Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Implementace REST WS rozhraní pro hru FactOrEasy

Petr Kupka

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Petr Kupka

Informatika

Název práce

Implementace REST WS rozhraní pro hru FactOrEasy

Název anglicky

Rest WS Interface for FactOrEasy game

Cíle práce

Prostudujte hru FactOrEasy a zaměřte se na administrační rozhraní hry. Navrhněte vhodnou datovou strukturu pro uložení nastavení hry:
- počet hráčů,
- počet her které mohou hrát
- jejich počáteční kapitál

Pro účel editace hodnot navrhněte vhodné REST WS. Toto rozhraní implementujte v jazyce Java. Řešení bude samostatný EJB modul (war,ear) s vystavenými restovými službami.

Metodika

Metodika bakalářské práce je založena na podrobné analýze potřeb hry FactOrEasy. Znalos nabyté studiem budou zhodnoceny a na jejich základě bude definován současný stav v nastavení hry. Na základě konzultací s vedoucím definujte rozšíření tohoto nastavení. Pro toto nastavení navrhněte vhodný datový model. Ten implementujte a propagujte formou REST WS. Tyto REST WS konzumujte vhodným Rest clientem. Hotové řešení otestujte a popište získané výsledky z testů.

Doporučený rozsah práce

43 p.

Klíčová slova

FactOrEasy, REST WS, Rozhraní, Datový model

Doporučené zdroje informací

Dodá školitel

Předběžný termín obhajoby

2018/19 ZS – PEF (únor 2019)

Vedoucí práce

Ing. Josef Pavlíček, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 23. 11. 2018

Ing. Martin Pelikán, Ph.D.
Vedoucí katedry

Elektronicky schváleno dne 23. 11. 2018

Ing. Martin Pelikán, Ph.D.
Děkan

V Praze dne 04. 03. 2019

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci " Implementace REST WS rozhraní pro hru FactOrEasy" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.4.2019 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Josefu Pavlíčkovi, Ph.D. za vedení práce a mému bývalému zaměstnavateli společnosti Capgemini za poskytnutí prostoru a zkušeností k realizaci práce.

Implementace REST WS rozhraní pro hru FactOrEasy

Abstrakt

Teoretická část práce je zaměřena na popis co je to webová služba, co je to REST, jak se používá, principy REST a srovnává REST se SOAP. Dále je uvedeno krátké představení aplikace FactOrEasy a jako poslední je představen také nástroj Maven.

V praktické části je zachycen postup návrhu REST rozhraní pro hru FactOrEasy. Dále postup zvolení technolgií pro implementaci navrženého REST rozhraní a navržení aplikace jako takové, zejména vrstvy aplikace. Poté je popsána samotná implementace RESTového rozhraní pomocí jazyku Java, kde autor popisuje jak postupoval při programování aplikace, popisuje jednotlivé kusy kódu a jak aplikace funguje.

Na závěr je krátce popsáno testování apliakce a výsledky testování.

Klíčová slova: FactOrEasy, REST WS, Rozhraní, Spring, Java, Hibernate

Rest WS Interface for FactOrEasy game

Abstract

Theoretical part of thesis is focused on description what is web service, what is REST, how it is used, REST principles and compares REST vs. SOAP. Then there is mentioned short introduction of game FactOrEasy and as last tool Maven is described too.

In practical part is captured process of designing REST interface for FactOrEasy. Then there is process of choosing technolgies for implementation of designed REST interface and design the application, especially application layers. After that is description of the implementation of the REST interface in programming language Java, where author of thesis describes how he progressed in programming of the application, describes single pieces of code and how application works.

In the end there is short description of application testing a its results.

Keywords: FactOrEasy, REST WS, Interface, Spring, Java, Hibernate

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Vztah mezi internetem a jeho uživateli se od jeho počátku ohromně vyvinul. Začalo to pouze statickými HTML stránkami, které obecně jen pracovali s textem, obrázky a odkazy a pro komunikaci mezi uživateli byly vyvinuty chaty, fóra a taky email.

Během posledních pár let však společnosti jako Google, Twitter, Amazon a Facebook začali měnit mnoho aspektů našeho denního života. Používáme je na všech možných zařízeních jako počítače, telefony, tablety a další. Dnešní svět prostě žije internetem a službami, které nabízí. Je tedy několik důvodů proč tyto společnosti byly schopny stát se tak populárními jako jsou dnes. Jeden z nich je určitě to, že všechny mají veřejně dostupné API, které jim pomohlo rozšířit jejich služby do všech zařízení a na web. Kvůli obsluze miliónů API požadavek (request)ů za minutu museli tyto společnosti přijít s efektivní architekturou. A tak nastal čas RESTu. Nejen pro vývojáře se stal REST velmi zajímavým při vytváření tzv. mashupů což jsou aplikace, které vytvářejí novou službu kombinací stávajících služeb, protože díky nim mohli jednoduše integrovat data z existujících služeb do služeb webových.

Dnes už většina veřejných API používá architektonický styl REST. Je to moderní styl, který je alternativou k webovým službám Simple Object Access Protocol (SOAP).

2 Cíl práce a metodika

Cílem práce bylo navrhnout vhodnou architekturu REST rozhraní pro administrační část hry FactOrEasy, zvolení frameworků pro implementaci rozhraní, návrh jednotlivých vrstev aplikace a samotné naprogramování RESTového rozhraní ve zvoleném programovacím jazyku Java a s ním spojených technologií.

Autor v práci představuje také teoretickou část, popis co vlastně jsou webové služby, jaké druhy jsou používány, rozdíl mezi jednotlivými druhy webových služeb a také správnými postupy navrhování a implementace webových služeb, kterými se autor řídil při programování RESTového API pro hru FactOrEasy.

Práce byla rozdělena do několika fází, jako první byl proveden návrh rozhraní. Návrh spočíval v správném využití metod HTTP protokolu (GET, POST, PUT a DELETE) pro získávání, zasílání, úpravu a mazání dat. Navrženy byly také jednotlivé vrsty aplikace dle ověřených způsobů tak, aby byla správně oddělená aplikační logika od přístupové vrstvy a také od vrstvy práce s databází.

Jako další krok byly zvoleny vhodné technologie pro implementaci. Programovcí jazyk Java byl daný, zvoleny byli hlavně frameworky aby aplikace odpovídala moderním trendům, byla rozšiřitelná a rychlá.

Samotné programování potom probíhalo implementacemi jednotlivých vrstev počínaje vrstvou prácující s databází, přes vrstvy aplikační logiky až po přístupovou vrstvu.

Po implementaci, byla pomocí REST klientů aplikace testována.

3 Teoretická východiska

3.1 Webové služby

Webová služba systém umožňující přenos dat pomocí protokolu HTTP. Pro komunikaci jsou používány hlavně XML a JSON formáty, které reprezentují zprávy, se kterými webová služba pracuje. Vzhledem k tomu, že ke komunikaci je používán protokol HTTP, který všechna přenáší v textové formě, je komunikace nezávislá na programovacím jazyku a prostředí, což je jejich velká výhoda, protože spolu mohou komunikovat např. dvě aplikace, které jsou napsané v odlišném programovacím jazyku, nebo jsou spuštěny na odlišným operačních systémeh.

Hlavní body, které by webová služba měla splňovat jsou:

- Je dostupná přes internet nebo intranet
- Používá standardizovaný formát ke komunikaci
- Není vázána na programovací jazyk či operační systém
- Je sebepopisující

3.2 Představení REST

Termín REST znamená Representational State Transfer a byl definován Royem Thomasem Fieldingem v jeho PhD disertační práci ”Architectural Styles and the Design of Network-based Software Architectures.” která byla publikována v roce 2000.

Podle této práce není REST softwarová architektura jako taková, ale je to soubor architektonických principů používané pro implementaci REST. Tyto principy mohou být shrnuty v následujících bodech: bezstavový klient/server protokol, jednotné rozhraní, univerzální syntaxe pro adresování a sebepopisující zprávy.

3.3 ROA

REST patří mezi architektury orientované na zdroje, zkráceně ROA (resource oriented architecture). Znamená to, že je celá architektura založená na konceptu tzv. zdroje (resource). Tímto pojmem označujeme u architektonického stylu REST jakýkoliv objekt uložený na síti, který lze v rámci systému identifikovat. Zdrojem mohou být jakákoliv data. Základní myšlenkou ROA je, že se nezabývá tolik způsobem, jak data měnit a zpracovávat.

Zabývá se spíše jejich strukturou, identifikací a reprezentací. Samotné operace, které se nad daty provádí, jsou vždy standardní.

3.4 Principy REST

3.4.1 Bezstavovost

Základním principem REST je bezstavovost. Princip bezstavovosti spočívá v tom, že požadavek(request) od klienta musí obsahovat všechny informace, které jsou potřeba pro jeho zpracování. Požadavek (request) nesmí spoléhat, že server již obsahuje některé inforamce potřebné ke zpracování například z předchozí komunikace. Bezstavovost tedy určuje zodpovědnost klientovi, který musí zasílat požadavek (request) se všemi informacemi.

3.4.2 Adresabilita

Tento princip říká, že každý zdroj by měl být dostupný přes unikátní adresu, která je vytvářena pomocí standardů pro vytváření takových adres. V praxi je tato adresa popsána použitím Unique Resource Identifiers (URI). Princip adresability říká, že všechny zdroje by měli být dostupné přes URI. Na webu to tak není vždycky, kvůli AJAX aplikacím, které mohou načítat obsah dynamicky bez změny URI. URI rozhraní by měli být navrhovány hiearchicky. Hiearchie zvyšují čitelnost a dělají jednoduší adresaci pro vývojáře. Je také možné uhodnout zdroj, který dělá jednoduším pro klienty objevit zdroj deduktivně.

3.4.3 Jednotné rozhraní

Princip jednotného rozhraní určuje, že rozhraní musí být vždy stejné a nesmí být závislé na prostředí. Princip jednotného rozhraní zajišťuje nepotřebnost zajišťovat pro každou novou aplikaci její připojení na REST. Technicky vzato, REST umožňuje použití nejednotného protokolu, ale protože HTTP nabízí všechny potřebné operace, je dobře známý a rozšířený, stal se pro REST služby standardem. Všechny interakce mezi klientem a zdroji jsou založené na základních HTTP metodách.

GET

GET je metoda HTTP, popisující požadavek na získání informací o zdroji. Na GET požadavek server odpovídá množinou hlaviček a reprezentací obsahující požadované zdroje.

PUT

Klient může poslat PUT požadavek (request), aby změnil existující zdroj nebo aby vytvořil nový na dané URI. Obvykle klient posílá reprezentaci spolu s PUT požadavkem (requestem). Server přečte reprezentaci a provede aktualizaci(update), resp. vytvoří zdroj pokud ještě na dané URI neexistuje. V porovnání s POST, PUT je více limitovaný operací, která nikdy nedělá nic víc než vkládá (put) na specifikovanou URI.

POST

Stejně jako PUT požadavek (request), POST požadavek (request) může vytvořit zdroj. Rozdíl je, že není vázaný na specifickou URI. Normálně je POST používán, když klient posílá data na server a server potom řekne klientovi kam vložit data. Server však může s POST požadavkem (requestem) udělat cokoli. Jak již bylo zmíněno může ukládat na URI jako PUT, ale může také poslat zpět HTTP hlavičku nebo nedělat nic. To dělá z metody POST více flexibilní než metodu PUT.

DELETE

DELETE požadavek (request) je potvrzení, že zdroj má být smazán. Klient nepotřebuje posílat s požadavkem (requestem) žádnou reprezentaci.

3.5 SOAP vs. REST

Orientace na zdroje REST služeb je často porovnávána s principy SOAP (Simple Object Access Protocol), což je protokol používaný pro výměnu zpráv po síti. Zprávy v SOAP protokolu putují po síti v XML reprezentaci a pro komunikaci používají HTTP/S protokol.

V kontrastu s REST, který používá pro práci s daty pouze základní HTTP metody, SOAP může pracovat s libovolnými operacemi. SOAP se nestará o reprezentaci zdrojů a

jejich zveřejňování, ale poskytuje prostředky pro definování vlastních operací, které manipulují s daty a tím umožnuje větší volnost a méně omezení.

Základní body a rozdíly ve kterých jsou REST a SOAP porovnávány jsou :

1. REST je navržen tak, aby sloužil pro zvěřejnění dat narozdíl od SOAP, který slouží spíše pro jejich zpracování.
2. REST služby jsou méně strukturované, což zajišťuje lepší čitelnost pro uživatele, zároveň ale kvůli tomu nemá tolik možnosti jako SOAP, kde je možné přesně určit typ a formát zpráv.
3. REST je bezstavový, SOAP podporuje složitější komunikaci (je tedy stavový) .

Mezi hlavní výhody RESTu a důvod proč je používat je hlavně jednoduchý vývoj klientů. Díky tomu, že webové služby jsou založeny na HTTP protokolu je jejich vývoj jednoduchý, pro klienty hezky čitelný a vzhledem k tomu, že většina platforem a jazyků nabízí nástroje pro práci s protokolem HTTP, je možné konzumovat REST služby odkudkoliv.

Další a největší výhodou REST je libovolný formát zpráv. Oproti SOAP protokolu, který omezuje jeho uživatele na formát XML je REST naprosto otevřený všem formátům použitelných pro webové služby.

Nízká provázanost klienta a služby je další velkou výhodou RESTu. Díky jednotnému rozhraní není klient se službou tolik provázán a je tedy možnost například rozšiřovat REST rozhraní bez toho, aby byla narušena funkčnost současných klientů.

Na závěr je nutné uvést také jednoduchost, díky které si REST uživatelé vybírají. Požadavky i odpovědi jsou velmi jednoduché a lidsky čitelné.

3.6 O hře FactOrEasy

FactOrEasy® je manažerskou simulací, jejímž předmětem je řízení výrobního podniku. Simulace byla vyvinuta na PEF ČZU v Praze za účelem podpory výuky předmětů z oblastí managementu, ekonomiky podniku, ekonomie a finančního řízení. FactOrEasy® (FOE) je k dispozici ve dvou verzích: offline a online. Obě verze lze použít jak společně tak samostatně zejména ve výuce na středních a vysokých školách, příp. ve vzdělávacích kurzech pro dospělé. Vzhledem k výzkumu a vývoji v pozadí simulace je podložena výsledky výzkumů, vědeckými publikacemi a i nadále se s výzkumným využitím simulace počítá.

Předmětem simulace FOE je rozhodování při řízení výrobního podniku. Účastníci simulace soupeří s konkurenčními podniky, které představují buď další studenti (offline verze) nebo jsou tvořeny umělou inteligencí (online verze). Jde o simulaci tržního prostředí, kde hráč provádí několik rozhodnutí v každém jejím kole. Délku simulace lze počtem kol nastavit. Základní princip celé simulace tvoří tři rozhodnutí, která jsou činěna v každém kole, a jejich pořadí je pevně dané: nákup materiálu, výroba produktů a prodej produktů na trhu. Online verze navíc přidává další dvě rozhodnutí, která může hráč činit kdykoliv: investovat do rozšíření výrobních kapacit (postavit novou továrnu) a vzít si půjčku. Jednotlivá rozhodnutí jsou provázána a určují strategii každého účastníka. V průběhu sehrávky mohou na trhu materiálu či výrobků nastat různé situace, které mají jednak teoretický základ v ekonomii a managementu, ale zároveň se vyskytují i v běžné manažerské praxi. Účastníci simulace tak dostávají možnost „zažít“ tyto situace v bezpečném prostředí simulace, což za přispění výkladu kvalifikovaného lektora vede k lepšímu pochopení dané problematiky. Tento způsob výuky tak sleduje moderní trendy ve vzdělávání založeném na vlastních zkušenostech (tzv. experiential learning). Nedílnou součástí simulace jsou proto doplňkové materiály v podobě manuálů. K dispozici jsou jednak standardní uživatelské manuály a návody popisující pravidla a ovládání simulace, ale také didaktická příručka pro lektora. Ta obsahuje přehled didaktických možností simulace, popis různých situací, které mohou nastat, jejich teoretický základ a popis způsobů jejich vysvětlení studentům v průběhu simulace.

3.7 Maven

Celým názvem Apache Maven je nástroj pro správu a automatizaci tzv. buildů aplikací, vyvinut společností Apache Software Foundation především pro aplikace psané v programovacím jazyku Java.

Oblasti které by, dle jeho tvůrců měl Maven pokrývat jsou :

- usnadnění procesu buildování,
- jednotný systém buildování,
- poskytování kvalitních informací o projektu,
- poskytování direktiv pro „best practices“,
- poskytnutí transparentního přidávání nových funkcí.

Maven podporuje modularitu a může obsahovat tzv. pluginy. Adresářová struktura Mavenu má v kořenovém adresáři pom.xml soubor popsán v další části a ostatní adresáře mezi které patří src/main/java, který obsahuje kompilované java soubory, src/main/resources obsahující soubory konfigurační nebo další soubory potřebné k projektu, src/test/java, který obsahuje třídy testů a poslední src/test/resources který obsahuje ostatní soubory potřebné k testovacím souborům.

Základním kamenem Mavenu je Project Object Model, což je model, který popisuje softwarový projekt z pohledu zdrojového kódu a také závislost na externích knihovnách a proces buildování. Project Object Model je definován v souboru pom.xml, který lze najít v kořenovém adresáři Mavenu. Projekt může být složen z více projektů nebo modulů, z nichž každý má svůj pom.xml soubor, který dědí vlastnosti od nadřazeného souboru. Díky tomu lze pak projekt sestavit jedním příkazem.

Proces buildu aplikace v Mavenu má určitý životní cyklus. Tato možnost umožňuje spouštět různé pluginy v různých fázích buildu nebo skončit build v určité fázi. Výchozí cyklus je nastaven takto :

- process-resources
- compile
- process-test-resources
- test-compile
- test
- package
- install
- deploy

Důležiou součástí jsou také tzv. dependencies, což jsou závislosti na externích knihovnách. Jednotlivé knihovny jsou definovány dle groupId a artifactId, kde groupId jednoznačně identifikuje projekt a artifactId je jméno souboru, který se vytvoří z buildu. Dle definice závislostí Maven potom automaticky hledá a instaluje potřebné knihovny.

4 Vlastní práce

4.1 Návrh

4.1.1 Návrh REST rozhraní

Při návrhu je vždy důležité si jako první určit požadavky. Úkolem této práce bylo vytvořit rozhraní pro administrační část hry FactOrEasy. To znamená, že bylo zapotřebí, aby ze vzdáleného zařízení přes REST rozhraní byly zprovozněny funkce, které umí číst, upravovat, vytvořit a smazat nastavení hry.

Potom co se určí funkční požadavky je zapotřebí nadefinovat zdroj tak aby rozhraní pracovalo nad zdrojem reprezentující nastavení hry. Vzhledem k předchozím zkušenostem využil autor pro reprezentaci zdroje formát JSON.

Nad tímto zdrojem byla potřeba provádět tzv. operace CRUD.

CRUD znamená – Create (vytvořit), Read(číst), Update(upravit), Delete(smazat). Tyto metody bylo nutné namapovat na metody HTTP protokolu, přes který REST rozhraní komunikuje, aby bylo možné pomocí protokolu provádět změny v databázi.

Dále byly na řadě koncové body(endpointy), což přístupové body aplikace, vystavené na serveru pomocí nichž se uživatel rozhraní připojuje. Velmi podceňovaná věc bývá pojmenování koncový bodů(endpointů) neboli určení URI adresy. Pojmenování by mělo být, dle standardů, vždy název zdroje v množném čísle a samozřejmě v angličtině.

V případě této práce tedy jádro URI bude vždy /gamesettings. Pokud uživatel zavolá pouze jádro tedy host/gamesettings metodou GET, rozhraní vrátí všechna nastavení. Pokud je potřeba přistoupit k jednotlivým nastavením, nezáleží jakou metodou, je použita základní část URI a přidáno lomítko s číslem id nastavení se kterým chceme manipulovat. Např. pokud chce uživatel získat nastavení s id=1 zavolá /gamesettings/1 metodou GET. Tuto adresu může uživatel použít také pro ostatní metody, PUT pro úpravu daného nastavení a DELETE pro smazání. Pro POST metodu, tedy pro vytvoření nastavení se opět použije pouze /gamesettings, protože při vytváření není nutno uvádět id, to je většinou vytvořeno a přiřazeno až databází. V případě této práce to nebylo nutné, ale pokud by bylo potřeba přidat více parametrů je to možné, správná praktika je oddělení parametrů lomítky. Navržené endpointy jsou uvedeny v následující tabulce.

URI\Metoda	GET	POST	PUT	DELETE
/gamesettings	Vrátí všechna nastavení	Vytvoří nové nastavení	-	-
/gamesettings/{id}	Vrátí nastavení s daným id	-	Upraví nastavení s daným id	Smaže nastavení s daným id

Tabulka 1 - Navržené endpointy

4.1.2 Návrh technologií

Dalším bodem návrhu je zvolení technologií, které budou použity pro naprogramování rozhraní. Vzhledem k tomu, že celá aplikace FactOrEasy byla naprogramovaná v jazyce Java a také vzhledem k autorově zkušenosti s tímto jazykem byla Java určena jako programovací jazyk i pro REST rozhraní. Původně bylo zamýšleno, že API bude vloženo do stávající aplikace, ale nakonec bylo rozhodnuto, že bude REST API jako samostatný modul, což zjednodušilo práci i výběr dalších technologií a zjednodušilo použití moderních frameworků.

U enterprise aplikací se také používá nástroj pro správu a řízení buildů aplikace. Do nedávna byl v tomto standardem Apache Maven, což byl nástroj, který byl vyvinut z Apache Ant společností Apache Software Foundation a jeho úkolem bylo standaridizovat a zjednodušit buildování aplikace. Od svého počátku byl Maven nejlepším nástrojem pro build aplikací. V roce 2016 však byl vyvinut nástroj Gradle, který je konkurencí Mavenu. Velkou výhodou Gradlu je jeho rychlost a výkon. Gradle dokáže provést build aplikace mnohokrát rychleji než Maven. Nastavení a konfigurace Gradlu je však dost odlišná a vzhledem k tomu, že tato práce nevyžadovala výkon a rychlost při buildu autor zvolil Maven se jehož konfigurací již měl zkušenosti.

V Javě se dá REST rozhraní naprogramovat více způsoby. Nejznámější frameworky pro implementaci REST jsou JAX-RS a Spring.

Funkčnost obou frameworků je dost podobná, JAX-RS je však starší. Použití je celkem jednoduché a stále se používá. JAX-RS má více implementací, ze kterých nejznámější je Jersey. Jeho nevýhodou a zásadní věcí proč autor tento framework nevybral bylo, že je naprogramován pouze pro implementaci REST a pro další práci jako je práce s databází by bylo nutné naučit se a použít jiné technologie. Proto byl vybrán framework Spring.

Spring, který obsahuje nejen práci s REST, ale má také komplexní technologie pro naprogramování celé enterprise aplikace, je neustále vyvíjející se open-source framework,

poprvé napsán Rodem Johnsonem v roce 2002 a jeho hlavní náplní je využití návrhového vzoru Inversion of Control. Tento návrhový vzor funguje na principu přesunutí zodpovědnosti za vytvoření a provázání objektů z aplikace na framework. Aktuální verze 5 už se liší od první verze a to hlavně velikostí jelikož Spring framework má nyní mnoho částí, kde každá z nich má na starosti určitý segment programování.

Základní části jsou :

- Container basics - základní kameny springu, Inversion of Control
- Aspect Oriented Programming (AOP)
- Data access and transactions - práce s daty a práci s databází
- Spring model-view-controller (MVC) - vše co je potřeba pro implementaci klasické MVC aplikace

Část pro implementaci REST patří do jádra frameworku. Jedny z nejnovějších a nejzajímavějších částí Springu jsou Spring Cloud a hlavně Spring Boot. Spring Boot byl velký skok dopředu pro programátory a to hlavně kvůli autokonfiguraci.

V době před Spring Bootem musel programátor vše konfigurovat v xml souborech, aby aplikace mohla fungovat. S příchodem Spring Bootu tyto povinnosti úplně neodpadly, ale velmi se zjednodušily. Další velkou výhodou je vestavěný aplikační server Apache Tomcat, díky němuž není potřeba aplikaci nasazovat na aplikační server jak to bylo dříve, ale je pouze spuštěn příkaz a aplikace během pár vteřin běží bez dalších komplikací za což vývojářům springu patří velké poděkování neboť ostatním programátorů vývoj usnadnily a proto také byl Spring Boot zvolen pro vývoj REST API pro FactOrEasy.

Složitým výběrem byla také technologie pro implementace vrstvy, která pracuje s databází. Aplikace FactOrEasy je, jak bylo zmíněno napsaná v EJB a s databází pracuje pomocí JPA. JPA je standard programovacího jazyka Java, který umožňuje objektově relační mapování (ORM). To usnadňuje práci s ukládáním objektů do databáze a naopak.

JPA má ale, stejně jako JAX-RS, několik implementací. Nejpopulárnější a nejpoužívanější je Hibernate. S Hibernatem měl autor zkušenosti a na všech předchozích projektech ho použil a proto byl využit Hibernate pro mapování objektů i pro tuto práci.

Poslední úkol bylo zvolit jaká technologie bude využita pro CRUD operace s databází. Těžký úkol to ale nebyl, protože další geniální věcí od Springu je část Spring Data, konkrétně rozhraní JpaRepository. Toto rozhraní se dá jednoduše implementovat a tím s dají vyřešit základní operace s databází. Vzhledem k tomu, že pro REST API nebylo

nic jiného než základní CRUD operace potřeba byl Spring Data jasnou a nejjednodušší volbou.

4.1.3 Návrh architektury aplikace

Technologie byly tedy zvolené a na řadě byl návrh architektury aplikace jako takové. Jedním z požadavků bylo, aby byla aplikace rozdělená „do hloubky", což znamenalo, že měla mít více vrstev, aby bylo jednoduché hledat chyby, rozšiřovat aplikaci a aby bylo jednoduché v kódu hledat. Moderní aplikace jsou rozdělovány do těchto vrstev :

- **Prezentační vrstva:** Tato vrstva má být zodpovědná za komunikaci s klientem, tedy přijímaní požadavků a zasílání odpovědí.
- **Aplikační vrstva:** Aplikační vrstva by měla být zprostředkovatelem komunikace mezi prezentační vrstvou a perzistentní vrstvou, tedy měla by zpracovat požadavky klienta, které jí předá prezentační vrstva a zaslat je dále aby byly požadavky vyřízeny na perzistentní vrstvě . V této vrstvě by měla být uchovávána hlavní logika aplikace
- **Doménová vrstva:** Tato vrstva obsahuje doménové entity, tedy jednotlivé modely aplikace, které jsou vyjádřením objektů databáze.
- **Perzistentní vrstva:** Perzistentní vrstva obsahuje všechny třídy, které mají za úkol práci s databází.

Prezentační vrstva v této aplikaci znamenala koncové body REST API tzv. endpointy, které budou vystaveny na serveru a zpřístupněny uživatelům. Jako další vrstvu, která nepatří mezi základní vrstvy autor zvolil tzv. Facade. Facade je jeden z design patternů, používaný u objektově orientovaných aplikací. Česky fasáda slouží jako přístupový bod mezi prezentační vrstvou a zbytkem aplikace. Zajišťuje aby prezentační vrstva přímo nekomunikovala s aplikační vrstvou, která obsahuje veškerou logiku. Po fasádě byla aplikována aplikační vrstva, ve které je hlavním úkolem zajistit komunikaci s perzistentní vrstvou. Vedle aplikační vrstvy je vrstva domén a jako poslední byla navržena perzistentní vrstva.

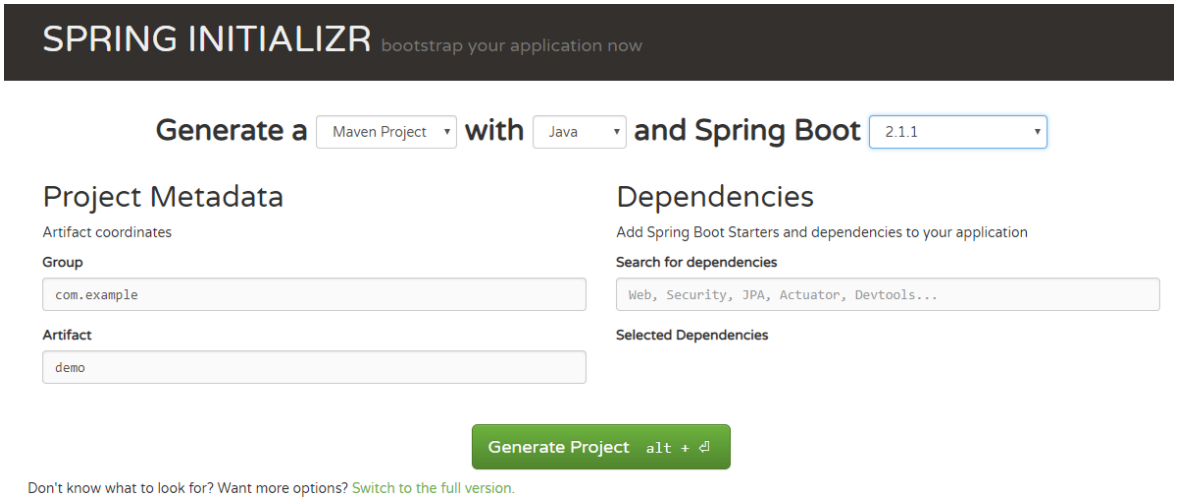
Po ujasnění všech požadavků, určení všech technologií a navržení architektury aplikace bylo zahájeno programování.

4.2 Implementace

4.2.1 Základní kostra aplikace

Jako první krok je nutné sestrojit kostru aplikace, se kterou pomůže Spring Initializr, který poskytuje Spring na svém webu. Tento nástroj na základě požadavků vygeneruje kostru aplikace postavenou na Spring Bootu. Stačí si zvolit maven nebo gradle, potom programovací jazyk, kde k výběru je Java, Kotlin nebo Groovy a verzi Spring Bootu. Dodatečná funkcionalita umožňuje nadefinovat si základní závislosti (dependencies).

Obrázek 1 - Spring Initializr



Spring Initializr vygeneruje složku aplikace, ve které lze najít pom.xml, což je základní soubor Mavenu, potom složku src, která obsahuje složky pro testy a hlavní složku, kam už patří zdrojový kód a vše co k je němu potřeba. Spring Initializr rovnou také vytvoří hlavní Java třídu pro Spring boot, přes kterou se dá celá aplikace spustit.
Tato třída je zatím velmi jednoduchá a stará se pouze o správné spuštění aplikace.

Obrázek 2 - Hlavní třída Spring Boot aplikace

```
@SpringBootApplication(scanBasePackages = "factoreasyrest")
public class Application extends SpringBootServletInitializer{

    public static void main(String[] args) { SpringApplication.run(Application.class, args); }

}
```

Třída je anotována anotací @SpringBootApplication, která označí třídu jako hlavní třídu Spring Boot aplikace a její parametr scanBasePackages určuje, který balíček má

Spring projít a vyhledat v něm Java Beans. Ve výše zmíněném pom.xml je nastaveno groupID a artifact ID, které byly uvedeny ve Spring boot initializru, což jsou identifikátory aplikace, které slouží hlavně proto, kdyby bylo potřeba použít závislotst na tuto aplikaci v jiném projektu.

Dále je nastaven balení aplikace(packaging), kde je nastaven formát do kterého bude aplikace při buildu přes Maven zabalena. Jako výchozí je nastaven formát jar, díky Spring Bootu a jeho vestavěnému aplikačnímu serveru lze totiž vytvořené jar spustit a aplikace funguje bez jakékoliv konfigurace či nasazení na server. Nastaveno může být také jméno aplikace.

Další částí pom.xml je uvedena část parent. Část parent nastavuje aplikaci tzv. rodičovskou aplikaci. U spring boot aplikací musí být jako parent nastaven spring boot starter parent, aby aplikace mohla správně fungovat.

Dále jsou nastaveny vlastnosti(properties), kde se obvykle určují verze závislotí nebo kódování.

Největší částí v pom bývají závislosti (dependencies). Dependencies jsou závislosti na vzdálených knihovnách, které určují, že projekt k jeho funkčnosti potřebuje třídy, které nejsou umístěny lokálně v projektu a pro správnou funkčnost je potřebuje získat, ať už z internetu nebo z jiného projektu. Spring initializr automaticky přidá závislost spring-boot-starter-web, která je potřebná pro správné fungování spring boot aplikace. Většina závislostí jsou pak přidávány až v průběhu implementace, díky moderním vývojovým prostředím, nemusí developer přidávat závislosti do pom ručně, ale vývojové prostředí nabízí přidání závislosti samo pokud je v kódu použita třída z některé ze vzdálených knihoven. U jednotlivých závislostí je vždy uveden identifikátor groupID, artifactID a verze.

Jako poslední je část build, ve které mohou být nastaveny různé pluginy a pomocné nástroje pro běh aplikace. Spring initializr přidá pouze spring-boot-maven-plugin, který je stejně důležitý pro běh aplikace jako určení parent aplikace spring boot parent.

Kostra apliakce je připravena a samotné programování může začít.

4.2.2 **Perzistentní vrstva**

Jako první z navrhnutých vrstev byla naprogramována perzistentní vrstva. Byl vytvořen balíček s názvem DAO neboli Data Access Object, který má obsahovat všechny

třídy perzistentní vrstvy. DAO je strukturální vzor umožňující izolaci aplikační logiky od zdroje dat. DAO představuje rozhraní díky kterému lze přistupovat k databázi.

Je obecným zvykem, že pro každou entitu je vytvořená Repository třída, což je třída která s entitou pracuje, tedy hlavně čte, upravuje a ukládá do databáze. V případě této práce, kde se pracuje pouze s entitou nastavení hry byla vytvořena třída s názvem FactOrEasyGameSettingsDAO. Jako první je dána třídě anotace @Repository která říká Springu, že tato třída je DAO objekt pracující s databází a umožní Springu tuto třídu injektovat. Anotace @Repository dědí od anotace @Component, což je základní anotace pro označení Java Beans ve Springu a je to konkrétní označení Java Beany, která má na starosti perzistenci objektů. Tato třída, dle návrhu, by měla umět vytvořit, upravit, smazat a číst entitu pro nastavení hry v databázi. Není potřeba žádná speciální funkce a tak díky Spring Data stačilo ze třídy udělat rozhraní, které bude dědit od JpaRepository.

JpaRepository je rozhraní patřící do Spring Data dědící od dalších tříd, díky kterým obsahuje všechny potřebné metody pro CRUD operace nad objektem a děděním od ní jsou zajištěny základní funkce pro práci s databází nad danou entitou. Pokud tedy programátor nemá speciální požadavky stačí vytvořit třídy, které dědí od JpaRepository pro každou entitu a má perzistentní vrstvu s CRUD operacemi hotovou, což ušetří mnoho práce. Zároveň ale JpaRepository umožňuje rozšíření funkcionalit. Je to chytrá třída, která pouze za pomoci názvu metody dokáže vytvořit například metodu pro vyhledání entity na základě jejího atributu a programátor nemusí nic programovat. Pokud například chce programátor vyhledávat všechny nastavení podle názvu stačí v DAO třídě oddděděné od JpaRepository vytvořit metodu findByNazev(String nazev), která vrácí seznam FactoriesgameSettings entit.

```
List<FactoriesGameSettings> findByNazev(String nazev);
```

Tento jeden řádek zajistí vyhledávání dle názvu entity. Pokud programátor potřebuje více než jen vyhledávat dle atributu dané entity, např. potřebuje vyhledávat podle atributu entity, kterou entita, právě implementující Dao třídy obsahuje nebo pokud potřebuje více zasahovat do sql příkazu, který jinak tvoří Spring, stačí vytvořit metodu anotovanou @Query a do závorek za anotaci napsat buď příkaz v JPQL, což je speciální jazyk odvozený z sql

```
@Query("SELECT u FROM User u WHERE u.status = 1")
Collection<User> findAllActiveUsers();
```

a nebo klasickou sql query, ke které se musí nastavit parametr nativeQuery na true.

```
@Query(
  value = "SELECT * FROM USERS u WHERE u.status = 1",
  nativeQuery = true)
Collection<User> findAllActiveUsersNative();
```

Toto jsou základní funkce, které pomohly a usnadnili práci na nejednom projektu. Oddděním DAO třídy od JpaRepository získá třída veškerou funkčnost, kterou má umět. Třída vypadá velmi jednoduše.

Obrázek 3 - Třída FactOrEasyGameSettingsDAO

```
@Repository
public interface FactOrEasyGameSettingsDAO extends JpaRepository<FactoriesGameSettings, Long> {
}
```

Při dědění JpaRepository musí být pomocí generiky řečeno, kterou entitu bude ukládat(FactoriesGameSettings) a datový typ id entity (Long). Protože entita FactoriesGameSettings už existuje ve stávající aplikaci FactOrEasy bylo by zbytečné vytvářet ji znovu a proto pouze stačilo přidat do Mavenu závislost na modul FactOrEasyCommon ,

```
<dependency>
	<groupId>eu.cz.culs.kii.Common</groupId>
	<artifactId>FactOrEasyCommon</artifactId>
	<version>1.0-SNAPSHOT</version>
</dependency>
```

kde se entita nachází a do třídy ji pouze importovat.

```
import eu.cz.culs.kii.common.factoreasycommon.dbentities.FactoriesGameSettings;
```

Tím je třída hotova a může být používána.

4.2.3 Aplikační vrstva

Nad perzistentní vrstvou je vrstva aplikační. V Javě se třídy v aplikační vrstvě nazývají Services nebo-li služby. Vedle balíčku DAO byl vytvořen balíček service, do kterého budou vytvářeny v budoucnu i další services pro ostatní funkce aplikace. Nyní je

ale hlavní úkol pouze vytvořit service pro herní nastavení. Dobrou praktikou v objektově orientovaném programování je použití rozhraní (interfaces), což jsou speciální třídy které pouze definují jak má třída implementující toto rozhraní komunikovat resp. jaké má vystavit metody, které půjdou na objektu tohoto typu volat. Bylo vytvořeno rozhraní FactOrEasyGameSettingsService a potřebné CRUD metody.

Obrázek 4 - Rozhraní FactOrEasyGameSettingsService

```
public interface FactOrEasyGameSettingsService {

    public FactoriesGameSettings read(Long id);

    public FactoriesGameSettings create(FactoriesGameSettings factoriesGameSettings);

    public FactoriesGameSettings modify(FactoriesGameSettings factoriesGameSettings);

    public void delete(Long id);

    public List<FactoriesGameSettings> readAll();

}
```

Potom je na řadě vytvořit klasickou Java třídu, která bude rozhraní implementovat. Aby se lépe orientovalo v kódu a rozlišilo se rozhraní od tříd, které je implementují, vytvořil se v balíčku service podsložka impl(implementation), kde budou umístěny třídy implementující rozhraní.

Třída je nazvána FactOrEasyGameSettingsServiceImpl. Stejně jako třída v perzistentní vrstvě musí být anotována, aby s ní Spring mohl pracovat, tentokrát ne @Repository ale @Service, což je také potomek anotace @Component, ale je určen pro service třídy. Dále je zapotřebí, aby třída implementovala dříve vytvořené rozhraní, což je provedeno přidáním slova **implements** za název třídy a za toto slovo název rozhraní, které má implementovat. Tím, že třída rozhraní implementuje musí implementovat i jeho metody.

V service třídě je potřeba volat perzistentní vrstvu, v této třídě konkrétně repository, které pracuje s objektem FactoriesGameSettings, aby bylo možné objekty ukládat, upravovat atd. v databázi. Aby repository mohlo být v service použito, je vhodné použít Dependency injection a injektnout repository třídu, tedy vložit závislost na repository do service třídy. Dependency injection je návrhový vzor, kterým lze implementovat Inversion of control, což je princip softwarového inženýrství, ve kterém je vytváření a kontrola objektů přenesena na kontejner nebo framework. Dependency

injection tedy zajistí odebrání zodpovědnosti třídám za získávání objektů, které potřebují ke své činnosti.

Tento krok lze díky Springu udělat více způsoby. Ke všem způsobům slouží anotace @Autowired, v každém způsobu je však použita jinde. Prvním ze způsobů je umístění anotace nad setter atributu injektované třídy.

Obrázek 5 - Ukázka injektnutí setteru

```
@Autowired
public void setFactOrEasyGameSettingsDAO(FactOrEasyGameSettingsDAO factOrEasyGameSettingsDAO) {
    this.factOrEasyGameSettingsDAO = factOrEasyGameSettingsDAO;
}
```

Dalším způsobem je umístění nad konstruktor třídy, ve které různé třídy injektujeme, kde do parametrů konstruktoru je umístěn parametr typu třídy, kterou je potřeba injektovat a v těle metody je nastaven na privátní atribut typu třídy, kterou je nutné injektovat. Nad konstruktor je potom umístěna anotace @Autowired .

Obrázek 6 - Ukázka injektnutí konstruktoru

```
private FactOrEasyGameSettingsDAO factOrEasyGameSettingsDAO;

@Autowired
public FactOrEasyGameSettingsServiceImpl(FactOrEasyGameSettingsDAO factOrEasyGameSettingsDAO) {
    this.factOrEasyGameSettingsDAO = factOrEasyGameSettingsDAO
}
```

Tento způsob se používá když objektů, které chce developer injektovat je více.

Poslední volbou je umístění anotace přímo nad vytvořený privátní atribut. Tento způsob se používá pokud je injektovaných tříd málo a nemá kvůli tomu cenu vytvářet konstruktor. Vzhledem k tomu, že v projektu je potřeba jen jedna injektovaná třída, tento způsob byl použit i v této aplikaci.

Obrázek 7 - Ukázka injektnutí atributu

```
@Autowired
private FactOrEasyGameSettingsDAO factOrEasyGameSettingsDAO;
```

Důležitou vlastností anotace @Autowired je, že se dá použít pouze nad třídami, které jsou anotovány anotací @Component nebo jejími potomky. Proto všechny třídy, se kterými chce developer pracovat, musí být anotovány jednou z potomků nebo samotnou anotací @Component.

Další na řadě je implementace jednotlivých metod. V případě této třídy je implementace jednoduchá. Vzhledem k tomu, že třída implementuje rozhraní, které jí udává třídy, se kterými bude pracovat všechny metody budou označeny anotací @Override, což znamená, že je to vlastní implementace třídy a že nebude používat implementaci metody z předka či interface. Zde je tedy ukázka základního principu objektově orientovaného programování, polymorfismu. Každá třída, která bude implementovat rozhraní bude umět stejné metody, ale každá si je implementuje po svém a tím je zajištěno, že když například na každém objektu v kolekci objektů implementující stejné rozhraní jehož metody má každý objekt naimplementované po svém zavoláme stejnou metodu, každá každý udělá něco jiného.

Zpět v service třídě je implementace provedena tak, že u metod, které pouze čtou (read, readAll) je zavolána příslušná metoda na repository a její výsledek je vrácen.

Obrázek 8 - Metody read a readAll Service třídy

```
@Override
public FactoriesGameSettings read(Long id) { return factOrEasyGameSettingsDAO.findOne(id); }

@Override
public List<FactoriesGameSettings> readAll() { return factOrEasyGameSettingsDAO.findAll(); }
```

U metody vytvoření a úpravy(create, modify) jsou předány metodám na repository objekty, které přišly na service a opět je vrácen výsledek.

Obrázek 9 - Metody create a modify Service třídy

```
@Override
public FactoriesGameSettings create(FactoriesGameSettings factoriesGameSettings) {
    return factOrEasyGameSettingsDAO.save(factoriesGameSettings);
}

@Override
public FactoriesGameSettings modify(FactoriesGameSettings factoriesGameSettings) {
    return factOrEasyGameSettingsDAO.save(factoriesGameSettings);
}
```

V poslední metodě smazání(delete), je zavolána metoda delete na repository a předáno číslo id, které bylo obdrženo, což je id objektu, které uživatel chce smazat.

Obrázek 10 - Metoda delete Service třídy

```
@Override
public void delete(Long id) { factOrEasyGameSettingsDAO.delete(id); }
```

Obrázek 11 - Třída FactOrEasyGameSettingsServiceImpl

```
@Service
public class FactOrEasyGameSettingsServiceImpl implements FactOrEasyGameSettingsService {

    @Autowired
    private FactOrEasyGameSettingsDAO factOrEasyGameSettingsDAO;

    @Override
    public FactoriesGameSettings read(Long id) {
        return factOrEasyGameSettingsDAO.findOne(id);
    }

    @Override
    public FactoriesGameSettings create(FactoriesGameSettings factoriesGameSettings) {
        return factOrEasyGameSettingsDAO.save(factoriesGameSettings);
    }

    @Override
    public FactoriesGameSettings modify(FactoriesGameSettings factoriesGameSettings) {
        return factOrEasyGameSettingsDAO.save(factoriesGameSettings);
    }

    @Override
    public void delete(Long id) { factOrEasyGameSettingsDAO.delete(id); }

    @Override
    public List<FactoriesGameSettings> readAll() { return factOrEasyGameSettingsDAO.findAll(); }


}
```

4.2.4 Facade vrstva

Aby prezentační vrstva nekomunikovala přímo se service vrstvou, byla vytvořena facade vrstva. Vedle balíčků DAO a Service přibyl tedy balíček Facade, do kterého budou umístěny všechna rohraní a implementační třídy pro tuto vrstvu.

Facade je návrhový vzor, který se používá k vytvoření jednotného rozhraní pro celou logickou skupinu tříd, které se tak sdruží do subsystému. Skládá se z jedné třídy, která fasádu tvoří. Ta je napojena na další třídy, se kterými pracuje. Zvenku je však vidět jen fasáda (od toho název), a ta zastupuje rozhraní pro celý subsystém. Celá složitá struktura tříd je v pozadí. Účel Facade vrstvy je komunikace mezi prezentační a aplikační vrstvou, tzn. že fasáda bude zpracovávat požadavky od prezentační vrstvy a předávat je aplikační vrstvě(service). Prezentační vrstva díky fasádě by tedy neměla vědět o aplikační vrstvě, neměla by s ní přímo pracovat a hlavně neměla by přímo pracovat s výsledky, které aplikační vrstva vrátí. Na fasádě tedy budou výsledky aplikační vrstvy převáděny do

vlastního modelu, který bude sloužit pro přenos dat. Tento model ještě před implementací samotné facade vrstvy musí být vytvořen.

Vedle ostatních balíčků facade, service a DAO vrstvy je vytvořen balíček model, který se obvykle tvoří v nových aplikacích jako první, ale protože v této práci byl zatím použit pouze model, který byl naimportován z jiného projektu, balíček je vytvářen až v tomto bodě. V něm bude třída FactOrEasyGameSettingsModel, která bude obsahovat všechny atributy třídy FactoriesGameSettings. Atributy nemusí být anotovány ani žádná další nastavení nemusí být provedena, jen vložení datového typu a názvu atributu. Pro vytváření instance této třídy musíbýt také konstruktor, což je metoda určená pro vytváření instancí tříd.

Aby k atributům byl přístup musejí být vytvořeny gettery a settery, což jsou přístupové metody k atributům tříd.

Obrázek 12 - Ukázka přístupových metod modelu

```
public Long getId() { return id; }

public void setId(Long id) { this.id = id; }

public int getAccount() { return account; }

public void setAccount(int account) { this.account = account; }

public int getNumberOfPlayers() { return numberOfPlayers; }

public void setNumberOfPlayers(int numberOfPlayers) { this.numberOfPlayers = numberOfPlayers; }

public int getMaterialStoreCosts() { return materialStoreCosts; }

public void setMaterialStoreCosts(int materialStoreCosts) { this.materialStoreCosts = materialStoreCosts; }

public int getProductsStoreCosts() { return productsStoreCosts; }

public void setProductsStoreCosts(int productsStoreCosts) { this.productsStoreCosts = productsStoreCosts; }

public int getStandardFactoryFixedCost() { return standardFactoryFixedCost; }

public void setStandardFactoryFixedCost(int standardFactoryFixedCost) {
    this.standardFactoryFixedCost = standardFactoryFixedCost;
}

public int getAutomatizedFactoryConstructionCosts() { return automatizedFactoryConstructionCosts; }

public void setAutomatizedFactoryConstructionCosts(int automatizedFactoryConstructionCosts) {
    this.automatizedFactoryConstructionCosts = automatizedFactoryConstructionCosts;
}
```

Tím je vše potřebné pro implementaci fasády hotovo.

Obrázek 13 - Model FactOrEasyGameSettingsModel (bez přístupových metod)

```
public class FactOrEasyGameSettingsModel {

    private Long id;
    private int account;
    private int numberOfPlayers;
    private int materialStoreCosts;
    private int productsStoreCosts;
    private int standardFactoryFixedCost;
    private int automatizedFactoryConstructionCosts;
    private int materialStore;
    private int productInStore;
    private int standardDevelopmentCost;
    private int automatizedDevelopmentCosts;
    private int standardFactoryNumber;
    private int automatizedFactoryNumber;
    private int initialNumberOfGamesPerPlayer;
    private int lapsPerSimulation;
    private Float loanLimitRatio;
    private int botBuyFactoryAggressionLevel;

    public FactOrEasyGameSettingsModel(Long id, int account, int numberOfPlayers, int materialStoreCosts,
                                       int productsStoreCosts, int standardFactoryFixedCost,
                                       int automatizedFactoryConstructionCosts, int materialStore,
                                       int productInStore, int standardDevelopmentCost, int automatizedDevelopmentCosts,
                                       int standardFactoryNumber, int automatizedFactoryNumber,
                                       int initialNumberOfGamesPerPlayer, int lapsPerSimulation, Float loanLimitRatio,
                                       int botBuyFactoryAggressionLevel) {
        this.id = id;
        this.account = account;
        this.numberOfPlayers = numberOfPlayers;
        this.materialStoreCosts = materialStoreCosts;
        this.productsStoreCosts = productsStoreCosts;
        this.standardFactoryFixedCost = standardFactoryFixedCost;
        this.automatizedFactoryConstructionCosts = automatizedFactoryConstructionCosts;
        this.materialStore = materialStore;
        this.productInStore = productInStore;
        this.standardDevelopmentCost = standardDevelopmentCost;
        this.automatizedDevelopmentCosts = automatizedDevelopmentCosts;
        this.standardFactoryNumber = standardFactoryNumber;
        this.automatizedFactoryNumber = automatizedFactoryNumber;
        this.initialNumberOfGamesPerPlayer = initialNumberOfGamesPerPlayer;
        this.lapsPerSimulation = lapsPerSimulation;
        this.loanLimitRatio = loanLimitRatio;
        this.botBuyFactoryAggressionLevel = botBuyFactoryAggressionLevel;
    }

    public FactOrEasyGameSettingsModel() {
    }
```

Zpět v balíčku facade je opět jako první vytvořeno rozhraní. Rozhraní bude tedy dost podobné rozhraní pro service třídu s rozdílem toho, že nepracuje s modelem FactoriesGameSettings, ale s nově vytvořeným modelem pro přenos dat, do kterého budou ve facade vrstvě ukládáná data , která pošle aplikační vrstva.

Obrázek 14 - Rozhraní FactOrEasyGameSettingsFacade

```
public interface FactOrEasyGameSettingsFacade {

    public FactOrEasyGameSettingsModel read(Long id);

    public FactOrEasyGameSettingsModel create(FactOrEasyGameSettingsModel factOrEasyGameSettingsModel);

    public FactOrEasyGameSettingsModel modify(FactOrEasyGameSettingsModel factOrEasyGameSettingsModel, Long id);

    public void delete(Long id);

    public List<FactOrEasyGameSettingsModel> readAll();

}
```

Pro třídu implementující toto rozhraní je vytvořena složka impl a v ní třída FactOrEasyGameSettingsFacadeImpl. Aby mohl Spring se třídou pracovat opět musí být anotována, tentokrát rodičovskou anotací @Component, protože pro Facade třídy nemá Spring konktrétní anotace, tak jako má pro perzistentní, aplikační a prezentační vrstvu.

Třída musí implementovat nově vytvořené rozhraní, takže za název třídy je přidáno **implements** FactOrEasyGameSettingsFacade, které implementuje rozhraní FactOrEasyGameSettingsFacade.

Obrázek 15 - Hlavička třídy FactOrEasyGameSettingsFacadeImpl

```
@Component
public class FactOrEasyGameSettingsFacadeImpl implements FactOrEasyGameSettingsFacade {
```

Aby bylo vše jak má, metody definované v interface musejí být implementovány. Tato třída také potřebuje volat metody service třídy a proto je třeba injektovat service třídu do fasády. To je provedeno stejně jako injektnutí repository v service, tedy použitím anotace @Autowired**.**

Obrázek 16 - Injektnutá třída FactOrEasyGameSettingsService

```
@Autowired
FactOrEasyGameSettingsService factOrEasyGameSettingsService;
```

Jak už bylo zmíněno, fasádní vrstva bude převádět výsledky aplikační vrstvy, předávat prezentační a naopak. Musí být tedy vytvořena metoda, která tuto konverzi bude jednoduše umožňovat, aby pak mohla být použita v hlavních CRUD metodách a nemusela být konverze implementována v každé z nich. Jako první může být situace, kdy klient zavolá metodu GET s určitým id, chce tedy získat nastavení s určitým id. Prezentační vrstva přijme požadavek a předá ho fasádě. Očekává odpověď, tedy objekt s daným id reprezentován modelem, který byl vytvořen pro přenos dat, protože prezentační vrstva neví

jaké další vrstvy pod fasádou jsou a neví s jakými modely pracuje, zná akorát Facade a model FactOrEasyGameSettingsModel se kterým pracuje a přijímá ho v požadavku (requestu) od klientů(více v části o implementaci prezentační vrstvy). Prezentační vrstva předá pouze id objektu, který chce najít a vrátit. Fasáda zavolá metodu read servisní vrstvy. Ta ale vrátí objekt reprezentován základním modelem FactoriesGameSettings. Data z tohoto objektu musí být převedena na model pro přenos

(FactOrEasyGameSettingsModel), aby mohla být předána dále prezentační vrstvě.

Tím pádem je potřeba metoda, která bude převádět model FactoriesGameSettings na model

FactOrEasyGameSettingsModel. Taková metoda bude nazvána convertModelToEntity (převede model pro přenos na entitu uklládanou v databázi), která jako parametr bude příjímat objekt typu FactoriesGameSettings a vracet bude objekt typu

FactOrEasyGameSettingsModel.

Obrázek 17 - Hlavička metody convertEntityToModel

```
private FactOrEasyGameSettingsModel convertEntityToModel(FactoriesGameSettings factoriesGameSettings){
```

Metoda je implementována tak, že na začátku je vytvořen objekt typu FactOrEasyGameSettingsModel a pomocí konstruktoru je vytvořena jeho instance, kdy do konstruktoru jsou předány hodnoty atributů objektu jenž byl předán (service vrstvou např. její metodou read)

Obrázek 18 - Tělo metody convertEntityToModel

```
FactOrEasyGameSettingsModel factOrEasyGameSettingsModel = new FactOrEasyGameSettingsModel(
        factoriesGameSettings.getId(),
        factoriesGameSettings.getAccount(),
        factoriesGameSettings.getNumberOfPlayers(),
        factoriesGameSettings.getMaterialStoreCosts(),
        factoriesGameSettings.getProductsStoreCosts(),
        factoriesGameSettings.getStandardFactoryFixedCost(),
        factoriesGameSettings.getAutomatizedFactoryConstructionCosts(),
        factoriesGameSettings.getMaterialStore(),
        factoriesGameSettings.getProductInStore(),
        factoriesGameSettings.getStandardDevelopmentCost(),
        factoriesGameSettings.getAutomatizedDevelopmentCosts(),
        factoriesGameSettings.getStandardFactoryNumber(),
        factoriesGameSettings.getAutomatizedFactoryNumber(),
        factoriesGameSettings.getInitialNumberOfGamesPerPlayer(),
        factoriesGameSettings.getLapsPerSimulation(),
        factoriesGameSettings.getLoanLimitRatio(),
        factoriesGameSettings.getBotBuyFactoryAggressionLevel()
```

a tím je vytvořen objekt který v sobě nese data vrácená z databáze v modelu se kterým umí prezentační třída pracovat a zároveň nemusí vědět co se děje pod Facade vrstvou. Hodnoty atributů předáného objektu ze Service vrstvy lze získat pomocí již zmíněných getterů, které když jsou zavolány na předaném objektu, vrátí hodnoty, které objekt nese. Celý vytvořený objekt typu FactOrEasyGameSettingsModel metoda vrátí a tím je metoda pro konverzi entity do modelu pro přenos hotová.

Obrázek 19 - Metoda pro konverzi entity na model

```
private FactOrEasyGameSettingsModel convertEntityToModel(FactoriesGameSettings factoriesGameSettings){
    FactOrEasyGameSettingsModel factOrEasyGameSettingsModel = new FactOrEasyGameSettingsModel(
            factoriesGameSettings.getId(),
            factoriesGameSettings.getAccount(),
            factoriesGameSettings.getNumberOfPlayers(),
            factoriesGameSettings.getMaterialStoreCosts(),
            factoriesGameSettings.getProductsStoreCosts(),
            factoriesGameSettings.getStandardFactoryFixedCost(),
            factoriesGameSettings.getAutomatizedFactoryConstructionCosts(),
            factoriesGameSettings.getMaterialStore(),
            factoriesGameSettings.getProductInStore(),
            factoriesGameSettings.getStandardDevelopmentCost(),
            factoriesGameSettings.getAutomatizedDevelopmentCosts(),
            factoriesGameSettings.getStandardFactoryNumber(),
            factoriesGameSettings.getAutomatizedFactoryNumber(),
            factoriesGameSettings.getInitialNumberOfGamesPerPlayer(),
            factoriesGameSettings.getLapsPerSimulation(),
            factoriesGameSettings.getLoanLimitRatio(),
            factoriesGameSettings.getBotBuyFactoryAggressionLevel()
    );
    return factOrEasyGameSettingsModel;
}
```

Zároveň ale bude potřeba také metoda, která bude dělat opačnou konverzi, tedy přenos FactOrEasyGameSettingsModel na FactoriesGameSettings, protože například při vytvoření nebo úpravě herního nastavení bude prezentační vrstva posílat celý objekt reprezentován modelem FactOrEasyGameSettingsModel, aplikační vrstva ale přijímá pouze model FactoriesGameSettings a proto je potřeba vytvořit další metodu s názvem convertModelToEntity, která bude jako parametr přijímat objekt typu FactOrEasyGameSettingsModel a vracet převedený objekt typu FactoriesGameSettings.

Obrázek 20 - Hlavička metody convertModelToEntity

```
private FactoriesGameSettings convertModelToEntity(FactOrEasyGameSettingsModel factOrEasyGameSettingsModel){
```

Metoda je naprogramována tak, že jako první se vytvoří prázdný objekt typu FactoriesGameSettings a do něj se postupně, pomocí setterů, nastaví data z předaného objektu, které získá zavoláním getterů na tomto objektu a naplněný objekt vrátí.

Tím je metoda naprogramovaná.

Obrázek 21 - Metoda pro konverzi modelu na entitu

```
private FactoriesGameSettings convertModelToEntity(FactOrEasyGameSettingsModel factOrEasyGameSettingsModel){
    FactoriesGameSettings factoriesGameSettings = new FactoriesGameSettings();
    factoriesGameSettings.setId(factOrEasyGameSettingsModel.getId());
    factoriesGameSettings.setAccount(factOrEasyGameSettingsModel.getAccount());
    factoriesGameSettings.setNumberOfPlayers(factOrEasyGameSettingsModel.getNumberOfPlayers());
    factoriesGameSettings.setProductsStoreCosts(factOrEasyGameSettingsModel.getProductsStoreCosts());
    factoriesGameSettings.setStandardFactoryFixedCost(factOrEasyGameSettingsModel.getStandardFactoryFixedCost());
    factoriesGameSettings.setAutomatizedFactoryConstructionCosts(factOrEasyGameSettingsModel.getAutomatizedFactoryConstructionCosts());
    factoriesGameSettings.setMaterialStore(factOrEasyGameSettingsModel.getMaterialStore());
    factoriesGameSettings.setProductInStore(factOrEasyGameSettingsModel.getProductInStore());
    factoriesGameSettings.setStandardDevelopmentCost(factOrEasyGameSettingsModel.getStandardDevelopmentCost());
    factoriesGameSettings.setAutomatizedDevelopmentCosts(factOrEasyGameSettingsModel.getAutomatizedDevelopmentCosts());
    factoriesGameSettings.setStandardFactoryNumber(factOrEasyGameSettingsModel.getStandardFactoryNumber());
    factoriesGameSettings.setAutomatizedFactoryNumber(factOrEasyGameSettingsModel.getAutomatizedFactoryNumber());
    factoriesGameSettings.setInitialNumberOfGamesPerPlayer(factOrEasyGameSettingsModel.getInitialNumberOfGamesPerPlayer());
    factoriesGameSettings.setLapsPerSimulation(factOrEasyGameSettingsModel.getLapsPerSimulation());
    factoriesGameSettings.setLoanLimitRatio(factOrEasyGameSettingsModel.getLoanLimitRatio());
    factoriesGameSettings.setBotBuyFactoryAggressionLevel(factOrEasyGameSettingsModel.getBotBuyFactoryAggressionLevel());
    return factoriesGameSettings;
}
```

Tyto dvě metody budou pomáhat při převodu jednotlivých objektů, aby s nimi mohli pracovat obě strany.

Kvůli metodě readAll(), která vrací list objektů typu FactoriesGameSettings byla vytvořena ještě metoda, která převede celý list. Metoda je nazvána convertEntityListToModelList, v parametru přijímá list objektů typu FactoriesGameSettings a vrací list objektů typu FactOrEasyGameSettingsModel.

Obrázek 22 - Hlavička metody convertEntityListToModelList

```
private List<FactOrEasyGameSettingsModel> convertEntityListToModelList(List<FactoriesGameSettings> factoriesGameSettingsList) {
```

V této metodě už není třeba programovat převod z jednotlivých typů, protože můžou být využity metody convertModelToEntity a convertEntityToModel. Na začátku metody je vytvořen prázdný spojový seznam (ArrayList) objektů FactOrEasyGameSettingsModel. Do tohoto seznamu budou ukládány převedené objekty z příchozího listu objektů FactoriesGameSettings. List z parametru metody musí být procházen for each cyklem, každý záznam v listu musí být převeden pomocí metody convertEntityToModel a pomocí metody add na nově vytvořeném listu na začátku metody je do listu přidán.

Každý záznam bude převeden a přidán do listu, který naplněný bude vrácen.

Obrázek 23 - Tělo metody convertEntityListToModelList

```
List<FactOrEasyGameSettingsModel> factOrEasyGameSettingsModelList = new ArrayList<~>();
for(FactoriesGameSettings factoriesGameSettings : factoriesGameSettingsList) {
    factOrEasyGameSettingsModelList.add(convertEntityToModel(factoriesGameSettings));
}
return factOrEasyGameSettingsModelList;
```

Tato metoda doplní metody convertModelToEntity a convertEntityToModel a tím jsou všechny potřebné metody pro převedení objektů hotové.

Obrázek 24 - Metoda pro konverzi listu modelů na list entit

```
private List<FactOrEasyGameSettingsModel> convertEntityListToModelList(List<FactoriesGameSettings> factoriesGameSettingsList) {
    List<FactOrEasyGameSettingsModel> factOrEasyGameSettingsModelList = new ArrayList<FactOrEasyGameSettingsModel>();
    for(FactoriesGameSettings factoriesGameSettings : factoriesGameSettingsList) {
        factOrEasyGameSettingsModelList.add(convertEntityToModel(factoriesGameSettings));
    }
    return factOrEasyGameSettingsModelList;
}
```

Hlavní CRUD metody jsou naprogramované pomocí metod pro převod typů objektů a volání service třídy.

V metodě read, která bere jako parametr id objektu, který chce uživatel získat je vytvořen objekt typu FactoriesGameSettings a do něj doplněn výsledek metody read na injektované service třídě.

Metoda ale vrací objekt typu FactOrEasyGameSettingsModel, takže je v metodě vytvořen další objekt tentokrát typu FactOrEasyGameSettingsModel, do kterého je vložen výsledek pomocné metody convertEntityToModel, které je jako parametr předán první objekt tedy výsledek volání metody read na service a tento objekt metoda vrátí.

Obrázek 25 - Metoda read třídy Facade

```
@Override
public FactOrEasyGameSettingsModel read(Long id) {
    FactoriesGameSettings factoriesGameSettings = factOrEasyGameSettingsService.read(id);
    FactOrEasyGameSettingsModel model = convertEntityToModel(factoriesGameSettings);
    return model;
}
```

Metoda readAll, která má za úkol vrátit všechna herní nastavení je jednoduchá. Na začátku vytváří List objektů typu FactOrEasyGameSettingsModel, který je inicializován

metodou convertEntityListToModelList, které je předán výsledek metody readAll na servisní třídě a celý list potom metoda vrací.

Obrázek 26 - Metoda readAll třídy Facade

```
@Override
public List<FactOrEasyGameSettingsModel> readAll() {
    List<FactOrEasyGameSettingsModel> factOrEasyGameSettingsModelList =
            convertEntityListToModelList(factOrEasyGameSettingsService.readAll());
    return factOrEasyGameSettingsModelList;
}
```

Pro metodu delete není potřeba nic převádět, protože z prezentační vrstvy přijde pouze id objetku ke smazání, to fasáda předá service vrstvě, která předá id ke smazání repository a repository se postará o smazání.
Na fasádní vrstvě je tak metoda delete programovaná pouze tak, že volá metodu delete na servisní třídě a předává jí paramter id.

Obrázek 27 - Metoda delete třídy Facade

```
@Override
public void delete(Long id) { factOrEasyGameSettingsService.delete(id); }
```

V metodě create, kterou prezentační vrstva volá za účelem vytvoření nového záznamu je vytvořen objekt typu FactoriesGameSettings, inicializován metodou convertModelToEntity s předaným parametrem objektu z prezentační vrstvy, představující nové herní nastavení.

Obrázek 28 - Ukázka kódu metody create třídy Facade

```
FactoriesGameSettings factoriesGameSettings = convertModelToEntity(gameSettings);
```

Potom volá metodu create na injektovaném objektu factOrEasyGameSettingsService, předává ji výše vytvořený objekt, což je nové nastavení převedené na datový typ FactoriesGameSettings a výsledek této metody, která po úspěšném uložení vrací nově vytvořený objekt, je předán zpět do objektu vytvořeném na začátku metody.

Obrázek 29 - Ukázka kódu metody create třídy Facade (volání service třídy)

```
factoriesGameSettings = factOrEasyGameSettingsService.create(factoriesGameSettings);
```

Aby mohla metoda výsledek vrátit i na prezentační vrstvu musí se opět převést zpět na typ FactOrEasyGameSettingsModel s čímž pomáhá metoda convertEntityToModel.

Obrázek 30 - Metoda create třídy Facade

```
@Override
public FactOrEasyGameSettingsModel create(FactOrEasyGameSettingsModel gameSettings) {
    FactoriesGameSettings factoriesGameSettings = convertModelToEntity(gameSettings);
    factoriesGameSettings = factOrEasyGameSettingsService.create(factoriesGameSettings);
    return convertEntityToModel(factoriesGameSettings);
}
```

Poslední metoda modify, která slouží k úpravě již vytvořeného nastavení je implementována tak, že jako první se zavolá metoda read ze servisní třídy, které je jako parametr dosazováno id, které bylo modify metodě na facade třídě předáno prezentační vrstvou v parametru a výsledek metody je inicializací nového objektu.
Potom je vytvořen objekt typu FactOrEasyGameSettingsModel nazvaný model, který je zatím bez inicializace.

Následuje podmínka kontrolující, zda není objekt který jsme získali na prvním řádku metody null, tedy pokud uživatel nechce upravovat záznam, který neexistuje. Pokud metoda read skutečně záznam s daným id našla a vrátila, tak je do prozatím neinicializovaného objektu model dosazen výsledek metody modify volané na objektu factOrEasyGameSettingsService, které je předán celý objekt předaný z prezentační vrstvy, reprezentující upravený záznam od klienta převeden pomocí metody convertModelToEntity na typ FactoriesGameSettings. Pokud ale metoda read z prvního řádku objekt s daným id nenašla a je tedy null, je volána místo metody modify metoda create, jíž je předán také upravený objekt z paramteru předaný prezentační vrstvou od klienta, samozřejmě převedený na typ FactoriesGameSettings a výsledek této create metody předán do objektu model. To zařídí aby uživateli, který chce upravit objekt, který neexistuje byl objekt nově vytvořen. Celá metoda ve výsledku vypadá takto

Obrázek 31 - Metoda modify třídy Facade

```
@Override
public FactOrEasyGameSettingsModel modify(FactOrEasyGameSettingsModel gameSettings, Long id) {
    FactoriesGameSettings factoriesGameSettings = factOrEasyGameSettingsService.read(id);
    FactOrEasyGameSettingsModel model;
    if(Objects.nonNull(factoriesGameSettings)){
        model = convertEntityToModel(factOrEasyGameSettingsService.modify(convertModelToEntity(gameSettings)));
    } else {
        factoriesGameSettings =  factOrEasyGameSettingsService.create(convertModelToEntity(gameSettings));
        model = convertEntityToModel(factoriesGameSettings);
    }
    return model;
}
```

Hotová třída FactOrEasyGameSettingsFacadeImpl je zobrazena na následujícím obrázku (bez pomocných metod pro převod, které jsou zobrazeny výše).

Obrázek 32 - Třída FactOrEasyGameSettingsFacadeImpl (bez pomocných metod)

```
@Component
public class FactOrEasyGameSettingsFacadeImpl implements FactOrEasyGameSettingsFacade {

    @Autowired
    FactOrEasyGameSettingsService factOrEasyGameSettingsService;

    @Override
    public FactOrEasyGameSettingsModel read(Long id) {
        FactoriesGameSettings factoriesGameSettings = factOrEasyGameSettingsService.read(id);
        FactOrEasyGameSettingsModel model = convertEntityToModel(factoriesGameSettings);
        return model;
    }

    @Override
    public FactOrEasyGameSettingsModel create(FactOrEasyGameSettingsModel gameSettings) {
        FactoriesGameSettings factoriesGameSettings = convertModelToEntity(gameSettings);
        factoriesGameSettings = factOrEasyGameSettingsService.create(factoriesGameSettings);
        return convertEntityToModel(factoriesGameSettings);
    }

    @Override
    public FactOrEasyGameSettingsModel modify(FactOrEasyGameSettingsModel gameSettings, Long id) {
        FactoriesGameSettings factoriesGameSettings = factOrEasyGameSettingsService.read(id);
        FactOrEasyGameSettingsModel model;
        if(Objects.nonNull(factoriesGameSettings)){
            model = convertEntityToModel(factOrEasyGameSettingsService.modify(convertModelToEntity(gameSettings)));
        } else {
            factoriesGameSettings =  factOrEasyGameSettingsService.create(convertModelToEntity(gameSettings));
            model = convertEntityToModel(factoriesGameSettings);
        }
        return model;
    }

    @Override
    public void delete(Long id) {
        factOrEasyGameSettingsService.delete(id);
    }

    @Override
    public List<FactOrEasyGameSettingsModel> readAll() {
        List<FactOrEasyGameSettingsModel> factOrEasyGameSettingsModelList =
                convertEntityListToModelList(factOrEasyGameSettingsService.readAll());
        return factOrEasyGameSettingsModelList;
    }
```

4.2.5 Prezentační vrstva

Po vrstvě Facade zbývá už jen prezentační vrstva, v Javě repzezentována tzv. controllerem. Controller má za úkol již přímo přijímat požadavky od uživatele, zpracovávat je a zasílat dál do aplikace. Controller vyjadřuje komunikační bod mezi uživatelem a aplikací a vystavuje na server tzv. endpointy, což jsou koncové body, přes které uživatel komunikuje s aplikací. Endpointy jsou reprezentovány URI adresami, které byly navrženy při návrhu aplikace. Jako poslední je tedy vytvořen balíček s názvem controller, který bude reprezentovat prezentační vrstvu a budou do něj umístěny všechny controllery.

Do balíčku se vytvoří třída FactOrEasyGameSettingsRESTController. To bude controller, který bude vystavovat endpointy pro práci s herním nastavením. Třída je anotována anotací @RestController což je potomek anotace @Controller , která jí označí jako přístupový bod k REST aplikaci a umožní tak vystavit již zmiňované endpointy. K této anotaci je přidána ještě anotace @RequestMapping, pomocí které lze určit jádro URI. V parametru této anotace je nastaveno */rest,* což bude root URI aplikace, takže každá URI bude tímto řetězcem začínat aby uživatel věděl, že volá REST aplikaci. Třída od žádné třídy nedědí, ani neimplementuje žádné rozhraní

Obrázek 33 - Hlavička controlleru

```
@RestController
@RequestMapping("/rest")
public class FactOrEasyGameSettingsRESTController {
```

Controller potřebuje volat facade vrstvu, konktréně třídu FactOrEasyGameSettingsFacadeImpl, čehož lze docílit znovu pomocí dependency injection. Třída je pomocí anotace @Autowired nainjektována.

Po použití depedency injection je nutné implementovat všechny navržené metody. Jako první bude naimplementována metoda pro získání konkrétního záznamu. Uživatel bude volat pomocí HTTP metody GET, URI **rest/gamesettings/{id}**. Aby byla metoda na této adrese zpřístupněna musí mít anotaci @RequestMapping a jako její parametry, value **"/gamesettings/{id}"** a method GET, což určí, že metoda je zpřístupněna pouze na HTTP metodě GET. V parametru metody musí být uvedena anotace @PathVariable, která uvnitř metody zajistí možnost pracovat s paramterem, který přijde v adrese. V této metodě přijde

parametr id a jeho datový typ bude long. Jako návratovou hodnotu metody na controlleru je použit spring objekt ResponseEntity, což je třída určená k těmto účelům. Této třídě jde genericky určit typ který bude nést a její výhodou je, že v něm lze jednoduše číst např. http hlavička, kód a další potřebné údaje, které například při logování nebo debuggingu mohou být velice prospěšné. Zároveň ji lze snadno využít pokud v návratové hodnotě nic nemá být a má se uživateli vrátit pouze zpráva o tom jak byl požadavek (request) proveden. Jako návratová hodnota této metody bude ResponseEntity<FactOrEasyGameSettingsModel>.

Uvnitř metody je potom volána pouze metoda read z fasády a je ukládána do objektu s názvem response(odpověď). Vrácena je response zabalená v ResponseEntity, pomocí volání statické metody ok a na ní metody body, která jako parametr bere tělo response zprávy, což bude objekt získaný z fasády. První metoda je hotova a jedna část aplikace je funkční.

Obrázek 34 - Metoda readGameSettings

```
@RequestMapping(value = "/gamesettings/{id}", method = RequestMethod.GET)
public ResponseEntity<FactOrEasyGameSettingsModel> readGameSettings(@PathVariable("id") long id) {
    FactOrEasyGameSettingsModel response = factOrEasyGameSettingsFacade.read(id);
    return ResponseEntity.ok().body(response);
}
```

Obdobně je naprogramována i metoda pro získání všech záznamů. Rozdíl je v přístupové adrese, kde nepotřebujeme žádný parametr, takže anotace pro tuto metodu vypadá následovně

Obrázek 35 - Hlavička metody readAllGameSettings

```
@RequestMapping(value = "/gamesettings",method = RequestMethod.GET)
```

Opět je použita HTTP metoda GET. Metoda je bez parametrů a jako návrátová hodnota bude list objektů typu FactOrEasyGameSettingsModel obalený v ResponseEntity. Tělo metody tvoří volání metody readAll na fasádní třídě a uložení jejího výsledku do List<FactOrEasyGameSettingsModel> s názvem response a navrácení listu pomocí ResponseEntity.

Obrázek 36 - Metoda readAllGameSettings

```
@RequestMapping(value = "/gamesettings",method = RequestMethod.GET)
public ResponseEntity<List<FactOrEasyGameSettingsModel>> readAllGameSettings() {
    List<FactOrEasyGameSettingsModel> response = factOrEasyGameSettingsFacade.readAll();
    return ResponseEntity.ok(response);
}
```

Jako další je metoda DELETE, díky které bude moci uživatel smazat určitý záznam. Hodnota anotace @RequestMapping bude použita stejnoá jako v metodě pro získání určitého záznamu, metoda však musí být nastavena na DELETE.

Obrázek 37 - Hlavička metody deleteGameSettings

```
@RequestMapping(value = "/gamesettings/{id}", method = RequestMethod.DELETE)
```

Také parametr bude použit stejný jako v metodě pro čtení konkrétního záznamu, návratová hodnota však bude jiná, protože metoda delete nic nevrací a tak v generické části ResponseEntity bude klíčové slovo Void, které označí beznávratovou hodnotu.

V těle metody je už jen zavolána metoda z Facade vrstvy, které je předáno id objektu ke smazání. Pomocí ReponseEntity a její statické metody ok a na ní zavolané metodě build je vrácena zpráva o úspěšném provedení požadavku.

Obrázek 38 - Metoda readAllGameSettings

```
@RequestMapping(value = "/gamesettings/{id}", method = RequestMethod.DELETE)
public ResponseEntity<Void> deleteGameSettings(@PathVariable("id") long id){
    factOrEasyGameSettingsFacade.delete(id);
    return ResponseEntity.ok().build();
}
```

Ke splnění požadavků aplikaci zbývají pouze zpřístupnit metody pro vytvoření a úpravu herního nastavení.

Pro vytvoření nového nastavení vznikla metoda createGameSettings, jejíž parametrem je objekt typu FactOrEasyGameSettingsModel obalený v entitě RequestEntity, což je podobná třída jako ResponseEntity, ale je využívána pro požadavky (requesty). V těle této entity bude aplikaci předáváno nové nastavení. Pokud vytvoření a uložení do databáze proběhne úspěšně bude metoda vracet nově vytvořený objekt, což řekne uživateli,

že objekt byl vytvořen a může jej dále využít. Výsledek bude jako v ostatních metodách obalen v ResponseEntity.

Anotace metody bude opět RequestMapping, kde value bude /gamesettings bez parametru jelikož uživatel bude posílat celý objekt a v parametru anotace method bude nastavená metoda POST.

Implementace metody vypadá tak, že se vytvoří objekt typu FactOrEasyGameSettingsModel s názvem response a ten se naplní pomocí metody create na factOrEasyGameSettingsFacade, které je předáváno tělo RequestEntity, pomocí metody getBody.

Response je následně stejně jako v read metodě obalena do ResponseEntity a je vrácena uživateli.

Touto metodou umožňuje aplikace vytvářet nový záznam a zbývá už jen umožnit uživateli úpravu záznamu.

Obrázek 39 - Metoda createGameSettings

```
@RequestMapping(value = "/gamesettings", method = RequestMethod.POST)
public ResponseEntity<FactOrEasyGameSettingsModel> createGameSettings(RequestEntity<FactOrEasyGameSettingsModel> requestEntity) {
    FactOrEasyGameSettingsModel response = factOrEasyGameSettingsFacade.create(requestEntity.getBody());
    return ResponseEntity.ok().body(response);
}
```

Pro úpravu záznamu slouží metoda modifyGameSettings se dvěma parametry. Jeden z parametrů bude stejně jako v create metodě RequestEntity obsahující upravený objekt nastavení a jako druhý parametr je id objektu, který bude upraven. Metoda po úspěšné úpravě daného nastavení vrátí objekt nastavení obalený v ResponseEntity.

V anotaci nad metodou je nastavena hodnota, kokrétně URI na **/gamesettings/{id}**, a method na PUT.

Tělo metody potom vypadá stejně jako v metodě create, s tím rozdílem, že místo metody create je na fasádě volána metoda modify, které je předáno tělo ReqeustEntity, reprezentující upravené nastavení a jeho id.

Stejně jako u create metody je vrácena response jako tělo ResponseEntity a metoda pro úpravu vypadá následovně

Obrázek 40 - Metoda modifyGameSettings

```
@RequestMapping(value = "/gamesettings/{id}", method = RequestMethod.PUT)
public ResponseEntity<FactOrEasyGameSettingsModel> modifyGameSettings(RequestEntity<FactOrEasyGameSettingsModel>
    requestEntity, @PathVariable("id") long id){
    FactOrEasyGameSettingsModel response = factOrEasyGameSettingsFacade.modify(requestEntity.getBody(),id);
    return ResponseEntity.ok().body(response);
}
```

Všechny metody by měly být hotovy a tím by jak controller tak celá práce programování měla být dokončena a aplikace by měla být připravena k použití.

Obrázek 41 - Třída FactOrEasyGameSettingsRESTController

```
@RestController
@RequestMapping("/rest")
public class FactOrEasyGameSettingsRESTController {

    @Autowired
    FactOrEasyGameSettingsFacade factOrEasyGameSettingsFacade;

    @RequestMapping(value = "/gamesettings",method = RequestMethod.GET)
    public ResponseEntity<List<FactOrEasyGameSettingsModel>> readAllGameSettings() {
        List<FactOrEasyGameSettingsModel> response = factOrEasyGameSettingsFacade.readAll();
        return ResponseEntity.ok(response);
    }

    @RequestMapping(value = "/gamesettings/{id}", method = RequestMethod.GET)
    public ResponseEntity<FactOrEasyGameSettingsModel> readGameSettings(@PathVariable("id") long id) {
        FactOrEasyGameSettingsModel response = factOrEasyGameSettingsFacade.read(id);
        return ResponseEntity.ok().body(response);
    }


    @RequestMapping(value = "/gamesettings", method = RequestMethod.POST)
    public ResponseEntity<FactOrEasyGameSettingsModel>
    createGameSettings(RequestEntity<FactOrEasyGameSettingsModel> requestEntity) {
        FactOrEasyGameSettingsModel response = factOrEasyGameSettingsFacade.create(requestEntity.getBody());
        return ResponseEntity.ok().body(response);
    }


    @RequestMapping(value = "/gamesettings/{id}", method = RequestMethod.PUT)
    public ResponseEntity<FactOrEasyGameSettingsModel> modifyGameSettings(RequestEntity<FactOrEasyGameSettingsModel>
        requestEntity, @PathVariable("id") long id){
        FactOrEasyGameSettingsModel response = factOrEasyGameSettingsFacade.modify(requestEntity.getBody(),id);
        return ResponseEntity.ok().body(response);
    }

    @RequestMapping(value = "/gamesettings/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<Void> deleteGameSettings(@PathVariable("id") long id){
        factOrEasyGameSettingsFacade.delete(id);
        return ResponseEntity.ok().build();
    }
```

4.2.6 **Konfigurace**

Kvůli požadavku na použití jndi, musela být nakonec ještě přidána jednoduchá konfigurace. Java Naming and Directory Interface (JNDI) je Java API které slouží pro vyhledávání dat a zdrojů podle jména. Zajišťuje nezávislost na databázi a bylo používané především v aplikacích psané pomocí EJB, které museli být nasazovány na server. Výhodou použití tohoto api je, že po nasazení aplikace na server se nastaví data resource, tedy odkaz na zdroj dat dle jména nastaveného v aplikaci a aplikace pak pracuje s jakýmkoliv zdrojem nastaveného na toto jméno. Vývojáři tedy docílí toho, že na různých servrech může aplikace pracovat s různými databázemi, nezávisle na typu databáze a typu serveru. Tato technologie už se nyní ve spring boot aplikacích nepoužívá, neboť tyto aplikace už fungují na jiném principu a většinou jsou jak aplikace tak databáze a ostatní technologie, které potřebuje zabaleny do balíčku a nasazovány jako celek na servery. Protože ale aplikace FactOrEasy takto funguje, musela tato konfigurace být provedena i v modulu pro REST API. Musela být vytvořena konfigurační třída ApplicationConfig, anotována antoací @Configuration, což označí třídu za konfigurační, obsahující jednu metodu dbConnectionProps vracející objekt nesoucí údaje pro připojení k databázi.

Objekt DbConnectionProps byl vytvořen pouze za účelem nesení dat. Je to jednoduchá třída obsahující atributy a jejich přístupové metody. Atributy jsou uvedeny následnovně.

```
String url;
String username;
String password;
String driverClassName;
String jndiName;
```

Třída je anotována @ConfigurationProperties(prefix = "connection"), což označí třídu jako properties nesoucí data.

Nejdůležitější třídou z nově vytvořených je DataSourceConfig, kde bude hlavní konfigurace pro jndi nastavena. Třída bude mít anotace @Configuration pro označení za konfigurační třídu, @EnableTransactionManagement umožnující správu transakcí @EnableJpaRepositories s parameterem basePackages = "factoreasyrest.dao" aby mohla pracovat s modely potřebné pro práci s databází.

Obrázek 42 - Hlavička konfigurační třídy DataSourceConfig

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = "factoreasyrest.dao")
public class DataSourceConfig {
```

Potom je injektnut DbConnectionProps, se kterým bude potřeba pracovat.Kvůli tomu, že spring boot aplikace lokálně obsahuje a používá aplikační server tomcat musí být tento server nastaven, aby bylo možné aplikaci spouštět lokálně.

V těchto metodách jsou nastavovány properties aplikačního serveru aby mohl používat údaje z properties souboru pro připojení.

Obrázek 43 - Konfigurační metoda tomcatFactory

```
@Bean
public TomcatEmbeddedServletContainerFactory tomcatFactory() {
    return new TomcatEmbeddedServletContainerFactory() {
        @Override
        protected TomcatEmbeddedServletContainer getTomcatEmbeddedServletContainer(Tomcat tomcat) {
            tomcat.enableNaming();
            return super.getTomcatEmbeddedServletContainer(tomcat);
        }
        @Override
        protected void postProcessContext(Context context) {
            ContextResource resource = new ContextResource();
            resource.setName(props.getJndiName());
            resource.setType(DataSource.class.getName());
            resource.setProperty("factory", "org.apache.tomcat.jdbc.pool.DataSourceFactory");
            resource.setProperty("url", props.getUrl());
            resource.setProperty("password", props.getPassword());
            resource.setProperty("username", props.getUsername());
            context.getNamingResources().addResource(resource);
        }
    };
}
```

V další metodě jndiDataSource(), která vrací objekt DataSource, jsou tentokrát již pro samotné jndi nastaveny údaje ze kterých má brát a hlavně jméno pro vyhledávání zdroje dat, které má po nasazení na server hledat.

Obrázek 44 - Konfigurační metoda jndiDataSource

```
@Bean(destroyMethod = "")
public DataSource jndiDataSource() throws IllegalArgumentException, NamingException {
    JndiObjectFactoryBean bean = new JndiObjectFactoryBean();
//      For local testing with springboot
//     bean.setJndiName("java:comp/env/" + props.getJndiName());
    bean.setJndiName(props.getJndiName());
    bean.setProxyInterface(DataSource.class);
    bean.setLookupOnStartup(false);
    bean.afterPropertiesSet();
    return (DataSource) bean.getObject();
}
```

Potom už zbývá je nastavení EntityManagerFactory, která je důležitou součástí zajištění připojení k databázi.

Obrázek 45 - Konfigurační metoda entityManagerFactory

```
@Bean
public EntityManagerFactory entityManagerFactory() throws SQLException, IllegalArgumentException, NamingException {
    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    vendorAdapter.setDatabase(Database.DEFAULT);
    vendorAdapter.setShowSql(true);
    LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan("eu.cz.culs.kii.common.factoreasycommon");
    factory.setDataSource(jndiDataSource());
    factory.afterPropertiesSet();
    return factory.getObject();
}
```

V této metodě je nastavena databáze na defaultní, to znamená, že není vázána na určitý typ a zařídí se až podle toho ke které databázi bude připojena. Dále jsou nastaveny balíčky pro sken entit a jndi data source, kde je použita metoda vracející DataSource vytvořena dříve.

Data ze kterých si potom toto nastavení bere, a nese je v nově vytvořeném objektu jsou umístněny ve zdrojích projektu v souboru application.properties a jejich předpona musí být connection, jelikož tak bylo nastaveno v DbConnectionProps.

```
#DB connection parameters
connection.url=jdbc:postgresql://localhost:5432/foe
connection.username=foe
connection.password=123456
connection.jndiName=jdbc/foe
```

Jak je zde vidět, je zde nastavená url, username a password, které slouží pouze pro lokální nastavení a odkazuje na lokální databázi pro testování. Tyto údaje si potom aplikace získá z nastaveného zdroje na serveru, který je nalezen pomocí jndiName, které je v souboru uvedené.

Poslední věcí je nastavit balení aplikace do war, aby mohla být nasazena na server. Nastavení je provedeno v pom.xml, jak již bylo uvedeno při popisu pom.

Nyní lze spustit build aplikace, který zabalí vše do war souboru a ten lze nasadit na server. Pro otestování funkčnosti byl použit GlassFish 5.0 server a databáze Postgres a MySql.

4.2.7 Testování

Pro testování byla vytvořena databáze a díky sql skriptům v projektu FactOrEasy byla vytvořena celá struktura databáze pro FOE. Pomocí rozšíření prohlížeče Google chrome, Advanced REST client, byly postupně provolávány všechny funkční metody.

Jako první byla otestováno vytvoření nového záznamu.

Obrázek 46 - Hlavička požadavku pro testování metody vytvoření

Do těla POST metody byl vložen nový objekt v JSON reprezentaci.

Obrázek 47 - Tělo testovacího požadavku metody vytvoření

```
{
  "account": 5800,
  "numberOfPlayers": 4,
  "materialStoreCosts": 300,
  "productsStoreCosts": 500,
  "standardFactoryFixedCost": 1000,
  "automatizedFactoryConstructionCosts": 0,
  "materialStore": 4,
  "productInStore": 2,
  "standardDevelopmentCost": 2000,
  "automatizedDevelopmentCosts": 3000,
  "standardFactoryNumber": 2,
  "automatizedFactoryNumber": 0,
  "initialNumberOfGamesPerPlayer": 5,
  "lapsPerSimulation": 0,
  "loanLimitRatio": null,
  "botBuyFactoryAggressionLevel": 0
}
```

Tímto způsobem bylo vytvořeno několik dalších objektů, aby autor mohl pracovat s více záznamy.

Jako další metoda byla otestována metoda pro úpravu záznamu.

Obrázek 48 - Hlavička požadavku pro testování metody pro úpravu

Do těla metody byl vložen objekt s id 1, kterému byly některé hodnoty změněny.

Obrázek 49 - Tělo testovacího požadavku pro metodu úpravy


```
{
  "id": 1,
  "account": 10000,
  "numberOfPlayers": 5,
  "materialStoreCosts": 300,
  "productsStoreCosts": 500,
  "standardFactoryFixedCost": 1000,
  "automatizedFactoryConstructionCosts": 0,
  "materialStore": 4,
  "productInStore": 2,
  "standardDevelopmentCost": 5000,
  "automatizedDevelopmentCosts": 3000,
  "standardFactoryNumber": 2,
  "automatizedFactoryNumber": 0,
  "initialNumberOfGamesPerPlayer": 5,
  "lapsPerSimulation": 0,
  "loanLimitRatio": null,
  "botBuyFactoryAggressionLevel": 0
}
```


Potom byla zavolána metoda pro získání jednoho záznamu, konkrétně záznamu s id 1.

Obrázek 50 - Hlavička požadavku pro testování metody pro čtení

Jako poslední bylo otestováno mazání jednotlivých záznamů. Smazány byly záznamy s id 2 a 3.

Obrázek 51 - Hlavička požadavku pro testování metody mazání

5 Výsledky a diskuse

Tato kapitola obsahuje obrázky s výsledky jednotlivých testů

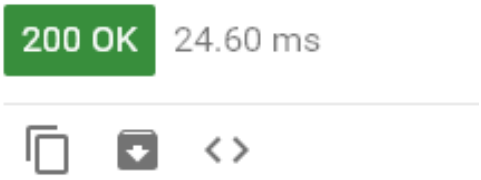
Obrázek 52 - Výsledek testování metody vytváření

Obrázek 53 - Výsledek testování metody úpravy

Obrázek 54 - Výsledek testování metody čtení

Metoda pro vrácení všech záznamů vrátila pole všech záznamů. Pro metodu mazání oba testy vrátily kód 200, který značí úspěšné smazání.

Obrázek 55 - Výsledek testování metody mazání



6 Závěr

Cílem této práce bylo navržení architektury REST rozhraní pro administrační část hry FactOrEasy, navržení tehcnologií pro naprogramování navrženého REST rozhraní, dále pak navržení architektury aplikace a samotné naprogramování aplikace dle návrhu a pomocí navržených technologií.

Rozhraní bylo navrženo dle moderních standardů, byly využity moderní technologie a knihovny spojené s programovacím jazykem Java, který byl zvolen pro naprogramování aplikace. Byla zvolena vícevrstvá architektura, aplikace byla navržena do čtyř vrstev a to Prezentační, Facade, Aplikační a Perzistentní. Aplikace byla úspěšně naprogramována dle návrhu, za pomocí moderních technologií v programovacím jazyce Java 8 a frameworku Spring a Hibernate. Aplikace byla vytvořena jako Spring Boot aplikace a nakonfigurována jako samostatný modul aplikace FactOrEasy. Aplikace využívá knihovnu Hibernate pro ORM mapování a Spring JPA pro operace s databází. Nástroj pro správu buildů aplikace je využíván Apache Maven. Aplikace byla taktéž nakonfigurována aby vyhověla požadavkům nezávislosti na prostředí a databáze a funguje tedy jako samostatný war modul, nasaditelný na kterýkoliv aplikační server.

Funkce aplikace byly po naprogramování otestovány za pomocí software Advanced REST client. Všechny testy proběhly úspěšně a aplikace funguje dle požadavků.

Aplikace by se dala dále rozšířit o další funkce, které by umožnily přístup přes REST rozhraní i k dalším částem aplikace. Vzhledem k tomu , že je apliakce

nakonfigurována a funguje bez problémů by tato rozšíření neměla být složitá´, stačilo by jen přidání tříd do jednotlivých vrstev pro potřebné části aplikace. Po tomto rozšíření by bylo dobré aplikaci zabezpečit, nyní není přístup ke koncovým bodům zabzpečen. Aplikace využívá knihovnu Spring, takže zabezpečení by se dalo jednoduše udělat za pomocí části Spring Security. Pokud by také byla aplikace rozšířena o více funkcí a kód by se rozšířil bylo by vhodné přejít z buildovacího nástroje Maven na Gradle, který je raychlejší a modernější.

Tato práce byla pro autora přínosná hlavně v oblasti znalosti REST webových služeb, protože autor předtím znal a programoval pouze SOAP webové služby, poznal tedy modernější způsob webových služeb, zjistil rozdíly a výhody RESTu. Jak po teoretické tak po praktické stránce se autor naučil standardy pro REST rozhraní a jejich návrh, které už během vypracovávání práce využil v praxi. Dále byla pro autora také zajímavá konfigurace nezávislosti Spring Boot aplikace na zdroji dat, která byla obtížná na naučení a nastavení, ale určitě bude přínosná pro budoucí práci v oblasti vývoje software v programovacím jazyce Java a frameworku Spring. Autor také zjistil více do hloubky jak fungují moderní Spring Boot aplikace nejen díky již zmíněné konfiguraci, ale také programování celého modulu už od začátku, díky čemuž měl autor možnost vytvořit úplně novu Spring Boot aplikaci úplně od počátečního stavu. Práce také autora obohatila o zkušenost testování, konkrétně vymýšlení a aplikování různých scénářů pro CRUD operace. Nastudování správných principů a návrh rozhraní bylo pro autora jednou z obtížnějších částí, vzhledem k tomu, že s tím neměl zkušenost Úplně nejobtížnější část, která však autorovi mnoho dala, byly různé konfigurace aplikace, naopak samotné naprogramování jednotlivých vrstev a hlavní logiky aplikace dle požadavků bylo díky moderním technologiím a hlavně knihovně Spring a Spring Bootu celkem jednoduché a rychlé.

7 Seznam použitých zdrojů

1. *Rest Resource Naming Guide.* [online].[cit. 2018-12-14].

   <https://restfulapi.net/resource-naming/>

2. *10 Best practices for better RESTful API.* [online].[cit. 2018-12-14].

   <https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/>

3. *Spring initializr.*[online].[cit. 2018-12-14].

   <https://start.spring.io/>

4. *Benefits and drawback of a Lazered Architecture.* [online].[cit. 2018-12-14].

   <https://www.pixelstech.net/article/1493900728-Benefits-and-Drawback-of-a-Layered-Architecture>

5. *Apache Maven.* [online].[cit. 2019-03-04].

   <https://cs.wikipedia.org/wiki/Apache_Maven>

6. *Intro to Inversion of Control and Dependency Injection with Spring*. [online].[cit. 2019-03-04].

   <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>

7. *The DAO pattern in Java.* [online].[cit. 2019-03-04].

   <https://www.baeldung.com/java-dao-pattern>

8. *Facade.* [online].[cit. 2017-03-14].

   <https://www.itnetwork.cz/navrh/navrhove-vzory/gof/gof-vzory-struktury/facade-navrhovy-vzor/>

9. *J2EE vzory.*[online].[cit. 2017-03-14].

   <https://dagblog.cz/j2ee-vzory-dao-odst%C3%ADn%C4%9Bn%C3%AD-zdroje-dat-d10d49463838>

8 Přílohy

Aplikace se zdrojovými kódy je dostupná v přiloženém zip souboru.