FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Implementace IT řešení pro volejbalový svaz
Student:	Martin Kop
Vedoucí:	Ing. Josef Pavlíček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Cílem bakalářské práce je navrhnout a implementovat IS pro volejbalový svaz

Postupujte dle definovaných bodů:
1. Ve spolupráci s budoucími uživateli proveďte analýzu požadavků na systém.
2. Navrhněte vhodnou architekturu řešení.
3. Vyberte vhodnou platformu.
4. Implementujute případy užití a uživatelské scénáře.
5. Aplikaci otestujte a definujte závěry.
6. Zdrojové kódy umístěte do sdíleného GIT repozitáře.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 17. prosince 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Mobilní aplikace pro ČVS

Martin Kop

Katedra softwarového inženýrství
Vedoucí práce: Ing. Josef Pavlíček, Ph.D.

4. června 2020

Poděkování

Chtěl bych poděkovat vedoucímu mé práce, Ing. Josefovi Pavlíčkovi, Ph.D. za věcné připomínky a odborné vedení práce. Dále vedení Českého volejbalového svazu, které realizaci aplikace umožnilo. Zároveň bych chtěl poděkovat mé rodině a všem přátelům, kteří mě během studií podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 4. června 2020 . . . . . .

Odkaz na tuto práci

Kop, Martin. *Mobilní aplikace pro ČVS*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Bakalářská práce se zabývá mobilní aplikací, zaměřenou na příznivce volejbalu. Aplikace se skládá ze dvou částí. Část, zahrnutá v této práci, je věnována architektuře, infrastruktuře a práci s daty. Druhá část, zpracovaná kolegou Markem Šulcem, se pak zaměřuje na design aplikace a ekonomicko–manažerskou analýzu. Architektura aplikace podporuje rozšířitelnost a flexibilitu pro případné změny v budoucnu. Ve vzniklé aplikaci příznivci volejbalu dohledají informace o uplynulých nebo nastávajících událostech všech týmů a soutěží. Volejbalovému svazu se podařilo uživatelům pomocí aplikace více zpřístupnit informace a nahradit nevyhovující akuální řešení.

Klíčová slova Mobilní aplikace, zpracování dat, architektura softwaru, volejbal, REST API, Vaadin, Java, aplikační server, databáze

Abstract

This thesis deals with a mobile application focused on volleyball fans. The final application is composed of two parts. The first part, which is considered in this thesis, is focused on software architecture, system infrastructure and data processing. The second part is the responsibility of my colleague Marek Šulc and is focusing on design, business and management analysis. The software architecture supports scalability and flexibility in case, some changes are needed in the future. Inside the implemented mobile application, volleyball fans can find information about past or future events of teams or leagues. The volleyball federation was able to make information more accessible for users an was able to replace the actual unsuitable solution.

Keywords Mobile app, data processing, software architecture, volleyball, REST API, Vaadin, Java, application server, database

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Mobilní telefony v dnešní době otevírají obrovský potenciál ve všech oborech. Není výjimkou ani sport, kde kromě aplikace, které vám pomohou trénovat, vám zprostředkovávají všechna data a možnost sledovat svůj nejoblíbenější tým z druhého konce světa. Většina rozšířených sportů v ČR má vlastní mobilní aplikaci, která zjednodušuje fanouškům možnost se o sport zajímat. Český volejbal takovouto aplikaci prozatím nevlastní a tato práce nedostatek řeší.

Mobilní aplikace bude prospěšná pro zvednutí zájmu o sport, který k ČR patří po mnoho let. V posledních letech byl však populárnější fotbal, hokej i florbal. Zvednutí zájmu o tento sport vede k hojnějším mládežnickým základnám, které vychovávají budoucí generace sportu. Pro osoby, které přímo sport neprovozují, je sledování sportu koníčkem a relaxací od povinností. Zároveň je sledování volejbalových zápasů vhodné jako celorodinná aktivita. Porovnáním s fotbalem a hokejem, které jsou hodně kontaktní sporty a jejich stadiony plné agresivních fanoušků, je sledování volejbalu jemnější alternativa. S tímto zaměřením budou navrhovány funkce a design aplikace.

Z odborného hlediska jsem si téma vybral z důvodu komplexnosti aplikace. Rád zkoumám existující frameworky a moderní technologie. Zadání má široký záběr a obsahuje mnoho různorodých oblastí, ve kterých se mohu seberealizovat. Jedná se o plnohodnotný projekt pro reálného klienta. Motivací je také fakt, že jsem sport provozoval celý život a mám k němu blízko. Mohl jsem poznat krásy tohoto sportu a jsem přesvědčený o tom, že by se měl v ČR rozvíjet a přiblížit se například Polsku, kde je volejbal národním sportem.

Kapitola 1

Cíl práce

Cílem je navrhnout a realizovat mobilní aplikaci pro Český volejbalový svaz (dále ČVS). Zejména navrhnout architekturu aplikace a nastavit její kompletní infrastrukturu. Dále na základě uživatelských požadavků, sesbíraných kolegou Markem Šulcem, odvodit systémové požadavky a z nich vycházet při návrhu systému. Úspěšně spojit tuto práci, která se zaměřuje na celkovou architekturu a práci s daty, s prací kolegy Marka Šulce, která se zaměřuje na design aplikace. Tento projekt následně otestovat a připravit na reálný provoz pro Český volejbalový svaz. Zároveň je cílem automatizovat sestavování a nasazování aplikace, což umožní efektivnější vývoj a údržbu při provozu aplikace.

Kapitola 2

Struktura softwarových systémů

V této kapitole popisuji přístupy členění a organizace softwarových systémů, tzv. architektonické vzory. Rozebírám, jak lze systém členit do modulů, jaké to má výhody a nevýhody. V druhé části jsou poté rozebrány návrhové vzory, které se již týkají konkrétních modulů systému. Návrhové vzory se nezaobírají celkovým obrazem aplikace, ale pouze dílčími částmi. Účel, tedy jak třídy organizovat a členit efektivně, zůstává neměnný. Softwarové systémy v sobě mají čím dál častěji zahrnuté i podpůrné funkce, které systém napříč jeho životním cyklem provázejí. Zmiňuji se tedy i o testování a automatizaci procesů.

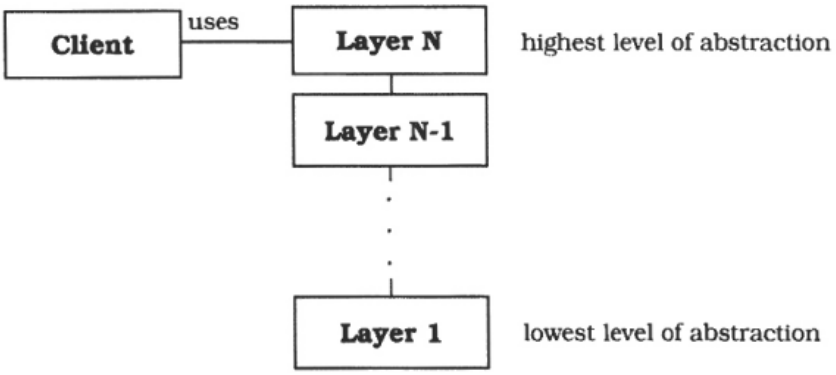
2.1 Architektonické vzory

Dle [2] jsou architektonické vzory na nejvyšší pozici v hierarchii vzorů. Mají za úkol pomoci softwarovému architektovi při návrhu fundamentální struktury aplikace. Typicky člení aplikaci do modulů, mezi kterými popisuje způsob komunikace. Jednotlivé moduly mohou poté použít svá řešení s respektem k architektonickému návrhu.

2.1.1 Vrstvená architektura

Nejpoužívanější architektonický vzor, vrstvenou architekturu, popisuje [2]: *„Vrstvená architektura pomáhá strukturovat aplikace, které je možné dekomponovat. Dekomponované úkoly spadají do skupin, kde každá skupina zastupuje jednu úroveň abstrakce.“* Ukázkovým příkladem mohou být komunikační protokoly. Ty jsou rozdělené do vrstev, uspořádaných od fyzického přenosu dat, až po aplikační vrstvu. Jako další příklad autor uvádí informační systém členěný na prezentační, aplikační, doménovou a databázovou vrstvu. Každá vrstva položená v hierarchii výš využívá rozhraní vrstvy přímo pod sebou.

Oproti základnímu monolitickému přístupu, kde jsou veškeré komponenty aplikace nekontrolovatelně svázané, má rozdělení do vrstev jasné výhody:



Obrázek 2.1: Příjem požadavku vrstvenou architekturou (zdroj [2])

- Při vývoji můžeme jednoduchým způsobem vrstvy distribuovat mezi jednotlivé týmy.
- Omezená provázanost modulů nám umožní jednodušší výměnu jedné z vrstev.
- Znovupoužitelnost vrstev mezi aplikacemi.
- Přehlednost systému ovlivňující kvalitu testování, dokumentace, návrhu, atd.

Vrstvený architektonický vzor blíže nespecifikuje složitost jednotlivých vrstev. Může se jednat o jednoduché kompresování dat nebo o složité subsystémy, které je nutné členit. Uvnitř jedné vrstvy by však měly být zastoupeny úkoly na stejné úrovni abstrakce.

Autor [2] uvádí nejčastější případ aplikace vrstvené architektury. A to v případě, kdy aplikace zpracovává uživatelské požadavky. Požadavek z pravidla zachycuje vrstva na nejvyšší úrovni abstrakce (viz obrázek 2.1). Požadavek je však schopna zpracovat jen částečně a zbytek deleguje na nižší vrstvu. Při delegaci volá vyšší vrstva funkce té nižší vícekrát. Požadavek je dekompozicí rozebrán na nižší úroveň abstrakce. Čím víc postupujeme v úrovni abstrakci hlouběji a ve vrstvách směrem dolů, narůstá počet požadavků. Dokud nenarazíme na nejnižší vrstvu, kde jsou konečně zprocesovány úkony na nejnižší úrovni abstrakce. Odpovědi poté kompozicí stoupají vrstvami nahoru, až je v poslední vrstvě složená kompletní odpověď na požadavek uživatele.

2.1.2 Architektura filtrů

Architektonický vzor filtrů slouží, dle [2], jako: *„Architektura, která definuje strukturu systémů zpracovávající velké toky dat. Každý krok zpracování je za-*

pouzdřen jako komponenta filtru. Data jsou sdílena mezi filtry pomocí trubek. Kombinování filtrů poté umožňuje sestavovat skupiny souvisejících systémů.“ Jako ukázkový příklad lze použít kompilátor programovacích jazyků. Zpracovávaný kód tvoří tok dat a je tedy vhodné použít pro kompilátor architekturu filtrů. Nejprve kód, zapsaný v ASCII znacích, projde analýzou syntaxe. Pokud je syntaxe validní, utvoří se strom, reprezentující strukturu programu. Ten následně postoupí do filtru zodpovědného za určení sémantiky. Strom s určenou sémantikou poté postupuje do filtru optimalizace. Na tomto příkladu vidíme, jak výstupu předchozího filtru využívá ten následující. Až konečně jejich řetězením docílíme zkompilování kódu. Tento přístup má určité výhody:

- Budoucí úpravy je možné realizovat překombinováním filtrů nebo jejich výměnou.
- Filtry lze použít opakovaně.
- Rozšířitelnost a flexibilita propojení s jinými systémy.

Jako jednu z klíčových charakteristik této architektury [2] uvádí, že tok dat je zpracováván inkrementálně. Filtr nečeká, až zpracuje kompletní vstup, než ho předá následujícímu. Místo toho je při zpracování jedné ucelené části dat tato část okamžitě předána dál. Tím v systému dochází k minimálnímu zpoždění, z čehož plyne dříve zmíněný vhodný případ užití při zpracování velkého množství dat.

2.1.3 Architektura MVC

Model-View-Controller architekturu, zkráceně MVC, popisuje [2] jako: *„Architektonický vzor rozdělující interakivní aplikaci do tří částí. Model obsahuje základní funkčnost a data aplikace. Pohledy prezentují informace uživateli. Kontrolery obsluhují vstupy od uživatele. Pohledy a kontrolery společně tvoří uživatelské rozhraní. Mechanismus propagace změn zajišťuje konzistenci mezi uživatelským rozhraním a modelem.“* Jako příklad autor uvádí volební informační systém. Ten by mohl například obsahovat tabulku pro zadávání sečtených výsledků ve volební místnosti. Sesbírané výsledky se ukládají do společného modelu. Nad modelem jsou pak vybudovaná různá uživatelská rozhraní, která uživateli data prezentují. Například mobilní aplikace, desktopová aplikace a grafika v televizním vysílání. Všechny nově sesbírané hlasy jsou po zadání okamžitě propagovány napříč celým systémem. Uveďme si největší výhody MVC architektury:

- **Více uživatelských rozhraní** - MVC důrazně odlučuje data od rozhraní pro uživatele. Umožňuje tak jednoduše přidávat či ubírat počet aktivních uživatelských rozhraní.

- **Synchronizace** - architektura je schopná i při mnoha uživatelských rozhraních udržet napříč systémem synchronizovaná data.

Důležité je si zmínit i dva velké nedostatky, které se sebou použití čisté MVC architektury nese:

- **Efektivita** - při častých změnách dat může být systém zahlcen, jelikož se snaží každou změnu v modelu propagovat do všech uživatelských rozhraní a udržet je tak aktuální.
- **Úzká provázanost s modelem** - v MVC architektuře přístupují pohledy a kontrolery k modelu přímo. Případné změny modelu mohou poškodit nebo úplně rozbít vazbu s uživatelskými rozhraními. Tento problém je o to významnější, o kolik víc existuje uživatelských rozhraní.

Autor [2] závěrem říká, že může být použití MVC architektury, společně s moderními nástroji návrhu uživatelského rozhraní, problematické. Spousta z těchto nástrojů si totiž vstupy od uživatele obstarává vlastním způsobem a programátorovi je pak ztíženo mít nad tímto procesem plnou kontrolu.

2.2 Návrhové vzory

První velkou publikací návrhových vzorů je [5]. Kniha, ve které čtyři autoři popisují nejčastěji opakující se problémy v objektově–orientovaném návrhu a jejich řešení. Dle jejich slov se návrhové vzory netýkají architekury softwarového systému ani detailních návrhů algoritmů. Algoritmy mohou být ve většině příkladů zapouzdřeny do jediné třídy objektově–orientovaného návrhu. Dle [5] je jejich místo právě někde uprostřed: *„Návrhové vzory v této knize, jsou popisy komunikujících objektů a tříd a jsou přizpůsobeny k řešení obecného návrhového problému v konkrétním kontextu.“* Návrhové vzory popsané v této knize jsou nezávislé na konkrétním objektově–orientovaném programovacím jazyce. Jak naznačuje citace, jedná se o předlohy řešení obecných problémů. Klíčové je tedy pro konkrétní problém určit, jaký návrhový vzor je možné použít, a zvážit klady či zápory. Nějaké návrhové vzory z knihy [5] si uvedeme a popíšeme jejich hlavní myšlenku.

2.2.1 Vzory inicializace

Zaměřují se na problémy objektově–orientovaného návrhu inicializace nových objektů a správu těchto instancí.

Builder

Rozděluje funkcionalitu objektu a jeho inicializaci. Používá se, pokud je konstrukce nového objektu složitá, například pokud se při inicializaci používají

další objekty. Uveďme si situaci, kdy se třída každého robota skládá z těla, hlavy, končetin a charakteristických metod robota. Při inicializaci chceme mít flexibilitu při sestavování nových typů robota, kteří budou mít sice stejné metody, ale budou se lišit třeba v typu končetin. Zmíněný návrhový vzor dává uživateli větší kontrolu při inicializaci tohoto objektu.

Singleton

Cílem singletonu je zaručit, že v systému bude existovat právě jedna instance dané třídy. Instancování třídy je odpovědnost třídy samotné. Pokud chce někdo další tuto třídu použít, získá na ni pouze odkaz, ale nedokáže vytvořit instanci další. Typicky by se mohlo jednat o třídu manažera tiskáren, kde chceme zaručit, aby existoval v systému právě jeden. Při existenci více manažerů najednou by mohlo dojít ke kolizím požadavků na tiskárny a ta by se nechovala očekávaným způsobem.

2.2.2 Strukturální vzory

Tato skupina návrhových vzorů se zaměřuje na správné členění tříd a logické delegování úkolů.

Composite

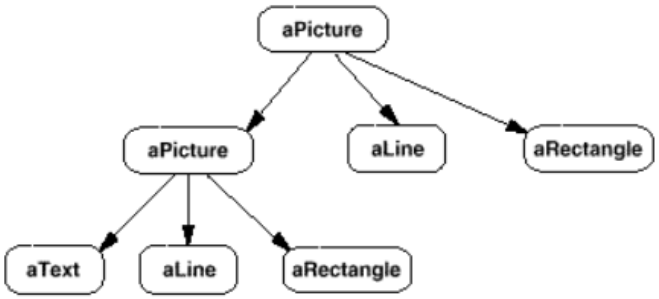
Návrhový vzor kompozice umožňuje atomické třídy seskupovat do tříd vyššího řádu. Navíc mohou být seskupovány také třídy vyšších řádů. To vede na rekurzivní stromovitou strukturu, kde uživatel používá atomické třídy stejným způsobem jako třídy vyššího řádu a dokáže je opakovaně seskupovat (viz obrázek 2.2). Jedno z mnoha aplikací je v oboru grafiky, kde jsou atomickými třídami základní tvary jako čtverec a trojúhelník. Tvary se dají seskupovat do složitějších tříd vyšších řádů, např. domeček složený z čtverce a trojúhelníku. A třídy vyšších řádů se dají seskupovat rekurzivně, např. město složené z více domečků, kraj složený z měst, atd.

Decorator

Dekorátor umožňuje k třídě dynamicky připojit dodatečnou funkcionalitu. Nabízí tak flexibilní alternativu k dědičnosti, která je součástí většiny objektově orientovaných jazyků. Dekorátor nám také umožní funkcionalitu přidat jenom k vybraným instancím.

Facade

Nad skupinou tříd, kterou můžeme nazvat subsystém, vytvoří rozhraní, přes které je jednodušší subsystém používat. K subsystému tak vytvoří vyšší úroveň



Obrázek 2.2: Návrhový vzor kompozice (zdroj [5])

abtrakce. Místo přímého přístupu k třídám subsystému, uživatel používá definované rozhraní. Tím se celý proces pro uživatele zjednoduší, jelikož můžeme zpřístupnit pouze metody, které uživatel opravdu využije. Můžeme také seskupit více metod, které uživatel často v posloupnosti volá, do jedné.

2.2.3 Vzory chování

Zaměřují se na funkční stránku OO návrhu, kde potřebujeme třídy navrhnout tak, abychom docílili požadovaného chování.

Iterator

Umožňuje přístup k elementům nějaké datové struktury, např. souboru elementů. Iterátor je často součástí programovacích jazyků, ve kterých zajišťuje procházení skrz datové struktury. Uživatele odstiňuje od implementace této struktury a nemusí ji tak znát, aby byl schopen projít všemi elementy, které v ní leží.

Observer

Definuje 1:N vazbu mezi objekty. Pokud pozorovaný objekt změní stav, jsou všichni, kteří s ním jsou provázáni, upozorněni, že se tak stalo. Řeší konzistenci dat napříč systémem. Lze si všimnout úzké spojitosti s MVC architektonickým vzorem popsaným dříve v této kapitole.

Template method

Definuje nějakou stěžejní část algoritmu a poté části algoritmu, které jsou flexibilní. Umožní zděděným třídám flexibilní části modifikovat bez zásahu do hlavní struktury algoritmu.

2.3 Testovací metody

Testování známe ze školních lavic. Vyučující nám zadá test a naším úkolem je ho vyplnit. Vyučující následně test vybere a vyhodnotí, zda naše znalosti překročily určité procento z probírané látky. Pokud ano, tak jsme testem prošli, pokud ne, tak jsme v testu neuspěli. V případě neúspěchu jsme odkázáni na doučení se látky a poté, co doplníme naše neznalosti, si test zopakujeme. Tento samý proces bychom mohli najít v testování softwaru. Jak vysvětluje [1], je obzvlášť důležité pochopit co testování není. *„Testování softwaru nemá dokázat, že je testovaný systém či program bez jakýchkoli závad. Jeho smyslem ani není všechny tyto defekty odhalit. Takový úkol je pro testovací tým nemožný dosáhnout.“* Autor je přesvědčen, že vezmeme–li v potaz jakýkoli složitější softwarový projekt z praxe, existuje nespočet možností:

- Jakou posloupnost akcí uživatel zvolí.
- Jaká data uživatel zadá.
- Na jak nakonfigurovaném zařízení software běží.
- V jakých podmínkách je software aktuálně provozován.

Z těchto důvodů nemůže existovat způsob, jak zkontrolovat, že systém neobsahuje žádnou chybu, či veškeré chyby nalézt. Stejně jako testování ve školních lavicích nemá ověřovat naši stoprocentní znalost, které je nemožné dosáhnout.

Závěrem [1] je, že testování je primárně o řízení rizik. Celý smysl testovacího procesu popisuje slovy: *„Efektivně poskytovat pravidelné, přesné a užitečné informace o kvalitě softwaru, které pak pomáhají řídit rizika testovaného systému.“* V našem případě se budeme zabývat metodami, které nám pomůžou tyto informace získat.

2.3.1 Statické testy

Slovo statické napovídá, že se jedná o formu testování, které neprovede žádné spuštění testovaného systému. Jak ale sděluje [1], jedná se spíše o definici, čím statické testování není. Dle [1], je statické testování kontrola prvků, které testovaný systém reprezentují. V případě softwaru se jedná typicky o požadavky na systém, diagram průchodu systémem, model entit a jejich vztahů, zdrojový kód, ale i samotné testovací případy. Jedná se o nejúčinnější metodu testování s ohledem na poměr ceny a efektivity. Odhalení chyby, například v diagramu, předchází tomu, aby se chyba propagovala do vývoje, behaviorálního testování, až propagování chyby do produkce.

Ne všechny firmy, vyvíjející software, používají metodiku statických testů, jelikož špatně odhadují náklady a finální přínost tohoto druhu testů. Formy

kontrol jsou například kontroly kódu druhou osobou, důkladné procházení modelů a jejich vlastností, programování ve dvojici, atd.

2.3.2 Behaviorální testy

Jak uvádí [1] v kapitole behaviorálních testů, významnou částí práce programátora je analyzovat, jakým způsobem se má systém v konkrétních situacích zachovat. Existuje velké riziko, že programátor nerozpozná všechny situace, které mohou nastat, nebo je analyzuje špatně.

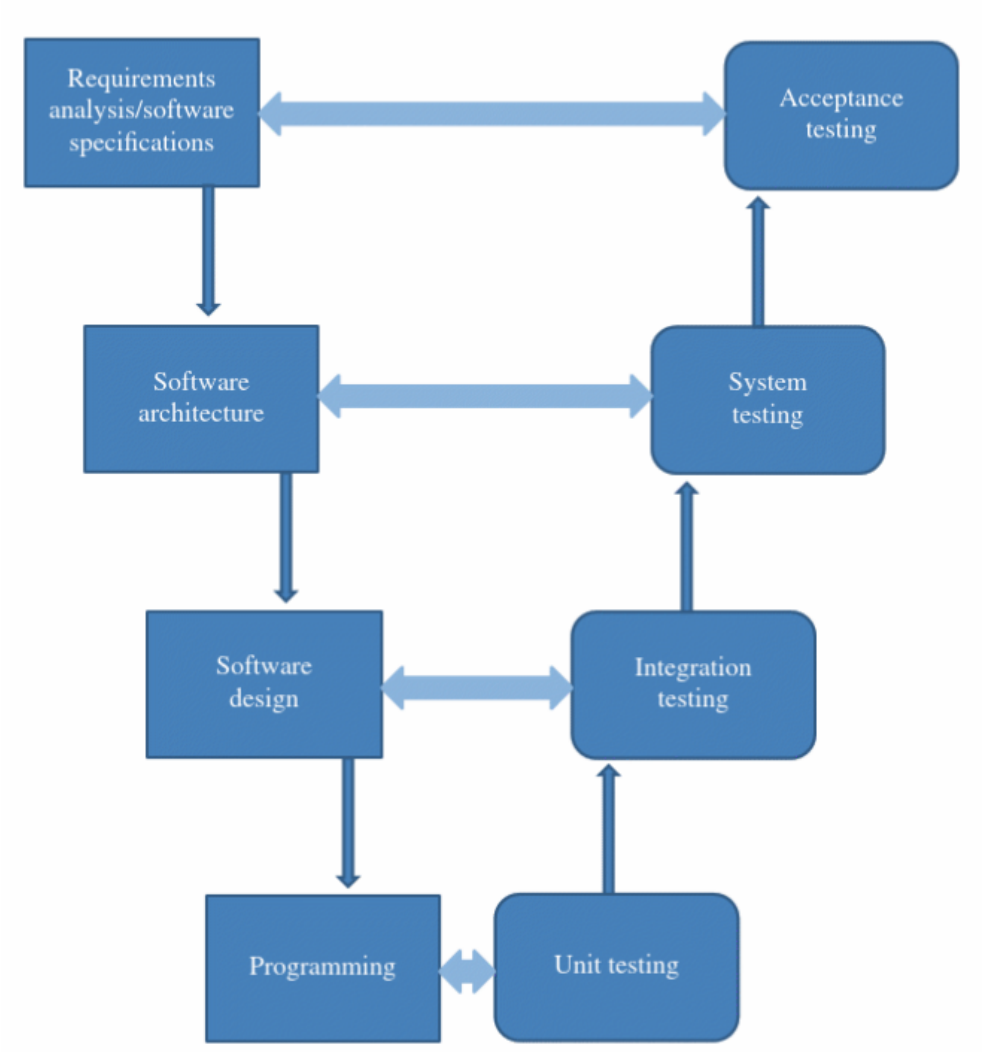
Častou chybou je, jak autor uvádí, rozpoznání mezních hodnot vstupů. Programovací jazyky pracují s různými datovými typy, reprezentující celá čísla, desetinná čísla nebo textové řetězce. Program musí být schopen reagovat, pokud je zadaný nevalidní formát nebo validní formát z nepovoleného rozsahu. Cílem behaviorálních testů je obsáhnout všechny tyto hodnoty a otestovat, že na ně systém reaguje správně.

Další metodikou, spadající do behaviorálních testů, jsou testy případů užití. Tyto testy modelují běžné případy, ve kterých bude uživatel software používat a ověřují, že bude jeho požadavek obsloužen. Nutné je si uvědomit, že různí uživatelé mohou aplikace užívat různým způsobem. OO design nám často tyto způsoby testů ulehčuje, jelikož případy užití jsou nějakým způsobem vytvořené při návrhu každé třídy.

2.3.3 V-Model

Z hlediska softwarového inženýrství je testování často prezentováno jako poslední fáze vývoje projektu. Dle [10] je více odpovídající představa testování jako kontinuálního procesu, který je součástí každé fáze v projektu. V-Model nám páruje všechny stupně softwarového návrhu na kategorii testů (viz obrázek 2.3). Na diagramu je zajímavý opačný průchod úrovněmi v návrhu a při testování. V návrhu většinou postupujeme směrem z venku dovnitř. Na začátku si specifikujeme uživatelské požadavky, postupujeme přes návrh softwarové architektury, návrh jednotlivých modulů až po samotnou implementaci. Při testování pak testujeme od nejnižší úrovně, po testování uživatelských požadavků. Skupiny statických a behaviorálních testů mohou být součástí každé z těchto kategorií testů.

- **Unit testy** - testují kvalitu implementace jedné částí systému, jednotlivé třídy a jejich metody.
- **Integrační testy** - ověřují, zda lze systém sestavit a ověřují komunikaci mezi komponentami systému.
- **Systémové testy** - testují aplikaci jako celek, zaměřují se na průchod aplikací a na testování uživatelských scénářů.



Obrázek 2.3: V-Model (zdroj [10])

- **Akceptační testy** - testy, které provozuje zákazník ve svém prostředí. Tyto testy jsou většinou odvozené ze zadání.

2.4 Automatizace procesů

CI/CD metody, jak uvádí [6], jsou způsoby automatizace procesu vývoje softwaru s cílem odhalit chyby v co nejranějším stádiu vývoje. Tohoto cíle se metody CI/CD snaží dosáhnout minimalizací lidského zásahu ve všech fázích vývoje softwaru. Konkrétněji popisuje [6] tyto metody následovně: *„CI/CD metody zahrnují kontinuální sestavování, testování a nasazování malých změn ve zdrojovém kódu. Tím minimalizovat případy vytvoření nového kódu na základech již nefunkčních předešlých verzí.“* Tyto metody jsou členěné do tří základních úrovní. Každá nadcházející úroveň automatizuje a kontinuálně provádí něco navíc od té předchozí:

- **Continuous Integration** - na této úrovni je po každém nahrání změn zdrojového kódu do sdíleného repozitáře provedeno sestavení a testování. Děje se tak i při vývojových větvích při vývoji. Vývojáři mají okamžitou zpětnou vazbu na změny, které byly provedeny.

