Doporučení pro implementaci udržitelných webových aplikací

Bc. Kristýna Tomanová

Diplomová práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Kristýna Tomanová**
Osobní číslo:	**A21166**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Doporučení pro implementaci udržitelných webových aplikací**
Téma práce anglicky:	**Recommendations for Implementation of Sustainable Web Applications**

Zásady pro vypracování

1. Vypracujte literární rešerši na téma zásady kvalitního programování (tj. čistý kód, architektura kódu).
2. Popište postupy pro implementaci RESTful API.
3. Rozveďte způsoby testování kódu.
4. Věnujte se zabezpečení webové aplikace.
5. V rámci praktické části aplikujte poznatky na konkrétní příklady implementace.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. MARTIN, Robert C. Clean code: A Handbook of Agile Software Craftmanship. Stoughton (Massachusetts): Pearson Education, c2009. Robert C. Martin series. ISBN 978-013-2350-884.
2. MARTIN, Robert C. The clean coder: A Code of Conduct for Professional Programmers. Crawfordsville (Indiana): Pearson, 2011. Robert C. Martin series. ISBN 01-370-8107-3.
3. KHORIKOV, Vladimir. Unit Testing: Principles, Practices and Patterns. 2020. Shelter Island (New York): Manning Publications, 2020, 304 s. ISBN 9781617296277.
4. FIELDING, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Irvine, California, 2000. Dostupné také z: https://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm. Disertace. University of California Irvine.
5. KRUG, Steve. Don't Make Me Think, Revisited: A Common Sense Approach to Web (and Mobile) Usability. 3rd ed. San Francisco (California): Pearson Education, 2013, 216 s. Voices That Matter. ISBN 978-032-1965-516.
6. OWASP Foundation: the Open Source Foundation for Application Security [online]. Wakefield (Massachusetts): OWASP Foundation, c2022 [cit. 2022-11-27]. Dostupné z: https://owasp.org/

Vedoucí diplomové práce: **Ing. Radek Vala, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **2. prosince 2022**
Termín odevzdání diplomové práce: **26. května 2023**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 26. května 2023

Bc. Kristýna Tomanová v.r.
podpis studenta

ABSTRAKT

Diplomová práce se věnuje metodám, kterými lze zajistit udržitelnost webové aplikace. Práce je soustředěna na zásady kvalitního programování, jejichž součástí je čistý kód, testování kódu, práce s gitem, architektura kódu a dále zajištění stabilní a spolehlivé aplikace pro uživatele se zaměřením na zabezpečení. V praktické části budou zásady použity na implementaci API pro ukázku zpracování.

Klíčová slova: čistý kód, webová aplikace, API, udržitelnost softwaru, Laravel

ABSTRACT

The master's thesis focuses on methods that can be used to ensure the sustainability of a web application. The thesis focuses on the principles of quality programming, which include clean code, code testing, working with git, code architecture, as well as ensuring a stable and reliable application for the user, with a focus on security. As a demonstration, the principles will be applied to an API implementation in the second part of the thesis.

Keywords: clean code, web application, API, web suitability, Laravel

Tímto bych chtěla poděkovat panu Ing. Radku Valovi, Ph.D. za odborné vedení práce a mým kolegům za cenné rady, které pomohly definovat obsah této práce.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Práce vývojáře vyžaduje kvalitní technické dovednosti, které lze získat studiem i praxí. Během studia dochází k získání teoretických znalostí, jejichž část je následně uplatněna v praxi a rozvíjí se směrem, který určuje především náplň práce. Důležitost těchto dovedností a znalostí nelze popřít a jsou základními stavebními kameny kariéry. Nicméně v otázkách praxe není, *jestli* se vývojáři podařilo práci odvést, ale především *jak dobře* a *jak rychle*, což jsou dva parametry, které jsou často v rozporu a existuje mezi nimi určitá nepřímá úměra. Během kariéry pak vývojář pracuje na tom, aby se tento koeficient zmenšoval, a nakonec došlo k tomu, že vyvíjí dobře a rychle (v lepším případě výborně a rychle). Problémem ovšem je neznámá rovnice tohoto výpočtu, která dělá z karierního postupu ne zcela jednoduchou záležitost.

Otázka rychlosti vývoje je velmi subjektivní a pro každého zcela individuální. Základním předpokladem této práce je, že rychlost se zlepšuje s nabývajícími zkušenostmi a znalostmi. Práce se soustředí na první parametr, a to je kvalita vývoje, kterou lze ovlivňovat již v raných stádiích kariéry.

Tato práce se věnuje tématům, která můžou vývojáři poskytnout možnosti ke zdokonalení a další odrazové body pro získání správného směru ke kvalitnímu kódu. Cílem této práce je představit začínajícím vývojářům základní témata, kterým se věnovat pro zlepšení svých dovedností a pracovní reputace.

V dnešní době již není nic neobvyklého být vývojářem a vyvíjet aplikace. Na pracovním trhu je spoustu vývojářů, spousta firem vývojáře hledá a aplikace je už téměř na všechno, co lze vymyslet. Proto je důležité získat takové kvality, které vývojáře odliší v davu a umožní mu získat konkurenční výhody.

I. TEORETICKÁ ČÁST

1 DOPORUČENÍ PRO KVALITNÍ PROGRAMOVÁNÍ

Hlavní náplní práce vývojáře je psaní kódu – programování. K získání znalostí programování je třeba naučit se libovolnému programovacímu jazyku díky informacím z knih, kurzů, tutoriálů atp. a následně získávat zkušenosti pomocí psaní programů od nejjednodušších ke složitějším.

Nicméně hodnota vývojáře nespočívá v tom, jak složité funkce zvládne naprogramovat a kolik programovacích jazyků ovládá, či jak dlouho už svou práci vykonává. Při náborových procesech je v dnešní době kladen důraz především na jemné dovednosti a zásady uchazeče a jejich aplikace přímo v kódu. Při splnění úkolu se tedy nepohlíží tolik na funkcionalitu, nýbrž na kvalitu poskytnutého řešení. Jinými slovy, pokud vývojář nezná některé implementační praktiky, doučí se je v rámci zaučování rychleji než v případě, že mu chybí správné návyky, které se učí mnohem obtížněji a často vyžadují čas a velké úsilí – a hlavně chtíč – k tomu, aby se požadovaným schopnostem naučil.

V těchto podkapitolách jsou sepsány užitečné návyky vývojáře, které mohou dopomoci ke zvýšení jeho kvality a ovlivnit jeho přístup k programování.

1.1 Zásady čistého kódu

Čistý kód se stal termínem, který by měl být vlastní každému programátorovi. Nicméně praktiky jsou často subjektivní a může být věc názoru, které jsou vhodné a které nikoliv. Praktiky jsou většinou sepsány do zásad, které popisují jejich pravidla dodržování.

Osvojení si psaní čistého kódu je dlouhodobá záležitost, a dá se říct, že ji nikdy nelze brát za uzavřenou, v rámci toho, že praktiky se často aktualizují a přibývají nové.

1.1.1 Kódové standardy

Většina jazyků má definována doporučení, kterých se lze držet pro zajištění čistého vývoje. Tato doporučení nejsou povinná, ale slouží jako základ pro přehledný projekt. Kódové standardy pro webový vývoj znázorňuje Tabulka 1.

Tabulka 1. Kódové standardy jazyků pro webový vývoj [vlastní zpracování]

Programovací jazyk	Standardy
Javascript	Airbnb Javascript Style Guide, Google Javascript Style Guide
Python	PEP – Python Enhancement Proposals, Google Python Style Guide
Ruby	Ruby Style Guide, Airbnb Ruby Style Guide
PHP	PSR – PHP Standards Recommendations, Zend Framework Coding Standard
Java	Oracle Java Code Conventions, Google Java Style Guide
C#	Microsoft C# Coding Conventions

1.1.2 Formátování

Prvním základním kamenem je určení konvencí správného formátování. Formátování i jeho kontrolu lze nastavit v IDE. Velmi často jsou i součástí již zmíněných kódových standardů. Pokud na kódu spolupracuje více vývojářů, tak rozdílné formátování nejenže působí nevzhledně, ale hlavně způsobuje zmatky během toho, co se kontroluje *pull-request*, kde jsou i tyto drobné úpravy zaznamenány a zvýrazněny, čímž odklání pozornost od závažnějších chyb, které je potřeba identifikovat.

1.1.2.1 Definice délky řádku

Pro přehlednost a jednoduché čtení a orientaci v kódu je doporučeno vyhnout se neomezené délce řádku, aby se zamezilo horizontálnímu posouvání v kódu. Tuto délku je potřeba nastavit na optimální hranici, která není příliš krátká, a především příliš dlouhá. Existují samozřejmě výjimky, kdy toto pravidlo dodržet nelze, nicméně je vhodné si hranici určit a překračovat ji opravdu jen v případě, že nelze postupovat jinak.

Délka je určená počtem použitých znaků a doporučuje se nastavit ji mezi 80 až 120 znaky na řádek a záleží i na konkrétním jazyce, který je použit a na preferenci vývojářů. [1] Často je doporučení stanoveno ve výše zmíněných konvencích.

1.1.2.2 Použití bílých znaků

Při psaní kódu dochází k velkému množství použití bílých znaků jako jsou mezery, tabulátory nebo nové řádky.

Pro správné odlišení mezer od tabulátorů, je doporučeno využít funkcionality IDE, která zvýrazní tyto znaky a pomůže s vizuální kontrolou správnosti použitých znaků.

Nejvíce se toto pravidlo používá při odsazení jednotlivých řádků. V běžných prostředích odsazení již probíhá automaticky a záleží právě na nastavení, které je třeba na počátku definovat. Kvůli délce řádku definované v předchozí kapitole, je mnohem výhodnější používat pro odsazení mezery namísto tabulátorů, zároveň ale tabulátory jsou jednodušší k použití. Nicméně dnešní technologie umožňují při použití tabulátoru vkládat mezery. Nastavení znaku odsazení tedy záleží čistě na preferenci a domluvě. Jak již bylo zmíněno několikrát, důležité je konvence dodržovat. Obrázek 1 ukazuje, jak IDE zobrazuje mezery k odsazení a Obrázek 2 zase zobrazení využitých tabulátorů pro odsazení.

```
class Controller extends BaseController
{
    use AuthorizesRequests;
    use ValidatesRequests;
}
```

Obrázek 1. Vizuální zobrazení použití mezer k odsazení [vlastní zpracování]

```
class Controller extends BaseController
{
——use AuthorizesRequests;
——use ValidatesRequests;
}
```

Obrázek 2. Vizuální zobrazení použití tabulátoru pro odsazení [vlastní zpracování]

K vertikálnímu formátování je vhodné oddělovat logické celky uvnitř funkcí pomocí prázdného řádku. Stejně tak by měly být konstantní velikosti mezer mezi funkcemi a dalšími bloky tříd.

V některých jazycích bývá ve zvyku využívat i horizontálního formátování čili zarovnávání textu pro zvýšení přehlednosti. Oddělují se tak například definice proměnných a jejich názvů nebo i přiřazování. Nevýhodou využití tohoto formátování je ovšem to, že, v případě zavedení nové delší proměnné, je nutná úprava celého bloku využívajícího horizontálního formátování, čímž je znepřehledněn seznam změn provedených v souboru.

U bílých znaků je vhodné se věnovat i větším detailům, jako je používání mezer či nových řádků mezi závorkami, za názvy funkcí nebo klíčových slov, které by měly být také skrze projekt nastaveny na stejné konvence. Těmto konvencím se podrobněji věnují kódové standardy jednotlivých jazyků a v této práci nebudou podrobněji rozebírány.

Důležitým aspektem tedy je správně si před začátkem projektu nastavit IDE, které bude tato pravidla dodržovat. Lze také pomocí nastavené klávesové zkratky využívat automatické opravy a pravidelně ji používat. Při práci v týmu je vhodné využít sdíleného společného nastavení, které nástroje v dnešní době podporují.

1.1.3 Typy písemných tvarů

Kromě správného určení názvu je dalším faktorem i dodržování ustáleného písemného tvaru. Ten by měl být určen ještě před začátkem implementace právě z důvodu konzistence. Tabulka 2 na následující straně definuje možné typy písemných tvarů.

Tabulka 2. Typy písemných tvarů [2]

Písemný tvar	**Slovní popis**	**Využití**
`flatcase`	malá písmena bez mezer	logo, domény
`UPPERCASE`	velká písmena bez mezer	konstanty
`camelCase`	velká první písmena slov vyjma prvního	proměnné, funkce
`PascalCase`	velká první písmena slov	třídy
`snake_case`	malá písmena, slova oddělená podtržítkem	SQL, proměnné, funkce
`SCREAMING_SNAKE_CASE`	velká písmena, slova oddělená podtržítkem	konstanty
`kebab-case`	malá písmena, slova oddělená spojovníkem	HTML, CSS
`Pascal_Snake_Case`	velká první písmena slov, slova oddělená podtržítkem	Ada
`COBOL-CASE`	velká písmena, slova oddělená spojovníkem	COBOL
`Train-Case`	velká první písmena slov, slova oddělená spojovníkem	hlavička HTTP

Existují různé konvence a doporučení pro jednotlivé jazyky. Hlavním důvodem zvolením určitého písemného tvaru je nejlepší čitelnost v rámci kódu.

V dnešní době už ale i doporučení jazyků má menší váhu a programátoři často pro nejlepší čitelnost volí `camelCase` pro proměnné a funkce, `PascalCase` pro třídy a `SCREAMING_SNAKE_CASE` pro konstanty.

1.1.4 Notace

Dalšími konvencemi, které jazyky využívají, jsou některé používané notace při pojmenování. Zde jsou uvedeny nejběžnější notace, se kterými se lze během programování setkat.

Podtržítková notace (anglicky *underscore notation*) využívá před názvem proměnných či funkcí podtržítko (příklad `_hiddenProperty`). Tato notace je využívána pro označení privátních prvků. Je doporučena používat v případě, že jazyk neumožňuje skrytí prvku z vnějšku, či pro označení mezi vývojáři, že proměnná nemá být modifikována zvenčí. [3]

Uherská notace (anglicky *Hungarian notation*) vkládá datový typ jako předponu proměnné (příklad `strName`). Toho je doporučeno využívat u netypových jazyků pro zlepšení čitelnosti kódu. S touto notací přišel v 70. létech minulého století americký miliardář Charles Simonyi, který je původem Maďar, když pracoval v Xerox PARC, a později ji společně s objektově orientovaným programováním zavedl ve firmě Microsoft, kde pracoval na WYSIWYG (*What You See Is What You Get* v překladu Co vidíš, to dostaneš)[1] textovém procesoru, ze kterého vzešly aplikace Word a Excel. [4] V Xerox PARC se využívalo pojmenování s datovým typem jako příponou a Simonyi navrhl toto pořadí změnit. Důvodem, proč je notace nazvaná uherská, je že maďarská jména se používají v opačném pořadí (nejdřív příjmení a poté jméno) než obvykle u nás – první jméno a potom příjmení. [6]

1.1.5 Jmenné konvence

Součástí pravidel čistého kódu je důležité dodržovat jmenné konvence. Správné pojmenování je zásadní pro orientaci v kódu a jedná se o základní a nejdůležitější pravidlo čistého kódu.

1.1.5.1 Definování záměru

Hlavní podstatou pojmenování je určení smyslu použití. Je to i hlavní princip aktuálního programování a výhoda oproti nízko-úrovňových jazyků jako je assembler. Pokud není definován vhodný název, pak už není důvod, proč se neodkazovat rovnou na adresu paměti. Jméno by mělo dostatečně odhalovat záměr tak, aby nebylo potřeba volbu jména nijak komentovat.

Toto pravidlo se týká nejen tříd, jejich metod a vlastností, ale i všech proměnných, funkcí, konstant, co se v kódu vyskytují. Usnadní čas jak ostatním vývojářům, tak i implementujícímu vývojáři, který se na kód podívá s časovým odstupem.

---

[1] WYSIWYG nástroje fungují na principu přesném zobrazení reálného výstupu již během editace souboru. Používají se právě pro editory k tisku souborů nebo tvorbu front-endových obsahů (pro vzhled aplikací).

1.1.5.2 Jednoznačný název

Velkým problémem můžou být dvojsmyslné názvy, nebo názvy, jejichž nositelé změnili svůj záměr s uplynulým časem a vylepšením. Nejúčinnější kontrolou je prozatím kontrola dalším vývojářem, který může problematická místa lépe identifikovat.

1.1.5.3 Ucelený tvar pojmenování

Stejně jako dvojsmysly, tak čistotu kódu ovlivňují i synonyma. Je vhodné definovat si jednotné pojmenování pro často používané názvy. Zde je nutné upozornit na používání knihoven třetích stran, které stanovené konvence v projektu mohou narušit. K tomuto je na místě zvážit zaobalení knihoven do vlastních tříd, které se budou v kódu využívat.

1.1.5.4 Proměnné

Proměnné datového typu `bool` by měl být ve formátu otázky s odpověďmi ano/ne. Například `shouldSelect`, `isSelected`, `willSelected` nebo `mustSelect`. Ostatní datové typy jsou pojmenovány podstatným jménem.

V případě proměnných obsahujících více položek je nutné nezapomenout na množné číslo názvu.

1.1.5.5 Funkce

Funkce by měly být pojmenovány slovesnou frází, nejčastěji potom v imperativu, například `createUser()`, `sendResponse()` atp.

Základním pravidlem u funkcí je, že by měla být malá. Jak již bylo definováno v předchozí kapitole, funkce by měla být malá na šířku, ale stejně tak i na výšku. Měla by být tak malá, že by neměla obsahovat zanořenou strukturu. To znamená, že každý blok `if`, `else` a `while` by měl být dlouhý jeden řádek. Často to znamená, že právě tento jeden řádek bude volání funkce s dostatečně deskriptivním názvem, který zdokumentuje záměr bloku. [6]

Velikost funkce je úzce spojená i s jejím jménem a záměrem. Funkce by měla dělat pouze jednu jedinou věc. To znamená, že součástí funkce je pouze jedna úroveň abstrakce. K tomuto se pojí i pravidlo, že třída by neměla vytvořit žádný vedlejší efekt. [6]

Funkce se dělí podle počtu argumentu na

- Niladické – funkce nemá žádný argument,
- Monadické – funkce má jeden argument,

- Dyadické – funkce má dva argumenty,
- Triadické – funkce má tři argumenty,
- Polyadické – funkce má více než tři argumenty.

Ideální funkce je niladická či monadická, ve výjimečně pak dyadická. Triadické a polyadické funkce by měly být redukovány na minimum. V případě toho, že je argumentů více než jeden, je třeba zvážit, zda by neměly argumenty být zastoupeny společnou třídou, nebo jestli by nešel využít list argumentů. [6]

1.1.5.6 Třídy

Pojmenování tříd je zpravidla podstatnými jmény. Třída má pro sebe samostatně vyhrazený soubor, jehož jméno odpovídá s jejím jménem.

Při psaní tříd je pravidlem, že funkce by měly být řazeny dle používání, jak jsou volány za sebou. Tím dojde k minimalizaci vertikální vzdálenosti tedy nutnosti posouvání se z místa na místo v souboru. [6]

1.1.6 Vývojové principy

Vývojové principy jsou nedílnou součástí čistého kódu. Jedná se o pravidla, která pomáhají udržovat kód přehledný a čistý. Stejně jako u samotného čistého kódu, principy jsou složité k osvojení a je důležité je míti na paměti.

1.1.6.1 Principy SOLID

Principy SOLID jsou doporučení transformovaná do sady pravidel, která byla vytvořena po letech shromažďování zkušeností softwarových vývojářů z celého světa. Tyto principy umožňují vývojáři psát software, který je jednodušší pro údržbu, pochopení a testování a rychlejší pro vývoj v týmu. [7]

Tyto principy obsahují pět následujících pravidel:

S – *Single Responsibility Principle* (česky princip jediné zodpovědnosti, zkráceně SRP), který říká, že třída by měla mít pouze jeden důvod ke změně neboli že třída by měla mít na starosti pouze jeden úkol.

O – *Open-Closed Principle* (česky princip otevřenosti-uzavřenosti, zkráceně OCP), který tvrdí, že objekt by měl být otevřený k rozšíření, ale uzavřený ke změnám.

L – ***Liskov Substitution Principle*** (česky Liskovové princip zastoupení, zkráceně LSP). Ten definuje: *Nechť $\phi(x)$ je vlastnost prokazatelná objektu x typu T. Potom $\phi(y)$ jsou pravdivé pro objekty y typu S, kde S je potomkem T.* [8] Což znamená že objekt rodičovské třídy by měl být nahraditelný jakýmkoliv objektem potomků. Ještě jinak řečeno, že každý potomek by se měl chovat stejně jako objekt rodičovské třídy.

I – ***Interface Segregation Principle*** (česky Princip segregace rozhraní, zkráceně ISP), který určuje, že klient by neměl být nucen implementovat rozhraní, které nepoužívá, nebo záviset na metodách, které nepoužívá.

D – ***Dependency Inversion Principle*** (česky Princip inverze závislostí, zkráceně DIP). Ten tvrdí, že entity musí záviset na abstrakcích, nikoliv konkrétních třídách. Neboli moduly vyššího levelu nesmí záviset na modulech nižšího level, ale měly by záviset na abstrakcích.

1.1.6.2 Princip KISS

Princip KISS je zkratka pro *Keep It Simple, Stupid* neboli česky „Udržuj to jednoduché, hlupáčku“. Tento princip si zakládá na tom, aby kód produkoval stejné nebo dokonce lepší výsledky s menší složitostí a náročností. Vždy jde o to napsat pouze a právě tolik kódu, kolik je vyžádáno. V programování to znamená:

- zajistit odpovídající názvy proměnných pro hodnoty, které drží,
- zajistit, aby názvy metod vypovídaly o jejich účelu,
- zajistit jedinou zodpovědnost tříd,
- komentovat pouze potřebné části kódu, které nelze definovat metodou,
- vyhnout se globálním prvkům, pokud je to možné,
- mazat redundantní kód nebo kód, který není používán. [9]

Kent Beck definoval sadu kritérií, které určují, zda je zdrojový kód dostatečně jednoduchý:

- kód je ověřen automatizovanými testy, které všechny byly pozitivně ohodnoceny,
- kód neobsahuje duplikace,
- kód od sebe odděluje odlišné záměry a odpovědnosti,
- kód je sestaven z minimálního počtu komponent (tříd, metod, řádků), které splňují předchozí tři body. [10]

1.1.6.3 Princip DRY

Princip DRY znamená *Don't Repeat Yourself* česky „neopakuj se". Tento princip byl zpopularizován knihou Pragmatický Programátor [11], ve které autor Andrew Hunt zmiňuje, že každý kousek znalosti musí mít jedinou, jednoznačnou a směrodatnou reprezentaci v systému.

1.1.6.4 Princip YAGNI

You Aren't Gonna Need It neboli YAGNI, česky „to nebudeš potřebovat", je princip, který apeluje na redukci nepotřebné funkcionality nebo logiky. Tento princip je součástí extrémního programování XP a zakládá na správné analýze požadavků, aby nedocházelo k „přeimplementování", tedy implementace více, než si požadavek žádá. Často je to z toho důvodu, že v plánování dochází k předvídání budoucnosti a snaha budování kódu v závislosti na ještě neexistujících podmínkách/požadavcích. [12]

V programátorské praxi to vypadá tak, že vývojář vytváří příliš mnoho abstrakcí, i když nejsou potřeba, nebo hledá logiku, která nemůže nastat.

1.1.6.5 Princip SoC

Princip rozdělení zájmů (anglicky *Separation of Concerns*, zkráceně SoC) je jednoduchý princip vyžadující rozdělení jednoho velkého bloku kódu do menších komponent, které provádějí kompletní a jednoduchou práci.

Pokud je napsána složitá funkce s příliš mnoha argumenty, pak by měla být rozdělena na menší komponenty. Funkce by měly být seskupeny do logických modulů, které naplňují požadovanou logiku. Funkce s podobným záměrem by měly být seskupeny dohromady. Naopak podobné funkce s odlišným záměrem by měly být odděleny pro lepší čitelnost. [13]

1.2 Vývojové strategie

Vývojová strategie určuje způsob postupu pro implementaci softwaru. V následujících podkapitolách budou představeny některé vývojové strategie, které se v dnešní době používají.

1.2.1 Vývoj řízený testy

Vývoj řízený testy (anglicky *test-driven development*, zkráceně TDD) je styl programování, který probíhá pomocí tří aktivit (kódování, jednotkové testování, *refactoring*) následovně:

1. Napsání jednoho jednotkového testu popisující část implementované funkcionality,

2. Spuštění daného testu a jeho selhání, kvůli chybějící implementaci,
3. Sepsání minimálního množství kódu pro splnění požadavků textu,
4. *Refactoring* kódu pro uspokojení pravidel jednoduchosti,
5. Opakování od bodu 1. [14]

1.2.2 Vývoj začínající rozhraním

Vývoj začínající rozhraní (anglicky *API-first development*) je vývojový model, který prioritizuje API. Tedy že koncový produkt bude nabízen zařízeními s klientskou aplikací, které budou konzumovat API. Tento přístup zajistí, že vývoj API bude konzistentní a znovupoužitelný, čili vede ke kvalitnímu designu API a lepší soudržnosti mezi *backend* a *frontend* částí vývoje.

1. Definice potřebných API a sepsání jejich případů užití,
2. Definice potencionálních koncových bodů (*endpoints*) pro jednotlivé případy užití
3. Určení API stakeholderů (neboli zainteresovaných stran)
4. Definice konvencí použitých pro API
5. Návrh API a tvorba testovacího (*mock*) API
6. Implementace API
7. Vydání API. [15]

Klientská aplikace může být navrhována od kroku 5, kdy může začít používat testovací koncové body (anglicky *mock endpoints*). *Mock* vytváří šablonu získané odpovědi API a tím slouží jako simulace *backend* části aplikace.

1.2.3 Vývoj řízený chováním

Vývoj řízený chováním (anglicky *Behavior-Driven Development*, zkráceně BDD) je koncept založený na vývoji řízeném testy, který se soustředí na požadavky koncového uživatele a jeho interakci s produktem. BDD používá jazyk specifický pro doménu a pevně daný syntax pro testy. Jedná se o vhodný přístup k automatizovanému testování, jelikož se zaměřuje na chování více než na implementaci kódu.

Po konzultaci klientových potřeb jsou definovány požadavky mezi klientem, zákazníkem, vývojářem a testerem. Ty jsou poté převedeny do strukturovaných scénářů, které slouží jako základ pro automatizované testy. Tyto scénáře bývají napsané ve specifickém jazyku (např. Gherkin). [16]

1.2.4 Vývoj řízený vlastnostmi

Vývoj řízený vlastnostmi (anglicky *Feature-driven development*, zkráceně FDD) organizuje softwarový vývoj okolo požadovaných funkcionalit. Postup tohoto vývoje je následující:

1. Vytvoření celkového modelu
2. Definování seznamu funkcí
3. Plánování po funkci
4. Navrhování po funkci
5. Implementování po funkci. [17]

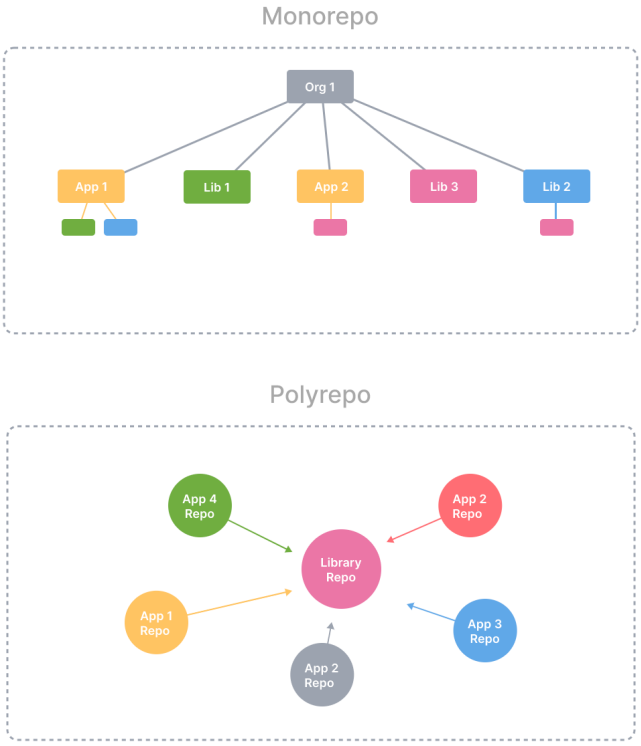
1.2.5 Vývoj řízený specifikací

Vývoj řízený specifikací (anglicky *Specification-driven development*, zkráceně SDD) se řídí pomocí specifikace. To znamená, že zároveň při implementaci jsou sepisovány části dokumentace. Tento styl předchází chybějícím dokumentacím a vybízí k udržení aktuální dokumentace kódu. [18]

1.3 Organizace repozitáře

Monorepo neboli monolitický repozitář je verzovaný repozitář, který obsahuje více příbuzných projektů, které na sobě nemusí logicky záviset. Tuto architekturu využívají společnosti jako Twitter, Microsoft, Facebook či Google stejně tak open-source projekty jako Laravel, Symfony, Babel, React aj.

Opakem monorepa je polyrepo, u kterého má projekt svůj repozitář zvlášť. Rozdíl je znázorněn na obrázku na další straně.



Obrázek 3. Rozdíl mezi monorepem a polyrepem [19]

U tohoto přístupu dochází k lepší spolupráci mezi týmy a lepší přehled nad projekty a sjednocení vývojových procesů. Problémy může dělat výkon některých operací jako je práce s gitem a průběh testů. [20]

1.4 Architektury softwaru

Architektury softwaru definují strukturu projektu spíše z vnějšího pohledu.

1.4.1 Monolitická architektura

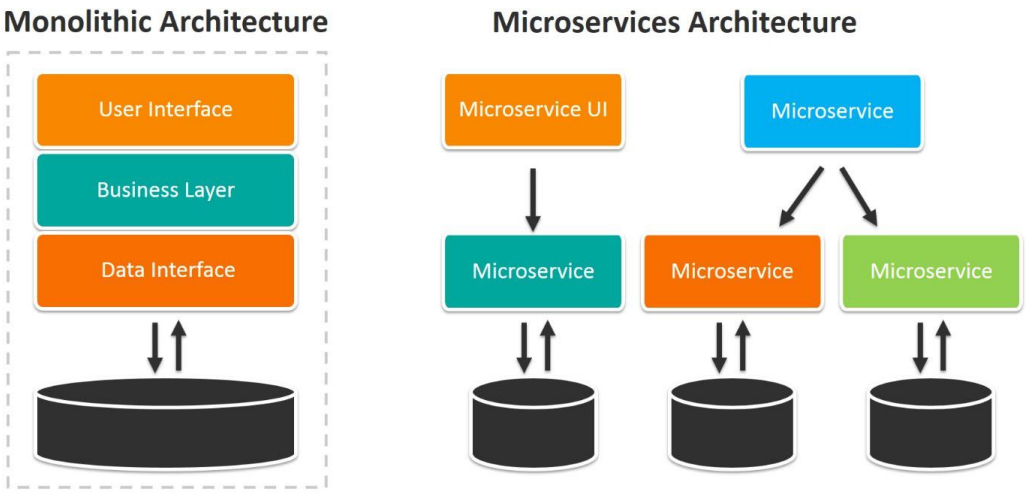
Monolitická architektura je tradiční koncept vývoje aplikací pomocí sjednoceného modelu pro celý software. Jedná se o samostatnou plnohodnotnou aplikaci, jejíž komponenty jsou úzce svázané a musí být dostupné během použití. [21]

1.4.2 *Microservice architecture* MSA

Architektura *Microservices* (anglicky *Microservice Architecture*, zkráceně MSA) je specifickou vývojovou strategií, která rozděluje aplikaci na nezávislé malé části. Každá z nich funguje jako nezávislá služba, která se jednotlivě upravuje.

Tuto architektura využívají společnosti Netflix, Amazon, Uber a eBay.

Obrázek níže znázorňuje graficky rozdíl mezi monolitickou architekturou a MSA.

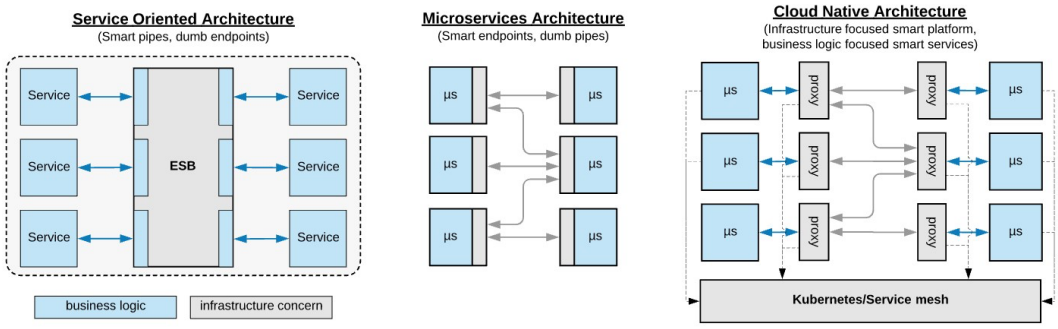


Obrázek 4. Rozdíl mezi monolitickou a architekturou MSA [22]

1.4.3 Servisně orientovaná architektura SOA

Servisně orientovaná architektura SOA (anglicky *Service Oriented Architecture*) umožňuje softwarovým komponentám fungovat jako oddělená a nezávislá jednotka. Každá služba v SOA obsahuje kód a data potřebná k provedení kompletní business funkce. Rozhraní servisu poskytuje slabou vazbu, což znamená, že mohou být volány s minimální znalostí implementace služby, čímž jsou odstraněny závislosti mezi aplikacemi. [23]

Obrázek 5 jsou znázorňuje rozdíly mezi SOA a MSA. Zatímco servisně orientovaná architektura spoléhá na ESB (anglicky *Enterprise Service Bus*), ve kterém se centralizovaná *routing* vrstva stará o koncové body, u architektury *microservice* má každá mikroslužba svoji *routing* vrstvu. Na obrázku je zobrazena i architektura CNA neboli *Cloud Native Architecture*, která posunuje MSA ještě do propracovanější podoby.



Obrázek 5. Rozdíl mezi MSA, SOA a CNA [24]

1.4.4 Serverless architektura

Architektura *serverless* je označení pro aplikace, které využívají třetí strany k vytvoření business logiky. Tímto je odstraněna potřeba udržování databází či server. [25]

Příkladem takových aplikací jsou právě klienti konzumující REST API.

1.5 Architektonické vzory

Architektonické vzory určují způsob rozdělení povinností softwaru mezi jeho jednotlivé vrstvy. Hlavními vrstvami softwaru je vrstva prezentační, aplikační, doménová, perzistentní a datová.

Prezentační vrstva je to, co vidí uživatel při používání softwaru. Tato vrstva komunikuje s uživatelem.

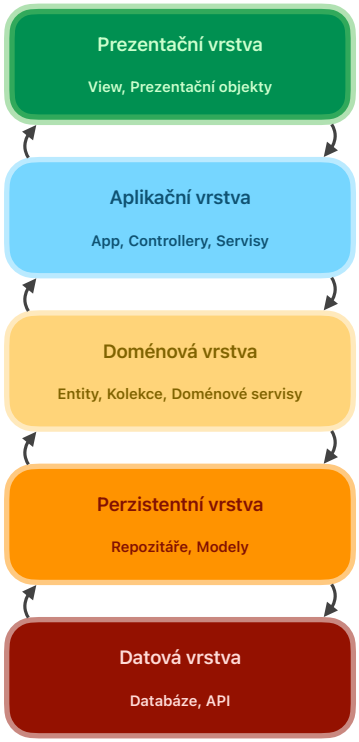
Aplikační vrstva má na starosti hlavní program architektury.

Doménová vrstva neboli business vrstva obsahuje business logiku. Business logika je sada pravidel, která říká systému, jak se chovat v závislosti na organizačních směrnicích. Tato vrstva určuje chování aplikace.

Perzistentní vrstva funguje jako protekce dat. Obsahuje kód potřebný pro přístup do databázové vrstvy jako je připojení nebo SQL příkazy.

Datová vrstva obsahuje data uložená systémem. Tato vrstva je nejnižší a naprosto skrytá. [26]

Tyto vrstvy graficky znázorňuje Obrázek 6.



Obrázek 6. Vrstvy softwaru [vlastní zpracování]

1.5.1 *Model-View-Controller* MVC

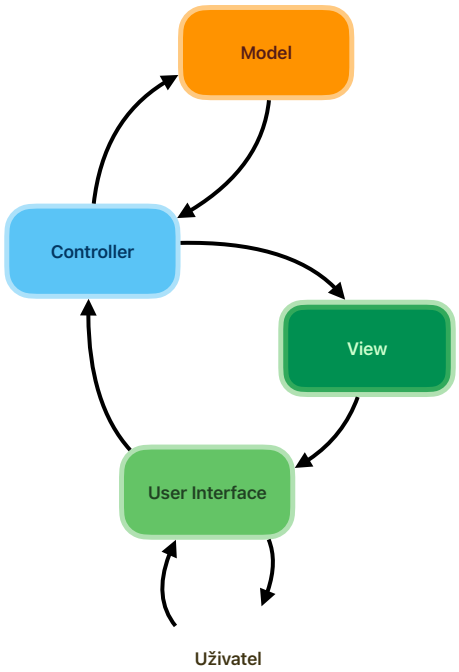
Architektonický vzor *Model-View-Controller*, zkráceně MVC, je nejpoužívanější architektonický vzor backend frameworků a je často využíván i frontend vývojáři. Z toho důvodu se jeho definice pro front-end a backend v detailech mírně odlišuje, nicméně princip zůstává stejný

Model je část architektury, která spravuje surová získaná data. Nachází se v perzistentní vrstvě.

View slouží ke komunikaci s uživatelem. Definuje, co uživatel vidí a jak může komunikovat s aplikací. Jedná se tedy o vrstvu prezenční.

Controller spojuje model a view. Controller obdrží uživatelův vstup a rozhoduje, co se s ním stane. [27]

Model MVC je graficky znázorněn na následující straně.



Obrázek 7. Architektonický vzor *model-view-controller* [vlastní zpracování]

Základním doporučením je, že v rámci jedné zodpovědnosti by controller měl pouze delegovat, co se stane v rámci požadavku, je důležité udržovat controller, co nejužší.

Z důvodu již zmíněného rozdělení zájmů (kapitola 1.1.6.5) a čistého kódu je vhodné rozšířit tuto architekturu ještě o část doménovou. V tuto chvíli model pouze definuje datový model, tedy jak vypadají uložené objekty a manipulaci s daty přebírá servisní vrstva. Servisní vrstva tedy funguje jako business vrstva, perzistentní vrstva obsahuje čistě práci s databází a o aplikační vrstvu se postará controller. Tato architektura se často označuje jako MVCS neboli *Model-View-Controller-Service*.

1.5.2 *Model-View-Presenter* MVP

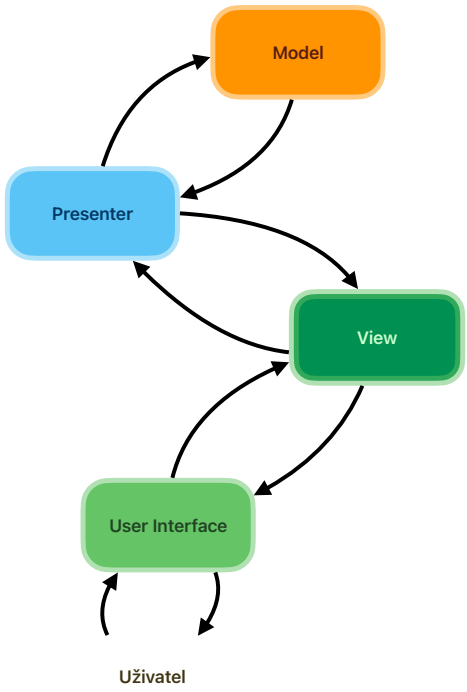
Architektonický vzor *Model-View-Presenter*, zkráceně MVP se svou podstatou hodí spíše na projekty, které definují uživatelské rozhraní.

Model stejně jako u MVC slouží k manipulací s daty.

View slouží k představení dat a reagování na uživatelské akce.

Presenter funguje stejně jako controller jako prostřední vrstva mezi modelem a view. Na rozdíl od něj ale presenter získává data přímo z komunikace od view pomocí jeho metod. [28]

Obrázek 8 znázorňuje model pro tento architektonický vzor.



Obrázek 8. Architektonický vzor *model-view-presenter* [vlastní zpracování]

1.5.3 *Model-View-ViewModel* MVVM

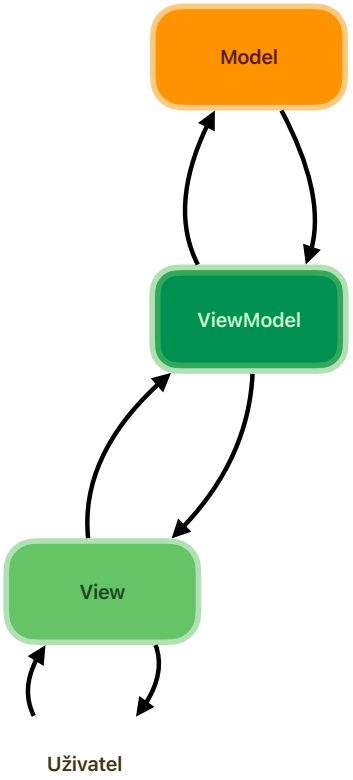
Architektonický vzor *Model-View-ViewModel*, zkráceně MVVM, byl vytvořen pod firmou Microsoft pro projekty v .NET. Na rozdíl od zmíněných vzorů je tento vzor řízen událostmi týkajících se přímo používaných modelů v daném view. View tedy v tomto případě naprosto nahrazuje uživatelské rozhraní.

Model stejně jako u předchozích architektur popisuje data.

View je u této architektury reprezentace rozhraní.

ViewModel drží stav aplikace, pomocí kterého View určuje svůj obsah. [29]

Grafické znázornění architektonického vzoru vyobrazuje Obrázek 9.



Obrázek 9. Architektonický vzor *model-view-viewmodel* [vlastní zpracování]

1.6 Návrhové vzory

Návrhové vzory definují ověřené nejlepší praktiky pro běžné problémy během objektově orientovaného vývoje. Popisují problém, řešení a jeho následky. [30]

Návrhové vzory se dělí na:

- vytvářející,
- strukturální,
- behaviorální.

Zástupci těchto vzorů budou definovány v následujících podkapitolách.

1.6.1 Návrhové vzory vytvářející

Vytvářející návrhové vzory (anglicky *creational design pattern*) jsou určeny k navržení inicializace objektu.

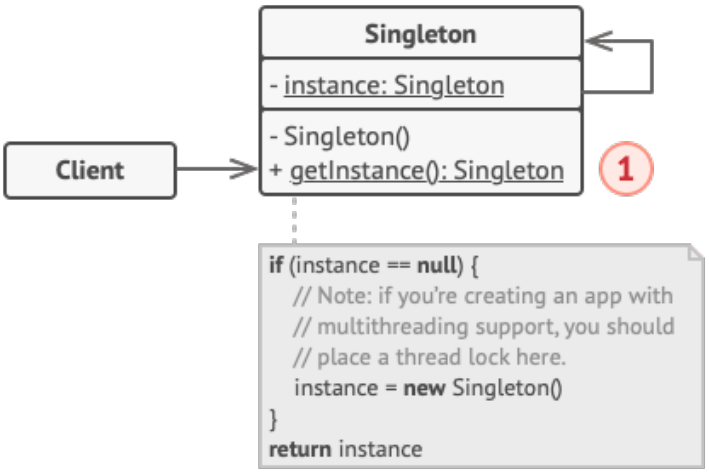
Mezi tyto vzory patří Singleton, Factory Method, Abstract Factory, Builder. [30]

1.6.1.1 Singleton

Návrhový vzor Singleton zajišťuje, že z určité třídy může existovat maximálně jedna instance. [30]

Singleton je typicky implementován pomocí privátní metody konstruktoru a veřejné statické metody getter singletonu, který je uložen v privátní statické finální proměnné.

Obrázek 10 zobrazuje model pro návrhový vzor Singleton.



Obrázek 10. Model návrhového vzoru Singleton [31]

Existují tři typy inicializace.

Eager inicializace vytváří instanci singletonu v době, když je inicializována proměnná. Příklad kódu v jazyce TypeScript definuje Zdrojový kód 1.

EagerSingleton.ts

```
public final class EagerSingleton
{

    private static readonly instance: EagerSingleton = new EagerSingleton();

    private constructor() {}

    public static getInstance(): EagerSingleton
    {
        return EagerSingleton.instance;
    }
}
```

Zdrojový kód 1. Koncept vzoru Singleton s *eager inicializací* [vlastní zpracování]

Lazy inicializace vytváří instanci singletonu až při jejím zavoláním, jak znázorňuje Zdrojový kód 2.

LazySingleton.ts

```
public final class LazySingleton
{

    private static readonly instance: LazySingleton;

    private constructor() {}

    public static getInstance(): LazySingleton
    {
        if (!LazySingleton.instance) {
            LazySingleton.instance = new LazySingleton();
        }
        return LazySingleton.instance;
    }
}
```

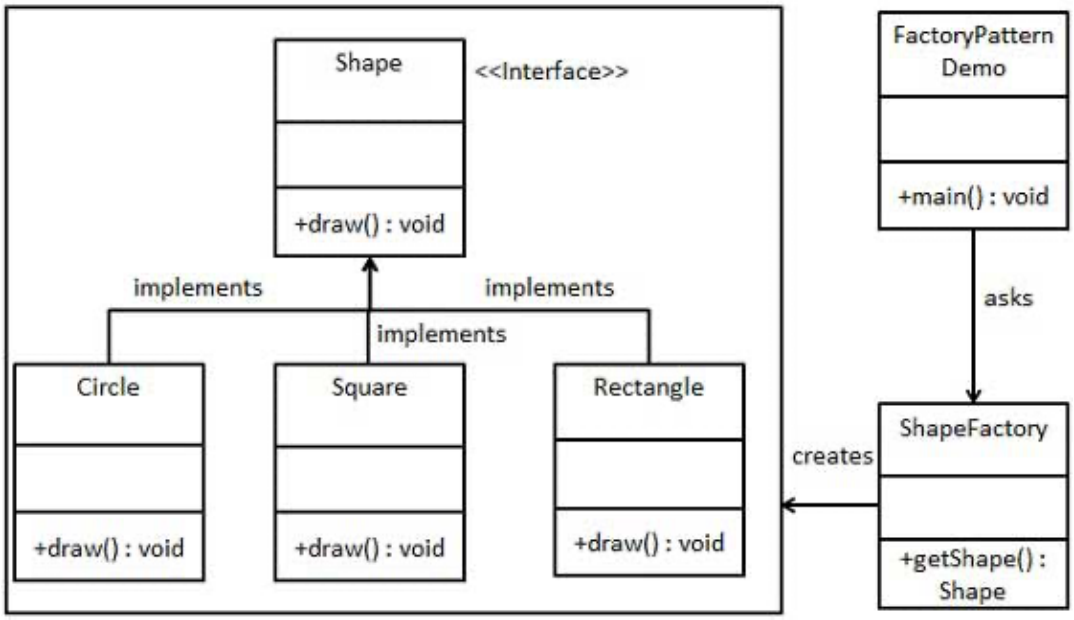
Zdrojový kód 2. Koncept vzoru Singleton s *lazy inicializací* [vlastní zpracování]

Existuje ještě poslední typ inicializace zvaný *thread-safe*, který využívá principu *lazy* inicializace ale s uzamčením singletonu během získání. [32]

Singleton se standardně využívá pro připojení k externím službám, jako jsou API či databáze.

1.6.1.2 Factory

Factory slouží k nepřímému vytvoření objektu a tím odstínění od dané logiky vytváření. Tento návrhový vzor poskytuje možnost získání různých objektů stejného původu. Existuje více možností, jak implementovat factory, jednou z nich, kterou uvádí Obrázek 11, kde je společným původem objektů rozhraní, které implementují. [30]



Obrázek 11. Model návrhového vzoru Factory [33]

V jiném případě se může jednat například o objekty stejné třídy s odlišnými parametry.

Zdrojový kód 3 znázorňuje ukázku kódu pro Factory, která vrací objekty se společným rozhraním.

Factory.php

```php
class Factory
{
    public function getObject($decidingParameter): SharedInterface
    {
        return match ($decidingParameter) {
            1 => new UniqueObject(),
            2 => new DifferentObject(),
            default => new OtherObject(),
        };
    }
}
```

Zdrojový kód 3. Návrhový vzor Factory [vlastní zpracování]

1.6.2 Návrhové vzory strukturální

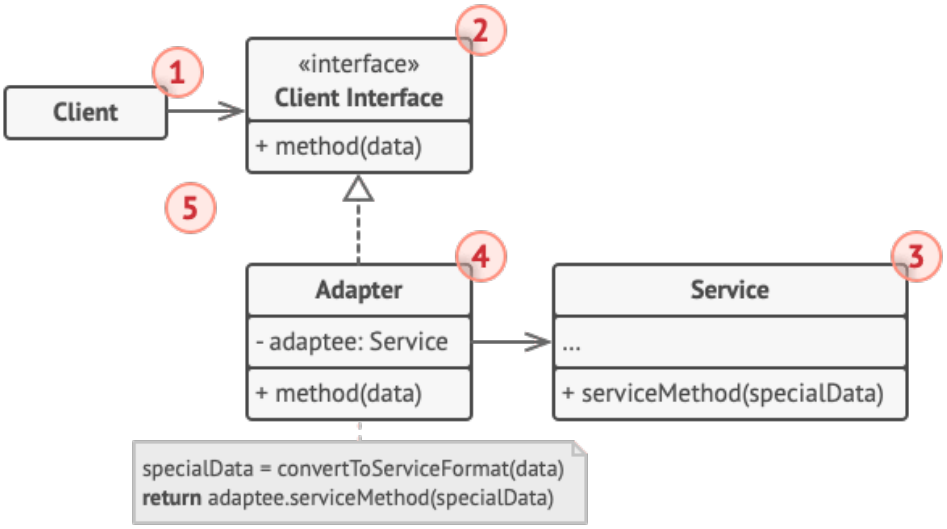
Strukturální návrhové vzory (anglicky *structural design patterns*) slouží k sestavení tříd do větších struktur k udržení jejich flexibility a efektivity.

Do této skupiny se řadí vzory Adapter, Bridge, Composite, Decorator, Facade, Flyweight a Proxy. [30]

1.6.2.1 Adapter

Návrhový vzor Adapter přizpůsobí rozhraní určité třídy rozhraní požadovanému klientem a umožní tak spolupráci tříd, které by kvůli nekompatibilním rozhraním nebyly schopné spolupracovat. [30]

Adapter je tedy třída, která umožňuje propojit funkcionalitu dvou nezávislých rozhraní díky použití kompozice propracovanějšího a adaptaci na jednodušší. Jeho model zobrazuje Obrázek 12.



Obrázek 12. Model návrhového vzoru Adapter [34]

Zdrojový kód 4, Zdrojový kód 5 a Zdrojový kód 6 pak znázorňují koncept jeho implementace.

OriginalInterface.php

```php
interface OriginalInterface
{
    public function originalMethod(): OriginalOutput;
}
```

Zdrojový kód 4. Koncept původního rozhraní vzoru Adapter [vlastní zpracování]

AdapteeInterface.php

```php
interface AdapteeInterface
{
    public function adapteeMethod(
        AdapteeParameter $adapteeParameter
    ): AdapteeOutput;
}
```

Zdrojový kód 5. Koncept adaptovaného rozhraní vzoru Adapter [vlastní zpracování]

Adapter.php

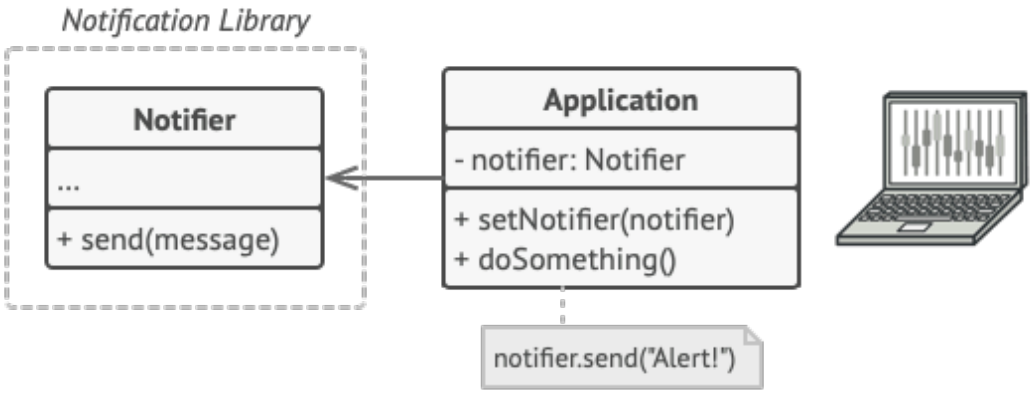
```php
class Adapter implements OriginalInterface
{
    private AdapteeInterface $adaptee;

    protected ?AdapteeParameter $adapteeParameter = null;

    public function __construct(AdapteeInterface $adaptee)
    {
        $this->adaptee = $adaptee;
    }

    public function originalMethod(): OriginalOutput
    {
         $adapteeOutput = $this->adaptee->adapteeMethod(
             $this->adapteeParameter
         );
         return $this->translateAdapteeOutputToOriginal(
             $adapteeOutput
         );
    }

    protected function translateAdapteeOutputToOriginal(
         AdapteeOutput $adapteeOutput
    ): OriginalOutput {
        return new OriginalOutput();
    }

    protected function setAdapteeParameter(
         AdapteeParameter $adapteeParameter
    ): Adapter {
        $this->adapteeParameter = $adapteeParameter;
        return $this;
    }
}
```

Zdrojový kód 6. Koncept třídy Adapter vzoru Adapter [vlastní zpracování]

1.6.2.2 Decorator

Návrhový vzor Decorator rozšiřuje daný objekt o novou funkčnost nebo změní stávající metody za běhu programu. Tento návrhový vzor nabízí flexibilní alternativu k vytváření

potomků tříd. [30] Na následujícím obrázku je model tohoto návrhového vzoru a následují ukázky obecné implementace.



Obrázek 13. Model návrhového vzoru Decorator [35]

ComponentInterface.php

```php
interface ComponentInterface
{
    public function baseExecute();
}
```

Zdrojový kód 7. Koncept rozhraní komponenty vzoru Decorator [vlastní zpracování]

ConcreteComponent.php

```php
class ConcreteComponent implements ComponentInterface
{
    public function baseExecute()
    {
        // TODO: Implement baseExecute() method.
    }
}
```

Zdrojový kód 8. Koncept implementace rozhraní komponenty vzoru Decorator [vlastní zpracování]

ComponentDecorator.php

```php
class ComponentDecorator implements ComponentInterface
{
    protected ComponentInterface $wrappee;

    public function __construct(ComponentInterface $component)
    {
        $this->wrappee = $component;
    }

    public function baseExecute()
    {
        return $this->wrappee->baseExecute();
    }
}
```

Zdrojový kód 9. Koncept rozhraní dekorátoru vzoru Decorator [vlastní zpracování]

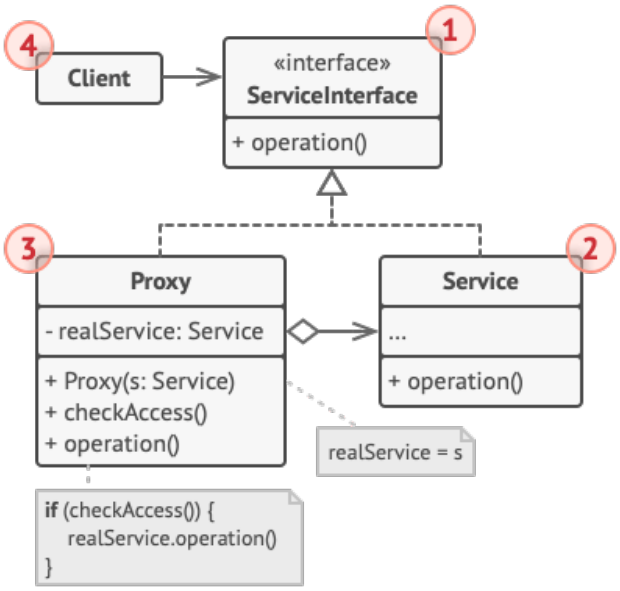
ConcreteDecorator.php

```php
class ConcreteDecorator extends ComponentDecorator
{
    public function baseExecute()
    {
        $this->decoratedPreviousExecute();
        parent::baseExecute();
        $this->decoratedFollowingExecute();
    }

    public function decoratedPreviousExecute()
    {

    }

    public function decoratedFollowingExecute()
    {

    }
}
```

Zdrojový kód 10. Koncept implementace rozhraní dekorátoru vzoru Decorator [vlastní zpracování]

1.6.2.3 Proxy

Návrhový vzor Proxy kontroluje přístup k objektu pomocí zástupce, který se používá místo vlastního objektu. [30] Model návrhového vzoru Proxy je vyobrazen na následující straně. Zdrojový kód 11 a Zdrojový kód 12 definuje vlastní objekt, zatímco Zdrojový kód 13 zástupný objet.



Obrázek 14. Model návrhového vzoru Proxy [36]

ServiceInterface.php

```php
interface ServiceInterface
{
    public function baseOperation();
}
```

Zdrojový kód 11. Koncept rozhraní služby vzoru Proxy [vlastní zpracování]

Service.php

```php
class Service implements ServiceInterface
{
    public function baseOperation()
    {

    }
}
```

Zdrojový kód 12. Koncept implementace rozhraní služby vzoru Proxy [vlastní zpracování]

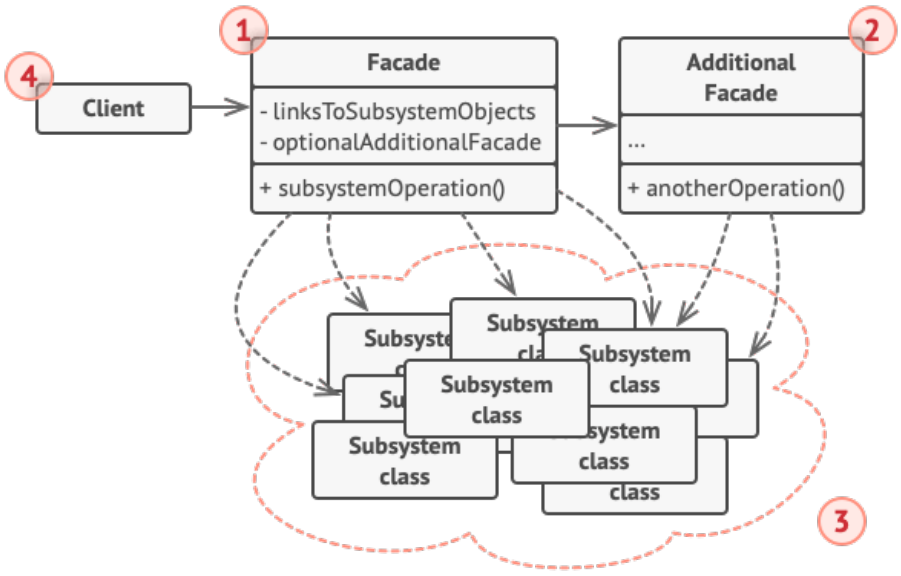
Proxy.php

```php
class Proxy implements ServiceInterface
{
    private Service $service;

    public function __construct(ServiceInterface $service)
    {
        $this->service = $service;
    }

    public function baseOperation()
    {
        // TODO: Implement baseOperation() method.
    }
}
```

Zdrojový kód 13. Koncept implementace třídy Proxy vzoru Proxy [vlastní zpracování]

1.6.2.4 Facade

Návrhový vzor Facade poskytuje zjednodušené rozhraní pro sérii rozhraní. Nabízí rozhraní, které zjednodušuje používání základního systému. [30] Model Facade je znázorněn níže.



Obrázek 15. Model návrhového vzoru Facade [37]

Kód návrhového vzoru obecně definuje Zdrojový kód 14.

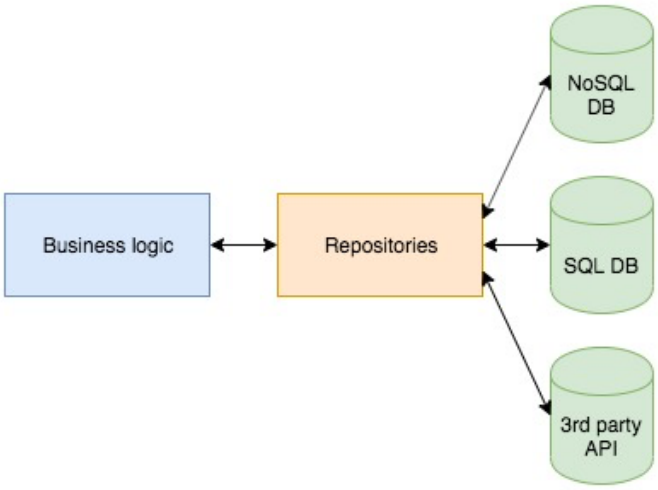
Facade.php

```php
class Facade
{
    protected Subsystem1 $subsystem1;

    protected Subsystem2 $subsystem2;

    protected Subsystem3 $subsystem3;

    protected Subsystem4 $subsystem4;


    public function subsystemOperation()
    {
        $this->subsystem1->method();
        $this->subsystem2->method();
        $this->subsystem3->method();
        $this->subsystem4->method();
    }
}
```

Zdrojový kód 14. Koncept implementace vzoru Facade [vlastní zpracování]

1.6.2.5 Repository

Často používaným vzorem je i vzor Repository, který se ale mnohdy nezařazuje do rozdělení, ale definuje se jako samostatný vzor. Repository slouží k abstrakci dat perzistentní vrstvy, aby oddělil doménovou vrstvu od té datové (znázorňuje Obrázek 16). Hlavní výhodou tedy je, že doménová vrstva neurčuje druh datové vrstvy a pro všechny různé druhy se chová stejně. Nezáleží tedy, zdali využijeme MySQL, MSSQL nebo API, díky tomuto vzoru se jejich použití pro doménovou vrstvu sjednocuje.



Obrázek 16. Model návrhového vzoru Repository [38]

Následující zdrojové kódy představují definici rozhraní návrhového vzoru (Zdrojový kód 15) a dva příklady jeho implementace (Zdrojový kód 16 a Zdrojový kód 17).

RepositoryInterface.php

```php
interface RepositoryInterface
{
    public function list();

    public function get(string $id);

    public function update(DataAccessObject $object);

    public function delete(DataAccessObject $object);
}
```

Zdrojový kód 15. Koncept rozhraní návrhového vzoru Repository [vlastní zpracování]

ApiInterface.php

```php
class ApiRepository implements RepositoryInterface
{
    public function list()
    {
        // call API to return list of the object
        // return result mapped to business model
    }

    public function get(string $id)
    {
        // call API to return detail of the object
        // return result mapped to business model
    }

    public function update(DataAccessObject $object)
    {
        // call API to update the object
        // return result mapped to business model
    }

    public function delete(DataAccessObject $object)
    {
        // call API to delete the object
        // return result mapped to business model
    }
}
```

Zdrojový kód 16. Koncept implementace návrhového vzoru Repository pro API [vlastní zpracování]

```php
class MSSQLRepository implements RepositoryInterface
{
    public function list()
    {
        // use MSSQL query to return list of the object
        // return result mapped to business model
    }

    public function get(string $id)
    {
        // use MSSQL query to return detail of the object
        // return result mapped to business model
    }

    public function update(DataAccessObject $object)
    {
        // use MSSQL query to update the object
        // return result mapped to business model
    }

    public function delete(DataAccessObject $object)
    {
        // use MSSQL query to delete the object
        // return result mapped to business model
    }
}
```

Zdrojový kód 17. Koncept implementace návrhového vzoru Repository pro MSSQL databázi [vlastní zpracování]

1.6.3 Návrhové vzory chování

Návrhové vzory chování (anglicky *behavioral design patterns*) pracují s interakcí mezi objekty.

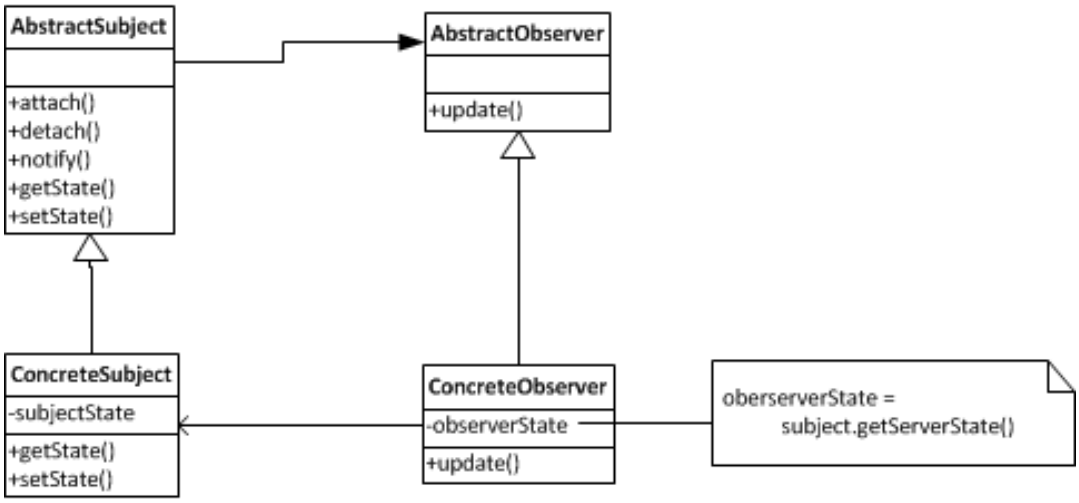
Příkladem jsou Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor. [30]

1.6.3.1 Observer

Návrhový vzor Observer definuje závislost 1:n mezi subjektem a libovolným počtem pozorovatelů. Při změně stavu subjektu jsou o této skutečnosti automaticky informovány všechny pozorovatele. [30]

Model návrhového vzoru je vyobrazuje Obrázek 17 na následující straně. Pro použití tohoto návrhového vzoru na `Subject` (Zdrojový kód 19) je potřeba implementovat třídu `Observer`

(Zdrojový kód 20), který je potomkem abstraktní třídy `AbstractObserver` (Zdrojový kód 18).



Obrázek 17. Model návrhového vzoru Observer [39]

AbstractObserver.php

```php
abstract class AbstractObserver
{
    protected Subject $subject;

    public abstract function update();
}
```

Zdrojový kód 18. Koncept abstraktní třídy Observeru [vlastní zpracování]

Subject.php

```php
class Subject
{
    private array $observers = [];

    public function attachObserver(AbstractObserver $observer)
    {
        $this->observers[] = $observer;
    }

    public function notifyObservers()
    {
        foreach ($this->observers as $observer)
        {
            $observer->update();
        }
    }
}
```

Zdrojový kód 19. Koncept předmětu vzoru Observer [vlastní zpracování]

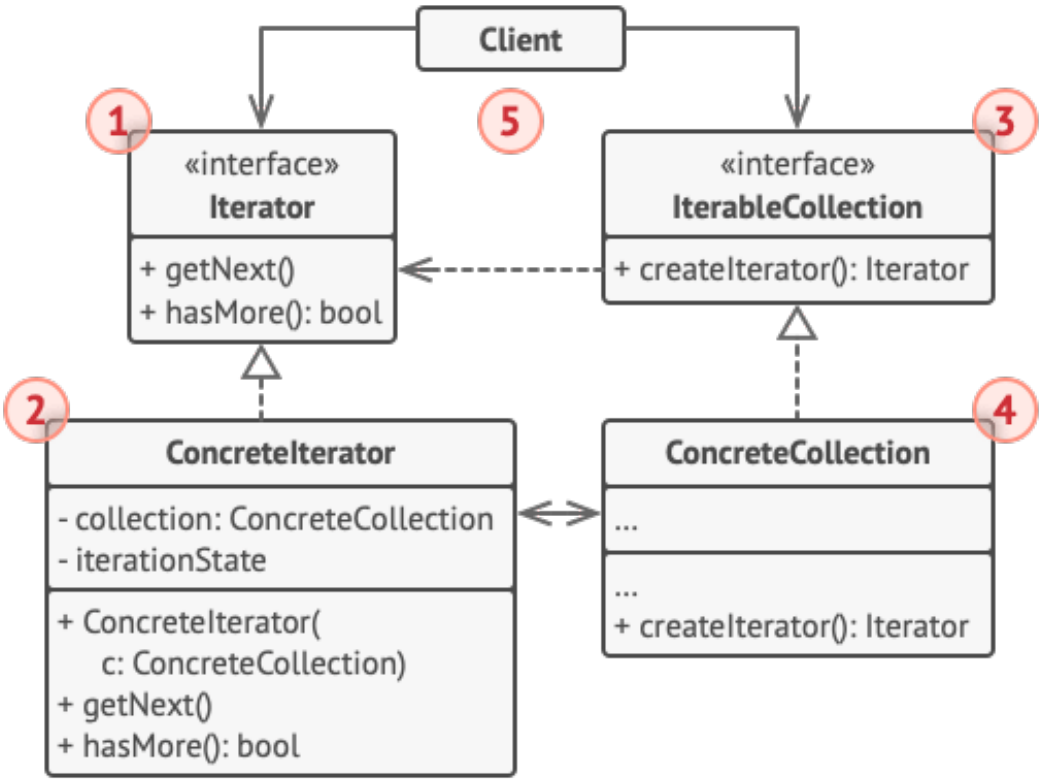
Observer.php

```php
class Observer extends AbstractObserver
{
    public function __construct(Subject $subject)
    {
        $this->subject = $subject;
        $this->subject->attachObserver($this);
    }

    public function update()
    {
        // TODO: Implement observe() method.
    }
}
```

Zdrojový kód 20. Koncept třídy Observer vzoru Observer [vlastní zpracování]

1.6.3.2 Iterator

Návrhový vzor Iterator, jehož model představuje Obrázek 18, umožňuje sekvenčně přistupovat k prvkům složených objektů bez zveřejnění jejich základní struktury. [30]



Obrázek 18. Model návrhového vzoru Iterator [40]

Zde dochází k použití rozhraní pro Iterator (Zdrojový kód 21) a kolekci (Zdrojový kód 22).

IteratorInterface.php

```php
interface IteratorInterface
{
    public function hasNext(): bool;

    public function next(): Item;
}
```

Zdrojový kód 21. Koncept rozhraní Iterator vzoru Iterator [vlastní zpracování]

CollectionInterface.php

```php
interface CollectionInterface
{
    public function createIterator(): IteratorInterface;
}
```

Zdrojový kód 22. Koncept rozhraní agregátoru vzoru Iterator [vlastní zpracování]

Následně jsou rozhraní implementována jako na následujících ukázkách.

CollectionIterator.php

```php
class CollectionIterator implements IteratorInterface
{
    private int $position = 0;

    private array $items = [];

    public function __construct(array $items)
    {
        $this->items = $items;
    }

    public function hasNext(): bool
    {
        if ($this->position < count($this->items)) {
            return true;
        }
        return false;
    }

    public function next(): Item
    {
        if (!$this->hasNext()) {
            $this->position = 0;
        }
        return $this->items[$this->position++];
    }
}
```

Zdrojový kód 23. Koncept implementace iterátoru vzoru Iterátor [vlastní zpracování]

CollectionIterator.php

```php
class Collection implements CollectionInterface
{
    private array $items = [];

    public function getItems()
    {
        return $this->items;
    }

    public function addItem($item)
    {
        $this->items[] = $item;
    }

    public function createIterator(): IteratorInterface
    {
        return new CollectionIterator($this->items);
    }

    public function getCount(): int
    {
        return count($this->items);
    }
}
```

Zdrojový kód 24. Koncept implementace agregátoru vzoru Iterator [vlastní zpracování]

Využití v kódu by potom mohlo vypadat jako v následující ukázce.

ItemCollectionExample.php

```php
$collection = new Collection();
$collection->addItem(new Item());
$iterator = $collection->createIterator();
while ($iterator->hasNext()) {
    $item = $iterator->next();
}
```

Zdrojový kód 25. Koncept použití vzoru Iterator [vlastní zpracování]

1.7 RESTful rozhraní pro programování aplikací

Rozhraní pro programování aplikací (anglicky *Application Programming Interface*, zkráceně API) definuje pravidla, která musí být dodržena v rámci komunikace s dalšími softwarovými systémy. [41] Výhodou implementování a zveřejnění API je nezávislost na platformě, která slouží k vytvoření uživatelského rozhraní.

Representational State Transfer neboli REST byla představena již v roce 2000 Royem Fieldingem jako architektonický přístup k návrhu webových služeb. [41]

1.7.1 Stavy API odpovědí

Při kvalitní implementaci REST API, je důležité, aby server vracel odpovědi se stavovým kódem, který odráží stav odpovědi na daný požadavek.

V případě úspěšného zpracování požadavku je klientovi vrácen třímístný kód začínající 2, jehož varianty jsou znázorňuje Tabulka 3. Jedná se o očekávaný stav odpovědi požadavku.

Tabulka 3. Úspěšné stavové kódy API odpovědí [vlastní zpracování]

Kód	Stav	Popis
200	*OK*	úspěšné vyhodnocení požadavku
201	*Created*	úspěšné vytvoření nového zdroje
202	*Accepted*	přijatý požadavek, který ještě nebyl zpracován
204	*No Content*	úspěšné provedení požadavku s prázdným tělem odpovědi
206	*Partial Content*	úspěšné vrácení požadované části požadavku

Pokud byl požadavek úspěšně zpracován, ale očekávaná odpověď se nachází na jiném místě, jsou použity třímístné kódy začínající na 3, jako ukazuje Tabulka 4.

Tabulka 4. Stavové kódy přesměrování API odpovědí [vlastní zpracování]

Kód	Stav	Popis
304	*Not Modified*	stejná odpověď je již v cache
303	*See Other*	požadovaná odpověď se na jiné lokaci

V případě chybových stavů jsou dvě možnosti, které rozlišují původ chyby.

V případě chyby na straně klienta začíná stavový kód číslem 4. Tyto chyby dávají najevo, že v případě úpravy požadavku lze získat požadovanou odpověď. Tabulka 5 na následující straně obsahuje stavové kódy klientských chyb typické pro API.

Tabulka 5. Kódy klientských chybových stavů API odpovědí [vlastní zpracování]

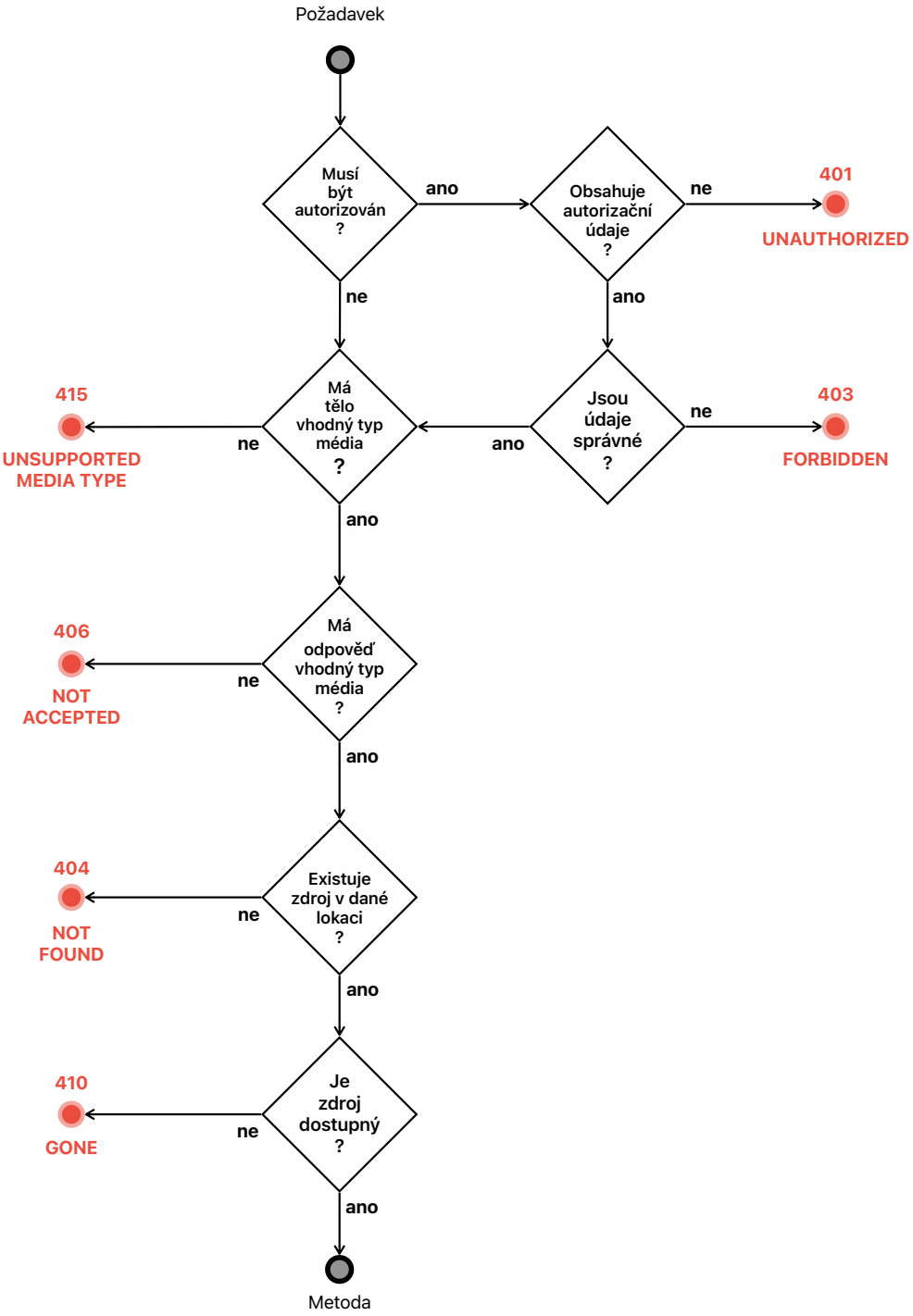
Kód	**Stav**	**Popis**
400	*Bad Request*	server nedokázal rozeznat, co klient požadoval
401	*Unauthorized*	chybí údaje k ověření přístupu
403	*Forbidden*	nesprávné údaje k ověření přístupu
404	*Not Found*	požadovaný zdroj není k dispozici
406	*Not Acceptable*	požadovaný formát odpovědi není k dispozici
409	*Conflict*	požadované úpravy nelze na současném stavu provést
410	*Gone*	požadovaný zdroj je záměrně nedostupný
415	*Unsupported Media Type*	server nerozumí formátu těla požadavku

V případě, že je chyba na straně serveru je vrácen stavový kód začínající 5. V tomto případě je jasné, že klient s chybou nemůže nic dělat a je třeba vyčkat na nápravu serveru. Typické stavové kódy pro API jsou znázorněny v tabulce níže.

Tabulka 6. Kódy chybových stavů serveru API odpovědí

Kód	**Stav**	**Popis**
500	*Internal Server Error*	server byl vyžádán provést nečekanou operaci
503	*Service Unavailable*	server není dostupný pro obdržení požadavků

Zatímco některé stavy jsou specifické pro určité HTTP metody (budou rozebrány v kapitole 1.7.2), jiné se vztahují pro všechny požadavky. Jak ke stavu dojít, bude uvedeno v diagramech, které znázorňují průběh kontroly požadavku. Následující diagram vyjadřuje počáteční zpracování požadavku a jeho případné chybové stavy, ke kterým lze dojít. V případě úspěšného průchodu touto logikou se postupuje dále v závislosti na požadované HTTP metodě.



Obrázek 19. Diagram stavů požadavku [vlastní zpracování]

1.7.2 HTTP Metody

HTTP definovalo sadu metod, které označují typ akce prováděné na zdroji. Pokud by URL bylo větou, pak zdroj je podmětem a HTTP metoda přísudkem.

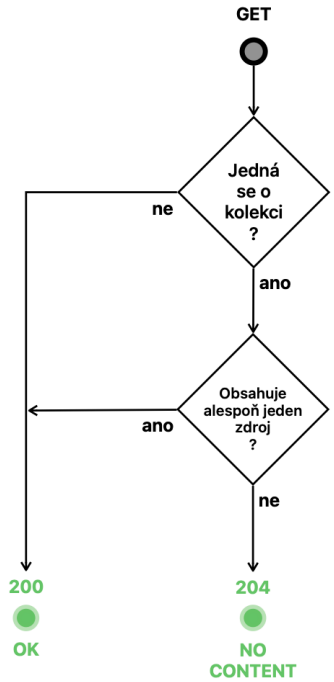
1.7.2.1 Metoda GET

Metoda GET slouží k získání reprezentace zdroje/zdrojů, které se nachází na uvedené URI adrese. Tento typ akce by neměl vyprodukovat žádný vedlejší účinek, tedy slouží pouze o výpis již existujících dat.

Úspěšná metoda typicky vrací stav 200 (*OK*). V jiných případech, kdy zdroj zcela na dané lokaci neexistuje, je to 404 (*Not Found*), nebo pokud byl smazán 410 (*Gone*).

Nastat může také případ, kdy se například požaduje kolekce s parametry filtrování nebo vyhledávání, pak v případě úspěšného zpracování požadavku, který ale nemá žádné vyhledávané výsledky, slouží stav 204 (*No Content*). [43]

Diagram stavů metody GET definuje Obrázek 20.



Obrázek 20. Diagram stavů odpovědí na GET požadavek [vlastní zpracování]

1.7.2.2 Metoda POST

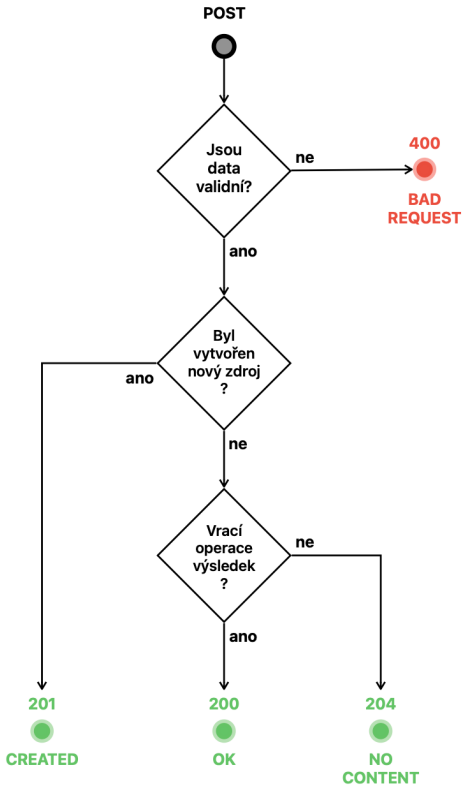
Metoda post žádá server o vytvoření nového zdroje. Označuje se jako *neidempotentní*, což znamená, že v případě duplikovaného požadavku má duplikovaný efekt.

Pokud POST metoda vytvoří nový zdroj, pak stav odpovídá 201 (*Created*), v hlavičce HTTP je umístěna URI lokace a tělo odpovědi obsahuje reprezentaci vytvořeného zdroje.

Pokud metoda pouze zpracovává požadovanou operaci nastávají dva případy. Jestliže operace zahrnuje výsledek v těle odpovědi, pak je vracen kód 200 (*OK*), v opačném případě vrátí 204 (*No Content*).

V případě, že tělo požadavku klienta obsahuje nevalidní data, server by měl vrátit 400 (*Bad Request*). Tělo odpovědi by mělo obsahovat podrobnější informace o důvodu zamítnutí, nebo odkaz na URI, která obsahuje více detailů. [43]

Metoda POST a její stavy odpovědí jsou znázorněny na obrázku níže.



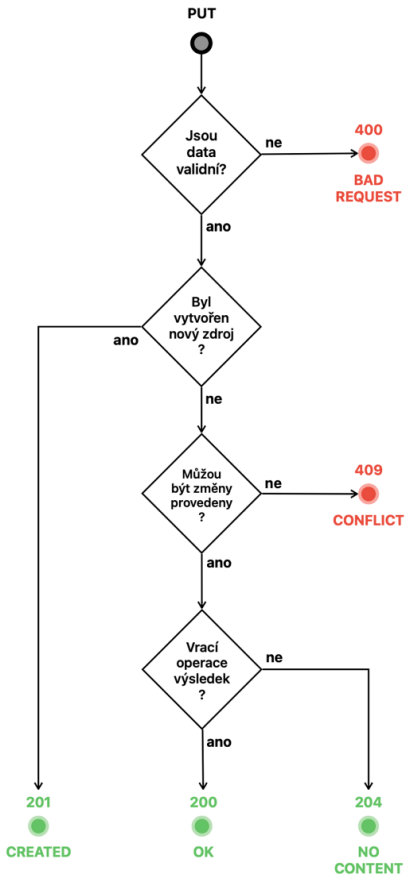
Obrázek 21. Diagram stavů odpovědí na POST požadavek [vlastní zpracování]

1.7.2.3 Metoda PUT

Metoda PUT požádá server o úpravu zdroje nebo jeho vytvoření v případě, že ještě neexistuje. Jedná se o idempotentní metodu čili při duplikaci požadavku je pouze jeden stejný efekt.

Pokud metoda PUT vytváří nový zdroj, vrací HTTP status 201 (*Created*) stejně jako při metodě POST.

Jestliže upravuje existující zdroj, vrací buď 200 (*OK*) nebo 204 (*No Content*). V některých případech může dojít k tomu, že zdroj nelze upravit v daném stavu. Potom odpověď vrátí 409 (*Conflict*), jak znázorňuje Obrázek 22.



Obrázek 22. Diagram stavů odpovědí na PUT požadavek [vlastní zpracování]

Pro snížení výřečnosti a zlepšení výkonnosti API je vhodné implementovat hromadné úpravy kolekcí. [43]

1.7.2.4 Metoda PATCH

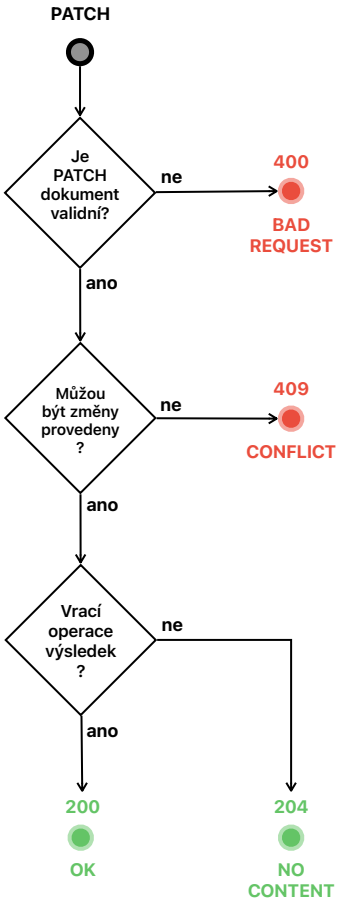
PATCH metoda slouží k částečné úpravě zdroje. Pomocí PATCH požadavku klient pošle sadu úprav k existujícímu zdroji ve formě patch dokumentu, který server zpracuje pro provedení úprav. Pro formát JSON existují dva typy PATCH dokumentu.

JSON Merge Patch je jednodušší dokument, který má stejnou strukturu jako originální zdroj, ale obsahuje pouze podmnožinu polí, které mají být upraveny nebo přidány. Pokud má být položka smazána, je hodnota pole nastavena na `null`. Z tohoto důvodu není vhodné tento formát používat pro zdroje, které přijímají `null` jako explicitní hodnotu položky. Dále také neuvádí pořadí, ve kterém by server měl změny aplikovat. Pro tento typ je nastavený media type na `application/merge-patch+json`.

JSON Patch je flexibilnějšího formátu. Tento formát byl navržen v roce 2013 v RFC 6902, který definuje konvence tohoto souboru. Specifikuje změny jako sekvenci operací, které mají být provedeny. Je schopný definovat operace přidání, odebrání, nahrazení, kopírování a validování. Takový soubor má nastaven media type na `application/json-patch+json`.

V případě, že typ patch dokumentu není podporován serverem, pak je vrácena odpověď 415 (*Unsupported Media Type*). Pokud je požadavek správného formátu, ale v patch dokumentu jsou chyby, pak se jedná o stav 400 (*Bad Request*). Poslední případ chyby je, pokud změny nemohou být aplikovány pro aktuální stav zdroje, pak jde o 409 (*Conflict*). [43]

Obrázek na následující straně znázorňuje diagram stavů metody PATCH.



Obrázek 23. Diagram stavů odpovědí na PATCH požadavek [vlastní zpracování]

1.7.2.5 Metoda DELETE

Pokud je operace smazání úspěšná, pak by měl server vrátit 204 (*No Content*) označující, že byl proces úspěšně vykonán, ale odpověď neobsahuje žádné jiné informace. Pokud zdroj neexistuje, pak by měl server vrátit 404 (*Not Found*). [43] Jednoduchý diagram je znázorněn níže.

Obrázek 24. Diagram stavů odpovědí na DELETE požadavek [vlastní zpracování]

1.7.3 Doporučení

Určení a dodržování konvencí je o to zásadnější u implementace API, kvůli jeho dalšímu použití vývojáři, a tedy určitému očekávanému chování.

1.7.3.1 Dodržení jmenných konvencí

Stejně jako v kódu je důležité udržet jmenné konvence i v případě API. Týká se to jak API objektů, tak i samotné URI adresy zdroje.

API objekt definuje zdroj, který je zaslán v odpovědi nebo požadavku. Pro tento objekt by měly být pevně nastavené konvence pojmenování klíčů, které by se neměly míchat. Následující tabulka definuje písemné tvary pro jednotlivé typy médií.

Tabulka 7. Písemné tvary klíčů pro jednotlivé typy médií [vlastní zpracování]

Typ média	Písemný tvar
`application/json`	`camelCase`
`application/xml`	`camelCase`, `snake_case`
`application/x-www-form-urlencoded`	`snake_case`

URI definuje lokaci zdroje na serveru. Pro URI je definován hlavní písemný tvar kebab-case.

Hlavní konvencí REST API je definování URI právě podle názvu zdroje. Lokace je určena názvem kolekce, a to jaká operaci je vykonaná na zdroji by měla určovat již zmíněná HTTP metoda.

1.7.3.2 Vyhnout se několika-úrovňové závislosti

V případě složitější struktury může vzniknout velká závislost objektů jako je například `/students/3/mobilities/20/courses`, či větší. Více úrovní ovlivňuje flexibilitu API, což vytváří problémy při změnách závislostí.

Doporučuje se používat maximálně dvouúrovňovou závislost, tedy `/resource/ID/resource`. V případě potřeby větších závislostí se lze odkazovat pomocí takzvaných HATEOAS. [44]

1.7.3.3 Umožnění filtrování v kolekcích

Pro optimalizaci výkonu je vhodné implementovat do kolekcí filtrování pomocí *query* parametrů. Pokud klientská aplikace požaduje pouze určitou podmnožinu dat a jsou vráceny všechny, zvyšuje se nárok časové zpracování, jelikož filtrování dat je prováděno až v rámci klientské aplikace. *Query* parametr je definován názvem filtrované položky a jeho hodnotou. V případě získáváních větších/menších hodnot než ve filtrovaném parametru, jako přípona k názvu *query* parametru se přidá `min` nebo `max` jak znázorňuje následující ukázka.

```HTTP
https://matchversity.utb.cz/courses?credits_min=4
```

Zdrojový kód 26. Příklad filtrování kolekce předmětů s minimálně 4 kredity [vlastní zpracování]

Stejně jako filtrování počtu vrácených zdrojů lze zavést i filtrování v rámci jednoho zdroje, tedy jednotlivých položek zdroje, zavedením *query* parametru `fields`, který přijímá názvy položek zdroje, které mají být vráceny v rámci odpovědi, podobně jako je tomu na ukázce níže.

```HTTP
https://matchversity.utb.cz/courses?fields=name,code
```

Zdrojový kód 27. Příklad filtrování jména a kódu předmětů [vlastní zpracování]

1.7.3.4 Umožnění stránkování velkých kolekcí

Jestliže je pravděpodobné, že kolekce nabude velkého počtu zdrojů, je nezbytné implementovat stránkování, které zamezí zdlouhavým a velkým odpovědím. Stránkování je definováno *query* parametry `limit`, který definuje počet zdrojů na jednu stránku a `offset`, který určuje číslo stránky. Příklad takové URI ukazuje Zdrojový kód 28.

```HTTP
https://matchversity.utb.cz/courses?limit=20&offset=2
```

Zdrojový kód 28. Příklad získání druhé stránky předmětů o 20 zdrojích [vlastní zpracování]

1.7.3.5 Umožnění řazení kolekcí

Řazení kolekcí stejně jako filtrování umožňuje optimalizaci klientských aplikací, jelikož nemusí provádět tyto operace u sebe. Řazení je definováno taktéž *query* parametrem, kdy parametr `sort` obsahuje hodnotu `asc` nebo `desc`, která určuje, kterým směrem je řazení prováděno a pak druhým *query* parametrem `sort_by`, který určuje položku, pomocí které jsou data seřazena. Ukázku znázorňuje Zdrojový kód 29.

```HTTP
https://matchversity.utb.cz/courses?sort=asc&sort_by=code
```

Zdrojový kód 29. Příklad získání kurzů seřazených sestupně podle kódu [vlastní zpracování]

Zmíněný způsob řazení ovšem nepodporuje mnohonásobné řazení s různým směrem, proto je možné uvést v *query* pouze jeden parametr sort_by, který bude přijímat pole položek určujících řazení se sufixem uvádějící směr, jak znázorňuje ukázka níže.

```HTTP
https://matchversity.utb.cz/courses?sort_by[]=code_asc&
sort_by[]=updatedAt_desc
```

Zdrojový kód 30. Ukázka vícenásobného řazení [vlastní zpracování]

1.7.3.6 Umožnění vyhledávání v kolekcích

Vyhledávání v kolekcích slouží k filtraci zdrojů podle části jejich textových hodnot. Implementace vyhledávání na *backend* opět uleví klientské aplikaci a dojde k optimalizaci výkonu. Příklad vyhledávání je uveden níže.

```HTTP
https://matchversity.utb.cz/courses?search=programming
```

Zdrojový kód 31. Příklad získání kurzů obsahujících slovo „programming“ [vlastní zpracování]

1.7.3.7 Umožnění rozdělení odpovědí s binárními daty

V případě příliš velkých dat v odpovědi může docházet ke zpoždění očekávané odpovědi a špatnému výkonu. Z tohoto důvodu se umožňuje rozdělit odpověď do více částí. Příkladem vhodného použití jsou odpovědi s obrázky či jinými soubory. [45]

1.7.3.8 Umožnění asynchronní odpovědi

V případě, že metody upravující stav systému (POST, PUT, PATCH a DELETE) mohou zabrat více času, pro optimalizaci výkonu klientských aplikací je vhodné uvažovat o asynchronních požadavcích. [46]

1.7.3.9 Verzování API

Jelikož na API jsou závislé klientské aplikace, je důležité udržet původní styl odpovědí, aby změna koncového bodu nenarušila chod aplikací. Kód klientských aplikací je totiž závislý na určitém tvaru odpovědi a jeho změna znamená, že aplikace neví, jak se chovat. V případě, že se API mění (např. změna business požadavků nebo vylepšení), tedy se mění struktura zdrojů či jejich závislosti, aby neměly změny dopad na stávající klientské aplikace, přidává se do API verzování. [47]

Verze je určená číslem označující pořadí změny zdroje.

Verzování API je možné provést několika způsoby:

- verzování URI – označení verze je umístěno v lokaci zdroje,
- verzování parametrem – označení verze nese *query* parametr lokace,
- verzování hlavičky – přidání vlastní HTTP hlavičky označující verzi,
- verzování v typu média – za typ média v akceptující hlavičce se přidá označení verze.

1.7.3.10 Dokumentování API

Dokumentace je velmi důležitým aspektem při vývoji API, která by neměla být přehlížena. Jedná se o přehledný nástroj pro klienty, jak API používat. V dokumentaci by měly být obsaženy:

- informace o záměru API,
- použité metody autentizace,
- přehled jednotlivých koncových bodů,
- příklady odpovědí na požadavky,

- očekávatelné statusy a chybové odpovědi,
- možné parametry koncových bodů.

1.8 Práce s verzovacím systémem

Ať už při samostatné práci nebo v týmu, je při vývoji softwaru zvykem používat verzovací nástroj. Ve většině případů však dochází k tomu, že je nástroj používán spíše z povinnosti a není využíván efektivně. Při správném použití však dokáže usnadnit práci a eliminovat nedostatky softwaru.

V následujících kapitolách budou rozvedeny doporučení pro každodenní zacházení s Gitem.

1.8.1 Branch

V případě použití Gitu na projekt je inicializován samotný repozitář pro verzování. To je zaznamenáno do počáteční *branch*. *Branch* je označení pro vývojovou větev, ve které dochází k úpravám kódu (přidávání a mazání).

Pro hlavní *branch* se v dnešní době využívá název `main`. Od názvu `master` se kvůli nevhodnosti pojmenování a kulturní citlivosti od roku 2020 opouští. [48]

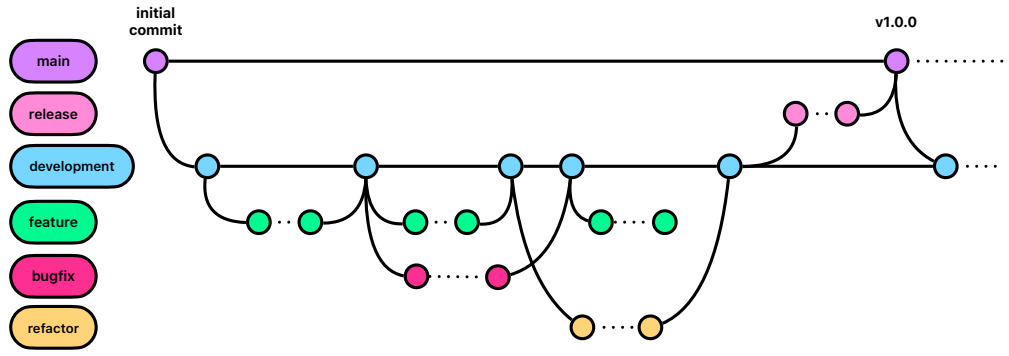
V rámci vývoje se nedoporučuje používat pouze hlavní *branch*. *Branch* se zakládá vždy při implementaci nové funkcionality.

Pojmenování pro *branch* hraje důležitou roli pro rychlou orientaci a čitelnost verzování. Při implementaci v týmu dochází k číslování jednotlivých úkolů, které bývají v názvech využity spolu se jménem autora. V případě toho, že autor pracuje sám na svém projektu, je název popsán slovně podle typu úprav. Pro pojmenování se nejčastěji využívá `kebab-case`, který zmiňovala Tabulka 2. Typy písemných tvarů. Dále je vhodné použít prefix, který vystihuje druh implementace. Uvedené typy *branch* pojmenování jsou uvedeny na následující straně.

Tabulka 8. Jmenné konvence branchí [vlastní zpracování]

Název branche	**Příklad**	**Použití**
`main`	-	Stabilní verze projektu
`development`	-	Probíhající vývoj projektu
`feature/<name>`	`feature/university-list`	Přidání nové funkcionality
`bugfix/<name>`	`bugfix/wrong-results`	Oprava chybné funkcionality
`refactor/<name>`	`refactor/update-library`	Úprava existujícího kódu
`release/<version>`	`release/1.0.0`	Příprava vydání nové verze

Obrázek níže znázorňuje *branching model* podle výše zmíněné tabulky.



Obrázek 25. *Branching model* [vlastní zpracování]

Každý bod znázorňuje *commit* a jeho barva typ pro *branch*. Model vychází z *branch* `main`, ze které je vytvořena *branch* `development`. Z ní následně vychází *branch* `feature`, `bugfix` a `refactor`, které slouží k přidávání *commit* kódu vývojářem. Jakmile je proveden poslední vývojový *commit*, vytvoří vývojář *pull-request*, který je následně schválen a za pomocí *merge* je přiveden zpět do `development` *branch*. V případě vydání verze je vytvořena `release` *branch* z `development`, u které je proveden *merge* do `main` *branch* a tam je vytvořen *tag* pro poslední *commit* indikující číslo vydané verze.

1.8.2 Commit

Základním prvkem verzovacího systému je *commit*. *Commit* slouží k zaznamenání změn v konkrétní *branch*. Jedná se právě o nejproblematičtější část, co se týče konzistence a čistoty.

Jednotlivý *commit* by měl obsahovat pouze **malé části kódu**. To umožní lepší přehled o provedených změnách. Je doporučeno, aby jeden *commit* obsahoval v řádech desítek, výjimečně stovek změn (přidání/smazání).

Committing by měl být dostatečně **častý**, aby se právě předešlo velkým a nepřehledným změnám.

Dále by změna měla být **kompletní**, tedy by měla obsahovat celou část funkčního kódu. To z toho důvodu, že každý *commit* by měl být **vratný**. V případě tedy, že je potřeba část implementované funkce vrátit, mělo by být možné provést pouze takzvaný *revert commit*, nikoliv přidávat další *commity* k odstranění části kódu.

Kompletní funkcionalita by měla být také **otestovaná**. Toto je totiž zásadní vlastnost pro *committing* a to, že: **Žádný *commit* by neměl rozbít současný stav v *branch*. Neboli každý *commit* by měl přispět malou kompletní a plně funkční funkcionalitou.**

Velmi důležitou a často opomíjenou částí je i samotná *commit* zpráva. Ta je podstatnou částí dobré čitelnosti verzovacího systému repozitáře. Zpráva by měla být složena následovně:

```
<Identifikátor úkolu> [<Typ>] <Samotná zpráva>
```

Zdrojový kód 32. Struktura *commit* zprávy [vlastní zpracování]

Samotné zprávě může předcházet volitelný identifikátor úkolu, který pomáhá přiřadit *commit* k úkolu v softwarech pro projektový management.

Dále se pro lepší čitelnost hodí přidat i typ, díky kterému jde na první pohled odlišit význam změny. Možné typy pro *commit* jsou znázorňuje Tabulka 9.

Tabulka 9. Typy commitů [vlastní zpracování]

Typ commitu	**Použití**
`Feature`	Přidání nového funkčního kódu
`Fix`	Oprava nefunkčního kódu
`Test`	Přidání nového testovacího kódu
`Refactor`	Vylepšení kódu
`Release`	Příprava na vydání kódu
`Documentation`	Přidání dokumentace
`Text`	Změna textace nebo překladu
`Design`	Přidání nového designového kódu
`Revert`	Vrácení commitu

Pro zprávu (včetně předpony) je důležité již na začátku definovat konvence a v průběhu práce na repozitáři tyto konvence dodržovat.

Zpráva by měla dostatečně vystihovat provedené úpravy v tomto commitu. Zpráva by měla být v imperativním tvaru, aby seděla do věty: „*If applied, this commit will* `<Samotná zpráva commitu>`.”

1.8.3 Pull request

Po naimplementování požadované funkcionality a dokončení práce na současné *branch,* a hlavně po otestování změn přichází na řadu *pull request*. Pokud se nejedná o vydání nové verze, míří *pull request* do *branch* určené pro vývoj.

Pull request slouží pro kontrolu nově přidaných změn vývojáři. Jakmile je *pull request* vytvořen, vývojář své změny sám zkontroluje a doladí detaily. V případě týmového vývoje je *pull request* zkontrolován alespoň jedním dalším vývojářem (tzv. *code review*).

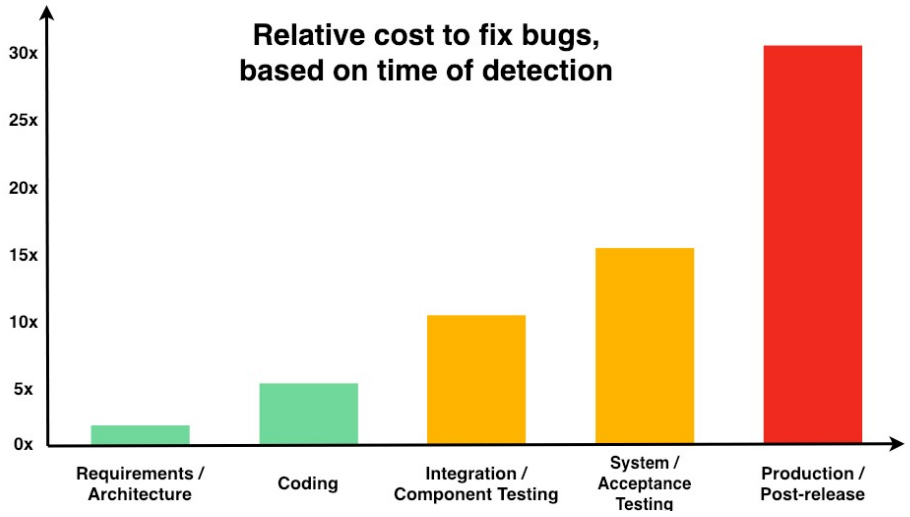
Většina webových služeb pro Git v dnešní době poskytuje podporu pro přidání nástrojů, které usnadňují *code review*, a je výhodné této příležitosti využít. O nástrojích, které pomáhají kontrolovat kód mluví 1.9.1 Statická analýza. Je třeba ale zdůraznit, že nástroje by měly

být používány již během vývoje samotným vývojářem a při *pull request* kontrole sloužit jen jako pojistka pro ostatní vývojáře, že byla analýza provedena. V opačném případě je *pull request* zahrnut komentáři a dalšími navazujícími opravami, které znepříjemňují *code review* a vytváří nečitelnost jak v *pull request*, tak v celém verzovacím systému projektu.

1.9 Ověření kódu

Při vývoji softwaru je nutností, aby implementace byla kontrolována nejen testerem, ale i samotným programátorem. Tato nutnost je bohužel ale pro řadu vývojářů i firem ještě neznámou záležitostí a vydávaný kód není testován. Zavedení testů totiž vede ke zvýšení nákladů na práci vývojáře a k větší spotřebě času na implementaci.

Při změně úhlu pohledu ovšem vidíme naprosto odlišné dopady. Netestovaný kód z pravidla obsahuje vysoké množství neodhalených chyb, které jen čekají na správný okamžik. V nejlepším případě tato chyba může být odhalena již vývojářem při zkoušce kódu, nebo při implementaci dalších funkcionalit navazující na chybnou implementaci. Podstatně nákladnější variantou je, že se na chybu narazí v *code review*, kde se sčítá čas kontrolujícího a implementujícího vývojáře. Následující obrázek ukazuje dopady chyb na cenu vývoje.



Obrázek 26. Relativní cena opravy chyb vzhledem k času odhalení [49]

1.9.1 Statická analýza

Pomocí statické analýzy je zkoumán samotný kód bez jeho spuštění pro nalezení chyb. Statická analýza je spouštěna před softwarovým testováním. Jedná se o odhalování nedostatků

za pomocí určených pravidel bez znalosti záměru kódu. Dynamické testování se často není schopné dostat ke všem nedostatkům, které snadno odhalí statická analýza. [52]

Jako statickou analýzu lze považovat samotné *code review*, které funguje jako manuální statická analýza. Jak již bylo zmíněno v kapitole *code review* je možné ulehčit pomocí nástrojů, které provádí statickou analýzu automaticky.

Pod statickou analýzu patří i takzvané *lint* nástroje. Ty kontrolují základní chyby ve formátování a stylu dle kódových standardů zmíněných v kapitole 1.1.1. *Lint* nástroje jsou dobrým pomocníkem v dodržení konvencí, a to hlavně z toho důvodu, že kontrolují podmínky, které jsou špatně odhalitelné při *code review* (mezery, odsazení atp.). Většina *lint* nástrojů zároveň zvládne tyto problémy i automaticky odstranit. [51] *Lint* nástroje pro webové aplikace znázorňuje tabulka níže.

Tabulka 10. *Lint* nástroje pro vývoj webových aplikací [vlastní zpracování]

Programovací jazyk	**Lint nástroje**
Javascript	ESLint, JSLint, JSHint
Typescript	TSLint
Python	Pylint, Flake8, Pyflakes
Ruby	RuboCop, Reek, Flog
PHP	PHP_CodeSniffer, PHPMD, PHPLint
Java	Checkstyle
C#	StyleCop, ReSharper, FxCop

Nicméně většina moderních IDE má v sobě již základní *lint* nástroje zabudovány, ale ne vždy obsahují všechny potřebné kontroly, jako výše zmíněné nástroje.

Nástroje statické analýzy ale dokáží víc než jen kontrolu stylu. Tyto nástroje zlepšují kvalitu kódu pomocí odhalení bugů a kontroly datového toku. Některé kontrolují i bezpečností problémy jako SQL injekci a *cross-site scripting* XSS. Nástroje statické analýzy jsou sepsány v tabulce dole. Jednotlivé příklady jazyků jsou znázorněny v tabulce na následující straně.

Tabulka 11. Nástroje statické analýzy pro vývoj webových aplikací [vlastní zpracování]

Programovací jazyk	Nástroje statické analýzy
Javascript	Flow
Typescript	TypeScript Analyzer
Python	Bandit, CodeSonar
Ruby	Brakeman
PHP	PHPStan
Java	FindBugs, Coverity, CodeSonar, PMD
C#	NDepend, Coverity

1.9.2 Dynamická analýza

Dynamická analýza vyžaduje na rozdíl od statické spuštění kódu. Stejně jako dynamická analýza není schopna zachytit všechny chyby, které by zachytila statická, tak je zase schopna odhalit jiné chyby, které statická neodhalí.

Dynamická analýza provádí testování kódu k odhalení nechtěných chyb, problémů s pamětí či pádů aplikace.

1.9.2.1 Debuggery

Debuggery jsou v dnešní době již obvyklým pomocníkem vývojáře. Umožňují kontrolu kódu pomocí krokování programu.

1.9.2.2 Profilery

Profilery slouží k analýze výkonu kódu. Měří čas provedení metod a počet jejich volání. Dále také kontrolují alokaci paměti a tzv. *garbage collection*. Některé z nich dokáží i vyhodnocovat volání dalších služeb, které ovlivňují trvání (volání databáze, třetích stran, …). Profilery mohou být buď serverové nebo desktopové. [52]

1.9.2.3 Fuzzery

Fuzzery odhalují zranitelnosti v aplikaci. Nástroje pro fuzzing provádí automatické testování kódu pomocí vkládání náhodných vstupů do aplikace nebo REST API. Jedná se o testování pomocí černé skříňky, tedy že není známa implementace aplikace. Jejich cílem je odhalit nečekané chování aplikace včetně jejího pádu. [53] Existují jak open-source nástroje (např. American fuzzy lop, boofuzz, ABNF Fuzzer), tak i komerční jako Codenomicon's product suite, Peach Fuzzing Platform apod. [54]

1.9.2.4 Bezpečnostní scannery

Bezpečnostní scannery jsou pokročilejší nástroje, které kontrolují správnou implementaci aplikace jako je zabezpečení hesel, komunikace, ochrana citlivých údajů, ale také třeba souběhy či uváznutí.

1.9.3 Testování kódu

Testování softwaru je proces vyhodnocení a ověření softwarového produktu, že dělá to, co dělat má. [55] Existuje mnoho druhů testování, které v různých fázích ověřuje aplikaci. Pro konkrétní práci jsou ale důležité ty, které patří do fáze vývoje – neboli ty, které provádí sám vývojář.

1.9.3.1 Jednotkové testování

Jednotkové testy představují první fázi testování, která nastává po (v případě TDD před) dokončení určité části kódu. Jedná se o základní ověření kvality práce vývojáře jím samotným. Toto testování je nejlevnější opravou chyb.

- oprava vyžaduje čas pouze jednoho člověka,
- je odhalena hned jakmile vývojář tento kód implementoval (pamatuje si, kde by mohla být chyba – neztrácí čas orientací se ve starším kódu),
- chyba nevyžaduje žádné vytváření reportů či dokumentace.

Jednotkové testy také nabádají vývojáře, aby psal zdrojový kód tak, aby byl dobře testovatelný, lépe čitelný a kvalitnější.

Nejdůležitějším efektem je ovšem to, že testy zůstávají v projektu a při přidání/úpravě funkcionality slouží jako ověření, že již naimplementovaná funkcionalita zůstala neporušená.

Jednotkovými testy prověřujeme:

- jednotlivé větve zpracování,
- speciální případy funkce, které mohou nastat,
- komplikované úseky,
- okrajové případy,
- chybná vstupní data,
- vyhazované výjimky.

Jednotkové testy poslouží i během opravování zavedených chyb. V případě, že byla nahlášena chyba v aplikaci, jednotkový test je implementován tak, aby chybu reprodukoval. Při jeho prvotním spuštěním test tedy selže. Projití tohoto testu značí, že byla chyba opravena. Ostatní testy pak verifikují, že oprava nerozbila jinou část kódu a samotný test chyby v aplikaci zůstane pro zaručení toho, že byla chyba zcela odstraněna. [56]

Následující tabulka zobrazuje ukázky dostupných nástrojů pro jednotkové testování.

Tabulka 12. Knihovny pro jednotkové testování pro jednotlivé jazyky webového vývoje [vlastní zpracování]

Programovací jazyk	**Knihovny pro jednotkové testování**
Javascript	Jest, Mocha, QUnit, Jasmine, Karma
Typescript	TS-Jest, Mocha, Chai, Sinon, Jasmine
Python	unittest, pytest, nose, doctest
Ruby	RSpec, MiniTest, Test::Unit, Cucumber
PHP	PHPUnit, Codeception, SimpleTest, Behat
Java	JUnit, TestNG, Mockito, PowerMock
C#	NUnit, xUnit.net, MSTest, FluentAssertions

1.9.3.2 Testování vlastností softwaru

Testování vlastností softwaru (anglicky *feature testing*) je prováděno vždy při přidání nové vlastnosti softwaru nebo modifikaci již existující vlastnosti. Pomáhá tak definovat funkčnost všech požadovaných vlastností.

Při testování vlastností je důležité rozumět požadavku. Tedy základem je mít správně nastavená akceptační kritéria implementované vlastnosti, která určují tělo testovací metody.

Toto testování je třeba, protože i když prochází všechny jednotkové testy, tak celková vlastnost nemusí být správně navázána. Navíc často se stává, že implementace jedné vlastnosti rozbije již existující vlastnost. Jedná se tedy o ověření aktuálního stavu softwaru a dokončených požadavků. [57]

Součástí testování vlastností je komunikace mezi objekty, metodami, tak i například HTTP požadavky a jejich odpovědi.

1.9.3.3 End-to-end Testování

End-to-End testování (zkráceně E2E) je ověření funkčnosti aplikace od začátku do konce v prostředí podobnému produkčnímu. Kontroluje integraci všech softwarových závislostí (hardware, internetové připojení, databáze, uživatelské rozhraní, …). [58]

Nástroje dostupné pro testování E2E jsou například Cypress, TestCafe, Selenium WebDriver, atd.

2 DOPORUČENÍ PRO KVALITNÍ ZABEZPEČENÍ APLIKACE

Naprosto nejdůležitějším aspektem při vývoji aplikací je jejich zabezpečení. V případě, že jsou ohrožena data uživatele, je ohrožen i jeho uživatelský zážitek stejně jako samotná reputace aplikace. Bezpečnost by měla být podstatná už jen z důvodu zachování profesionálního jednání a budování důvěry s uživateli.

Zabezpečení je téma, které se neustále vyvíjí a vylepšuje, stejně jako se stupňují a vylepšují útoky na aplikace.

Důležitým bodem pro zabezpečení aplikace je minimalizace technologického dluhu pomocí zajištění nejnovějších verzí používaných technologií. Nelze tedy zanedbávat udržování (anglicky *maintenance*) aplikace, které by mělo být prováděno pravidelně a dostatečně často.

V této práci budou zmíněna témata jako je zabezpečení ukládaných dat, obecná témata ochrany aplikace, správa chyb a logování, bezpečnostní organizace a regulace.

2.1 Zabezpečení ukládaných dat

Aby byla dostatečně zabezpečená ukládaná data, je třeba se při jejich zpracování věnovat hned několika aspektům.

Prvním aspektem je zajištění jejich pravosti. V případě, že uživatel zadává data, je třeba vždy jeho data zkontrolovat, aby byla zajištěna jejich správnost, která udržuje aplikaci v konzistentním stavu. Aplikace by měla validovat data v uživatelském rozhraní pro rychlou opravu, a provádět důslednější kontroly při zpracovávání požadavku na serveru.

Dalšími důležitými aspekty jsou například dostatečné zabezpečení serveru, komunikace a již zmíněné časté udržování. Tato kapitola je věnována zajištění bezpečnosti proti odcizení dat díky zašifrování informací.

2.1.1 Hash Algoritmy

Pro zajištění soukromí uživatele je nutné chránit citlivé údaje pomocí šifrování.

Hash algoritmus je jednocestná funkce, která převede vstupní data o jakékoliv délce na zašifrovaná data o pevné délce. Používá matematické operace ke zmenšení vstupu na bitový řetězec, které budou natolik komplikované, že jej počítače nejsou schopny vrátit zpět do původní podoby před šifrováním. [59]

2.1.1.1 Bezkoliznost

Další vlastností algoritmu je bezkoliznost, která udává, že pro dva různé vstupy algoritmus zaručuje různý výstup.

Definice 1: *Hashovací funkce h je **silně bezkolizní**, pokud je výpočetně nemožné najít dvě zprávy $M_1 \neq M_2$, pro které platí $h(M_1) = h(M_2)$.*

Definice 2: *Hashovací funkce h je **slabě bezkolizní**, pokud je ke zprávě M_1 výpočetně nemožné najít takovou zprávu M_2, pro kterou platí $M_2 \neq M_1$ a zároveň $h(M_1) = h(M_2)$.*

Tuto charakteristiku zařizuje tzv. lavinový efekt, který při změně 1 bitu vstupní zprávy zajistí změnu každého bitu s pravděpodobností $\frac{1}{2}$. [59]

2.1.1.2 Narozeninový paradox

Narozeninový paradox je 50% šance, že v místnosti s 23 lidmi mají dvě osoby narozeniny ve stejný den.

V rámci hash algoritmů to znamená, že pro 40bitovový kód existuje pravděpodobnost 50 %, že mezi 2^{20} hash řetězci bude existovat kolize. Pro 256bitový kód je tato pravděpodobnost u 2^{128} dokumentů.

Tento paradox prokazuje, že kromě kvality algoritmu záleží i na délce výstupu. [60]

2.1.1.3 Kryptografické solení

Kryptografická sůl anglicky *salt* představuje náhodné byty přidané ke vstupu do hashovací funkce. To způsobí, že i dva stejné vstupy vytvoří ale odlišný hash. Sůl se připojuje k hashovanému vstupu. [61]

2.1.1.4 Kryptografické pepření

Kryptografický pepř je další způsob, kterým okořenit vstup. Narozdíl od soli, pepř není náhodně generovaný, ale je to statická hodnota pro všechna data. Pepř je uložen v jiné části aplikace, aby bylo zamezeno jeho zjištění. Pepř je často vybírán administrátorem stránky. [61]

2.1.1.5 Bezpečné algoritmy

V rámci pokroku v počítačových technologií je nutné nespoléhat na nejstarší hash algoritmy, ale spíše se zaměřit na ty novější. V roce 2007 bylo vydáno prohlášení NBÚ, kde bylo

doporučeno nepoužívat hash algoritmy s výstupem menším než 160 bitů. [62] Dále doporučilo přejít od SHA-1 algoritmu na novou generaci SHA-2 v horizontu 3-5 let.

SHA-1 se stále ještě používá v některých protokolech jako je SSL, PGP, SSH, IPSec, kde slouží jako defaultní hash algoritmus a je tedy nutné ho nahradit za vyšší verzi. [59]

Secure Hash Algorithm 2 (zkráceně SHA-2) byl vydán v roce 2002 jako nástupce SHA-1 z roku 1993. Oba algoritmy vychází z Message-Digest Algorithm 5 (MD5), který je v každé verzi zkomplikován a optimalizován. Do rodiny SHA-2 patří:

- SHA-256 s hash o 256 bitech, který je využíván například pro bitcoinové transakce
- SHA-384 s hash o 284 bitech,
- SHA-512 s hash o 512 bitech. [59]

Secure Hash Algorithm 3 Z důvodu obav o prolomení rodiny algoritmů SHA-2 byla již v roce 2008 vyhlášená soutěž o nástupce SHA-2. Finalisty soutěže se stali BLAKE, Grøstl, JH, Keccak a Skein a výhercem se nakonec stal v roce 2015 Keccak. [59]

SHA-3 je tedy založena na zcela odlišných principech než předchozí generace. Je založen na novém kryptografickém přístupu zvaném houbová konstrukce, kde jsou data absorbována do houby, a nakonec vymačkána ven.

NIST ale nevnímá SHA-3 jako nahrazení SHA-2, nýbrž jako rozšíření dosavadních možností.

Mezi jeho verze patří SHA3-224, SHA3-256, SHA3-384, SHA3-512. Dále obsahuje rozšířené výstupní funkce SHAKE-128, SHAKE-256. Tyto funkce slouží k *hashingu* domén, randomizovanému *hashingu*, zašifrování streamu a generování MAC adres. [63]

Bcrypt byl vybudován v roce 1999 pro zpomalení útoků hrubou silou. V procesu *hashingu* je zahrnuto i osolení, které chrání před útoky duhovými tabulkami.

Je založen na symetrické blokové šifře blowfish tj. Písmeno b v názvu a hashovací funkci crypt používanou UNIX heslovacím systémem. Bcrypt má v sobě zabudováno i již zmíněné solení. [64]

Scrypt byl vyvinut jako vylepšení SHA-256 a své místo našel v blockchainové komunitě. Poprvé byl publikován v roce 2009. Má menší energetické nároky než SHA-256 a těžba je 4x rychlejší než u Bitcoinu. Využívá se pro šifrování peněženek, souborů a hesel. [65]

Argon2 je hash algoritmus, který v roce 2015 vyhrál soutěž Password Hashing Competition a od té doby zůstal na pozici doporučených algoritmů pro *hashing* hesel. Je ve třech verzích:

- Argon2i, který je odolný vůči útoku postranním kanálem,
- Argon2d, který je odolný vůči prolamování na GPU,
- Argon2id, což je kombinace obou výše zmíněných verzích.

Tento algoritmus lze přizpůsobit podle třech parametrů: [66]

- požadovaný čas výpočtu,
- požadovaný nárok na paměť,
- počet paralelních vláken.

Nabízí také možnost zvolení délky výsledného klíče. [67] Argon2 nicméně slouží k pomalému *hashing* hesel a vyplatí se pro časy vyšší než 1 s. Je to reakce na rychlé *hashing* SHA-256, z logiky věci, že co lze rychle „zahashovat“, lze rychleji i prolomit.

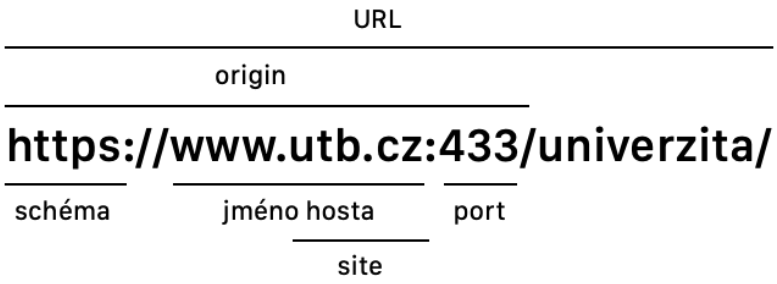
2.2 Ochrana aplikace

Ochrana aplikace v této práci zahrnuje základní informace a znalosti, která se k tématu zabezpečení využívají.

V následujících podkapitolách je rozebrán rozdíl origin a site, autentizace a autorizace.

2.2.1 Rozeznání origin a site

Jednotný Lokátor Zdroje anglicky Uniform Resource Locator známý především zkráceně jako URL, je webová adresa, která specifikuje umístění informačního souboru na internetu. Příklad URL s vysvětlením jeho jednotlivých částí je uveden na obrázku níže.



Obrázek 27. Příklad URL s vysvětlením origin a site [vlastní zpracování podle [68]]

Origin je kombinace schématu (též označovaného jako protokol), jména hosta a volitelně portu. U origin rozlišujeme, zda je shodný s původním, čemuž se říká same-origin, nebo je odlišný tedy cross-origin. Rozeznání origin je znázorněno v tabulce níže.

Site je část jména hosta označovaná jako eTLD+1. Je složena z eTLD neboli efektivní domény nejvyššího řádu a domény před ní. Efektivní doména nejvyššího řádu je registrovaná ve Veřejném seznamu přípon (anglicky *Public Suffix List*), který obsahuje kromě domén nejvyššího řádu jako je např. .com, .cz, .net, tak i domény nižšího řádu, které ovšem nedefinují celou site, jako je .co.uk. U site rozlišujeme stejně jako u origin, jestli je stejného původu. Pokud je eTLD+1 stejná, pak je same-site, jinak je cross-site. [68]

Vysvětlení rozdílů termínu origin a site jsou v tabulce níže.

Tabulka 13. Vysvětlení same/cross-origin a same/cross-site [vlastní zpracování podle [68]]

Původní URL	**Nová URL**	**Vysvětlení**	**Same (✓) nebo Cross (✗)**	
			Origin	**Site**
https://www.utb.cz:433	https://www.muni.cz:433	Odlišná doména	✗	✗
	https://utb.cz:433	Bez subdomény	✗	✓
	https://fai.utb.cz:433	Odlišná subdoména	✗	✓
	http://www.utb.cz:433	Odlišný protokol	✗	✓
	https://www.utb.cz:80	Odlišný port	✗	✓
	https://www.utb.cz:433	Stejný origin	✓	✓
	https://www.utb.cz	Stejný implicitní port	✓	✓

2.2.2 Autentizace

Autentizace je proces ověření identity sloužící k ochraně dat. Při autentizaci uživatel poskytne údaje, které jsou dohodnutou sdílenou informací mezi uživatelem a systémem.

V následujících podkapitolách jsou stručně uvedeny některé příklady možností autentizace.

2.2.2.1 Přihlašovací jméno a heslo

Základní přihlašovací metodou je pomocí jména a hesla. V tomto případě je třeba zajistit bezpečné uložení údajů a jejich zabezpečení se věnovat i v dlouhodobém hledisku.

Zatímco přihlašovací jméno může být veřejné (např. e-mailová adresa), heslo musí být zcela soukromé. Kvůli jejich důvěrnosti je nutné hesla chránit před krádeži, jelikož jsou velmi často právě z důvodu slabého zabezpečení napadány.

Je důležité uživatele přimět k používání dostatečně bezpečných hesel, nicméně se také vyhnout příliš restriktivním pravidlům, která mohou uživatele spíše odradit.

Při ukládání hesel je nutné myslet na jejich zabezpečení. Odrazujícím případem je například Facebook, který ukládal miliony hesel z Instagramu v prostém textu čili čitelné podobě. [69] Heslo je třeba uložit v zabezpečené formě s pomocí hash, o kterých byla zmíněna kapitola 2.1.1.5. Zároveň pro zlepšení zabezpečení použít i již popsané kryptografické solení a pepření.

Důležitým faktorem je zde předávání údajů. Heslo by nikdy nemělo být přenášeno skrze nezabezpečenou komunikaci. Vždy by měla být použitá HTTPS metoda POST, jelikož GET zobrazuje data v čitelné podobě během komunikace.

2.2.2.2 OAuth

OAuth neboli *Open Authorization* je standardem pro autentizaci umožňující se přihlásit přes již existující přihlašovací údaje jiné služby (Google, Facebook atp.). Tato autorizace je dostupná skrze token, který je poskytnutý službou třetí strany pomoci takzvaného autorizačního toku.

První verze byla vydána jako autorizační metoda pro API Twitteru v roce 2007. O tři roky později společnost IETF OAuth Working Group vydala první návrh OAuth 2.0, která poskytuje uživateli možnost udělit přístup aplikaci třetí strany webovým zdrojům bez sdílení hesla. Jedná se ale o zcela nový protokol, který není zpětně kompatibilní s OAuth 1.0.

1. klientská aplikace vyžádá autorizaci pro přístup k chráněným zdrojům,
2. vlastník zdroje se autentizuje a autorizuje požadavek k přístupu ke zdroji,
3. klient vyžádá přístupový token od autorizačního serveru pomocí autorizačního přístupu získaného v předchozím kroku,
4. pokud je klientova identita autentizována a autorizační přístup je validní, pak autorizační server poskytne přístupový token klientovi,

5. klient může nyní požadovat chráněné zdroje pomocí přístupového tokenu,
6. pokud je přístupový token validní, pak jsou mu od serveru vráceny požadované zdroje. [70]

2.2.3 Autorizace

Autorizace zajišťuje ověření přístupu k chráněným zdrojům. Autorizaci lze rozdělit na několik typů definovaných v následujících podkapitolách.

2.2.3.1 Autorizace založená na atributu

Autorizace založená na atributu vychází z *Attribute-Based Access Control* (zkráceně ABAC). Vtomto případě je kontrolováno, že uživatel má přístup k danému zdroji na základě určité vlastnosti. Příkladem je, že uživatel může přistupovat ke zdroji (např. alkohol) po dosažené určité věkové hranice. [71]

2.2.3.2 Autorizace založená na roli

Klasickým příkladem je kontrola přístupu ke zdroji na základě uživatelovy role, která vychází z *Role-Based Access Control* (zkráceně RBAC). Jedná se například o přístup na administrátorské stránky apod. [71]

2.2.3.3 Autorizace založená na vztahu

V případě další běžné autorizace jde o přístupu ke zdrojům, ke kterým má určitý vztah. Tato autorizace vychází z *Relationship-Based Access Control* (anglicky ReBAC). Například zákazník může přistupovat k objednávkám, které sám vytvořil. [71]

2.3 Správa chyb a logování

Standardním způsobem správy chyb je zařízení jejich logování, které slouží především pro dlouhodobé sledování softwaru. Principem logování je zaznamenávání aktuálního stavu aplikace, které je řízeno v kódu.

Typicky sledujeme:

- výskyt chyb,
- výskyt výjimek,
- běžnou činnost softwaru,
- konfiguraci programu.

Výhoda logování spočívá v poskytnutí posloupnosti zpráv, které umožňují prohledávání v časové souvislosti.

Zapisované informace se ukládají do souborů (nejčastěji .txt soubory), se kterými lze následovně pracovat. Dále je možné informaci převádět i například na konzoli nebo pro zpracování externími aplikacemi pro lepší přehled a vytvoření statistik.

Logy slouží pro:

- dohledání vzniklé chyby,
- informování o stavu softwaru,
- ověření funkčnosti softwaru,
- dokumentaci běhu softwaru. [56]

Jak již bylo zmíněno v kapitole pro REST API, je důležité vracet jak adekvátní status k odpovědi, tak i dostatečně konkrétní zprávu o chybě

Pro správný monitoring nasazeného systému je vhodné zasílat chybové logy na zvolenou externí službu.

Součástí čistého kódu je doporučeno pro typ chyby vytvořit i třídu, která se o nahlášení chyby stará. Zamezí se tak duplikaci kódu a zpřehlední se správa chybových hlášek.

V rámci dokumentaci API již bylo zmíněno, že by součástí popisu koncového bodu měly být přiloženy příkladné odpovědi. K tomu se vztahují i chybové odpovědi a ukázané možné chybové hlášky, které může klient očekávat. [72]

Velmi podstatným problémem, který je ovšem důležité zmínit, je že velmi podrobná správa chyb může vytvořit zranitelnost systému. Toto se týká případů, kdy je ve zprávách zmíněná cesta ke zdrojovému souboru, obsah databáze atp.

Z tohoto důvodu je důležité mít správu chyb pod kontrolou a věnovat jí pozornost. [73]

2.4 Bezpečnostní organizace a frameworky

Existuje řada frameworků, které se zabývají bezpečností aplikací. Jejich význam spočívá v publikaci doporučení, čemu věnovat pozornost a jaké zranitelnosti je nutné prověřit.

V následujících podkapitolách budou stručně uvedeny základní frameworky, které mohou vývojáři pomoci k zajištění bezpečnosti jeho softwaru.

2.4.1 OWASP

Open Web Application Security Project neboli OWASP je mezinárodní nezisková organizace zabývající se, jak už z názvu vyplývá, bezpečností webových aplikací. Základním principem této organizace je, že všechny jejich materiály jsou zdarma a lehce dostupné na jejich webové stránce, aby měl každý příležitost vylepšit bezpečnost svých aplikací. Materiály obsahují dokumentaci, nástroje, videa a fóra.

Mezi jejich nejznámější projekt patří ovšem OWASP TOP 10. Jedná se o aktualizovaný seznam deseti nejkritičtějších slabin webových aplikací, které jsou sestaveny týmem bezpečnostních expertů z celého světa. OWASP TOP 10 slouží k vytvoření povědomí o bezpečnosti a důležitých rizik, kterým je třeba věnovat pozornost. [74]

2.4.2 CWE

Common Weakness Enumeration CWE je seznam softwarových bezpečnostních slabin v softwarech a hardwarech, která je tvořena díky zpětné vazbě CWE komunity. CWE je sponzorováno MITRE Corporation.

Podobně jako OWASP zveřejňuje seznam 25 běžných slabin. CWE se zaměřuje na slabiny softwaru, firmwaru, hardwaru nebo služby. Nicméně zatímco OWASP řeší obecné problémy softwaru, tak CWE obsahuje i problémy týkající se určité technologie, či programovacího jazyka. [75]

2.5 Regulace

Při návrhu softwaru je třeba také počítat i s regulacemi, které se v daném státě nacházejí. V dnešní době je nejpodstatnější regulací GDPR (*General Data Protection Regulation*), která se týká zemí Evropské unie.

Pro vývojáře tato regulace znamená několik věcí: [76]

2.5.1.1 Implementovat bezpečný kód

Tento bod by měl být dodržen nehledě na regulace, nicméně o to háklivější téma to je, když není dodržen.

Součástí tohoto téma je třeba se soustředit na kvalitní autentizaci, a především autorizaci přístupu k citlivým datům.

2.5.1.2 Provádět pravidelné a časté bezpečnostní testy

Bezpečnost aplikace by měla být pravidelně testována. Tím mohou být například již zmíněné testy dynamické analýzy nebo přímo penetrační testy.

2.5.1.3 Neukládat nepotřebná data

Dalším výrazným faktorem je používání nepotřebných dat pro aplikaci. Sbírání dat, které nemají souvislost s produktem je proti pravidlům GDPR, je tedy na místě zvažovat, zda se je nutné danou informaci ukládat, či ne.

2.5.1.4 Zabezpečovat citlivá data

Je třeba věnovat pozornost i zabezpečení citlivých dat a to nejen hesel. Je potřeba zvážit šifrování některých informací.

2.5.1.5 Umožnění správy uživatelských dat

Mezi důležitý bod GDPR patří právo uživatele být zapomenut a právo na smazání dat. Proto je důležité v rámci aplikace implementovat správu uživatele, která umožní jeho informace upravit/smazat, stejně jako zlikvidovat celý jeho profil.

Smazání dat by mělo nastat i v případě, že uživatel aplikaci již nepoužíval určitou dobu. Doba vypršení platnosti dat je dána v zásadách ochrany osobních údajů.

2.5.1.6 Implementace cookies

Dnes již nepřehlédnutelný prvek webových stránek se týká schvalování ukládání informací pomocí tzv. cookies.

Internetové cookies jsou malé textové soubory dat, které obsahují informace v uživatelově webovém prohlížeči. Cookies se dělí na cookies první strany a třetí strany.

Cookies první strany jsou ukládány webovou aplikací, kterou uživatel navštívil a fungují jen v této aplikaci. Tyto cookies pomáhají uživateli v jednodušším průběhu aplikací díky zapamatování si informací o jeho aktivitách.

Cookies třetí strany se týkají externích domén, které uživatel nenavštívil. Sledují uživatele skrze stránky, díky kterým cílí na uživatele.

Cookies se dále dělí na permanentní a relační. Permanentní zůstává v prohlížeči na delší dobu, zatímco relačním cookies vyprší platnost, jakmile je prohlížení ukončeno.

Těchto souborů se kromě GDPR týká ještě zákon o cookies neboli ePrivacy Directive, která ochraňuje uživatelova práva na soukromí díky tomu, že si mohou zvolit, jaké údaje společnosti shromažďují. Netýká se to ovšem jen HTTP cookies, ale také web beacons, e-tagů, Flash cookies a HTML5 Local Storage.

Cookies se dělí na několik typů:

- Nezbytné (anglicky *strictly necessary*),
- Analytické (anglicky *performance*),
- Funkční (anglicky *functionality*),
- Marketingové (anglicky *targeting/advertising*).

Pro zákon jsou nejdůležitější striktně nezbytné, jelikož jsou nutné pro funkci aplikace. Striktně nezbytné cookies nevyžadují žádnou kontrolu a mohou být nastaveny bez jakéhokoliv modelu pro schvalování. Nutné je si ovšem uvědomit jejich definici a dodržovat jejich používání pouze na úrovni funkcionálních potřeb. I přesto, že není nutné tyto cookies schvalovat, je doporučením uživatele informovat o jejich použití, aby byli schopni je identifikovat a oddělit od ostatních typů.

Zákon už nicméně neudává, jakým způsobem by měl uživatel rozhodovat, zda cookies jsou použity nebo ne. Není tedy dáno, že musí mít na výběr určit použití každého typu zvlášť či je seskupit.

Modely schvalování cookies se dělí na několik typů. [77]

1. **Schválení informováním.** V tomto modelu dojde pouze o informaci, že stránka používá cookies. Rozhodnutí uživatele tedy spočívá v tom, že buď použití příjme, nebo musí stránky opustit. Jedná se o nejjednodušší přístup, který vyžaduje nejméně úsilí.
2. **Implicitní schválení.** Rozdíl od předchozího modelu je takový, že má uživatel možnost již nastavené použití zrušit.
3. **Schválení pokračováním.** Tento model na rozdíl od předchozích dvou nemá na počátku nastaveno použití cookies, ale následující interakce uživatele bude brána jako souhlas s jejich využíváním. Výjimkou schvalující interakce je pokračování na detail o schválení informací. V některých zemí je tento model určen jako minimální možný.
4. **Explicitní schválení.** Zde jsou cookies zablokovány do doby, než uživatel provede specifickou akci, která označuje jeho přijetí cookies. Většinou je to provedeno pomocí tlačítka „Přijímám“ nebo „Souhlasím“. Tento model vyvolal řadu implementací takzvaných cookies zdí (anglicky *cookie walls*), které při vstupu na stránku zamezí

uživateli jakoukoliv jinou interakci, dokud nedojde ke schválení. V České republice je tento typ vyžadován od roku 2022.

5. **Smíšené schválení.** Tento přístup nastavuje některé cookies implicitně a žádá o nastavení dalších.

Postupem pro zacházení s cookies je tedy:

1. Roztřízení cookies podle technologie
2. Nastavení konzistentních jmen
3. Rozdělení cookies podle jejich typu
4. Definování zásad používání souborů cookie.

II. PRAKTICKÁ ČÁST

3 VÝVOJ POMOCÍ API-FIRST PŘÍSTUPU

Ať už je vývoj řízen pomocí API-first nebo ne, pro udržování API je vhodné použít softwarový nástroj, který umožňuje správu API. Existuje několik možností jako je například Postman, Insomnia, RapidAPI, HTTPie, Hoppscotch a další. Pro tuto práci byla využita aplikace Postman, přes kterou byl řízen API-first vývoj.

V prvním kroku dojde k vytvoření definice, ze které je následně vytvořena dokumentace, která vygeneruje jednotlivé kolekce endpointů, které po nastavení lze spouštět a testovat.

3.1 Definice API endpointů

Definice se píší buďto v OpenAPI nebo v JSON formátu. OpenAPI je jazyk přímo určen ke specifikaci endpointů a je založený na jazyku YAML.

Součástí specifikace je definice endpointů pomocí cesty, HTTP metody, použitých parametrů, těla požadavku a odpovědi. Ukázka struktury je na následující stránce.

index.yaml

```yaml
openapi: '3.0.0'
info:
    version: '1.0.0'
    title: API Title
    description: Describe the API.

servers:
    - url: https://api.example.com/v1
      description: ...

paths:
    /resources:
        post:
            summary: Create Title
            requestBody:
                - required: true
                - content:
                        application/json:
                            schema:
                                ...
            responses:
                "201":
                    description: 201 Created
                    content:
                        application/json:
                            schema:
                                ...
    ...
    /resources/{resourceId}:
        get:
            summary: Detail Title
            ...
components:
    responses:
        BadRequest:
            content:
                ...
    schemas:
        Resource:
            ...
securitySchemes:
    ApiKey:
        ...
security:
    - ApiKey: []
```

Zdrojový kód 33. Struktura specifikace API v jazyku YAML [vlastní zpracování]

Kvůli větším rozměrům API co se počtu endpointů týče, je vhodné specifikaci rozdělit do zanořené struktury souborů. Zvýší se tím přehlednost a lépe se se specifikací pracuje.

Jelikož endpointy v části `paths` rozdělit nelze, využívá se rozdělení na jejich objekty, na které se potom odkazuje pomocí reference. To se týká parametrů, schémat, příkladů nebo odpovědí, které jsou pak definovány v části `components`. Tuto část lze přesunout do dalších souborů.

Ukázka takto vypadajícího takzvaného kořenového souboru `index.yaml` je na následující straně. Kořenový soubor značí hlavní bod specifikace, ze kterého poté vzniká dokumentace. V ideálním případě by v položce `components` byly ještě `examples` a `parameters`, ale tuto možnost Postman prozatím nepodporoval.

Jednotlivé komponenty odkazují na svoje indexy v podsložkách. Ty obsahují seznam položek definovaných objektech. Odkaz na seznam je proveden pomocí cesty, zatímco jednotlivé položky jsou už definovány pomocí reference na objekt. Příklad indexu pro schémata je potom na další stramě v ukázce Zdrojový kód 35.

Cesty zdroje jsou tedy vždy rozděleny na dvě skupiny – s parametrem a bez parametru. Cesty s parametrem definují *endpoint* pro získání všech zdrojů a *endpoint* pro vytvoření nového zdroje. Příklad jejich definice je uveden v ukázce Zdrojový kód 36. Všechny koncové body pracující s jedním konkrétním zdrojem jsou v cestě s parametrem, který je definován jednoznačným identifikátorem zdroje.

index.yaml

```yaml
openapi: '3.0.0'
info:
    version: '1.0.0'
    title: 'UNI API'
    description: ...
servers:
    - url: https://api.example.com/v1
      description: ...

paths:
    /locations:
        $ref: './paths/locations.yaml'
    /universities:
        $ref: './paths/universities.yaml'
    /universities/{universityId}:
        $ref: './paths/university.yaml'
    /users:
        $ref: './paths/users.yaml'
    /users/{userId}:
        $ref: './paths/user.yaml'
    /universities/{universityId}/students:
        $ref: './paths/students.yaml'
    /universities/{universityId}/students/{studentId}:
        $ref: './paths/student.yaml'
    /universities/{universityId}/administrators/:
        $ref: './paths/administrators.yaml'
    /universities/{universityId}/administrators/{administratorId}:
        $ref: './paths/administrator.yaml'

components:
    responses:
        $ref: './responses/index.yaml'
    schemas:
        $ref: './schemas/index.yaml'
securitySchemes:
    ApiKey:
        type: apiKey
        in: header
        name: X-Api-Key
    security:
        - ApiKey: []
```

Zdrojový kód 34. Kořenový soubor specifikace API [vlastní zpracování]

schemas/index.yaml

```yaml
Error:
    $ref: "./error/error.yaml"
FieldError:
    $ref: "./error/field-error.yaml"
FieldErrors:
    $ref: "./error/field-errors.yaml"

Id:
    $ref: "./id.yaml"

Location:
    $ref: "./university/location.yaml"
University:
    $ref: "./university/university.yaml"
UniversityAdministration:
    $ref: "./university/university-administration.yaml"
UniversityRelations:
    $ref: "./university/university-relations.yaml"

UserAdministration:
    $ref: "./user/user-administration.yaml"
User:
    $ref: "./user/user.yaml"
UserCredentials:
    $ref: "./user/user-credentials.yaml"
UserRoles:
    $ref: "./user/user-roles.yaml"

Administrator:
    $ref: "./user/student/administrator.yaml"

Student:
    $ref: "./user/student/student.yaml"
```

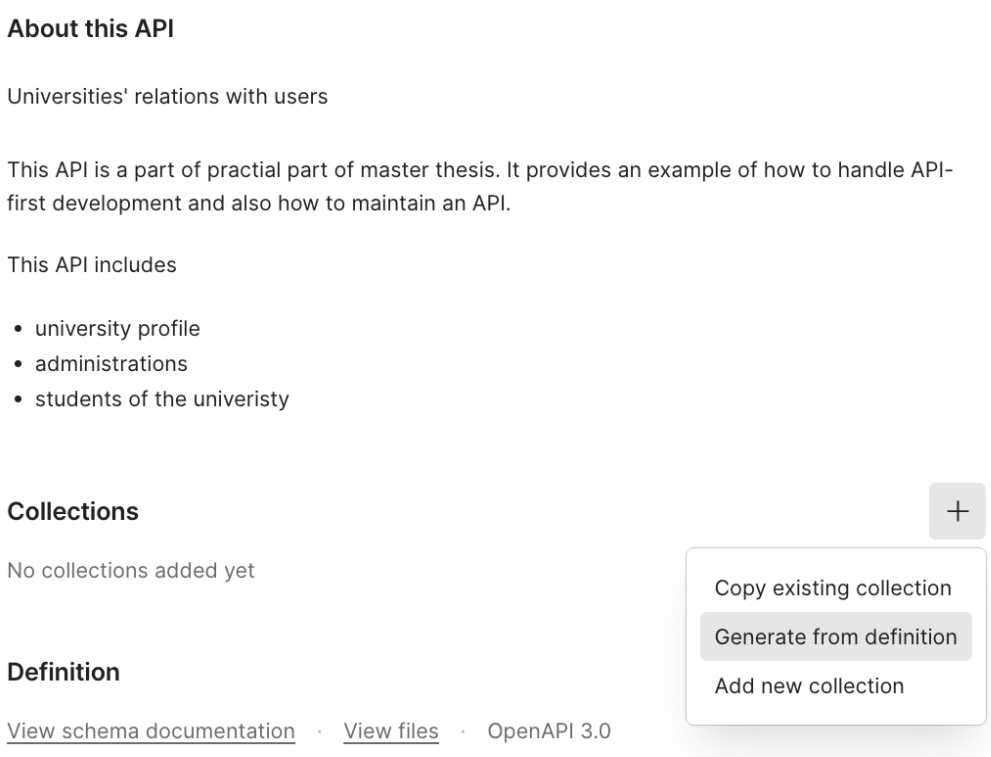
Zdrojový kód 35. Soubor index pro schémata specifikace [vlastní zpracování]

universities.yaml

```yaml
  post:
      summary: Create new university
      description:
      requestBody:
          required: true
          content:
              application/json:
                  schema:
                      $ref: "#/components/schemas/University"
      responses:
          "201":
              description: ...
              content:
                  application/json:
                      schema:
                          $ref: "#/components/schemas/University"
          "400":
              $ref: "#/components/responses/BadRequest"
          "401":
              $ref: "#/components/responses/Unauthorized"
          "403":
              $ref: "#/components/responses/Forbidden"
          "406":
              $ref: "#/components/responses/NotAccepted"
          "415":
              $ref: "#/components/responses/UnsupportedMediaType"
          500":
              $ref: "#/components/responses/InternalServerError"
  get:
      summary: List all universities
      description: Retrieves the university resources that are available.
      security: []
      responses:
          "200":
              description: ...
              content:
                  application/json:
                      schema:
                          type: array
                          items:
                              oneOf:
                                  - $ref: "#/components/schemas/University"
                                  - $ref:
                                   "#/components/schemas/UniversityAdministration"
  ...
```

Zdrojový kód 36. Příklad zpracování API specifikace cest [vlastní zpracování]

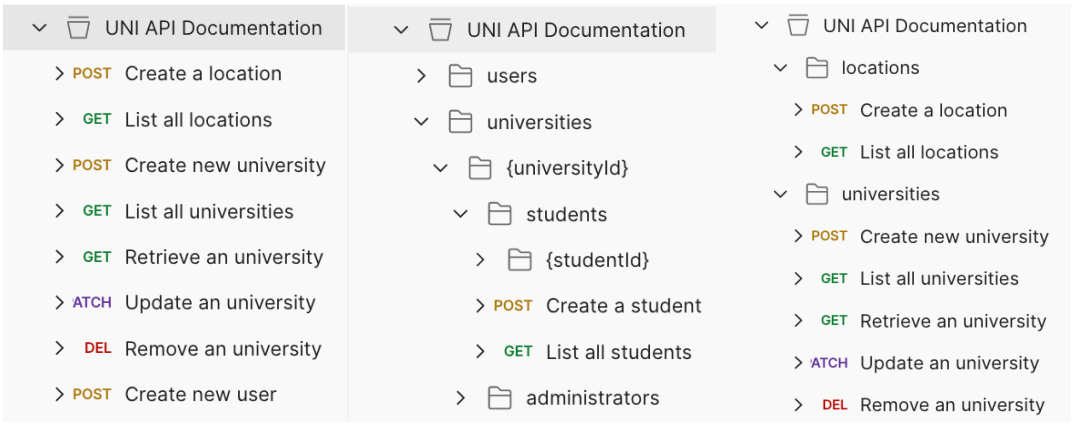
Vytvoření takového schématu pak pomůže přímo k vygenerování dokumentace a kolekcí. Způsob vygenerování kolekcí ze specifikace je znázorněn na obrázku níže.



Obrázek 28. Ukázka generování kolekce ze specifikace [vlastní zpracování]

Existují tři přístupy, jakým vygenerovat kolekce:

- Čistý seznam – výběr tagu bez jejich specifikace u koncových bodů (Obrázek 29 vlevo).
- Zanořené dle cesty – výběr cesty vytvoří zanořenou strukturu i s parametry (Obrázek 29 uprostřed).
- Pomocí tagu – výběr tagu s jejich specifikací vytvoří hlavní složky bez zanoření (Obrázek 29 vpravo).



Obrázek 29. Rozdíl mezi generováním API kolekcí [vlastní zpracování]

3.2 Vývoj API

Pro vývoj byl zvolen framework Laravel. Ten je zaměřen na vývoj webové aplikace i s uživatelským rozhraním, takže pro jeho použití na vývoj API je třeba určitých úprav.

V první řadě jde o smazání prvků spojených s vývojem UI. Jedná se o odstranění zdrojů pro web a přenastavení middleware autentizace na `'api'`.

V dalších krocích následuje přidání prvků, které API využívá, to se týká například definování odpovědí, úprava validace, správa chyb atp.

K nastavení odpovědí je vytvořena třída `Error`, která definuje obsah těla odpovědi při chybě a abstraktní třída chybné odpovědi, která je využita pro definování známých stavů odpovědí. Definice třídy znázorňuje Zdrojový kód 37.

app/Http/Responses/Error/Data/Error.php

```php
<?php

declare(strict_types=1);

namespace App\Http\Responses\Data\Error;

use App\Http\Responses\JsonSerializableTrait;
use JsonSerializable;

class Error implements JsonSerializable
{
    use JsonSerializableTrait;

    protected int $status;

    protected string $message;

    protected string $description;

    protected Details $details;

    /**
     * @var FieldError[]|null
     */
    protected ?array $errors = null;
}
```

Zdrojový kód 37. Třída `Error` [vlastní zpracování]

Pro práci s chybovými odpověďmi je vytvořena abstraktní třída na ukázce Zdrojový kód 38, ze které jsou vytvořeni jednotliví potomci podle jejich status kódu. Třída přetěžuje `Illuminate\Http\JsonResponse`, která vytváří odpověď v konstruktoru. V tomto případě je tedy v konstruktoru vytvořen objekt `Error`, kde jsou nastaveny statické prvky jako je kód stavu, název stavu nebo jeho slovní popis. Položky jako jsou podrobnosti o vzniku chyby nebo způsob opravy již statické nejsou, kvůli jejich možné závislosti na datech. Poslední možností je seznam chyb položek, který se používá především při 400 (Bad Request).

app/Http/Responses/Error/AbstractErrorJsonResponse.php

```php
<?php

declare(strict_types=1);

namespace App\Http\Responses\Error;

use App\Http\Responses\Error\Data\Details;
use App\Http\Responses\Error\Data\Error;
use App\Http\Responses\Error\Data\FieldError;
use Illuminate\Http\JsonResponse;

abstract class AbstractErrorJsonResponse extends JsonResponse
{
    public const STATUS_CODE = 0;

    protected const MESSAGE = '';

    protected const DESCRIPTION = '';

    protected ?array $fieldErrors = null;

    public function __construct()
    {
        parent::__construct(
            $this->createError(),
            static::STATUS_CODE
        );
    }

    private function createError(): Error
    {
        return (new Error())
            ->setStatus(static::STATUS_CODE)
            ->setMessage(static::MESSAGE)
            ->setDescription(static::DESCRIPTION)
            ->setDetails($this->createDetails())
            ->setErrors($this->getErrors());
    }

    private function createDetails(): Details
    {
        return (new Details())
            ->setReason($this->getReason())
            ->setResolution($this->getResolution());
    }

    abstract protected function getReason(): string;

    abstract protected function getResolution(): string;

}
```

Zdrojový kód 38. Abstraktní třída na zpracování chybných odpovědí [vlastní zpracování]

To, kdy se chyba zavolá je již závislé na konkrétním použití. Pro všechny potřebné stavové kódy je vytvořena výjimka pomocí příkazu `artisan`.

```
> sail artisan make:exception NotAcceptableException
```

Ten ve složce `app/Exceptions` vytvoří požadovanou třídu. Pro jednoduché použití výše definované struktury odpovědí je vytvořeno rozhraní jako v ukázce Zdrojový kód 39, které bude každá výjimka implementovat.

app/Exceptions/ApiResponsable.php

```php
<?php

declare(strict_types=1);

namespace App\Exceptions;

use Illuminate\Contracts\Support\Responsable;

interface ApiResponsable extends Responsable
{
    public function toResponse($request): AbstractErrorJsonResponse;
}
```

Zdrojový kód 39. Definice rozhraní `ApiResponsable` [vlastní zpracování]

Toto rozhraní rozšiřuje již existující rozhraní Laravel, které je používáno právě pro výjimky. Využití najde v třídě `Handler` (Zdrojový kód 41), která má na starosti obsluhu výjimek. Rozhraní definuje metodu `toResponse()`, která z výjimky vytvoří odpověď zaslanou zpět klientovi. Ukázka jeho implementace je znázorněna v ukázce Zdrojový kód 40.

app/Exceptions/NotAcceptedException.php

```php
<?php

namespace App\Exceptions;

use App\Http\Responses\Error\Client\NotAcceptableJsonResponse;
use Exception;

class NotAcceptedException extends Exception
    implements ApiResponsable
{
    public function __construct()
    {
        parent::__construct(
            NotAcceptableJsonResponse::DESCRIPTION,
            NotAcceptableJsonResponse::STATUS_CODE,
        );
    }

    public function toResponse($request): NotAcceptableJsonResponse
    {
        return new NotAcceptableJsonResponse(
            $request->getAcceptableContentTypes()
        );
    }
}
```

Zdrojový kód 40. Definice výjimky [vlastní zpracování]

Díky tomu můžeme použít třídu `Handler` a přetížit její původní chování, které používá výjimky Laravel framework. K tomu slouží metoda `render()`, která zpracovává odesílanou odpověď. Díky jejímu přetížení dojde vždy k vrácení odpovědi typu `AbstractErrorJsonResponse`, tedy k vlastnímu druhu odpovědi na výjimky. V případě, že výjimka našeho typu není, je zvolená vhodná odpověď díky `ErrorJsonResponseFactory` (Zdrojový kód 49), která podle kódu výjimky vrátí požadovanou chybu. V případě, že nedošlo k definování výjimky, dojde k vrácení odpovědi 500 (Internal Server Error). Třída `Handler` má na starosti také nahlašování chyb podle konfigurace. Pomocí položky `$dontReport` jsou označeny výjimky, u kterých nemá dojít k logu.

app/Exceptions/Handler.php

```php
<?php

namespace App\Exceptions;

use App\Http\Responses\Error\AbstractErrorJsonResponse;
use App\Http\Responses\Error\ErrorJsonResponseFactory;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;
use Illuminate\Http\Request;
use Throwable;

class Handler extends ExceptionHandler
{
    protected $dontFlash = [
        'current_password',
        'password',
    ];

    protected $dontReport = [
        BadRequestException::class,
        ForbiddenException::class,
        UnauthorizedException::class,
    ];

    public function render(
        $request,
        Throwable $e
    ): AbstractErrorJsonResponse {
        if ($e instanceof ApiResponsable) {
            return $e->toResponse($request);
        }

        return ErrorJsonResponseFactory::send($e->getCode());
    }
}
```

Zdrojový kód 41. Přetížená definice třídy `Handler` [vlastní zpracování]

Následně je nutné definovat, kde výjimky nastávají.

Pro status 406 (Not Acceptable) a 415 (Unsupported Media Type) je vytvořen middleware, který je spuštěn vždy před zpracováním požadavku. Na následující straně je ukázán příklad vytvořeného middleware pomocí příkazu `artisan`.

```
> sail artisan make:middleware CheckAcceptableResponse
```

app/Http/Middleware/CheckAcceptableResponse.php

```php
<?php

namespace App\Http\Middleware;

use App\Http\Responses\NotAcceptableJsonResponse;
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class CheckAcceptableResponse
{
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->expectsJson() !== true) {
            throw new NotAcceptedException();
        }

        return $next($request);
    }
}
```

Zdrojový kód 42. Middleware pro kontrolu hlavičky Accept [vlastní zpracování]

Pro kontrolu typu požadavku může middleware vypadat následovně:

app/Http/Middleware/SupportJsonRequest.php

```php
<?php

namespace App\Http\Middleware;

use App\Http\Responses\UnsupportedMediaTypeJsonResponse;
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class SupportJsonRequest
{
    public function handle(Request $request, Closure $next): Response
    {
        if (
            $request->isJson() !== true
            && !is_null($request->getContentTypeFormat())
        ) {
            throw new UnsupportedMediaTypeException();
        }

        return $next($request);
    }
}
```

Zdrojový kód 43. Middleware pro kontrolu hlavičky Content-Type [vlastní zpracování]

Nastavení 404 (Not Found) je o něco složitější kvůli implicitně volané Symfony výjimce. Lze k tomuto využít takzvaný fallback v souboru `routes/api.php`, kde úplně na konci souboru bude definováno přetížení nenalezené route jako je tomu v následující ukázce.

routes/api.php

```php
<?php

use App\Http\Responses\NotFoundJsonResponse;
use Illuminate\Support\Facades\Route;

...

Route::fallback(function () {
    return new NotFoundException();
});
```

Zdrojový kód 44. Ošetření chyby 404 (Not Found) [vlastní zpracování]

Je třeba dát si pozor, že při tomto ošetření často nastává výjimka 405 (Method Not Allowed), která by tedy měla být odchycena v `ErrorJsonFactory`.

Při implementaci autentizace je třeba přetížit i výjimky, které jsou generovány automaticky. Pro přetížení 401 (Unauthorized) lze využít middleware `Authenticate`. Ten v původní verzi řeší přesměrování na jinou stránku v případě chybějícího přihlášení. To ale není u API potřeba. Aby bylo dosaženo požadované funkcionality, je místo metody `redirectTo()` přetížena metoda `handle()`, která místo původní výjimky (Zdrojový kód 45) vrátí nově definovanou (Zdrojový kód 46).

app/Http/Middleware/Authenticate.php

```php
<?php

namespace App\Http\Middleware;

use Illuminate\Auth\Middleware\Authenticate as Middleware;
use Illuminate\Http\Request;

class Authenticate extends Middleware
{
    protected function redirectTo(Request $request): ?string
    {
        return $request->expectsJson() ? null : route('login');
    }
}
```

Zdrojový kód 45. Původní middleware autentizace [Laravel]

app/Http/Middleware/Authenticate.php

```php
<?php

namespace App\Http\Middleware;

use App\Http\Responses\UnauthorizedJsonResponse;
use Closure;
use Illuminate\Auth\AuthenticationException;
use Illuminate\Auth\Middleware\Authenticate as Middleware;
use Illuminate\Http\Request;

class Authenticate extends Middleware
{
    public function handle(
        $request,
        Closure $next,
        ...$guards
    ): mixed {
        try {
            $this->authenticate($request, $guards);
        } catch (AuthenticationException) {
            throw new UnauthorizedException();
        }

        return $next($request);
    }
}
```

Zdrojový kód 46. Upravený middleware autentizace pro ošetření chyby 401 (Unauthorized) [vlastní zpracování]

V případě odpovědi 403 (Forbidden) je třeba tuto chybu řešit až na úrovni autorizace, která bude zmíněna v zabezpečení.

Velmi důležitým faktorem při vytváření API je validace. Defaultně je validace polí prováděna pomocí třídy `\Illuminate\Foundation\Http\FormRequest`, u které v případě invalidního vstupu dojde k přesměrování uživatele a navrácení chybných polí. Tomuto chování lze zabránit vytvořením dědící třídy `ApiRequest`, která bude přetěžovat původní metodu neúspěšné validace jako na ukázce Zdrojový kód 47.

app/Http/Requests/ApiRequest.php

```php
<?php

declare(strict_types=1);

namespace App\Http\Requests;

use App\Http\Responses\BadRequestJsonResponse;
use Illuminate\Contracts\Validation\Validator;
use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\ValidationException;

class ApiRequest extends FormRequest
{
    protected function failedValidation(Validator $validator): never
    {
        $errors = (new ValidationException($validator))->errors();

        abort(new BadRequestJsonResponse($errors));
    }
}
```

Zdrojový kód 47. Přetížené validování vstupních dat pro ošetření chyby 400 (Bad Request) [vlastní zpracování]

Odpověď Bad Request je přitom definována následovně.

app/Http/Responses/Error/Client/BadRequestJsonResponse.php

```php
<?php

declare(strict_types=1);

namespace App\Http\Responses\Error\Client;

use App\Http\Responses\Error\Data\FieldError;
use Symfony\Component\HttpFoundation\Response;

class BadRequestJsonResponse extends AbstractErrorJsonResponse
{
    public const STATUS_CODE = Response::HTTP_BAD_REQUEST;

    protected const MESSAGE = 'Bad Request';

    protected const DESCRIPTION = 'Invalid parameters...';

    /**
     * @param array<string, array<int, string>> $errors
     */
    public function __construct(array $errors = [])
    {
        foreach ($errors as $field => $messages) {
            foreach ($messages as $message) {
                $this->fieldErrors[] = (new FieldError())
                    ->setField($field)
                    ->setMessage($message);
            }
        }
        parent::__construct();
    }
}
```

Zdrojový kód 48. Třída pro vytvoření odpovědi 400 (Bad Request) [vlastní zpracování]

Zbývající chybové hlášky jsou odchyceny až během zpracování požadavku. Pro zlepšení závislosti třídy controller na chybových hláškách byly vytvořeny dvě factory, které samy rozhodují o odpovědi, která odpověď bude odeslána.

V případě chyby je odpověď vybrána podle status kódu. Tato factory je definována v kódu níže.

app/Http/Responses/Error/ErrorJsonResponseFactory.php

```php
<?php

declare(strict_types=1);

namespace App\Http\Responses\Error;

use App\Http\Responses\Error\Client\ConflictJsonResponse;
use App\Http\Responses\Error\Client\ForbiddenJsonResponse;
use App\Http\Responses\Error\Client\NotFoundJsonResponse;
use App\Http\Responses\Error\Client\UnauthorizedJsonResponse;
use App\Http\Responses\Error\Server\InternalServerErrorJsonResponse;

class ErrorJsonResponseFactory
{
    public static function send(
        int $statusCode
    ): AbstractErrorJsonResponse {
        return match ($statusCode) {
            ConflictJsonResponse::STATUS_CODE
            => new ConflictJsonResponse(),
            ForbiddenJsonResponse::STATUS_CODE
            => new ForbiddenJsonResponse(),
            NotFoundJsonResponse::STATUS_CODE
            => new NotFoundJsonResponse(),
            NotAllowedJsonResponse::STATUS_CODE
            => new NotAllowedJsonResponse(),
            UnauthorizedJsonResponse::STATUS_CODE
            => new UnauthorizedJsonResponse(),
            default => new InternalServerErrorJsonResponse(),
        };
    }
}
```

Zdrojový kód 49. Třída pro generování chybové odpovědi [vlastní zpracování]

Pokud je odpověď úspěšná, její typ se rozhoduje v závislosti na datech a jejich vytvoření, jako je tomu na následující ukázce.

app/Http/Responses/Success/SuccessfulJsonResponseFactory.php

```php
<?php

declare(strict_types=1);

namespace App\Http\Responses\Success;

class SuccessfulJsonResponseFactory
{
    public static function send(
        mixed $data = []
    ): AbstractSuccessfulJsonResponse {
        if (is_array($data) && count($data) === 0) {
            return new NoContentJsonResponse();
        } else {
            return new OkJsonResponse($data);
        }
    }

    public static function sendNew(
        mixed $data
    ): AbstractSuccessfulJsonResponse {
        return new CreatedJsonResponse($data);
    }
}
```

Zdrojový kód 50. Třída pro zpracování úspěšné odpovědi [vlastní zpracování]

Tělo controller metody pak může vypadat následovně:

app/Http/Controllers/UserController.php

```php
public function update(
    UpdateUserRequest $request,
    int $id
): JsonResponse {
    $user = $this->service->findById($id);
    if (!($user instanceof UserData)) {
        return ErrorJsonResponseFactory::send(
            JsonResponse::HTTP_NOT_FOUND
        );
    }

    $isUpdated = $this->service->updateUser(
        $user,
        UserData::fromRequest($request)
    );

    if ($isUpdated === false) {
        return ErrorJsonResponseFactory::send(
            JsonResponse::HTTP_CONFLICT
        );
    }
    return SuccessfulJsonResponseFactory::send();
}
```

Zdrojový kód 51. Příklad generování chyb v update metodě [vlastní zpracování]

3.3 Testování koncových bodů

V doporučení dokumentace frameworku je uvedeno, že vývojář by měl psát především testy vlastností než jednotkové testy. Testy vlastností API mají funkcionalitu testování samotných koncových bodů. Dochází tedy k testování jednotlivých odpovědí, jestli jsou validní.

Jako příklad je uvedený *test case* pro ověření koncového bodu pro získání seznamu lokací (Zdrojový kód 52).

tests\Feature\University\Location\ListLocationTest.php

```php
<?php

namespace Tests\Feature\University\Location;
...
class ListLocationTest extends TestCase
{
    public function testApiReturnsLocationsResponse(): void
    {
        Location::factory()->count(5)->create();

        $response = $this
            ->withHeader('Accept', 'application/json')
            ->withHeader('Content-Type', 'application/json')
            ->withHeader(
                'Authorization',
                $this->getSuperAdminToken()
            )->getJson('/api/locations');

        $response->assertStatus(Response::HTTP_OK);
    }

    public function testApiReturnsNotAcceptable(): void
    {
        $response = $this
            ->withHeader('Content-Type', 'application/json')
            ->withHeader(
                'Authorization',
                $this->getSuperAdminToken()
            )->get('/api/locations');

        $response->assertStatus(Response::HTTP_NOT_ACCEPTABLE);
    }

    public function testApiReturnsForbidden(): void
    {
        $response = $this
            ->withHeader('Accept', 'application/json')
            ->withHeader('Content-Type', 'application/json')
            ->withHeader('Authorization', $this->getStudentToken())
            ->getJson('/api/locations');

        $response->assertStatus(Response::HTTP_FORBIDDEN);
    }

    public function testApiReturnsUnauthorized(): void
    {
        $response = $this
            ->withHeader('Accept', 'application/json')
            ->withHeader('Content-Type', 'application/json')
            ->getJson('/api/locations');

        $response->assertStatus(Response::HTTP_UNAUTHORIZED);
    }
}
```

Zdrojový kód 52. Ukázka testu vlastností [vlastní zpracování]

4 IMPLEMENTACE

Laravel Framework využívá architektonický vzor MVC napojený na ORM (anglicky Object Relational Mapping) Eloquent, který slouží pro namapování dat z perzistentní vrstvy na objekty. Nicméně Laravel využívá ORM nejen jako datovou vrstvu ale napříč celou aplikací, od autorizace a validace požadavku až po zaslání zdrojů pro API. Tento přístup často usnadňuje práci, nicméně se příliš neslučuje s principy uvedenými v teoretické části.

4.1 Data Transfer Objects

První úpravou je použití Data Transfer Object, které budou využity v doménové vrstvě. Jde o princip, kdy tento objekt slouží právě jako přenosová vrstva mezi daty od uživatele a daty v databázi. Je vytvořeno rozhraní (Zdrojový kód 53), které definuje metody vytvoření objektu z požadavku, modelu a kolekce, toto rozhraní pak implementuje třída DTO, která zároveň definuje položky, které konkrétní model definuje.

app/Dto/DataInterface.php

```php
<?php

declare(strict_types=1);

namespace App\Dto;

use App\Http\Requests\RequestInterface;
use Illuminate\Database\Eloquent\Collection;
use Illuminate\Database\Eloquent\Model;
use JsonSerializable;

interface DataInterface extends JsonSerializable
{
    public static function fromRequest(
        RequestInterface $request
    ): DataInterface;

    public static function fromModel(Model $model): DataInterface;

    public static function fromCollection(
        Collection $collection
    ): array;
}
```

Zdrojový kód 53. Rozhraní pro DTO [vlastní zpracování]

4.2 Datová vrstva

Pro vytvoření datové vrstvy je použit návrhový vzor Repository, který přijímá modely z perzistentní vrstvy a posílá je v DTO do doménové vrstvy a naopak. Příklady implementace návrhového vzoru jsou níže.

app/Repositories/LocationRepositoryInterface.php

```php
<?php

declare(strict_types=1);

namespace App\Repositories;

use App\Dto\LocationData;

interface LocationRepositoryInterface
{
    public function createLocation(LocationData $data): LocationData;

    public function getAllLocations(): array;
}
```

Zdrojový kód 54. Příklad rozhraní návrhového vzoru Repository pro objekt Lokace [vlastní zpracování]

app/Repositories/Eloquent/LocationRepository.php

```php
<?php

declare(strict_types=1);

namespace App\Repositories\Eloquent;

use App\Dto\LocationData;
use App\Models\Location;
use App\Repositories\LocationRepositoryInterface;

class LocationRepository implements LocationRepositoryInterface
{
    public function createLocation(LocationData $data): LocationData
    {
        $locationModel = Location::create($data->toArray());

        return LocationData::fromModel($locationModel);
    }

    public function getAllLocations(): array
    {
        return LocationData::fromCollection(Location::all());
    }
}
```

Zdrojový kód 55. Příklad implementace návrhového vzoru Repository pro objekt Lokace [vlastní zpracování]

Aby došlo k propojení implementace s rozhraním je použita *Dependency Injection* pomocí vytvoření `RepositoryServiceProvider`, jak ukazuje následující Zdrojový kód 56.

app/Providers/RepositoryServiceProvider.php

```php
<?php

namespace App\Providers;

use App\Repositories\Eloquent\LocationRepository;
use App\Repositories\Eloquent\UserRepository;
use App\Repositories\LocationRepositoryInterface;
use App\Repositories\UserRepositoryInterface;
use Illuminate\Support\ServiceProvider;

class RepositoryServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->bind(
            LocationRepositoryInterface::class,
            LocationRepository::class
        );
        ...
    }
...
}
```

Zdrojový kód 56. Implementace `RepositoryServiceProvider` [vlastní zpracování]

4.3 Doménová vrstva

Jako doménová vrstva slouží *service*. Ta přijímá a posílá DTO, které podle potřeby business logiky upravuje. *Service* je umístěna mezi *Controller* a *Repository*. Jako příklad je uvedena jednoduchá *service* pro `Location`, v tomto případě jde pouze o předání dat. Nicméně v kapitole autentizace bude ukázka *service* se složitější logikou.

app/Services/LocationService.php

```php
<?php

declare(strict_types=1);

namespace App\Services;

use App\Dto\LocationData;
use App\Repositories\LocationRepositoryInterface;

class LocationService
{
    protected LocationRepositoryInterface $locationRepository;

    public function __construct(
        LocationRepositoryInterface $locationRepository
    ) {
        $this->locationRepository = $locationRepository;
    }

    public function createLocation(
        LocationData $locationData
    ): LocationData {
        return $this->locationRepository->createLocation(
            $locationData
        );
    }

    public function listLocation(): array
    {
        return $this->locationRepository->getAllLocations();
    }
}
```

Zdrojový kód 57. Implementace `LocationService` [vlastní zpracování]

4.4 Aplikační vrstva

Díky použitým návrhovým vzorům a principu jediné odpovědnosti došlo k výraznému zredukování logiky nejen pro model ale i pro *controller*. Na následující straně je definován *controller* pro uložení zdroje `Location` a zobrazení jeho seznamu.

app/Http/Controllers/LocationController.php

```php
<?php

namespace App\Http\Controllers;

use App\Dto\LocationData;
use App\Http\Requests\ListLocationRequest;
use App\Http\Requests\StoreLocationRequest;
use App\Http\Responses\Success\SuccessfulJsonResponseFactory;
use App\Services\LocationService;
use Illuminate\Http\JsonResponse;

class LocationController extends Controller
{
    protected LocationService $service;

    public function __construct(LocationService $service)
    {
        $this->service = $service;
    }

    public function list(): JsonResponse
    {
        return SuccessfulJsonResponseFactory::send(
            $this->service->listLocation()
        );
    }

    public function store(
        StoreLocationRequest $request
    ): JsonResponse {
        $location = $this->service->createLocation(
            LocationData::fromRequest($request)
        );

        return SuccessfulJsonResponseFactory::sendNew($location);
    }
}
```

Zdrojový kód 58. Implementace `LocationController` [vlastní zpracování]

V tuto chvíli má controller na starosti pouze transformaci dat od uživatel do DTO a zavolání service, která vykoná požadovaný dotaz.

Data jsou před vstupem do metody *controller* zvalidována pomocí vlastní sady pravidel a dochází zde i k autorizaci. Další informace o autorizaci budou v kapitole 6.2.

routes/app/Http/Requests/StoreLocationRequest.php

```php
<?php

namespace App\Http\Requests;

use App\Exceptions\ForbiddenException;
use App\Exceptions\UnauthorizedException;
use App\Models\Location;
use App\Models\User;

/**
 * @property int $id
 * @property string $city
 * @property string $country
 * @property string $continent
 */
class StoreLocationRequest extends ApiRequest
    implements LocationRequestInterface
{
    public function rules(): array
    {
        return [
            'city' => 'required|string',
            'country' => 'required|string',
            'continent' => 'required|string'
        ];
    }
}
```

Zdrojový kód 59. Validace dat `Location` před vstupem do controller metody [vlastní zpracování]

5 PROVĚŘENÍ KÓDU

Již při inicializaci projektu je vhodné zavést do projektu nástroje pro prověření kódu. Čím dříve jsou nástroje zavedeny tím více je kód pod kontrolou.

5.1 Statická analýza

Nejjednodušší, co může vývojář pro kontrolu svého kódu udělat, je použít nástroje statické analýzy. Laravel má již dostupný svůj nástroj pro statickou analýzu, který je již v projektu zahrnut během instalace, s názvem Pint. Tento nástroj analyzuje kód pomocí zadaného přednastavení a následně i automaticky opravuje chyby stylistického charakteru. Přednastavení je defaultně přímo pro Laravel, ale zvolit i jiné nástroje, jako je PSR-12. Ukázka použití nástroje na projekt je zobrazena níže.

```
vendor/bin/pint --preset psr12
✓✓✓✓✓✓.....................................................
.........................................
———————————————————————————————————————————————— PSR 12
FIXED   ..................... 107 files, 6 style issues fixed
✓ .../...create_users_table.php                  class_definition
✓ .../...create_password_reset_tokens_table.php  class_definition
✓ .../...create_failed_jobs_table.php            class_definition
✓ .../...create_personal_access_tokens_table.php class_definition
✓ .../...add_role_to_users_table.php             new_with_braces,
                                                class_definition
✓ .../...create_locations_table.php              new_with_braces,
                                                class_definition
```

Zdrojový kód 60. Výstup z nástroje Pint [vlastní zpracování]

Klasickými a zažitými nástroji, které lze použít, jsou například PHP Code Sniffer, PHP Mess Detector. Níže je příklad výstupu PHP Code Sniffer a na následující straně PHP Mess Detector.

```
> phpcs --standard=PSR12 app tests

FILE: app/Http/Controllers/UserController.php
----------------------------------------------------------------------
FOUND 0 ERRORS AND 1 WARNING AFFECTING 1 LINE
----------------------------------------------------------------------
 94 | WARNING | Line exceeds 120 characters; contains 151 characters
----------------------------------------------------------------------

FILE: app/Http/Responses/CreatedJsonResponse.php
----------------------------------------------------------------------
FOUND 1 ERROR AFFECTING 1 LINE
----------------------------------------------------------------------
 19 | ERROR | [x] The closing brace for the class must go on the next
 line after the body
----------------------------------------------------------------------
PHPCBF CAN FIX THE 1 MARKED SNIFF VIOLATIONS AUTOMATICALLY
----------------------------------------------------------------------
Time: 380ms; Memory: 10MB
```

Zdrojový kód 61. Příklad výstupu PHP Code Sniffer [vlastní zpracování]

```
> phpmd app,tests ansi phpmd.xml

FILE: app/Http/Controllers/UserController.php
----------------------------------------------------------------------
 25 | VIOLATION | The class UserController has a coupling between
 objects value of 14. Consider to reduce the number of dependencies
 under 13.

Found 1 violation and 0 errors in 686ms
```

Zdrojový kód 62. Příklad výstupu PHP Mess Detector [vlastní zpracování]

Pro důslednější kontrolu kódu slouží nástroj PHPStan, který byl založen českým vývojářem Ing. Ondřejem Mirtesem v roce 2016. PHPStan má deset úrovní, kdy vyšší úroveň obsahuje všechny kontroly té nižší.

Při nastavení na již rozběhnutý projekt je v pořádku nastavit úroveň na 0 a postupně zvyšovat během oprav. Při použití na nový projekt je na vývojáři, kterou úroveň si nastaví, ale samozřejmě čím vyšší, tím lepší.

Již zmíněná nultá úroveň obsahuje fatální chyby, které nelze ignorovat a měly by být opraveny co nejrychleji – jedná se o neznámé třídy, funkce, metody, chybějící argumenty a nikdy neinicializované proměnné.

Následující úrovně obsahují kontroly magických metod, datových typů, mrtvého kódu atp.

5.2 Dynamická analýza

Pro zajištění kontroly nad kódem, je vhodné mít v projektu nastavený krokový *debugging*, který byl zmíněn v kapitole 1.9.2.1.

Pro PHP slouží k tomuto prověření nástroj xDebug, který lze do projektu zavést. V Laravel projektu je zaveden xDebug pomocí přidání řádku do `.env` souboru jako na následující ukázce.

.env

```
...
SAIL_XDEBUG_MODE=debug
...
```

Zdrojový kód 63. Přidání xDebug do Laravel projektu [79]

Jestliže je k projektu používán Docker, není za potřebí žádné další modifikace. Po tomto je nutné znovu spustit kontejner, aby se nastavily provedené úpravy.

Další kroky jsou závislé na používané vývojovém prostředí. Při použití PHPStormu dojde k zapnutí sluchátka, díky kterému bude spuštěno odposlouchávání připojení k xDebugu.

Následně je potřeba nastavit druhá strana komunikace. Existuje více způsobů, jak lze komunikaci inicializovat. Jednou možností je nastavení xDebug cookie, jehož definice je znázorněna na další straně.

```http
Cookie: XDEBUG_SESSION=start
```

Zdrojový kód 64. Definice xDebug cookie [80]

To lze udělat u požadavku pomocí uživatelského rozhraní. Pro možnost debuggování všech endpointů je ale výhodnější nastavit cookie globálně. K tomu bude využit tzv. *pre-request script* definován přímo na kolekci, díky kterému je skript spouštěn před každým provedeným požadavkem. Tento skript je definován následovně:

Pre-request skript

```
const cookieJar = pm.cookies.jar();
const xdebugCookieKey = 'XDEBUG_SESSION';
const url = pm.environment.get("baseUrl");
const xDebugCookie = {
    name: xdebugCookieKey,
    value: 'start',
    httpOnly: true
}

cookieJar.set(url, xDebugCookie, function (error) {
    if (error) {
        console.log(error);
    }
});
```

Zdrojový kód 65. Pre-request skript pro vložení xDebug cookie [vlastní zpracování]

Poté je ještě důležité přidat server k povoleným doménám. To lze udělat ve správě cookies u requestu, kde je aktuální přidána doména (stejná jako `baseUrl`).

6 ZAJIŠTĚNÍ BEZPEČNOSTI

V této práci bude zmíněna pro zabezpečení metody autentizace a autorizace spolu se zabezpečením hesel.

6.1 Autentizace

Pro autentizaci byla využita knihovna Sanctum, která pro API využívá Bearer token. K tomuto je projekt připraven. V migracích se již nachází příkaz pro vytvoření tabulky pro tokeny `2019_12_14_000001_create_personal_access_tokens_table.php`, takže je třeba pouze spustit migraci.

```
> sail artisan migrate
```

Dalším krokem je přidání middleware do `App/Http/Kernel` jak ukazuje Zdrojový kód 66 na řádku 19.

app/Http/Kernel.php

```
1  <?php
2
3  namespace App\Http;
4
5  use Illuminate\Foundation\Http\Kernel as HttpKernel;
6  use Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful;
7
8  class Kernel extends HttpKernel
9  {
10      ...
11
12     /**
13      * The application's route middleware groups.
14      *
15      * @var array<string, array<int, class-string|string>>
16      */
17     protected $middlewareGroups = [
18         'api' => [
19             EnsureFrontendRequestsAreStateful::class,
20             ...
21         ],
22     ...
23 }
```

Zdrojový kód 66. Aktivace middleware pro Sanctum [vlastní zpracování]

Poté je třeba naimplementovat autorizační *controller*, ve kterém jsou implementovány operace pro registraci, přihlášení a odhlášení, který se napojí na vytvořenou *service*

`AuthService`. Následující ukázka představuje metody pro registraci uživatele a na následující straně jsou ukázky pro přihlášení a odhlášení.

app/Services/AuthService.php

```php
<?php

declare(strict_types=1);

namespace App\Services;

use ...

class AuthService
{
    protected UserRepositoryInterface $userRepository;

    protected HashAdapter $hashAdapter;

    protected ?UserData $loggedUser = null;

    ...

    public function registerUser(
        UserData $user
    ): AuthenticatedUserData {
        $this->hashUsersPassword($user);
        $user = $this->userRepository->create($user);
        $token = $this->userRepository->createToken($user);
        return new AuthenticatedUserData($user, $token);
    }

    protected function hashUsersPassword(UserData $user): void
    {
        if (is_null($user->getPassword())) {
            throw new InternalServerErrorException(
                'Expected a password to hash.'
            );
        }

        $hashedPassword = $this->hashAdapter->createHash(
            $user->getPassword()
        );
        $user->setPassword($hashedPassword);
    }
    ...
}
```

Zdrojový kód 67. Metody registrace autorizační service [vlastní zpracování]

app/Services/AuthService.php

```php
<?php
...
class AuthService
{
    ...

    public function loginUser(
        UserData $loginRequest
    ): ?AuthenticatedUserData {
        if (!is_string($loginRequest->getEmail())) {
            throw new Exception('Expected email for login.');
        }
        $this->loggedUser = $this->userRepository->findByEmail(
            $loginRequest->getEmail()
        );

        if (!(
             ($this->loggedUser instanceof UserData)
            && $this->confirmCredentials($loginRequest)
        )) {
            return null;
        }
        $token = $this->userRepository->createToken(
            $this->loggedUser
        );

        return new AuthenticatedUserData($this->loggedUser, $token);
    }

    protected function confirmCredentials(
        UserData $enteredCredentials
    ): bool {
        if (!is_string($enteredCredentials->getPassword())) {
            throw new InternalServerErrorException(...);
        }
        if ($this->loggedUser instanceof UserData) {
            $this->hashAdapter->setHashedValue(
                $this->loggedUser->getPassword()
            );
            return $this->hashAdapter->matchesWithHashedValue(
                $enteredCredentials->getPassword()
            );
        }
        throw new Exception('Expected Logged user data.');
    }

    public function logoutUser(UserData $authenticatedUser): true
    {
        $this->userRepository->deleteToken($authenticatedUser);
        return true;
    }
}
```

Zdrojový kód 68. Metody přihlášení a odhlášení autorizační service [vlastní zpracování]

Metody `AuthService` se odvolávají na `UserRepository`, kde dochází k manipulací s databází. Ukázky kódu představuje Zdrojový kód 69. Autentizační metody v Repository.

app/Repositories/Eloquent/UserRepository.php

```
...
class UserRepository implements UserRepositoryInterface
{
    public function create(UserData $user): UserData
    {
        $userModel = new User();
        $userModel->firstName = $user->getFirstName();
        $userModel->lastName = $user->getLastName();
        $userModel->email = $user->getEmail();
        $userModel->password = $user->getPassword();
        if ($userModel->save() !== true) {
            throw new InternalServerErrorException(
                'Could not save the user.'
            );
        }
        return UserData::fromModel($userModel);
    }

    public function createToken(UserData $user): string
    {
        $userModel = User::find($user->getId());
        if (!($userModel instanceof User)) {
            throw new Exception(
                'Could not generate token. User was not found.'
            );
        }
        return $userModel->createToken('apiToken')->plainTextToken;
    }

    public function deleteToken(UserData $user): void
    {
        $userModel = User::find($user->getId());
            if (!($userModel instanceof User)) {
                throw new Exception(
                    'Could not delete token. User was not found.'
                );
            }
        $userModel->tokens()->delete();
    }
}
```

Zdrojový kód 69. Autentizační metody v Repository [vlastní zpracování]

6.2 Autorizace

Pro zajištění autorizace jsou ochráněny koncové body pomocí Sanctum middleware. Obecně je Laravel nastaven pro autorizaci buďto při validaci požadavku nebo v metodě controller. Nicméně kvůli principu jediné zodpovědnosti je vhodné nechat autorizaci na autorizačním

middleware, který se postará o kontrolu přístupu uživatele. Autorizace je definována pomocí metody `can()` na danou route jako předvádí Zdrojový kód 70.

routes/api.php

```php
<?php

Route::middleware('auth:sanctum')
    ->group(function () {
        Route::controller(LocationController::class)
            ->group(function () {
                Route::post('/locations', 'store')
                    ->can('create', Location::class);
                Route::get('/locations', 'list')
                    ->can('viewAny', Location::class);
            });
        Route::controller(UserController::class)
            ->group(function () {
                Route::get('/users', 'list')
                    ->can('viewAny', User::class);
                Route::get('/users/{id}', 'detail')
                    ->can('view', [User::class, 'id']);
                Route::patch('/users/{id}', 'update')
                    ->can('update', [User::class, 'id']);
                Route::delete('/users/{id}', 'delete')
                    ->can('delete', [User::class, 'id']);
            });
    });
```

Zdrojový kód 70. Vyžádání autentizace pro koncové body [vlastní zpracování]

Pro definici autorizačních pravidel slouží v Laravel frameworku Policy. Tam lze pomocí vytvoření metod definovat pravidla přístupu, která budou ověřena. Pravidla mohou být založena na roli, na vztahu i na atributu.

Hlavní pravidla jsou definována v nově vytvořené rodičovské třídě v následujícím zdrojovém kódu.

App/Policies/AbstractPolicy.php

```php
<?php

declare(strict_types=1);

namespace App\Policies;

use App\Models\User;

abstract class AbstractPolicy
{
    public function isUserSuperAdmin(User $user): bool
    {
        return $user->role === User::SUPER_ADMIN;
    }

    public function isUserAdmin(User $user): bool
    {
        return $user->role === User::ADMIN;
    }

    public function isUserOwner(User $user, int $ownerId): bool
    {
        return $user->id === $ownerId;
    }
}
```

Zdrojový kód 71. Rodičovská třída Policy [vlastní zpracování]

Policy lze vytvořit pomocí artisan příkazu.

```
> sail artisan make:policy UserPolicy --model=User
```

Následně je vyžádána úprava třídy pro dědění od `AbstractPolicy` a v ukázce Zdrojový kód 72 byly metody upraveny tak, aby místo modelu zdroje ověřovala jeho identifikátor.

app/Policies/UserPolicy.php

```php
<?php

namespace App\Policies;

use App\Models\User;

class UserPolicy extends AbstractPolicy
{
    public function viewAny(User $user): bool
    {
        return $this->isUserSuperAdmin($user);
    }

    public function view(User $user, int $id): bool
    {
        return $this->isUserOwner($user, $id)
            || $this->isUserSuperAdmin($user);
    }
}
```

Zdrojový kód 72. Pravidla pro manipulaci s uživatelem [vlastní zpracování]

V případě úprav odpovědí jako tomu bylo v této práci je pro správnou funkčnost nutné ještě přetížit middleware definovaný autorizaci. Ten totiž právě v případě použití znázorněném ve Zdrojový kód 70 generuje výjimku s neplatným kódem a dochází k narušení chování aplikace. Aby tomuto bylo zabráněno, jsou výjimky odchyceny a nahrazeny za vlastní jako je znázorněno v ukázce Zdrojový kód 73. Poté je nutné tento middleware ještě vyměnit v `App/Http/Kernel`, kde je definován jako alias `'can'`.

app/Http/Middleware/Authorize.php

```php
<?php

namespace App\Http\Middleware;

use App\Exceptions\ForbiddenException;
use App\Exceptions\UnauthorizedException;
use Closure;
use Illuminate\Auth\Access\AuthorizationException;
use Illuminate\Auth\Middleware\Authorize as LaravelAuthorize;
use Illuminate\Http\Request;

class Authorize extends LaravelAuthorize
{
    public function handle(
        $request,
        Closure $next,
        $ability,
        ...$models
    ) {
        try {
            $this->gate->authorize(
                $ability,
                $this->getGateArguments($request, $models)
            );
        } catch (AuthorizationException) {
            throw new ForbiddenException();
        }

        return $next($request);
    }
}
```

Zdrojový kód 73. Přetížení `Authorize` middleware [vlastní zpracování]

6.3 Bezpečnost hesel

Zabezpečení uloženého hesla je zajištěno pomocí hash algoritmu Argon2id, který byl zmíněn v teoretické části práce 2.1.1.5 Bezpečné algoritmy. Tento algoritmus lze v Laravel použít pomocí fasády Hash, ze které si lze vybrat algoritmus pro hashing jako je Bcrypt, Argon, Argon2id.

Na následující stránce je definována třída pomocí návrhového vzoru Adapter z kapitoly 1.6.2.1, která definuje metodu pro vytvoření hash řetězce a metodu pro kontroly shodnosti uloženého hash řetězce s hodnotou v otevřeném textu.

app/Adapters/HashAdapter.php

```php
<?php

declare(strict_types=1);

namespace App\Adapters;

use Exception;
use Illuminate\Hashing\Argon2IdHasher;
use Illuminate\Support\Facades\Hash;

class HashAdapter
{
    private Argon2IdHasher $hasher;

    protected ?string $hashedValue = null;

    public function __construct(?string $hashedValue = null)
    {
        $this->hasher = Hash::createArgon2idDriver();
        $this->hashedValue = $hashedValue;
    }

    public function createHash(string $plainValue): string
    {
        return $this->hasher->make($plainValue);
    }

    public function matchesWithHashedValue(string $plainValue): bool
    {
        if (is_null($this->hashedValue)) {
            throw new Exception('Hashed value was not initialised.');
        }
        return $this->hasher->check($plainValue, $this->hashedValue);
    }
}
```

Zdrojový kód 74. Hash adapter pro zabezpečení hesel [vlastní zpracování]

7 DOPORUČENÁ LITERATURA VÝVOJÁŘE

Na závěr tato práce zmíní doporučené literární zdroje, které mohou pomoci vývojáři se zdokonalením v psaní kódu, testování, vytváření uživatelského rozhraní nebo zabezpečení.

7.1 Čistý kód

Robert C. Martin: *Clean Code*

Robert C. Martin: *The Clean Coder*

Robert C. Martin: *Clean Architecture*

Andrew Hunt a David Thomas: *The Pragmatic Programmer*

Michael C. Feathers: *Working effectively with legacy code*

Martin Fowler: *Refactoring: Improving the Design of Existing Code*

7.2 Testování

Vladimir Khorikov: *Unit Testing Principles, Practices, and Patterns*

Steve Freeman a Nat Pryce: *Growing Object-Oriented Software, Guided by Tests*

Kent Beck: *Test-Driven Development: By Example*

7.3 Design

Steve Krug: *Don't Make Me Think: A Common Sense Approach to Web Usability*

Jesse James Garrett: *The Elements of User Experience: User-Centered Design for the Web and Beyond*

Stephen Anderson: *Seductive Interaction Design: Creating Playful, Fun, and Effective User Experiences*

Alan Cooper, Robert Reimann, a David Cronin: *About Face: The Essentials of Interaction Design*

Jenifer Tidwell: *Designing Interfaces: Patterns for Effective Interaction Design*

7.4 Zabezpečení

Dafydd Stuttard a Marcus Pinto*: The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*

Bryan Sullivan a Vincent Liu: *Web Application Security: A Beginner's Guide*

Michal Zalewski: *The Tangled Web: A Guide to Securing Modern Web Application*

Jean-Philippe Aumasson: *Serious Cryptography: A Practical Introduction to Modern Encryption*

Dan Bergh Johnsson, Daniel Deogun, a Daniel Sawano: *Secure by Design*

7.5 Obecné knihy

John Sonmez: *The Software Developer's Life Manual*

Frederick Brooks: *The Mythical Man-Month*

Dafydd Stuttard a Marcus Pinto: *The Web Application Hacker's Handbook*

Gayle Laakmann McDowell: *Cracking the Coding Interview*

Jon Bentley: *Programming Pearls*

Steve McConnell: *Code Complete: A Practical Handbook of Software Construction*

ZÁVĚR

Práce zpracovává témata věnující se kvalitní implementaci. V teoretické části jsou definovány základní praktiky čistého kódu jako je formátování, jmenné konvence a vývojové principy, které pomáhají udržet přehlednost projektu. Dále jsou popsány pojmy z vývoje softwaru, které se věnují způsobu vývoje. Ve vývojových strategií byly zmíněny strategie TDD, BDD, SDD, FDD a API-first development. V organizaci repozitáře je definován rozdíl mezi organizací monorepo a polyrepo. Architektury softwaru představují monolitickou architekturu, mircroservices, SOA a Serverless. V architektonických vzorech je popsán rozdíl mezi MVC, MVP a MVVM. Kapitola návrhových vzorů definuje zástupce z vytvářejících, strukturálních a návrhových vzorů chování.

Důležitými kapitolami je především RESTful rozhraní pro programování aplikací a práce s verzovacím systémem, kde byly navrženy konkrétní postupy pro kvalitní vývoj. V kapitole věnující se vývoji API jsou definovány scénáře stavových kódů a byly vytvořeny diagramy pro jednoduché znázornění možných stavů. Následuje sada doporučení pro implementaci API. Kapitola věnující se verzovacímu systému popisuje vhodné praktiky jak pracovat s Gitem.

V neposlední řadě se teoretická část věnuje ověření kódu, jehož obsahem je statická a dynamická analýza kódu a testování prováděné vývojáři, a zabezpečení aplikací, které zmiňuje témata jako jsou hash algoritmy, autentizace a autorizace, správa chyb a bezpečnostní organizace (OWASP a CWE) a regulace (GDPR a cookies).

V praktické části jsou poznatky implementovány na inicializaci API projektu za použití PHP frameworku Laravel a aplikace Postman pro specifikaci API. Praktická část obsahuje podrobný výklad inicializace projektu pro API a definování čisté architektury a nastavení správných praktik. Na začátku je popsáno použití API-first vývoje k vytvoření specifikace API a definování jednotlivých koncových bodů. V této části je ukázka přizpůsobení projektu pro zasílání JSON odpovědí a způsob definování MVCS architektury a použití návrhových vzorů jako je Repository, Factory a Adapter.

SEZNAM POUŽITÉ LITERATURY

[1] GREGORI, Sven. Ask Hackaday: Are 80 Characters Per Line Still Reasonable In 2020?. *Hackaday* [online]. Pasadena (California): Supplyframe, c2023, June 18, 2020 [cit. 2023-05-26]. Dostupné z: https://hackaday.com/2020/06/18/ask-hackaday-are-80-characters-per-line-still-reasonable-in-2020

[2] FELICIOTTI, Andy. Case Styles in Programming: 5 Common Programming Naming Conventions Explained. *TitleCapitalize* [online]. TitleCapitalize.com, c2023, May 17, 2022 [cit. 2023-05-26]. Dostupné z: https://titlecapitalize.com/programming-case-styles/

[3] SINGH, Sarika. What is the meaning of single underscore prefix with Python variables?. *Tutorials Point* [online]. Hyderabad: Tutorials Point India Private Limited, c2023, 23 Nov 2022 [cit. 2023-05-26]. Dostupné z: https://www.tutorialspoint.com/What-is-the-meaning-of-single-underscore-prefix-with-Python-variables

[4] Spaceflight Participant Charles Simonyi. In: *National Aeronautics and Space Administration* [online]. Washington, DC: NASA, b. r. [cit. 2023-05-26]. Dostupné z: https://www.nasa.gov/pdf/163534main_simonyi.pdf

[5] SPOLSKY, Joel. Making Wrong Code Look Wrong. Joel on Software [online]. New York: Spolsky, b. r., May 11, 2005 [cit. 2023-05-26]. Dostupné z: https://www.joelonsoftware.com/2005/05/11/making-wrong-code-look-wrong/

[6] MARTIN, Robert C. *Clean code: A Handbook of Agile Software Craftmanship*. Stoughton (Massachusetts): Pearson Education, c2009. Robert C. Martin series. ISBN 978-013-2350-884.

[7] OLORUNTOBA, Samuel. SOLID: The First 5 Principles of Object Oriented Design. DigitalOcean [online]. Lenexa (Kansas): DigitalOcean, c2023, November 30, 2021 [cit. 2023-05-26]. Dostupné z: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

[8] LISKOV, Barbara H. a Jeannette M. WING. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*. 1994, **16**(6), 1811-1841. ISSN 0164-0925. Dostupné z: doi:10.1145/197320.197383

[9] UTOSU, Kwerenachi. The K.I.S.S Principle in Programming. DEV Community [online]. New York: DEV Community, c2016 - 2023, 15. 7. 2020 [cit. 2023-05-26]. Dostupné z: https://dev.to/kwereutosu/the-k-i-s-s-principle-in-programming-1jfg

[10] Rules of Simplicity: Definition. *Agile Alliance* [online]. Dallas (Texas): Agile Alliance, c2023 [cit. 2023-05-26]. Dostupné z: https://www.agilealliance.org/glossary/rules-of-simplicity/

[11] THOMAS, David a Andrew HUNT. *The pragmatic programmer: your journey to mastery*. 20th anniversary edition. Boston: Addison-Wesley, [2020]. ISBN 978-0135957059.

[12] HUNTER, Tatum, URWIN, Matthew, ed. YAGNI (You Aren't Gonna Need It) Principle Helps Devs Stay Efficient: Why build something now when you can put it off forever?. *Built In: United We Tech.* [online]. Chicago (Illinois): Built In, c2023, Oct 03, 2022 [cit. 2023-05-26]. Dostupné z: https://builtin.com/software-engineering-perspectives/yagni

[13] ARORA, Nipun. Key principles in Software — DRY, KISS, YAGNI, SOLID and other Acronyms: Separation of Concerns (SoC). *Dev Genius: Coding, Tutorials, News, UX, UI and much more related to development*[online]. San Francisco: A Medium Corporation, b. r., Mar 27, 2022 [cit. 2023-05-26]. Dostupné z: https://blog.devgenius.io/key-principles-in-software-dry-kiss-yagni-solid-and-other-acronyms-98e5575a6942

[14] TDD. *Agile Alliance* [online]. Dallas (Texas): Agile Alliance, c2023 [cit. 2023-05-26]. Dostupné z: https://www.agilealliance.org/glossary/tdd

[15] WAGNER, Janet. Understanding the API-First Approach to Building Products. *Swagger* [online]. Somerville: SmartBear Software, c2023 [cit. 2023-05-26]. Dostupné z: https://swagger.io/resources/articles/adopting-an-api-first-approach/

[16] CSER, Tamas. Behavior-driven development. *Functionize* [online]. Walnut Creek: Functionize, c2023, April 6, 2023 [cit. 2023-05-26]. Dostupné z: https://www.functionize.com/automated-testing/behavior-driven-development

[17] Feature Driven Development (FDD). *ProductPlan*[online]. Santa Barbara: ProductPlan, c2023 [cit. 2023-05-26]. Dostupné z: https://www.productplan.com/glossary/feature-driven-development

[18] HASELAARS, Tim. An introduction to spec-driven API development. *Apideck* [online]. Antwerp: Apideck, c2023 [cit. 2023-05-26]. Dostupné z: https://blog.apideck.com/spec-driven-development-part-1

[19] Monorepo vs. polyrepo. In: *Monorepo.tools* [online]. Gilbert (Arizona): Narwhal Technologies, c2022 [cit. 2023-05-26]. Dostupné z: https://monorepo.tools

[20] FERNANDEZ, Tomas. What is monorepo? (and should you use it?). *Semaphore* [online]. Novi Sad: Rendered Text, c2023, 15 Jul 2022 [cit. 2023-05-26]. Dostupné z: https://semaphoreci.com/blog/what-is-monorepo

[21] AWATI, Rahul a Ivy WIGMORE. Monolithic architecture. *TechTarget* [online]. Newton (Massachuttes): TechTarget, c1999-2023, May 2022 [cit. 2023-05-26]. Dostupné z: https://www.techtarget.com/whatis/definition/monolithic-architecture

[22] JOHNSON, Jonathan a Laura SHIFF. What Is Microservice Architecture? Microservices Explained. *Bmc* [online]. Houston (Texas): BMC Software, c2005-2023, March 8, 2021 [cit. 2023-05-26]. Dostupné z: https://www.bmc.com/blogs/microservices-architecture/

[23] What is service-oriented architecture (SOA)?. *IBM*[online]. New York: IBM Corporation 1994, c2023 [cit. 2023-05-26]. Dostupné z: https://www.ibm.com/topics/soa

[24] IBRYAM, Bilgin. Microservices in a Post-Kubernetes Era. In: *InfoQ* [online]. Ontario: C4Media, c2006-2023, Sep 01, 2018 [cit. 2023-05-26]. Dostupné z: https://www.infoq.com/articles/microservices-post-kubernetes

[25] TAKYAR, Akash. What is serverless architecture and how it works?. *LeewayHertz* [online]. San Francisco: LeewayHertz, c2023 [cit. 2023-05-26]. Dostupné z: https://www.leewayhertz.com/what-is-serverless-architecture/

[26] What Are the 5 Primary Layers in Software Architecture?. *Indeed* [online]. Austin (Texas): Indeed, c2023, March 11, 2023 [cit. 2023-05-26]. Dostupné z: https://www.indeed.com/career-advice/career-development/what-are-the-layers-in-software-architecture

[27] HERNANDEZ, Rafael D. The Model View Controller Pattern – MVC Architecture and Frameworks Explained. *FreeCodeCamp* [online]. Oakland (California): Free Code Camp, April 19, 2021 [cit. 2023-05-26]. Dostupné z: https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/

[28] KUMAR, Amit. Understanding Model-View-Presenter. *Medium* [online]. San Francisco: Medium Corporation, Jun 2, 2017 [cit. 2023-05-26]. Dostupné z: https://medium.com/@amitkma/understanding-model-view-presenter-1d79448d39cd

[29] DAJBYCH, Václav. Mvvm: model-view-viewmodel. DotNETportal.cz [online]. Herceg, c2023, 21.04.2009 [cit. 2023-05-26]. Dostupné z: https://www.dotnetportal.cz/clanek/4994/MVVM-Model-View-ViewModel

[30] BÖHMER, Marian. *Návrhové vzory v PHP: [23 vzorových postupů pro rychlejší vývoj]*. Brno: Computer Press, 2012. ISBN 978-802-5133-385.

[31] SHVETS, Alexander. Singleton. *Refactoring.Guru*[online]. Pamplona: Refactoring.Guru, c2014-2023 [cit. 2023-05-26]. Dostupné z: https://refactoring.guru/design-patterns/singleton

[32] SONI, Devin. What Is a Singleton?. *Medium* [online]. San Francisco: Medium Corporation, Jul 31, 2019 [cit. 2023-05-26]. Dostupné z: https://betterprogramming.pub/what-is-a-singleton-2dc38ca08e92

[33] *Design Pattern – Factory Pattern* [online]. *Tutorials Point* [online]. Hyderabad: Tutorials Point India Private Limited, c2023 [cit. 2023-05-26]. Dostupné z: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

[34] SHVETS, Alexander. Adapter. *Refactoring.Guru*[online]. Pamplona: Refactoring.Guru, c2014-2023 [cit. 2023-05-26]. Dostupné z: https://refactoring.guru/design-patterns/adapter

[35] SHVETS, Alexander. Adapter. *Refactoring.Guru*[online]. Pamplona: Refactoring.Guru, c2014-2023 [cit. 2023-05-26]. Dostupné z: https://refactoring.guru/design-patterns/decorator

[36] SHVETS, Alexander. Adapter. *Refactoring.Guru*[online]. Pamplona: Refactoring.Guru, c2014-2023 [cit. 2023-05-26]. Dostupné z: https://refactoring.guru/design-patterns/proxy

[37] SHVETS, Alexander. Adapter. *Refactoring.Guru*[online]. Pamplona: Refactoring.Guru, c2014-2023 [cit. 2023-05-26]. Dostupné z: https://refactoring.guru/design-patterns/facade

[38] PRUDHOMME, Antonie. How we started using the repository pattern. *Spendesk: Engineering Blog*[online]. Spendesk, 2019.11.24 [cit. 2023-05-26]. Dostupné z: https://engineering.spendesk.com/posts/repository-pattern-at-spendesk/

[39] [Design Pattern]: The Observer Pattern. In: That's the way [online]. Blogspot [cit. 2023-05-26]. Dostupné z: https://thatsthaway.blogspot.com/2011/01/design-pattern-observer-pattern.html

[40] SHVETS, Alexander. Adapter. *Refactoring.Guru*[online]. Pamplona: Refactoring.Guru, c2014-2023 [cit. 2023-05-26]. Dostupné z: https://refactoring.guru/design-patterns/iterator

[41] What Is A RESTful API? *AWS* [online]. Seattle, Washington: Amazon Web Services, c2023 [cit. 2023-05-26]. Dostupné z: https://aws.amazon.com/what-is/restful-api/

[42] MARTIN, Ekuan, Jasoun BOUSKA, Mick ALBERTS, et al. RESTful web API design: What is REST? *Microsoft | Learn* [online]. Microsoft, c2023, 03/28/2023 [cit. 2023-05-26]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design#what-is-rest

[43] MARTIN, Ekuan, Jasoun BOUSKA, Mick ALBERTS, et al. RESTful web API design: Conform to HTTP semantics. *Microsoft | Learn* [online]. Microsoft, c2023, 03/28/2023 [cit. 2023-05-26]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design#conform-to-http-semantics

[44] MARTIN, Ekuan, Jasoun BOUSKA, Mick ALBERTS, et al. RESTful web API design: Use HATEOAS to enable navigation to related resources. *Microsoft | Learn* [online]. Microsoft, c2023, 03/28/2023 [cit. 2023-05-26]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design#use-hateoas-to-enable-navigation-to-related-resources

[45] MARTIN, Ekuan, Jasoun BOUSKA, Mick ALBERTS, et al. RESTful web API design: Support partial responses for large binary resources. *Microsoft | Learn* [online]. Microsoft, c2023, 03/28/2023 [cit. 2023-05-26]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design#support-partial-responses-for-large-binary-resources

[46] MARTIN, Ekuan, Jasoun BOUSKA, Mick ALBERTS, et al. RESTful web API design: Asynchronous operations. *Microsoft | Learn* [online]. Microsoft, c2023, 03/28/2023 [cit. 2023-05-26]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design#asynchronous-operations

[47] MARTIN, Ekuan, Jasoun BOUSKA, Mick ALBERTS, et al. RESTful web API design: Versioning a RESTful web API. *Microsoft | Learn* [online]. Microsoft, c2023, 03/28/2023 [cit. 2023-05-26]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design#versioning-a-restful-web-api

[48] HEINZE, Carolyn. Why GitHub renamed its master branch to main. TheServerSide: Your Enterprise Java Community [online]. Newton (Massachuttes): TechTarget, c2000–2023, 24 Nov 2020 [cit. 2023-05-26]. Dostupné z: https://www.theserverside.com/feature/Why-GitHub-renamed-its-master-branch-to-main

[49] The exponential cost of fixing bugs. In: *DeepSource*[online]. DeepSource, c2023, January 29, 2019 [cit. 2023-05-26]. Dostupné z: https://deepsource.com/blog/exponential-cost-of-fixing-bugs

[50] BELLAIRS, Richard. What Is Static Analysis? Static Code Analysis Overview. *Perforce* [online]. Minneapolis: Perforce Software, c2023, February 10, 2020 [cit. 2023-05-26]. Dostupné z: https://www.perforce.com/blog/sca/what-static-analysis

[51] BELLAIRS, Richard. What Is Lint Code? And What Is Linting and Why Is Linting Important. *Perforce*[online]. Minneapolis: Perforce Software, c2023, March 19, 2019 [cit. 2023-05-26]. Dostupné z: https://www.perforce.com/blog/sca/what-static-analysis

[52] Alexandra. What is Code Profiling? Learn the 3 Types of Code Profilers. *Stackify* [online]. Huntington Beach: Netreo, c2023, May 20, 2020 [cit. 2023-05-26]. Dostupné z: https://stackify.com/what-is-code-profiling/

[53] AS., Kumar. What is Fuzzing: Types, Advantages & Disadvantages. *SecurityFocal* [online]. c2023, July 20, 2021 [cit. 2023-05-26]. Dostupné z: https://www.securityfocal.com/what-is-fuzzing-types-advantages-disadvantages/

[54] https://stackify.com/what-is-code-profiling/ Fuzzing. *OWASP* [online]. Wakefield: OWASP Foundation, c2023 [cit. 2023-05-26]. Dostupné z: https://owasp.org/www-community/Fuzzing

[55] What is software testing?. *IBM* [online]. New York: IBM Corporation 1994, c2023 [cit. 2023-05-26]. Dostupné z: https://www.ibm.com/topics/software-testing

[56] HEROUT, Pavel. *Testování pro programátory*. České Budějovice: Kopp, 2016. ISBN 978-807-2324-811.

[57] What Is Feature Testing And Why Is It Important. *Software Testing Help* [online]. San Diego: Flynndustries, c2008-18, April 28, 2023 [cit. 2023-05-26]. Dostupné z: https://www.softwaretestinghelp.com/feature-testing-tutorial/

[58] What is End-to-End Testing? E2E Testing Full Guide. *Katalon* [online]. Atlanta: Katalon, c2023 [cit. 2023-05-26]. Dostupné z: https://katalon.com/resources-center/blog/end-to-end-e2e-testing

[59] OŠŤÁDAL, Radim. Teoretický základ a přehled kryptografických hashovacích funkcí. In: *Radim Ošťádal - publikace* [online]. Brno: Masarykova Univerzita Brno, 2012, Březen 2012, s. 6 [cit. 2023-05-26]. Dostupné z: https://is.muni.cz/www/ostadal/hash_overview.pdf

[60] Kolize hashovacích funkcí a narozeninový paradox (25. díl). *ISVS.cz* [online]. Praha: ADVICE.CZ, c2001-2022, 20. září 2007 [cit. 2023-05-26]. Dostupné z: https://2011-2015.isvs.cz/kolize-hashovacich-funkci-a-narozeninovy-paradox-25-dil/

[61] *What Is Peppering in Password Security and How Does It Work?* [online]. www.makeuseof.com, c2022, APR 23, 2022 [cit. 2023-05-26]. Dostupné z: https://www.makeuseof.com/what-is-peppering-how-does-it-work/

[62] *Prohlášení NBÚ k využívání hashovacích funkcí* [online]. Praha: NBÚ, c2022, 1.1.2007 [cit. 2023-05-26]. Dostupné z: https://www.nbu.cz/cs/aktualne/prohlaseni-a-tiskove-zpravy/776-4150-prohlaseni-nbu-k-vyuzivani-hashovacich-funkci/

[63] Hash Algorithm Comparison: MD5, SHA-1, SHA-2 & SHA-3. *Code signing store* [online]. The SSL Store™, c2022 [cit. 2023-05-26]. Dostupné z: https://codesigningstore.com/hash-algorithm-comparison

[64] Hashing in Action: Understanding bcrypt. *Auth0 blog* [online]. Okta, c2022, February 25, 2021 [cit. 2023-05-26]. Dostupné z: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/

[65] RHODES, Delton. Scrypt: An Overview of the Scrypt Mining Algorithm. Komodo Academy [online]. Komodo Academy, c2023, Apr 29, 2022 [cit. 2023-05-25]. Dostupné z: https://komodoplatform.com/en/academy/scrypt-algorithm/

[66] SZOSTAK, Marcin. How to improve user password security with Argon2?. *Boldare* [online]. Boldare, c2023, April 19, 2022 [cit. 2023-05-26]. Dostupné z: https://www.boldare.com/blog/how-to-improve-user-password-security-with-argon2/

[67] *Password Hashing Competition: and our recommendation for hashing passwords: Argon2* [online]. PHC, 2019-04-25 [cit. 2023-05-26]. Dostupné z: https://www.password-hashing.net/

[68] Understanding "same-site" and "same-origin." *Web.dev* [online]. Google Developers, c2022, Apr 15, 2020 [cit. 2023-05-26]. Dostupné z: https://web.dev/same-site-same-origin/

[69] KASTRENAKES, Jacob. Facebook stored millions of Instagram passwords in plain text: A lot more than initially stated. In: *The Verge* [online]. Washington, D.C.: VOX MEDIA, c2023, Apr 18, 2019 [cit. 2023-05-26]. Dostupné z: https://www.theverge.com/2019/4/18/18485599/facebook-instagram-passwords-plain-text-millions-users

[70] COBB, Michael a Stephanie MANN. OAuth. *TechTarget*[online]. Newton (Massachuttes): TechTarget, c2019-2023 [cit. 2023-05-26]. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/OAuth

[71] What is Authorization?. *Auth0* [online]. San Francisco: Okta, c2023 [cit. 2023-05-26]. Dostupné z: https://auth0.com/intro-to-iam/what-is-authorization

[72] Error Handling Best Practices. *Auth0* [online]. San Francisco: Okta, c2023 [cit. 2023-05-26]. Dostupné z: https://auth0.com/docs/troubleshoot/error-handling-best-practices

[73] FERRAGAMO, Jeremy, Jim BIRD, et al. Improper Error Handling, . *OWASP* [online]. Wakefield: OWASP Foundation, c2023 [cit. 2023-05-26]. Dostupné z: https://owasp.org/www-community/Improper_Error_Handling

[74] What is OWASP?. Cloudflare [online]. San Francisco: Cloudflare, c2023 [cit. 2023-05-26]. Dostupné z: https://www.cloudflare.com/learning/security/threats/owasp-top-10/

[75] 2022 CWE Top 25 Most Dangerous Software Weaknesses. *CWE: Common Weakness Enumeration*[online]. Bedford (Massachusetts): The MITRE Corporation, c2006-2023, August 03, 2022 [cit. 2023-05-26]. Dostupné z: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[76] How To Meet GDPR Requirements In Mobile And Web Applications. *Selleo* [online]. Bielsko, c2023, Aug 18, 2020 [cit. 2023-05-26]. Dostupné z: https://selleo.com/blog/how-to-meet-gdpr-requirements-in-mobile-and-web-applications

[77] Five Models for Cookie Consent. CookiePro [online]. London: OneTrust, c2023, September 17, 2021 [cit. 2023-05-26]. Dostupné z: https://www.cookiepro.com/knowledge/five-models-for-cookie-consent/

[78] Cookies od začátku roku 2022 pouze se souhlasem. In: Úřad pro ochranu osobních údajů [online]. Praha: Úřad pro ochranu osobních údajů, c2013, 25. 11. 2021 [cit. 2023-05-26]. Dostupné z: https://www.uoou.cz/cookies-od-zacatku-roku-2022-pouze-se-souhlasem/d-53646

[79] Laravel Sail: Debugging With Xdebug. *Laravel: The PHP Framework For Web Artisans* [online]. Little Rock (Arkansas): Laravel, c2011-2023 [cit. 2023-05-26]. Dostupné z: https://laravel.com/docs/10.x/sail#debugging-with-xdebug

[80] Step Debugging: HTTP Cookie. *Xdebug: Debugger and Profiler Tool for PHP* [online]. Rethans, c2002-2023 [cit. 2023-05-26]. Dostupné z: https://xdebug.org/docs/step_debug#cookie-init

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ABAC	*Attribute-Based Access Control*
API	Rozhraní pro programování aplikací (anglicky *Application Programming Interface*)
BDD	Vývoj řízený chováním (anglicky *Behavior Driven Development*)
CD	Průběžné doručení (anglicky *Continuous Delivery*)
CI	Průběžná integrace (anglicky *Continuous Integration*)
CNA	*Cloud Native Architecture*
CSS	Kaskádové styly (anglicky *Cascading Style Sheets*)
CWE	*Common Weakness Enumeration*
DIP	Princip inverze závislostí (anglicky *Dependency Inversion Principle*)
DRY	"Neopakuj se" (anglicky *Don't Repeat Yourself*)
DTO	*Data Transfer Object*
E2E	*End-To-End*
ESB	*Enterprise Service Bus*
eTLD	*Effective Top Level Domain*
FDD	Vývoj řízený vlastnostmi (anglicky *Feature Driven Development*)
GDPR	*General Data Protection Regulation*
GPU	*Graphics Processing Unit*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IDE	Vývojové prostředí (anglicky *Integrated Development Environment)*
IETF	*Internet Engineering Task Force*
IPSec	*Internet Protocol Security*
ISP	Princip segregace rozhraní (anglicky *Interface Segregation Principle*)
JSON	*JavaScript Object Notation*
KISS	"Udržuj to jednoduché, hlupáčku" (anglicky *Keep It Simple Stupid*)
LSP	Liskovové princip zastoupení (anglicky *Liskov Substitution Principle)*
MD5	*Message Digest Algorithm 5*
MSA	Mikroservisní architektura (anglicky *Microservice architecture*)

MSSQL	*Microsoft SQL Server*
MVC	*Model-View-Controller*
MVCS	*Model-View-Controller-Service*
MVP	*Model-View-Presenter*
MVVM	*Model-View-ViewModel*
NBÚ	Národní bezpečnostní úřad
OAuth	*Open Authorization*
OCP	Princip otevřenosti-uzavřenosti (anglicky *Open-Closed Principle*)
ORM	*Object Relational Mapping*
OWASP	*Open Web Application Security Project*
PEP	*Python Enhancement Proposals*
PGP	*Pretty Good Privacy*
PHP	Hypertextový preprocesor (anglicky *Hypertext Preprocessor*)
PSR	*PHP Standards Recommendations*
RBAC	*Role-Based Access Control*
ReBAC	*Relationship-Based Access Control*
REST	Representational state transfer
SDD	Vývoj řízený specifikací (anglicky *Specification-driven development)*
SHA	*Secure Hash Algorithm*
SOA	Servisně orientovaná architektura (anglicky Service-oriented architecture)
SoC	Princip rozdělení zájmů (anglicky Separation of Concerns)
SOLID	SRP + OCP + LSP + ISP + DIP
SPA	Jednostránková aplikace (anglicky *Single-page Application*)
SQL	Strukturovaný dotazovací jazyk (anglicky *Structured Query Language*)
SRP	Princip jediné odpovědnosti (anglicky *Single Responsibility Principle*)
SSH	*Secure Shell*
SSL	*Secure Sockets Layer*
TDD	Vývoj řízený testy (anglicky *Test Driven Development*)
UI	*User Interface*
URI	*Uniform Resource Identifier*

URL	*Uniform Resource Locator*
WYSIWYG	Co vidíš, to dostaneš (anglicky *What You See Is What You Get*)
XP	Extrémní programování (anglicky *Extreme programming*)
XSS	*Cross-site scripting*
YAGNI	„to nebudeš potřebovat” (anglicky *You Aren't Gonna Need It*)
YAML	*YAML Ain't Markup Language*

SEZNAM OBRÁZKŮ

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

- CD

PŘÍLOHA P I: CD

CD přiložené k diplomové práci obsahuje:

- diplomovou práci ve formátu PDF,
- zdrojové kódy praktické části.