Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Frameworky pro vývoj Java webových aplikací

Diplomová práce

Autor: Jiří Brádle
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové duben 2016

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

Hradec Králové dne 18. 4. 2016 Jiří Brádle

Poděkování

Rád bych zde poděkoval doc. Ing. Filipu Malému, Ph.D. za výpomoc při dokončování závěrečné podoby práce a její samotné zastřešení. Dále Ing. Pavlu Krbálku, Ph.D. za pomoc při volbě tématu a všech podnětů, které vedli k výsledné podobě práce.

Anotace

Cílem této práce je představit čtenáři nové způsoby vývoje webových aplikací postavených na platformě Java. V rámci práce jsou představeny dva přístupy k vývoji. Prvním přístupem je čistě serverová aplikace. Jedná se o detailní popis vývoje a použité technologie. Druhý přístup ukazuje nový způsob vývoje klientské aplikace. V první části práce je čtenář seznámen se základními pojmy a principy vývoje webových aplikací. Druhá část se zabývá výběrem a popisem technologií použitých v rámci práce. Třetí část je pak ukázkou použití vybraných technologií, která obsahuje i praktickou část.

Annotation
Title: Development in Java web application frameworks

The aim of this diploma thesis to present readers a new way to develop web applications based on the Java platform. Two approaches of web development are presented in this thesis. The first approach is purely server side web application. This is a detailed description of the development and applied technologies. The second approach demonstrates a new way to develop client side web applications. In the first part the reader is familiarized with the basic concepts and principles of web application development. The second part deals with the selection and description of the technologies used in the thesis. The third part is an example of using selected technology, which also includes practical examples.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

1. Úvod

Webové aplikace jsou v dnešní době nepoužívanějším způsobem pro přenos a práci s informacemi. Téměř každá ze společností využívá webové aplikace například za účelem zisku nebo pro zjednodušení interních operací. Takové aplikace je potřeba udržovat aktuální a v případě chyb je i rychle opravit. Proto vzniká zájem o kvalitní programátory, kteří jsou schopni pohotově a jednoduše provést změnu nebo opravu. Aby byl programátor na trhu chtěný a neměl problém sehnat zakázku, nebo případně zaměstnání, měl by se zajímat a znát obecně používané přístupy.

Důvodem vzniku této práce byl popud jedné společnosti, která chce sjednotit pravidla a technologie používané právě při vývoji webových aplikací. Jelikož jsou ve zmíněné společnosti všechny projekty vytvářené na platformě Java, zaměřuje se práce na právě tuto platformu. Práce obsahuje vybraný přístup pro dva druhy webových aplikací. Prvním je aplikace, která je pouze serverová. Serverová aplikace bude postavena na technologiích, které jsou dobou ověřené. Zároveň však bude obohacena o novější přístupy. Druhou aplikací je klientská, která bude postavena jenom na nejnovějších technologiích.

1.1. Cíl práce

Hlavním cílem práce je seznámit čtenáře s nejmodernějším přístupem při tvorbě webových aplikací. Čtenář by měl získat informace o základních konceptech vybraných technologií a následně tyto informace uplatnit v praxi. Aby bylo na první pohled zřejmé, že je zvolený přístup opodstatněný, vzniknou v rámci práce i ukázky praktického použití.

1.2. Omezení a předpoklady práce

Čtenář by měl dobře znát platformu Java. Dále by měl být seznámen se základními principy a koncepty vývoje webových aplikací. Práce se snaží většinu základní konceptů zmínit, ale bude mnohem lépe čitelná pro čtenáře, kteří již mají nějaké zkušenosti s vývojem webových aplikací.

1.3. Framework

Při vývoji velkých aplikací, jako jsou například webové, často nastává situace, kdy se některé části řešení začnou opakovat, což může vést k několika problémům. Vzhledem k tomu, že na větších aplikacích pracuje i více vývojářů, opakující se řešení se mohou jádru lišit. Přesto je výsledek stejný. Tím se stávají projekty neudržitelné. Spousta vývojářských týmů na tento problém reaguje podobně. A to tak, že opakující se řešení vyčlení z výsledného projektu do separátních projektů, které jsou pak v budoucnu dále využívány. Tyto vyčleněné řešení

je již možné nazývat frameworkem. Existuje spousta firem, které si vytváří vlastní framework, což ale přináší nátlak na nové vývojáře, kteří se musí s individuálním přístupem seznámit. Obecně práce s interními frameworky není pro vývojáře příliš lukrativní. Jelikož v dnešní době existuje velká škála frameworků, které zabíhají do různých problematik, a jsou veřejně přístupné. A hlavně za nimi stojí velká komunita lidí, kteří přispívají ke zlepšování samotného frameworku.

Framework lze definovat jako ucelený soubor obecných řešení, které je možné použít pro zefektivnění vývoje komplexních aplikací [1]. Součástí většiny frameworků je mechanismus pro řízení programu, kdy prováděné akce nejsou způsobeny uživatelem, ale samotným frameworkem. Tento mechanismus je ve většině případů neměnný a programátor by do něj neměl příliš zasahovat. Framework ale musí nějakým způsobem umožnit vlastní zavedení do aplikace a spolupráci s ostatními komponentami. K tomuto účelu většina frameworků vystavuje rozhraní a abstraktní třídy, které může programátor pomocí vlastní implementace využít a přistoupit tak k jádru frameworku.

Při výběru vhodného frameworku je důležité zohlednit následující faktory:

- Velikost společnosti nebo komunity spravující daný framework
- Propracovanost veřejné dokumentace
- Podpora v případě problémů

Jak již bylo zmíněno, existuje mnoho frameworků, které se zaměřují na různá odvětví problematik. Jedním z takových odvětví je právě vývoj webových aplikací. Tyto frameworky se zaměřují hlavně na problémy vznikající při vývoji webových aplikací. Obsahují například prostředky pro práci s databází a nástroje pro generování webového obsahu. I když jsou většinou zaměřeny na vytváření dynamických webových stránek, je možné je využít i pro statické stránky.

2. VÝVOJ WEBOVÝCH APLIKACÍ

Z počátku byla Java využívána pouze k vytváření tzv. stand–alone aplikací, které byly vždy určeny pouze pro danou činnost a nepotřebovaly podporu jiných služeb. Postupem času se však rozrostla do dalších oblastí vývoje. Jednou z těchto oblastí jsou právě webové aplikace. Při obecném pohledu na problematiku webu lze rozdělit web na statické a dynamické stránky. Statické stránky obsahují neměnný obsah, kdežto dynamické stránky jsou schopny generovat obsah za pomoci dalších webových komponent. Slouží tedy pouze pro předávání informace uživateli. Webové aplikace, na rozdíl od webových stránek umožnují uživateli provádět akce a uložit výsledek. Vývoj webové aplikace je od základu rozdílný od vývoje stand–alone aplikace. Na tuto problematiku se zaměřuje Java EE (Enterprice Edition).

Klíčové součásti webové aplikace jsou:

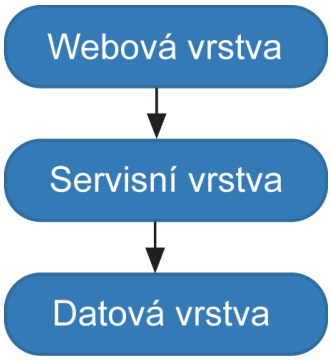
- *Platforma Java EE* – Souhrn API specifikací pro stavbu jednotlivých bloků webové aplikace.
- *Webový kontejner* – je implementací API specifikací platformy Java EE. Jeho hlavním úkolem je poskytovat služby a spouštět jednotlivé komponenty webu.
- *Webové komponenty* – Ty jsou spravované webovým kontejnerem. Mezi webové komponenty patří například servlety, JSP, filtry a listenery.

2.1. PLATFORMA JAVA EE

Hlavní cíle platformy Java EE jsou:

- Poskytnout souhrn API specifikací pro stavbu jednotlivých bloků webové aplikace.
- Standardizovat a snížit komplexnost aplikací vyvíjených v podnikových sférách. Toho je docíleno zavedením vícevrstvých aplikací.

Ve vícevrstvých aplikacích je funkcionalita rozdělena do různých částí programu, které se nazývají vrstvy.



Obrázek 1 – Vrstvy webové aplikace [autor]

Obrázek 1 zobrazuje zjednodušený pohled na vrstvy webové aplikace. Každá z těchto vrstev má za úkol pokrýt určitou oblast webové aplikace. Takovéto rozdělení na jednotlivé vrstvy se nazývá Separation of Concerns (Oddělení zodpovědnosti) [2]. Podstatou tohoto konceptu je oddělit závislosti jednotlivých vrstev tak, aby nedocházelo k jejich překrývání. K docílení požadovaného rozdělení je u webových aplikací používáno tzv. hrubozrnné rozhraní (coarse–grained – skrývá velké komponenty). Výsledkem je, že programátor, který má na starosti jednu z vrstev, se nemusí zajímat o vnitřní implementaci jiné vrstvy. Jediné co ho zajímá je funkčnost a účelová funkcionalita.

Webová vrstva webové aplikace obsahuje komponenty z Javy EE, jako jsou servlety a JSP. Webová vrstva může provolávat servisní vrstvu, ale neměla by mezi těmito dvěma vrstvami vznikat závislost. Například změna v servisní vrstvě by neměla nijak negativně ovlivnit funkcionalitu webové vrstvy.

Servisní vrstva obsahuje business logiku a komponenty z platformy Java EE, jako jsou Enterprice JavaBeans (EJB). Servisní vrstva má přístup k datové vrstvě, ale opět zde není žádná vyšší závislost. Ve skutečnosti by servisní vrstva neměla ani vědět o existenci datové nebo webové vrstvy. Tato vrstva vystavuje rozhraní pro webovou vrstvu.

Datová vrstva slouží k práci s daty, jako je ukládání, načítání a vyhledávání. Z platformy Java EE obsahuje komponenty jako JDBC (databázový konektor) a JPA (Java Persistance API). Tato vrstva by neměla obsahovat žádnou business logiku. Datová vrstva by měla být abstrakcí konečného perzistentního mechanismu a vystavovat pouze rozhraní pro servisní vrstvu.

Při psaní této kapitoly, čerpal autor z [3].

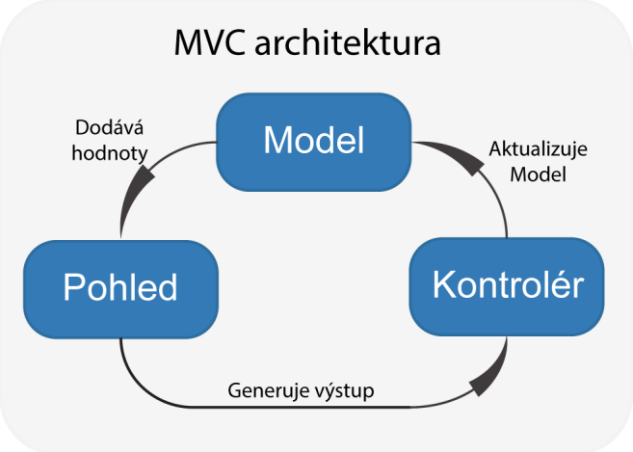
2.2. MVC

Při vývoji aplikace je MVC jedním z nejznámějších návrhových vzorů. Aktuálně je pravděpodobně nejvíce využíván právě při vývoji webových aplikací. Hlavní myšlenkou MVC architektury je oddělení výpočetní logiky programu od prezenční logiky. Jedná se o dobrý příklad Separation of Concerns principu. Při vývoji webové aplikace je tento návrhový vzor využíván hlavně ve webové vrstvě, kde odděluje logiku pro vykreslení stránky od samotného získání či zpracování dat u HTTP požadavků.

Architektura MVC se skládá ze tří logických částí, kterými jsou:

- Model (model)
- View (pohled)
- Controller (kontrolér)

Následující obrázek ukazuje základní podobu MCV architektury.



Obrázek 2 – Obecná MVC architektura [autor]

Model obsahuje data aplikace, na kterých je možné provádět požadované operace. V aplikaci může být realizován jako pole objektů reprezentujících například databázovou strukturu. Objekty modelu mohou využívat návrhový vzor „Manager“, díky kterému získávají logiku potřebnou pro získání dat (například SQL dotazy do databáze). Základním předpokladem MVC architektury je, že model obsahuje i logickou část aplikace. To je ale ve většině implementací řešeno na úrovni kontroléru.

Pohled má na starosti pomocí dat z modelu vytvořit výstup. Nejčastěji se jedná o nějaký systém šablon, které generují požadovaný výstup. V případě webových aplikací se jedná o HTML šablony obohacené o značky (tagy nebo atributy), které umožnují šablonovacímu systému

vkládat proměnné. Zároveň poskytují jednoduché operace s daty. Jedním z příkladů je vypsání v cyklu nebo ověření na platnost nějaké podmínky. Pohled by v architektuře MVC neměl vůbec vědět, odkud data přišla.

Kontrolér je poslední komponentou MVC architektury. Ten má na starosti umožnit uživateli provádět akce, na které budou reagovat model a pohled. Hlavním úkolem kontroléru je vyhodnotit akci od uživatele a podle toho změnit model. Ve chvíli kdy je model změněn, oznámí tuto skutečnost pohledu, který bude muset vygenerovat nový výstup. Samotné oznamování pohledu je možné také řešit pomocí chytřejšího modelu, který v případě změny přepne svůj stav. Pohled pak může tento stav sledovat a generování výstupu spouštět sám.

2.3. WEBOVÝ SERVER A STRUKTURA BALÍKU WEBOVÉ APLIKACE

Pro spuštění webové aplikace je nutné vybrat vhodný server. V rámci Java EE existují různé implementace aplikačních serverů, které mají na starosti spravovat životní cyklus všech komponent. Tuto roli ale z velké části obstarává vybraný framework. Takže je možné použít pouze část aplikačního serveru, kterou je webový server. Ten má na starosti především registraci a mapování servletů, na které budou přesměrovány http požadavky ze strany klienta. Implementací takového serveru je například Tomcat[1] od společnosti Apache, který je používán i v rámci této práce.

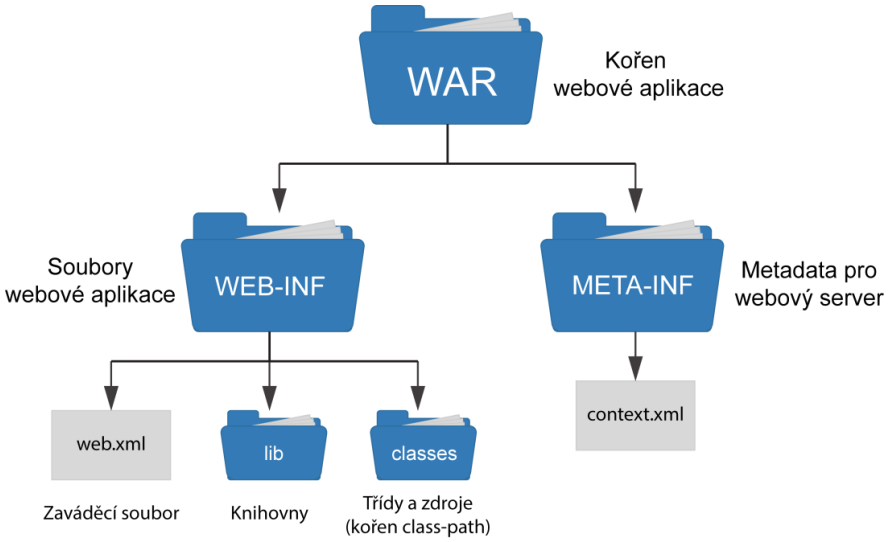
Distribuční balík Tomcat serveru má následující strukturu složek.

- *conf* – konfigurační soubory serveru
- *lib* – sdílené Java knihovny
- *logs* – logovací soubory aplikací
- webapps – zabalené a rozbalené distribuce webových aplikací

Aby byl webový server schopný webovou aplikaci spustit, musí být dodržena určitá struktura její distribuce. Distribuční balíky se nazývají WAR soubory. Jedná se pouze o složku, která je komprimována pomocí ZIP komprese (místo přípony *zip* je *war*). Tyto soubory

[1] http://tomcat.apache.org

jsou při nasazení aplikace na server umístěny do složky webapps, kde si je Tomcat rozbalí a nastartuje. Takový balík pak musí dodržet strukturu, kterou ukazuje následující obrázek.



Obrázek 3 – Struktura balíku WAR [autor]

Zaváděcí soubor *web.xml* slouží jako informace pro webový server pro nastavení mapování servletů. Do verze Servlet API 2.0 byl tento souboru jedinou možností, jak servlet zaregistrovat. Od verze 3.0 přichází další možnosti registrace servletů. Například pomocí anotace `@WebServlet`, které je jako parametr *urlPatterns* nastavena cesta v mapování. Druhým způsobem, který využívá většina frameworků, je pomocí rozhraní `ServletContainerInitializer`. Implementace tohoto rozhraní je při startu provolána s instancí kontextu webové serveru. Do ní je pak možné registrovat servlety v rámci aplikace.

3. PŘEHLED A VÝBĚR STÁVAJÍCÍCH FRAMEWORKŮ

Vzhledem k velkému množství frameworků sloužících k implementacím webových aplikací, není jednoduché vybrat ten správný. Tato práce měla být částečně zaměřená na samotný výběr. Jelikož byla práce tvořena ve spolupráci se společností, která se zabývá i vývojem webových aplikací, a zaměstnává velké množství vývojářů, probíhal výběr frameworků ve více vlnách a s pomocí výše postavených a zkušených vývojářů.

První framework, který bylo nutné vybrat, je zaměřen na serverovou část webové aplikace. Jedná se tedy o jádro, které bude provádět výpočty (v některých případech také generovat samotný obsah) na straně serveru. Vzhledem k bohatým zkušenostem společnosti se zvoleným frameworkem nebylo nutné velkého rozmýšlení. Jedná se o Framework Spring, jenž je v dnešní době nejpopulárnějším Java frameworkem nejen pro webové aplikace, ale zaměřuje se i na mnoho dalších odvětví. Spring Framework je popsán v následující kapitole 3.1.

V druhém kole se hledal vhodný Framework pro implementaci Java skriptové klientské webové aplikace. V tuto chvíli má ve společnosti největší zastoupení framework Angular[2]. Jelikož se ale jedná o moderní společnost, je zde patrná snaha o modernizaci a využívání nejnovějších technologií. Nedávno Facebook představil několik frameworků, které umožňují implementaci právě klientských webových aplikací. To zvedlo vlnu pozitivního (i negativního) zájmu o nové způsoby a možnosti vývoje. Hlavním frameworkem, který byl představen je Facebook React[3]. Ten zpracovává operace s uživatelským rozhraním. Druhým frameworkem je Facebook Flux[4], který má na starosti pozadí aplikace. Oba tyto frameworky jsou popsány v následující kapitole 3.2.

3.1. SPRING FRAMEWORK

Projekt Spring Framework je nejznámější a nejpoužívanější open–source implementací IoC (Inversion of Control) kontejneru. To například dokazuje výzkum z [4], který mapuje využívání nejznámějších frameworků pro vývoj Java webových aplikací.

Primárním účelem Spring frameworku je správa jednotlivých komponent aplikace a zjednodušení vývoje Java aplikací. Jádro tohoto frameworku je možné použít v jednoduché Java

[2] https://angularjs.org
[3] https://facebook.github.io/react
[4] https://facebook.github.io/flux

aplikaci, ale existuje i spoustu rozšíření do dalších okruhů vývoje. Jedním z nejvýznamnějších je rozšíření do webových aplikací.

První verze projektu byla vydána v červnu roku 2003. Byla napsána Rodem Johnsonem, který ji zveřejnil ještě před oficiálním vydáním a to v roce 2002 s publikací jeho knihy Expert One–on–One J2EE Design and Development. V průběhu dalších let došlo k vydání nových průlomových verzí, ale samotný princip v jádru projektu se neměnil.

3.1.1. HISTORIE

- Říjen 2002 – přichází na svět první pohled na implementaci
- Červen 2003 – vydáno pod jménem Java Spring a licencováno jako Apache 2.0
- Březen 2004 – verze 1.0, která obsahovala všechny hlavní funkce
- Říjen 2006 – verze 2.0 s lepším AOP
- Listopad 2007– verze 2.5 s napojováním pomocí anotací
- Prosinec 2009 – verze 3.0 Java konfigurace
- Prosinec 2013 – verze 4.0, kde nebyly žádné velké změny v jádru.

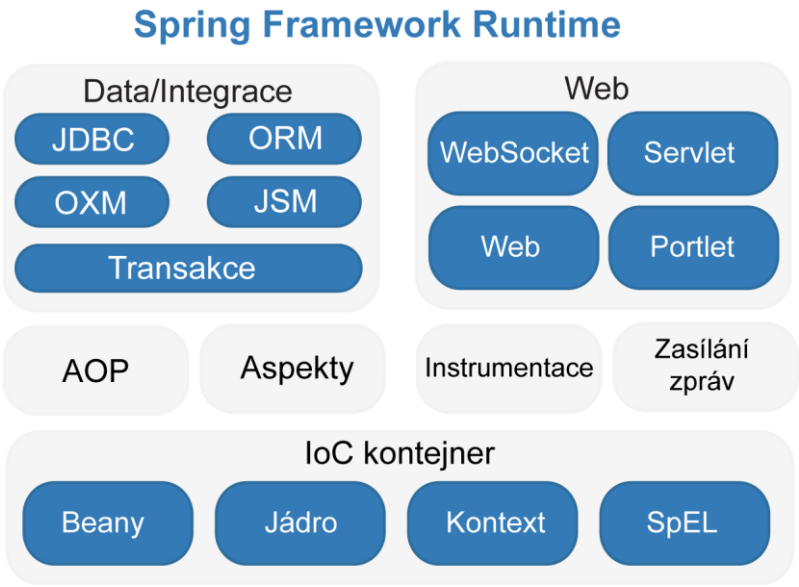
V době psaní této práce je aktuálních verzí projektu Spring Framework 4.2.5, která byla vydána v únoru 2016, a bude i použita pro POC projekt.

3.1.2. MODULY

Spring framework rozděluje svoje funkce zhruba do 20 oddělených modulů. Tyto moduly jsou pak dále sdružovány do následujících skupin:

- Hlavní kontejner
- Přístup k datům a integrace
- Web
- AOP (Aspektově orientované programování)
- Instrumentace
- Zasílání zpráv
- Testování

Následující diagram zachycuje základní přehled jednotlivých modulů Spring frameworku a jejich rozdělení.



Obrázek 4 – Moduly Spring Framework (převzato z [7], upraveno)

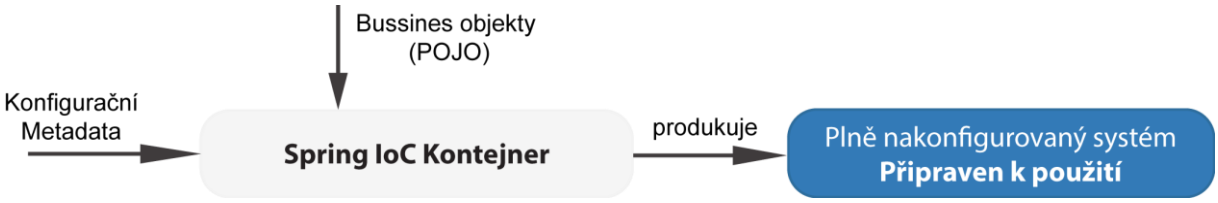
3.1.3. Spring IoC kontejner

Java aplikace je obecný název, který pokrývá velkou množinu aplikací, a to od jednoduchých až po serverové podnikové aplikace. Jedno však mají většinou společné. Skládají se z objektů, které mezi sebou spolupracují a vytváří tak funkční celek.

Ačkoliv platforma Java poskytuje nepřeberné množství funkcí pro vývoj aplikací, postrádá prostředky pro sestavení jednotlivých komponent do souvislého celku. Tudíž tento úkol většinou padá na analytiky a vývojáře. Existují různé návrhové vzory a principy, jako jsou třeba Factory, Buildry, Dekorátory, Service lokátory a další [8]. Tyto vzory jsou ale pouze ověřená doporučení, kterým byl dán název a popis, toho co přináší a jakou problematiku řeší. Tento přístup však vyžaduje, aby každý z vývojářů dodržoval jednotně definované postupy. Problém je, že existuje mnoho výkladů těchto vzorů, a není snadné udržet jednotný přístup napříč projekty.

Spring Framework Inversion of Control (IoC) reaguje na tento problém zavedením jednotného přístupu pro skládání komponent do plně funkční aplikace tím, že převádí zmíněné návrhové vzory do podoby kódu. Ten pak může být jednoduše integrován do vyvíjených aplikací a pomáhá vytvářet robustní a snadno udržovatelné aplikace.

Obecný příklad principu Spring kontejneru je naznačen na následujícím obrázku.



Obrázek 5 – Obecný princip Spring IoC kontejneru (převzato z [7], upraveno)

Hlavním z principů IoC je tzv. Dependency Injection (DI) – přeloženo jako „Injektování závislostí". Při tomto procesu musí být nejprve vytvořena definice pro sestavení objektů (např. argumenty konstruktoru, atributy zpřístupněné skrze setry). Tyto závislosti pak kontejner dodá (injektuje) při vytvoření objektu. Jelikož objekty sami obsahují definici pro správné sestavení, je tento přístup svým způsobem inverzní (IoC).

Objekty, které jsou páteří aplikace, a jsou spravovány Spring IoC kontejnerem, se nazývají *beany* (zrno). Beana je objekt, který je inicializován, sestaven a jinak řízen Spring IoC kontejnerem. Je tak jednou z mnoha objektů aplikace, které jsou mezi sebou nějakým způsobem závislé. Informace o závislosti mezi těmito objekty jsou zapsány pomocí konfiguračních metadat, které využívá kontejner.

Existují různé implementace Spring IoC kontejneru. Všechny jsou vystavené skrze rozhraní `ApplicationContext`. Rozdílné implementace se liší hlavně v možnostech konfigurace a samotné podstatě aplikace. Konfigurační metadatové soubory je možné zapisovat třemi způsoby.

Prvním způsobem je pomocí XML konfigurační souborů, který je podporován od začátku projektu. Hlavní implementaci IoC kontejneru, který na vstupu přijímá XML konfigurační soubory z classpath je `ClassPathXmlApplicationContext`. Následující výpis je ukázkou vytvoření instance zmíněného aplikačního kontextu, kde soubor *xmlContext.xml* je konfiguračním souborem pro kontejner.

Výpis 1 – cz.jbradle.example.spring.context.XmlApplicationContextExample

```
ApplicationContext context =
        new ClassPathXmlApplicationContext("spring/xmlContext.xml");
```

Pro příklad konfigurace beany je vytvořen jednoduchý objekt `ExampleBean`, který obsahuje pouze dva atributy. Jeden je požadován v konstruktoru a druhý jako volitelný atribut, který je nastavován pomocí setru. Jedná se pouze o ukázku, takže není dodrženo například zapouzdření.

Výpis 2 – cz.jbradle.example.spring.context.ExampleBean

```
public class ExampleBean {

    int number;
    String text;

    public ExampleBean(int number) { this.number = number; }

    public void setText(String text) { this.text = text; }
}
```

Následující výpis je ukázkou XML konfiguračního souboru, který říká kontejneru, aby vytvořil instanci objektu `ExampleBean`. Do atributu konstruktoru je nastavena příkladová hodnota a následně je nastaven nepovinný atribut *text*. Toto nastavení je znázorněno v následujícím výpisu.

Výpis 3 – spring/xmlContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="exampleBean" class="cz.jbradle.example.spring.context.ExampleBean">
        <constructor-arg value="123"/>
        <property name="text" value="Example text"/>
    </bean>
</beans>
```

Pro získání beany pak již stačí říci aplikačnímu kontextu pomocí metody *getBean* aby vrátil instanci inicializovaného objektu.

Druhým způsobem konfigurace je s pomocí anotací, které byly představeny ve verzi 2.5. Tento přístup rozšiřuje možnosti konfigurace pomocí XML. Zároveň přináší novou funkcionalitu, a tou je skenování komponent. Pomocí skenování komponent je možně nastavit balík, který bude obsahovat objekty označené anotací `@Component` (nebo jinou, která je touto anotaci označená). Spring IoC kontejner následně prohledá zadaný balík a inicializuje beany podle konfiguračních anotací. Hlavní z těchto konfiguračních anotací je `@Autowired`, která říká kontejneru, aby dodal beanu, která má stejnou definici (implementuje rozhraní, nebo je přímo instancí) jako objekt pod touto anotací. Pro detailnější určení beany je možné využít například anotaci `@Qualifier`.

V následujícím výpisu je ukázka objektu z předchozího příkladu obohaceného o konfigurační anotace.

Výpis 4 – cz.jbradle.example.spring.context.AnnotationExampleBean

```
@Component
public class AnnotationExampleBean {

    int number;

    @Autowired
    String text;

    @Autowired
    public AnnotationExampleBean(int number) {
        this.number = number;
    }
}
```

Takto připravenému objektu stačí pouze zajistit, aby byly v aplikačním kontextu přítomny dvě beany, jedna typu *Integer* a jedna typu *String*. Pro účel příkladu jsou tyto beany zaregistrovány pomocí XML konfigurace. Zároveň je zapotřebí aktivovat konfiguraci pomocí anotací a nastavit balík pro skenování. To je znázorněno na následujícím výpisu.

Výpis 5 – spring/annotationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>
    <context:component-scan base-package="cz.jbradle.example.spring.context"/>

    <bean id="text" class="java.lang.String">
        <constructor-arg value="Example text"/>
    </bean>

    <bean id="number" class="java.lang.Integer">
        <constructor-arg value="123"/>
    </bean>
</beans>
```

Třetím a nejnovějším způsobem konfigurace je pomocí Java konfiguračních tříd. Tento způsob přichází ve verzi 3.0. Jedná se o alternativní způsob proti XML konfiguraci. Tím umožňuje vytvářet čisté Java projekty. Java je bezpečnější při psaní (provádí se kompilace), zároveň je v ní možné snadno vyhledávat a procházet. XML konfigurace mohou snadno nabývat na velikosti a stát se nepřehledným.

Implementací aplikačního kontextu je `AnnotationConfigApplicationContext`, který na konstruktoru přijímá definici konfigurační třídy. Následující výpis ukazuje inicializaci aplikačního kontextu.

Výpis 6 – cz.jbradle.example.spring.context.JavaConfigApplicationContextExample

```
ApplicationContext context =
        new AnnotationConfigApplicationContext(JavaConfig.class);
```

Konfigurační třída musí být označena anotací `@Configuration`. Zaregistrování beany pak probíhá pomocí metody označené anotací `@Bean`, která vrací instanci požadovaného objektu. Následující výpis je příkladem konfigurace se stejným výsledkem jako v prvním příkladu. Pokud není specifikovaný název beany přímo v anotaci, bude v aplikačním kontextu beana zaregistrována pod stejným názvem, jako je název metody (v tomto případě *exampleBean*).

Výpis 7 – cz.jbradle.example.spring.context.JavaConfig

```
@Configuration
public class JavaConfig {

    @Bean
    public ExampleBean exampleBean() {
        ExampleBean exampleBean = new ExampleBean(123);
        exampleBean.setText("Example Text");
        return exampleBean;
    }
}
```

Existuje velké množství názorů na to, kterou metodu používat. Vzhledem k tomu, jak dlouho byla používána XML konfigurace, bývá přechod na novou Java konfiguraci často dáván stranou. V této práci byl zvolen přístup vývoje čistého Java projektu, což znamená žádné XML konfigurace. Proto je využita kombinace konfiguračních anotací s konfigurací Javou.

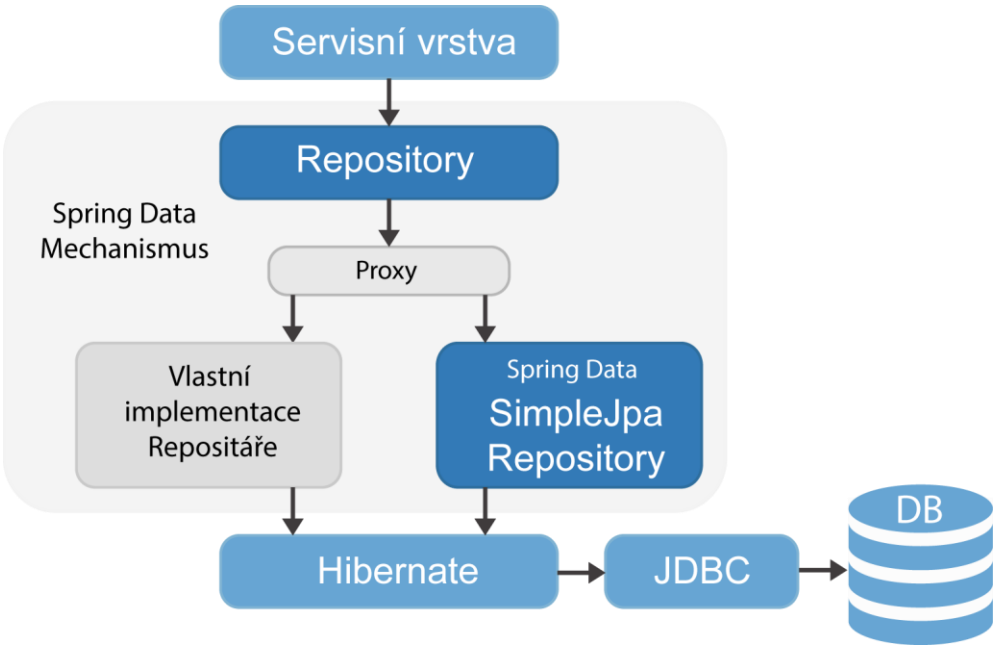
3.1.4. KOMPONENTY PRO VÝVOJ WEBOVÝCH APLIKACÍ

V této kapitole budou popsány hlavní komponenty potřebné pro vývoj webových aplikací pomocí Spring frameworku. Příklady použití budou rozebrány i v praktické části týkající se vytvoření POC projektu.

3.1.4.1. SPRING – DATA JPA

Projekt Spring Data[5] sdružuje velké množství modulů pro práci s daty. Jedním z nejpoužívanějších v rámci vývoje webových aplikací je právě modul Spring Data JPA. Jedná se o Spring implementaci datové vrstvy, která zapouzdřuje JPA třetích stran. V podstatě je postavena na základních kamenech ORM (Object–Relational–Mapping) projektu Hibernate[6]. Umožnuje ale využití i dalších ORM frameworků. Součástí modulu je například podpora pro EclipseLink[7] a OpenJPA[8].

Princip Spring Data JPA je naznačen na následujícím obrázku.



Obrázek 6 – Princip Spring Data [autor]

Spring Data se zaměřuje na vytvoření datové vrstvy s co nejmenším množstvím vlastních implementací pro získání požadovaných dat. Proto zavádí pojem „Repository Abstraction“ (abstrakce uložiště) [9]. Základním rozhraním pro toto řešení je `Repository`. Toto rozhraní přebírá jako argumenty typ entity a typ jejího primárního klíče. Samo o sobě ale neříká, jaké operace s danou entitou bude možné provádět. Slouží tedy k označení rozšiřujícího rozhraní o

[5] http://projects.spring.io/spring-data
[6] http://hibernate.org
[7] http://www.eclipse.org/eclipselink
[8] http://openjpa.apache.org

informace nutné k zavedení do aplikace. Současně existují další rozšiřující rozhraní, která jsou součástí modulu. První z nich je `CrudRepository`, které již definuje základní CRUD (Create–Read–Update–Delete) operace. Druhou úrovní rozhraní repositáře je `PagingAndSortingRepository`, které rozšiřuje repositář o funkce řazení a stránkování entit. Nejvyšší definované rozhraní je `JpaRepository`, které definuje funkce specifické pro JPA. V případě, že není definována vlastní implementace repositáře, je použita Spring implementace `SimpleJpaRepository`, která implementuje všechny tři zmíněné rozhraní.

Vlastní rozhraní se poté definují pomocí rozšiřujícího rozhraní. Samotné dotazy pro data je možné definovat několika způsoby. Hlavním způsobem je pomocí názvu a parametrů metody na rozhraní. Dalším způsobem je použití anotace `@Query`, která jako hlavní parametr přijímá JPA dotaz.

Pro příklad použití je vytvořena databázová tabulka *EXAMPLE_ENTITY*, která obsahuje sloupec *ID* jako primární klíč a sloupec *VALUE* obsahující textovou hodnotu. Dále je vytvořena sekvence *SEQ_ID*, která bude sloužit ke generování hodnoty primárního klíče při vytvoření nového záznamu.

Výpis 8 – sql/create-db.sql

```
CREATE TABLE EXAMPLE_ENTITY (
  ID NUMBER(5) PRIMARY KEY,
  VALUE        VARCHAR(255)
);

CREATE SEQUENCE SEQ_ID INCREMENT BY 1 NOCYCLE;
```

K této tabulce koresponduje entita z následujícího výpisu.

Výpis 9 – cz.jbradle.example.spring.web.model.ExampleEntity

```
@Entity
@Table(name = "EXAMPLE_ENTITY")
public class ExampleEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
            generator = "ID_GENERATOR")
    @SequenceGenerator(name = "ID_GENERATOR", sequenceName = "SEQ_ID")
    @Column(name = "ID", unique = true, nullable = false)
    private Integer id;

    @Column(name = "VALUE")
    private String value;

    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }
    public String getValue() { return value; }
    public void setValue(String name) { this.value = name; }
}
```

Entity jsou definované pomocí Java EE specifikace. Z předchozího výpisu je možné vidět, jakým způsobem je definovaný generátor pro primární klíč. Vzhledem k tomu, že se jedná o příklad, není pojmenování sekvence a název generátoru velmi výstižný. Vždy je dobré si na začátku implementace určit konvenci, se kterou budou tyto názvy vytvářeny. Název databázové sekvence by měl obsahovat tabulku a její účel. Pro tento příklad je vhodné pojmenovat sekvenci *SEQ_EXAMPLE_ENTITY_ID*. To samé platí pro název generátoru, kdy v názvu opět chybí název entity. Lepší název pro tento generátor je *EXAMPLE_ENTITY_ID__GENERATOR*.

Repositář, který bude schopný s takovou entitou pracovat, by tedy měl být definován rozhraním rozšiřující rozhraní `JpaRepository`.

Následující výpis je ukázkou definice rozhraní umožňující uložení entity a její načtení podle hodnoty *value*.

Výpis 10 – cz.jbradle.example.spring.web.persistence.ExampleRepository

```
public interface ExampleRepository
        extends JpaRepository<ExampleEntity, Integer> {

    List<ExampleEntity> findByValue(String value);

    @Query("select e from ExampleEntity e where e.value = :value")
    List<ExampleEntity> findByQuery(@Param("value") String value);
}
```

První metoda je příkladem definování dotazu pomocí názvu a parametrů metody. Taková metoda tedy říká „vyhledej pomocí value“. Implementace repositáře poté bude očekávat jako vstupní parametr hodnotu, kterou doplní do vygenerovaného dotazu. Druhá metoda je příklad použití anotace. Obě metody mají totožný výsledek.

Pro správné spuštění je v konfiguraci potřeba aktivovat JPA repositáře a inicializovat dvě beany. Zapnutí repositářů se provádí pomocí anotace `@EnableJpaRepositories`, která potřebuje mít nastavenou jednu z možností určení pozice definic (rozhraní) repositářů. To je možné provést pomocí parametru *basePackages*, který je nastaven na textovou hodnotu cesty ve struktuře balíku. Případně pomocí parametru *basePackageClasses*, který obsahuje instance typu Class (nebo pole) z onoho balíku.

Dále je nutné zaregistrovat beanu typu `PlatformTransactionManager`, která slouží k základním transakčním operacím. Ty jsou potřebné pro práci s databázovými daty. Důležité je si uvědomit, kdy transakce skutečně začíná. Bez dalšího nastavení budou transakce začínat až ve chvíli provolání repositáře. Takže z něho jsou vráceny entity s uzavřenou session. Jakým způsoben se zapínají transakce před provoláním je popsáno v následující kapitole.

Nakonec `EntityManagerFactory`. Jak již bylo řečeno, Spring Data zapouzdřují hlavně Hibernate. Tudíž je možné použít jejich entity manažer. Pro snadnou konfiguraci obsahuje modul `HibernateJpaVendorAdapter`, který zapouzdřuje základní konfigurační úkony.

Vzhledem k tomu, že byla v příkladu použita implementace `JpaTransactionManager`, která pro správnou inicializaci vyžaduje `EntityManagerFactory`, obě beany očekávají, že bude v aplikačním kontextu zaregistrován `DataSource`. V praktické části je ukázka jeho vytvoření bez nutnosti vytvoření externí databáze.

V následujícím výpisu je ukázána minimální konfigurace nutná pro správnou inicializaci aplikačního kontextu.

Výpis 11 – cz.jbradle.example.spring.web.config.PersistenceConfig

```
@Configuration
@EnableJpaRepositories(basePackageClasses = ExampleRepository.class)
class PersistenceConfig {

    @Bean
    public PlatformTransactionManager transactionManager(
            DataSource dataSource) throws Exception {
        return new JpaTransactionManager(
                entityManagerFactory(dataSource).getObject()
        );
    }

    @Bean
    public FactoryBean<EntityManagerFactory> entityManagerFactory(
            DataSource dataSource) {
        LocalContainerEntityManagerFactoryBean factory =
                        new LocalContainerEntityManagerFactoryBean();
        HibernateJpaVendorAdapter vendorAdapter =
                        new HibernateJpaVendorAdapter();
        factory.setDataSource(dataSource);
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("cz.jbradle.example.spring.web.model");
        return factory;
    }
}
```

Použitá implementace entity manažeru podporuje skenování balíku s entitami. Je možné mu dodat cestu k entitám pomocí metody *setPackageToScan*. Tento přístup nahrazuje konfiguraci JPA pomocí souboru *persistence.xml*. V tuto chvíli stačí pouze zajistit, aby v aplikačním kontextu existovala beana typu DataSource, a již je možné začít plně využívat možnosti Spring Data JPA.

Při psaní této kapitoly, čerpal autor z [9] a [10].

3.1.4.2. SPRING – SERVICE

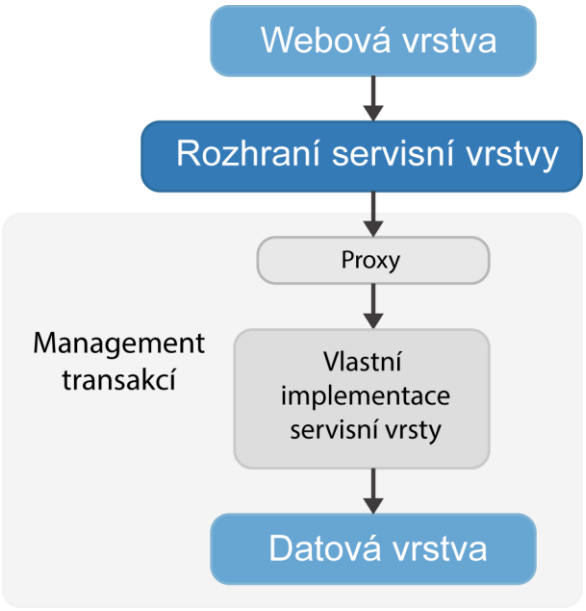
Prvním krokem při vývoji servisní vrstvy by mělo být správné definování rozhraní, pomocí kterého bude kontrolér získávat a upravovat data (může ale také například spouštět asynchronní operace). Toto rozhraní se pak stane vstupní bránou do samotné aplikace. Výsledek definice rozhraní by měl být rozdělen do několika menších rozhraní. Toto rozdělení je pak většinou závislé na rozhraních datové vrstvy (každá entita = jedno rozhraní) a požadavcích webové vrstvy.

Implementace servisní vrstvy pomocí Spring je asi nejsnazší částí. Vzhledem k tomu, že obsahuje pouze logickou část aplikace, je její náročnost pouze ze strany funkcionality.

Kvůli tomu, že servisní vrstva obsahuje převážně logiku aplikace, není možné jí nějaký způsobem generalizovat. Takže se pro její implementaci využívají pouze základní vlastnosti IoC kontejneru. Implementaci rozhraní servisní vrstvy stačí zaregistrovat pomocí konfigurace do aplikačního kontextu tak, aby bylo možné k ní odkudkoliv přistoupit. Aby byl kód přehlednější, a bylo možné snadno určit, v jaké vrstvě se aplikace právě nachází, definuje Spring anotaci `@Service`, která je označená jako komponenta.

Hlavním přínosem frameworku Spring do servisní vrstvy je zpravování transakcí. V předchozí kapitole byl představen projekt Spring Data JPA. Pro jeho konfiguraci je nutné, aby obsahoval transakčního manažera. Toho je možné využít i na úrovni servisní vrstvy, a začít transakci dříve než při provolání repositáře. K tomuto účelu slouží anotace `@Transactional`, kterou lze aplikovat na celou třídu nebo případně pouze na vybrané metody. V této anotaci lze nastavit několik konfiguračních parametrů, které budou použity při založení transakce (metoda má přednost před třídou). Označená metoda nebo třída musí být provolána skrze proxovaný objekt, aby bylo Spring AOP schopno upozornit transakčního manažera na požadavek o založení transakce. Pokud již transakce existuje (a nejedná se o založení sub transakce) nebude vytvořena nová. V případě vnitřního provolání metody nebude transakce spuštěna. Dále je zapotřebí v konfiguraci aktivovat transakce na úrovni aplikačního kontextu. To je možné provést jednoduše pomocí anotace `@EnableTransactionManagement`.

Následující obrázek ukazuje, jakým způsobem by měly být spravovány komponenty servisní vrstvy.



Obrázek 7 – Pohled na servisní vrstvu [autor]

Následující výpisy jsou příklady konfigurace, jednoduchého rozhraní a implementace servisní komponenty. Tato komponenta obsahuje metodu pro vrácení kolekce hodnot z databáze.

Výpis 12 – cz.jbradle.example.spring.web.config.ServiceConfig

```
@Configuration
@EnableTransactionManagement
@Import({PersistenceConfig.class})
@ComponentScan(basePackageClasses = ExampleService.class)
class ServiceConfig {
}
```

Výpis 13 – cz.jbradle.example.spring.web.service.ExampleService

```
public interface ExampleService {
    List<String> getAllValues();
}
```

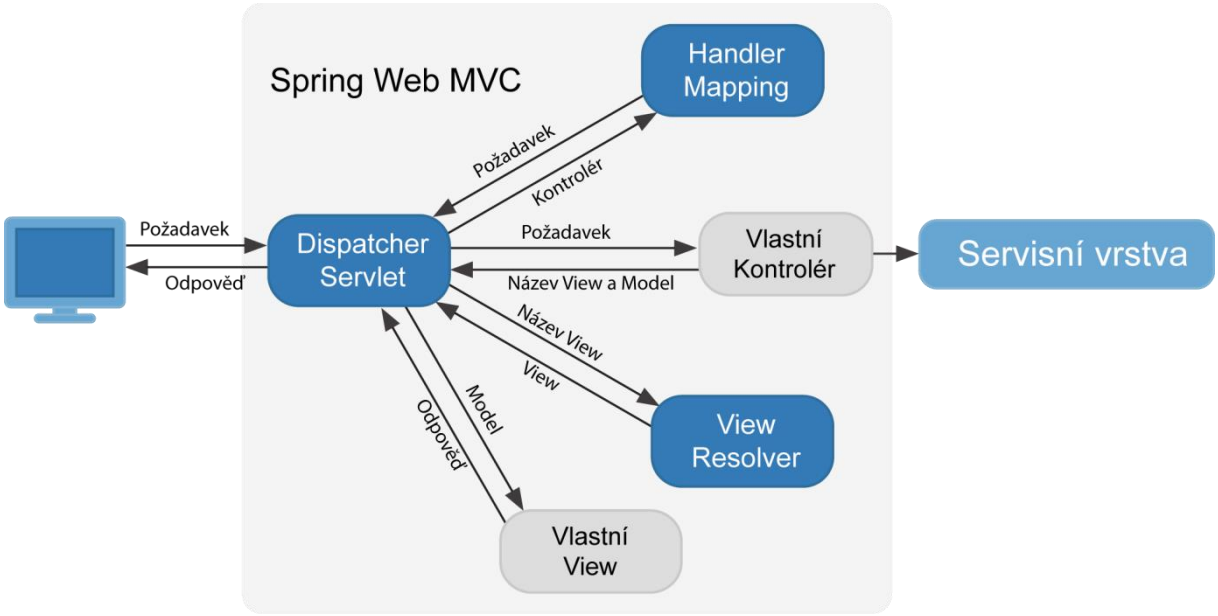
Výpis 14 – cz.jbradle.example.spring.web.service.ExampleServiceImpl

```
@Service
class ExampleServiceImpl implements ExampleService {

    @Autowired
    private ExampleRepository repository;

    @Override
    @Transactional(readOnly = true)
    public List<String> getAllValues() {
        return repository.findAll()
                .stream()
                .map(ExampleEntity::getValue)
                .collect(Collectors.toList());
    }
}
```

3.1.4.3. SPRING – WEB MVC

Jedná se o Springovou implementaci MVC návrhového vzoru popsaného v kapitole 2.2. Kompletní mechanizmus zapouzdřuje `DispatcherServlet`, který složí jako vstupní brána pro HTTP požadavky z prohlížeče. Ten inicializuje dva základní aplikační kontexty. Prvním kontextem je hlavní (nazývaný též aplikační), který spravuje životní cyklus komponent servisní a datové vrstvy. Druhým kontextem je servletový, který má na starosti komponenty spojené právě s Web MVC. Servletový kontext v případě nenalezení beany ještě vyhledává beanu v aplikačním kontextu. V opačném směru na sebe kontexty nevidí. Servletový kontext v sobě obsahuje několik základních komponent, potřebných pro správné vyřízení příchozího požadavku.

Pro vyřízení požadavku jsou zapotřebí dvě hlavní komponenty, které jsou součástí Spring MVC. První důležitou komponentou je `HandlerMapping`. Ten má na starosti určit, která metoda bude na kontroléru zavolána. Nejpopulárnější implementací je `RequestMappingHandlerMapping`, který podporuje mapování pomocí anotací. Ale existují i další implementace. Druhý je `ViewResolver`, který na základě názvu pohledu a požadavku určí `View`. To doplní data z modelu, a vygeneruje odpověď. Princip implementace Spring MVC je na následujícím obrázku.



Obrázek 8 – Tok zpracování požadavku ve Spring MVC (převzato z [14], upraveno)

Z předchozího obrázku je vidět, že ke správnému zpracování požadavku chybí ještě dvě komponenty.

První jsou vlastní kontroléry. Ty umožnují přístup do samotného jádra aplikace, které je zapouzdřeno rozhraním servisní vrstvy. Jak již bylo zmíněno, nejpopulárnější konfigurací kontroléru je pomocí anotací, která byla představena ve verzi 2.5. Anotace pro označení objektu jako kontrolér je `@Controller`, která je označená anotací `@Component`, takže umožňuje aplikačnímu kontextu takový objekt zaregistrovat jako beanu pomocí skenování komponent. K samotné konfiguraci toto ale nestačí. Ještě je nutné definovat metodu, kterou dispečer zavolá pro získání modelu a názvu pohledu. K tomuto účelu slouží anotace `@RequestMapping`, která jako hlavní hodnotu přijímá text reprezentující cestu (nebo její část) z požadavku. Pomocí dalších atributů anotace je možné detailněji specifikovat mapovaní. Například pomocí atributu *method* je možné definovat typ požadavku (GET, POST, …). V následujícím výpisu je ukázka kontroléru, který při požadavku */example* nastaví do modelu kolekci hodnot získaných ze servisní vrstvy. Poté již pouze vrátí název pohledu, který bude mít dispečer k dispozici pro výběr správného pohledu, které vygeneruje odpověď.

Výpis 15 – cz.jbradle.example.spring.web.controller.ExampleController

```
@Controller
public class ExampleController {

    @Autowired
    private ExampleService service;

    @RequestMapping("/example")
    public String example(Model model) {
        model.addAttribute("exampleValues", service.getAllValues());
        return "example";
    }
}
```

Nyní již handler mapping požádá aplikační kontext o všechny beany typu kontrolér, a pomocí jejich konfiguračních anotací nastaví mapování.

Existuje více možností zápisu metod v kontroléru, se kterými si dispečer dokáže poradit. Například nemusí být zadán objekt Model, a dispečer bude očekávat na výstupu instanci objektu `ModelAndView`. Pro detailnější možnosti konfigurace kontrolérů je možné využít dokumentace projektu Spring MVC.

Druhou komponentou je pohled (View), který má na starosti vygenerovat obsah odpovědi. Existuje několik implementací pohledů. Nejznámější implementací je `JstlView`, které je součástí Spring projektu. Tento pohled pracuje s klasickými JSP (Java Server Pages) šablonami. K tomu, aby se pohled správně připravil, musí být použita vhodná implementace view resolveru. Ve většině případů jsou samotné JSP šablony součástí projektu, takže je možné využít `UrlBasedViewResolver`, který šablony načítá pomocí relativní cesty od kořene sestavené aplikace. Struktura webové aplikace je popsána v kapitole 2.3.

V následujícím výpisu je ukázka JSP šablony. Jedná se o vypsání hodnot z kolekce v modelu do tagu pro nadpis.

Výpis 16 – WEB-INF/jsp/example.jsp

```
<!-- JSP -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<body>
<c:forEach items="${exampleValues}" var="value">
    <h1>${value}</h1>
</c:forEach>
</body>
</html>
```

V rámci této práce je pro generování pohledu využita novější technologie Thymeleaf. Popis použití a konfigurace je uveden v praktické části kapitola 4.4.3.

Nyní je potřeba zajistit, aby během startu webového serveru proběhlo zaregistrování dispather servletu a inicializace aplikačního kontextu. Konfigurační třída by měla obsahovat informaci pro aplikační kontext, která zařídí, aby dohledal požadované beany s anotací `@Controller`. K tomu je možné využít anotaci `@ComponentScan`, které je jako parametr *basePackageClasses* zadána definice třídy jednoho z kontrolérů v balíku, který má být prohledán. Dále je potřeba zaregistrovat view resolver. Následující výpis je ukázkou konfigurace servlet kontextu.

Výpis 17 – cz.jbradle.example.spring.web.config.WebConfig

```
@Configuration
@ComponentScan(basePackageClasses = ExampleController.class)
class WebConfig {

    @Bean
    public ViewResolver viewResolver(){
        InternalResourceViewResolver viewResolver =
                        new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/jsp/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

Pro zaregistrování dispather servletu Spring vystavuje rozhraní `WebApplicationInitializer`, který využívá možnosti inicializace servletu v rámci aplikace. Nejjednodušší způsob inicializace je pomocí abstraktní třídy `AbstractAnnotationConfigDispatcherServletInitializer`, která vyžaduje implementaci tří metod. První metoda je *getServletMappings*, která musí vrátit mapování dispather servletu. Druhá metoda je *getServletConfigClasses*, která vrátí pole konfiguračních tříd.

Ty jsou pak použity k inicializaci servlet kontextu. Zde by mělo být konfigurováno vše, co obstarává webová vrstva aplikace. A třetí metoda *getRootConfigClasses* vrací pole konfiguračních tříd, které slouží k inicializaci aplikačního (root) kontextu. Zde se pak nachází konfigurace ke zbytku aplikace.

Následující výpis je ukázkou inicializační třídy, která umožní spuštění aplikace, sestavené v rámci předchozích příkladů.

Výpis 18 – cz.jbradle.example.spring.web.config.WebInitializer

```
public class WebInitializer
        extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{ WebConfig.class };
    }

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{ ServiceConfig.class };
    }
}
```

Nyní již stačí spustit web server a zavolat požadavek */example*.

Při psaní této kapitoly, čerpal autor z [13].

3.1.4.4. SPRING – REST API

REST (Representational State Transfer) – je architektura rozhraní, navržená pro distribuované prostředí. Jedná se o rozšíření HTTP požadavků o možnosti čtení, editování nebo mazání dat. Z názvu vyplývá, že systém, který získává data ze zdroje, pracuje pouze s reprezentací dat. V této práci je využit pro komunikaci mezi klientskou a serverovou aplikací.

Stav aplikace je reprezentovaným daty, které odběratel získá. Data získaná z REST rozhraní, mohou být vyjádřena několika způsoby. Nejčastěji však v podobě XML nebo JSON. Pro operace s daty jsou využívány čtyři základní metody http požadavků, pomocí kterých je možné provádět základní CRUD operace.

- GET – získání stavu aplikace (reprezentované daty v odpovědi)
- POST – zaslání dat
- PUT – přidání dat (specializovaný POST)
- DELETE – smazání dat

Implementaci REST rozhraní pomocí String frameworku je možné provést jednoduše pomocí MVC modulu popsaného v předchozí kapitole. Výpisy uváděné v této kapitole jsou ze zdrojových kódů umístěných v modulu *examples/spring–rest–example.*

Principem implementace je zajistit, aby data z kontroléru byla zpracována do těla odpovědi některou z možných reprezentací. Jedním ze způsobů, jak převést data a reprezentační podobu, je vytvořit vlastní implementaci `ViewResolver`, který bude pro všechny názvy pohledu vracet stejnou implementaci pohledu `MappingJackson2JsonView`. Tento pohled přebere data z modelu, a přemapuje je na korespondující JSON objekt. Následující výpis je ukázkou implementace takového resolveru.

Výpis 19 – cz.jbradle.example.spring.rest.view.JsonResolver

```
public class JsonViewResolver implements ViewResolver {

   @Override
   public View resolveViewName(String viewName, Locale locale)
         throws Exception {
      return new MappingJackson2JsonView();
   }
}
```

Ten je pak možné zaregistrovat místo původně použitého resolveru, který hledal JSP šablony. Následující výpis je ukázkou registrace beany, která nahradí původní `InternalResourceViewResolver`.

Výpis 20 – cz.jbradle.example.spring.web.config.WebConfig

```
   @Bean
   public ViewResolver jsonViewResolver(){
      return new JsonViewResolver();
   }
```

Při stejném dotazu jako v předchozí kapitole se nyní místo webové stránky vrátí JSON reprezentace dat. Následující výpis ukazuje data vrácená v odpovědi na požadavek */example*.

Výpis 21 – JSON reprezentace příkladového modelu

```
{
  "exampleValues": [
    "value1",
    "value2",
    "value3",
    "value4"
  ]
}
```

V praktické části je použit jeden z dalších možných Spring implementací REST rozhraní.

3.2. REACT + FLUX

Jak již bylo zmíněno v úvodu hlavní kapitoly, zde budou popsány dva hlavní frameworky od společnosti Facebook, které umožňují implementaci klientské webové aplikace. Jedná se o React a Flux.

3.2.1. REACT

Tento framework byl poprvé použit v roce 2011 pro komponentu s novinkami ve webové aplikaci Facebook. Byl vytvořen Jordanem Walkerem, který je softwarovým inženýrem Facebooku. Jako open–source byl zpřístupněn až v květnu roku 2014. Jedná se o JS knihovnu, která umožňuje provádět operace s DOM (Document Object Model – objektový model dokumentu) HTML dokumentu.

Základní myšlenka pro optimální zobrazování změn spočívá ve faktu, že samotné prohlížeče jsou implementované tak, aby byly schopny rychle vykreslovat změny v DOM. To znamená, že dokáže efektivněji přepsat pouze část webového obsahu na rozdíl od celé stránky. K tomuto účelu zavedl React takzvaný Virtuální DOM. Jedná se o reprezentaci zobrazovaného obsahu, která je uložená v paměti. Proti němu jsou prováděny veškeré změny, které jsou pak porovnány s aktuálním zobrazovaným obsahem (virtuální DOM si tedy drží dva stavy před/po), a do prohlížeče jsou promítnuty pouze rozdíly. To umožňuje vývojáři programovat tak, jako by měnil kompletní obsah, ale React zajistí, aby se v prohlížeči měnilo pouze to, co je skutečně potřeba.

Další novinkou v rámci frameworku React je syntaxe JSX umožňující jednoduché vytváření komponent. Jedná se o rozšíření Java skriptové syntaxe o klasické XML tagy. Každá z komponent musí implementovat metodu *render*. Ta vrací obsah, který bude zapsán do virtuálního DOMu. Následující výpis je ukázkou jednoduché komponenty, která v parametrech přebírá atribut *value* (hodnota) a vykresluje nadpis s touto hodnotou. Všechny JS zdrojové kódy jsou v této práci psány ve standartu ES6 (ECMAScript 2015). Díky tomu je možné přiblížit JS vývoj i Java programátorům.

Výpis 22 – Příkladová React komponenta

```
import React from "react";

class ExampleComponent extends React.Component {
    render() {
        return (
            <h1> {this.props.value}</h1>
        );
    }
}
export default ExampleComponent;
```

Detailní popis implementace komponent a tvorby aplikace je popsán v praktické části.

3.2.2. FLUX

Flux je architektura, kterou využívá Facebook pro implementaci klientských webových aplikací. Doplňuje vlastnosti React frameworku o řízení toku dat. V podstatě se jedná o alternativní návrhový vzor, který je konkurencí MVC návrhového vzoru. Samotný projekt Facebook Flux je implementací ono návrhové vzoru. Takže je možné ho přímo využívat.

Flux obsahuje čtyři základní komponenty.

- Store[9] (sklad)
- Action (akce)
- Dispatcher (dispečer)
- View (pohled)

Sklad obsahuje logiku aplikace a udržuje její stav. Může být přiřazována k modelu v MVC vzoru, ale na rozdíl od něho v sobě udržuje stav více objektů. Se stavem objektů zároveň může udržovat stav celé aplikace (nebo její části). Je možné mít napříč aplikací několik skladů, v podstatě se to i doporučuje. Pro každou doménu aplikaci by měl existovat aspoň jeden sklad.

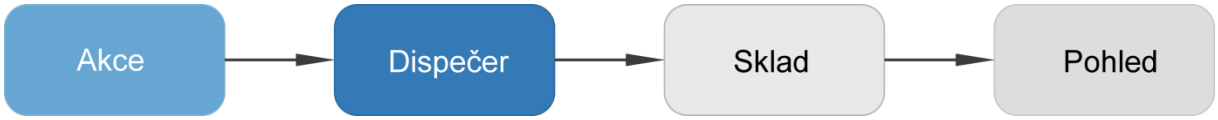
Akce je objekt, který je vyvolán v některé části aplikace. Může se jedna o akci vyvolanou pohledem (uživatelem), nebo například akci vyvolanou nějakým plánovačem. Hlavní výhodou vzoru je to, že akce může být vyvolána odkudkoliv, například z React komponenty (z pohledu). Tím přináší možnost více směrového toku dat.

Dispečer je samostatná jednotka v celé aplikaci, která má na starosti veškerý tok dat. Jedním z jeho hlavních úkolů je distribuovat jednotlivé akce příslušným skladům. Všechny sklady mají v dispečeru zaregistrovanou zpětnou vazbu (callback funkci), která je provolána v případě zjištění nové akce.

[9] Vzhledem k tomu, že „Store“ nemá na starosti pouze práci s daty, ale obsahuje i logiku aplikace, je možné říci, že překlad jako „Sklad“ není nejlépe zvolený.

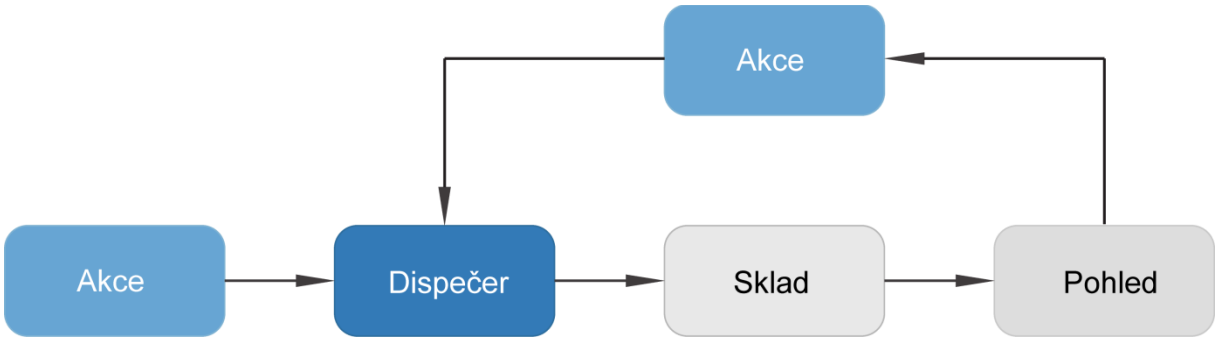
Pohled je objekt, který má na starosti zobrazování dat. Aktuálně jsou jako implementace pohledu nejčastěji využívány React komponenty. V případě, že získají zpětnou vazbu od skladu, doptají sklad o nová data a pomocí metody *setState* nastaví nový stav komponenty. Vnitřně je pak provolána metoda *render*, která překreslí komponentu a aktualizuje stav ve všech podřízených komponent.

Následující obrázek ukazuje tok dat, při zpracování akce v jednom směru.



Obrázek 9 – Flux – jedno–směrný dok dat (převzato z[15], upraveno)

Ten však neukazuje hlavní výhodu vzoru Flux. Tou je možnost více směrného toku dat, který je znázorněn na následujícím obrázku.



Obrázek 10 – Flux – obou–směrný dok dat (převzato z [15], upraveno)

V tomto případě vyvolá pohled v návaznosti na změně stavu novou akci, která je opět zpracována dispečerem. Toto přináší mnoho výhod. Jednou z nejdůležitějších výhod je jednoduché vytváření testů. Protože při provedení operace mohou být všechny zpracování akcí rozpadnuty na jednoduchou podobu, kterou ukazuje Obrázek 9. Tu pak lze jednoduše pokrýt testem (testy).

Vzhledem k tomu, že se jedná pouze o návrhový vzor, bylo nutné vybrat i vhodnou implementaci. V závislosti na příspěvku z [20] byla vybrána implementace Reflux. V praktické části je použit framework, který je založený na návrhovém vzoru Flux, která ale není Facebook implementací. Popis frameworku a ukázky jeho použití jsou v ní popsány.

4. PROOF OF CONCEPT VYBRANÝCH TECHNOLOGIÍ

Pojem Proof of Concept (POC) označuje obecný postup při realizaci určité metody nebo myšlenky k prokázání její proveditelnosti. Případně jde o demonstraci, která si bere za cíl ověřit, že některé koncepce nebo teorie mají potenciál k využití. POC je obvykle malého rozsahu a je zaměřena na jednu určitou problematiku.

Jako POC pro ukázku vybraných technologií byla navrhnuta jednoduchá webová aplikace. Tato webová aplikace je pouze seznamem objektů reprezentujících jednotlivé frameworky, ve které bude možné provádět základní operace nad těmito entitami. Jako jsou vytvoření, editace, zobrazení detailu, vyhledání a smazání.

Veškeré zdrojové kódy, obsažené v této práce, jsou k dispozici ve veřejném repositáři *GitHub*[10], konkrétně na adrese: https://github.com/jbradle/java–web–frameworks–poc

4.1. VÝVOJOVÉ PROSTŘEDÍ

Vzhledem k tomu, že je tato práce z části technická, a obsahuje praktické ukázky, je její neoddělitelnou součástí také výběr vhodného vývojového prostředí podporující daný programovací jazyk. Vývojové prostředí je často nazýváno jako IDE („Integrated Development Environment“, Software s integrovanými vývojovými nástroji). Součástí většiny IDE jsou vývojové nástroje, jako jsou například editor, kompilátor, debuger a další. Všechny mají za úkol jediné, a to zjednodušit a zefektivnit vývoj aplikace. Pro platformu Java EE existuje mnoho vývojových prostředí, mezi nejznámější patří například Eclipse[11], NetBeans[12] nebo InteliJ IDEA[13].

Veškeré výpisy z kódu, diagramy a i samotné POC projekty jsou vytvářeny pomocí autorem oblíbeného IDE InteliJ IDEA. Ta je největším produktem společnosti JetBrains. Jedná se o velmi přehledné vývojové prostředí. V základu toto vývojové prostředí je obsaženo mnoho podpůrných nástrojů pro vývoj právě webových aplikací. Mezi nejzajímavější součásti patří následující:

[10] https://github.com
[11] https://eclipse.org
[12] https://netbeans.org
[13] https://www.jetbrains.com/idea

- **Maven** – podpora při vývoji Maven projektů
- **Spring** – kompletní analýza konfigurací aplikačních kontextů
- **Persistance** – podpora pro ORM
- **Databázový klient** – klient pro velkou škálu databázových systémů

Tento výpis je pouze ukázkou základních a velmi užitečných komponent pro vývoj webové aplikace. InteliJ IDEA obsahuje i repositář rozšiřujících nástrojů, kterých je nepřeberné množství. Ty je možné do prostředí doinstalovat a získat tak další podporu.

Jedinou nevýhodou tohoto produktu je jeho cena. Nově zavedli v JetBrains systém pronájmu a již není možné zakoupit licenci na určitou verzi s tím, že by platila napořád. První rok pronájmu pro fyzickou osobu nyní vyjde cca na 4 000Kč. Což může většinu vývojářů odradit. JetBrains ale nabízí i verzi zdarma, která ale neobsahuje většinu podpůrných nástrojů. To bohužel snižuje jeho úroveň.

4.2. ŠABLONA POC PROJEKTU

Prvním krokem při vývoji webové aplikace je vytvoření HTML šablony. U každé šablony by na první pohled mělo být zřejmé, k čemu bude daná aplikace sloužit. Z tohoto důvodu je dobré, vytvářet takovou šablonu jako funkční celek. Pro tyto účely byla šablona zpracována jako samostatný, oddělený projekt. Je uložena v kořenovém adresáři repositáře *java–web–framework–poc* ve složce *tempalte–poc.*

Jedná se o skupinu HTML souborů, které budou sloužit jako maska pro POC projekty. Při jejich vytváření byl využit projekt Bootstrap[14], který je oblíbeným a jednoduchým HTML, CSS a JS frameworkem pro rychlý vývoj responsivních webových stánek. Další nástroje využité pro snadnou implementaci webových stránek, které byly použity, jsou:

- *Npm*[15] – balíčkovací manažer
- *Gulp*[16] – automatizační nástroj
- Browser–sync[17] – nástroj pro synchronizaci s prohlížečem

[14] http://getbootstrap.com
[15] https://www.npmjs.com
[16] http://gulpjs.com

Zmiňované nástroje budou blíže vysvětleny v pozdějších kapitolách. V tomto případě byl *npm* použit pro získání distribuce *bootstrap*, *jquery* a *font–awesome*. Dále pro získání vývojových balíků *gulp* a *browse–sync*. Všechny závislosti jsou popsány v souboru *package.json* viz:

Výpis 23 – package.json

```
{
  "name": "poc-templates",
  "version": "1.0.0",
  "description": "Java Web Frameworks POC project frond-end templates",
  "author": "Jiri Bradle <bradle.jiri@gmail.com>",
  "devDependencies": {
    "browser-sync": "^2.9.11",
    "gulp": "^3.9.0"
  },
  "dependencies": {
    "bootstrap": "^3.3.6",
    "font-awesome": "^4.5.0",
    "jquery": "^2.1.4"
  }
}
```

Gulp je v tomto případě použit ke spuštění a inicializaci browser–sync, který po spuštění bude kontrolovat všechny změny v HTML šablonách a v reálném čase je promítat do prohlížeče. Pro jeho nastavení je zapotřebí definovat soubor *gulp.js* viz:

Výpis 24 – gulpfile,js

```
var gulp = require('gulp');
var browserSync = require('browser-sync');

gulp.task('browser-sync', function () {
    var files = [
        '*.html'
    ];

    browserSync.init(files, {
        server: {
            baseDir: '.'
        }
    });
});

gulp.task('default', ['browser-sync']);
```

[17] https://www.browsersync.io

Gulp tedy pouze jednoduše spustí Browser–sync, který bude reagovat na všechny změny v HTML souborech, jež jsou umístěny v kořenovém adresáři. Nejprve musí být Gulp nainstalovaný jako globální závislost (Stačí pouze jednou). Toho se dá snadno docílit pomocí Npm viz příkaz:

```
$ npm install –g gulp
```

Nyní již stačí nainstalovat moduly pomocí Npm a spustit hlavní Gulp úkol viz příkazy:

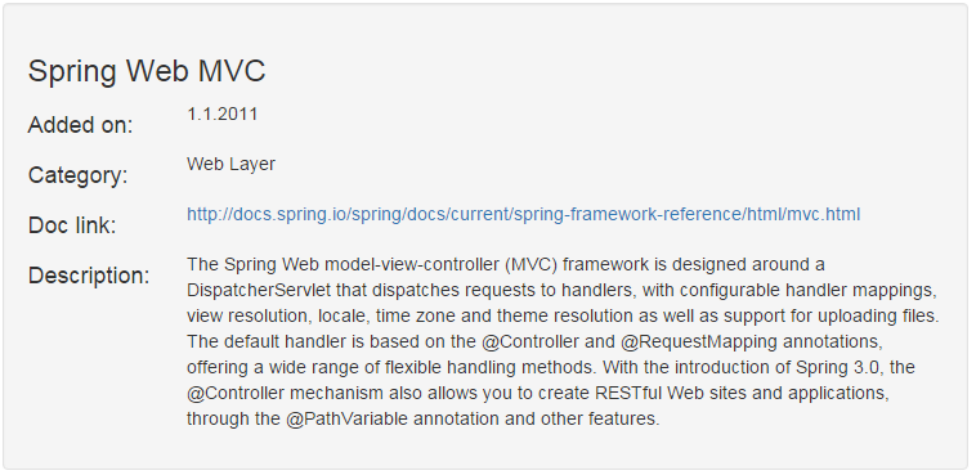
```
$ npm install
$ gulp
```

Takto může vypadat jednoduchý základ pro vývoj šablon, který ušetří spoustu času a zároveň může být jednoduše distribuován bez zbytečných zdrojových kódů třetích stran.

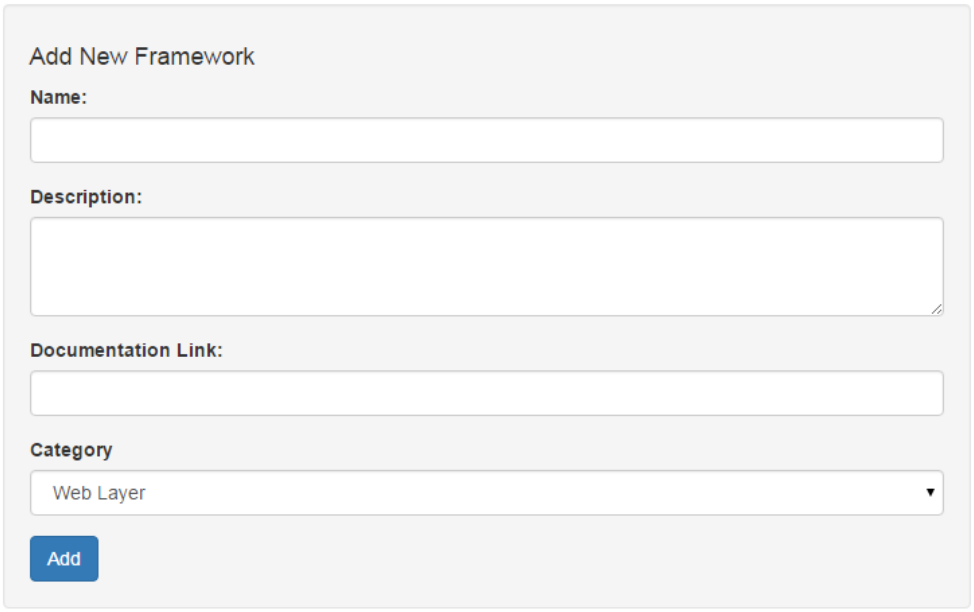
Pro potřeby POC byl vytvořen jednoduchý příklad webové aplikace, která by měla postačit k ukázce jednotlivých přístupů k tvorbě webových aplikací. Jedná se o jednoduchý seznam entit (v tomto případě frameworků), kde je možné provádět základní operace. Hlavní stránka je zobrazení seznamu všech frameworků viz Obrázek 11.

Obrázek 11 – Šablona seznamu frameworků [autor]

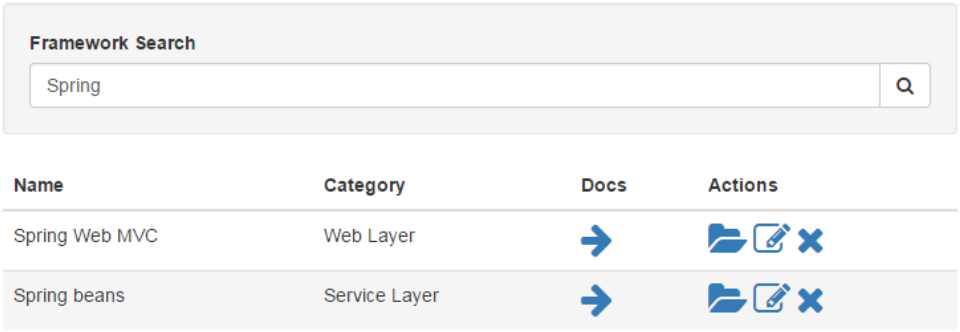
Každý z řádků obsahuje kolekci operací, které jsou možné s danou entitou provést. První ikona složky slouží k zobrazení detailu, druhá ikona pro editaci entity a třetí pro její smazání. Pro zobrazení detailu je použita šablona viz Obrázek 12, pro přidání nového frameworku šablona viz Obrázek 13 a pro vyhledání šablona viz Obrázek 14.



Obrázek 12 – Šablona detailu frameworku [autor]



Obrázek 13 – Šablona pro přidání nového frameworku [autor]



Obrázek 14 – Šablona pro vyhledání frameworku [autor]

4.3. Předek pro konfiguraci POC projektů

Udržovat projekty, které obsahují mnoho závislostí, může být ve většině případů celkem náročné. Z tohoto důvodu vznikl projekt od firmy Apache, který se nazývá Maven[18]. Jedná se o nástroj pro správu a sestavení Java aplikací založený na principu POM (Project object model). Po instalaci Maven se vytvoří lokální repositář, do kterého budou ukládány všechny sestavy a distribuce závislostí v podobě JAR knihoven. Pokud není knihovna k dispozici, pokusí se jí Maven stáhnout z oficiálního maven repositáře[19].

Všechny potřebné informace pro sestavení projektu jsou obsaženy v soboru *pom.xml*, jenž je součástí každého z modulů. POC moduly jsou sdružené pod jedním otcovským (parent) projektem. V tomto předkovi jsou definované společné vlastnosti všech POC projektů, tudíž se během vývoje bude rozrůstat.

Výpis 25 – parent pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>cz.jbradle.poc</groupId>
    <artifactId>java-web-frameworks-poc</artifactId>
    <packaging>pom</packaging>
    <version>1.0.0-SNAPSHOT</version>
    <name>Java Web Frameworks POC</name>
    <description>Parent pom for all Java Web Frameworks POC project</description>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

[18] https://maven.apache.org
[19] http://mvnrepository.com

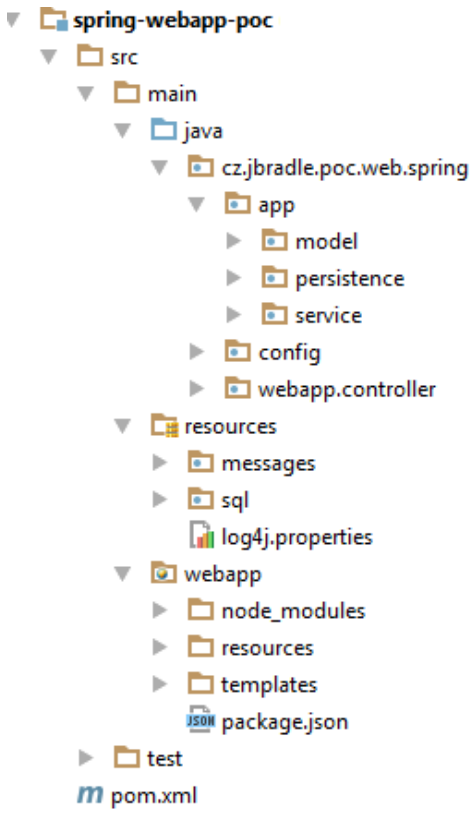
V ukázce se jedná pouze o jednoduchou konfiguraci. Je zde řečeno, jak se bude projekt jmenovat a do jaké skupiny patří. Dále pomocí parametru *packaging* nastaveného na hodnotu *pom* je indikováno, že se jedná o projekt sdružující moduly (parent). V elementu build, který slouží k sestavení projektu, je nastaven plugin *maven–compiler–plugin* pro kompilaci zdrojového kódu, ten je nastaven na Java 1.8. Během vývoje jednotlivých modulů bude rozšiřován o společné závislosti, rozšíření a konfiguraci.

4.4. Spring Web + Thymeleaf POC

V této kapitole je popsána praktická ukázka použití technologií popsaných v předchozí kapitole zabývající se vývojem webové aplikace ve frameworku Spring. Zdrojové kódy jsou k nalezení v kořenovém adresáři repositáře *java–web–framework–poc* ve složce *spring–web–poc.*

4.4.1. Maven konfigurace a struktura projektu

Vzhledem k tomu, že je aplikace vyvíjena s pomocí Maven nástroje, je nutné, aby byla dodržena určitá struktura projektu. Zároveň je zapotřebí označit modul v tagu *packaging* jako *war*. Následující obrázek ukazuje výslednou strukturu projektu.



Obrázek 15 – Struktura Spring POC projektu [autor]

Struktura projektu je rozdělena to tří částí. Všechny tyto části musí být umístěny ve složce *src/main/*, aby byl Maven schopný se zdrojovými kódy pracovat. Nedílnou součástí

každého projektu jsou také testy. Ty se nachází ve složce *src/test/*, která reflektuje strukturu složky se zdrojovými kódy.

První složka se zdrojovými kódy je *java*, která obsahuje veškeré Java třídy projektu. V rámci balíkovací struktury, která má prefix *cz.jbradle.poc.web.spring*, jsou rozděleny podle jejich zaměření. V balíku *app* jsou obsaženy veškeré třídy pro samotnou aplikační část projektu, tedy od servisní vrstvy k datové. Balík *config* pak obsahuje všechny konfigurační třídy včetně inicializační třídy webové aplikace. V balíku *webapp* se nachází kontroléry (v tomto případě jeden) a případně další komponenty potřebné pro správnou funkci servletového kontextu.

Ve složce *resources*, která obsahuje ostatní zdrojové soubory, je v tomto případě umístěn zdrojový balík (Resource Bundle) obsahující lokalizační zprávy, které budou zmíněny v části popisující šablonovací nástroj Thymeleaf. Dále obsahuje soubor obsahující SQL skript pro vytvoření databáze a konfigurační soubor pro logovací nástroj log4j.

Poslední složkou projektu je *webapp*. Ta obsahuje všechny zdroje pro načtení obsahu webové stránky. Nejdůležitější z nich je složka *templates*, která obsahuje šablony pro Thymeleafe. Ve složce *node_modules* jsou dotaženy pomocí npm distribuce zdrojových kódů třetích stran. V *resources* pak mohou být vlastní zdroje (CSS, JS, obrázky…).

V kořenu projektu se nachází soubor *pom.xml*, který slouží jako konfigurační soubor pro Maven. Aby byl Maven schopný získat všechny potřebné knihovny, musí tento soubor obsahovat výpis všech závislostí. Závislosti projektu jsou zapisovány pomocí tagu *dependencies*. Zároveň je možné definovat verze jednotlivých závislostí využitím tagu *properties*. Následující výpis je ukázkou zápisu závislosti v *pom.xml*.

Výpis 26 – Ukázka zápisu závislosti v pom.xml

```
<properties>
	<spring.version>4.2.5.RELEASE</spring.version>
</properties>

<dependencies>
	<dependency>
		<groupId>org.springframework</groupId>
		<artifactId>spring-webmvc</artifactId>
		<version>${spring.version}</version>
	</dependency>
</dependencies>
```

Další tag, jenž umožňuje řízení závislostí je *scope* (rozsah). Ten říká, v jaké fázi sestavení aplikace jsou závislosti potřeba. Pokud není nastaven, bude knihovna nahrána do aplikace a k dispozici po celou dobu sestavení. V tomto projektu jsou využity tři rozsahy viditelnosti knihoven. Prvním je rozsah *test*, který říká mavenu, že knihovna bude k dispozici pouze pro

spuštění testů. Druhým rozsahem je *compile*, který je v tomto projektu použit pro knihovny určené k logování. A třetí je *provided*, který říká, že knihovna bude k dispozici při spuštění aplikace, ale nemá být součástí balíku. Tento rozsah je použit pro knihovnu Java Servlet API, která je součástí webové serveru.

Následující tabulka obsahuje výpis závislostí potřebných pro správný chod aplikace.

Tabulka 1 – Výpis závislostí serverové aplikace

GroupId	ArtifactId	Version	Popis
org.springframework	spring–webmvc	4.2.5.RELEASE	Spring Web MVC
org.springframework	spring–data–jpa	1.10.1.RELEASE	Spring Data JPA
org.thymeleaf	thymeleaf	2.1.4.RELEASE	Jádro Thymeleaf
org.thymeleaf	thymeleaf–spring4	2.1.4.RELEASE	Podpora pro Spring
ma.glasnost.orika	orika–core	1.4.6	Mapovací nástroj
org.hibernate	hibernate–entitymanager	5.0.4.Final	Hibernate pro Spring Data
com.h2database	h2	1.4.190	Embeded DB
javax.servlet	javax.servlet–api	3.1.0	Java EE Servlet API
javax.persistence	persistence–api	1.0.2	Java EE Persistance API
javax.transaction	jta	1.1	Java EE Transactions

Další důležitou konfigurací v pom.xml je nastavení sestavení. Zároveň je možné využít maven rozšíření pro spuštění webového serveru bez vlastní instance serveru. K sestavení distribučního archivu WAR je využit *maven–war–plugin*, a pro spuštění webového serveru *tomcat7–maven–plugin*. Následující výpis je ukázkou takové konfigurace.

Výpis 27 – Konfigurace sestavení v pom.xml

```
<build>
	<plugins>
		<plugin>
			<groupId>org.apache.maven.plugins</groupId>
			<artifactId>maven-war-plugin</artifactId>
			<version>2.1.1</version>
			<configuration>
				<failOnMissingWebXml>false</failOnMissingWebXml>
			</configuration>
		</plugin>
		<plugin>
			<groupId>org.apache.tomcat.maven</groupId>
			<artifactId>tomcat7-maven-plugin</artifactId>
			<version>2.2</version>
			<configuration>
				<port>8080</port>
				<path>/</path>
			</configuration>
		</plugin>
	</plugins>
</build>
```

Projekt je implementován bez použití zaváděcího souboru *web.xml*, je tedy nutné v konfiguraci pluginu vypnout kontrolu na výskyt tohoto souboru. A vzhledem k tomu, že jsou tyto pluginy využívány napříč projekty, je možné konfiguraci umístit do konfigurace parent projektu.

Pomocí následujícího příkazu se spustí sestavení aplikace pomocí Maven.

```
$ mvn clean install
```

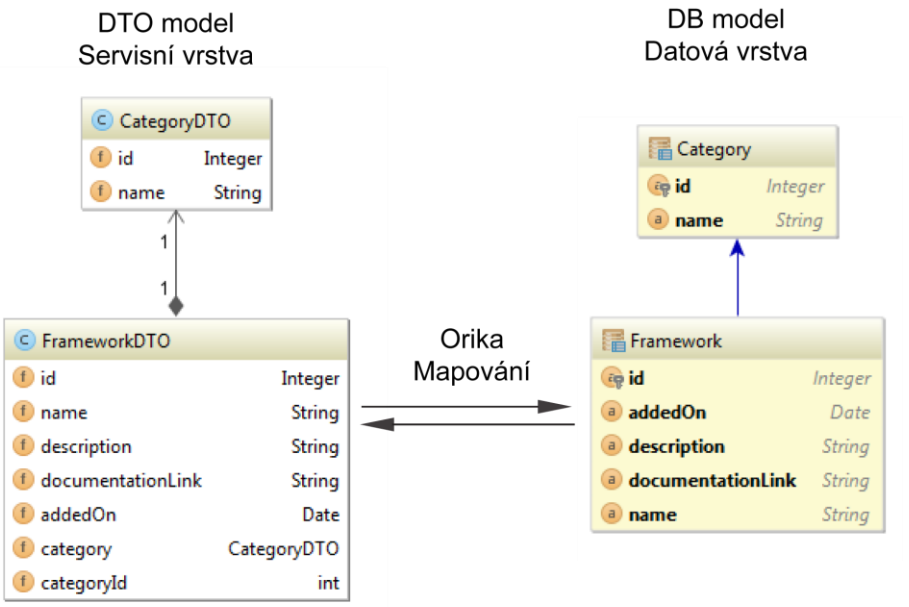
Následně je již možné spustit instanci serveru pomocí příkazu.

```
$ mvn tomcat7:run
```

4.4.2. WEBOVÁ APLIKACE

Při vývoji webové aplikace bylo postupováno podobným způsobem jako v teoretické části. Během vývoje větších aplikací je často naráženo na jeden problém. Entitní objekty modelu jsou z aplikace předávány nezinicializované. Vzhledem k již uzavřené transakci, mohou způsobovat problémy při jejich čtení. Například pro Hibernate je jeden z nejčastěji řešených problémů nezinicializovaná kolekce, která má za následek *LazyInitializationException*. Tento problém je možné řešit několika způsoby. Jak této výjimce předcházet a jak jí řešit je dobře popsáno ve článku [11] .Jedním z nich je nastavení vazební anotace na `FetchType.EAGER`. To ale způsobí, že budou kolekce načítány vždy, i když to není nutné. To může způsobit znatelné snížení rychlosti aplikace. Druhým a pravděpodobně nejvhodnějším způsobem je vytvořit dva modely. První bude reprezentovat databázové entity a druhý entity, které budou k dispozici na rozhraní servisní vrstvy. Tyto nové objekty se pak nazývají DTO („Data Transfer Object“, objekt pro přenos dat). Poté je možné na úrovni servisní vrstvy, kdy je ještě transakce otevřená, využít konverzí mechanizmus, který převede databázové entity na DTO. Pro tento projekt byl vybrán konverzní mechanizmus Orika. Orika se zaměřuje na vytvoření mapovacího nástroje, který bude co nejvíce automatizovaný a rychlý[17].

Jsou tedy vytvořeny dva druhy objektů v modelu, které na sebe 1:1 korespondují. Proto není nutná žádná speciální konfigurace na úrovni mapování a Orika sama převede objekty mezi sebou. Následující diagram ukazuje strukturu objektů modelu.



Obrázek 16 – Spring POC model diagram [autor]

Pro zavedení mapovacího mechanizmu do projektu stačí vytvořit jednu beanu do konfigurační třídy `AppConfig`, která je definovaná pomocí rozhraní `MapperFacade`. Orika obsahuje implementaci takového rozhraní `ConfigurableMapper`, jež je možné dále konfigurovat jejím poděděním. Jelikož je při změně kategorie přenášena pomocí parametru *categoryId* v DTO, je zapotřebí při mapování tuto informaci zpracovat. Všechny ostatní atributy v modelu jsou typem i názvem stejné, a budou mapovány automaticky.

Pro vlastní konfiguraci mapování obsahuje Orika dva hlavní druhy abstraktních tříd. První je `CustomMapper`, která v definici přebírá informace o typech, které je schopný mapovat. Slouží k mapování mezi objekty. Druhým je `CustomConverter`, který slouží pro konverzi mezi datovými typy. Může být použit například pro převod typu `Date` na `String`.

Vzhledem k použití Orika mapovacího nástroje v kombinaci se Spring frameworkem, byla vytvořena speciální beana, které spojuje výhody obou systémů. Hlavní požadavek na tuto beanu je, aby byla schopná sama dohledat všechny vlastní implementace mapperů a konvertorů, a následně je všechny automaticky zaregistrovat do mapovacího kontextu. K tomuto účelu je možné využít dvou rozhraní, která jsou obsažena ve Spring frameworku. Prvním je `ApplicationContextAware`. Toto rozhraní rozšiřuje objekt metodu *setApplicationContext*, která je provolána při vytvoření beany aplikačním kontextem a na vstupu dostává instanci onoho kontextu. Ten je pak nastaven jako atribut beany. Druhým rozhraním je `InitializingBean`, které rozšiřuje objekt o void metodu *afterPropertiesSet*. Ta je provolána na samotném konci inicializace beany, takže je již atribut obsahující aplikační kontext nastaven. V této metodě proběhne doptání aplikačního kontextu o všechny beany typu `Converter` a

Mapper, které jsou poté zaregistrovány do mapovacího kontextu. Následující výpis je výsledná implementace.

Výpis 28 – cz.jbradle.poc.web.spring.app.mapping.ContextAwareMapperBean

```
public class ContextAwareMapperBean extends ConfigurableMapper
        implements ApplicationContextAware, InitializingBean {

    private MapperFactory factory;
    private ApplicationContext applicationContext;

    @Override
    protected void configure(final MapperFactory factory) {
        this.factory = factory;
    }

    @Override
    public void setApplicationContext(
            ApplicationContext applicationContext)
            throws BeansException {
        this.applicationContext = applicationContext;
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        addAllSpringBeans();
    }

    private void addAllSpringBeans() {
        final Map<String, Converter> converters = applicationContext
                        .getBeansOfType(Converter.class);
        converters.values().forEach(this::addConverter);

        final Map<String, Mapper> mappers = applicationContext
                        .getBeansOfType(Mapper.class);
        mappers.values().forEach(this::addMapper);
    }

    public void addConverter(final Converter<?, ?> converter) {
        factory.getConverterFactory().registerConverter(converter);
    }

    public void addMapper(final Mapper<?, ?> mapper) {
        factory.classMap(mapper.getAType(), mapper.getBType())
                .byDefault()
                .customize((Mapper) mapper)
                .register();
    }
}
```

V případě registrace mapperu je provolána metoda *byDefault*. Ta zajistí, aby se stejné atributy namapovaly samy. Pokud by provolána nebyla, bude Orika očekávat, že se mapování provádí kompletně ve vlastní implementaci mapperu. Následující výpis je ukázkou zaregistrování mapperu do aplikačního kontextu.

Výpis 29 – Konfigurace beany Orika mapování

```
@Bean
public MapperFacade mapperFacade() {
    return new ContextAwareMapperBean();
}
```

Nyní je potřeba zajistit, aby byly vlastní implementace konvertorů spravovány aplikačním kontextem. Toho je možné docílit několika způsoby. Nejjednodušším způsobem je označit tyto implementace anotací `@Component` a umístit je tak, aby je byl aplikační kontext schopný zachytit při skenování komponent. V POC projektu je tedy zapotřebí při konverzi nastavit v objektu `Framework` správnou hodnotu `Category` v závislosti na parametru *categoryId* z DTO. Následující výpis je implementací mapperu, který toto zajistí.

Výpis 30 – cz.jbradle.poc.web.spring.app.mapping.CustomFrameworkMapper

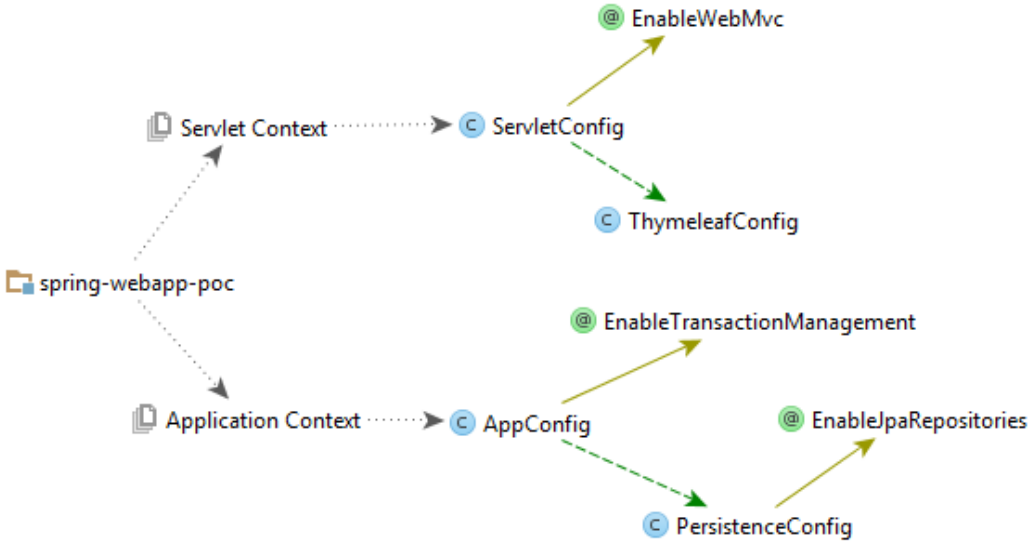
```
@Component
class CustomFrameworkMapper extends CustomMapper<Framework, FrameworkDTO> {

   @Autowired
   private CategoryRepository categoryRepository;

   @Override
   public void mapBtoA(FrameworkDTO frameworkDTO, Framework framework,
                       MappingContext context) {
        framework.setCategory(
                categoryRepository.getOne(frameworkDTO.getCategoryId()));
   }
}
```

Pak je již možné získat mapper v servisní vrstvě pomocí anotace `@Autowired` a začít používat. Následující výpis je ukázkou použití mapování na servisní vrstvě.

Výpis 31 – Příklad použití mapování

```
@Autowired
private MapperFacade mapper;


@Override
public FrameworkDTO getFrameworkById(int id) {
    return mapper.map(frameworkRepository.findOne(id), FrameworkDTO.class);
}
```

Konfigurace webové aplikace je podobná konfiguraci z teorie. Rozdíl v konfiguraci je použití šablonovacího nástroje Thymeleaf, který bude představen v následující kapitole. Obrázek 17 obsahuje závislosti mezi konfiguračními třídami a zlepšuje pohled na strukturu aplikačních kontextů webové aplikace.



Obrázek 17 – Diagram konfigurace webové aplikace [autor]

Pro snadné testování a vývoj je možné využít tzv. embeded data source. Jedná se o databázový systém, který se inicializuje během startu aplikace, a všechna data jsou uložena v operační paměti. Pro tyto účely je využita knihovna *com.h2database:h2*. Součástí balíku Spring Data JPA jsou i podpůrné nástroje pro práci s embeded databází. V tomto případě je možné využít `EmbeddedDatabaseBuilder`, kterému se nastaví typ databáze (v tomto případě H2) a SQL skript pro její inicializaci. Konfigurace data source tímto způsobem je ukázán v následujícím výpisu.

Výpis 32 – Konfigurace DataSource pomocí embeded databáze

```
@Bean
public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder
            .setType(EmbeddedDatabaseType.H2)
            .addScript("sql/create-db.sql")
            .build();
}
```

Během vývoje druhého POC projektu byla veškerá aplikační logika (včetně konfigurace) a její závislosti vyčleněny do separátního modulu, který se nazývá *spring-app-poc*. To bylo provedeno z důvodu, aby napříč projekty nevznikal duplicitní kód. V balíku na classpath se jedná o cestu *cz.jbradle.poc.web.spring.app*. Při pohledu do pom.xml jednotlivých modulů je poté možné dobře vidět rozdělení závislostí aplikační a servletové části projektu.

4.4.3. THYMELEAF VIEW

Jak již bylo zmíněno v teoretické části o Spring webových aplikacích, musí být do aplikačního kontextu zaveden systém pro generování pohledu. Popsán byl rozsáhle používaný přístup pomocí JSP šablon. Druhou méně známou implementací pohledu je projekt Thymeleaf[20]. V dnešní době se však stává velmi oblíbeným nástrojem. Jedná se o šablonovací systém schopný generovat HTML, XML, JavaScript, CSS nebo jenom obyčejný text. Nejčastěji je používán právě v práci s pohledy, ale je schopný zpracovávat i mnoho dalších formátů. Například může sloužit pro generování PDF dokumentů.

Hlavním cílem projektu, je umožnit vytváření čistých, dobře formátovaných a přehledných šablon. Na rozdíl od JSP se jedná o zcela neinvazivní přístup, kdy jsou obohacovány HTML tagy o nové atributy (mapovací metadata), pomocí kterých pak šablonovací nástroj doplní potřebná data. Praktické využití je takové, že pokud designer vytvoří šablonu pro webovou aplikaci pomocí čistého HTML jazyka, je tato šablona stále plně funkční i po obohacení o mapovací metadata. Takže je možné, aby v případě požadavku na změnu mohla být úprava provedena na již používané šabloně, bez nutnosti startování aplikačního serveru. Což u JSP není možné, případně je to velmi nepohodlné. V následujícím výpisu je ukázka rozdílu mezi JSP a Thymeleaf při tvorbě šablony.

Výpis 33 – WEB–INF/jsp/example.jsp + Thymeleaf podoba

```
<!-- JSP -->
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<body>
<c:forEach items="${exampleValues}" var="value">
    <h1>${value}</h1>
</c:forEach>
</body>
</html>

<!-- Thymeleaf -->
<html xmlns=http://www.w3.org/1999/xhtml xmlns:th="http://www.thymeleaf.org">
<body>
<th:block th:each="value : ${exampleValues}">
    <h1 th:text="${value}">Template value</h1>
</th:block>
</body>
</html>
```

[20] http://www.thymeleaf.org

Z přechozího výpisu je možné vidět, jakým způsobem funguje vytváření šablony. Do šablony je zapotřebí doplnit namespace „*http://www.thymeleaf.org*“, který je většinou označován zkratkou *th*.

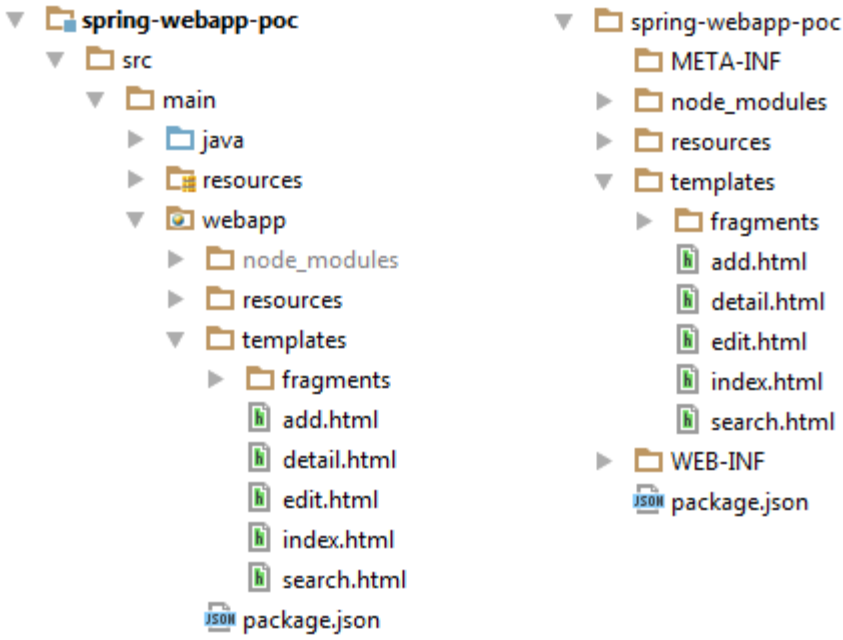
Pro zavedení do projektu je nutné vytvořit konfiguraci, ve které se nastaví šablonovací systém tak, aby spolupracoval s komponentami Spring MVC. Součástí Thymeleaf je modul pro integraci s právě tímto modulem, jenž konfiguraci velmi usnadní.

Následující výpis je ukázkou konfigurace Thymeleaf.

Výpis 34 – cz.jbradle.poc.web.spring.config.ThymeleafConfig

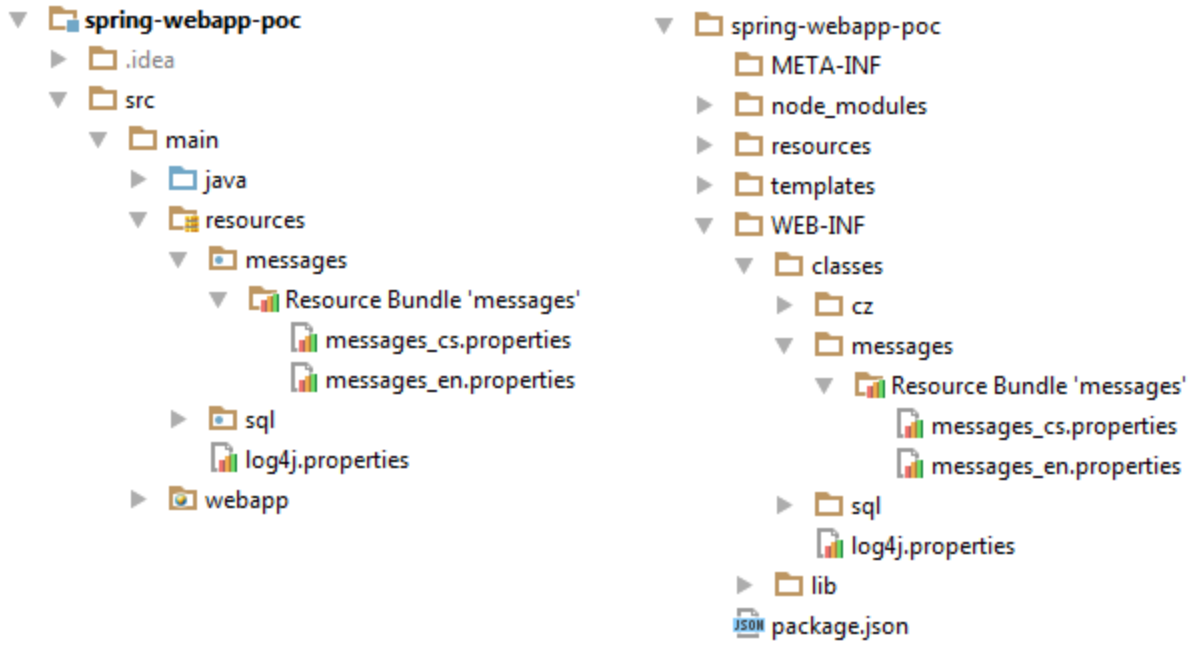
```
@Configuration
class ThymeleafConfig {

    @Bean
    public ServletContextTemplateResolver templateResolver() {
        ServletContextTemplateResolver resolver =
                        new ServletContextTemplateResolver();
        resolver.setPrefix("/templates/");
        resolver.setSuffix(".html");
        resolver.setCharacterEncoding("UTF-8");
        resolver.setTemplateMode("HTML5");
        resolver.setCacheable(false);
        resolver.setOrder(1);
        return resolver;
    }

    @Bean
    public SpringTemplateEngine templateEngine() {
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver());
        return templateEngine;
    }

    @Bean
    public ThymeleafViewResolver thymeleafViewResolver() {
        ThymeleafViewResolver resolver = new ThymeleafViewResolver();
        resolver.setTemplateEngine(templateEngine());
        resolver.setCharacterEncoding("UTF-8");
        return resolver;
    }

    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource messageSource =
                        new ResourceBundleMessageSource();
        messageSource.setBasename("messages/messages");
        messageSource.setDefaultEncoding("UTF-8");
        return messageSource;
    }
}
```

Jak již bylo zmíněno v kapitole 3.1.4.3, pro generování výstupu je nutné registrovat beanu, která implementuje rozhraní `ViewResolver`. Implementací tohoto rozhraní v Thymeleaf je `ThymeleafViewResolver`, kterému je potřeba nastavit atribut *templateEngine*, Ten musí být typu `SpringTemplateEngine`. Jedná se přímo o implementaci, takže ji není možné generalizovat. Tato implementace potřebuje nastavit atribut *templateResolver*, který má na starosti vyhledání a načtení šablony do systému, a je implementací rozhraní `ITemplateResolver`. Jelikož se jedná o webovou aplikaci, je možné využít `ServletContextTemplateResolver`, který bude vyhledávat pomocí servlet kontextu. Pro správné vyhledávání je nutné nastavit cestu od hlavní složky webové aplikace. Toho je docíleno pomocí atributů *prefix* a *suffix*, kde prefix říká cestu k šablonám a sufix jejich příponu. Dále je dobré nastavit kódování a mód, ve kterém jsou šablony vytvářeny. Následující obrázek ukazuje umístění šablon v projektu a ve výsledném WAR balíku.



Obrázek 18 – Umístění šablon Projekt vs. WAR [autor]

Čtvrtá beana je typu `MessageSource`, která bude sloužit pro získání hodnot statických zpráv. To umožní jednoduše lokalizovat aplikaci. Všechny lokalizační soubory jsou seskupeny do imaginárního balíku (Resource Bundle) *messages*, který je ve složce *messages* na classpath. Takovým imaginárním balíkem je myšleno to, že všechny mají název s prefixem *messages_* a příponou *.properties*, kořenem názvu je pak lokalizační zkratka. V rámci projektu je vytvořena lokalizace pro český a anglický jazyk. Následující obrázek ukazuje umístění lokalizačních souborů v projektu a ve WAR balíku.



Obrázek 19 – Struktura balíku zpráv Projekt vs. WAR [autor]

V tuto chvíli je již aplikace plně nakonfigurovaná, a je možné začít vytvářet šablony.

V následující části jsou popsány základní koncepty pro vytváření Thymeleaf šablon.

Atributy

Thymeleaf namespace obsahuje velké množství rozšiřujících atributů, pomocí kterých jsou generována výstupní data. Většina atributů jsou pouze odkaz na původní HTML atributy. Například v tagu pro odkaz je možné použít *th:href*. Obsah tohoto atributu bude po zpracování doplněn do atributu *href*. Dále obsahuje atributy, které slouží k řízení sestavení výstupních dat. Mezi nejdůležitější atributy patří následující výpis. U každého atributu je uveden zápis a případně výsledek po vygenerování.

th:text – doplní textovou hodnotu do HTML tagu.

```
<h1 th:text="Value 2">Value 1</h1>          → <h1>Value 2</h1>
```

th:with – nastavuje lokální hodnotu proměnné pro další zpracování. Používá se například při práci s fragmenty.

```
<div th:with="number=2">
    <h1 th:text="'Value '+ ${number}">Value 1</h1>   →   <div>
</div>                                                       <h1>Value 2</h1>
                                                         </div>
```

th:block – je používán v případě, že nelze použít jiný HTML tag

```
<th:block th:with="number=2">
    <h1 th:text="'Value '+ ${number}">Value 1</h1>
</th:block>
```

→

```
<h1>Value 2</h1>
```

th:each – je používán pro iterování skrze kolekce. V příkladu je předpokládáno, že existuje kolekce nazvaná *numbers*, která obsahuje hodnoty 1 a 2,

```
<th:block th:each="number : ${numbers}">
    <h1 th:text="Value + ${number}">Value 1</h1>
</th:block>
```

→

```
<h1>Value 1</h1>
<h1>Value 2</h1>
```

th:fragment – označuje HTML tag jako fragment, který může být znovu použit na jiném místě šablony. Všechny fragmenty musí být vyčleněny do zvláštní šablony, která může obsahovat více fragmentů. V tomto případě je fragment umístěn v šabloně *fragments.html*.

```
<div th:fragment="exampleFragment">
    <h1 th:text="Value + ${number}">Value 1</h1>
</div>
```

th:include – přepíše tag fragmentem a sloučí th atributy z obou tagů. Následující příklad využívá fragment z předchozího příkladu.

```
<div th:include="fragments :: exampleFragment"
th:with="number=2">
    <h1>Value 1</h1>
</div>
```

→

```
<div>
    <h1>Value 2</h1>
</div>
```

th:replace – kompletně přepíše tag fragmentem. Při přepsání je odstraněn atribut with, a výsledná šablona tedy nemá nastavenou proměnou *number*. Proto je výsledkem doplnění null hodnoty.

```
<div th:replace="fragments :: exampleFragment"
th:with="number=2">
    <h1>Value 1</h1>
</div>
```

→

```
<div>
    <h1>Value null</h1>
</div>
```

Proměnné a výrazy

Aby šablonovací nástroj mohl správně doplňovat hodnoty do atributů, zpracovává výrazy zadané v hodnotách atributů pomocí vlastní syntaxe. Tato syntaxe se řídí běžnými standarty zápisu a rozšiřuje je o některé další výrazy přinášející více funkcionality. Následující seznam popisuje základní výrazy, které jsou možné použít pro generování obsahu.

- Výrazy:
  - Proměnná: `${…}` Z modelu nebo lokálních proměnných.
  - Zpráva: `#{…}` Ze zdroje zpráv (umožňuje lokalizaci).
  - Odkaz: `@{…}` Generuje podle URL, na které běží aplikace.
  - Výběr: `*{…}` Vybírá atribut objektu (Použit ve formulářích)
- Literály:
  - Textové `'text'`, `'další text'`,…
  - Číselné `0`, `34`, `3.0`, `12.3`,…
  - Logické `true`, `false`
  - Null `null`
  - Tokeny `text, token,…` Stejné jako textové, ale nedovolují bílé znaky
- Textové operace:
  - Spojení textu `+`
  - Substituce `|Název je ${nazev}|`
- Numerické operace:
  - Binární operátory `+`, `–`, `*`, `/`, `%`
  - Negace čísla `–`
- Logické operátory
  - Logické operátory `and`, `or`
  - Logická negace `!`, `not`
- Porovnání
  - Porovnání `>`, `<`, `>=`, `<=` (`gt`, `lt`, `ge`, `le`)
  - Rovnost `==`, `!=` (`eq`, `ne`)
- Podmíněnost
  - Když–tak `(if) ? (then)`
  - Když–tak–jinak `(if) ? (then) : (else)`
  - Kontrola existence `(value) ?: (defaultvalue)`

Všechny vypsané výrazy je možné kombinovat a zanořovat, čímž se může malá část logiky přenést na stranu šablony. To by se však mělo používat co nejméně, aby nevznikali nepřehledné šablony, které budě těžké číst a analyzovat.

V kapitole 4.2 byla vytvořena šablona POC projektu. Pro příklad použití šablonovacího nástroje Thymeleaf je v následujícím výpisu ukázka implementace tabulky z obrázku 11.

Výpis 35 – Tabulka z templates/fragments/example.html

```
<table class="table table-hover" th:fragment="table">
    <thead>
        <tr>
            <th th:text="#{framework.table.name}">Name</th>
            <th th:text="#{framework.table.category}">Category</th>
            <th th:text="#{framework.table.docs}">Docs</th>
            <th th:text="#{framework.table.actions}">Actions</th>
        </tr>
    </thead>
    <tbody>
      <tr th:each="framework : ${frameworks}">
            <td th:text="${framework.name}">Spring Web MVC</td>
            <td th:text="${framework.category.name}">Web Layer</td>
            <td>
                <a href="http://docs.spring.io"
                   th:href="${framework.documentationLink}">
                    <i class="fa fa-arrow-right fa-2x"></i>
                </a>
            </td>
            <td>
                <a href="../detail.html" title="Detail"
                th:href="@{/detail/{id}(id=${framework.id})}">
                    <i class="fa fa-folder-open fa-2x"></i>
                </a>
                <a href="#" title="Edit"
                th:href="@{/edit/{id}(id=${framework.id})}">
                    <i class="fa fa-edit fa-2x"></i>
                </a>
                <a href="#" title="Remove"
                th:href="@{/delete(id=${framework.id})}">
                    <i class="fa fa-remove fa-2x"></i>
                </a>
            </td>
        </tr>
    </tbody>
</table>
```

Celá tabulka je označena jako fragment table. Tu je pak možné použít v hlavních šablonách. V projektu byl tento fragment použit v šabloně *index.htm*, která zobrazuje výpis všech frameworků, a v šabloně *search.html*, která slouží pro vyhledávání.

V hlavičce tabulky je příklad použití výrazu, který doplní text ze zdroje zpráv (v tomto případě z lokalizačního souboru *messages_cs.properties*). Do nadpisů sloupců jsou doplněny nové hodnoty, které jsou lokalizovány.

Dále je v těle tabulky využit atribut *each*, který iteruje skrze kolekci *frameworks* z modelu získaného v kontroléru. K právě vybranému objektu z iterované kolekce je následně možné

přistupovat ve vnitřní části tagů. V prvních dvou případech jsou vypsány atributy do těla *td* tagu. Následuje ukázka naplnění hodnoty do *href* parametru tagu odkazu.

Poslední sloupec tabulky obsahuje operace, které jsou možné s objekty provádět. Všechny operace využívají výraz pro práci s odkazy. První dvě operace pro zobrazení detailu a editaci objektu využívají způsob předání identifikátoru pomocí proměnné v cestě adresy (Spring – `@PathVariable`). Operace pro smazání objektu předává identifikátor pomocí proměnné požadavku (Spring – `@RequestParam`). Následující obrázek ukazuje výsledný pohled na vygenerovanou tabulku z šablony.

Název	Kategorie	Dokumentace	Akce
Spring Beans	Service Layer		
Spring Context	Service Layer		
Hibernate	Data Layer		
Spring Web MVC	Web Layer		

Obrázek 20 – Výsledná tabulka POC projektu [autor]

Druhou větší komponentou je formulář pro vytvoření a editaci objektu framework. Tu je možné nalézt opět v šabloně s fragmenty *templates/fragments/example.html* s názvem *framework–edit*. Pro její velkost a nepřehlednost však není zahrnutá ukázka použití v tomto dokumentu.

Více příkladů a možností využití nástroje je možné získat na stránkách s dokumentací projektu [18], ze které čerpal i autor, při psaní této kapitoly.

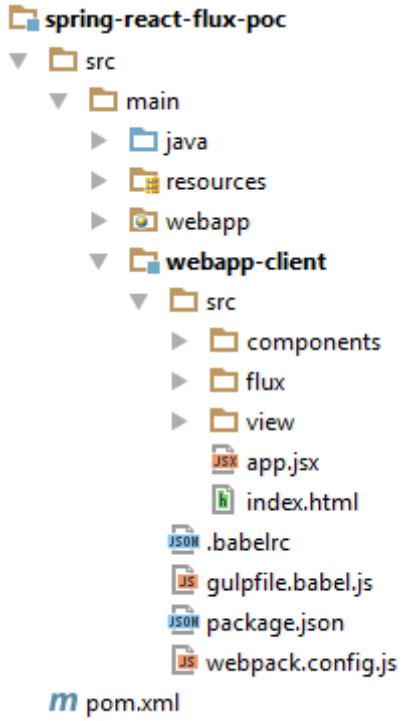
4.5. REACT, REFLUX + SPRING POC

V této kapitole je rozebrána praktická ukázka použití technologií popsaných v předchozí kapitole zabývající se vývojem klientské webové aplikace ve frameworcích React a Flux. Zdrojové kódy jsou k nalezení v kořenovém adresáři repositáře *java–web–framework–poc* ve složce *spring–react–flux–poc.* Tento modul je závislý stejně jako předchozí POC na aplikačním modulu *spring–app–poc*.

4.5.1. KLIENTSKÁ APLIKACE

Na začátek je potřeba upozornit na celkové zkušenosti autora s vývojem klientských aplikací. Jedná se o první rozsáhlejší vývoj takové aplikace a první blízké setkání s React a Flux frameworky. Jak již bylo zmíněno v teoretické části, je celý tento projekt psaný v ES6. A to včetně konfigurace podpůrných nástrojů. Modul samotné klientské aplikace je umístěn uvnitř projektu na cestě *src/main/webapp–client*, aby bylo možné jednoduše distribuovat zkompilovanou aplikaci do serverové webové aplikace.

Prvním krokem při vývoji klientské aplikace je vytvoření prostředí, které umožní jednoduchý a rychlý vývoj. Následující obrázek ukazuje strukturu projektu.



Obrázek 21 – Struktura projektu React + Flux POC [autor]

Ze struktury projektu je vidět, že pro automatizaci sestavení byl stejně jako při vytváření šablony POC projektů v kapitole 4.2 použit nástroj Gulp. Tentokrát má však na starosti mnohem

více úkolů. Konfigurační soubor pro Gulp je psán jako celý projekt v ES6. Aby mohli být zdrojové kódy psané v ES6, je potřeba dodat do projektu kompilátor, který umožní převod z ES6 do čistého JS kódu. K tomuto účelu byl zvolen projekt Babel[21]. Jelikož se jedná pouze o nástroj, který je použit při kompilaci kódu, jsou závislosti v souboru *package.json* vedené jako *devDependencies*. Babel je také použit jako kompilátor pro JSX.

Aby mohl být konfigurační souboru pro Gulp psaný v ES6, je nutné v kořenovém adresáři modulu mít umístěn soubor *.babelrc*, který mu říká, jaký kompilátor při čtení konfigurace použít. Následující výpis je ukázkou onoho souboru.

Výpis 36 – Ukázka souboru .babelrc

```
{
  "presets": ["es2015"]
}
```

Následující seznam je souhrn použitých nástrojů pro usnadnění vývoje a sestavení aplikace. Dále obsahuje i závislosti výsledné aplikace.

Tabulka 2 – Výpis závislostí klientské aplikace

Název modulu	Popis
Vývojové závislosti	
`babel-core`	Jádro kompilátoru Babel
`babel-loader`	Zaváděcí modul pro Babel
`babel-preset-es2015`	ES6 kompilátor
`babel-preset-react`	JSX kompilátor
`browser-sync`	Automatická obnova aplikace v prohlížeči
`del`	Gulp rozšíření pro mazání souborů
`gulp`	Jádro automatizačního nástroje Gulp
`gulp-if`	Gulp rozšíření pro tvorbu podmínek
`gulp-load-plugins`	Gulp rozšíření pro usnadnění konfigurace
`gulp-size`	Gulp rozšíření pro výpis velikosti souborů
`gulp-uglify`	Gulp rozšíření pro kompresy výsledné aplikace
`gulp-util`	Gulp rozšíření o pomocné nástroje
`gulp-watch`	Gulp rozšíření pro kontrolu změn v souborech
`gulp-webpack`	Gulp rozšíření pro spolupráci s modulem Webpack
`orchestrator`	Gulp rozšíření o způsoby řízení úkolů
`webpack`	Nástroj pro sestavení aplikace
`yargs`	Podpora práce s argumenty z příkazové řádky

[21] https://babeljs.io

Aplikační závislosti	
`history`	Závislost pro react–router
`moment`	Závislost pro react–time (chyba balíku)
`promise-polyfill`	Knihovna vzoru Promise (asynchronní zpracování)
`react`	Jádro frameworku React
`react-dom`	React komponenta pro vykreslování do DOM
`react-router`	React komponenta pro směrování aplikace
`react-time`	React komponenta pro práci s časem
`reflux`	Implementace Flux návrhového vzoru
`superagent`	Klient pro REST komunikaci se serverem

Nyní je již možné začít konfigurovat sestavování aplikace. Jako každá webová aplikace i tato začíná vytvořením HTML souboru, který bude vstupní branou do aplikace. Zde je tímto souborem *src/index.html*, jenž načítá základní knihovny pro Bootstrap, aby je bylo možné využít v rámci komponent. V dnešní době existuje velké množství způsobů, jak do takovéto aplikace dodat styly, ale problematika není řešena v rámci této práce, tudíž jsou načteny takto z externího repositáře. Hlavní částí tohoto vstupního HTML souboru je div a odkaz na zkompilovanou aplikaci, jak ukazuje následující výpis.

Výpis 37 – src/index.html

```
<div id="app" class="container"></div>

<script src="js/app.js" th:src="@{/js/app.js}"></script>
```

Na tento div se pak odkáže aplikace a začne do něho generovat obsah. Jelikož je aplikace ve výsledku dotahována pomocí Thymeleaf, je nutné aby *index.xml* obsahoval *th* atributy.

Pro distribuci *index.html* je použit jednoduchý úkol, který pouze soubor vezme, do příkazové řádky vypíše jeho velikost, a nakopíruje ho cílového adresáře *webapp*. Následující výpis je ukázkou takovéto úkolu.

Výpis 38 – gulpfile.babel.js – HTML úkol

```
var dist = '../webapp/';
var src = 'src/';

gulp.task('html', () => {
    return gulp.src(src + 'index.html')
        .pipe(gulp.dest(dist))
        .pipe(size({title: 'html'}));
});
```

Následuje úkol pro sestavení samotné aplikace. Pro řešení závislostí je použit projekt Webpack[19], který umožňuje jednoduché sestavení aplikace. Webpack je rozsáhlé řešení pro zprávu sestavení a samotného vývoje klientských aplikací. V tomto případě je použit pouze pro jeho nejpodstatnější operaci, a tou je projít kompletně zdrojové kódy, a všechny moduly, které jsou napříč aplikací použity, a zabalit je do jednoho souboru. Konfigurace pro webpack je umístěna v kořenovém adresáři v souboru *webpack.config.js*. Následující výpis je ukázkou konfigurace.

Výpis 39 – webpack.config.js

```
export default (production) => ({
    entry: './src/app.jsx',
    output: {
        path: __dirname,
        filename: 'app.js'
    },
    debug: !production,
    module: {
        loaders: [{
            test: /\.jsx?$/,
            exclude: /node_modules/,
            loader: 'babel',
            query: {
                presets: ['react', 'es2015']
            }
        }]
    },
    devtool: !production ? 'eval-source-map' : ''
});
```

V konfiguraci je definován vstupní soubor, kterým je *src/app.jsx*. Dále konfigurace říká webpacku, aby použil na všechny JSX soubory kompilátory pro ES6 (ES2015) a JSX (React). Na vstupu konfigurace je příznak, zda se jedná o produkční sestavení. Pokud ne, webpack použije vývojový nástroj *eval–source–map*, který provede načtení (vyhodnocení) jednotlivých modulů ve vedlejším prostředí prohlížeče, a tím umožní procházet přehledně originální zdrojové kódy přímo z prohlížeče. Pokud bude produkční sestavení zapnuté, budou všechny moduly sestaveny bez formátování v jednom nečitelném souboru. Načtení webpack konfigurace v gulp konfiguraci je provedeno viz následující výpis.

Výpis 40 – Načtení webpack konfigurace – gulpfile.babel.js

```
import webpackConfig from "./webpack.config";
var production = yargs.argv.production;
var config = webpackConfig(production);
```

V rámci konfigurace je pro snadné použití modulu webpack použita instance modulu *gulp–load–plugins*, která je schovaná pod znakem dolaru ($). Úkol pro samotné sestavení aplikace je znázorněn v následujícím výpisu.

Výpis 41 – Sestavení JS aplikace – gulpfile.babel.js

```
gulp.task('app', () => {
    return gulp.src(config.entry)
        .pipe($.webpack(config))
        .pipe(gulpif(production, uglify()))
        .pipe(gulp.dest(dist + '/js'))
        .pipe(size({title: 'app'}));
});
```

Tento úkol tedy vezme vstupní JSX souboru *src/app.jsx*. Následně jeho obsah zpracuje pomocí modulu webpack s načtenou konfigurací. Pokud je zapnuté produkční sestavení, zpracuje výstup z webpacku modulem *uglify*, který minimalizuje velikost zdrojových kódů. Což například u tohoto projektu znamená zmenšení z 1.5MB na 0,5MB. Na závěr nakopíruje výsledný soubor do složky *webapp/js*. Tím vznikla finálně sestavená klientská aplikace.

Takto připravené prostředí ale ještě není ideální pro samotný vývoj. Aby byla práce co nejjednodušší a nejefektivnější, obsahuje konfigurace gulp ještě další úkoly. Následující výpis je ukázkou zbylých úkolů, které jsou následně popsány.

Výpis 42 – Úkoly pro automatické sestavování – gulpfile.babel.js

```
gulp.task('watch', () => {
    gulp.watch(src + 'index.html', ['html']);
    gulp.watch(src + '**/*.jsx', ['app']);
});

gulp.task('browser-sync', () => {
    browserSync({
        files: dist + 'js/app.js',
        server: {
            baseDir: '../webapp/'
        }
    });
});

gulp.task('default', ['html', 'app'], () => {
    gulp.start(['browser-sync', 'watch']);
});

gulp.task('clean', (callBack) => {
    return del([dist + 'js/**/*.*', dist + '*.html'],
        {force: true}, callBack);
});

gulp.task('build', ['clean'], () => {
    gulp.start(['html', 'app']);
});
```

- **watch –** kontroluje HTML a JSX soubory, a v případě změny spustí jejich sestavení.
- **browser–sync –** hlídá změny v sestavené aplikaci, a v případě změny aktualizuje prohlížeč.
- **default** – spustí sestavení aplikace. Počká na dokončení úkolů pro sestavení a následně spustí úkoly *watch* a *browser–sync*.
- **clean** – smaže všechny soubory sestavené aplikace.
- **build** – spustí *clean* úkol. Počká na dokončení a následně spustí nové sestavení aplikace.

Pro první sestavení aplikace je nutné nainstalovat všechny moduly. Toho je docíleno následujícími příkazy. V případě, že se jedná o produkční sestavení, je možné přidat nepovinný parametr *production.*

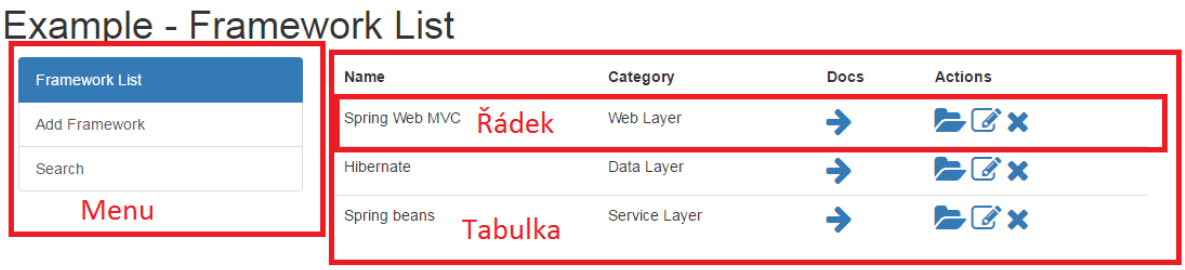
```
$ npm install
$ gulp buil —production
```

Pro vývoj pak již stačí spustit úkol *default* pomocí následujícího příkazu.

```
$ gulp
```

V tuto chvíli je již prostředí připravené k samotnému vývoji.

Prvním krokem při implementaci klientské webové aplikace je definování jednotlivých komponent. Nejprve je dobré určit jasné celky, které bude možné znovu použít v dalších místech aplikace. Nejsnazší způsob určení komponent je ze šablon, které vznikly v rámci kapitoly 4.2. Následující obrázek je ukázkou definice komponent z šablony pro zobrazení seznamu frameworků.



Obrázek 22 – Identifikace možných React komponent [autor]

Během analýzy šablon bylo identifikováno 5 základních komponent. První tři jsou znázorněny na předchozím obrázku. Jedná se o komponentu pro zobrazení menu a tabulky frameworků, která obsahuje kolekci komponent reprezentujících řádek tabulky. Další dvě komponenty jsou formulář pro editaci záznamu a pro jeho zobrazení. Těchto 5 komponent však není maximální počet komponent, které by bylo možné definovat, ale postačuje k názorné ukázce požití frameworku. Všechny komponenty jsou umístěny ve složce *src/components*.

Jak již bylo zmíněno, vstupním souborem do aplikace je nyní src/app.jsx. Tam bude první zavedení aplikace na stránku. Ve frameworku React je první vykreslení na stránku možné provést následující funkcí.

Výpis 43 – Vykreslení první komponenty

```
ReactDOM.render(component, document.getElementById('app'));
```

Tato funkce přebírá jako první argument instanci kořenové React komponenty a jako druhý element v DOMu, do kterého bude vykreslena. V tomto případě se jedná o tag *div* s identifikátorem *app* ze souboru *index.html*.

Jako kořenová komponenta je použita komponenta knihovny react–router [22]. Ta umožňuje přebírání a řízení aplikace v závislosti na aktuální adrese. Součástí knihovny je několik základní komponent, které slouží k definici chování aplikace. Hlavní komponentou je směrovač (`Router`). Ten slouží jako obálka pro jednotlivé směrovací komponenty, které se nazývají cesty (`Route`). Každá z cest přebírá jako atribut *path* hodnotu z adresy prohlížeče a atribut *component*, který obsahuje komponentu, která se má na dané cestě zobrazit. Cesty je možné do sebe vnořovat, a tak může vzniknout konfigurace zobrazování pro celou aplikaci

[22] https://github.com/reactjs/react-route

(nebo jenom její část). Vzhledem k tomu, že jsou komponenty zobrazovány na celou stránku, je možné je pojmenovat jako pohled (view). Všechny pohledy jsou umístěny ve složce *src/view*. Každá komponenta nadřazené cesty má pak k dispozici aktivní komponentu podřízené cesty přístupnou v parametrech (*props.children*). Následující výpis je kompletní souboru app.jsx, který je jednoduchou ukázkou konfigurace pro React–route.

Výpis 44 – src/app.jsx

```
import React from "react";
import ReactDOM from "react-dom";
import {Router, Route, IndexRoute, hashHistory} from "react-router";
import Main from "./view/main.jsx";
import Home from "./view/home.jsx";
import Add from "./view/add.jsx";
import Search from "./view/search.jsx";

const routes = (
    <Router history={hashHistory}>
        <Route path='/' component={ Main }>
            <IndexRoute component={ Home }/>
            <Route path='home' component={ Home } />
            <Route path='add' component={ Add } />
            <Route path='search' component={ Search } />
        </Route>
    </Router>
);

ReactDOM.render(routes, document.getElementById('app'));
```

Z předchozího výpisu je možné vidět složení pohledů, které budou představovat budoucí aplikaci. Takto nakonfigurovaná směrovací komponenta bude na kořenové adrese (/) vykreslovat komponentu `Main`, která obsahuje pouze titulek a podle další úrovně adresy používá pohledy `Home`, `Add` a `Search`. Každý z pohledů bude považován za samostatnou část aplikace, která si udržuje vlastní stav.

Než bude popsán příklad pohledu, je nutné seznámit se s použitou implementací návrhového vzoru Flux. Pro POC projekt byla použita knihovna RefluxJS[23], která vychází právě z onoho návrhového vzoru. Knihovna byla přepsána z implementace Facebook Flux tak, aby byla vývojově přívětivější, například byl odstraněn dispečer.

[23] https://github.com/reflux/refluxjs

Každá akce si obstarává funkcionalitu dispečeru sama. Knihovna nabízí škálu funkcí, které umožňují provádět velké množství operací. V základu ale stačí pár poznatků.

Reflux obsahuje tři základní komponenty, které vychází ze vzoru Flux. Všechny tyto komponenty mohou mít nastavené funkce, které budou poslouchat. První komponentou je Sklad (Store), který naslouchá akcím. V případě, že je v jakékoliv části vyvolána akce, sklad na ni nějakým způsobem zareaguje. Definování akcí je proti Facebook Flux o poznání jednodušší [20]. Každá akce může být rozdělena na sub akce, které mohou být vyvolané například z kontextu pohledu (ale také z jakéhokoliv jiného místa). Následující výpis je ukázkou definování akce s potomky a příklad jejich použití.

Výpis 45 – src/flux/action/frameworkAction.jsx

```
const FrameworkActions = Reflux.createActions({
    'getFrameworks': {children: ['completed', 'failed']},
    'searchFrameworks': {children: ['completed', 'failed']},
    'showEdit': {children: ['completed', 'failed']},
    'showDetail': {},
    'deleteFramework': {children: ['completed', 'failed']},
    'saveFramework': {children: ['completed', 'failed']}
});

FrameworkActions.getFrameworks.listen(function () {

    get(REST_URI + 'getAllFrameworks')
        .then((frameworks) => {
            this.completed(frameworks);
        }).catch((err) => {
        this.failed(err);
    });

});
```

V první části jsou pomocí reflux metody *createActions* interně vytvořeny objekty akcí, které je možné v aplikaci vyvolat. Následně je jedné z akcí přidělena funkce, na kterou bude naslouchat. Tato funkce provede doptání serveru o data. Metoda *get* vrací `Promise`, která v případě úspěšného provolání http GET požadavku na server vrátí kolekci frameworků. V opačném případě vrátí chybu. Pokud je provolání úspěšné, je vyvolána sub akce *completed,* jinak *failed*.

Následně je možné definovat Sklad, který bude na tyto akce reagovat. Reflux opět umožňuje snadnou definici takového skladu. Při vytvoření je nejprve řečeno, jakým akcím bude sklad naslouchat. Pak je již možné definovat metody, které budou provolány v případě zjištění akce. Tyto metody jsou pojmenovány následujícím způsobem. V případě, že je provolána hlavní akce, bude se metoda jmenovat *onNázevAkce*. Například pro akci *getFrameworks* z předchozího výpisu se metoda bude jmenovat *onGetFrameworks*. Pro sub akce se pak metody pojmenovávají

jako *NázevAkce+NázevSubAkce*. V tomto případě tedy *getFrameworksCompleted* a *getFrameworksFailed*. Následující výpis je ukázkou skladu, který bude reagovat na akci `FrameworkAction`.*getFrameworks()* a její sub akce.

Výpis 46 – src/flux/store/frameworkStore.jsx

```
let FrameworkStore = Reflux.createStore({
    listenables: FrameworkAction,

    init() {
        this.frameworks = [];
    },

    onGetFrameworks() {
        this.trigger({
            frameworks: this.frameworks,
            showDetail: false
        });
    },

    getFrameworksCompleted(frameworks) {
        this.frameworks = frameworks;

        this.trigger({
            frameworks: this.frameworks,
            showDetail: false
        });
    },

    getFrameworksFailed(error) {
        this.trigger({
            error: error
        });
    }
});
```

Jak je vidět z předchozího výpisu, je možné definovat například i metodu *init*, která slouží k načtení počátečního stavu před vytvořením skladu. Každý sklad vystavuje metodu *trigger*. V případě, že je tato metoda provolána, provede sklad pro všechny pohledové komponenty, které mu naslouchají, provolání metody, kterou při registraci obdržel.

Nyní již k samotným React pohledovým komponentám. Jak bylo zmíněno, pohledové komponenty budou v tomto případě samostatné a budou si udržovat vlastní stav. Nastavení stavu je provedeno v konstruktoru komponenty. Následně je možné využít pomocné metody React komponent pro zaregistrování ke skladu a případné odhlášení. V každé React komponentě je možné definovat metodu *componentDidMount*, jež je provolána po načtení komponenty a metodu *componentWillUnmount*, jež je provolána před jejím odebráním. Zaregistrování ke skladu je tedy nutné provést po načtení a odhlášení před odebráním komponenty. Zaregistrování skladu se provádí pomocí metody *listen*, která jako parametr přebírá referenci

na metodu, která bude provolána v případě aktualizace dat ze skladu (zavolání metody *trigger*). Následující výpis je ukázkou komponenty `Home`, která se zaregistruje a zobrazí menu a tabulku frameworků.

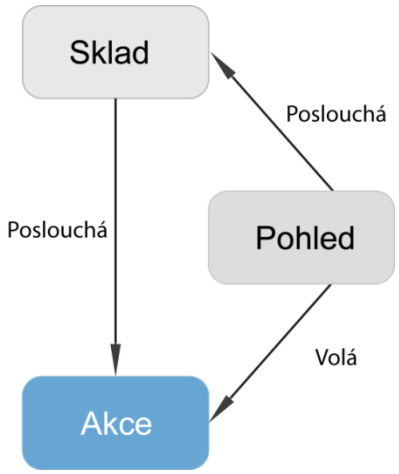
Výpis 47 – src/view/home.jsx

```
export default class Home extends React.Component {

    constructor(props) {
        super(props);
        this.state = {
            frameworks: [],
            showDetail: false,
            showEdit: false
        };
    }

    componentDidMount() {
        this.unsubscribe =
            FrameworkStore.listen(this.onStatusChange.bind(this));
        FrameworkAction.getFrameworks();
    }

    componentWillUnmount() {
        this.unsubscribe();
    }

    onStatusChange(state) {
        this.setState(state);
    }

    render() {
        return (
            <div className="row">
                <Menu  {...{active : 1, linkParams : linkParams}}/>
                <div className="col-md-8">
                    <FrameworkTable {...this.state}/>
                </div>
            </div>
        );
    }
}
```

Z výpisu je možné vidět, že si komponenta po zavedení do aplikace zaregistruje do skladu metodu *onStatusChange*, která nastaví nový stav komponenty. Metoda *setState* je vystavována React komponentou a v případě jejího provolání se vyvolá i samotné překreslení (metoda *render*). Následně zavolá akci pro získání frameworků.

Komponenta `FrameworkTable` je jednoduchá komponenta, která podle stavu buď zobrazuje kolekci frameworků nebo detail či formulář pro editaci frameworku. Ze samotného kódu je možné snadno vidět, jakým způsobem fungují, a tudíž není v textu dále podrobněji popsáno.

Pro lepší pochopení fungování Reflux, následuje obrázek, který ukazuje vztahy mezi jednotlivými objekty.



Obrázek 23 – Reflux tok dat (převzato z [20], upraveno)

V této kapitole byly popsány nejdůležitější poznatky při vývoji klientské části webové aplikace. Nyní bude následovat krátká kapitola zabývající se serverovou částí aplikace.

4.5.2. Serverová aplikace

Serverová část aplikace je v mnoha směrech podobná aplikaci, která byla popsána v kapitole 4.4. Pro první dotažení aplikace je požit Thymeleaf, kde vzhledem k tomu, jakým způsobem je do složky *webapp* vytvářena aplikace, musela být pozměněna jeho konfigurace. Jedná se pouze o odstranění prefixu, který mířil do složky *templates*. Nyní bude šablona hledána přímo z kořenového adresáře, kde se nachází zaváděcí soubor *index.html*.

Pro první načtení stránky zůstal původní kontrolér pouze s metodou *home*, která ho nechá načíst vstupní šablonu. Následující výpis je ukázkou zaváděcího kontroléru.

Výpis 48 – cz.jbradle.poc.web.spring.webapp.controller.MainController

```
@Controller
class MainController {

    @RequestMapping(value = "/index", method = RequestMethod.GET)
    public String home() {
        return "index";
    }

}
```

Následně byl vytvořen nový kontrolér, který slouží jako REST rozhraní do aplikace. Vzhledem k tomu, že je použit `ThymeleafViewResolver`, je možné tento kontrolér implementovat velmi jednoduše. Stačí ho označit anotací `@RestController`, která je součástí

Spring Web frameworku. Namapované metody pak budou vracet objekty či kolekce objektů, které budou následně zkonvertovány do JSON reprezentace dat. Tento kontrolér obsahuje i všechny ostatní metody potřebné pro správnou funkci aplikace. REST rozhraní je zároveň dobré mapovat na separátní adresu. V tomto případě je použit prefix */rest*. Následující výpis je ukázkou kontroléru, který bude vracet data pro dotaz na získání všech frameworků (*getAllFrameworks*).

Výpis 49 – cz.jbradle.poc.web.spring.webapp.controller.FrameworkController

```
@RestController
@RequestMapping("rest/")
class FrameworkController {

    @Autowired
    private FrameworkService service;

    @RequestMapping(value = "/getAllFrameworks")
    public List<FrameworkDTO> getAllFrameworks() {
        return service.getAllFrameworks();
    }

}
```

V tuto chvíli je již možné spustit server a klientská webová aplikace bude připravena k použití.

5. SHRNUTÍ VÝSLEDKŮ

Tato diplomová práce si kladla za úkol několik cílů. Prvním z nich bylo seznámení čtenáře s obecnými koncepty a pojmy, které se pojí s vývojem webových aplikací a ve druhé řadě i samotné seznámení s vybranými frameworky. Těchto cílů bylo dosaženo v rámci kapitol 2 a 3.

V třetí kapitole, která se zabývá podrobným popisem možností vývoje webových aplikací pomocí Spring frameworku a obecným popisem frameworků React a Flux pro klientské aplikace, vznikl referenční dokument, který by mohl posloužit novým vývojářům, kteří se chtějí této problematice věnovat. V kapitole o Spring Frameworku je nejprve popsáno jádro a základní použití frameworku. Následně jsou detailně popsány hlavní komponenty použité pro vývoj webových aplikací.

Čtvrtá kapitola obsahuje popis vytvoření POC projektů, které ukazují praktické použití vybraných frameworků. V první části je popsáno použité vývojové prostředí. Následuje popis společného předka projektů, který sdružuje společnou konfiguraci POC projektů. Tento předek má pod sebou několik modulů, které jsou navzájem provázané. V praktické části vznikly tři funkční moduly, ze kterých je možné vycházet při novém vývoji.

Prvním modulem je jednoduchá sestava, určená pro webového designera, která umožnuje snadný a rychlý vývoj. Sestava je plně automatizovaná a ukazuje i způsoby přebírání zdrojových kódů třetích stran.

Druhý modul je ukázkou webové aplikace, která je čistě serverová. Jedná se o aplikaci postavenou na frameworku Spring MVC a ukazuje použití všech modulů pro vývoj webových aplikací pomocí String frameworku popsaných v teoretické části. Dále čtenáře seznamuje s novým způsobem generováním obsahu, a to pomocí šablonovacího nástroje Thymeleaf.

Třetí modul je ukázkou implementace klientské webové aplikace, která je postavená na moderních frameworcích React a Reflux (implementace Flux návrhového vzoru).

6. ZÁVĚR A DOPORUČENÍ

Výběr a používání frameworků je v dnešní době jedno z nejčastěji probíraných témat. Ať už se jedná o velké společnosti či malé týmy vývojářů. Pro samotný výběr je k dispozici mnoho možností, jakým směrem se vydat. Ke každému z těchto směrů existuje spousta dokumentů, které dané přístupy definují. Málo kdy jsou však sepsány v ucelené podobě, jako je tomu například v této práci, ve které jsou popsány dva přístupy k vývoji. Tím by tato práce mohla být zajímavou pro určitou skupinu lidí. Jelikož obsahuje i samotné funkční projekty, nabývají tak jednotlivé popisy na důvěryhodnosti.

Vzhledem k tomu, že byl autor zaměřený spíše na platformu Java, byla pro něj klientská aplikace nejtěžší částí práce. Výsledný modul však předčil očekávání a přinesl pro autora nový pohled na tuto problematiku. Ten povede k bližšímu seznámení s daným tématem a snahu o maximální využití nového přístupu.

Celkově tato práce ani z daleka nepokrývá možnosti a přístupy vývoje webových aplikací. Může ale sloužit jako referenční dokument pro vývojáře, kteří se této problematice chtějí věnovat. Určitě tak zrychlí jejich první kroky a navede je správným směrem.

V profesi vývojáře není nikdy jednoduché držet krok s dobou. Nové technologie a přístupy přichází prakticky s každým dalším dnem. Spousta lidí je buď přehlíží, nebo se zarputile drží starých a osvědčených postupů. Neříkám, že všechny změny vedou k lepším závěrům. Některé změny jsou dobré a jiné zase být nemusí. Určitě je však dobré zkoušet nové věci a nežít v minulosti.

Autor tedy doporučuje nebát se změn a zkoušet nové přístupy. Takový vývojář se pak stává neocenitelným přínosem pro programátorskou komunitu a nikdy nebude mít problém najít uplatnění na poli svého oboru.

7. LITERATURA

[1] PECINOVSKÝ, Rudolf. *OOP a Java 8: Návrh a vývoj složitějšího projektu vyhovujícího zadanému rámci* [online]. 2015 [cit. 2016-04-05]. ISBN 978-80-87924-03-7. Dostupné z: https://books.google.cz/books/about/OOP_a_Java_8.html?id=MsTNCgAAQBAJ&redir_esc=y

[2] The Art of Separation of Concerns. *Aspiring Craftsman* [online]. 2008 [cit. 2016-04-05]. Dostupné z: http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/

[3] LAYKA, Vishal. *Learn Java for web development*. New York: Apress, 2014 [cit. 2016-04-05]..

ISBN 9781430259848.

[4] Top 4 Java Web Frameworks Revealed: Real Life Usage Data of Spring MVC, Vaadin, GWT and JSF. *ZEROTURNAROUND* [online]. 2015 [cit. 2016-04-05]. Dostupné z: http://zeroturnaround.com/rebellabs/top-4-java-web-frameworks-revealed-real-life-usage-data-of-spring-mvc-vaadin-gwt-and-jsf/

[5] Amuthan, G. *Spring MVC: Beginner's Guide*. Packt Publishing, 2014.

ISBN 9781783284870.

[6] Sharma, J., Sarin A. *Getting started with Spring Framework*. CreateSpace Independent Publishing Platform, 2014. ISBN 9781480013971.

[7] *Spring Framework Reference Documentation* [online]. 2015 [cit. 2016–04–04]. Dostupné z: http://docs.spring.io/spring/docs/current/spring–framework–reference/htmlsingle

[8] PECINOVSKÝ, Rudolf. *Návrhové vzory: 33 vzorových postupů pro objektové programování*. Computer Press. 2007. ISBN 8025115824.

[9] POLLACK, Mark. *Spring Data: modern data access for enterprise Java*. 1st Edition. Sebastopol, CA: O'Reilly, 2013. ISBN 1449323952.

[10] *Spring Data JPA – Reference Documentation* [online]. 2015 [cit. 2016–04–04]. Dostupné z: http://docs.spring.io/spring–data/jpa/docs/current/reference/html

[11] How to avoid Hibernate Lazy Initialization Exception. *Ignacio Suay* [online]. 2014 [cit. 2016-04-20]. Dostupné z: http://ignaciosuay.com/how-to-avoid-hibernate-lazy-initialization-exception/

[12] *Hibernate ORM documentation (5.0)* [online]. 2015 [cit. 2016–04–04]. Dostupné z: http://hibernate.org/orm/documentation/5.0/

[13] *Web MVC framework* [online]. 2015 [cit. 2016–04–04]. Dostupné z: http://docs.spring.io/autorepo/docs/spring/current/spring–framework–reference/
html/mvc.html

[14] MAK, Gary. *Spring recipes: a problem–solution approach*. Berkeley, Calif.: Apress, 2008. ISBN 1590599799.

[15] *React – Getting Started* [online]. [cit. 2016–04–05]. Dostupné z: https://facebook.github.io/react/docs/getting–started.html

[16] *Flux – Overview* [online]. [cit. 2016–04–05]. Dostupné z: https://facebook.github.io/flux/docs/overview.html#content

[17] *Orika – User Guide* [online]. [cit. 2016–04–05]. Dostupné z: http://orika–mapper.github.io/orika–docs/

[18] *Thymeleaf – Tutorial: Using Thymeleaf* [online]. [cit. 2016–04–05]. Dostupné z: http://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html

[19] *Webpack Documentation* [online]. [cit. 2016–04–05]. Dostupné z: https://webpack.github.io/docs/

[20] *React.js architecture – Flux VS Reflux* [online]. [cit. 2016–04–05]. Dostupné z: http://blog.krawaller.se/posts/react-js-architecture-flux-vs-reflux/

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2015/2016

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Brádle Jiří	Přibyslav 27, Nová Paka - Přibyslav	I1300559

TÉMA ČESKY:

Frameworky pro vývoj Java webových aplikací

TÉMA ANGLICKY:

Development in Java web application frameworks

VEDOUCÍ PRÁCE:

doc. Ing. Filip Malý, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl práce:

Zjednodušit výběr frameworků používaných na všech architektonických vrstvách webové aplikace. Seznámení se stávajícími technologiemi. Výsledek práce by měl pomoci v tomto oboru odpovědět na otázku, zda dokáže výběr nových vhodných technologií realizovat levnější vývoj a v jakých případech užití.

Osnova práce:

1. Úvod
2. Vývoj webových aplikací
3. Přehled a výběr stávajících frameworků
4. Proof of concept vybraných technologií
5. Závěr
6. Literatura

SEZNAM DOPORUČENÉ LITERATURY:

Podpis studenta: ..... **Datum:** .....

Podpis vedoucího práce: ..... **Datum:** .....

(c) IS/STAG , Portál - Podklad kvalifikační práce , I1300559 , 16.04.2016 15:35