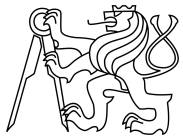
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



# ZADÁNÍ DIPLOMOVÉ PRÁCE

| | |
|---|---|
| **Název:** | Knihovna pro správu autorizačních a autentifikačních údajů pro projekt psaný ve Spring frameworku |
| **Student:** | Bc. Vít Steklý |
| **Vedoucí:** | Ing. Josef Pavlíček, Ph.D. |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Webové a softwarové inženýrství |
| **Katedra:** | Katedra softwarového inženýrství |
| **Platnost zadání:** | Do konce letního semestru 2016/17 |

## Pokyny pro vypracování

Seznamte se se způsoby autentizace uživatelů a autorizace zařízení protokolem OAUTH 2.0. Seznamte se s Java frameworky Spring a Hibernate. Prozkoumejte knihovnu Spring Security. Využijte API této knihovny, navrhněte univerzální knihovnu pro konfiguraci autentizačních údajů uživatelů a správu jejich zařízení pomocí zmíněného protokolu. Pomocí metod vytvořené knihovny by mělo být možné uživatele registrovat, mazat přihlašovací údaje, zobrazit využívaná zařízení a autorizované aplikace. K výsledné knihovně vytvořte jednoduchou aplikaci. Aplikaci otestujte, ověřte její bezpečnost. Samotnou knihovnu vystavte k dispozici jako open source řešení.

## Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 13. prosince 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

# Knihovna pro správu autorizačních a autentifikačních údajů pro projekt psaný ve Spring frameworku

***Bc. Vít Steklý***

Vedoucí práce: Ing. Josef Pavlíček, Ph.D.

29. června 2016

# Poděkování

Rád bych tímto poděkoval Ing. Josefu Pavlíčkovi, Ph.D. za jeho klidný, také svědomitý přístup k vedení, této diplomové práce. Také bych rád ocenil jeho rady a připomínky, které mi poskytl při vedení a konzultování diplomové práce.

# Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 29. června 2016 .....

### Odkaz na tuto práci

Steklý, Vít. *Knihovna pro správu autorizačních a autentifikačních údajů pro projekt psaný ve Spring frameworku.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

# Abstrakt

This master thesis proposes as implementation of library for managing users accounts in the Spring framework based on Java. For its abstractness deals libraries Spring Security and Spring OAuth2.0, for their ability to abstract connections.

**Klíčová slova** Java, Spring Framework, OAuth2.0, Hibernate, Spring Security, správa uživatelů

# Abstract

This master thesis proposes implementation of library for managing users in the Spring framework based on Java. For its abstractness discussing about libraries Spring Security and Spring OAuth2.0, for their ability to abstract connections.

**Keywords** Java, Spring Framework, OAuth2.0, Hibernate, Spring Security, user managment

# Obsah

# Seznam obrázků

# Seznam tabulek

# Úvod

V dnešní době, velkém množství zadaných konceptů vývoje aplikací, obsahuje požadavky na zpřístupňování pouze určitých částí systému, určitým uživatelům, uchováním jejich identit při tvorbě dat nových, či nutnost pouhé selekce dat dle konkrétního uživatele systému. Tato problematika se tedy následně řeší různými druhy autentizace (přihlášení) a autorizace (uživatelské role, skupiny).

Spousty uživatelů má tak nyní po velké škále aplikací rozmístěno množství svých účtů. Díky různým požadavkům na složitosti zadávaných hesel, možnosti automatického „zapamatování“ dochází k brzkému zapomenutí samotným uživatelem. Tuto problematiku se v dnešní době snaží vyřešit projekty jako například OpenID, implementace LDAP služeb, či moderní sociální sítě typu Facebook, či účty Google. Tato uživatelská nesporná výhoda přináší problematiku s absencí takových účtů samotných uživatelů. Prozrazuje určité množství soukromých informací o samotných uživatelích provozovatelům autentizačních služeb. Získávají totiž informace o tom, které služby a s jakou časovou frekvencí jsou uživateli využívány. Ač jsou tyto údaje v drtivém množství zpracovávány anonymně, výsledné statistické informace mohou výrazně napomoct k analytickému byznys rozhodování. Firmy a jejich vývojová oddělení tak nerada tyto uživatelsky přívětivé způsoby využívají a jsou často nucena si samotná registrační prostředí vyvíjet vlastními prostředky.

Pokud prostředí nevyžaduje vyšší bezpečnostní nároky, například kvůli choulostivosti obsahu, tak po samotných uživatelích je vyžadován nepsaný standard — registrace, potvrzení emailové adresy prostřednictvím zaslaného emailu a obdobná implementace zapomenutého hesla. Z pohledu správců systému je třeba tyto uživatele mít možnost spravovat — tvorba, editace, smazaní, blokace uživatele (CRUD operace) a reset hesla. Nedílnou součástí zadání IT projektů tak velmi často bývá tvorba prostředí pro manipulaci s uživatelskými účty.

## Motivace

V popisovaném problému tak můžeme nalézt společný význam pro skupinu věcí a úkonů. Opakovaný vývoj této části se tak stává finančně zbytečným nákladem. Mimo jiné vývoj opakujících se částí (bez ohledu na druh zakázky) samotné programátory nebaví. Vytváří tak prostor špatně napsaných klíčových částí systému, čímž mnohdy zvyšuje míru zranitelnosti celého systému. Typickým příkladem může být incident, kdy hackeři prolomili soukromou e-mailovou schránku premiéra Bohuslava Sobotky, a stáhli z ní desítky zpráv, ve kterých se projednávají státní i soukromé záležitosti [2].

Vzniká zde prostor pro generalizaci, jelikož se v mnohých programech vyskytují logicky stejné celky (ať už se jedná o objekty, funkce nebo cokoliv jiného) na stejné úrovni [3]. Tyto logické celky tak můžeme abstrahovat na vyšší vrstvy, či jako pomocné knihovny a v dalším vývoji efektivně využít. Vytvořené knihovny, které se zveřejní pod některou z otevřených licencí, se mohou rychle stát daleko robustnějšími knihovnami. Takový nástroj nám urychlí samotný vývoj a s malými náklady nabídne víc, než samotný zákazník očekává.

## Cíl práce

Cílem této práce bude vytvořit knihovnu pro řešení problematiky user managmentu v Java frameworku Spring I/O. Její robustnost bude postavena na podrobném pochopení zmíněného frameworku. Komponenta zajistí svoji stavbou proti existujícím interfacům a navrhne případ jejich implementace. Silný důraz práce bude kladen na znovupoužitelnost tak, aby byla komunitním přínosem.

## Členění textu

Text této práce je rozdělen do pěti kapitol. V první části prozkoumáme zadané technologie. Ve druhé kapitole jsou definovány funkční a nefunkční požadavky, ke kterým vymodelujeme případy užití a připravíme scénáře. Kapitola třetí obsahuje návrh a implementaci celé knihovny s popisem řešení problematiky, která v průběhu implementace nastala. Čtvrtá kapitola pojednává o testování vytvořeného kódu, probere několik testovacích způsobů, které na závěr vyhodnotí. Poslední kapitole se bude zabývat způsobem publikace naprogramované aplikace.

Kapitola 1

# Technologie

V této kapitole se budeme zabývat postupnou rešerší jednotlivých technologií, kterých se tato práce týká. Aby generalizace, respektive abstrakce komponenty, byla využitelná i na jiných projektech, je nutné technologie důkladně prozkoumat a porozumět zavedenému konceptu. Pokud využijeme stejnou architekturu a standardy známé z napojovaných projektů, které při tvorbě využíváme, bude abstrakce komponenty efektivnější. Samotný kód se následně stane daleko čitelnější pro ostatní programátory. Navíc se nám lépe podaří využít hlubších vrstev systému, a tak podchytit daleko rozsáhlejší množství přístupů. Jen tak vytvoříme robustnější řešení, takové, co bude pro ostatní vývojáře atraktivní při volbě mezi vlastní tvorbou a využitím tohoto kódu.

Následující odstavce popíší technologie, jejich architekturu a životní cykly nejdůležitějších komponent. Pro znovupoužitelnost tvořené aplikace odstavce zmíní historické milníky tak, aby byly použity pouze technologicky perspektivní knihovny. V odstavcích budeme vycházet především z knih *Introducing Spring Framework: A Primer* [4], *Spring in Action* [5], *Spring Security 3.1* [6] a velkého množství manuálů dostupných na webu.

## 1.1 Hibernate

Modelování aplikací se inspiruje reálným světem ve snaze zachycení reality okolního světa. Poznatky jsou na základě analýzy problematiky zachycovány jako entity, zatímco v relačním světě databází je prezentují řádky, v objektově orientovaném jazyce třídy [7].

Cílem objektově relačního mapování je synchronizace využívaných objektů v aplikaci s jejich záznamy v databázovém systému. Jedná se o technologii sloužící k převodu dat mezi relační databází a objektovým modelem. Převádí data z databázového světa a jednotlivé řádky prezentuje jako virtuální objekty. Synchronizace zajišťuje celkovou persistenci dat, díky které lze virtuální objekty využívat ve zbytku prostředí [8]. V architektonickém modelu MVC tak mnohdy suplují modelovou část.

S tímto nápadem v roce 2001 přišel Gavin King, který tehdy vyvíjel pro společnost Cirrus Technologies. Společnost, která se zabývala především technologií EJB2. Jeho cílem, bylo vytvořit knihovnu více komplexnější s lepšími vlastnostmi, než tomu bylo u knihovny EJB2. V roce 2013 vychází Hibernate 2, který se dočká výrazných vylepšení oproti předchozí verzi, což výrazně pomáhá popularizaci tohoto ORM[9]. Momentálně je Hibernate spravován společností RedHat, jež jej odkoupila v roce 2006 od firmy JBoss. Společně s touto akvizicí přešel i tvůrce tohoto projektu, který se na samotném vývoji stále podílí.

Popularitu tohoto projektu nemusíme měřit pouze pomocí velikosti komunity, četností využití, ale také tím, že v projektu našlo inspiraci velké množství jiných ORM postavených nad konkurenčními jazyky. Mezi takové knihovny patří NHibernate, který našel inspiraci dokonce i v názvu. Jedná se o nepřímý port pro jazyk C#.

V poslední době sílící popularita NoSQL databází vytváří napětí v této komunitě, což vedlo k posilování síly projektu s názvem Hibernate OGM. Projekt, který je pospolu s Hibernatem vyvíjen ve stejné společnosti, udržel jednotné rozhraní JPA (Java Persistence API). Knihovna komunikuje s databázovými systémy MonhoDB a Neo4j [10]. Jednotné rozhraní pomůže vývojářům velmi efektivně migrovat do světa NoSQL své vyvinuté systémy. Vývojář tak může velmi pohodlně využít všech výhod relačních, dokumentových a grafových databází uvnitř jedné aplikace, aniž by výrazně musel rozšiřovat své znalosti o MapReduce framework či Graph Query Language.

### 1.1.1 Java Persistence API

Interface jenž vznikl za účelem abstraktního objektově relačního mapování (dnes již můžeme říct objektově databázového mapování), typicky definuje entitu jako objekt, reprezentovanou například tabulkou v relační databázi, či objektem v dokumentové databázi. Každá instance této třídy představuje řádek své tabulky, nebo prvek v dynamickém poli [11].

#### 1.1.1.1 Požadavky entitní třídy

JPA definuje několik základních vlastností, které je nutné u těchto entitních tříd nakonfigurovat tak, aby došlo ke správnému mapování.

- Třída musí být anotována pomocí *javax.persistence.Entity*.
- Třída může obsahovat více konstruktorů, ovšem musí být bez parametrů typu *public* nebo *protected*.
- Třída a její metody nesmí být deklarovány jako *final*.
- V případě, že chceme, aby session beana pracovala s instancemi této třídy, musíme implementovat rozhraní *Serializable*.

- Lze využít dědění i ne-entitních tříd.
- Atributy třídy musí být deklarovány jako *private*, *protected* nebo *package-private*. Přístup k atributům je nutné zajistit přes gettery a settery nebo jiné metody[11].

#### 1.1.1.2 Životní cyklus entity

Pro hlídání životního cyklu entit (vyobrazený na obrázku 1.4) slouží instance EntityManager. EntityManager API slouží k vytvoření, odstranění nebo vyhledání entitních tříd za pomocí primárního klíče, nebo za pomoci dotazu [12]. Životní cyklus se skládá z:

**public void persist(Object entity);**

- Pokud se přechod podařil, dojde ke změně stavu entity na *managed*.
- Při příštím zavoláním *flush* nebo *commit* dojde k persistenci instance do databáze.

**public void remove(Object entity);**

- Stav entity se mění na *removed*.
- *flush* či *commit* instanci odstraní z databáze.

**public void refresh(Object entity);**

- Vyvolá synchronizační proces virtuální instance se záznamem v databázi.
- Obnovovací metoda je určitě pro dlouho trvající transakce, při kterých existuje nebezpečí ztráty/změny dat.

**public Object merge(Object entity);**

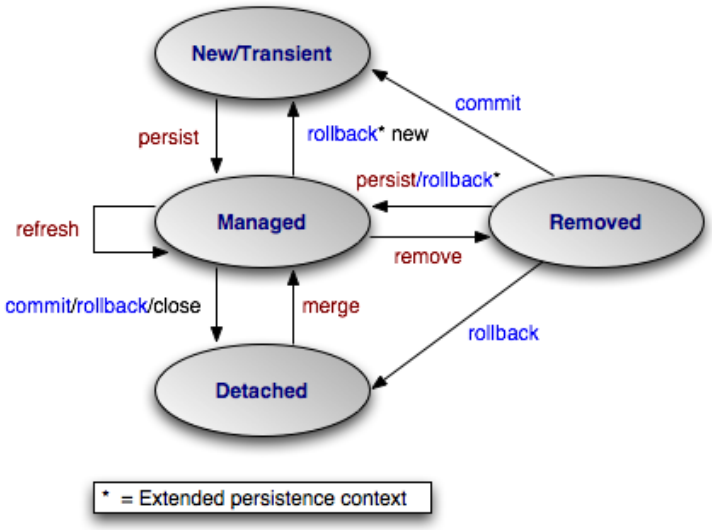
- Tato metoda se nejčastěji využívá v případě, že je entita ve stavu *detach*.
- Ukládá modifikované data entity.
- V případě, že entita byla v průběhu času v databázi změněna, dojde k vyvolání výjimky.

**public void lock (Object entity, LockModeType mode);**

- Tato metoda uzamkne danou entitu. Uzamknutí je definováno dvěma možnostmi:

READ ostatní transakce mohou k objektu přistupovat pouze pro čtení

WRITE ostatní transakce nemohou současně číst a zapisovat [13]




Obrázek 1.1: životní cyklus entity [13]

### 1.1.2 JPQL a HQL

Hibernate Query Language (HQL) a Java Persistence Query Language (JPQL) jsou objektově zaměřené dotazovací jazyky, které se svým charakterem podobají standardnímu SQL. JPQL představuje rozhraní ve stejné abstraktní úrovni jako JPA, vzniklo na základě inspirace z HQL, a tudíž tvoří její podmnožinu. JPQL dotaz je tedy vždy platný v HQL, opačně to pravdou však není. Oba dotazovací jazyky nabízejí typově necitlivé (non-type-safe) způsoby, jak provádět operace jednotlivých dotazů. Pokud trváme na typově bezpečných dotazech (type-safe), lze využít objektového dotazování skrze Criteria API (někdy také jako CriteriaQueries) [14].

#### 1.1.2.1 Dotazy prostřednictvím JPQL

Jestliže dotaz typu SELECT zapíšeme pomocí BNF, vypadá následovně [14]:

```
select_statement ::=
        [select_clause]
        from_clause
        [where_clause]
        [groupby_clause]
        [having_clause]
        [orderby_clause]
```

Oproti standardní SQL formě se výrazně neliší. Zásadní je objektový přístup kdy je dotaz vykonáván nad objekty, tedy entitami. Na rozdíl od SQL, jazyk

JPQL respektuje nadefinované vztahy mezi jednotlivými entitami [15]. Programátor se znalostí SQL by tak neměl mít výraznější problém s využitím zmíněného dotazovacího jazyka.

Intuitivně se znalostmi SQL vytvoříme i další dotazy typu *UPDATE*, *INSERT* či *DELETE*. Přístup k jednotlivým parametrům třídy je prováděn dle konvencí jazyka Java, prostřednictvím teček. Spojení dvou entitních tříd lze provést skrze *JOIN*, který ve variantách *INNER*, *LEFT*, *RIGHT* funguje obvyklým způsobem.

```
String query =
                "UPDATE Accounts a " +
                "SET a.name = :newName " +
                "WHERE a.id = :id";
entityManager.createQuery(query)
        .setString("newName",newName)
        .setString("id",1)
        .executeUpdate();
```

#### 1.1.2.2 Criteria API

Jednou z nejvýznamnějších novinek druhé verze knihovny JPA byla část Criteria API. Jedná se o konstruktivně objektové skládání dotazů (non-string-based) [16]. Toto pojetí se odvrací od dotazů tvořených prostřednictvím textových řetězců.

Za pomocí CriteriaBuilder dochází k tvorbě sémantického stromu uzlů tvořících CriteriaQuery. Stejně jako tomu je u JPQL statické a dynamické dotazy jsou předány metodě EntityManager (createQuery) k vytvoření dotazu. Poté jsou provedeny za použití metod Query API. Jedná se tedy o vyšší vrstvu. Využití Criteria API má tedy několik výhod [15]:

- Díky typové kontrole odpadají některá ověření (dotaz je tedy rychleji kompilován)
- Na sestavování dotazu je nahlíženo více objektově, klesá tak riziko výskytu chyb a překlepů (není třeba využívat zástupných identifikátorů)
- Dotaz je tvořen dynamicky, lze ho tvořit efektivně například pomocí objektového modelu - dekorátor
- Na chyby se přichází během kompilace, ne až za běhu

I přes nesporné výhody není Criteria API tak výrazně využíváno, jelikož práce s ním je poznání méně přehlednější. Programátor si musí vybavit sémantiku standardního SQL dotazu, který následně rozebere, a vytvoří tak nový dotaz. Přehlednost běžného dotazu *SELECT* lze posoudit na následujícím příkladu:

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Account> query
        = cb.createQuery(Account.class);
Root<Account> account = query.from(Account.class);
query.where(cb.equal(Account_.id, 1));
List<User> result
        = em.createQuery(query).getResultList();
```

## 1.2 Spring Framework

Spring Framework je populární open-source aplikační rámec pro vývoj J2EE aplikací [17]. Nabízí komplexní a konfigurovatelné prostředí pro vývoj moderní Java enterprise aplikace nasaditelné takřka na kterékoliv platformě. Klíčovým prvkem je architektonická podpora, díky které se vývojové týmy mohou zaměřit především na business úroveň tvořené aplikace, aniž by se zabývaly časově náročnou konfigurací prostředí [18]. Hlavními funkcemi Springu jsou dependency injection (DI) a aspect-oriented programming (AOP) [5].

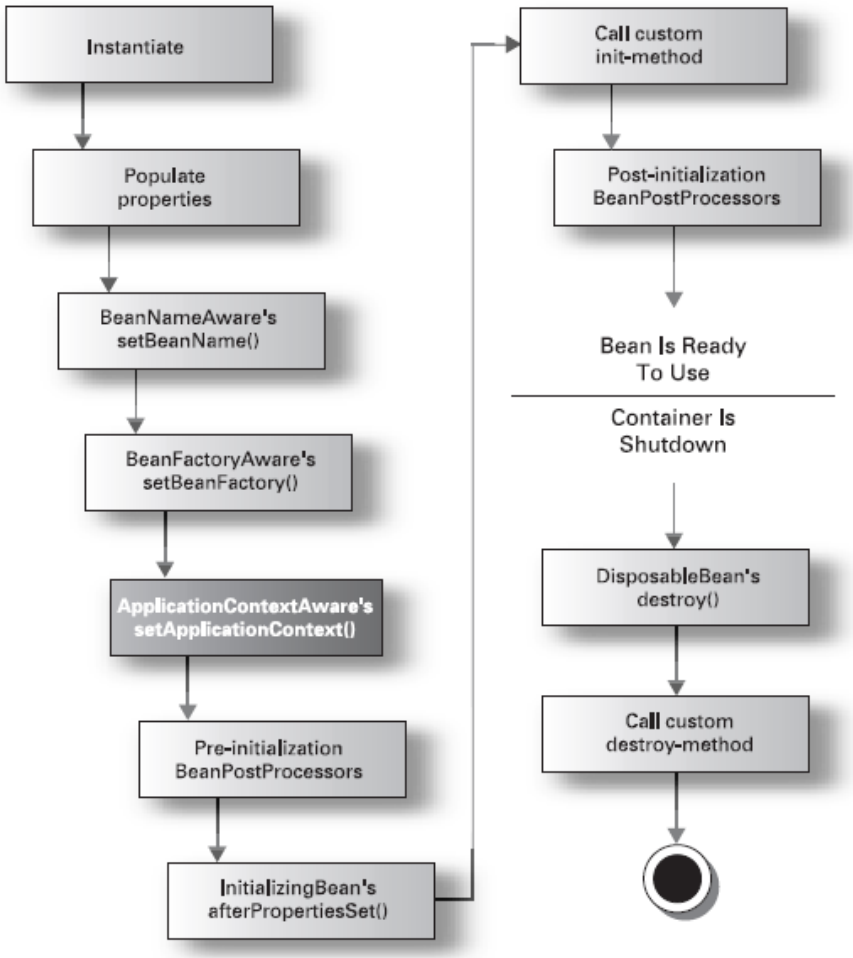
První verze spatřila světlo světa v červnu roku 2003 pod licencí Apache 2.0. Základní důvody vzniku souvisí s problematikou vývoje J2EE aplikací. Odstraňuje závislosti na nejednotně umístěných konfiguračních souborech a jejich problematické srozumitelnosti. Zvyšuje míru abstraktnosti, čímž výrazně zjednodušuje používání dalších částí platformy. Řešení různých aplikačních domén je bez nutnosti použití EJB, například transakční zpracování. Implementuje komponenty pro přístup k datům (například Hibernate). Při vývoji bylo myšleno i na psaní unit testů, a tak výrazně usnadňuje jejich implementaci. Odstraňuje těsné programovací vazby jednotlivých POJO objektů a vrstev, jelikož implementuje návrhový vzor Inversion of Control. Nabízí možnost volby implementace (EJB, POJO) vrstev pro aplikační architekturu, architektura tak předepisuje implementaci [17].

### 1.2.1 Architektura

Framework je založen na návrhovém vzoru Inversion of Control (někdy také jako IoC Container). Tento návrhový vzor uvolňuje vztah mezi těsně svázanými komponentami. Tento kontejner vytváří objekty, tzv. Bean. Spring tyto objekty produkuje nejčastěji na základě konfiguračních souborů ve formátu XML, který obsahuje definice pro inicializaci těchto Bean.

Po inicializaci těchto objektů přichází na řadu Dependecy Injection, které do námi tvořených objektů inicializují potřebné třídy. Způsoby vložení takových objektů jsou *Setter Injection*, *Constructor Injection* a *Interface Injection* [17].

Spring zná dva typy kontejnerů. První je definovaný podle rozhraní *org.springframeworks.bean.factory.BeanFactory*. Jedná se o základní typ, pos-




Obrázek 1.2: životní cyklus Beany [19]

kytuje podporu pro inicializaci a odpojení beany ve chvíli, kdy jsou prostřednictvím DI vkládány do našich objektů. Druhý typ kontejneru je definován rozhraním *org.springframeworks.context.ApplicationContext*. Následné beany jsou inicializovány na základě konfiguračních souborů, které mohou být umístěny uvnitř *class path*, v externím souboru, či načteny z webové stránky. Beany vytvořené prostřednictvím těchto kontejnerů prochází životním cyklem[19].

#### 1.2.1.1 Životní cyklus Spring Bean

1. Spring nalezne definici požadované beany a vytvoří ji (inicializuje ji).

2. Za pomocí metody *getBean* dojde k vložení bean definovaných uvnitř

této beany.

3. V případě, že je implementován interface *BeanNameAware*, dojde k zavolání metody *setBeanName*, která definuje ID beany, pro možné vkládání beany do jiných objektů.

4. Jestliže beana implementuje interface *BeanFactoryAware*, dojde k volání metody *setBeanFactory*.

5. Za předpokladu, že se jedná o kontejner typu *ApplicationContext* (respektive rozhraní *ApplicationContextAware*), je volána metoda *PostProcessBeforeInitalization*.

6. V případě, že je *BeanPostProcessor* navázán na jinou beanu, dojde k volání metody *metoda setApplicationContext*.

7. Nyní dochází k volání speciální inicializační metody.

8. V poslední řadě se zjistí, zda je definován interface *BeanPostProcessor*, který volá metodu *PostProcessAfterInitailzation*.

9. **V tomto okamžiku je beana k dispozici**

10. Když beana implementuje interface *DisponsableBean*, je volána metoda *destory*.

11. Nakonec je zavolána standardní *destroy-method*.

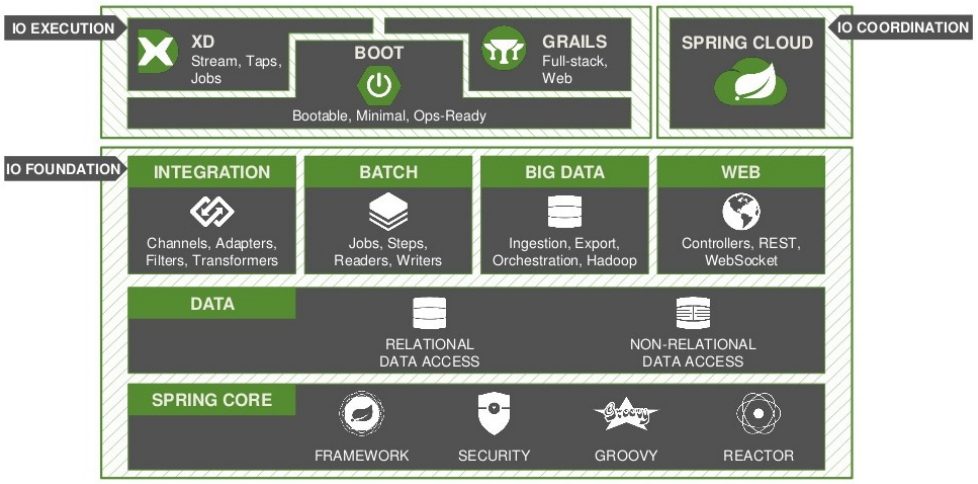
### 1.2.2 Spring I/O

Rok 2013 je vývojovým milníkem této platformy. V dubnu tohoto roku se do komunity připojuje společnost Pivotal Software, Inc., která vyčleňuje nové vývojové týmy a soustřeďuje se na podporu Grails, RabbitMQ, Tomcatu, Redisu a Cloud Foundry služeb. Vývojáři jsou tak schopni zvládnout více problémů než kdy jindy v mnohem kratším čase. Pouze tak mohl framework udržet krok s dnešními technologiemi, zejména v problematice Cloudů, BigData a mobilního vývoje. Spring tak přichází jako nová větší platforma nazvaná jako Spring I/O [20].

Avšak Spring I/O není nová technologie, jedná se pouze o nový způsob náhledu na vývoj nad tímto frameworkem. Tento pohled přišel s verzí 4, jež přináší pojmy jako Spring Boot, Spring XD a Spring Hadoop [4] a další.

#### 1.2.2.1 Spring, Groovy a Grails

Groovy je objektově orientovaný programovací jazyk pro platformu Java [22]. Jedná se o nadstavbu tohoto jazyka, o jeho určitou alternativu. Tento jazyk byl inspirován jazyky Python, Ruby, Perl a Smaltalk. Mezi jeho hlavní výhody




Obrázek 1.3: schéma platformy Spring I/O [21]

patří zjednodušení syntaxe, především ve struktuře objektového zápisu. Mezi výhody zjednodušené syntaxe patří redukce a vizuálního zjednodušení kódu. Kompilace probíhá přímo do *bytecode*, který funguje na standardních JVM. Aplikace je tedy spustitelná ve stejných platformách jako všechny další Java aplikace. Je tedy do značné míry vnímáno jako nadmnožina Javy [23].

Většina souborů typu Java jsou tak funkční v Groovy. Ačkoli oba jazyky jsou si podobné, Groovy kód může být kompaktnější, protože nevyžaduje všechny prvky, které Java naopak vyžaduje [24]. Groovy funkce nejsou k dispozici v Javě, jelikož Groovy zahrnuje jak statické, tak dynamické psaní, přetěžování operátorů, nativní syntaxe pro seznamy a asociativní pole (či mapy), nativní podporu pro regulární výrazy, polymorfní iterace výrazů vložených uvnitř řetězců, další pomocné metody a bezpečný operátor *?* pro kontrolu hodnoty *null* (například *promena?.metoda()* a *promena?.atribut*) [25].

S téměř každým jazykem přichází framework. Groovy využilo možností propojení s knihovnami psanými v Javě. Tvůrce jazyka nejvíce oslovil Spring, na základě kterého vyvinuli framework Groovy. Groovy tak tvoří robustní, rozšiřitelný a lehce použitelný základ pro webové aplikace [4]. Díky využití tohoto frameworku tvůrcům Grails odpadlo spoustu práce. Zaměřili tedy svoji pozornost směrem ke konceptu Java DSL (domain-specific language). Díky společnému bytecodu se rozhodli vrátit výpomoc Springu a publikovali knihovnu Spring DSL, prostřednictvím čehož jednu z nejlepších vlastností programovacího jazyka Groovy, přenesli tradičním Javistům do Springu.

Jestliže jste se někdy setkali s tvorbou Makefilu, nebo návrhem webových CSS, tak jste již částečně na DSL narazili. Na rozdíl od univerzálního jazyka, jakým je samotná Java, DSL je svým rozsahem a schopností poměrně omezená. Jak již název napovídá DSL se intenzivně zaměřuje na určitý typ problému,

na vyjádření řešení v omezeném rozsahu. Díky syntaxi jsou DSL jednoduché a výstižné [26].

Komunita stojící za Springem okamžitě reaguje a ve čtvrté verzi přináší nový typ kontejneru - *org.springframework.context.support. GenericGroovyApplicationContext*, postavený na kontejneru typu *ApplicationContext*. Dochází tak k implementaci možnosti tvorby Bean z *GroovyObject* [27].

#### 1.2.2.2 Spring Boot

Tento plugin platformy předkládá nový zrychlený pohled do vývoje standalone a production-ready aplikací s nutností minimální konfigurace [4]. Jedná se totiž o pohled convention-over-configuration, a tak jím podpořené aplikace stačí už pouze spustit. Odbourává konfigurační závislost nástrojů třetích stran. Umožní tak spuštění aplikace s minimálním úsilím.

Mezi hlavní výhody patří [28]:

- Tvorba soběstačných (stand-alone) aplikací
- Obsahuje Tomcat nebo Jetty webserver, díky čemuž odpadá nutnost nasazování WAR souborů
- Obsahuje startovací konfiguraci POM souboru, čímž zjednodušuje konfiguraci Mavenu (případně základní konfiguraci pro Gradle)
- Definuje prvotní konfiguraci celého Springu, a tak aplikace dalších komponent nevyžadují obtížné nastavení
- Bez nutnosti jakéhokoliv generování kódu, či XML konfigurací
- Nabízí předpřipravené nástroje na měření metrik, kontroly aktivnosti a externí nastavení

## 1.3 Spring Security

Spring Security je mocný a velmi přizpůsobitelný nástroj pro řešení problematiky autentizace a autorizace. Stal se standardem pro řešení této problematiky v aplikacích postavených na Springu. Jeho největší síla pramení v tom, jak snadno jej lze využít pro splnění vlastních požadavků [29]. Jeho nepřímými konkurenty jsou knihovny typu Java Authentication and Authorization Service, nebo Java EE Security, které nabízí velké množství stejných způsobů zabezpečení, avšak Spring Security představuje jinou implementačně jednoduší formu. Jeho způsob implementace umožňuje nepřeberné množství zdrojů, které lze využívat pro běžné bezpečnostní postupy. Vlastnosti uvedených knihoven implementuje způsobem top-to-bottom. Nabízí tak out-of-the-box integrace s dnes běžně používanými způsoby autentizace. Celá knihovna je tak velmi přizpůsobitelná potřebám aplikace s minimálním úsilím

ze strany vývojáře. Pomocí tohoto se knihovna stala velmi úspěšnou a byla portována i do jiných frameworků.

Jeho hlavními přednostmi je jeho komplexita a rozšířitelná podpora různých přístupů autentizace a autorizace. Zabývá se ochranou proti vnějším útokům typu fixací na sezení (session fixation attack), clickjacking či CSRF. Úroveň integrace sahá až do úrovně Servlet API. Knihovna je psána s ohledem na provázanost se Spring Web MVC, díky čemuž je využitelná u běžných, ale i webových aplikací.

### 1.3.1 Autentizace

Ověřování je jedním ze dvou klíčových pojmů zabezpečení. Při vývoji bezpečných aplikací je nutné provést proces ověření proklamované identity subjektu, která se pokouší dostat požadavek ke zdroji [6].

Existují mnohé formy autentifikace, které můžou být aplikovány, ať už softwarovou nebo hardwarovou formou, z nichž má každá své vlastní výhody a nevýhody. Veškeré systémy tak rozlišují dvě formy - neautorizovaný (anonymní) a autentizovaný (zabezpečený) uživatel. Anonymní uživatel tak běžně nedisponuje následujícími funkcemi:

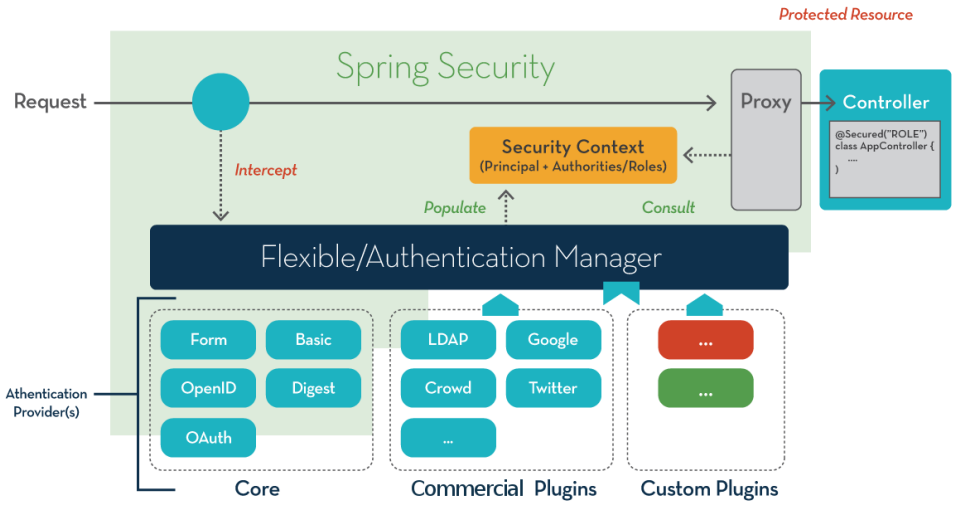
- Opakované přihlášení do systému
- Zobrazení citlivých dat, jako jsou jména, rodná čísla, adresy či informace o kreditních kartách
- Možnost manipulace se stavem celého systému, či změnou jeho konfigurace

#### 1.3.1.1 Credential-based autentizace

Tento způsob ověření funguje na základě důvěryhodnosti třetí strany k provedení ověření ve vztahu klient/server [31]. Pojem *credential* říká, že se jedná o cokoliv, co dostatečně prokazuje identitu uživatele.

Typickým příkladem může být webové přihlášení do běžného emailového účtu. Přihlášení nejčastěji probíhá pomocí uživatelské jména a hesla. Server poskytovatele schránky porovná uživatelské jméno se seznamem účtů v databázi. U daného záznamu ověří, zda se zadané heslo shoduje. Na základě těchto údajů ověří poskytovatel, že jste platným uživatelem.

Kromě ověření v databázovém systému strana serveru může takové ověření provádět kdekoliv jinde. Typickým příkladem jsou centralizované firemní adresářové servery, například Microsoft Active Directory [6], OpenLDAP, nebo dnes hojně využívané uživatelské adresářové servery Facebook, Google nebo OpenID. Pro účely sjednocení ověření vůči adresářovému systému vznikl protokol LDAP.




Obrázek 1.4: Spring Security architektura autentizace [30]

LDAP je založen na bázi technologie X.500, která byla publikována v roce 1988. Většina adresářových serverů implementuje vlastní uživatelské rozhraní, jež samo o sobě řeší problematiku registrace uživatelů a zapomenutého hesla. Napojení knihovny, která je tvořena na základě této práce, by mělo být postaveno tak, aby bylo později možné adresářové služby tohoto typu používat.

#### 1.3.1.2 Více faktorová autentizace

I když metoda credential-based 1.3.1.1 ověření k účtu prostřednictvím uživatelského jména a hesla je nejčastější způsob přihlášení, tak tato jednoduchá kombinace mnohdy nedokazuje, že se jedná opravdu o daného uživatele. Velké množství informačních systémů uživatelské jméno zveřejňuje, uživatelé pod tímto jménem mnohdy vystupují. Ostatní systémy uživatelské jméno staví na emailové adrese, která je lehce dohledatelná. Takové ověření tedy zdaleka neprokazuje, že se jedná o přihlášeného uživatele, neboť tento údaj je velmi lehce dohledatelný. Většina uživatelů je chráněna tedy pouze tajným heslem, kdy při nesplnění důkladných bezpečnostních pravidel tento jediný faktor — heslo, je uhodnutelné. Bohužel špatně zabezpečených účtů se na internetu nachází celá řada, jelikož samotným uživatelům je často jedno, jak bezpečné přihlašovací údaje zadal. Pro zajištění vyšší bezpečnosti, zamezení problémům týkajících se úniku a krádeží dat, způsobených prolomením nebo odcizením hesla, se využívá dvoufaktorová autentizace. Kromě faktoru hesla, přibývá o jeden faktor navíc, čímž dochází ke snížení rizika bezpečnostního incidentu [32].

Typickým případem je například výběr peněz z bankomatu. Uživatel zde přichází se svou platební kartou, ke které zadává osobní PIN. Tento typ ověření je podobný uživatelskému jménu a heslu kromě toho, že uživatelské jméno je

zakódováno na kartě v podobě čipu, nebo magnetického proužku. Kombinace fyzické karty a uživatelsky zadaného PINu umožňuje bance zjistit, zda máte přístup k účtu. Kombinace hesla a fyzického zařízení (karta) je dvoufaktorovu autentizací.

Mimo fyzické zařízení se mezi dvoufaktorové autentizace řadí také následné ověření pomocí mobilního zařízení. Tento způsob se začal hojně využívat například u internetového bankovnictví, kdy po zadání identifikátoru a hesla, dochází k výzvě zadání číselného kódu. Číselný kód je po úspěšném průchodu první úrovní autentizace zaslán na mobilní zařízení pomocí SMS. Telefonní číslo je zadáváno během zakládání bankovního účtu, u kterého se nachází jeho samotný majitel. Proti prozrazení hesla je uživatel chráněn mobilním zařízením, proti ztrátě zařízení je chráněn heslem.

V poslední době došlo k hojnému rozšíření aplikací na bázi TOTP a HOTP, vycházející z RFC 6238. Jednou z takových aplikací je například Google Authenticator. Tato aplikace běží na mobilním zařízení bez nutnosti internetu, datového či mobilního spojení. Aplikace poskytuje šesti až osmimístné jednorázové heslo, které je postaveno na klíči a časovém faktoru. Díky tomu se toto jednorázové heslo v čase mění a zabraňuje tak jednoduchému prolomení.

#### 1.3.1.3 Hardwarová autentizace

Hardwarová autentizace, anglicky také nazývána jako security token či cryptographic token, je fyzické zařízení sloužící pro autorizaci uživatele. V dnešní době je mnohdy nahrazováno softwarovým klíčem. Zařízení obsahují token, který slouží místo hesla. Vlastnictvím tohoto elektronického klíče, uživatel dokazuje svoji identitu.

Některé klíče ukládají digitální podpis, či biometrická data obsahující informace například o otiscích prstů. Ty nejbezpečnější klíče mívají ochranu proti násilnému otevření obalu, kdy dochází k znehodnocení obsahu. Mnohé USB klíče obsahují miniaturní klávesnice pro zadání kontrolního PINu, tak aby došlo k více faktorové autentizaci. Kromě USB se využívají ještě RFID karty, nebo bezdrátová zařízení přenášející obsah prostřednictvím Bluetooth do klientského systému [33].

Typickým příkladem z reálného světa může být RFID klíč od moderního automobilu. Když se přiblížíte ke svému autu, dochází k aktivaci klíče, který zašle svůj obsah (token) řídící jednotce. Jestliže nastavení řídící jednotky autorizuje přijatý token, respektive pokud se jedná o klíč k danému autu, dochází k automatickému odemčení.

### 1.3.2 Autorizace

Po dokončení autentizace obvykle následuje autorizace, což je souhlas, schválení, umožnění přístupu či provedení konkrétní operace daným subjektem.

Autenticita je pak vlastnost subjektu, jejíž přítomnost se procesem autentizace ověřuje [34].

Autorizace je druhý bezpečnostní koncept Spring Security. Jedná se o klíčovou vlastnost a je nutné porozumět její implementaci. Oprávnění používá informace, které byly potvrzeny během autentizace. Určuje, zda by uživatel měl mít přístup ke konkrétnímu zdroji.

Selhání aplikace v tomto bodě by znamenalo kritickou ztrátu bezpečnosti. Pakliže se podíváme na příklad z bankovního sektoru, schopnost posílat peníze z cizího účtu by bylo kritické narušení bezpečnosti. Prostřednictvím autorizace dáváme tuto možnost pouze na účet přiřazený k uživateli.

Spring Security staví model na dvou principech. Prvním je mapování práv. Práva jsou využívány v případech, kdy jednotlivým uživatelům chceme dát specifické funkce. Například v bankovním systému chceme přidat přístup konkrétnímu uživateli do sekce se statistickými údaji o jeho účtu, kam má standardní přístup pouze bankéř. Druhým modelem jsou autority (někdy známo jako role) - například bankéř, administrátor, a podobně.

Nastavení autorizace tak bývá v kompetenci orgánů, které systém spravují, tedy správce systému, vedoucí oddělení atd. Ve větších systémech nastavení autorizace bývá umožněno (delegováno) v rámci doménových vrstev, například vedoucí prodejny, opatrovník a podobně. Autorizace bývají tedy vyjádřeny jako seznamy řízení přístupu a schopností. Koncový uživatel systému by tak měly mít přístup pouze na data, se kterými potřebuje pracovat [35].

V mnohých systémech figurují anonymní uživatelé (či hosté) jako koncoví uživatelé (například u eshopu). I tito anonymní uživatelé procházejí procesem autorizace, jelikož právě takoví uživatelé mají často nejmenší možná oprávnění z celého systému.

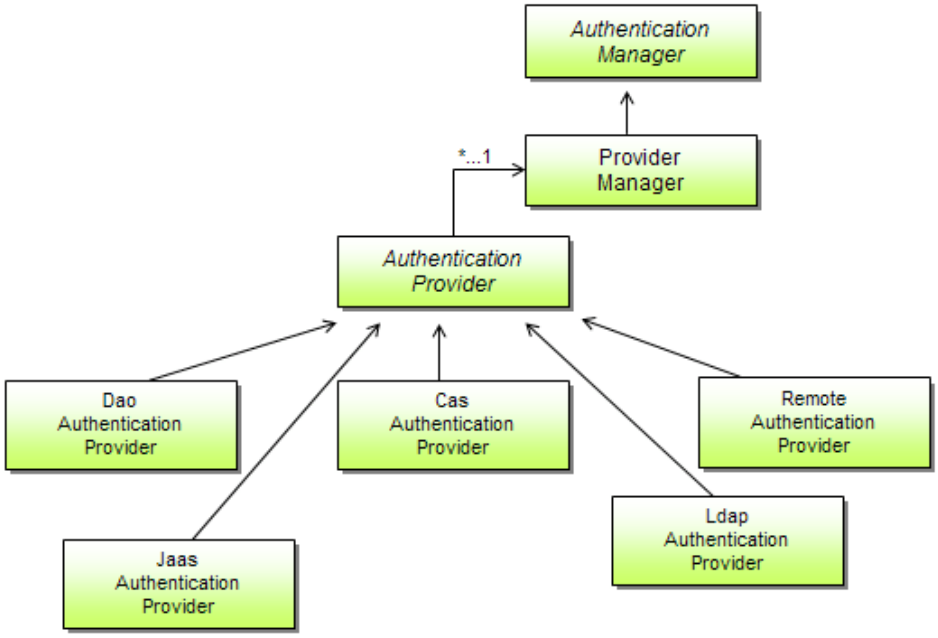
### 1.3.3 API

Jak již byla napsáno úvodem této podkapitoly, Spring Security je mocný nástroj. Snaží se tak efektivně implementovat rozebranou teorii a co nejsnazším způsobem ji nabídnout vývojářům. Řeší nepřeberné množství problémů teorie Autentizace 1.3.1 a Autorizace 1.3.2. Následující část shrne způsoby implementace této problematiky.

#### 1.3.3.1 Autentizace

Hlavním interafacem autentizační části je objekt typu *AuthenticationManager*. Jedním z jeho předpřipravených způsobů implementace je objekt *ProviderManager*, který deleguje odpovědnost za ověřování k jednomu nebo více zprostředkovatelů ověřování, jak je znázorněno na obrázku 1.5.

Účelem třídy *ProviderManager* je řešení problematiky autentizace uživatelů proti jednomu a více zdrojů správy identit. Třída provádí postupný průchod kolekcí založenou na rozhraní *AuthenticationProvider*, která imple-




Obrázek 1.5: Architektura AuthenticationManager [36]

mentuje metody na ověření uživatele do doby, než některý z nich uspěje. V případě, že vyzkouší všechny a neuspěje, uživatel zůstane neověřený. Díky tomu je možné implementovat podporu více mechanismů pro jeden přihlašovací požadavek.

Pokud v naší aplikaci chceme získat úspěšný autentizační mechanismus, lze o něj zažádat dvěma způsoby. Prvním způsobem je volání *SecurityContextHolder.getContext().getAuthentication();*. Druhým způsobem je anotace *@AuthenticationPrincipal*. Výstupem anotace užívané uvnitř parametrů metody, či volání skrze SecurityContext nám vrací objekt postavený na rozhraní *Authentication*. V případě, že se uživatel nenalezne, je navrácena hodnota null. Pokud chceme získat objekt reprezentující specifického uživatele, interface je přetížen metodou *getPrincipal*.

#### 1.3.3.2 Autorizace

Autorizační část Spring Security je tvořena především anotacemi. Metody představující jednotlivé routy naší aplikace lze zabezpečit použitím anotací. V případě, že naše routa má být zabezpečena na základě uživatelovy role, lze využít anotaci *@Secured*.

V případě, že bezpečnost metody závisí na hodnotách tříd, využívají se

*@PreAuthorize* a *@PostAuthorize* antoace. *@PreAuthorize* je postaven nad vstupními informacemi do metody. Nejčastěji tak kontroluje uživatelovu roli, nebo na základě vstupního parametru, který je předáván metodě. *@PostAuthorize* kontroluje autorizaci na základě výstupu metody. Taková metoda vrací určitá data, založená například na informaci o přihlášeném uživateli. Právě v takovém případě může být žádoucí vyhodnocení na základě informace ukryté v daném objektu. Na výstupní objekt se odvolává pomocí speciální proměnné nazvané *returnObject*.

V případě, že naše servisy pracují s dynamickým polem dat, lze využít anotací *@PreFilter* a *@PostFilter*. Jejich využití je obdobné, avšak s předpokladem, že vstupním nebo výstupním parametrem je pole. Na takovéto pole se odvoláváme pomocí proměnné *filterObject*, která reprezentuje obecný prvek pole. Anotaci *@PreFilter* lze cílit pouze na jeden ze vstupních parametrů, který je možné specifikovat pomocí anotačního atributu *filterTarget*.

Doporučený způsob implementace těchto anotací je na servisní vrstvě vyvíjené služby. Atribut anotace, string konfigurující nastavení, je možné skládat z více výrazů (výčet implementovaných výrazů naleznete v tabulce 1.1) a spojovat je prostřednictvím logických spojek *AND* a *OR*. V případě využití více anotací jsou spojovány prostřednictvím logické spojky *AND*. Pro zrychlení vykonávání těchto příkazů, je možné využít externí knihovny *AclCache* a *Ehcache*, které výrazně urychlí vykonávání některých složitějších autentizačních dotazů.

```
@Secured ({"ROLE_USER", "ROLE_ADMIN"})
public void addUser(String name, String pwd);

@PreAuthorize ("hasRole('ROLE_ADMIN')")
public void resetPassword(User user);

@PostAuthorize("returnObject.name == principal.name or"+
                + " hasRole('ROLE_ADMIN')")
public User getUser(String name);

@PreAuthorize ("#user.name == principal.name or "+
                + " hasRole('ROLE_ADMIN')")
public void editUser(User user);

@PostFilter ("filterObject.boss == principal.name")
public List<User> getEmployees();

@PreFilter("filterObject.boss == principal.name")
public void addEmployees(List<User> employees);
```

| Výraz | Popis |
|---|---|
| hasRole([role]) | TRUE pokud má roli |
| hasAnyRole([role1,role2]) | TRUE pokud má některou z rolí |
| principal | objekt reprezentující konkrétního uživatele, např. UserDetails |
| authentication | proměnná AuthenticationPrincipal |
| permitAll | TRUE vždy |
| denyAll | FALSE vždy |
| isAnonymous() | TRUE pokud je uživatel anonymní |
| isRememberMe() | TRUE pokud je uživatel automaticky přihlášený |
| isAuthenticated() | TRUE pokud uživatel není anonym |
| hasIpAddress('192.168.1.0/24') | TRUE zda IP adresa odpovídá předpisu (pouze web) |
| isFullyAuthenticated() | TRUE pokud uživatel není anonym a nejedná se o automaticky přihlášený účet |

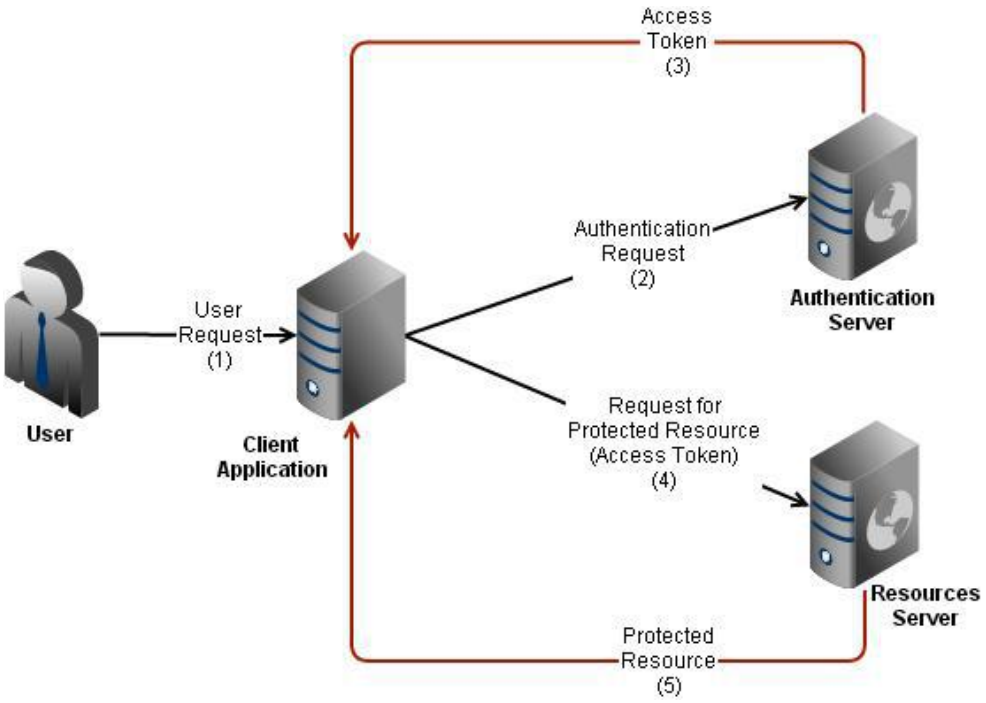
Tabulka 1.1: Seznam implementovaných výrazů [1]

#### 1.3.3.3 Předpřipravená implementace providerů

Jeden z velmi využívaných *AuthenticationProvider*ů je třída *UserDetailsAuthenticationProvider*. Jedná se o základní implementaci jednoduchého jednofaktorového způsobu přihlášení, které prostřednictvím rozhraní *UserDetailsService* řeší proces autentizace - vyhledání a ověření uživatele na základě uživatelského jména. *UserDetailsService* definuje metodu *loadUserByUsername(String username)* s výstupním objektem, který je definován rozhraním *UserDetails*.

Díky těmto dvěma třídám můžeme napojit načtení uživatele z libovolného zdroje a pomocí abstrakce spojené s rozhraním *UserDetails* nastavit autentizaci uživatele (předat hash hesla pro porovnání) a nastavit autorizační parametry (jeho role a práva). Samotný Spring Security nabízí i implementační řešení těchto rozhraní, a to v podobě *JdbcUserDetails* a *InMemoryUserDetails*.

Implementace třídy *InMemoryUserDetails* je využívána především pro testovací účely, kdy plnění databáze je zbytečným časovým nákladem, případně se hodí pro aplikace, které nepočítají s dynamickým přidáváním uživatelských přístupů. Častým způsobem inicializace této třídy je *Bean*a definovaná XML souborem. K načítání dat, respektive správě uživatelů, je definovaná rozhraním *UserDetailsManager*, která je určena standardními CRUD operacemi.




Obrázek 1.6: autentizační schéma protokolu OAuth 2.0 [38]

## 1.4 Spring OAuth

Knihovna Spring Security Oauth byla vytvořena za účelem podpory standardizovaného způsobu autentizace The Open Authentication, dle RFC číslo 6749. Jedná se o otevřený protokol, který byl navržen Blainem Cookem a Chrisem Messinou. Cílem protokolu bylo navrhnout bezpečný způsob autentizace a autorizace oproti API různých služeb, a to jednotně pro desktopové, mobilní i webové aplikace [37].

Aktuální verze knihovny implementuje jak klientskou, tak i serverovou část. Ačkoli je knihovna vyvíjena jako celek, její produkční distribuce jsou rozděleny dle verze OAuth protokolu, tedy Spring OAuth 1.0 a Spring OAuth 2.0.

### 1.4.1 Protokol OAuth 2.0

Tento protokol svoji užitečnost nabízí především ve způsobu ověření a autorizaci uživatele z externí aplikace, či jiných zdrojů. Umožňuje tak oddělit klientský, autentizační a datový server mezi nezávislé výpočetní jednotky (například servery).

Pro pochopení procesu protokolu popíšeme jednotlivé kroky dle obrázku 1.6. Uživatel vytváří žádost na klientskou aplikaci (1) s přihlašovacími údaji.

Klientská aplikace zasílá požadavek s údaji (2) autentizační části, která vrací specifický token relačně postavený k ověřeným údajům (3). Následně uživatel může provést dotaz, který se týká datového serveru, kdy klientská aplikace zasílá požadavek (4) společně s autorizačním tokenem na datovou část. Datová část ověří token u autorizačního serveru a vykoná žádaný úkol nad daty, případnou odpověď zasílá klientské aplikaci (5).

Požadavek klienta obsahuje uživatelské údaje a klientské pověření (*grant_type=password*). Standard definuje čtyři typy grant_type (*authorization_code*, *password*, *client_credentials* a *refresh_token*). Jestliže autentizační aplikace ověří přístupové údaje vrátí *access_token*, jeho čas životnosti a token pro obnovení (*refresh_token*). Dle nastavení datové aplikace takový token zasíláme prostřednictvím http hlavičky (jako header, query řetězce typu POST, nebo GET).

Jakmile vyprší platnost tokenu, dochází k jeho zneplatnění. Datová aplikace začíná vracet hlášku typu *invalid_token*, jelikož expirovaný token není možné ověřit. Klientská aplikace prostřednictvím *refresh_tokenu* bez znalosti přihlašovacích údajů žádá prostřednictvím *grant_type=refresh_token* o token nový [38].

Autorizační server kromě přehledu uživatelských účtů udržuje seznam aktivních tokenů a společně s nimi i refresh_token. Pokud autorizační server drží relaci tokenu a uživatelského účtu ve vazbě typu n:1, může řádek tabulky reprezentující seznam tokenů obsahovat například informaci o zařízení. Informaci můžeme získat například prostřednictvím HTTP hlavičky (velmi nám k tomu napomáhá hlavička *User-Agent*), případně vlastní řešení způsobem získávání a zpracovávání hodnoty *client*. Jakmile tento údaj známe, můžeme uživateli umožnit odstranění tokenu (odstranění záznamu), čímž trvale odhlásíme jeho zařízení.

### 1.4.2 Bezpečnost

Jedná se o poměrně silnou bezpečnostní výhodu, jelikož klientská aplikace, běžící například na mobilním zařízení, trvale neuchovává žádné uživatelské přihlašovací údaje — zejména heslo, či jiné autentizační informace. Ztráta zařízení tak neznamená odcizení údajů. Bez nutnosti změny hesla je tedy po teoretické stránce možné deaktivovat token přidělený specifickému zařízení.

Celý protokol je psán na vyšší vrstvě protokolu HTTP. Standardní verze tak běží v nezašifrovaném prostředí a hrozí reálné riziko odcizení přihlašovacích údajů, nebo samotného tokenu útokem typu man-in-the-middle. Je tedy velmi důležité si pohlídat způsob připojení mezi jednotlivými částmi aplikace, případně posunout komunikační vrstvu na šifrovaný protokol HTTPS.

Pokud je klientská aplikace reprezentována webovou aplikací psanou například Javascriptovým frameworkem AngularJS, nebo React, není vhodné tokeny předávat prostřednictvím query řetězce metody GET. V takovém případě se token nachází v URL adrese dotazu zasílaného na datovou aplikaci.

Nezkušený uživatel tak tuto URL adresu z otevřenější části aplikace vykopíruje a zašle ji svému kolegovi. Běžným případem je zasílání odkazu na obrázek. Protistrana tak získává přístup do systému pod cizí identitou. V nejhorším případě se tato informace dostává do indexace internetových vyhledávačů a po dobu životnosti tokenu tak existuje nebezpečí podvržení identifikovaného uživatele.

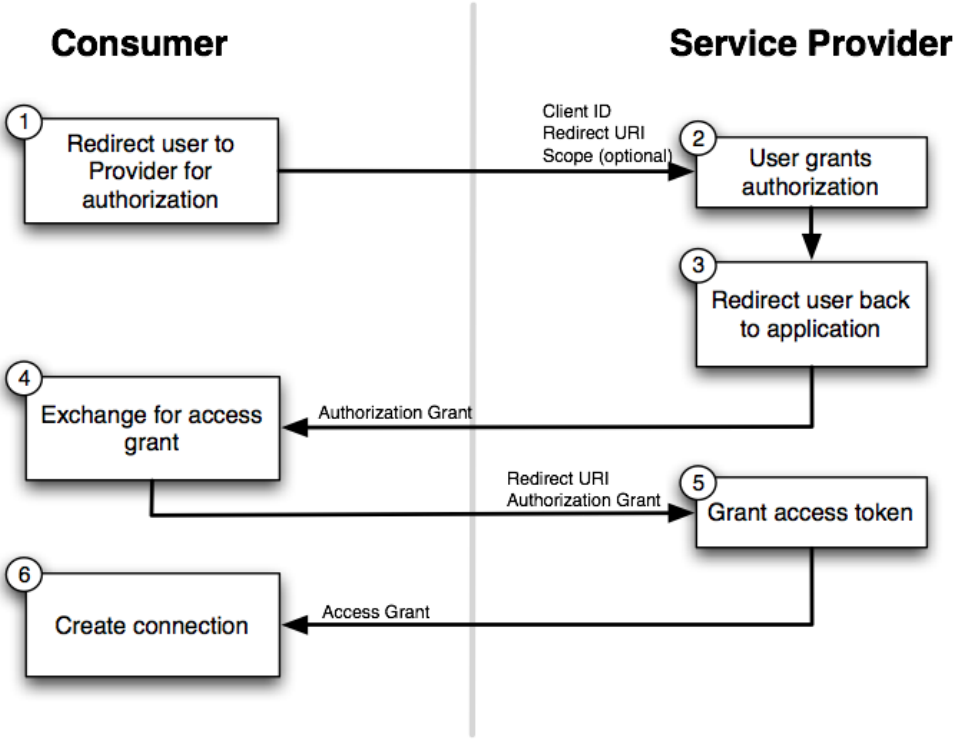
Celá bezpečnost tohoto protokolu spočívá v důvěryhodnost klientské aplikace. Pokud tvořené komunikační rozhraní postavené na protokolu OAuth má veřejné API, k němuž se připojují tvůrci aplikací třetích stran, můžeme povolit autentizační způsob pouze skrze *grant_type=authorization_code*. Tento způsob je použitelný pouze pro webově postavené aplikace. Dochází k přesměrování uživatele na logovací bránu, která běží přímo na straně autentizačního serveru, jež po úspěšném přihlášení přesměruje uživatele zpět do klientské aplikace, kdy je token a refresh_token předán prostřednictvím parametru. Tento postup je znázorněn na sekvenčním diagramu 2.3. V případě, že se nejedná o webovou aplikaci je jediné možné řešení uzavřená klientská knihovna, která obdobně vyvolá přihlašovací dialog, nebo sdílí přístupový a refresh token prostřednictvím sdíleného prostoru specifikovaný prostředím operačního systému. Typickým případem může být operační systém Android, který tyto prostředky sdílí přes objektovou třídu *android.accounts.AccountManager*.

Aby bylo možné detekovat, případně zablokovat klienty třetích stran, nebo je jakýmkoliv způsobem spravovat, protokol OAuth 2.0 implementuje nové dvě nepovinné hodnoty - *client_id* a *client_secret*. Tyto atributy jsou nejčastěji předávány metodou basic-authentication, tedy skrze hlavičku HTTP požadavku. Prostřednictvím těchto údajů dochází k autentizaci klientské aplikace, díky čemuž můžeme rozhodovat i o jejich autorizačních vlastnostech, například účetní software a bankovní účet. Aplikace tohoto typu může dostat autorizační vlastnosti ke čtení výpisů, avšak zákaz k provádění nových příkazů bez ohledu na autorizační vlastnosti uživatele.

### 1.4.3 API

Jak již bylo vysvětleno v kapitole 1.3.3.1, Spring Security autentizaci a autorizaci řeší prostřednictvím *AuthenticationManager*, který přes *ProviderManager* definuje problematiku jednotlivých způsobů přihlášení. Spring OAuth tak poskytuje mechanismus rozhraní *AuthenticationProvider*. Celá knihovna se tak dělí na několik části. Tyto části jsou z vnějšku aplikace spravovány skrze REST rozhraní specifikované OAuth protokolem.

Jedna z částí se zabývá správou tokenů. Prostřednictvím *tokenExtractor* dochází ke kontrole *ServletRequest*, parsování tokenu a jeho následné ověření. Vyhledávání platných tokenů je řešeno rozhraním *TokenStore*, správu a jejich obnovu má na starosti *AuthorizationServerTokenServices*. Interface *OAuth2AccessToken* představuje jeden konkrétní přístupový token. Vyhledání a správa tokenů je řešena autorem předpřipraveným objektem *TokenServices*, jež imple-




Obrázek 1.7: Sekvenční diagram webového přihlášení OAuth 2.0 [39]

mentuje rozhraní *ResourceServerTokenServices*, které definuje danou problematiku za pomocí JDBC konektoru.

V případě využití této knihovny je nutné dodat implementaci rozhraní *UserDetailsService*, popisované v kapitole 1.3.3.3, definující uživatelské účty a jejich oprávnění. Hlavní přispěvatel knihovny Dave Syer plnohodnotně využívá zavedenou architekturu Spring Security a pomoci dobře navržené abstrakce spojuje knihovnu s rozhraní *UserDetailsService*. Pro samostatné využití je nutné dodat implementaci tohoto rozhraní popisovaného v kapitole 1.3.3.3. Rozhraní definuje uživatelské účty a jejich oprávnění. Využití knihovny se tak pro existující aplikace, při dodržení předdefinovaných rozhraní, stává jednoduchým úkonem.

Posledním důležitým rozhraním je *ClientDetailsService*, které řeší problematiku *client_id* a *client_security* popisované v předchozí kapitole 1.4.2. Výstupem této servisy je *ClientDetails*. Rozhraní definuje další množinu autorizačních rolí. Výstupem celého provideru je tedy sjednocení množin uživatelských rolí a rolí přihlášené klientské aplikace. Pomocí spojování autorizačních výrazů (z tabulky 1.1) uvnitř anotací popisovaných v kapitole 1.3.3.2 je možné řešit přístupy nejen podle autentizovaného uživatele, ale i podle klientské aplikace.

Ani na *ClientDetailsService* autor nezapomíná a implementuje jej skrze JDBC konektor. Samotná knihovna tak vyžaduje pouze konfiguraci zdrojů, které již existují ze stávající aplikace, nebo spojením knihoven Spring Security a Spring OAuth 2.0 a je nabídnuto řešení v podobě klasických relačních databází. S aplikací knihoven si tak dokáží poradit začátečníci, kteří nemusí mít rozsáhlé informace o problematice tohoto protokolu. Zároveň vývojáři řešící hlubší problematiku nasazení díky abstrakci mohou napsat vlastní napojení s jinými databázovými systémy, nebo provést rozdělení na části dle obrázku 1.6, kdy autorizační část může být zcela oddělena od datové části [39].

## 1.5 Souhrn zvoleného ekosystému platformy

Závěr této kapitoly bude věnován souhrnu, na základě kterého bude určeno směřování vývoje knihovny. Právě kvůli správnému směru tak, aby tvořená knihovna měla opravdu smysluplné využití, musela být věnována rozsáhlá část této práce. Tvořená technologie tak bude dodržovat kontextové zvyklosti z již existujících knihoven.

### 1.5.1 Problematika uživatelů

Bezpečnostní knihovna Spring Security vytváří velký prostor pro efektivní využití různých způsobů autentizace a autorizace. V případě, že si vývojář zvolí vlastní autentizační způsob, využívá interface *UserDetailsService* či některou z jeho implementací. Tvořená knihovna by tak tento interface měla obohatit o požadované schopnosti, tedy možnost registrace a resetování hesla. Existence *InMemoryUserDetailsManager*, která umožňuje paměťovou správu uživatelů, byla vytvořena mezi rozhraní *UserDetailsManager*, které implementuje základní CRUD operace s uživateli (více v kapitole 1.3.3.3). Tuto vrstvu tedy využijeme, jelikož se jedná o úkony spojené s registrací, či změnou hesla.

Posledním důležitým rozhraním je *UserDetails* reprezentující záznam jednoho konkrétního uživatele. Interface vychází z rozhraní *Serializable*, můžeme tak v případě potřeby počítat s možností uložení celého uživatelského objektu do databáze.

### 1.5.2 Správa zařízení

Z povahy specifikace protokolu OAuth 2.0 je patrné, že o token žádá každá prvně spuštěná klientská aplikace na novém zařízení. Pro správu tokenů v knihovně Spring OAuth 2.0 slouží implementace objektu *TokenStore*. Naším cílem bude tuto knihovnu obohatit o ukládání dalších dostupných informací, které se standardně zasílají skrze HTTP protokol — například hlavička *User-Agent*, či na nižší vrstvě — IP adresa. Prostřednictvím těchto údajů můžeme uživateli předat informaci o tom, které zařízení zhruba z jakého místa naposledy využilo

daný token. Uživatel, nebo správce tak bude mít možnost rozhodnout, zda je token v pořádku, nebo zda jej chce zablokovat. Blokací deaktivuje klientskou aplikaci v cílovém zařízení bez nutnosti změny hesla.

Záznamy jednotlivých tokenů jsou reprezentovány rozhraním *OAuth2AccessToken*. Toto rozhraní bude nově obsahovat dva nové atributy pro získání User-Agenta a poslední přihlášené IP adresy. V budoucnu je možné tuto funkcionalitu rozšířit o držení seznamu IP adres, přičemž pomocí registrace IP adres bude možné lokalizovat přibližnou naposledy přihlášenou adresu. Samozřejmě se nebude jednat o příliš relevantní informaci, jelikož bude maskovatelné například pomocí VPN a podobně.

### 1.5.3 Předpřipravená JDBC implementace

Většina těchto dostupných rozhraní má vlastní implementaci skrze JDBC. Tyto standardní protokoly přepíšeme a nabídneme vývojářům možnost využití entitních objektů postavených na knihovně Hibernate, čímž lépe odstíníme některé databázové systémy, nabídneme pokročilejší správu například revizí, či zrychlíme samotný chod autentizačních procesů díky cachování. Díky Hibernate OGM částečně pokryjeme i možné řešení přes NoSQL databáze.

Kapitola 2

# Analýza

Jak již bylo v předchozí kapitole popsáno, cílem práce je vytvořit knihovnu, která se bude držet běžných zvyklostí frameworku a jeho dostupných knihoven. Knihovnu, která vytvoří efektivní sérii rozhraní, sloužící k registraci, resetování hesla a správě přihlášených zařízení.

V této části diplomové práce se podíváme na rešerši řešení v jiných frameworcích, v jiných jazycích, shrneme model požadavků a případy užití. Vytvoříme tak přesnou technickou specifikaci celé práce.

## 2.1 Rešerše řešení v jiných technologiích

I když námi tvořená aplikace v jazyku Java ve veřejně dostupné podobně neexistuje, stále existuje možnost získání cenných zkušeností načtením best-practices a vyvarováním se nejčastějším chybám. V podkapitolách se podíváme na způsoby řešení v konkurenčních jazycích, které jsou v dnešní době ve velké míře využívány. Rozebereme jejich problematiku autentizace a autorizace, zjistíme, zda nějakým způsobem generalizují registrační průchod, či řeší problematiku zapomenutého hesla. Vyvodíme jejich výhody, které při návrhu implementujeme a nevýhody, které robustním návrhem eliminujeme.

### 2.1.1 .NET Framework

Jedním z nejvýraznějších konkurentů Javy je jazyk C#. Prozkoumání řešení problematiky autentizace a autorizace ve frameworku .NET se stává nezbytností.

Implementace framework řeší přes sérii rozhraní. Každý uživatel systému je reprezentován objektem *ApplicationUser*, který funguje pospolu s autentizačním objektem *IdentityRole*. Následnou registrací *UserManager*, jehož způsob uchovávání dat je postaven na rozhraní *EntityFrameworkStores*, dokáže autentizační způsoby ukládat do různých databázových systémů. Oproti Spring Security zvládá implementaci lépe generalizovanou na systémy jiného než

relačního typu. Díky tomu plní částečné schopnosti Hibernatu, čímž *InMemoryUserDetails* a *JdbcUserDetails* zvládá jedním řešením, avšak nevýhodu vidíme na předem předefinovaných atributech uživatelského objektu.

Problematika registrace nových uživatelských účtů a přihlašování uživatelů je řešena objektem *SignInManager*, kterým je genericitou určen uživatelský typ objektu. Vývojář prostřednictvím *RegisterViewModel* a *LoginViewModel* může ovlivňovat vzhledy formulářů. Řešení myslí i na problematiku aktivačních emailů a zapomenutých hesel, před třídu AuthMessageSender nabízí možnost nastavení potvrzovacího emailu.

Zajímavé je pojetí vložení vícera způsobů přihlášení. Systém je vytvořen tak, aby bylo možné sekvenci mezi pokusem a úspěchem o přihlášení vkládat libovolný kód. Je tedy příležitost do této sekvence vložit přihlášení přes Facebook a podobně. Celý systém je ovšem navržen tak, že každý uživatel vystupující v systému má svoji interní reprezentaci uživatelského objektu v jedné databázi. Ukázka takové sekvence je na příkladu níže [40]:

```
var user = await _userManager.FindByNameAsync(model.Email);
if (user != null) {
    if (!await _userManager.IsEmailConfirmedAsync(user))
    {
        ModelState.AddModelError(string.Empty,
        "Confirmed email to log in.");
        return View(model);
    }
}
var result = await _signInManager
        .PasswordSignInAsync(model.Email, model.Password,
         model.RememberMe, lockoutOnFailure: false);
if (result.Succeeded) {
    return RedirectToLocal(returnUrl);
}
if (result.IsLockedOut) {
    return View("Lockout");
} else {
    ModelState.AddModelError(string.Empty,
        "Invalid login attempt.");
    return View(model);
}
```

Autorizace je zde řešena stejným způsobem jako u Spring Security. Nabízí několik druhů anotací reprezentujících různé možnosti a uživatelské přístupy k jednotlivým kontrolerům. Základní anotací je *[Authorize]* a *[AllowAnonymous]*, které vyžádají přihlášení, případně vytvoří výjimky konkrétním metodám. Anotace *[Authorize]* podporuje volitelný parametr *Roles*, který umožní specifikovat pole potřebných rolí. Pro tvoření složitějších bezpečnostních pra-

videl slouží parametr *Policy*. Bohužel omezování práv dle vstupních parametrů, nebo výstupního objektu tento framework přes anotace neumožňuje.

### 2.1.2 PHP Symfony - FOSUserBundle

PHP framework Symfony2, který byl uveden v roce 2011 po dvou letech aktivního komunitního vývoje, se velmi inspiroval Java frameworkem Spring. Symfony je organizováno jako sada znovu použitelných komponent a svazků. Neobsahuje databázový systém, či podporu NoSQL, nicméně využívá pro svoje propojení PHP ORM systém Doctrine. Systém Doctrine opět našel svoji velkou inspiraci v Hibernate.

Bohužel PHP je čistě interpretační skriptovací jazyk, díky čemuž je výkonnostně daleko pomalejší. Každý návštěvník webu napsaného v tomto skriptovacím jazyce, zavádí zcela novou instanci. To znamená, že paměťové sdílení je nutné řešit skrze specifické knihovny (například Memcache). Kvůli náročnosti se také nevyužívá většího množství rozhraní, proti kterým by se psalo, jelikož interpretační jazyk s každou instancí (návštěvníkem) musí opětovně veškeré soubory načítat. Díky těmto faktorům jazyk nebývá používán na větší enterprise systémy, i když se zde najdou určité výjimky typu Facebook a podobně.

I přes tyto nepříjemné vlastnosti se framework inspiroval i v knihovně Spring Security a vytvořil stejnojmenný plugin Symfony Security, který poskytuje obdobný flexibilní bezpečnostní rámec. Umožňuje také načítání uživatelů z konfigurace, databáze, nebo z dalších míst, právě díky pohodlným interfacům. Autorizační část je řešena obdobným anotačním způsobem.

Největší předností PHP Symfony je knihovna FOSUserBundle. Tato knihovna staví svoji popularitu na rychlé implementaci, snadném využití a takřka okamžité podpoře pro běžnou činnost uživatelských účtů. Tato knihovna staví logiku na implementaci ORM entity, která je děděná z abstraktní třídy *BaseUser*. Tato abstraktní třída definuje především celočíselný atribut ID, na kterém následně staví veškeré své vazby, jímž může být například reset hesla.

U běžných webových stránek tento systém může být zcela dostačující, ovšem u složitějších systémů řešících hlubší problematiku se zdá být nedostačující. Omezenost v podobě konfigurace primárních atributů neumožní přechod na jiný doménový prvek - třeba UUID, využívaný pro databázově neblokující INSERT. Stejně tak závislost na emailu v nějakém velmi sofistikovaném systému nemusí být vždy zcela žádoucí. Tyto potřeby tak nutně znamenají zásah do celého systému této knihovny, nebo nutnost napsání řešení problematiky od samotného začátku.

Inspirativním prvkem této knihovny může být doménový model (tabulka 2.1). Kromě standardních prvků jako id, username a email jsou zde hodnoty *confirmation_token* a *password_requested_at*. Prázdná hodnota *password_requested_at* značí, že *confirmation_token* je token aktivující účet (v případě, že proměnná *locked* je 0). Pokud atributy žádosti o heslo má konkrétní datum, potvrzovací token zde slouží jako token pro resetování hesla.

| Name | Type |
|---|---|
| id | int(11) |
| username | varchar(255) |
| username_canonical | varchar(255) |
| email | varchar(255) |
| email_canonical | varchar(255) |
| enabled | tinyint(1) |
| salt | varchar(255) |
| password | varchar(255) |
| last_login | datetime |
| locked | tinyint(1) |
| expired | tinyint(1) |
| expires_at | datetime |
| confirmation_token | varchar(255) |
| password_requested_at | datetime |
| roles | longtext |
| credentials_expired | tinyint(1) |
| credentials_expire_at | datetime |

Tabulka 2.1: Entita User v knihovně FOSUserBundle

V konfiguraci knihovny se dá určit délka platnosti, která je následně přičtena k datu vytvoření žádosti.

## 2.2 Model požadavků

Na základě rešerše existujících řešení a zvyklostí z uživatelského pohledu této problematiky byl vytvořen seznam požadavků, které definují soupis hlavních vlastností systému.

### 2.2.1 Funkční požadavky

Seznam funkčních požadavků obsahuje seznam nutných úkolů, aktivit a akcí, které knihovna musí vykonávat. Analýza funkčních požadavků bude použita jako základní seznam funkcí systému pro funkční analýzu [41]. Tuto část tedy rozděluji na několik dílčích částí.

#### 2.2.1.1 Registrace

- Poskytne rozhraní pro vytvoření nového uživatele
- Umožní řešit problematiku potvrzení uživatelského emailu

#### 2.2.1.2 Zapomenuté heslo

- Po žádosti uživatele zašle unikátní klíč pro resetování hesla na uživatelovu emailovou adresu
- Umožní uživateli s časově omezenou platností klíče resetovat heslo k jeho uživatelskému účtu

#### 2.2.1.3 Správa zařízení

- Zobrazí seznam uživatelů
- Každý uživatel bude mít výpis přidělených tokenů
- Jednoduchá možnost zneplatnění vybraných klíčů
- Jednotlivé klíče budou obsahovat další informace
  - IP adresa
  - Datum posledního použití
  - Datum přidělení
  - User-agent

### 2.2.2 Nefunkční požadavky

Práce se také zamýšlí nad seznamem vlastností a omezení knihovny. Níže uvedené požadavky se vztahují na všechny části práce.

- Bude navrhována v souladu se standardy a uznávanými konvencemi frameworku Spring, knihoven Spring Security a Spring OAuth2.0
- Naváže a rozšíří existující rozhraní nacházející se v knihovnách uvedených výše
- Veškeré funkční části budou psány proti interfacům tak, aby je bylo možné nahrazovat vlastním dílčím kódem
  - Kladení důrazu na rozšířitelnost
- Implementace všech rozhraních za pomocí JDBC / Hibernate a JavaMailSender
- Softwarové požadavky
  - OpenJDK 8 (JVM)
  - Spring Framework 4.x
  - Spring Security 4.x
  - Spring OAuth 2.x
- Bezpečnost generovaných potvrzovacích klíčů

## 2.3 Případy užití

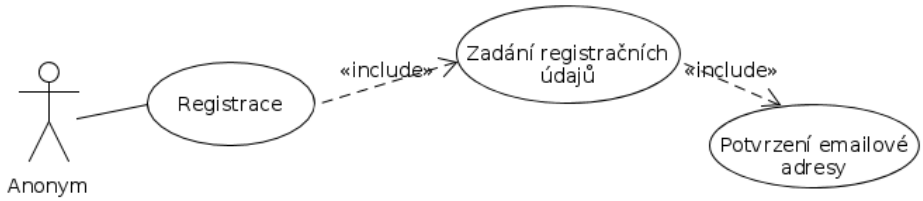
V této sekci se práce věnuje přípravě případů užití. Definuje seznam kroků, které tvoří interakci mezi uživatelem a systémem.

### 2.3.1 Seznam účastníků

Z funkčních požadavků lze vyvodit tři uživatelské role, které mají v systému své logické zastoupení.

#### 2.3.1.1 Anonymní uživatel

Nový návštěvník, který v systému nemá uživatelský účet, není možné ho tedy autentizovat a vystupuje jako anonymní uživatel. Jeho žádaným cílem je vytvoření uživatelského účtu, svoji autentizaci prokazuje vlastnictvím emailové adresy. Autorizaci provádí kliknutím na příslušný odkaz, obsahující potvrzovací unikátně vygenerovaný klíč. Do takového účtu není možné se před potvrzením emailové adresy přihlásit. Klíč má pouze omezenou dobu životnosti, poté je nutné provést opětovnou registraci.



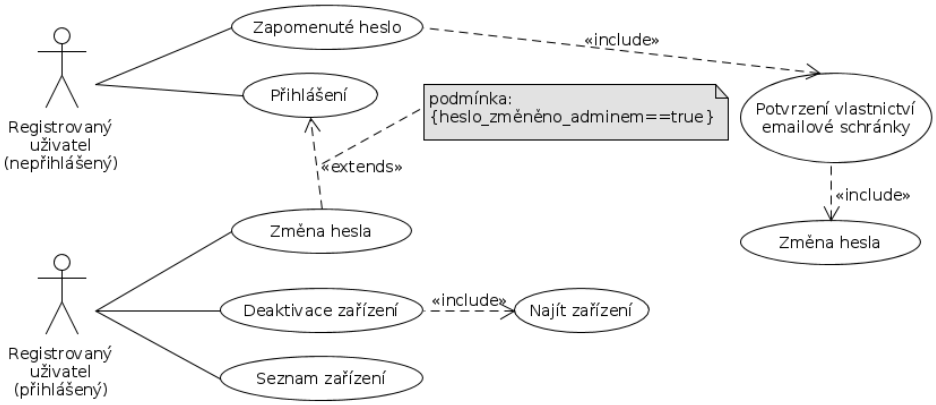

Obrázek 2.1: Anonymní uživatel

#### 2.3.1.2 Registrovaný uživatel

Uživatel, který je již v systému registrován, avšak není schopen zvládnout proces autentizace, jelikož zapomněl heslo, má zájem o vyvolání procesu zapomenutého hesla. Samotný proces vyvolá ověřením své emailové adresy tak, že mu na ni dorazí email s unikátním ověřovacím klíčem. Po zadání tohoto klíče dochází k výzvě na zadání nového hesla.

Druhou žádanou činností je zobrazení seznamu přihlášených zařízení, kterým může jednotlivé tokeny odebrat, čímž dojde k jejich odhlášení bez nutnosti jejich fyzické přítomnosti.

Přihlášený uživatel se bojí o bezpečnost hesla (například z důvodu prozrazení), v jeho zájmu tak vystává požadavek na jeho změnu.




Obrázek 2.2: Registrovaný uživatel

#### 2.3.1.3 Správce systému

Správce systému, či jiný uživatel s obdobnou úrovní autorizace je kontaktován uživatelem stránek s prosbou o změnu hesla. Správce systému tak vyvolává proces zapomenutého hesla s emailovým potvrzením libovolnému uživateli systému. Přímá změna hesla bude podmíněna brzkou expirací a vynucením automatické změny - zde vyházíme z praktické zkušenosti, kdy většina správců systému zadává při této žádosti všem stejná hesla, z čehož se stává bezpečnostní problém.

Tato role může být také kontaktována se žádostí o zablokování zařízení například při jeho ztrátě. Správce má tedy přehled o všech zařízeních uživatele s možností odebrání libovolných tokenů.


Obrázek 2.3: Správce systému

### 2.3.2 Scénáře

U nejdůležitějších případů užití probereme jejich scénáře tak, aby nedošlo k opomenutí některého z nezbytných kroků. V jednotlivých bodech jsou popsány interakce mezi aktérem a systémem. V tomto základním toku neřešíme možné chyby, předpokládá se zde zcela bezchybný průběh, který vždy vede k úspěšnému splnění cíle.

Alternativní scénáře, jako například nevalidní vstup, nebo špatný ověřovací klíč, povedou vždy k chybě. Tato chyba ukončí scénář a vyzve uživatele ke zopakování operace se správnými údaji. Modelování alternativních scénářů tak pro svoji jednoduchost zde nebudeme uvádět.

#### Registrace

1. Anonym zasílá příkaz pro vytvoření nového účtu
    - obsahující seznam vstupních parametrů, tedy požadované uživatelské atributy
2. Systém generuje uživatelský záznam a ukládá ho jako uzamčený
3. Novému uživateli je zaslán email s jedinečným potvrzovacím klíčem (s omezenou životností)
4. Nepřihlášený uživatel zasílá ověřovací příkaz společně s potvrzovacím klíčem
5. Systém povoluje účet uživatele (uživatel se může přihlásit)

#### Zapomenuté heslo

1. Nepřihlášený uživatel zasílá příkaz s žádostí o resetování hesla
2. Uživateli je zaslán email s jedinečným potvrzovacím klíčem (s omezenou životností)
3. Nepřihlášený uživatel zasílá příkaz s novým heslem a jedinečným klíčem
4. Systém mění heslo

#### Změna hesla

- Uživatelem
    1. Uživatel se přihlásí do systému
    2. Provede změnu hesla

- Správcem
  1. Správce nalezne uživatele, kterému chce změnit heslo
  2. Provede změnu hesla
  3. Jakmile dojde k přihlášení tohoto uživatele, systém vynutí proces změny hesla

#### Deaktivace zařízení

- Uživatelem
  1. Uživatel zobrazí seznam zařízení
  2. V seznamu nalezne příslušné zařízení, které chce deaktivovat
  3. Zasílá příkaz pro deaktivaci zařízení - systém odebírá zařízení
- Správcem
  1. Správce si zobrazí seznam uživatelů, kde najde příslušný záznam
  2. Zobrazí si všechna uživatelova přihlášená zařízení
  3. Zasílá příkaz pro deaktivaci - systém odebírá zařízení

### 2.3.3 Splnění požadavků

Aby nebylo nic opomenuto v samotném návrhu, je nutné zkontrolovat, zda připravené scénáře a případy užití pokrývají kompletní výčet funkčních požadavků. Dle tabulky 2.2 lze konstatovat, že připravené scénáře odpovídají zadání.

| | **Případy užití** | | | |
|---|---|---|---|---|
| **Požadavky** | Registrace | Zapomenuté heslo | Změna hesla | Deaktivace zařízení |
| 2.2.1.1 Registrace | x | | | |
| 2.2.1.2 Zap. heslo | | x | x | |
| 2.2.1.3 Správa zařízení | | | | x |

Tabulka 2.2: Kontrola splnění všech požadavků

## 2.4 Závěr analýzy

Podívali jsme se na rešerši problému v jiných technologií. Tato rešerše nám poslouží společně s průzkumem naší technologie při návrhu. Vydefinovali jsme podle zadání a běžných způsobů řešení problematiky seznam funkčních požadavků. Vytvořili jsme tak případy užití, seznamy účastníků a jim různé scénáře.

V závěru jsme si ověřili, že všechny naše případy užití pokrývají zadané požadavky.

V další kapitole vytvoříme vhodnou architekturu, která s ohledem na scénáře připraví vhodné základy tvořené aplikace. Kromě výčtu funkčních požadavků nesmíme opomenout i výčet nefunkčních požadavků, kde nám důraz na rozšířitelnost vnáší nutnost vytvoření konfigurační třídy v terminologii Springu konfiguračního adaptéru, držícího sérii konfigurovatelných endpointů sloužících pro plnění scénářů.

Takové třídy následně musí volat interface generující komunikační stránku. Například HTML šablony, nebo REST rozhraní. Nutné také bude správné využití zapouzdření pro generování emailových zpráv, kdy samotné nasazení na různých produkčních prostředí, může znamenat zcela rozdílné konfigurace a také je třeba myslet na rozdílné požadavky kladené na vývojáře ve vizuální a obsahové podobě zasílaných emailů.

Kapitola 3

# Návrh a implementace

Další významnou částí celé práce je problematika správného návrhu. Na základě analýzy vzniklo několik pracovních verzí, než se podařilo zformulovat tu správnou. Při návrhu bylo nutné hledět na několik faktorů, které spolu přinášely mnoho komplikací. Faktory, na které byl brán největší ohled:

- Přehlednost a srozumitelnost (pouhé JavaDoc nestačí)
- Znovupoužitelnost celé knihovny
- Standardy a zvyklosti Javy a samotného Springu
- Znovupoužitelnost vnitřních tříd

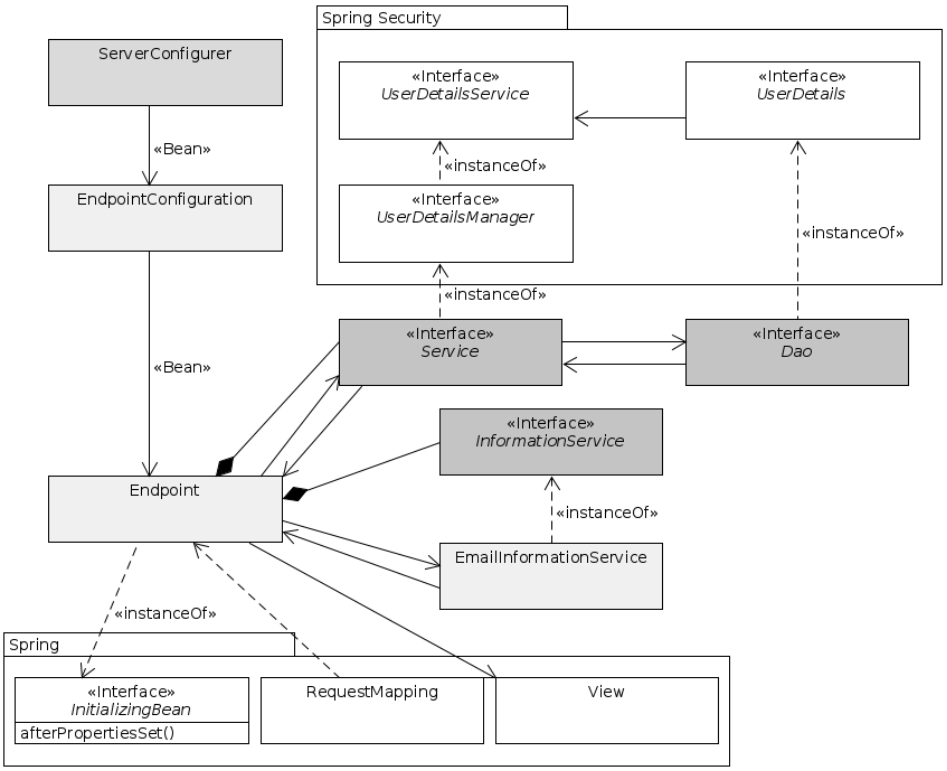
## 3.1 Architektura

Pro zajištění celkové přehlednosti a srozumitelnosti muselo být voleno mezi běžnými architekturami. Mezi takovou architekturou, která kromě své přehlednosti okamžitě vývojáři napoví, co v jaké části nalezne, co kam má případně vepsat, aby pozměnil její funkcionalitu a nenarušil části jiné. Padla tak volba na běžně využívanou architekturu MVC.

V celém konceptu byla složka *controlleru* zvolena jako stěžejní a označena tedy jako tzv. *endpoint*. Tato složka drží reference na rozhraní reprezentující *model*. Samotný *controller* je volán prostřednictvím servlet mappingu, kdy prostřednictvím abstrakce mapuje vstupní parametry do instance třídy reprezentující DAO. Po potřebné komunikaci s modelem je výstupem *controlleru* objekt typu *View*, jež je mapován prostřednictvím *Java Servlet* na JSP.

Takový koncept umožní uživateli krom předpřipravených objektů do celého konceptu zahrnout svoji problematiku. Prostřednictvím rozhraní modelu (respektive *services*) může kdokoliv napsat propojení s libovolným databázovým

typem, propojením s LDAPem, nebo jakkoliv představující službu, která umožní namapování na jednoduchý interface (viz. obrázek 3.1). Vzhled výstupů ovlivní pomocí vzhledové JSP stránky. Se vstupem mu pomůže *UserMessanger* reprezentující vstup, který pomocí implementace rozhraní upraví do vlastní potřeby tak, aby se na výstupu nenacházel hash hesla a další nežádoucí informace. Díky této implementaci lze ovlivnit validaci vstupních elementů. Případně vývojář může využít již předpřipravených implementací, například skrze JDBC.




Obrázek 3.1: Architektura knihovny

Z pozice vývojáře stačí pouze implementovat zmíněná rozhraní na svůj business model, díky tomu je knihovna plně konfigurovatelná. Vznikla tak otázka, jak efektivně vykonat konfiguraci. Aby byl zachován zvyk využívaný ve Springu, bylo nutné využít architektury Inversion of Control. Využijeme abstrakce třídy, která v několika metodách bude mít vstupní parametry s kontejnery nesoucí nastavení. Architektura IoC v životním cyklu před spuštěním zavolá tuto konfigurační třídu a její metody, do kterých zašle objekty s výchozím nastavením. Pomocí zavolaných metod uživatel upraví tyto přepravky k potřebnému obrazu. Vloží do nich například vlastní model a jiné třídy založené na potřebných rozhraních, se kterými naprogramovaný endpoint umí

komunikovat.

V poslední řadě při návrhu architektury bylo nutné zaměřit se na znovupoužitelnost uvedené logiky. K tomu dopomohla myšlenka, jež tvrdila, že nemusí mít zájem využívat celé spektrum funkcí, může mít zájem pouze o dílčí scénář užití. Návrh tak kopíruje tento architektonický prvek pro každý logicky oddělitelný scénář. Jednotlivé scénáře tak mohou mít vlastní konfigurační nastavení a nejsou tak navzájem úzce provázané. Vývojář implementuje pouze rozhraní, která se dílčího scénáře týkají a tak není zbytečně zatěžován logikou, kterou by nevyužil. Návrh dílčího celku pro resetování hesla v rámci navržené architektury je vyobrazen na obrázku 3.2. Došlo tak k logickému rozdělení na:

- Registrace a nastavení uživatelského účtu
- Aktivace účtu
- Problematika zapomenutého hesla

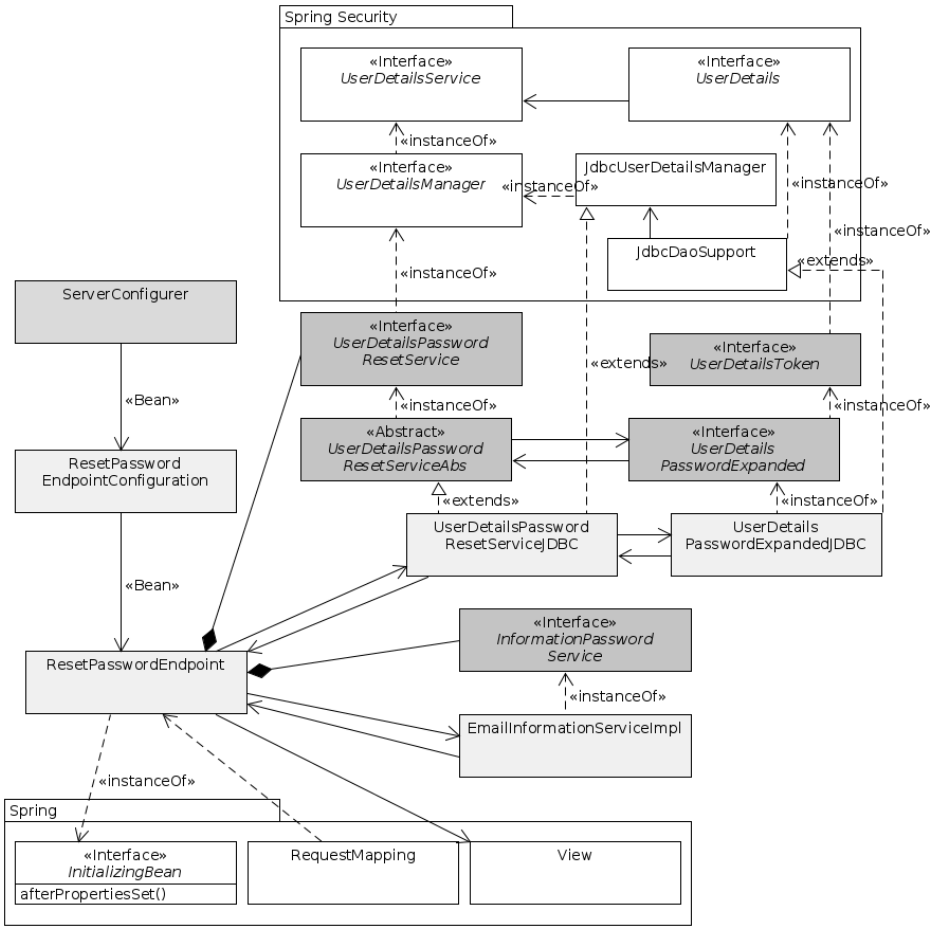
Posledním úkolem návrhu je problematika správy uživatelských zařízení. Dosavadní návrh využívá pouze knihoven Spring a Spring Security. Správa uživatelských rozhraní je závislá na knihovně Spring OAuth. Došlo tak k od dělení tohoto celku od ostatních prvků, aby vývojář nebyl nucen knihovnu Spring OAuth implementovat do svého softwaru. Implementace této funkce bude pouze rozšíření stávající knihovny. Dojde k navázání na již připravené rozhraní. Dojde k celkové abstrakci, která naváže na existující rozhraní a obohatí jej o informační hodnotu spojenou se zmiňovanými údaji uváděnými v kapitole věnující se analýze technologie 1.5.2.

## 3.2 Rozhraní

Jak již bylo zmíněno v předchozí kapitole, stavebním kamenem celé knihovny je série propojených rozhraní. Samotný controller držící naprogramovanou sekvenci korektního průchodu dílčím procesem postupně volá definice rozhraní. Tato kapitola popíše ty nejdůležitější.

### 3.2.1 ConfirmationToken

Základním principem při ztrátě hesla, případně ověření autentizace údaje (emailu, telefonního čísla, ...), je náhodně vygenerovaný klíč. Uživatel, jenž chce ověřit svůj emailový účet, musí prokázat vlastnictví emailové schránky, na kterou byl účet registrován. Do této schránky mu je zaslán klíč, který předá prostřednictvím URL do prostředí aplikace, ověří tak její vlastnictví. Takový klíč je v drtivé většině textový řetězec dostatečně dlouhý, aby nebyl uhodnutelný.




Obrázek 3.2: Architektura části ResetPassword

Některé systémy můžou ovšem vyžadovat jiný způsob ověření, nebo přímo vícefaktorový způsob. Umožnit změnu hesla pouze na základě emailu může být pro systémy s nejvyšší mírou bezpečnosti nedostačující. Při vývoji této knihovny bylo myšleno i na tento požadavek, a tak potvrzovací klíč není staven na typu *String*, nýbrž na rozhraní *ConfirmationToken*. Mezi nejdůležitější metody tohoto rozhraní patří *equals*, *random* a parsování z URL. Nesmíme opomenout její dědičnost interfacu *serializable* (například pro možnost uložení do DB bez nutnosti psaní vlastního JDBC).

Při využití serializace je důležité myslet na vlastnosti databázového systému tak, aby data byla ukládána způsobem, který umožňuje vhodné vyhledávání v databázi. Hodí se tedy jakýkoliv indexovatelný atribut v případě relačních databází.

Toto rozhraní se tedy chová jako adaptér. U vícefaktorového ověření tento

objekt může nést libovolně dlouhý znakový řetězec a k němu pole o několika číslech. Zmíněných pár čísel může být odesláno například na mobilní telefon, řetězec na email. Cílem je zkompletovat tyto dva údaje do URL, ze které se opětovně adaptér vytvoří. Kliknutí na odkaz v emailu vyvolá vývojářem napsanou vlastní routu do formuláře, kam zadá kód z mobilu, společně s *hidden* informací o klíči z emailu. Odesláním na definovanou routu se tato instance naparsuje z URL. Následně ji endpoint za pomocí *equals* porovná s informací deserializovanou z databáze. Pokud metoda *equals* potvrdí shodnost, uživateli je změněno heslo.

### 3.2.2 UserDetails

Rozhraní *UserDetails* vychází z knihovny Spring Security. Jedná se o rozhraní DAO, které slouží především pro potřeby autorizace, avšak částečně i autentizace. Toto univerzální rozhraní je přenášeno celou dobu systémem. V případě přihlášení je nejčastěji přenášeno formou *sessions*. Spring OAuth2 přenáší identifikátor tohoto objektu prostřednictvím *authorization_token*. Lze s ním pracovat prostřednictvím anotací a statických objektů, které byly popsány v kapitole 1.3.3.1.

Rozhraní umožňuje získat informace o aktivnosti účtu, uživatelském jménu, o hashi hesla, roli a právu. Jedná se pouze o definici getterů, pro náš účel je tedy nutné vytvořit i rozhraní setterů a přidat metody, které naše knihovna bude využívat pro své účely. Rozhraní tak připravuje výbornou výchozí pozici pro nadstavbu uživatelského DAO objektu.

### 3.2.3 UserDetailsToken

Vygenerovaný token je potřeba ukládat v relaci s uživatelem. Uplatňujeme tak rozhraní *UserDetails*. Pro využití této knihovny není nutné předpokládat, že by jeden účet měl mít více klíčů, jelikož lze uvažovat metodou platnosti poslední žádosti, například pokud žádáte o token k resetování hesla, určitě nebude sloužit token pro aktivaci účtu. Uživatel, který nemá aktivovaný účet, nebude resetovat heslo a uživatel, který již má aktivní účet, nebude řešit problematiku aktivace účtu. Token tedy lze ukládat pouze do jednoho sloupce databázové tabulky, respektive jako jeden atribut.

Rozhraní tak implementuje metody pro ověření, zda uživatel má nějaký token přidělen, pro uchování a jeho získání. Samotné uložení je řešeno na straně servisy, která také díky specifickému rozhraní umožňuje implementovat více tabulkové provedení, provedené operací JOIN. Takový by byl postup zajímavý v případě, že by někdo chtěl povolit více aktivních klíčů.

### 3.2.4 InformationService

Předchozí rozhraní vytvořila nový problém. Sekce tedy otevírá problém zasílání emailů. Vznikla tak nová problematika zaslání potvrzovacího tokenu.

Schopnosti, které sebou přináší rozhraní, *ConfirmationToken*, otevírají otázku řešení zasílání potvrzovacích kódů. Pokud by všechny kódy přišly na jeden a ten samý email, ověření by bylo vázáno čistě na emailovou adresu a jakékoliv zkomplikování takového klíče by do určité míry ztrácelo svůj potenciál. V knihovně tak muselo zákonitě vzniknout místo pro komponentu řešící problematiku zaslání ověřovacího klíče.

Došlo k vytvoření nové servisy, která definuje základní úkony pro zaslání autorizačních informací dle povahy dílčí části. Tvůrce, který chce zasílat klíč pro resetování hesla například na mobil, tak implementuje nad rozhraním *InformationPasswordService* metodu *sendResetPassword*. Jejím vstupním parametrem je rozhraní postavené na *UserDetailsToken*. Toto rozhraní se výborně hodí pro vygenerování ať už emailu obsahující veškeré uživatelské náležitosti, či odeslání zprávy, tak i samotný objekt *ConfirmationToken*, za jehož pomoci si vývojář vygeneruje potvrzovací URL, obsah SMS, či jakýkoliv jiný způsob ověření, který považuje za dostatečně důvěryhodný.

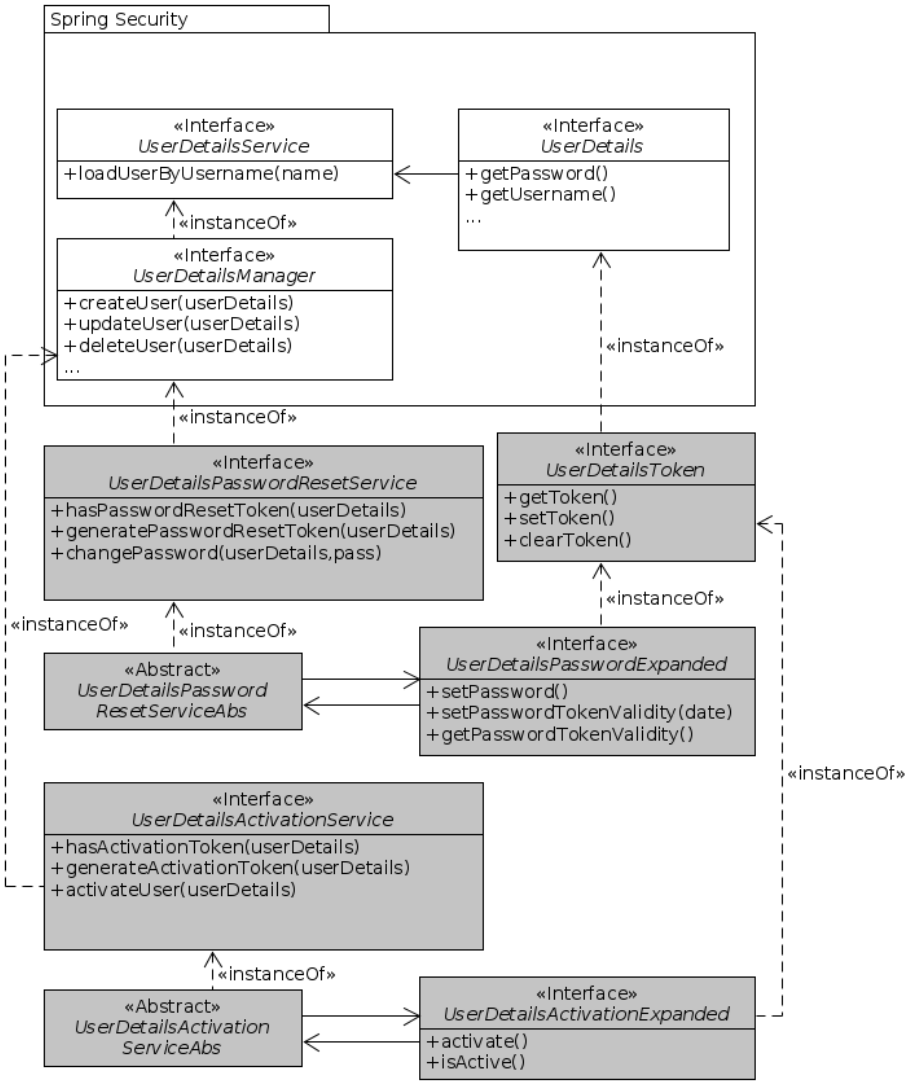
### 3.2.5 UserDetailsService

UserDetailsService je jedno z nejdůležitějších rozhraní celé knihovny. Spring Security pomocí tohoto iterfacu získává DAO objekt *UserDetails* na základě uživatelského jména (případně například emailové adresy). Proces autentizace srovná hash zaslaného hesla s hashem získaným z DAO objektu. Pokud se hesla shodují, je uživatel autentizován.

Podobným způsobem bude fungovat dědičné rozhraní této knihovny. Na základě již implementované knihovny získáme uživatele podle předaného uživatelského jména, následně endpoint porovná metodou equals navrácené tokeny. Pokud budou tokeny shodné, zavolá endpoint příslušnou metodu na rozhraní servisy, například pro změnu hesla, nebo aktivaci účtu.

Taková rozhraní, jakým je například *UserDetailsPassword*, vychází z rozhraní *UserDetailsManager*, které je také definováno v jádru Spring Security. *UserDetailsManager* vycházející z *UserService* tak dává možnost kompletní CRUD operace nad uživatelským objektem. Přidané objekty v poděděných rozhraních této knihovny tak nabízejí pouze možnosti složitější implementace práce s potvrzovacími klíči.

Vývojář vlastní implementací tohoto servisního objektu jednoduše napojí celou knihovnu na libovolný databázový systém. Takové napojení na Hibernate se jeví jako velmi jednoduché řešení. Celé propojení spočívá pouze v definování *UserDetails* DAO objektu uživatelského učtu anotací *@Entity* (samozřejmě s nutností vytvoření příslušných atributů a definování getterů a setterů), s využití metod merge, persist u *EntityManager*u v *UserDetailsService*.




Obrázek 3.3: Ukázka provázanosti se Spring Security

### 3.2.6 UserMessanger

V případě, že vývojář využívá vlastní implementace u takového Hibernate řešení, metoda merge způsobí průnik atributů, přičemž nové přepíší ty stávající. Takové řešení ovšem nemusí být zcela bezpečné. Pokud by útočník modifikoval vstupní požadavek a vložil by do něj nežádoucí atribut, mohl by ho propašovat až do výsledné databáze. Aby se předešlo této nebezpečné vlastnosti, bylo vytvořeno rozhraní návrhového vzoru typu messanger.

Cílem tohoto rozhraní je vytvoření specifického objektu reprezentujícího

DAO uživatelský účet přijímaný uživatelovým requestem. Nejdůležitější metodou je metoda *convert*, která z messangeru získá *UserDetails*. Vývojář tak má plně v rukou možnost odstínění vstupních request parametrů od dat propisovaných do databáze. Stejně tak mu je nabídnuto jednoduché řešení problému s podobou přijímaných požadavků. Ač nestandardní požadavek na přizpůsobení rozhraní, může být i s použitím této knihovny velmi lehce splněn.

## 3.3 Návrhové třídy

Nad všemi rozhraními uvedenými v předchozí kapitole byla provedena implementace s použitím JDBC. Vývojáři mohou využít těchto připravených tříd, nebo si vytvořit třídy vlastní nad svými databázovými systémy.

Nachází se zde ovšem třídy, které mají svoji implementaci pevně danou. Jedná se o třídy, jaké v MVC architektuře představují především controller. Implementace těchto tříd stanovuje samotné procesy, které musí proběhnout pro splnění cíle případů užití popsaných v kapitole 2.3. K takovým nejtypičtějším třídám patří například Endpoint.

### 3.3.1 Endpoint

Samotné vyvolání endpointu je zajištěno anotací *@Controller*. Využitím této anotace dochází uvnitř knihovny Spring k tvorbě map. Tyto mapy jsou prostřednictvím *DispatcherServlet* načítány. Dle konfigurace vycházející z anotování metod pomocí *@RequestMapping* dochází k volání specifických metod.

Každý případ užití bude specifikován několika metodami anotovanými skrze *@RequestMapping*. Podoba anotací se bude řešit především parametry anotace - *produce*, *consume*. Pro každý proces uvnitř scénáře budou napsány alespoň dvě routy. Jakmile zařízení klienta bude v hlavičce požadavku uvádět „*Accept: text/html*“, bude volána metoda s nastavením „*consume="text/html"*“. Výstupem takové metody bude objekt typu View (případně ModelAndView). Vstupní parametry načteme pomocí *@RequestParam*.

Pokud dotaz bude typu „*Accept: application/xml*“, „*Accept: application/json*“ nebo „*Accept: application/hal+json*“, uživatel k těmto metodám přistupuje REST přístupem. Pro načtení vstupů využijeme *@Valid @RequestBody*, čímž použijeme vnitřních komponent pro automatické provedení validace a parsování vstupní informace. Výstup této třídy bude vždy *ResponseEntity*, která náš výstupní objekt převede do XML, nebo JSON formátu.

Bohužel toto řešení zapříčinilo jeden problém, který by vývojářům využívající tuto knihovnu mohl velmi vadit. Co když připravené url adresy vývojář, který chce použít knihovnu, využívá k jiným potřebám, nebo je z nějakého jiného důvodu chce přepsat na vlastní? Zde se práce inspirovala řešením tohoto problému v knihovně Spring OAuth. Knihovna zde zadávání speciální anotaci *@FrameworkEndpoint*. Tato anotace přiřazuje objekt *FrameworkEndpointHandlerMapping*, který dědí z *RequestMappingHandlerMapping* a obo-

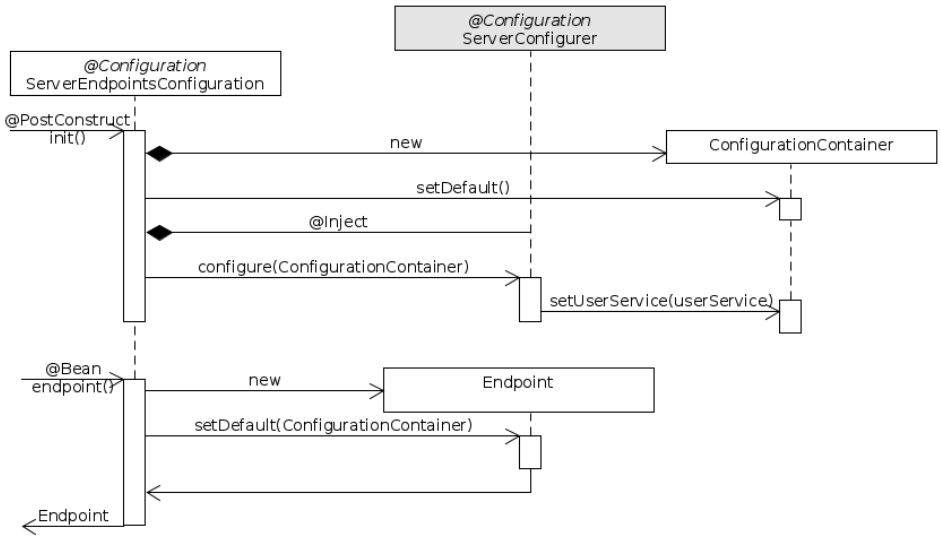
hacuje tak tuto anotaci o pomocné public metody. Pomocí těchto metod je možné dodatečně, v konfiguraci aplikace, tyto url adresy metod měnit. Vývojář tak při konfiguraci celého prostředí má možnost předefinovat adresy endpointů na adresy vlastní.

## 3.4 Komunikace tříd

Při tvorbě knihovny bylo nutné se zaměřit i na správnou komunikaci mezi jednotlivými třídami. Následující podkapitoly pojednávají o komunikačních případech, které se vyskytly v jednotlivých částech knihovny. Komunikace byly významně ovlivněny využitou architekturou daných částí systému. Níže bude zmíněna architektura, její způsob komunikace a následně demonstrace na specifickém případu uvnitř knihovny. Pro zjednodušení nebude uváděn kód, ale dojde k popisu zjednodušeného grafického vyobrazení pomocí sekvenčních diagramů.

### 3.4.1 Konfigurovatelnost knihovny

Obtížnější částí práce je správný pohled na architektonický vzor Inversion of Control. Bylo nutné vytvořit takové rozhraní, které připraví výchozí stav a prostřednictvím abstraktního konfiguračního objektu umožní vývojáři vkládat své implementace jednotlivých částí do systému knihovny.




Obrázek 3.4: Sekvenční diagram konfigurace knihovny

Knihovna využívá anotace *@Configuration*, díky které dochází k vytvoření *ServerEndpointConfiguration*. Právě ta za pomocí *@Autowired* anotace načítá

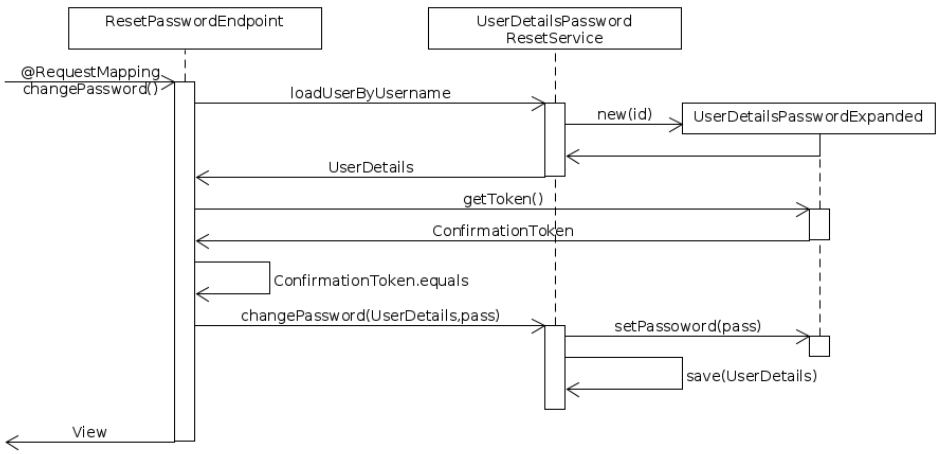
všechny instance tříd *ServerConfigurer*, které jsou vytvářeny vývojářem. *@PostConstruct* nám ve třídě *ServerEndpointConfiguration* zařídí možnost zavolání vývojářovy konfigurační třídy v životním cyklu před spuštěním samotné aplikace ke konci procesu inicializace aplikace (viz. 1.2.1.1). Do této vývojářem vytvořené konfigurační třídy pošleme *ConfigurationContainer* s výchozím nastavením. Tento kontejner drží nejdůležitější reference, třeba interface modelu. Vývojář tak vloží svoji vlastní servisu, nebo využije metody, které vytvoří předpřipravené servisy.

Sekvence těchto volání je vyobrazena na sekvenčním diagramu 3.4, kde je světle šedivou barvou vývojářem psaný objekt, ostatní jsou třídy součástí knihovny.

### 3.4.2 Endpoint a Servisy

Hlavním architektonickým vzorem využitým v práci je architektura MVC. Jak již bylo zmíněno na samotném začátku této kapitoly, bude se jednat o komunikaci mezi endpointem a servisou. Spring nám výrazně ulehčil komunikaci mezi endpointem a částí view, jelikož tato část je knihovnou Spring MVC vyřešena. Dostatečná abstrakce stačí, aby endpoint vracel objety typu *View*, *ModelAndView*, *ResponseEntity* a další.

Dle příslušných anotací *@RequestMapping* dochází k běhu specifických metod. Metody následně obsahují přesný postup, který volá nad nakonfigurovaným rozhraním metody příslušný servis. Metody tak postupně provádějí žádoucí úkol. Samotná sekvence je prováděna na základě běžných standardů vypozorovaných z jiných řešení, také popsaných v kapitole rešerše 2.1.




Obrázek 3.5: Sekvenční diagram resetování hesla

Na obrázku 3.5 je vyobrazen diagram průběhu resetování hesla, ve chvíli

kdy již uživatel obdržel například prostřednictvím své emailové schránky ověřovací token, endpoint se prvně pokusí získat specifického uživatele podle unikátního výrazu (uživatelské jméno, nebo mnohdy emailová adresa). Výstupem servisy je DAO objekt UserDetails představující uživatelský záznam. Tento objekt by měl být založen na interfacu *UserDetailsPasswordExpanded*, který zároveň obsahuje informaci o potvrzovacím tokenu. Endopoint následně srovná, zda se potvrzovací klíč shoduje s tím, který je uložen v databázi. Pokud ano, dojde ke změně hesla a k uložení informací do databáze. Během ukládání dochází k zneplatnění původního tokenu.

## 3.5 Závěr kapitoly

První částí výsledku implementace je knihovna, která vývojářům pomůže rychle implementovat řešení problematiky spojené s řešením uživatelských účtů uvnitř jejich aplikace. Úspěšně došlo ke splnění generalizace celého řešení, což vývojářům umožňuje napojení na prakticky jakýkoliv databázový systém, libovolný doménový návrh. Celá knihovna je velmi jednoduše implementovatelná a konfigurovatelná za pomocí jednoduchého kódu. Její nasazení by tak nemělo být problematické ani pro začínající programátory, nebo programátory neznající problematiku frameworku Spring.

Druhá část řeší správu zařízení přihlášených pomocí protokolu OAuth 2.0. Tato část je ukázkou vlastní implementace rozhraní definovaných knihovnou Spring OAuth nad knihovnou Hibernate. Vývojáři se tak mohou touto knihovnou inspirovat, případně využít připraveného způsobu implementace ve svých aplikacích.

Vzhledem k tomu, že se využití knihovny v jiných aplikacích stará o bezpečnostní problematiku uživatelských účtů, pro praktickou použitelnost je nutné zvýšit její důvěryhodnost. Pro zvýšení bezpečnosti je nutná tvorba alespoň základních testů tak, aby knihovna nevykazovala nepředvídatelné chování a její údržba byla dostatečně konzistentní. Mimo jiné kód knihovny bude veřejně publikován. To povede k vyšší důvěryhodnosti, k opravě chyb a pakliže bude knihovna přijata komunitou, tak dojde k jejímu dalšímu vývoji. Takové problematice je věnována další kapitola této diplomové práce.

Kapitola 4

# Testování

Nedílnou součástí softwarového vývojového procesu je testování. Tato součást je mnohdy opomíjena z časových důvodů. V případě, že se samotný vývoj protáhne, spoustu projektových manažerů je nuceno spořit čas na této položce. Mnohdy dochází k vypuštění této části i z finančních důvodů. Praxe ovšem ukazuje, že následné náklady na odstraňování chyb, způsobené nedostatečným otestování, mnohou být několikanásobně vyšší, než náklady spojené s testováním. Již jsme se v historii několikrát setkali s chybou, která vedla k úniku citlivých informací, například čísla kreditních karet. Jedná se o častý způsob jak služby přijdou o důvěru uživatelů. Při přípravě testů je nutné, aby typy testů byly připraveny tak, aby produkt neobsahoval skryté bezpečnostní, architektonické nebo jiné nedostatky.

Jedním z hlavních požadavků vedoucího práce byla veřejná publikace. Pro celkové uveřejnění knihovny je nutné se přesvědčit, že knihovna neobsahuje žádné chyby, případně pouze jejich zanedbatelné množství. V případě velkého množství chyb by byla publikace zmařena hned na samém počátku, jelikož s prvotním dojmem, vytvořeným na negativních reakcí, by žádné nové vývojáře ke svému využívání nezískala. V této kapitole projdeme několik standardních typů testů, které bylo na základě možných dostupných zdrojů provést.

## 4.1 Assembly test

V průběhu vývoje celé této práce byly prováděny tzv. „assembly test“. Během tvorby docházelo k důkladnému prověřování zdrojového kódu a výstupní knihovny. Standardně se tento test provádí kontrolou jiným programátorem, bohužel nyní je knihovna pro splnění prohlášení vyvíjena pouze mou osobou. Docházelo k postupnému uvolňování knihovny, prozatím interně mezi přáteli, kteří ve Springu vyvíjejí. Byla tak získána poměrně silná zpětná vazba. Některé požadavky se stihly promítnout v této práci, další se promítnou časem po uveřejnění a dalším postupném vývoji.

Toto základní testování dokázalo podchytit spíše problémy s dostatečnou úrovní generalizace. Samotné chyby toto rychlé nasazování neodhalilo. Bylo tedy nutné v průběhu k jednotlivým částem knihovny psát testy jednotlivých jednotek.

## 4.2 JUnit

Po vytvoření dílčích částí celého kódu a jejich ověření, přišly na řadu testy jednotek. U většiny jednotek tak tyto testy hodnotí jednotlivé metody jejich objektů. Jak již název napovídá, jedná se o testy, které slouží k testování menších částí zdrojového kódu. Jako jednotku si lze představit dílčí část rozebíranou v kapitole 3 Návrh a implementace.

Správný postup je takový, že vývojář bezprostředně po napsání kódu metody píše test. Procesní přístup pojmenovaný jako „Test Driven Development" dokonce zavádí psaní testů na základě rozhraní ještě před tvorbou kódu samotné metody. Při testování by nemělo být opomíjeno testování mezních hodnot, například zaslání vstupů s hodnotou null, nebo čísla s minimální / maximální typovou hodnotou.

### 4.2.1 Metody equals a hashcode

Důležitými metodami v aplikaci psané v jazyce Java jsou implementace metod *equals* a *hashcode*. Tyto metody jsou využívány například pro porovnávání *ConfirmationToken*. Kvůli bezpečnosti v našem systému tak muselo dojít k jejich řádnému otestování.

Správná funkcionalita metody equals musí splňovat:

- Podmínky ekvivalence
  - Reflexivita — pro jakýkoliv objekt a, kde a != null platí že metoda a.equals(a) vrací true
  - Tranzitivita — pro jakýkoliv objekty a,b,c kde a != null, b != null a c !=null, pokud metoda a.equals(b) je true a b.equals(c) je true, potom a.equals(c) je také true
  - Symetrie — pro jakýkoliv objekty a,b kde a != null a b != null, metoda a.equals(b) je true pouze v případě, že b.equals(a) je také true
- Konzistentnost — pokud nedojde ke změně hodnot pak metoda a.equals(b) vrací v průběhu programu konzistentně true nebo false
- V případě, že a != null pak metoda a.equals(null) vrací false

Správná funkcionalita metody hashcode musí splňovat:

- Konzistentnost — pokud nedojde ke změně hodnot pak metoda a.hashcode() vrací stejnou hodnotu
- V případě, že equals dvou proměnných a.equals(b) vrací true, potom a.hashcode() a b.hashcode() musí vracet stejnou hodnotu

### 4.2.2 Mockupy

Mockupy jsou objekty, které se snaží simulovat chování objektů, které napodobují. Tento typ objektů se využívá v případě, že potřebujeme nahradit skutečný objekt, který by se neprakticky začleňoval do testování. Za pomocí mockupů lze kontrolovat chování spojení s testovaným objektem bez nutnosti jeho samotné implementace.

V našem případě mockup objekty využijeme při testování našich endpointů, kdy bez nutnosti implementace servis ověříme, zda ze strany endpointu dochází ke správné komunikaci vůči rozhraní servisy.

Mockup testům je ve framework Spring věnován poměrně velký prostor a pro tento účel je vyhrazena speciální část knihovny *org.springframework.test*. Kromě standardních mockupů nabízí i testy pro aplikace typu MVC v úrovni srovnávacích metod View objektů, takovým konkrétním může být srovnávač *ModelAndViewAssert*. Nachází se zde i testovací mockup pro rozhraní JNDI, který implementuje vrstvu, prostřednictvím které je možné testovat část implementace přes Hibernate nebo JDBC.

### 4.2.3 Metriky

U tohoto typu testů jsou dvě základní metriky - úspěšnost a code coverage. Úspěšnost napsaných testů byla pochopitelně 100%, jinak by knihovna byla považována za nefunkční. Hodnota code coverage udává míru pokrytí testovaného kódu. Aplikace s vysokou mírou pokrytí má vyšší pravděpodobnost, že neobsahuje žádné další chyby.

Bohužel kvůli vysoké náročnosti při psaní testů dochází v tuto chvíli pouze k nízké míře pokrytí, otestujeme pouze stěžejní části. Nedošlo k testování propisování implementace skrze JDBC do databáze. Stejně tak nedošlo k testování ukázkových uživatelských prostředí, především v části práce, která se věnuje správě uživatelských zařízení. Díky tomu metrika code coverage nedosahuje vysokých hodnot.

### 4.2.4 Testované jednotky

V následující tabulce 4.1 je stručný soupis testovaných jednotek společně s výsledky.

| Třída | Využití mockupu | Počet testů | Výsledek |
|---|---|---|---|
| ConfirmationTokenImpl | ✗ | 4 | 100% |
| UserRestDetailsImpl | ✓ | 3 | 100% |
| ActivationAccountEndpoint | ✓ | 2 | 100% |
| ResetPasswordEndpoint | ✓ | 2 | 100% |
| RegistrationAccountEndpoint | ✓ | 2 | 100% |
| UserDetailsActivationServiceAbs | ✓ | 2 | 100% |
| UserDetailsPasswordResetServiceAbs | ✓ | 2 | 100% |
| TokenDeviceStore | ✓ | 3 | 100% |

Tabulka 4.1: Výsledky jednotkových testů

I přes vytvořené testy jednotek se dá předpokládat, že knihovna v takto rané fázi obsahuje skryté chyby. Ač byla provedena důkladná rešerše, může zde existovat nějaký problém v celém business konceptu vytvořené knihovny, či skrytá chyba. S tímto problémem se pokusím vypořádat uveřejněním zdrojových kódů. Mělo by tak dojít k postupnému odladění všech chyb a dalšímu vylepšování této knihovny. Tento krok je velmi důležitý, jelikož tyto skryté chyby by v budoucnu mohly znamenat i bezpečnostní riziko.

## 4.3 Akceptační testy

| Požadavek | Stav |
|---|---|
| **Registrace** | kompletně splněno |
| rozhraní pro vytvoření nového uživatele | ✓ |
| potvrzení uživatelského emailu | ✓ |
| **Zapomenuté heslo** | kompletně splněno |
| unikátní klíč pro resetování hesla | ✓ |
| časově omezená platnost klíče a resetování hesla | ✓ |
| **Správa zařízení** | splněno jako ukázkový kód |
| seznam uživatelů | ✓ |
| výpis přidělených tokenů | ✓ |
| zneplatnění vybraných klíčů | ✓ |
| | ✓ |
| další informace u klíčů | jako implementace nad rozhraním |

Tabulka 4.2: Tabulka akceptačních testů

Po řádném testu jednotek, který prošel se 100% úspěšností, přichází na řadu akceptační testy. Akceptační testy jsou běžně prováděny na straně zákazníka. Obvykle jsou tyto testy prováděny zákazníkovým týmem testerů, který testuje aplikaci podle připravených scénářů. Tyto scénáře vycházejí ze zadání a jsou diskutovány i s dodavatelem.

Pro případ této práce vytvořím seznam funkčních požadavků a pokusím se vyhodnotit, zda došlo k jejich splnění. Vyhodnocení testu je znázorněno tabulkou 4.2. Z této tabulky je patrné, že zadání bylo splněno s jedním drobným nedostatkem. Nepodařilo se zařídit dostatečnou úroveň generalizace správce zařízení. Kód tak pravděpodobně nebude ostatními vývojáři využíván v jejich aplikacích a bude tak pouze sloužit jako inspirativní způsob řešení problému. Pokud se tak stane, tak i přes absenci dostatečné abstraktnosti problému, lze úkol považovat za splněný.

## 4.4 Souhrn výsledků testů

Na základě zpětné vazby od oslovených vývojářů, 100% splnění napsaných jednotkových testů a dle seznamu funkčních požadavků lze usuzovat, že je knihovna připravena na svoji publikační fázi. Na tomto základě je patrné, že neobsahuje žádné kritické chyby, které by ji mohly poškodit po uveřejnění.

Kapitola 5

# Publikace

Tato kapitola je věnována veřejné publikaci vyvinuté knihovny. Bude se jednat pouze o plán, který bude v průběhu následujících měsíců realizován snad s co nejmenšími změnami. Na publikační výsledky se tak může těšit až odborná komise, před kterou bude probíhat obhajoba samotné práce. Další možností je vyhledání knihovny a případné výsledky si odvodit vlastní cestou. S publikací knihovny úzce souvisí i ukázka jednoduchost jejího nasazení na libovolný doménový model. Jednoduchost nasazení lze považovat za přednost celé práce, proto ji věnujeme poslední podkapitolu.

## 5.1 Licence

Používání a rozšiřování softwaru s sebou přináší také nutnost zabývat se tím, co se s ním vůbec smí dělat. Tyto věci řeší licence a licenční smlouvy. Je dobré správně pochopit jejich podstatu, aby sloužily ku prospěchu všem — tedy jak uživatelům, tak autorům [42]. Nezbytnou součástí kapitoly se tak stává volba vhodné licence.

Vzhledem k aspektům popsaných úvodem celé kapitoly, můžeme vytyčit několik kritérií, které by licence měla splňovat:

- veřejné zdrojové kódy
- možnost bezplatného využití knihovny
- umožnit vylepšování, upravování a šíření jakýchkoliv kopií

Největší otázkou se tak stává způsob náhledu na copyleft. Copyleft je zvláštní použití autorského práva. Při vytvoření odvozeného díla z díla, jež je dostupné jen pod copyleft licencí, musí být toto odvozené dílo nabízeno pod stejnou (copyleft) licencí jako dílo původní [43].

Využití copyleftové licence by u svobodného díla omezilo použitelnost opět pouze na díla, která budou volně dostupná. Nemusíme být zarytými zastánci

ryze svobodného softwaru, komerční složku tedy považujme za důležitou součást celého historického IT vývoje. Mimo jiné, pokud umožníme dílo využívat i v komerčně založených aplikacích, určitě tak podpoříme velikost případné komunity.

Rozhodnutí tak padá na některou z permisivních typů licencí. Tyto licence umožňují vývojářům s kódem dělat skoro cokoliv, mimo jiné upravený kód klidně uzavřít. Za největší výhodu lze považovat využitelnost i v takových projektech, ve kterých nelze vyhovět copyleftovou licencí. Bohužel nezaručí stálou softwarovou svobodu. Může se tak stát, že někdo objeví chybu a její opravu nezveřejní nebo zveřejní pouze za úplatek.

### 5.1.1 Licence Apache

Jedná se o poměrně starý typ licence, který je tvořen poměrně rozsáhlým obsahem. Díky velkému obsahu přináší nestravitelně složitý text, kterého se velké množství tvůrců bojí, jelikož obtížné právní definice jsou pro běžné vývojáře příliš nečitelné. Navíc zde existuje určitá pasáž o nutnosti zachování informace o Apache licenci v následující kopii, bez nutnosti držení se její platnosti.

Poměrně zajímavým bodem této licence je nutnost souboru *NOTICE*. Do tohoto souboru by měl vývojář, který využije script takto licencovaný, přidat informace o tom, že součástí je volně dostupná knihovna, přidat informaci o tom, jak ji je možné pořídit a kdo je jejím tvůrcem. Tento zajímavý fakt tak výrazně stěžuje případnou redistribuci knihovny s pouze drobnými změnami.

Této licence po důkladném prozkoumání nevyužijeme, ač je soubor *NOTICE* zajímavou vlastností, tak samotná knihovna bez dalšího business obsahu nemá smysl. Aplikace, která bude tuto knihovnu využívat, tak svoji podstatu nebude mít založenou na problematice správy uživatelských účtů, ale pravděpodobně na něčem daleko podstatnějším.

### 5.1.2 Licence BSD

BSD licence je licence pro svobodný software, která je označována jako jedna z nejsvobodnějších. Byla vyvinuta obchodní organizací při University of California, Berkeley, která tuto licenci používala pro práce nad operačním systémem BSD. Umožňuje volné šíření licencovaného obsahu, přičemž vyžaduje pouze uvedení autora a informace o licenci, spolu s upozorněním na zřeknutí se odpovědnosti za dílo [44].

Kód takto licencovaný, může být využit prakticky jakkoliv. Stěžejním bodem je původní copyright s kopií původní licence. Součástí zveřejnění autorova jména by měl být navíc dovětek, že se původní autor zříká veškeré odpovědnosti.

Avšak problém v této licenci můžeme shledat právě v propagační klauzuli. Propagační klauzule je uváděna v následujícím přeloženém znění: „Tento produkt zahrnuje software vytvořený *vlastníkem práv* a přispěvatelů“ [44]. Projekt

GNU se svojí copyleft licencí GNU GPL tvrdí, že BSD licence není kompatibilní s licencí GNU GPL právě kvůli zmíněné klauzuli.

### 5.1.3 Licence MIT

Licence, která vznikla Massachusettském technologickém institutu, mnohdy označována pod názvem **licence X11**. Jedná se opět o permisivní typ svobodné licence. Software využívající kód takové licence musí splňovat podmínku, že text licence bude dodávaný spolu s daným softwarem. Obecně se jedná o zjednodušenou podobu licence BSD, neřeší například umístění takového textu, který díky tomu může být ve výsledné aplikaci zcela schovaný. Mezi její výhody patří vlastnost, že GPL s ní povoluje explicitně kombinaci.

Díky těmto vlastnostem licenci MIT zvolíme pro vytvářenou knihovnu. Její výhodnost lze vnímat v oblasti flexibility, co se týká propojení s aplikacemi postavených na jiných licenčních ujednání. Zároveň aplikace stále ponese určitý odkaz mého jména bez toho, aniž by takové uveřejnění pro vývojáře znamenalo výraznou nepříjemnost. Souhrn těchto vlastností tedy nebude překážkou pro případný komunitní růst.

## 5.2 Git

Kromě správné volby licence bylo nutné učinit rozhodnutí v podobě způsobu publikace kódu. Vzhledem k tomu, že by kód měl být veřejně dostupný, rozhodnutí padá na dnes již nejčastější způsob publikace prostřednictvím systému pro správu kódu — Git.

Git je distribuovaný systém správy verzí vytvořený Linusem Torvaldsem, původně pro vývoj jádra Linuxu. Původně se jednalo o nízkoúrovňový základ pro vývoj různých systémů správy verzí, ale časem se Git vyvinul do samostatně použitelného systému správy verzí [45].

Mimo jiné při rozhodování ve výběru služby budu klást důraz na takový výběr, který nabízí i další podpůrné nástroje, například bug tracker, wiki a další. Mezi nejznámější služby v dnešní době patří BitBucket, GitHub, nebo vlastní řešení GitLab.

Vlastní serverové řešení v podobě GitLabu můžeme prakticky ihned vyloučil, jelikož pro dané účely není vhodné. Vytváří uzavřenou komunitu svým vlastním autentizačním a autorizačním systémem, čímž by byly přispěvatelé nuceni procházet procesem registrace, což by bylo velmi odrazující. Navíc by bylo nutné řešit problematiku instalace a údržby serveru. BitBucket je zajímavá volba řešení, avšak v opensource komunitě není tak hojně využívaný jako GitHub.

### 5.2.1 GitHub

Kromě zmíněných důvodů je tu ještě důvod komunitní. Samotný framework Spring je na tomto serveru zpravován. Díky tomu se zde nachází i velké množství dalších knihoven a to včetně Spring Security i Sprigng OAuth. Volba správy kódu tak padá na GitHub.

Tento nástroj pro správu kódu nabízí mimo jiné i nástroje podporující další rozvoj. Mezi nejzajímavější vlastnosti patří:

- automatické dokumentace v podobě *README.md* a vedení changelogu v podobě souboru *CHANGELOG.md*
- systém pro správu a sledování problémů (issue tracking)
- wiki stránky
- pull requesty, commity, diffy a grafy

### 5.2.2 GitFlow

Vzhledem k tomu, že Git nabízí celou sérii způsobů, jak řídit verzování, vzniká množství procesních cest. Jedním z nejpoužívanějších je „GitFlow“. O tomto způsobu prvně napsal jeden z Git evangelistů Scott Chacon. Hlavní body tohoto Git flow jsou [46]:

- Všechno v master větvi je vždy připraveno na deploy
- Když chcete pracovat na něčem novém, vytvořte si novou větev s popisným názvem vedle master větve
- Komitujte lokálně a a pushujte svou práci do vaší větve na serveru
- Když potřebujete zpětnou vazbu, nebo radu, nebo když si myslíte, že větev je připravena k mergi, otevřete nový pull request
- Potom, co někdo jiný zkontroloval a podepsal novou funkčnost, ji můžete včlenit do master větve
- Jakmile je včleněno a pushnuto do master větve, můžete a měli byste udělat okamžitě deploy

Prvky tohoto GitFlow budeme udržovat při dalším vývoji této knihovny.

## 5.3 Nasazení

Nezbytnou součástí publikace a také obsahem souboru *README.md* musí být návod a ukázka způsobu použití ve vlastním projektu. Podstatnou částí se tak stává ukázka kódu konfiguračního souboru a popis rozhraní, které uživatel musí implementovat v případě, že chce využít knihovnu na svém doménovém modelu.

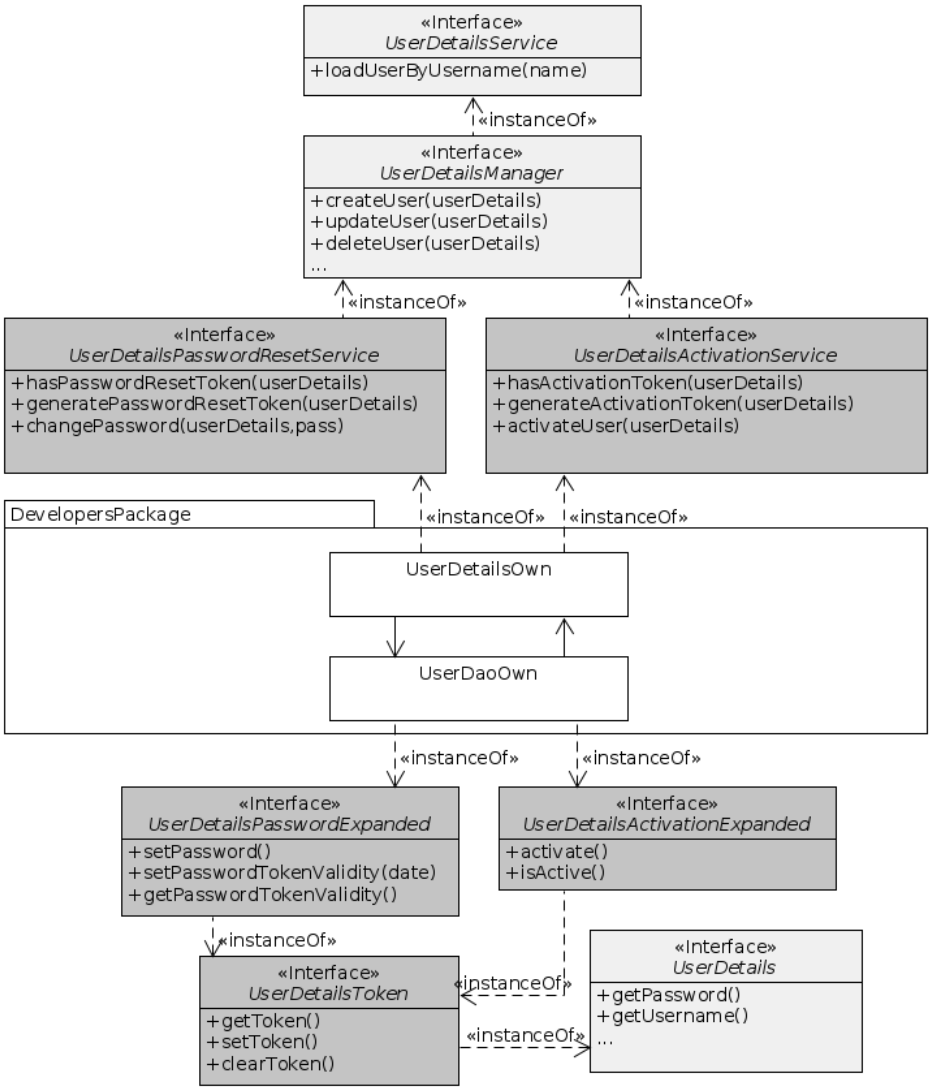
```
@Configuration
public static class UserManagmentConfiguration
        extends UserManagmentConfigurerAdapter {

    @Autowired
    private UserDetailsService userDetailsServiceOwn;


    @Override
    public void configure(
                UserManagmentEndpointsConfigurer endpnt)
                throws Exception {
        endpnt
            .userMessangerClass(UserMessangerImpl.class)
            .userDetailsService(userDetailsServiceOwn);
    }

}
```

Výše uvedený kód, je ukázkou minimální implementace s napojením na vlastní doménový model. Kromě tohoto kódu je nutné vytvořit *Bean*u, která bude inicializovat nový objekt typu *UserDetailsService*. Tento objekt musí být postaven na rozhraních uvedených na obrázku 5.1. V prostředí Hibernate by se mělo jednat o několik řádků kódu, při využití JDBC lze využít připravené ukázky a následně si ji upravit pro svůj kontext. Mimo jiné bude nutné implementovat jednoduché getter a setter metody u DAO objektu uživatele, v případě Hibernatu u uživatelské entitní třídy podle rozhraní vyobrazených na stejném obrázku.

## 5.4 Shrnutí připravených publikačních kroků

Kompletní kód včetně návodu pro vlastní implementaci bude uveřejněn a spravován v systému GitHub. Nadále bude vyvíjen, a to jak soukromým způsobem, tak i komunitním. Pro udržení použitelnosti bude spravován dle pravidel souhrnně označovaných jako Git Flow.

Pro vytvořenou knihovnu jsme vybrali licenci MIT především z důvodů otevřenosti. Knihovna tak spadá do kategorie opensource, díky čemuž bude




Obrázek 5.1: Diagram znázorňující diagram nutnosti implementace pro specifický doménový model

moci být využitelná v ostatních projektech. Jedná se o licenci permisivního typu, kterou lze využít prakticky libovolně, i v komerčních projektech. Kromě hlaviček zdrojových kódů se informace o licenčních ujednáních budou nacházet v souboru *LICENSE.md*.

Součástí zdrojového kódu umístěného na GitHubu bude soubor *README.md*, který bude obsahovat stručný návod a ukázku implementačních řešení. Mimo jiné provedeme základní konfigurační popis. Díky tomu zvládnou implementaci i začínající vývojáři.

# Závěr

Během několika let, co jsme se potkávali na Českém vysokém učení technickém v Praze, na fakultě informačních technologií, jsme se v mnoha předmětech setkával s vývojovým jazykem Java. V průběhu tohoto času si mnozí z nás jazyk poměrně oblíbili, možná se tak stalo díky srovnání s jazykem C++, s nímž jsme pracovali během předchozího bakalářského studia.

Studium bylo poměrně obecně zaměřené, avšak se i zde nacházely předměty, které se poměrně detailně věnovaly úzce zaměřeným problémům. Jedním z takových na katedře softwarového inženýrství byl předmět „Pokročilé databázové systémy“. Během semestru se nám prostřídalo hned několik učitelů, kteří se specializovali na jednotlivé technologie dle osnov. Tehdy nás poprvé vyučoval vedoucí práce Ing. Josef Pavlíček, Ph.D. Jeho tématem byla technologie ORM, kterou nám vysvětloval za pomocí knihovny Hibernate. V té době jsme se s Javou poprvé setkali v úrovni, která mnohým byla blízká. Nejednalo se o její základy, ani o komplexní enterprise systémy.

O technologii se nás spoustu začalo aktivněji zajímat. Spoustu z nás postupem času především během komerčně prováděné práce narazili na Spring, který je právě s Hibernatem velmi úzce provázaný. Technologie natolik oblíbená, že byla zvolena pro tuto diplomovou práci.

Bohužel žádné takové zadání nebylo vypsáno, a tak bylo nutné po konzultaci s několika potencionálními vedoucími dostat zadání vlastní. Následně se vedení této práce ujal pan doktor Pavlíček, což se stalo výbornou volbou, jelikož právě pan doktor byl ten, kdo nás s touto technologií alespoň okrajově seznámil. Mimo jiné jeho několikaletá praxe ve společnosti IBM znamenala výbornou kvalifikovanost pro vedení takto specifického úkolu.

Při tvorbě samotného zadání jsme se snažili kromě technologického směru klást důraz i na reálnou užitečnost. Na jiných projektech jsme se setkali s daleko robustnějšími řešeními problematiky správy uživatelů, čímž nás mohl Spring velmi překvapit, jelikož v dalších ohledech má převážně rozsáhlejší aspekty. Šel tak cítit potenciál, že by výsledná práce mohla mít reálné využití. Mimo jiné mnozí z nás tuto problematiku zrovna řeší na svém současném pro-

jektu, a tak zadání bylo výbornou volbou.

## Zhodnocení průběhu práce

Celou práci bylo nutné pojmout velmi abstraktně, tak, aby ji bylo možné reálně využít i na jiném projektu. Jakékoliv řešení, které by nebylo dostatečně , by nebylo využitelné na jiném doménovém modelu než na tom, který by práce připravila. Pak by knihovna nemohla mít dostatečný úspěch a komunitně by určitě nebyla dále vyvíjena. Díky těmto problémům jsme museli daleko více proniknout do samotných technologií, než jsme se v průběhu studia dostali.

Během rešerše technologie Hibernate jsme si zopakovali životní cyklus entitních tříd. Prošli jsme si celé rozhraní Java Persistence API, čímž jsme si vyjasnili hranici mezi rozhraním a samotným Hibernatem. Narazili jsme také na Hibernate OGM. Díky tomu byla knihovna psána proti rozhraní JPA, čímž je využitelná i v jinak orientovaných databázových systémech, než právě v relačních.

Po technologické stránce jsme se zdokonalili ve frameworku Spring, kde jsme se důkladně seznámili s provedením architektury Inversion of Control. Prozkoumali jsme API knihovny Spring Security, což pomohlo k využití některých předpřipravených rozhraní, díky čemuž je knihovna daleko snadnější při své implementaci. V neposlední řadě byl průzkum knihovny OAuth, která nás z důvodů své nedostatečné dokumentace donutila zapátrat v samotných zdrojových kódech.

Díky vlastnímu zadání a poměrně běžnému úkolu tak samotná část analýzy nebyla obtížnou úlohou. Jak již bylo výše uvedeno, s podobnými řešeními jsme se již všichni setkali a nebyl problém je důkladně projít a vzít si z nich jen perspektivní aspekty.

Zdrojové kódy knihovny Spring OAuth nám pak pomohly při volbě celé architektury a způsobu provedení. Tato část byla časově nejnáročnější. Vytvořili jsme několik konceptů, než jsme došli k aktuální podobě, díky níž je knihovnu možné využít na jakémkoliv doménovém modelu, jakémkoliv projektu a s minimálním vývojovým úsilím.

I když testy samotné knihovny nebyly napsány v takové míře, jaké by si knihovna zasloužila, bylo dbáno na zásady, které byly zmíněny v předmětu „Návrhové vzory a OOP“. V předmětu se pracovalo s mockupy a naučili jsme se pracovat s technologií JUnit, která byla následně využita v této práci. Vědomosti jsme si rozšířili o nadstavbu nacházející se v samotném Springu. Díky těmto testům se nám podařilo odladit řadu chyb, které by se jinak nacházely v odevzdávané práci.

## Publikace a další vývoj

Pro vedení této diplomové práce měl pan doktor Pavlíček mimo jiné jednu podstatnou podmínku. Chtěl, aby výsledná práce byla veřejně publikována. Samozřejmě při takto genericky postavené knihovně se publikace přímo nabízí. Bylo tím však rozšířeno zadání o problematiku s tím spojenou.

Tímto požadavkem vznikla celá nová kapitola diplomové práce, jelikož kromě technologické rešerše bylo nutné udělat i jednoduchou rešerši ohledně licenčních ujednání. Při rozhodování o volbě licenčního ujednání jsme tak využili i znalostí z bakalářského předmětu „Právo a informatika“. Díky rešerši licencí jsme získali nadhled nad rozdíly a kritérii, které se týkají autorských práv u opensource projektů. Zjistili jsme, že spousty ostatních projektů tyto licenční ujednání nedodržují správně.

## Přínos práce

Podařilo se nám vytvořit knihovnu generizující problematiku uživatelských účtů, konkrétně registrace, aktivace účtů a zapomenutého hesla. Byla také vytvořena ukázka řešení pro správu uživatelských zařízení na protokolu OAuth ve frameworku Spring. Vytvořenou knihovnu je možné spojit s libovolným doménovým modelem. Díky tomu má knihovna potenciál stát se nástrojem, který výrazně ulehčí vývoj v oblasti uživatelských účtů.

Nyní stačí doufat, že knihovna vytvořená na základě této práce bude skutečně přijata cílovou komunitou. Přijetí bude odrazem reálné použitelnosti napsaného kódu. Pokud se to podaří, lze výsledek považovat za úspěch, který mě bude motivovat v její další údržbě a vývoji.

# Literatura


[1] Pivotal Software, Inc.: *15. Expression-Based Access Control*. [tabulka, cit. 2016-06-02]. Dostupné z: `http://docs.spring.io/spring-security/site/docs/3.0.x/reference/el-access.html`

[2] Hackeři tvrdí, že se nabourali do e-mailové schránky premiéra Sobotky. *iDNES.cz [online]*, leden 2016, [cit. 2016-04-29]. Dostupné z: `http://zpravy.idnes.cz/hackeri-nabourali-e-mail-premiera-sobotky-fgd-/domaci.aspx?c=A160105_132452_domaci_fer`

[3] Princip zobecnění a abstrakce. *Živě.cz [online]*, září 2009, [cit. 2016-05-09]. Dostupné z: `http://principyprogramovani.blog.zive.cz/2009/09/princip-zobecneni-a-abstrakce/`

[4] Gutierrez, F.: *Introducing Spring Framework: A Primer*. Heinz Weinheimer, třetí vydání, ISBN 978-1-43026-532-0.

[5] Walls, C.: *Spring in Action*. Manning Publications Co., třetí vydání, ISBN 978-1-93518-235-1.

[6] Winch, R.; Mularien P.: *Spring Security 3.1*. Packt Publishing Ltd., druhé vydání, ISBN 978-1-84951-826-0.

[7] Objektově relační mapování. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2014, [cit. 2016-05-15]. Dostupné z: `https://cs.wikipedia.org/wiki/Objektově_relační_mapování`

[8] Object-relational mapping. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2016, [cit. 2016-06-01]. Dostupné z: `https://en.wikipedia.org/wiki/Object-relational_mapping`

[9] Frank nan Dong: *About / History Hibernate ORM*. Wordpress.com [online], 2016, [cit. 2016-05-25]. Dostupné z: `https://fndong.wordpress.com/2016/03/14/about-hibernate-orm/`

[10] Red Hat: *Hibernate OGM [online]*. [cit. 2016-05-25]. Dostupné z: http://hibernate.org/ogm/

[11] Oracle: *The Java EE 5 Tutorial [online]*. [obrázek, cit. 2016-05-28]. Dostupné z: http://docs.oracle.com/javaee/5/tutorial/doc/bnbqa.html

[12] Red Hat: *Interface EntityManager*. [cit. 2016-05-25]. Dostupné z: http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html

[13] The Apache Software Foundation: *Entity Lifecycle Management*. [cit. 2016-05-25]. Dostupné z: http://openjpa.apache.org/builds/1.2.3/apache-openjpa/docs/jpa_overview_em_lifecycle.html

[14] Red Hat JBoss Middleware: *HQL and JPQL*. [cit. 2016-05-26]. Dostupné z: https://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch11.html

[15] JPQL. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2013, [cit. 2016-05-26]. Dostupné z: https://cs.wikipedia.org/wiki/JPQL

[16] DeMichiel, L.: *Java Persistence 2.0 Public Draft: Criteria API*. Blogs Oracle.com [online], 2008, [cit. 2016-05-24]. Dostupné z: https://blogs.oracle.com/ldemichiel/entry/java_persistence_2_0_public1

[17] Spring Framework. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2016, [cit. 2016-05-29]. Dostupné z: https://cs.wikipedia.org/wiki/Spring_Framework

[18] Pivotal Software, Inc.: *Spring Framework*. [cit. 2016-05-29]. Dostupné z: https://projects.spring.io/spring-framework/

[19] Ramani, P.: Spring Container & Bean Life Cycle. *Piyush Ramani @ Java, Blogspot.cz [online]*, 2013, [cit. 2016-06-01]. Dostupné z: http://piyushramani8.blogspot.cz/2013/07/spring-container-bean-life-cycle.html

[20] Long, J.: Have You Seen Spring Lately? *Pivotal Software, Inc [online]*, 2013, [orbázek, cit. 2016-06-01]. Dostupné z: https://blog.pivotal.io/pivotal/products/have-you-seen-spring-lately

[21] Mark Fisher & Mark Pollack: Develop Powerful Big Data Applications Easily with SpringXD. 2014, [orbázek, cit. 2016-06-01]. Dostupné z: http://www.slideshare.net/SpringCentral/develop-powerful-big-data-applications-easily-with-springxd

[22] Groovy. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2016, [cit. 2016-06-04]. Dostupné z: https://cs.wikipedia.org/wiki/Groovy

[23] Oracle Corporation: *Introducing Groovy.* [cit. 2016-06-04]. Dostupné z: http://www.oracle.com/technetwork/articles/java/groovy-1695411.html

[24] König, D.; King, P.; Laforge, G.; Skeet J.: *Groovy in Action.* Manning Publications, second edition vydání, ISBN 978-1-93518-244-3.

[25] Groovy.codehaus.org: *Groovy Language Documentation, 3.2. Differences with Java.* [cit. 2016-06-05]. Dostupné z: http://docs.groovy-lang.org/latest/html/documentation/index.html#_differences_with_java

[26] Creating DSLs in Java, Part 1: What is a domain-specific language? *JavaWorld, Inc. [online]*, 2008, [cit. 2016-06-04]. Dostupné z: http://www.javaworld.com/article/2077865/core-java/core-java-creating-dsls-in-java-part-1-what-is-a-domain-specific-language.html

[27] Pivotal Software, Inc.: *Class GenericGroovyApplicationContext.* [cit. 2016-05-04]. Dostupné z: http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/support/GenericGroovyApplicationContext.html

[28] Pivotal Software, Inc.: *Spring Boot.* [cit. 2016-06-02]. Dostupné z: http://projects.spring.io/spring-boot/

[29] Pivotal Software, Inc.: *Spring Security.* [cit. 2016-06-02]. Dostupné z: http://projects.spring.io/spring-security/

[30] Authentication Providers. *AnswerHub: DZone Software [online]*, 2014, [obrázek, cit. 2016-06-02]. Dostupné z: http://docs.answerhub.com/articles/1134/authentication-providers.html

[31] IBM: *Understanding credentials-based authentication.* [cit. 2016-05-28]. Dostupné z: https://www.ibm.com/support/knowledgecenter/SGVKBA_3.2.0/com.ibm.rsct320.admin/bl503_credba.htm

[32] Šnajdr, P.: Dvoufaktorová autentizace: mýty a realita. *CCB spol. s r.o, SystemOnLine.cz [online]*, 2013, ISSN 1802-615X, [cit. 2016-05-27]. Dostupné z: https://www.systemonline.cz/it-security/dvoufaktorova-autentizace-myty-a-realita.htm

[33] Security token. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2016, [cit. 2016-05-28]. Dostupné z: https://en.wikipedia.org/wiki/Security_token

[34] Autentizace. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2014, [cit. 2016-05-29]. Dostupné z: https://cs.wikipedia.org/wiki/Autentizace

[35] Authorization. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2016, [cit. 2016-05-29]. Dostupné z: https://en.wikipedia.org/wiki/Authorization

[36] GigaSpaces Technologies: *Spring Security*. [obrázek, cit. 2016-06-03]. Dostupné z: http://docs.gigaspaces.com/xap97/introducing-spring-security.html

[37] OAuth. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2016, [cit. 2016-06-03]. Dostupné z: https://cs.wikipedia.org/wiki/OAuth

[38] Ghisellini, M.: *5 minutes with – Spring OAuth 2.0*. Wordpress.com [online], 2014, [obrázky, cit. 2016-06-04]. Dostupné z: https://techannotation.wordpress.com/2014/04/29/5-minutes-with-spring-oauth-2-0/

[39] Pivotal Software, Inc.: *Spring Social Reference: 2.2.1. OAuth2 service providers*. [obrázek, cit. 2016-06-03]. Dostupné z: http://docs.spring.io/spring-social/docs/1.1.0.RELEASE/reference/htmlsingle/#section_oauth2ServiceProviders

[40] Microsoft: *Introduction to Identity [online]*. [kód, cit. 2016-05-17]. Dostupné z: https://docs.asp.net/en/latest/security/authentication/identity.html

[41] Analýza požadavků. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2015, [cit. 2016-05-18]. Dostupné z: https://cs.wikipedia.org/wiki/Analýza_požadavků

[42] Jelínek, L.: Svobodné licence – základní kámen svobodného softwaru. *CCB spol. s r.o, LinuxEXPRES.cz [online]*, 2014, ISSN 1801-3996, [cit. 2016-06-12]. Dostupné z: http://www.linuxexpres.cz/svobodne-licence-zakladni-kamen-svobodneho-softwaru

[43] Copyleft. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2015, [cit. 2016-06-13]. Dostupné z: https://cs.wikipedia.org/wiki/Copyleft

[44] BSD licence. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2015, [cit. 2016-06-14]. Dostupné z: https://cs.wikipedia.org/wiki/BSD_licence

[45] Git. *Wikipedia: the free encyclopedia [online]*, San Francisco (CA): Wikimedia Foundation, 2016, [cit. 2016-06-15]. Dostupné z: `https://cs.wikipedia.org/wiki/Git`

[46] Láng, P.: Problémy s git-flow. *Langi.cz [online]*, 2011, ISSN 1802-615X, [cit. 2016-06-15]. Dostupné z: `http://langi.cz/webarna/problemy-s-git-flow`

[47] Beuth University of Technology Berlin (App.Sc.): *NoSQL DEFINITION: Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable [online].* [cit. 2016-06-01]. Dostupné z: `http://nosql-database.org/`

Příloha **A**

# Seznam použitých zkratek

**LDAP** Lightweight Directory Access Protocol

**CRUD** Create, Read, Update, Delete

**ORM** Object-relational mapping

**MVC** Model-view-controller

**HQL** Hibernate Query Language

**EJB2** Enterprise JavaBeans

**JPA** Java Persistence API

**NoSQL** non SQL / non relational[47]

**JPQL** Java Persistence Query Language

**SQL** Structured English Query Language

**BNF** Backusova-Naurova forma

**API** Application Programming Interface

**J2EE** Java Platform, Enterprise Edition

**DI** Dependency Injection

**AOP** Aspect-Oriented programming

**POJO** Plain Old Java Object

**IoC** Inversion of Control

**XML** Extensible Markup Language

**DSL** Domain-specific Language

**CSS** Cascading Style Sheets (Kaskádové styly)

**JVM** Java Virtual Machine

**LDAP** Lightweight Directory Access Protocol

**SMS** Short Message Service

**TOTP** Time-based One-time Password Algorithm

**HOTP** HMAC-based one-time password

**HMAC** Hash-based message authentication code

**USB** Universal Serial Bus

**RFID** Radio Frequency Identification

**CSRF** Cross-site Request Forgery

**ACL** Access Control List

**JDBC** Java Database Connectivity

**OAuth** Open Authentication standard

**RFC** Request For Comments

**HTTP** Hypertext Transfer Protocol

**HTTPS** Hypertext Transfer Protocol Secure

**REST** Representational State Transfer

**VPN** Virtual Private Network

**IP** internetový protokol

**UUID** Universally Unique Identifier

**HTML** HyperText Markup Language

**DAO** Data access object

**JSP** JavaServer Pages

**URL** Uniform Resource Locator

**MIME** Multipurpose Internet Mail Extensions

**JNDI** Java Naming and Directory Interface

**BSD** Berkeley Software Distribution

**MIT** Massachusettském technologickém institutu

**GNU GPL** GNU General Public License

Příloha **B**

# Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `jar` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `usrman` ..... správa uživatelů
    - `devman` ..... správy zařízení
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS