScript událostí

V některých případech navržená React struktura komponent znesnadňuje kontrolu komponent, které jsou v hierarchii výš. V našem případě se např. jedná o komponenty, zobrazující načítací obrazovku. Abychom načítací obrazovku aktivovali (např. při čekání na odpověď požadavku – viz obr. 16), v rámci naší aplikace můžeme kdekoli odeslat událost, kterou odchytí listener registrovaný na načítací komponentě. V tento moment listener předá událost komponentě a ta na základě vstupu vyhodnotí, jak změnit stav komponenty.

```
async function fetchCourses() {
    LoadingEvent.dispatch(true);
    const courses = await getCourses();
    setCourses(courses);
    LoadingEvent.dispatch(false);
}
```

Obr. 16 Příklad využití JS události v aplikaci. Zdroj: vlastní tvorba

6 Závěr

V této bakalářské práci jsme se zaměřili na návrh a implementaci webové aplikace pro zadávání, odevzdávání a automatickou kontrolu programátorských prací. Během vypracovávání práce byl sestaven návrh podle funkčních a nefunkčních požadavků a uživatelských rolí a potřeb. Návrh řešení je tvořen z několika wireframes popisující důležité obrazovky, které uživatel používá při práci s aplikací. V rámci návrhu byly stanoveny technologie aplikace skládající se z frameworků, knihoven a jazykových nadstaveb, jež se využívali při vývoji. Vyprodukovaný systém usnadňuje proces hodnocení programátorských úkolů a poskytuje uživatelům efektivní a intuitivní nástroj pro správu a kontrolu a hodnocení jejich práce. Pro své fungování systém využívá API rozhraní IS Stag zajišťující automatické propsání uživatelských údajů po prvním přihlášení a data potřebná pro chod aplikace.

Aplikaci je možné dále rozvíjet. Mezi možné cesty budoucího vývoje patří např. vytvoření nových automatizovaných Docker prostředí skrze Dockerfiles umožňující podporu dalších prostředí, které v podstatně nemusí být ani orientované na opravu programátorských prací. V rámci aplikace lze provádět vylepšení uživatelského rozhraní a zvyšovat jeho intuitivnost a použitelnost. Nedostatkem automatické opravy programátorských prací v našem systému spočívá v izolovanosti kontejnerů. Uvnitř kontejneru nelze k sestavení aplikace použít externí knihovny dostupné ke stažení v rámci balíčkovacích systému jako např. Npm pro Node.js programy. V momentálním nastavení se musí tyto knihovny nahrávat spolu se zdrojovým kódem. Použitím velkého množství knihoven se však zvyšuje velikost vloženého souboru a kvůli omezení velikosti vstupního souboru může být zdrojový kód odmítnut validací aplikace. Toto nastavení omezuje studenty využívat a učit se s knihovnami. Na druhou stranu je toto nastavení důležité pro bezpečnost infrastruktury a vede studenty k vymýšlení vlastních řešení pro implementaci. Systém také nedisponuje kontrolu plagiátů, protože by se v tomto případě jednalo od nad rámec této bakalářské práce.

Navzdory nedostatkům systému je práce kompletní a výsledný projekt odpovídá návrhu. Systém je funkční, responzivní a správně komunikuje s externím systémem Stag. V rámci prezentace projektu bylo vytvořeno testovací prostředí dostupné online. K tomuto prostředí lze nalézt více informací v návodu ke zprovoznění aplikace.

Seznam literatury

[1] *Vývoj interaktivní webové aplikace*. Diplomová práce. Pardubice: UNIVERZITA PARDUBICE FAKULTA ELEKTROTECHNIKY A INFORMATIKY, 2016.

[2] *Webový server*. Online. In: Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation, 2001-, 4. 10. 2023. Dostupné z: https://cs.wikipedia.org/wiki/Webov%C3%BD_server. [cit. 2024-07-10].

[3] YEAGER, Nancy a MCGRATH, Robert E. *Web server technology : the advanced guide for World Wide Web information providers*. San Francisco: Morgan Kaufmann Publishers, 1996. ISBN 1-55860-376-X.

[4] *About the Apache HTTP Server Project*. Online. HTTPS://HTTPD.APACHE.ORG. Apache HTTP server project. 2008. Dostupné z: https://httpd.apache.org/ABOUT_APACHE.html. [cit. 2024-07-10].

[5] *Nginx*. Online. In: Wikipedia: the free encyclopedia. San Francisco (CA): Wikimedia Foundation, 2001-, 5. 7. 2024. Dostupné z: . [cit. 2024-07-10].

[6] SURWASE, Vijay. REST API Modeling Languages - A Developer's Perspective. *IJSTE - International Journal of Science Technology & Engineering*. 2016, vol. 2, no. 10, s. 4. ISSN 2349-784X.

[7] HARDT, Dick. *The OAuth 2.0 Authorization Framework*. 32. 2012. ISSN 2070-1721. Dostupné také z: https://datatracker.ietf.org/doc/html/rfc6749.

[8] JONES, Michael B.; BRADLEY, John a SAKIMURA, Nat. *JSON Web Token (JWT)*. 33. 2015. ISSN 2070-1721. Dostupné také z: https://datatracker.ietf.org/doc/html/rfc7519.

[9] SINGH, Siddharth. MVC Framework: A Modern Web Application Development Approach and Working. International Research Journal of Engineering and Technology (IRJET). 2020, vol. 7, no. 1, s. 4. ISSN 2395-0072.

[10] Responsive Web Design, Discoverability, and Mobile Challenge. *Library Technology Reports*. 2013, vol. 49, no. 6, s. 11.

[11] SAKS, Elar. *JavaScript frameworks: Angular vs React vs Vue*. Bakalářská práce. Helsinki: Haaga-Helia University of Applied Sciences, 2019.

[12] LAZUARDY, Mochammad Fariz Syah a ANGGRAINI, Dyah. Modern Front End Web Architectures with React.Js and Next.Js. *International Research Journal of Advanced Engineering and Science*. 2022, vol. 7, no. 1, s. 10. ISSN 2455-9024.

[13] International Journal for Research in Applied Science & Engineering Technology (IJRASET) ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538 Volume 10 Issue VII July 2022- Available at www.ijraset.com 298© IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 | Comparative Analysis on Front-End Frameworks for Web Applications. *International Journal for Research in Applied Science & Engineering Technology*. 2022, vol. 10, no. 7, s. 12. ISSN 2321-9653.

[14] LEVLIN, Mattias. DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte. Diplomová práce. Turku: Åbo Akademi, 2020.

[15] MARASHDIH, Abdalla Wasef a ZAABA, Zarul Fitri. Web Security: Detection of Cross Site Scripting in PHP Web Application using Genetic Algorithm. *International Journal of Advanced Computer Science and Applications*. 2017, vol. 8, no. 5, s. 13.

[16] CVETKOVIĆ, Stevica; JANKOVIĆ, Dragan. A comparative study of the features and performance of orm tools in a. net environment. In: *Objects and Databases: Third International Conference, ICOODB 2010, Frankfurt/Main, Germany, September 28-30, 2010. Proceedings 3*. Springer Berlin Heidelberg, 2010. p. 147-158.

[17] COSMINA, Iuliana; HARROP, Rob; SCHAEFER, Chris a HO, Clarence. *Pro Spring 5*. Online. Berkeley, CA: Apress, 2017. ISBN 978-1-4842-2807-4. Dostupné z: https://doi.org/10.1007/978-1-4842-2808-1. [cit. 2024-07-17].

[18] BEAN, Martin. *Laravel 5 essentials*. Packt Publishing Ltd, 2015.

[19] LOCK, Andrew. *ASP. NET core in Action*. Simon and Schuster, 2023.

[20] MARDAN, Azat. Express. js Guide: The Comprehensive Book on Express. js. Azat Mardan, 2014.

[21] SCHENKER, Gabriel N. Learn Docker-Fundamentals of Docker 18. x: Everything you need to know about containerizing your applications and running them in production. Packt Publishing Ltd, 2018.

[22] THE AXIOS PROJECT. *Axios*. Online. 2020. Dostupné z: https://axios-http.com/. [cit. 2024-07-17].

Návod ke zprovoznění aplikace

Aplikace ke svému spuštění a následnému správnému fungování vyžaduje nainstalované prostředí Docker. Protože tato platforma využívá virtualizace dostupné v Linux operačním systému, je v případě spuštění aplikace nutné mít Docker prostředí propojené s nainstalovaném prostředí WSL. K vývoji byl použit Docker verze 4.32.0 a Docker engine verze 26.1.4. Po instalaci prerekvizit lze aplikaci spustit v kořenu projektu skrze příkaz „docker compose up“. Při spuštění tohoto příkazu se vytvoří patřičné kontejnery aplikace včetně databázového kontejneru. Po spuštění je webová aplikace na hostitelské stroji dostupná v prohlížeči pod URL http://localhost:8080.

Systém lze také provozovat bez příkazu „docker compose“. Pomocí dockeru lze spustit kontejnery zvlášť přes „docker run“ příkaz. Před tímto příkazem se musí však ještě sestavit vlastní Docker image projektu. Navíc musí být pro tyto účely dostupná databáze, jejíž URL lze editovat .env souboru aplikace.

Příklad Docker run příkazu:
docker build -t fixer/node .

docker run -d --restart always --name fixer -p 80:80 -u node -v /var/run/docker.sock:/var/run/docker.sock -v $(pwd):$(pwd) -w $(pwd) fixer/node bash -c "npm ci && npm run prisma:setup && npm start"

Protože aplikace vyžaduje poměrně náročnou instalaci Docker prostředí, bylo v rámci testování funkcionalit a usnadnění interakce vytvořeno testovací prostředí dostupné skrze URL http://krystofliskovec.cz:6181 a je pro testování **doporučováno**. Testovací prostředí nabízí změnu role uživatele přímo v aplikaci pro snaší používání.

Přílohy

Zdrojový kód aplikace