- **Continuous Delivery** - kromě sestavení a otestování zdrojového kódu je také automatizovaný proces nasazení. Tento krok však doprovází stále lidský faktor, který rozhoduje například o tom, jestli se provedené změny mají nasadit do testovacího prostředí nebo do produkčního prostředí. Samotný proces nasazení, který mnohdy není triviální, je ale proveden po učinění těchto rozhodnutí automaticky.

- **Continuous Deployment** - proces nasazování je prováděn plně automaticky bez jakéhokoli lidského zásahu

2.4.1 Gitlab CI/CD

Předpokládáme-li, že zdrojový kód je verzován systémem Git a že je spravován na platformě Gitlab, která metodiku CI/CD implementuje, můžeme si detailněji popsat, jak systém CI/CD na této platformě funguje.

Pro zprovoznění je třeba umístit skrytý konfigurační soubor *„.gitlab-ci.yml“* do kořenové složky projektu, verzovaným pomocí verzovacího systému Git. Každý blok změny (tzv. commit) zdrojového kódu, vyvolá po nahrání do sdíleného repozitáře soubor povelů (tzv. pipeline), způsobem, jakým je vytvořený konfigurační soubor. Dle nastavené úrovně automatizace se provede buď pouze sestavení a otestování změn nebo i nasazení sestavené aplikace. Pipelines jsou rozdělené do různých fází (tzv. stage), zodpovědné vždy za jednu skupinu úkolů (tzv. job). Fáze, které jsou v Gitlab CI/CD definované jako výchozí, vycházejí z CI/CD metody popsanou dříve:

- **Sestavení** - zkontroluje, jestli je celý projekt sestavitelný.
- **Testování** - ověří funkční požadavky softwaru.
- **Nasazení** - nasadí změny do prostředí, kde se software spustí.

Jednotlivým úkolům můžeme nastavit, zda se mají spustit plně automaticky nebo vyžadují manuální spuštění. Jako výchozí hodnota je nastaveno automatické spuštění. Lze ji konfigurovat pomocí paramteru *„when“*.

Celý tento proces musí být prováděn na nějakém stroji. Gitlab dává k dispozici ke každému projektu, tvz. Runner, což je stroj, na kterém všechny tyto procesy běží. Pro neplacený účet je využívání přiřazeného stroje omezeno na určitý počet hodin měsíčně. Na tomto stroji se spouští Docker Image, který si musíme v konfiguračním souboru specifikovat pomocí hodnoty *„image“*. Až v něm se nám teprve spouští veškerý proces automatizace a jeho jednotlivé úkoly. Gitlab CI/CD má další pokročilé funkce, jako nastavování proměnných před každým manuálním provedením úkolu, statistiky, časované spouštění úkolů a mnoho dalších. Pro kontext této práce je však popsaný základní princip automatizace procesů vývoje dostačující.

Kapitola 3

Technologie

V této kapitole si detailněji rozebereme technologie, které nás nejspíš při návrhu a implementaci aplikace neminou. Celkových potencionálních technologií je nespočet a proto si uvedeme ty nejdůležitější a nejzajímavější.

3.1 HTTP protokol

Hypertext Transfer Protocol, zkráceně HTTP, hraje dle [8] obrovskou roli v kontextu webových serverů. Jedná se o protokol, který popisuje, jakým způsobem mezi sebou komunikují dva a více webových serverů. Existuje více verzí, ale tou nejčastěji používanou, jak uvádí [8], je HTTP 1.1 z roku 1999. Tento protokol operuje na bázi požadavku a odpovědi. Klient tedy posílá serveru požadavky, server požadavky zpracovává a posílá klientovi zpět odpověď. Každá komunikace končí úspěchem nebo neúspěchem, z čehož vyplývá, že komunikace nikdy nedospěje do speciální situace.

3.1.1 Požadavek

Na počátku komunikace musí být klientem vytvořen požadavek a odeslán protistraně (serveru). Struktura má pevně definovanou strukturu, ve které na první řádce najdeme vždy příkaz. Ten se skládá ze tří částí: metoda, URI a verze. Metoda serveru říká, jaká operace je po něm požadována. URI specifikuje cestu na serveru a následuje verze HTTP protokolu. Úryvek ze seznamu metod, které protokol používá, si uvedeme v následující tabulce:

Následují hlavičky, což jsou dvojice klíč/hodnota, blíže upřesňují parametry požadavku. Za hlavičkou se nachází tělo, oddělené od hlavičky prázdným řádkem. Tělo je však nepovinné, jelikož u nějakých požadavků, např. GET, není potřeba.

Název	Význam
CONNECT	Nastartuje připojení
GET	Vyžaduje data od protistrany
PUT	Umisťuje zdroje na protistraně
POST	Posílá data protistraně
DELETE	Maže zdroje

Tabulka 3.1: Ukázka používaných metod HTTP a jejich význam

3.1.2 Odpověď

Po zpracování server vytvoří odpověď na požadavek klienta. I ta má pevně určenou strukturu, která se od struktury požadavku liší. Na první řádce najdeme opět příkaz, tentokrát složený z verze, stavového kódu a názvu kódu. Verze odpovídá používané verzi HTTP protokolu, stejně jako u požadavku. Ukázka možných stavových kódů, jejich názvu a významu je v následující tabulce:

Kód	Název	Význam
200	OK	Požadaek proběhl úspěšně
202	Accepted	Autorizace byla úspěšná
400	Bad Request	Nevalidní syntaxe
404	Not Found	Data nebyla nalezena
500	Server Error	Chyba na straně serveru

Tabulka 3.2: Ukázka používaných stavových kódů HTTP a jejich význam

Obdobně jako u požadavku následují hlavičky pro upřesnění vlastností odpovědi. Po hlavičkách se nachází řádkem oddělené tělo, které je nepovinné.

3.1.3 HTTPS

Velkým nedostatkem HTTP protokolu je, že data posílaná přes síť, nejsou žádným způsobem šifrovaná. Tato slabina může být zneužita a data posílaná po síti odcizena třetí stranou. Z tohoto důvodu vznikl protokol HTTPS, který je rozšířením původního HTTP o prvek bezpečnosti. Tento protokol se v podstatě stal standardem ve všech aplikacích, ve kterých jsou sdílené, pro uživatele citlivé, údaje.

HTTPS využívá SSL certifikáty, pomocí kterých je mezi dvěmi zařízeními, komunikujícími na síti, vytvořené bezpečné, šifrované spojení. SSL certifikáty jsou součástí serveru, se kterým klient komunikuje. Klient nejprve vyžaduje od serveru prokázat svou identitu. Server pošle kopii svého SSL certifikátu klientovi a ten se následně rozhodne, jestli bude serveru důvěřovat nebo ne. Většinou si klient ověřuje původ certifikátu u nějaké CA, což je autorita posky-

tující věrohodné certifikáty. Při ověření důvěryhodnosti je pak mezi klientem a serverem vytvořena šifrovaná komunikace a zamezeno tak třetím stranám informace odposlouchávat. [11] [3]

3.2 Databázové systémy

Z počátku si definujme tři pojmy důležité pro orientaci v problematice databázových systémů. [4] definuje pojmy DBS, Databáze a DBMS následovně: *„Databázový systém, neboli DBS, je systém uchovávající digitální záznamy. Cílem DBS je správa informací a jejich zpřístupnění kdykoli je třeba. Databáze typicky uchovává související data v počítačovém systému. Databázový management systém, neboli DBMS je potom sada programů, které umožňují databázi spravovat.“* Jako typické nástroje, které DBMS navíc poskytuje, uvádí [4] tato:

- Definování datových struktur, jejich vztahů a závislostí
- Manipulace s daty pomocí operací vkládání, čtení, upravování a mazání
- Spravování přístupových práv k datům
- Podpora pro programovací jazyky

Následně autor objasňuje souvislosti mezi těmito pojmy, kde je závěr následující. Databázový systém (DBS) je tvořen z dvou hlavních částí, DBMS a databáze. Databáze jsou záznamy informací a způsob, jakým jsou záznamy strukturované. DBMS je software, který záznamy spravuje. Provádí nad daty například operace vkládání, mazání, řeší přístupová práva k datům, apod. Tyto dvě části dohromady tedy tvoří databázový systém.

3.2.1 Relační databáze

Relační databázový model, jak uvádí [4], je zdaleka tím nejpoužívanějším. Tento koncept je postavený na matematických principech teorie množin a lineární algebry. Relační model zároveň obstál v praxi po mnoho let a tím si zajistil většinovou část mezi používanými databázovými modely. Dále si vysvětlíme základní pojmy, na kterých je koncept vybudován:

- **Entita** - objekt, který jsme schopni v dané problematice popsat, pojmenovat a uschovávat informace. Například ve škole, mohou být entitami: učitel, žák, učebnice, poznámka, výplata, atd.
- **Atributy** - vlastnosti, které entitu konkrétně popisují. Učebnice může mít atributy: název, předmět, rok vydání, atd.
- **Relace** - tabulka vyjadřující vztah mezi entitami.

- **Primární klíč** - jeden či více atributů, které jednoznačně identifikují konkrétní záznam. Častým řešením je přidání do tabulky identifikátor, unikátní číslo napříč systémem.

V relačním databázovém modelu existují tři základní typy binárních relací, tedy relací mezi dvěma tabulkami. Typ relace A:B vyjadřuje způsob, jakým se záznamy z tabulky A mapují na záznamy z tabulky B:

- **One-to-One (1:1)** - mistr vyučuje vždy pouze jednoho učně a učeň jiného mistra nemá.
- **One-to-Many (1:M)** - jeden učitel vyučuje na prvním stupni ZŠ pouze žáky jeho třídy a ti jiného učitele nemají.
- **Many-to-Many (M:M)** - učitel na střední škole učí více tříd a třídy mají více učitelů na různé předměty.

Structured Query Language (SQL), jak zmiňuje [4], je standartním jazykem DBMS. Jedná se o interaktivní dotazovací jazyk. Není procedurální, ale deklarativní. To znamená, že uživatel vytvářející dotaz, přímo neřeší, jakým způsobem dosáhnout výsledku, ale deklaruje, jaký výsledek chce získat. DBMS dotaz zpracuje, nejlepším způsobem vyhodnotí a získá data z databáze. Pro ukázku si uvedeme příkazy:

- **SELECT** - získá z databáze data, jejichž podmínky uživatel specifikuje.
- **INSERT** - vloží nová data do databáze.
- **UPDATE** - upraví existující data v databázi.
- **DELETE** - smaže data z databáze.
- **COMMIT** - permanetní uložení změn databáze.
- **DROP TABLE** - smaže celou databázovou tabulku.

Existuje spousta dalších principů, týkajících se relačních databází. Transakce, normalizace databázových schémat, ad. V kontextu této práce je však nebudeme uvádět, zmínili jsme jen základní. Relační databázový model je robustním a jednoduchým rešením a v době jeho vzniku převálcoval stávající řešení. V dnešní době však existují alternativy, které mohou být výhodnější v určitých kontextech použití.

3.2.2 Dokumentové databáze

Jako důvod, vedoucí k vzniku nových typů databází, uvádí [7] omezení flexibility a škálovatelnosti relačních modelů. Jejich vývoj popisuje autor takto: *„Programátoři byli nespokojeni s řešením objektově–orientovaného mapování na relační model. Tento přístup byl při ukládání komplexních objektů nedostačující. JSON v té době přerostl XML a stal se způsobem jak serializovat data. Nějaké systémy začali do sloupečků relační databáze ukládat data v JSON formátu. Bylo jenom otázkou času, kdy někdo odstraní přebytečnou vrstvu relační databáze a vytvoří databázový systém, ve kterém budou data ukládána přímo v JSON formátu. Těmto databázím se začalo říkat dokumentové.“* Programátoři si dokumentové databáze oblíbili, jelikož odpadá proces překládání objektů do relačního formátu.

Relační databáze uživatele nutí data dekomponovat a ukládat odděleně v jednotlivých tabulkách, které jsou propojeny relacemi. Dokumentové databáze v tomto ohledu dávají volnost a umožňují ukládat do jednoho dokumentu opakovaně zanořené objekty. Data mohou být uložena typicky v XML nebo JSON formátu, z hlediska úspornosti však v praxi používáme většinou JSON.

Projekt, který si na trhu nerelačních databází získal dominatní pozici, je dle [7] MongoDB. Jedná se o open-source projekt a k dnešnímu dni nejčastěji používanou alternativou k relačním databázím jako jsou MySQL či Oracle. Dotazovací jazyk MongoDB je založen na principu Javascriptu a JSON, plní obdobnou funkci jako SQL v relačních DBMS.

3.3 Perzistence s JPA

Ve vícevrstvých architekturách softwarových systémů nám perzistentní vrstva leží mezi databázovou vrstvou a logikou aplikace. Ve světě programovacího jazyku Java, jak zmiňuje [13], je přímo pro tuto vrstvu navržena technologie JPA (Java Persistence API), která nám zprostředkovává spojení mezi Javou a relačními databázemi.

3.3.1 ORM

ORM (Object-relational mapping) je technologie, která je součástí JPA, ale princip se vztahuje na všechny OO jazyky. Primárním cílem ORM je propojení OOP a modely relační databáze, jak popisuje [13]. V Javě vytvoříme třídu s názvem a atributy, která bude odpovídat tabulce v relační databázi. Tento typ třídy se nazývá entitní třída. Vztahy jsou mezi tabulkami definované také v rámci tříd. Aby mohla být třída rozpoznána ORM jako entitní, musí být v případě Javy správně oanotované. Anotace nám specifikují vlastnosti tabulky, atributů a relací, podle kterých nám poté ORM vygeneruje databázové tabulky.

3.3.2 Entitní manažer

Entitní manažer (ang. Entity Manager), je další z technologií zahrnutých v JPA. Abstrahuje nám databázový systém a jeho konkrétní implementaci. Programátor tak nemusí vytvářet žádné SQL dotazy a komunikovat s databází přímo. Místo toho nám tato abtrakce přímo v jazyce Java dává k dispozici metody pro manipulaci s daty. Je tak mnohem komfortnější práce s daty a odpadá nutnost znalosti SQL dotazovacího jazyka.

3.4 Bezpečnost

Správné uchování dat, udržení soukromí uživatelů, správné uchovávání hesel a další obdobné funkcionality, to vše spadá do oboru bezpečnosti. Jedná se o komplexní a stále se vyvíjející obor. Každá aplikace, systém, platforma, apod. má na bezpečnost jiné nároky. Určují se podle citlivosti a množství dat nebo hodnotě informací, které může útočník získat. V kontextu naší aplikace si do detailu rozebereme bezpečné uchovávání hesel. [12]

3.4.1 Uchovávání hesel

V dnešní době, mají rychle se rozšiřující informační technologie za důsledek stále rozsáhlejší práci s daty. Tím, co naše data chrání před zneužitím, jsou hesla. Ty nás chrání způsobem, že konkrétní informace zpřístupní pouze tomu, kdo heslo zná. Dle [9], který tvrdí že: *„Bezpečnost hesel se stala kritickým problém víc než kdy předtím. Obzvlášť s tolika hackery, kteří je chtějí zneužít.“* [9] zastává názoru, že žádná aplikace není plně zabezpečená. Proto bychom měli myslet na scénáře, kdy útočník získá přístup do databáze s uloženými hesly. Z tohoto důvodu je nezbytné hesla chránit nejvyšším možným stupněm bezpečnostních mechanismů a zabránit útočníkovi hesla zjistit. Postupně si uvedeme přístupy ukládání hesel, kde každý další bude o úroveň bezpečnosti dál, než ten předchozí.

Naivní přístup

Nejnaivnějším přístupem, kterým lze hesla ukládat, je v neupravené podobě. Při registraci uživatel zadá heslo, aplikace si ho převezme a uloží do databáze. Pokud tedy uživatel „Honza“ zadá při zakládání účtu heslo „heslo123“, v databázi bude vytvořen záznam z této dvojice, jak je naznačeno v tabulce 3.3. Při prolomení bezpečnosti databázových serverů se útočník pouhým nahlédnutím do tabulky uživatelů k heslům dostane.

Hashování

Prvním, o něco sofistikovanějším způsobem je, dle názoru [9], hesla hashovat. Hashování je jednosměrný proces převodu vstupních dat na náhodná výstupní

Uživatel	Heslo
Honza	heslo123
Petr	AivJeuCka

Tabulka 3.3: Naivní ukládání hesla

data konstantní bitové délky. Pokud tedy od uživatele aplikace získá heslo, projde nejprve procesem hashování a do databáze je uložena až výsledná hash. Pokud uvážíme stejný scénář jako při naivním přístupu a hashovat budeme například algoritmem SHA-1, výsledný záznam bude vypadat jako v tabulce 3.4. Existují však tzv. Rainbow tabulky, což jsou velké databáze předpočítaných hashů z obrosvkého množství standartně používaných hesel. Tímto způsobem lze do Rainbow tabulky nahlédnout a získat originální heslo z hashe okamžitě. K vyzkoušení je k dispozici jedna z mnoha Rainbow tabulek na tomto odkazu: https://md5decrypt.net/en/Sha1. Pokud v tabulce zkusíme dohledat hash Honzova hesla, okamžitě dostaneme původně zahadné heslo. Heslo Petra v tomto případě v tabulce není, protože si nezvolil standartně používané heslo. Nejsme tak schopní, v jeho případě, původní heslo získat.

Uživatel	Heslo
Honza	849b28dcbe2c37b2c60d994e5dbd4b21535d0701
Petr	5043dd9e6371a6756d366ccce3eff42603e6c644

Tabulka 3.4: Ukládání hashovaného hesla

Hashování se solí

Abychom se vyvarovali útokům přes Rainbow tabulku, zavádíme tzv. solení. Jak popisuje [9]: *„Jak už jste správně odhadli, jedná se o techniku, kdy k naším hashím ještě něco přidáme, tak aby byly těžší na prolomení.“* Jedna z vlastností bezpečného hash algoritmu je fakt, že pokud minimálním způsobem změníme vstupní data, výsledná hash bude kompletně jiná. Tuto vlastnost zde využijeme. V našem ukázkovém příkladu opět získáme od uživatele „Honza“ heslo „heslo123“. Zároveň vygenerujeme pro každého nového uživatele sůl. Dle doporučení [9] alespoň 16-bytů dlouho. Pro uživatele jsme tedy vygenerovali např. „82f72719837619e4“. Postup je obdobný, jako v případě hashování bez soli, s tím rozdílem, že vstupem je heslo zřetězené s textovou reprezentací soli. V našem případě „heslo12382f72719837619e4“. Výstupní hash uložíme do databáze. Jak [9] zmiňuje, abychom byli schopni při přihlašování heslo ověřit, musíme znát sůl daného uživatele. Sůl tedy uložíme v tabulce společně s heslem. Vzniklý záznam vypadá jak je uvedeno v tabulce 3.5. Zlepšení úrovně bezpečnosti spočívá v tom, že je nepravděpodobná existence Rainbow tabulky

pro náhodně generovanou sůl. Pokud útočník vnikne do databáze, musí si ji vytvořit. I tento způsob je však náchylný k útokům hrubou silou.

Uživatel	Sůl	Heslo
Honza	82f72719837619e	f4a576bf0f73fa780824206a919ae6c79ca12db0

Tabulka 3.5: Ukládání hashovaného hesla se solí

Hashování se solí a faktorem obtížnosti

Hashovací algoritmy jsou velice rychlé, což je nežádoucí vlastnot. Útočník pak může použít útok hrubou silou. V tomto útoku jednoduše vyzkouší všechny možnosti. Dnešní výpočetní síla strojů dokáže vyzkoušet všechny možnosti i celkem dlouhých hashů v nebezpečně krátkém čase. Tento čas se vztahuje k prolomení jedné kombinace soli a hesla, tedy jednoho uživatele. Abychom tento čas dostali pod kontrolu, existují algoritmy jako BCRYPT, PBKDF2 a další. Mimo parametry hesla a soli je v těchto algoritmech přítomný ještě parametr náročnosti, který prímo ovlivňuje počet provedených iterací. Nastavením velké náročnosti můžeme čas pro výpočet jediné hashe drasticky zvýšit. Důležité je ale zvážit poměr bezpečnosti a použitelnosti. Záznam může v tomto případě vypadat např. jako v tabulce 3.6.

Uživatel	Sůl	Náročnost	Heslo
Honza	82f72719837619e	15	$2b$10$2rDOS4fWBk.JmQF...

Tabulka 3.6: Ukládání hesla s faktorem náročnosti

Kapitola 4

Analýza

V této kapitole si stanovíme požadavky. V souladu s nimi poté vybereme vhodnou platformu pro provoz aplikace. Na základě platformy pak učiníme technologické závěry.

4.1 Požadavky

V této části si shrneme výsledky získané průzkumem kolegy Marka Šulce. Zároveň se zamyslíme a odvodíme systémové požadavky, které jsou z hlediska softwarového inženýrství podstatné.

4.1.1 Uživatelské požadavky

Sběr uživatelských požadavků byl z většinové části úkol kolegy Marka Šulce. Pro účely této práce je však vhodné si zjištěné výsledky rozebrat a popsat i zde. Uživatelské požadavky byly sbírány systémem anonymních dotazníků, osobními rozhovory s potencionálními uživateli a roli hrála i naše vlastní zkušenost. V prostředí volejbalu se pohybujeme po monoho let a jsme tak dobře obeznámeni s potřebami volejbalové komunity a častými vzory chování.

Jako nejzákladnější uživatelský požadavek lze považovat poptávku po aplikaci pro mobilní zařízení. Český volejbalový svaz disponuje pouze webovou stránkou, primárně pro počítače. Webová stránka je dostupná i v mobilní verzi, ale ve velkém měřítku omezená dostupnými informacemi i uživatelskou přívětivostí. Dle výsledků dotazníku zmínila většina respondentů minimálně jeden výrazný nedostatek aktuálního řešení. Tím nejčastějším důvodem je špatná struktura dat a velice obtížná dohledatelnost a orientace. Zároveň jsme zjistili, že dotazník vyplnilo 43% respondentů na mobilním zařízení. Z těchto informací lze potvrdit zájem o mobilní aplikaci a prostor pro zlepšení aktuálního řešení, hlavně v oblasti strukturalizace dat.

V otevřené otázce, co by uživatelé přivítali v aplikaci nejvíce, se většina shoduje. Mimo opětovně kladeného důrazu na přehlednost, si uživatelé přejí

systém oblíbených, který by prohledávání pro konkrétního uživatele mnohonásobě zjednodušil a zrychlil. Systém oblíbených spočívá v možnosti si vytvořit seznam soutěží, hráčů a týmů, na které bude mít uživatel k dispozici odkaz v rychle dostupné oblasti. Nemusí tedy opakovaně vyhledávat stejné soutěže a procházet tak celou strukturu soutěží. Najde ji pouze jednou a poté ji má k dispozici v systému oblíbených.

4.1.2 Systémové požadavky

První metrika, kterou se charakterizuje softwarový systém, je zátěž. Po konzultaci s ČVS odhadujeme počet uživatelů řádově v tisících měsíčně. Množství uživatelů, kteří budou aplikaci používat ve stejný okamžik, bude tak mnohem menší. Dalo by se tak odhadnout, že návalově by systém měl zvládnout obsloužit počet uživatelů řádově ve stovkách.

Z hlediska charakteristiky domény považujeme za důležitější modularitu systému. Projekt má velký potenciál pokračovat i v budoucnosti. Z hlediska zvyšujícího se množství uživatelů používající mobilní zařízení nelze opomenout, že by bylo vhodné integrovat do mobilní aplikace veškerý systém ČVS. Zároveň je možné, že se zvolené technologie časem ukáží jako nedostatečné a bude je nutné z části obměnit. Aby všechno zmiňované bylo možné, musí tak být systém navržený již od počátku. V aktuálním případě se jeví jako vhodný přístup vrstvená architektura, která má zajistit to, že bude možné jednotlivé vrstvy měnit bez většího zásahu do těch ostatních.

4.1.3 Požadavky na infrastrukturu

Standardním přístupem je dnes při vývoji používat verzovací systém. V dnešní době máme na výběr téměř jediného kandidáta, a to je Git. Používání Gitu je i součástí zadání této práce. Mimo ukládání všech verzí systému, nám hlavně umožňuje efektivně spolupracovat na vývoji. Již aktuální stav si to vyžaduje, jelikož na projektu spolupracujeme s kolegou Markem Šulcem. Zároveň nám Git otevírá spoustu dalších možností, které s tímto nástrojem přicházejí.

Druhým požadavkem, který si dáme z hlediska infrastuktury, je automatizace sestavování, testování a nasazování změn. Jelikož je předpokládané další pokračování tohoto projektu, je žádoucí automatizovat proces sestavování, testování a nasazování již od začátku. V souladu s metodikou vývoje softwaru a metodik CI/CD tak zajistíme vyšší míru důvěry ve vyvíjený software a ušetření lidských zdrojů při provádění opakovaných činností. Tímto způsobem se také zvyšuje pravděpodobnost odhalení chyb co nejdříve, což ale závisí ve značné míře na kvalitě testů.

4.2 Volba platformy/hostingu

Cílem je vybrat vhodnou platformu pro provoz našeho kompletního systému. Chceme provozovat server, který může být jak aplikační, tak se v určitém případě může jednat o Docker kontejnery. Ač ČVS disponuje vlastní databází, ze které bude aplikace brát většinu informací, chceme provozovat i databázi vlastní, pro data, která jsou specifická pro naši aplikaci nad rámec ČVS. Proto je ideálním scénářem, pokud by platforma umožňovala provoz i databázového serveru. Měli bychom tak vše na jednom místě a nemuseli mít separátní platformu pro aplikační servery a pro ty databázové. Rozeberme si tedy následující možnosti.

4.2.1 Heroku

Heroku platforma má zdarma program, jinak je účtována měsíčně za fixní částku. Částka je za jeden *„Dyno“*, což je jedna funkční jednotka aplikace omezená na 512 RAM. Heroku podporuje následující, pro nás zajímavé, programovací jazyky:

- Node.js
- Ruby
- Python
- Java
- PHP

První dojem se schází s dojmem po týdenním používání. Platforma je velice zmatečná. Velké množství funkcionalit není v rámci Heroku platformy, ale musí se přidat extra. Ani po týdnu se nám nepodařilo úspěšně spustit testovací aplikaci. Zdá se, že Heroku sice umožňuje mít velikou kontrolu na všemi elementy systému, ale to je zároveň jeho nevýhoda. Aplikaci nelze jedoduše nahrát na předem vybraný server a spustit. Je zapotřebí sepsat *„Procfile“*, což je soubor, který ovládá procesy uvnitř obecného serveru. Aplikační server musíme tedy nejprve nainstalovat, nakonfigurovat a spustit. Subjektivní dojem je, že Heroku platforma není vhodná pro klienty, kteří chtějí jednoduchost a přímočarost, ale pro klienty, kteří chtějí plnou kontrolu.

4.2.2 Jelastic

Jedná se o software, který je charakteristický tím, že je účtován dle aktuální spotřeby. Většina konkurenčních platforem si účtuje fixní částku měsíčně. Jelastic je tedy platforma, ale není poskytovatelem. Existují řádově desítky poskytovatelů celosvětově, kteří tuto platformu poskytují. Liší se lokací, což

ovlivňuje rychlost odezvy serverů, a cenou, kde si každý poskytovatel platformy Jelastic určuje vlastní cenu.

Jelastic nabízí rozsáhlou škálu technologií a zaměřuje se na to, mít opravdu vše na jednom místě. Začneme–li od spodu hierarchie informačního systému, jedná se o následující:

- **Virtuální stroje** - Debian, Ubuntu, CentOS.
- **Databáze** - MySQL, PostgreSQL, MariaDB, MongoDB, aj.
- **Aplikační servery** - Java (Glassfish, Tomcat, Payara), Node.js, Python, PHP, aj.
- **Vyvažování zátěže**
- **SSL**

Subjektivně působila platforma po týdnu používání velice intuitivně. Při testování je systém účtování za spotřebu vítaný. Jelikož se platí zanedbatelná částka, pokud server zapínáme pouze při testování nových funkcí a vypínáme přes noc. To umožňuje si vyzkoušet desítky nových techologií bez nutnosti zaplatit si každou z nich dopředu.

4.2.3 Závěr

Existuje mnoho dalších provozovatelů, kteří provozují pouze služby pro webové stránky nebo pro izolované databázové servery. Nebylo nutné hledat dále po tom, co jsme vyzkoušeli platformu Jelastic. Její cenová politika nám umožňuje experimentovat s novými technologiemi za nízkou cenu. Intuitivní uživatelské rozhraní odstiňuje implementační detaily od uživatele a je vhodná pro klienty, kteří nemají dřívější profesionální zkušenost s provozováním vlastních systémů. Jelastic disponuje velmi dobře zpracovanou dokumentací s ukázkami kódu a obrázky. V ní nalezneme názorné návody pro všechny druhy technologií, které platforma nabízí. Jelikož ale Jelastic není provozovatelem svého softwaru, vybrali jsme provozovatele MassiveGrid.com, který tuto platformu poskytuje za nejnižší cenu a vlastní server ve Frankfurtu. Upřednostnili jsme tuto formu před poskytovatelem Unispace, který disponuje serverem v České republice, ale cenově je až třikrát dražší.

4.3 Databáze

V této části si zanalyzujeme poznatky, získané o typech databází a určíme, jaký bude pro naší aplikaci nejvhodnější. Již v rešerši jsme předpokládali dva typy jako favorizované. A to relační databáze a dokumentové databáze. To je v souladu s analýzou platformy Jelastic, která nám přesně tyto dva typy

nabízí. Mezi oběma typy databázových systémů máme pak výběr z mnoha implementací. Při srovnání relačních a dokumentových databází, kde v dokumentových převažuje jako formát JSON, jsme dospěli k volbě relačního přístupu. Důvodem je, že volíme Javu jako primární programovací jazyk pro implementaci aplikace. Java má mnoho nástrojů přizpůsobených právě pro práci s relačními databázemi, což by se mohlo v budoucnosti vyplatit. Dokumentové databáze jsou spíš novějším přístupem a nabyli jsme dojem, že se hodí pro jinou volbu technologií. Dokumentové databáze bychom volili za předpokladu, že by se jednalo o aplikaci, kterou by bylo nutné v budoucnosti intenzivně škálovat z hlediska složitosti. Jsme přesvědčeni, že pro aplikace, které mají menší velikost, je vhodné používat také relační databáze. V poli relačních databází máme více potencionálních zástupců. Analyzovali jsme názory komunity a konzultovali výběr se zkušenějšími kolegy. Dospěli jsme k závěru, že pokud nemá naše aplikace speciální požadavky, je vhodné zvolit MySQL. MySQL je nejpoužívanější open–source relační databázový systém a je také nabízený jako možnost na platformě Jelastic.

Kapitola 5

Návrh

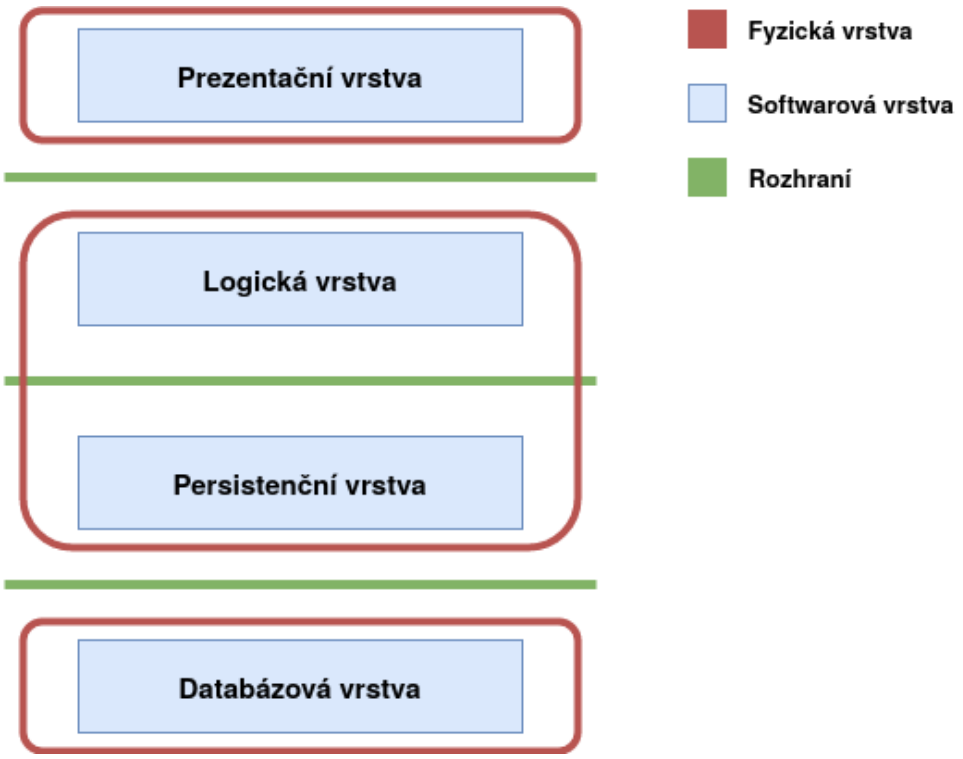
Volíme vícevrstvou architekturu, kde každá vrstva plní jednu z esenciálních částí systému. Vrstvy jsou uspořádané za sebou a každá využívá vrstvy přímo pod sebou. Vrstvy mezi sebou komunikují přes dobře definované rozhraní a případná změna v jedné nerozbije celý systém. Jako příklad si můžeme uvést situaci, kdy bude nutné vyměnit typ databáze. Architektura nám umožňuje po nastavení nové databáze a nahrání zálohovaných dat aplikaci, databázi přepojit. Zbytek aplikace na novou databázi přepojíme jednoduchou změnou konfigurace persistenční vrstvy. Databázi jsme vyměnili s minimálním zásahem do kódu. Důležité je si uvědomit rozdíl mezi typem vrstvev:

- **Fyzická vrstva** - je oddělena fyzicky od zbytku aplikace a typicky propojena přes síť.
- **Softwarová vrstva** - jedná se pouze o logické a funkcionální oddělení kódu.

Na obrázku 5.1 můžeme vidět popsané rozdělení fyzických a softwarových vrstev aplikace. Klientskou vrstvu tvoří zařízení, jako počítače nebo mobily, která se k aplikaci připojují. Serverová část tvoří jádro naší aplikace. Obsahuje aplikační server zpracovávající užívatelské požadavky dle definované logiky. Poslední vrstva, týkající se ukádání dat, bude oddělený server, spuštěn nezávisle na serverové části. Rozeberme si jednotlivé fyzické vrstvy.

5.1 Datová fyzická vrstva

Zodpovědnost datové fyzické vrstvy je uchovávat bezpečně data, oddělená od aplikace a definovat vhodné rozhraní pro jejich zpřístupnění. Toto rozhraní je typicky součástí databázových systémů, kde přes síť k datům přímo přistupujeme a následně čteme, či s nimi jinak manipulujeme. V našem případě bude databázová vrstva obsahovat dva typy databází, které jsou na sobě zcela nezávislé.



Obrázek 5.1: Celková architektura aplikace

5.1.1 Databázová softwarová vrstva ČVS

Prvním typem je databáze ČVS, který je majitelem těchto dat a databázi provozuje. Jsou to data, která budou tvořit základní informační hodnotu aplikace. Jedná se o veškerá data spojená se soutěžemi, zápasy, týmy, hráči, beachovými turnaji, reprezentačními turnaji apod. Nad touto databází nemáme kontrolu a je nám zpřístupněna pouze pro čtení. Rozebereme si jednotlivě všechny entity používané z této databáze v aplikaci. Entity bylo třeba ručně namodelovat podle dokumentace, která nám byla poskytnuta. Jsou použity pouze atributy, které aplikace skutečně využívá. Skutečných atributů je v extrémních případech a několik desítek.

Entita článku

Tato entita (viz tabulka 5.1) popisuje článek a všechna jeho metadata. Pomocí této entity jsme schopni zavést sekci zpravodajství, které bude určené právě pro čtenáře článků.

Atribut	Popis
nazev	Titulek článku
perex	Úvod článku, který dává uživateli kontext
obsah	Samotný obsah článku v HTML formátu
obr	Odkaz na hlavní obrázek článku

Tabulka 5.1: Model entity článku

Entita beachového turnaje

Popisuje turnaje, konané v rámci beachového volejbalu a jeho náležitosti ve strukturované podobě (viz tabulka 5.2).

Atribut	Popis
nazev	Titulní název turnaje s místem konání
body	Bodové ohodnocení turnaje
popis	Detailní popis turnaje
kategorie	Věková kategorie a pohlaví hráčů
datum_kvalifikace	Datum konání kvalifikační části turnaje
datum_turnaje_start	Datum začátku hlavní části turnaje
datum_turnaje_konec	Datum ukončení hlavní části turnaje

Tabulka 5.2: Model entity beachového turnaje

Entita zápasu

Obsahuje údaje ohledně zápasu odehraného, či naplánovaného v rámci soutěží šestkového volejbalu (viz tabulka 5.3). Pokud je zápas již odehraný, v entitě je uveden i výsledek. Pokud ještě není odehraný, obsahuje pouze předem známé údaje.

Atribut	Popis
kolo	Kolo soutěže do kterého zápas patří
domaci	ID týmu domácích
hoste	ID týmu hostů
cas1	Čas konání zápasu
vysledek	Výsledek zápasu zapsaný v definovaném textovém formátu

Tabulka 5.3: Model entity zápasu

Entita hráče

Tato entita popisuje základní údaje jakéhokoli hráče registrovaného v rámci soutěží šestkového volejbalu (viz tabulka 5.4).

Atribut	Popis
id	ID hráče, které se liší mezi různými soutěžemi
jmeno	Křestní jméno
prijmeni	Příjmení
gender	Pohlaví hráče (hodnota M nebo Ž)
narozeni	Datum narození v textovém formátu

Tabulka 5.4: Model entity hráče

Entita detailu hráče

V této entitě jsou zachycená data specifikovaná pouze u hráčů extraligových soutěží (viz tabulka 5.5). Detailnější data o hráčích jsou atraktivní zejména u hráčů nejvyšších soutěží a jsou používaná pro propagaci hráčů nebo tvoření grafiky při přímých přenosech.

Atribut	Popis
vyska	Výška hráče v centimetrech
narodnost	Národnost hráče
pozice	Volejbalová specialiaze
cislo_dresu	Číslo dresu

Tabulka 5.5: Model entity detailu hráče

Entita týmu

Entitu týmu (viz tabulka 5.6) je třeba si neplést s klubem. Klub je právnická osoba, která zaštiťuje více věkových kategorií a jejichž jménem se pak prezentují jednotlivé týmy klubu v soutěžích. Týmy jsou tedy skupiny hráčů ze stejného klubu, registrované v nějaké ze soutěží. Pokud hráč hraje ve třech soutěžích, musí být členem všech třech týmů, které jsou v této soutěží registrované. Týmy v soutěžích vystupují jménem klubu, pod který spadají.

Entita týmu v tabulce

Tato entita tvoří záznam týmu v celkové tabulce soutěže (viz tabulka 5.7). Obsahuje strukturované informace o posbíraných bodech, z již odehraných zápasů. Pomocí těchto informací můžeme sestavit celkovou tabulku soutěže,

Atribut	Popis
id	ID týmu unikátní napříč soutěžemi
skupina_cislo	ID týmu v dané soutěži
nazev	Plný název týmu většinou odpovídající názvu klubu
zkraceny	Zkrácený název
skupina_kod	Kód soutěže, ve které je tým registrovaný
skupina_id	ID soutěže, ve které je tým registrovaný

Tabulka 5.6: Model entity týmu

seřazenou právě podle množství sesbíraných bodů. V případě rovnosti bodů se pořadí určuje dle složitěji definovaného klíče. Podle tabulky se například po ukončení základní části určuje, jaké týmy postupují do play–off a jaké se utkají o sestup do nižší soutěže.

Atribut	Popis
p.	Výsledná pozice v tabulce
družstvo	Zkrácený název týmu
utkání	Počet aktuálně odehraných utkání
V3	Počet utkání vyhraných za tři body (3:0 nebo 3:1)
V2	Počet utkání vyhraných za dva body (3:2)
P1	Počet utkání prohraných za bod (2:3)
P0	Počet utkání prohraných za nula bodů (0:3 nebo 1:3)
body	Celkový počet doposud získaných bodů

Tabulka 5.7: Model entity týmu v tabulce

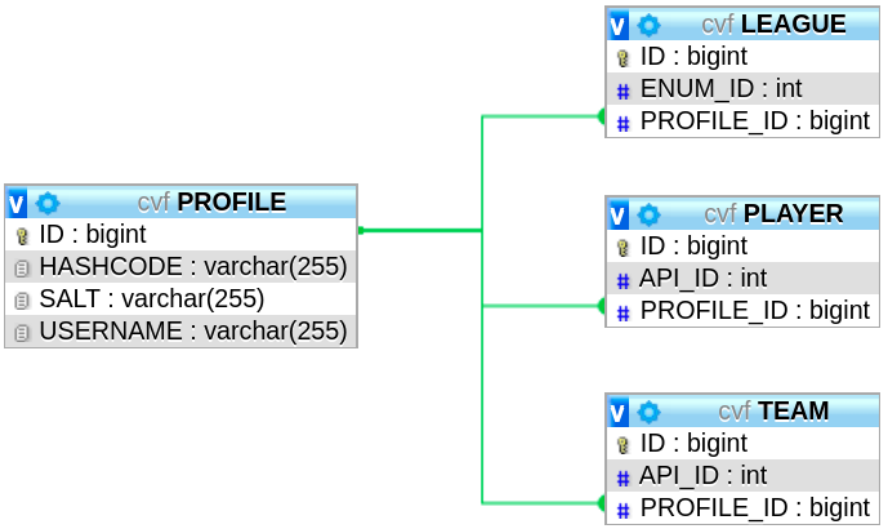
5.1.2 Databázová softwarová vrstva aplikace

Druhou částí databázové vrstvy jsou data, která bude naše aplikace uchovávat, kvůli potřebám konkrétních funkcionalit. Primárně se bude jednat o funkce vyplývající z uživatelských požadavků nebo funkce nutné pro dlouhodobou strategii. Z uživatelských požadavků určitě vyplývá uchovávání uživatelský účtů, pro implementaci mechanismu tvoření oblíbených položek, např.: týmů, hráčů, apod. Na obrázku 5.2 vidíme konceptuální model entit, nacházejících se v aplikační databázi, nad kterou máme plnou kontrolu a budeme ji sami provozovat. Model zajišťuje funkci oblíbených a správu uživatelských profilů.

5.2 Serverová fyzická vrstva

V serverové vrstvě běží samotná aplikace. V našem případě se bude jednat o aplikační server Glassfish, na kterém budou spuštěné naše aplikace napro-

gramované v Javě. Jednotlivé aplikace budou tvořit softwarové vrstvy architektury.



Obrázek 5.2: Relační databázový model aplikační části

5.2.1 Persistenční softwarová vrstva

Persistenční vrstva zjednodušuje nadřazené vrstvě manipulaci s databází. Typicky, ale ne nutně, tak činí pomocí API. S využitím HTTP požadavků můžeme dle možností API data vytvářet, číst, upravovat a mazat (tzv. CRUD operace). V našem případě budou v persistenční vrstvě, stejně jako v databázové, zastoupeny dva typy API.

API ČVS

ČVS má nad databází vytvořené funkční API, které má naše aplikace svolení používat. Přístup máme však pouze k podmožině dat, s oprávněním ke čtení (metodou GET). Z této skutečnosti také vyplývá jedna z potřeb vytvoření vlastní databáze, jelikož nemáme oprávnění v databázi ČVS cokoli vytvářet či upravovat. Tabulku dostupných a využívaných funkcí API si uvedeme v tabulce 5.8. Číslo ve složených závorkách nám označuje číslo parametru a pro každý tzv. endpoint je k dispozici pouze metoda GET.

URI endpointu	Popis
/hraci	Hráči kteří někdy hráli extraligu
/beach/turnaje	Registrované beachové turnaje
/redakce/clanky/last	Deset nejaktuálnějších článků
/redakce/rubriky/{1}/clanky	Články z rubriky {1}
/skupiny/{1}/zapasy	Ne/odehrané zápasy soutěže {1}
/skupiny/{1}/tabulka	Celková tabulka soutěže {1}
/skupiny/{1}/druzstva	Týmy registrované v soutěži {1}
/skupiny/{1}/druzstva/{2}/soupiska	Soupiska týmu {2} ze soutěže {1}

Tabulka 5.8: Přehled metod API ČVS

API aplikace

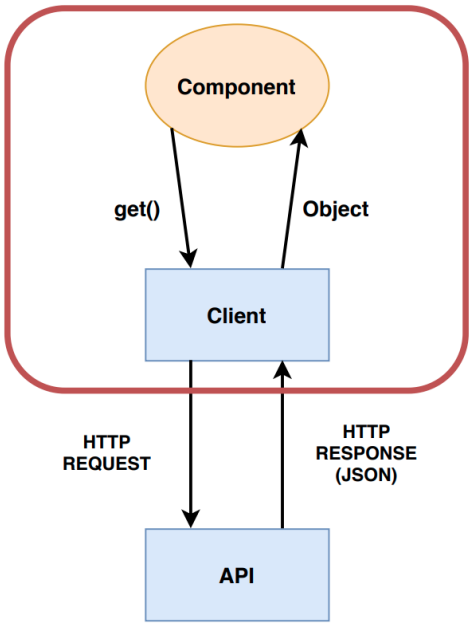
RESTové API vytvoříme i nad aplikační databází. V API budou zpřístupněny veškeré CRUD funkce: vytváření, čtení, editování a mazání entit. Narozdíl od připraveného API na straně ČVS si můžeme navrhnout vlastní strukturu a tím co nejefektivnějším způsobem umožnit nadřazeným vrstvám rozhraní používat. Nějaké funkcionality chybějící v API ČVS musíme totiž nahrazovat v logické části aplikace. To je nežádoucí, narušuje to architekturu aplikace a zvyšuje neefektivitu. Těmto nedostatkům se při návrhu našeho API chceme vyvarovat. Metody jsou navržené v tabulce 5.9.

Metoda	URI endpointu	Popis
GET	/profile	Všechny profily
POST	/profile/	Vytvoř nový profil
GET	/profile/{1}	Profil {1}
PUT	/profile/{1}	Změň profil {1}
DELETE	/profile/{1}	Smaž profil {1}
GET	/profile/count	Počet profilů

Tabulka 5.9: Přehled metod API aplikace

5.2.2 Logická softwarová vrstva

Tato vrstva tvoří hlavní funkcionalitu aplikace. Pokud se předchozí softwarové vrstvy soustředily pouze na zpracovávání a přístup k datům, v této vrstvě se konečně data využívají k tvoření komponent, ve kterých se budou data strukturovat a logicky organizovat. Logická softwarová část je dělena na dvě hlavní části, které se ale nedají považovat za samostatné vrstvy. Obě části jsou stále úzce svázány. Mluvíme spíše o logickém oddělení tříd v kódu dle jejich účelu.



Obrázek 5.3: Tok dat mezi komponentou a API

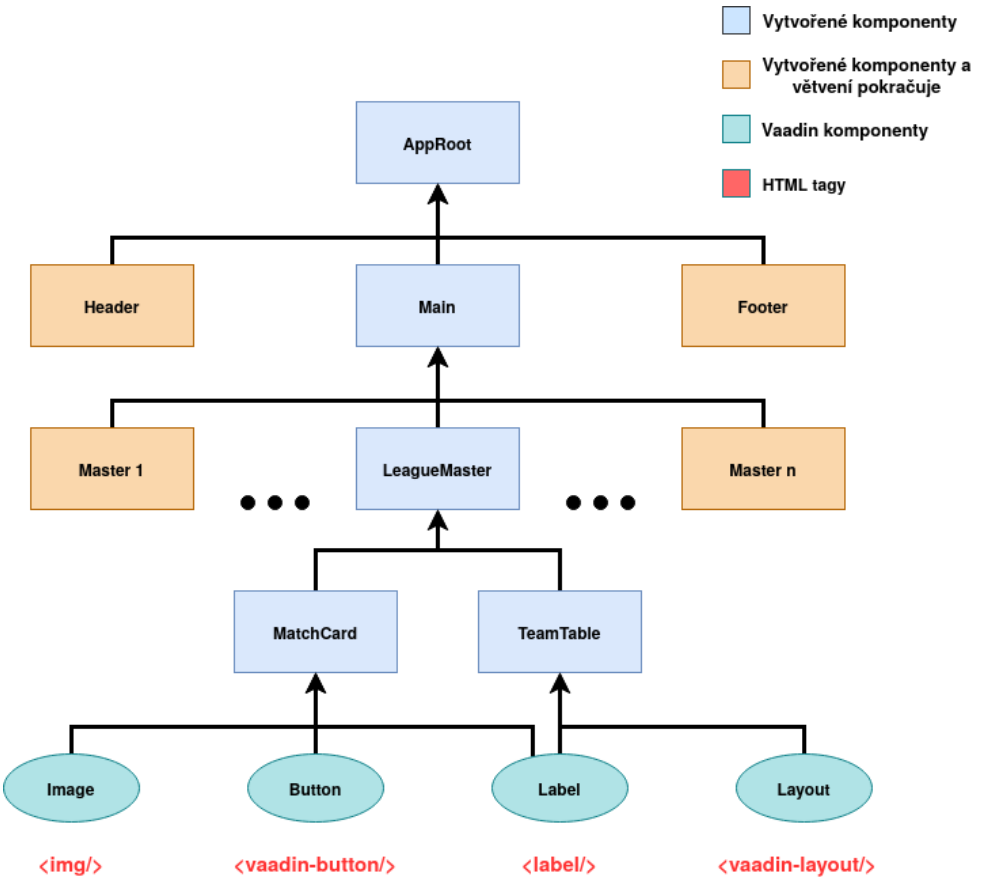
Klientská část

Jako spojový bod aplikační logiky s persistenční vrstvou bude skupina klientských tříd, což je první část logické vrstvy. Úkolem klientských tříd je vytvořit odpovídající HTTP požadavek s validní autorizací a odeslat ho persistenční vrstvě. Jako odpověď persistenční vrstva případně vrátí data, která mohou dorazit ve více možných formátech, my si volíme formát JSON. Dalším úkolem klienta je získaný JSON přečíst a vytvořit z něho odpovídající Java objekt v paměti aplikace. S vytvořenými objekty už mohou jednotlivé komponenty pracovat na vyšší úrovni abstrakce. Diagram tohoto procesu můžeme vidět na obrázku 5.3.

Komponentální část

Doteď jsme pouze zpracovávali data, v této části přicházíme k té hlavní podstatě vzniku aplikace. Jádro aplikace tvoří zpracovaná data, která jsou prezentována koncovému uživateli ve strukturované a intuitivně prohledávatelné formě. Umožňuje uživateli procházet aplikací a zjišťovat si potřebné informace. Pro získávání dat používají komponenty nejvyšší úroveň abstrakce, která je k dispozici, takže klientské třídy popsané v dřívější kapitole.

Pro tuto část volíme návrhový vzor kompozice, který je vhodný pro využití jak s Vaadin frameworkem, tak obecně pro aplikace tohoto typu. Jádro apli-



Obrázek 5.4: Ukázka části hierarchického stromu komponent

kace bude členěné do komponent, které se nechají vnořovat a opakovaně používat. Návrhový vzor definuje komponenty atomické, což jsou listy hierarchického stromu, ze kterých se nakonec ostatní komponenty skládají. Ty nazýváme komponenty vyššího řádu. V našem případě jsou dva způsoby, jak na problematiku nahlížet. Z hlediska Vaadin frameworku jsou atomické komponenty HTML tagy, které tvoří jednotlivé Vaadin komponenty. Oproti tomu jsou z hlediska aplikace, atomické Vaadin komponenty a z nich skládáme komponenty vyšších řádů, pro zobrazování informací například o hráčích. Přibližme si hierarchický strom diagramem 5.4.

V něm můžeme vidět, že kořenová komponenta je AppRoot, která slouží zároveň jako vstupní bod aplikace. Tvoří jakýsi rám aplikace. AppRoot používá tři základní komponenty prvního řádu:

- **Header** - tvoří horní panel aplikace.
- **Footer** - je navigační panel na dolní straně aplikace.

- **Main** - zabíra většinu displeje a tvoří hlavní obsah aplikace.

Uvniř Main komponenty se budou přepínat komponenty druhého řádu, Master komponenty. Zvolíte-li přehled soutěží definovaný uvnitř master komponenty soutěží nahradí se stávající master komponenta a obsah se zobrazí. Třetí a následující řády jsou již komponenty pro zobrazování dat a vznikají skládáním Vaadin komponent.

Na této úrovni se má práce potkává s prací kolegy Marka Šulce a komponenty si rozdělujeme následovně. Společně definujeme hierarchii komponent a schvalujeme, jaké by měli existovat a z jakých subkomponent by měly být složené. Společným úsilím tedy vzniká hierarchický strom, jehož část je uvedena v diagramu 5.4. Mým úkolem je komponentu napojit na persistenční vrstvu s pomocí klientů, tím získá přístup k datům. Zároveň řeším algoritmickou logiku komponenty, jako například hashování hesel, apod. Kolega komponenty finálně zapracuje do aplikace a dále upravuje jejich logiku a vzhled.

5.3 Klientská fyzická vrstva

Tuto fyzicky oddělenou část tvoří jednotlivá zařízení uživatelů. Jedná se o počítače, mobilní zařízení typu Android či IoS, tablety, aj. Aplikace je však zamýšlena primárně pro mobilní zařízení a bude tedy uzpůsobena zobrazení na vertikálně orientovaném displeji. S tím je i spojena v určité míře responzivita pro přizpůsobení se různým poměrům stran mobilních zařízení.

5.3.1 Prezentační softwarová vrstva

V našem případě bude prezentační vrstva aplikace nainstalovaná na zařízeních klienta, v určitých případech lze uvažovat i prohlížeč. Aplikace bude velice primitivní, z důvodu volby architektury. Aplikace bude v podstatě zrcadlit funkcionalitu prohlížeče a pouze přes celý displej zobrazí výstup z logické vrstvy. Komunikace mezi prezentační a logickou vrstvou zajišťuje z velké části Vaadin framework.

Kapitola 6

Implementace řešení

6.1 Implementace databázové vrstvy

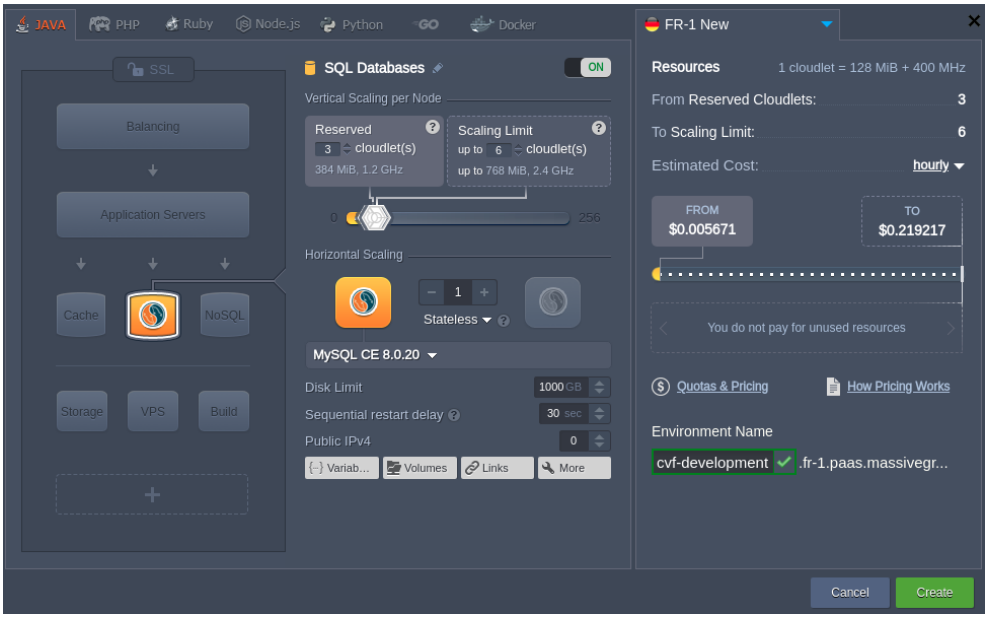
Z návrhu víme, že součástí aplikace budou dvě databáze. První je databáze ČVS, která obsahuje data o všech hráčích, týmech, soutěžích, atd. Druhou databází pak bude databáze implementována námi, která bude sloužit pro funkcionality spojené s aplikací. Budou to primárně data spojená s profily uživatelů a jejich výběrem oblíbených. Implementaci této databáze se budeme nyní věnovat.

Jako první část, kterou budeme v naší infrastruktuře zavádět, je DBMS server. Na něm poběží kompletní databázový systém. Z předchozích kapitol již víme, že naší volbou je relační databáze MySQL, kterou platforma Jelastic podporuje. Uvnitř plaftormy si tedy zavedeme nejprve nové prostředí (viz obrázek 6.1). V této fázi nás zatím z nabídky zajímá pouze databázová vrstva. Vybereme si MySQL a nastavíme parametry databázového serveru, jako je výkon nebo paměť. Následně necháme MySQL server v našem prostředí vytvořit.

Po úspešném vytvoření nám na e-mail byly zaslány přihlašovací údaje do DBMS, který se u MySQL databází nazývá *phpMyAdmin*. Po přihlášení do phpMyAdmin se nám otevřou všechny možnosti související se správou databáze. Nyní můžeme uvnitř databáze vytvářet nová schémata. Vytvořené tabulky nám korespondují s návrhem.

6.2 Implementace persistenční vrstvy

ČVS má k dispozici API, které je vybudované nad jejich databází. To nám umožňuje data získávat pomocí HTTP požadavků na nějakou z cest tohoto API, místo přímého přístupu do databáze. Podobný mechanismus chceme vybudovat i nad námi vytvořenou databází z předchozí kapitoly.



Obrázek 6.1: Vytvoření DBS na platformě Jelastic

6.2.1 Implementace API aplikace

Jako první krok je nutné namodelovat entitní třídy, které odpovídají schématu ve vytvořené databázi, tedy obrázku 5.2. Entitní třídy musí být korektně oanotované Java anotacemi. Technologie ORM pak bude schopná, na základě takto namodelovaných entit, komunikovat s databází.

Kromě ORM je ještě potřeba implementovat navržené metody v tabulce 5.9. Pomocí JPA a technologie entitního manažera, kterou má v sobě zahrnutou, se nám podařilo všechny metody dle popisu implementovat. Po poslání HTTP požadavku na endpoint API, se požadavek zpracuje a rozpozná jeho metoda. Je–li tato metoda implementovaná, požadavek se obslouží. Ukázku, jak byly metody implementovány, můžeme vidět v ukázce kódu 1.

6.2.2 Vytvoření serveru

V naší infrastruktuře nyní potřebujeme vytvořit aplikační server, na kterém poběží REST API napojené na naši databázi z předchozí kapitoly. Otevřeme si tedy platformu Jelastic a nastavení našeho, již dříve vytvořeného, prostředí. Vidíme, že nám zde již běží MySQL server. Z nabídky aplikačních serverů vybereme Glassfish 5.1 a nastavíme parametry jako je minimální výkon, maximální výkon, paměť, atd. Po vytvoření serveru je možné server spustit a otevřít v prohlížeči. Na serveru ještě není nahrána žádná aplikace, tudíž se nám zobrazí pouze defaultní oznámení, že je server v pořádku.

```
@Stateless
@Path("profile")
public class ProfileFacadeREST extends AbstractFacade<Profile> {
    @PersistenceContext(unitName = "cvf-rest_war-SNAPSHOTPU")
    private EntityManager em;

    public ProfileFacadeREST() {
        super(Profile.class);
    }

    @POST
    @Override
    @Consumes({MediaType.APPLICATION_JSON})
    public void create(Profile entity) {
        super.create(entity);
    }

    @PUT
    @Path("{id}")
    @Consumes({MediaType.APPLICATION_JSON})
    public void edit(@PathParam("id") Long id, Profile entity) {
        super.edit(entity);
    }

    @DELETE
    @Path("{id}")
    public void remove(@PathParam("id") Long id) {
        super.remove(super.find(id));
    }
 ...
}
```

Listing 1: Ukázka implementace REST API

Předtím než na něho nahrajeme REST API projekt, musíme server nakonfigurovat. Naším cílem je vytvořit JDBC spojení s databázovým serverem. Přihlásíme se tedy do konfiguračního panelu serveru na portu 4848. Přihlásíme se přihlašovacími údaji, které jsme obdrželi v e-mailu po vytvoření serveru. V postraní liště přepneme na záložku Resources a následně JDBC. V ní založíme nový Connection Pool dle návodu z dokumentace. Následně vytvoříme JDBC Resource napojený na tento Connection Pool.

Na JDBC resource můžeme nyní odkázat v našem REST API projektu a projekt nahrát na aplikační server. Projekt získá přes JDBC Resource a Connection Pool přístup do databáze. Na konkrétních URI si pak můžeme zkontrolovat, že aplikace opravdu vrací požadovaná data a chová se definovaným způsobem pro všechny endpointy API a jeho implementované metody.

6.3 Implementace logické vrstvy

V této části systému se nám konečně spojí oba datové proudy, jak proud ČVS, tak proud aplikační. Tato vrstva se rozděluje na dvě hlavní části. První je klientská, které má na starost získávání dat z persistenční vrstvy pomocí HTTP požadavků. Získaná data poté zpracuje a připraví k použití uvnitř komponent. Druhou část tvoří hierarchicky uspořádané komponenty. Zde konečně implementujeme logiku aplikace, jakým způsobem reaguje na vstupy uživatele, jaká data zobrazí za jakých pomínek a mnoho dalších funkcí.

6.3.1 Založení projektu

Jako vývojové prostředí si volíme Netbeans IDE, ve kterém bude provádět primárně úpravu zdrojového kódu. V Netbeans vytvoříme nový projekt, který se bude týkat frontendu aplikace, což bude jeho stěžejní část. Jako typ projektu zvolíme webovou aplikaci v Javě, což zajistí, že bude projekt spustitelný na jednom z aplikačních serverů Javy.

Zvolený aplikační server Glassfish si musíme nejprve stáhnout. Stáhneme si nejnovější verzi Glassfish 5.1 od Eclipse. Aplikační server poté naimportujeme do Netbeans IDE ve složce „Services“. Server nyní můžeme spustit a náasledně na něj nahrát prototypní aplikaci, kterou nám vygeneroval Neatbeans IDE.

Následně do projektu přidáme závislost na Vaadin 14, což je Java framework, kerý budeme pro implementaci frontendu využívat. Závislost přidáme do souboru „pom.xml“, který slouží pro veškerou konfiguraci projektu. Sestavíme základní kód který má ověřit, že je Vaadin 14 správně nainstalovaný a připravený k používaní v projektu. Jsme připraveni implementovat.

6.3.2 Klientská část

Jako příklad si uveďme situaci, kdy chceme získat informaci o hráčích ČVS s fiktivním jménem Pavel Novák. Klient nejprve vytvoří odpovídající HTTP

požadavek metody GET s autorizační hlavičkou, která zajistí přístup aplikace do API ČVS. Po odeslání požadavku na správnou URL v API přijde jako odpověď seznam všech hráčů. Je nutné získat všechny hráče, jelikož API hledání hráčů dle jmen nepodporuje. Klient je nakonfigurovaný tak, že odpověď vyžaduje ve formátu JSON. Entitu hráče musíme mít předběžně namodelovanou, tedy specifikovat, jaké atributy, jakého typu entita hráče obsahuje. Veškeré modely jsme si uvedli v kapitole návrhu. Model a získaná data zkombinujeme a nástroj Gson nám vytvoří inicializovaný objekt. Ten již můžeme používat ve zbytku aplikace. Celý tento proces je zachycený v ukázce kódu 2.

6.3.3 Komponenty

Komponenty tvoříme společně s kolegou Markem Šulcem, kde se on specializuje na design a já na práci s daty. Komponenty se pomocí Vaadin 14 frameworku tvoří postupnou kompozicí komponent. Pro každou komponentu je pak třeba spojit moji část, práci s daty a část kolegy, stylování a zapracování do designu aplikace. Na této úrovni je nutná úzká spolupráce a velmi častá komunikace.

6.3.4 Nahrání na server

Na rozdíl od předchozích vrstev, pro Vaadin projekt máme veškerou infrastrukturu připravenou. Na server nyní můžeme nahrát aplikaci, kterou jsme si připravili v předchozím kroku. Aplikaci si musíme nejprve sestavit. Sestavení pomocí Neatbeans IDE nám v tomto případě nebude stačit, jelikož je třeba provést produkční sestavení. Toho docílíme spuštením příkazu „mvn package -Pproduction“ v kořenové složce aplikace. Produkční sestavením nalezneme ve složce „target“. Jedná se o soubor s koncovkou „.war“. Otevřeme si platformu Jelastic kde máme v našem prostředí již vytvořený Glassfish server. Jako balíček vybereme produkční sestavení, tedy „.war“ soubor ze složky „target“.

Po nahrání Vaadin projektu na server si můžeme zkontrolovat výsledek v prohlížeči. Frontend si veškerá data bere ze dvou dostupných REST API, které zobrazuje uživateli pomocí komponent. Ty v sobě zahrnují i veškerou logiku při průchodu aplikací.

6.4 Implementace prezentační vrstvy

Z velké části se o prezentační vrstvu stará námi používaný Vaadin framework. Ten generuje HTML, CSS a hlavně Javascript soubory, které se odesílají uživateli například do prohlížeče. Vaadin se poté stará o komunikaci mezi uživatelem a naším serverem a obsluhuje akce, které uživatel vykonává. Finální design aplikace, který spadá právě do prezentační vrstvy, je v kompetenci kolegy Marka Šulce a je předmětem jeho bakalářské práce. Zde si vrstvu zmiňujeme pouze pro úplnost.

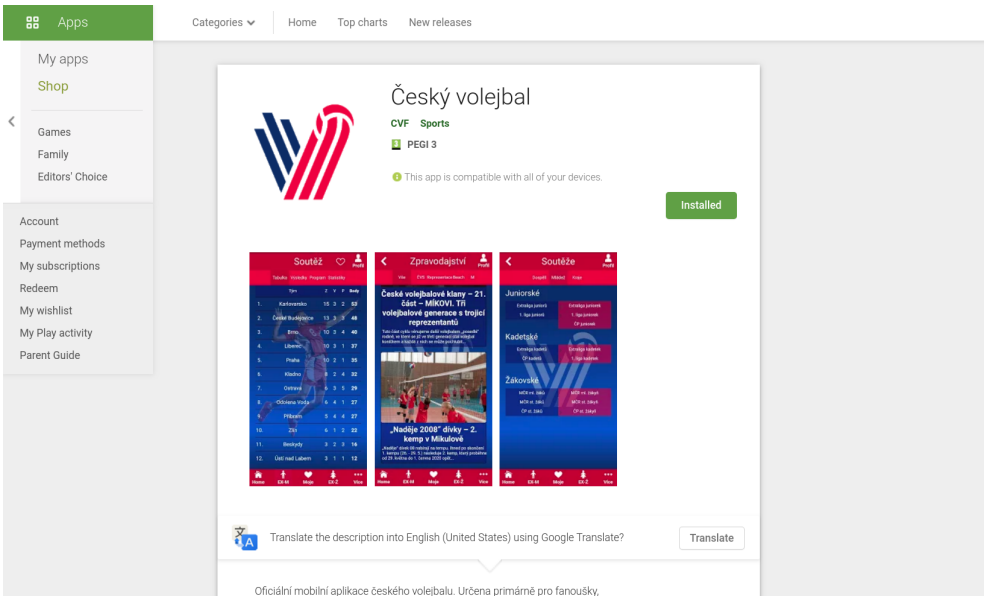
```
public abstract class Client {
    ...
    Gson converter = new Gson();

    protected <T> T getObject(String path, Class<T> type) {
        return converter.fromJson(getRaw(path), type);
    }

    private String getRaw(String path) {
        String response = "";

        // Connect to the web server endpoint
        URL serverUrl = new URL(URL + path + "?type=json");
        HttpURLConnection urlConnection =
            (HttpURLConnection) serverUrl.openConnection();

        // Set HTTP method as GET
        urlConnection.setRequestMethod("GET");

        // Include the HTTP Basic Authentication payload
        urlConnection.addRequestProperty("Authorization",
                                         basicAuthPayload);

        // Read response from web server
        BufferedReader httpResponseReader = null;
        httpResponseReader = new BufferedReader(
                new InputStreamReader(
                    urlConnection.getInputStream())
        );

        String lineRead;
        while(!(lineRead = httpResponseReader.readLine())) {
            response += lineRead;
        }
    }
    ...
}
```

Listing 2: Ukázka implementace klientské třídy

6.4.1 Mobilní aplikace

Posledním krokem, který nám ale chybí, je vytvořit mobilní aplikaci, kterou si může uživatel stáhnout do svého zařízení. Aplikace je v našem případě primitivní, jelikož bude pouze zobrazovat výsledky, které obdrží od našeho serveru. Mobilní aplikace je navržena tak, že neobsahuje téměř žádný nativní kód pro dané zařízení, ale pouze zobrazuje webovou aplikaci. Činí tak pomocí komponenty, která dokáže do okna aplikace nahrát obsah jakékoli webové stránky nebo webové aplikace. V našem případě tedy propojíme tuto komponentu s naší webovou aplikací.

Naším cílem je aplikaci dostat ke koncovému uživateli, musíme ji tedy publikovat na obchodě Google Play. Po založení účtu a zaplacení jednorázového poplatku 25$ máme k dispozici nástroj Google Play Console. Tento nástroj slouží vývojářům mobilních aplikací. Provádí je celým procesem publikace aplikace a nastavením její prezentace v Google Play. Po konfiguraci všech nastavení a vyplnění všech formulářů se nám podařilo předat aplikaci ke kontrole. Ta proběhla v pořádku a aplikace je nyní k dispozici v Google Play a ke stažení na zařízení Android.6.2



Obrázek 6.2: Aplikace publikovaná v Google Play

6.5 Konfigurace CI/CD

Nyní si ukážeme, jak kroky, které jsme museli aplikovat manuálně zautomatizovat pomocí Gitlabu. Gitlab umožňuje nakonfigurovat CI/CD, což nám

umožní provádět libovolné kroky automaticky, při každé další publikované změně zdrojového kódu. Tyto kroky se specifikují ve skrytém souboru „.gitlab-ci.yml“, který musíme umístit do kořenové složky repozitáře. Tyto kroky se následně provedou automaticky při každém dalším nahrání změn na Gitlab.

Kroky rozdělíme do tří kategorií. Sestavení, testování a publikace. První dva kroky budou prováděné automaticky při každé změně zdrojového kódu a budou validovat provedené změny. Kroky v rámci kategorie sestavení budou mít za úkol ověřit, zda se podařilo celou aplikaci sestavit. Kroky v rámci kategorie testování následně spustí veškeré testy a zvalidují funkcionální stránku aplikace. Více v kapitole testování. Poslední skupina kroků, publikační, bude mít na starost nahrávání či mazání aplikace na serveru na platformě Jelastic, který jsme konfigurovali v předchozím kroku.

Gitlab dává k dispozici virtuální stroj, na kterém všechny kroky běží. Na něm spustíme docker image a nainstalujeme potřebné aplikace. Nejprve se automaticky aplikace sestaví a otestuje. Následně máme možnost manuálně publikovat aplikaci. V tomto kroku se pomocí ssh připojení napojíme na Glassfish server na platformě Jelastic. Zkopírujeme produkčně sestavenou aplikaci na server a nainstalujeme pomocí aplikace „asadmin“. Změny se úspešně nahráli na server.

Kapitola 7

Testování

Poslední kapitolou této práce je testování. Dle teorie, kterou jsme si uvedli dříve, se testy dělí na statické a behaviorální. My se zaměříme na behaviorální testy, které zkoumají především to, jestli se systém chová v situacích správně. Konkrétně se zaměříme na dvě podkategorie behaviorálních testů, unit testy a integrační testy. Jelikož tato práce není zaměřena na design aplikace, což je kompetence kolegy Marka Šulce, nebudeme provádět testování uživatelského rozhraní. Zaměříme se na práci s daty, což je hlavní náplní této práce.

7.1 Statické testy

Statické testy byly aplikované zejména v průběhu návrhu a realizace aplikace, nedají se tedy efektivně zahrnout až do konečné fáze testování. Probíhaly primárně formou kontroly diagramů a kódu, kde jsme s kolegou Markem Šulcem využili již existující spolupráce a kontrolovali si tak diagramy a kód navzájem. Kontrola kódu byla realizována pomocí platformy Gitlab, kdy jsme před každým aplikováním změn do hlavní vývojové větve aplikace zkontrolovali, zda jsou změny funkční.

7.2 Unit testy

Unit testy zkoumají chování částí systému izolovaně. Protože jazykem, ve které byla aplikace vyvíjena, je Java, což je jazyk objektově–orientovaný, bude se jednat především o testování tříd. Pro tyto testy jsme si vytipovali tedy třídy, ve kterých má smysl testovat i jiné věci než rozhraní. Naše volba padla na:

- **Hashování** - tvoří stěžejní bod bezpečnosti uživatele a jeho hesla. Testováním hashování si tedy chceme ověřit základní vlastnosti počítaných a generovaných hodnot.

- **Inicializace kartiček** - kartička je komponenta, která nám v aplikaci zobrazuje entitu a její atributy. Chceme se tak ujistit, že kartičky se inicializují a entitu zpracují.
- **Inicializace master komponent** - podobně jako u kartiček si chceme ověřit, zda se master komponenta inicializuje. Navíc však tato komponenta obsahuje mnoho subkomponent, u kterých zkoumáme, zda se načtou spolu s ní.

Uvedeme si přehlednou tabulku 7.1, ve které může čtenář prozkoumat, jaké testy byly provedeny. Všechny testy, kromě čtyř testů na inicializaci master komponent, byly úspěšné. Po bližším prozkoumání důvodu, proč testy neprošly, bylo zjištěno, že se jedná o integrační problém a ne problém funkcionální. Komponenty byly tedy vyhodnocené také jako v pořádku.

7.3 Integrační testy

Tento druh testů ověřuje, jak mezi sebou komunikují komponenty systému. V našem případě se bude jednat primárně o testování komunikace mezi logickou a persistenční vrstvou aplikace. Zaměříme se na testování připojení k API Českého volejbalového svazu a měření jeho kvality. Cílem těchto testů je zkontrolovat, zda jsou ze všech datových zdrojů API úspěšně stažena data, a zároveň kvantifikovat, jakou rychlostí. To pomůže odhalit případné nedostatky v algoritmech a povede k jednodušší optimalizaci aplikace.

Stejně jako v části unit testů si výsledky uvedeme v tabulce 7.2. Můžeme si všimnout, že v metodách klientů PlayerClient a TeamClient se objevují časy ve vyšších jednotkách sekund. To je odpovídající, jelikož byla již při vývoji spozorována tato oblast jako problematická. Důvodem je velké množství požadavků, které se vykonávají. Velké množsví požadavků je mnohem problematičtější než velké množství dat. Protože časově nejdražší je při komunikaci vytvořit spojení mezi aplikací a API. Jediným řešením tak je počet požadavků minimalizovat. API ČVS však neumožňuje lepší způsob a tak jsme nuceni problém řešit až v budoucnosti. Mimo to jsou ale časy akceptovatelné a všechny datové zdroje jsou funkční.

Číslo	Skupina	Název	Splněno
1.1	Hashování	Různost vygenerovaných solí	ANO
1.2	Hashování	Konzistence hashování se solí	ANO
1.3	Hashování	Rozdíl hashe při rozdílu soli	ANO
1.4	Hashování	Rozdíl hashe při rozdílu hesla	ANO
2.1	Přihlášení	Přihlášení profilu do aplikace	ANO
2.2	Přihlášení	Odhlášení profilu z aplikace	ANO
3.1	Kartičky	Inicializace kartičky beachového turnaje	ANO
3.2	Kartičky	Inicializace kartičky oblíbené	ANO
3.3	Kartičky	Inicializace kartičky zápasu	ANO
3.4	Kartičky	Inicializace kartičky hráče	ANO
3.5	Kartičky	Inicializace kartičky hráče v soupisce	ANO
3.6	Kartičky	Inicializace kartičky týmu	ANO
3.7	Kartičky	Inicializace kartičky týmu v tabulce	ANO
3.8	Kartičky	Inicializace kartičky týmu vertikálně	ANO
4.1	Master	Inicializace komp. článku	NE
4.2	Master	Inicializace komp. beache	NE
4.3	Master	Inicializace komp. beachového turnaje	ANO
4.4	Master	Inicializace komp. oblíbených	ANO
4.5	Master	Inicializace komp. domů	ANO
4.6	Master	Inicializace komp. soutěže	ANO
4.7	Master	Inicializace komp. soutěží	ANO
4.8	Master	Inicializace komp. přihlášení	ANO
4.9	Master	Inicializace komp. zápasu	ANO
4.10	Master	Inicializace komp. menu	ANO
4.11	Master	Inicializace komp. zpravodajství	NE
4.12	Master	Inicializace komp. hráče	ANO
4.13	Master	Inicializace komp. profilu	ANO
4.14	Master	Inicializace komp. hledání	ANO
4.15	Master	Inicializace komp. týmu	NE

Tabulka 7.1: Výsledky Unit testování

Číslo	Skupina	Metoda	Čas	Splněno
5.1	Klient zpravodajství	Inicializace	0.001s	ANO
5.2	Klient zpravodajství	Všechny články	2.367s	ANO
5.3	Klient zpravodajství	Nejžhavější články	0.369s	ANO
5.4	Klient zpravodajství	Beachové články	1.278s	ANO
5.5	Klient zpravodajství	Články o CVF	1.572s	ANO
5.6	Klient zpravodajství	Reprezentační články	0.672s	ANO
6.1	Klient beache	Inicializace	0.237s	ANO
6.2	Klient beache	Všechny turnaje	0.001s	ANO
6.3	Klient beache	Turnaje mužů	0.001s	ANO
6.4	Klient beache	Turnaje žen	0.001s	ANO
6.5	Klient beache	Turnaje juniorů	0.001s	ANO
6.6	Klient beache	Turnaje juniorek	0.001s	ANO
7.1	Klient soutěže	Inicializace	0.001s	ANO
7.2	Klient soutěže	Všechny zápasy	0.291s	ANO
7.3	Klient soutěže	Naplánované zápasy	0.298s	ANO
7.4	Klient soutěže	Odehrané zápasy	0.296s	ANO
7.5	Klient soutěže	Tabulka soutěže	0.141s	ANO
8.1	Klient hráče	Inicializace	0.001s	ANO
8.2	Klient hráče	Všichni hráči	33.03s	ANO
8.3	Klient hráče	Kartičky všech hráčů	30.29s	ANO
8.4	Klient hráče	Soupiska hráčů v soutěži	0.400s	ANO
9.1	Klient týmu	Inicializace	0.001s	ANO
9.2	Klient týmu	Všechny týmy	10.00s	ANO
9.3	Klient týmu	Kartičky všech týmů	7.366s	ANO
9.4	Klient týmu	Seznam týmů v soutěži	0.166s	ANO
9.5	Klient týmu	Mapa týmů v soutěži	0.379s	ANO
9.6	Klient týmu	Tým v soutěži	0.163s	ANO

Tabulka 7.2: Výsledky integračních testů a měření

Závěr

Podařilo se navrhnout vhodnou vícevrstvou architekturu, která je flexibilní a v budoucnu rozšiřitelná. Úspěšně se podařilo vybudovat infrastrukturu aplikace a automatizovat procesy, které souvisejí se sestavováním, testováním a nasazováním aplikace. Podařilo se implementovat aplikaci v rámci odvozených požadavků a také byla tato část aplikace zdárně propojena s částí kolegy Marka Šulce. Vznikla tak plnohodnotná a provozuschopná mobilní aplikace pro Český volejbalový svaz, kerá byla úspěšně otestována. Mobilní aplikace fanouškům umožňuje si prohlížet přehlednou formou data českého volejbalu na jejich mobilních zařízeních.

Práce byla nejnáročnější v oblasti vybudování infrastruktury. Založení databázového serveru s MySQL, spuštění aplikačního serveru GlassFish a jeho konfigurace, automatizovat proces sestavování, testování a nasazování pomocí CI/CD a nakonec sestavenou aplikaci registrovat do obchodu Google Play a publikovat. Bylo nutné navázat a přizpůsobit se již existujícímu API ČVS a plnit další jejich požadavky na aplikaci, o kterých bylo v průbehu jednáno. Zároveň byla práce charakteristická tím, že byla prací v kolaboraci s kolegou Markem Šulcem, bylo tedy zapotřebí spolupracovat také tímto způsobem. Spolupráce probíhala velmi efektivně na pravidelné bázi, kdy bylo nutné specifikovat společně aplikaci jako celek a zároveň konzultovat části, ve kterých se naše práce protínaly.

Volba vrstvené architektury aplikace se ukázala jako vhodná, primárně z důvodu možných úprav a rozšíření, v případě dalšího zájmu ČVS. V určitých místech má aplikace nedostatky ve výkonu, primárně z důvodu nevyhovujícího API, kde je třeba problémy řešit obklikou. Vzniká tak zbytečně komplikované a neefektivní řešení. Výhledem do budoucna je celou práci společně s kolegou Markem Šulcem prezentovat před výborem ČVS a navázat dlouhodobou spolupráci. Aplikace má potenciál, při dalších rozšířeních, zefektivnit administrativu svazu a celkově zmodernizovat český volejbal a přispět tak k atraktivitě sportu.

Během této práce jsem získal mnoho schopností, zejména z oblasti pro-

vozu aplikace. Zjistil jsem, že vývojem aplikace zdaleka práce nekončí. Tím, jak aplikace stavěla na již existujícím API Českého volejbalového svazu, bylo důležité neustále komunikovat potřeby aplikace a hledat řešení, což byla zajímavá zkušenost. Práce mě bavila také z důvodu, že společně s prací mého kolegy Marka Šulce utvořila větší a komplexnější projekt, než jaký bychom byli schopni vytvořit jednotlivě. Právě její sofistikovanost mě nutila se učit neustále novým věcem a technologiím a během doby, co jsem práci realizoval, posunula mé programátorské kvality mnohem dál. Získal jsem přehled ve spoustě oblastí informačních technologií a věřím, že mi získané zkušenosti pomohou v mém budoucím životě.

Bibliografie

[1] Rex Black. *Pragmatic Software Testing - Becoming an Effective and Efficient Test Professional*. [cit. 17. 3. 2020]. Wiley, 2016. ISBN: 978-04-7012-790-2. URL: http://web.a.ebscohost.com.ezproxy.techlib.cz/ehost/ebookviewer/ebook/ZTAwMHh3d19fMzY0Mjc0X19BTg2?sid=2f8c5615-357e-4950-af77-f29baa28f2ad@sdc-v-sessmgr02&vid=0&format=EB&rid=1.

[2] Frank Buschmann et al. *Pattern-oriented Software Architecture - A System of Patterns*. [cit. 1. 2. 2020]. Wiley, 1996. ISBN: 978-04-7195-869-7. URL: https://daneshjavaji.files.wordpress.com/2018/02/sznikak_jegyzet_pattern-oriented-sa_vol1.pdf.

[3] Entrust Datacard. "How Does SSL Work?" In: (). [cit. 14. 2. 2020]. URL: https://www.entrustdatacard.com/pages/ssl.

[4] Elvis C. Foster a Shripad Godbole. *Database Systems - A Pragmatic Approach*. 2. vyd. [cit. 27. 2. 2020]. Apress, 2016. ISBN: 978-14-8421-192-2. URL: https://link-springer-com.ezproxy.techlib.cz/content/pdf/10.1007%2F978-1-4842-1191-5.pdf.

[5] Erich Gamma et al. *Design Patterns - Elements of Reusable Object-Oriented Software*. [cit. 7. 2. 2020]. Addison-Wesley Professional, 1995. ISBN: 978-81-7808-135-9. URL: http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf.

[6] *GitLab CI/CD*. [cit. 25. 2. 2020]. GitLab. URL: https://docs.gitlab.com/ee/ci/.

[7] Guy Harrison. *Next Generation Databases - NoSQL, NewSQL and Big Data*. [cit. 4. 3. 2020]. Apress, 2015. ISBN: 978-14-8421-330-8. URL: https://link-springer-com.ezproxy.techlib.cz/content/pdf/10.1007%2F978-1-4842-1329-2.pdf.

[8] Jörg Krause. *Introducing Web Development*. [cit. 13. 2. 2020]. Apress, 2016. ISBN: 978-14-8422-498-4. URL: https://link-springer-com.ezproxy.techlib.cz/content/pdf/10.1007%2F978-1-4842-2499-1.pdf.

[9] Fernando Mavoral. *Instant Java Password and Authentication Security*. [cit. 11. 4. 2020]. Packt Publishing, 2013. ISBN: 978-18-4969-776-7. URL: https://ebookcentral.proquest.com/lib/techlib-ebooks/reader.action?docID=1572913.

[10] Ali Mili a Fairouz Tchier. *Software Testing - Concepts and Operations*. [cit. 3. 4. 2020]. Wiley, 2015. ISBN: 978-11-1866-287-8. URL: https://ebookcentral.proquest.com/lib/techlib-ebooks/reader.action?docID=4040909.

[11] Jeff Teh. "HTTP vs. HTTPS". In: (). [cit. 14. 2. 2020]. URL: https://seopressor.com/blog/http-vs-https/.

[12] John R. Vacca. *Computer and Information Security Handbook*. 2. vyd. [cit. 7. 4. 2020]. Morgan Kaufmann, 2013. ISBN: 978-01-2394-397-2. URL: https://ebookcentral.proquest.com/lib/techlib-ebooks/reader.action?docID=1195617.

[13] Yuli Vasiliev. *Beginning Database-Driven Application Development in Java™ EE - Using GlassFish*. [cit. 13. 3. 2020]. Apress, 2008. ISBN: 978-14-3020-963-8. URL: https://link-springer-com.ezproxy.techlib.cz/content/pdf/10.1007%2F978-1-4302-0964-5.pdf.

Příloha A

Seznam použitých zkratek

ČVS Český volejbalový svaz

OO Objektově–orientovaný

OOP Objektově–orientované programování

JPA JaVa Persistence Provider

ORM Object–relational mapping

API Application Programming Interface

REST Representational State Transfer

CI/CD Continuous integration/delivery

CA Certificate Authority

SSL Secure Sockets Layer

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON Javascript Object Notation

XML Extensible Markup Language

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `thesis.pdf` ..... text práce ve formátu PDF
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `orm` ..... namodelované entity
    - `rest` ..... zdrojové kódy RESTového API
    - `ui` ..... zdrojové kódy uživatelského rozhraní
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX