ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Sedliský** Jméno: **Filip** Osobní číslo: **411741**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové systémy**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Flexibilní modularizace zdrojového kódu

Název bakalářské práce anglicky:

On flexible source code modularization

Pokyny pro vypracování:

Prozkoumejte možnosti transformací zdrojového kódu v Javě.
Věnujte pozornost jak možnosti dynamické introspekce [1], tak i použití parserů [3].
Prozkoumejte aspektově-orientovaný způsobe transformace [2].
Analyzujte a navrhněte způsob transformace middleware modulu na menší moduly v technologii Java EE
Implementujte prototyp flexibilní modularizace a otestujte možnost na příkladu Java EE aplikace.

Seznam doporučené literatury:

[1] Ira R. Forman and Nate Forman. 2004. Java Reflection in Action (In Action Series). Manning Publications Co., Greenwich, CT, USA.
[2] Tomas Cerny, Karel Cemus, Michael J. Donahoo, and Eunjee Song. 2013. Aspect-driven, data-reflective and context-aware user interfaces design. SIGAPP Appl. Comput. Rev. 13, 4 (December 2013), 53-66.
[3] JavaParser http://javaparser.org/

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Tomáš Černý MSc., Ph.D., Software Engineering and Networking FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **04.10.2017** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce:
do konce zimního semestru 2018/2019

Podpis vedoucí(ho) práce Podpis vedoucí(ho) ústavu/katedry Podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

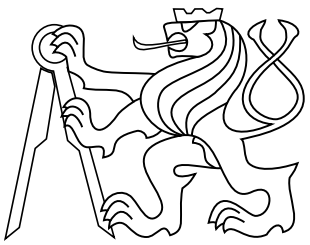
Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání Podpis studenta

bakalářská práce

Flexibilní modularizace zdrojového kódu

Filip Sedliský



Květen 2018

Ing. Tomáš Černý, MSc., Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická, Katedra počítačů

Poděkování

Chtěl bych poděkovat vedoucímu práce panu Ing. Tomáši Černému, MSc., Ph.D. za vstřícnost, ochotu a cenné rady při vytváření bakalářské práce.

Prohlášení

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

V Praze dne 23. května 2018 .....

Abstrakt

Předmětem této bakalářské práce je prozkoumání možností transformace zdrojového kódu v Javě, navrhnutí prototypu flexibilní modularizace pro transformace middleware modulu na menší moduly v technologii Java EE a její implementace pomocí knihovny JavaParser. Dále práce pojednává o otestování možností prototypu na příkladu Java EE aplikace a na závěr se zabývá otestováním výkonnosti Java EE aplikace v monolitu oproti distribuované verzi, která byla rozdělená prototypem.

Klíčová slova

JavaParser, prototyp flexibilní modularizace, Java EE, rozdělení aplikace, modul, monolit, distribuovaná verze

Abstrakt

The purpose of this bachelor thesis is to investigate various possibilities of a source code transformation in Java, to design a prototype of a flexible modularization for a middleware module transformation to smaller modules in Java EE technology and its implementation by using the JavaParser library. Furthermore, the thesis deals with examining of the prototype options on the Java EE application example, as well as performance testing of the Java EE application in the monolith in comparison with the distributed version that was split by the prototype.

Keywords

JavaParser, prototyp of flexible modularization, Java EE, dividing app, module, monolith, distributed version

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Tato bakalářská práce pojednává o možnostech transformace zdrojového kódu v Javě pomocí reflexe, aspektově orientovaného způsobu programování a pomocí knihovny JavaParser. Dále se zabývá vytvořením prototypu flexibilní modularizace a otestování na demonstrační aplikaci.

První část pojednává o knihovně JavaParser, která byla použita pro implementaci prototypu. Zanalyzujeme obecnou strukturu aplikace vyvíjené v jazyce Java EE. Navrhneme řešení pro prototyp flexibilní modularizace. Představíme si demonstrační aplikaci, která slouží k otestování vytvořeného prototypu a následně otestujeme výkonnost demonstrační aplikace v monolitu či distribuované verzi.

V následující části si definujeme důležité pojmy související s touto prací, formálněji definujeme požadavky na prototyp a vytvoříme slovník pojmů pro jednodušší orientaci. Ukážeme si na jednotlivých příkladech možnosti použití prototypu a otestujeme korektnost prototypu na demonstrační aplikaci.
Na závěr zhodnotíme výsledky práce.

1.1 Cíl práce

Hlavním cílem této bakalářské práce je analyzovat možnosti implementace a vytvoření prototypu flexibilní modularizace pro transformaci middleware modulu na menší moduly v technologii Java EE. Poté následné otestování prototypu na demonstrační aplikaci.

Součástí práce bude vyhodnocení výkonu demonstrační aplikace v monolitu či distribuované verzi.

2 Rešerše

2.1 Knihovna JavaParser

Knihovna JavaParser umožnuje se zdrojovým kódem v jazyce Java zacházet jako s Java objektem. Tento objekt se nazývá Abstract Syntax Tree (AST). Dále poskytuje mechanismus pro procházení stromu nazvaný Visitor Support. To dává vývojářům schopnost zaměřit se na zajímavé aspekty ve zdrojovém kódu, namísto psaní vlastního algoritmu pro procházení stromu[1].

V neposlední řadě knihovna poskytuje strukturu pro manipulaci se zdrojovým kódem. Struktura slouží pro jednodušší zápis do souborů a také vývojářům poskytuje nástroj pro vygenerování zdrojového kódu.

2.1.1 Abstract Syntax Tree

Představme si, že zdrojový kód se může reprezentovat jako strom s jedním začínajícím bodem. Ten se poté dělí na větve podle významu, které se dále dělí na menší části, dokud nenarazíme na konec nebo na list stromu.

Strom reprezentující Java kód má kořen reprezentující celý soubor. Pro všechny hlavní prvky souboru jsou uzly připojené ke kořeni, například importy nebo třídní deklarace. Z jedné třídní deklarace může vést více uzlů reprezentující třídní proměnné nebo její metody[1].

Obrázek 2.1 Ukázka stromové struktury Abstract Syntax Tree, převzato z [1]

Uzly ve stromu mají jednoduchá pravidla, všechny uzly s výjimkou kořene mají jednu vstupní relaci, kořen nemá žádnou vstupní relaci. Všechny uzly mají nula a více výstupních relací. Ty uzly, které nemají žádnou výstupní relaci se nazývají listy stromu. Naopak ty uzly, které mají výstupní relaci se nazývají větví a jsou rodičem jednoho nebo více potomků[1].

V případě uzlu, který má alespoň jednoho potomka, se u vytvořeného objektu knihovnou JavaParser reprezentují jeho potomci jako List<Node> zvaný **childredNode** ve třídě Node.

2.1.2 třída JavaParser

Třída JavaParser se používá pro vytváření AST ze zadaného kódu. Nejčastěji se používá pro vytváření AST ze zadaného souboru obsahující kompletní Java třídu, ale také ho můžete využít pouze pro vytvoření části zdrojového kódu, například k vytvoření třídy s několika metodami[1].

2.1.3 třída CompilationUnit

Třída CompilationUnit je reprezentací zdrojového kódu psaného v jazyce Java ze souboru obsahující syntakticky správnou Java třídu. Označujeme ji jako kořen AST vytvořený třídou JavaParser metodou **.parse()**. Považuje se za vstupní uzel, ze kterého máte plný přístup ke všem ostatním uzlům stromu[1].

2.1.4 Třídy Visitor

Pro jednodušší prohledávání v AST se používají třídy Visitor, kde definujete co hledáte. Poté Visitor najde všechny uzly dané kategorie, jako například všechny deklarace metod nebo všechny členské proměnné apod.[1].
Jednoduchý visitor pro vypsání všech anotací v deklaraci třídy může vypadat následovně:

```
1  public static class AnnotationClassVisitor
2        extends VoidVisitorAdapter<Void> {
3
4    @Override
5    public void visit(ClassOrInterfaceDeclaration cid, Void arg) {
6      super.visit(cid, arg);
7      cid.getAnnotations()
8         .forEach(ann -> System.out.println(ann.getName()));
9    }
10 }
```

a použití visitoru:

```
1  CompilationUnit cu = JavaParser.parse(new
       FileInputStream($pathToFile));
2  VoidVisitor<?> annotationClassVisitor = new
       AnnotationClassVisitor();
3  annotationClassVisitor.visit(cu, null);
```

2.2 Reflexe

Reflexe je schopnost programu získat informace o vlastní struktuře v čase běhu.
K prozkoumání struktury musí program znát vlastní reprezentaci, této informaci říkáme metadata. V objektově orientovaném programování jsou metadata uspořádána do objektů, které se nazývají meta objekty (metaobjects). Procesu při prozkoumávání meta objektů v čase běhu nazýváme introspekcí[2].

Reflexe podporuje tři techniky, jak můžeme umožnit změnu chování:

- přímá modifikace meta objektu
- oprerace pro používání metadat
- přímluva (intercession), ve které je povolen přístup ke kódu v různých fázích běhu programu

Java podporuje širokou škálu operací pro používání metadat. Tyto funkce umožňují vytvářet software flexibilní. Aplikace naprogramované pomocí reflexe se lépe adaptují na změnu požadavků.

Reflexe v Javě dokáže z objektů získat různé informace, mezi které patří:

- anotace
- metody
- členské proměnné
- konstruktory
- balíček, ve kterém se objekt nachází
- implementující rozhraní
- jméno objektu
- a další

Příklad reflexe pro vypsání názvů metod třídy Book:

```
1 Book book = new Book();
2 Class c = book.getClass();
3 Method methods []= c.getMethods();
4 for (Method m : methods) {
5    System.out.println("Method␣name:␣" + m.getName());
6 }
```

Ve třídě **c** máme reprezentovaný objekt třídy Book, ze kterého zjistíme veškeré metody třídy Book.

2.3 Práce, které se zabývají transformací zdrojového kódu

Programů, které se zabývají analýzou a transformací zdrojového kódu v Javě je určitě nespočet. Dokazují to dvě vybrané práce v podsekcích 2.3.1 a 2.3.2, o kterých se zmiňuji a pouze nastiňuji o čem pojednávají.

Nicméně prototyp flexibilní modularizace pro transformaci aplikací v Javě EE vyvíjený v této bakalářské práci nebyl zatím realizován, a proto jsme se rozhodli prototyp vytvořit.

2.3.1 Práce č.1: Certifying a java type resolution function using program transformation, annotation, and reflection

Tato práce se zabývá správným určením kanonické formy referenčního typu vzhledem ke kontextu. Jednoduchý příklad, který nastiňuje tento problém je vidět na obrázku 2.2.

Úvod práce: *In Java, type resolution—the determination of the canonical form of a type reference with respect to a given context—requires complex analysis. A simple example highlighting the essence of the type resolution problem is shown in 2.2 [3].*

```
package p1;

public class A {
    class B1 {}
    class innerA extends B {
        B1 myB1; // What is the canonical name
                 // of the type reference B1?
                 // Is it p1.A.B1 or p1.A.B.B1?
    }


    class B {
        private class B1 {}
    }
}
```

Obrázek 2.2 Ukázka rozlišení typu u proměnné, převzato z [3].

2.3.2 Práce č.2: Semi-automatic code-to-code transformer for Java

Tato diplomová práce pojednává o možnostech změně softwarových knihoven v semi-automatickém procesu.

Abstract: *Having the ability to perform large automatic software changes in a code base gives new possibilities for software restructuring and cost savings. The possibility of replacing software libraries in a semi-automatic way has been studied. String metrics are used to find equivalents between two libraries by looking at class- and method names. Rules based on the equivalents are then used to describe how to apply the transformation to the code base. Using the abstract syntax tree, locations for replacements are found and transformations are performed. After the transformations have been performed, an evaluation of the saved effort of doing the replacement automatically versus manually is made. It shows that a large part of the cost can be saved. An additional evaluation calculating the maintenance cost saved annually by changing libraries is also performed in order to prove the claim that an exchange can reduce the annual cost for the project[4].*

2.4 Slovník pojmů

Pro jednodušší orientaci tu zrekapitulujeme pojmy, které byly představeny v této kapitole a přidáme některé další, které budou velmi důležité v kapitolách následujících.

AST	Abstract Syntax Tree
CompilationUnit	strom reprezentující zdrojový kód ze syntakticky správné Java třídy
Monolit	Java EE aplikace, která není rozdělená
Distribuovaná verze	Java EE aplikace, která je rozdělená do několika modulů
Prototyp	vyvíjená aplikace, která transformuje Java EE aplikaci do modulů
Modul	část Java EE aplikace, která byla rozdělena prototypem
Frontend	část serveru, která obsahuje uživatelské rozhraní a je viditelné pro uživatele
Backend	část serveru, která je skrytá před uživatelem, obsahuje business logiku a přístup do databáze

Tabulka 2.1 Slovník pojmů.

3 Analýza

Práce má za cíl vytvořit prototyp pro rozdělení cílové aplikace do modulů v technologii Java EE.

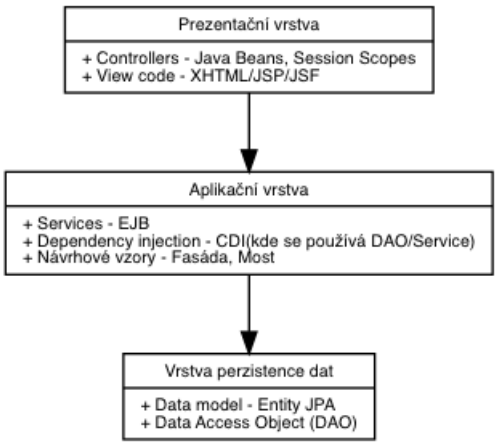
Byly prozkoumány dvě možnosti pro rozdělení do modulů. První možností je použít reflexi v Javě, druhou použít knihovnu JavaParser. Abychom zjistili, jakou technologii bude výhodnější použít, musíme se seznámit s obecnou strukturou Java EE aplikace.

3.1 struktura Java EE aplikace

Obecně se Java EE aplikace rozděluje do tří vrstev[5]:

- prezentační vrstva
- aplikační vrstva
- vrstva perzistence dat

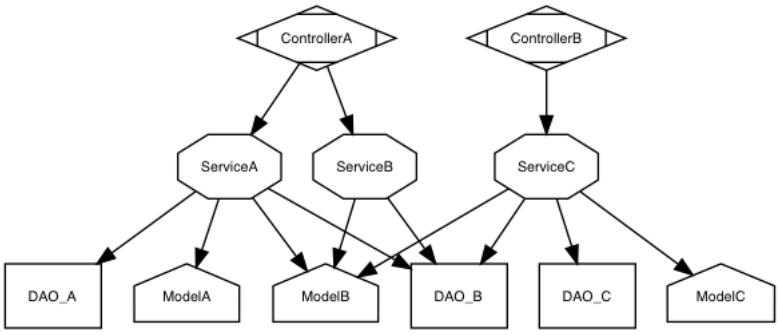
kde propojení jednotlivých vrstev je vidět na obrázku 3.1.



Obrázek 3.1 Rozdělení Java EE aplikace do tří vrstev.

Controllers z prezentační vrstvy využívá rozhraní Services, Services dále poožívají Model a vrstvu DAO. Můžeme si tedy všimnout, že se aplikace dá transformovat do stromové struktury, kde kořeny jsou jednotlivé Controllers, uzly jsou Services a listy poté Model a vrstva DAO. Je samozřejmé, že každá aplikace bude vypadat trochu odlišně, ale princip by měl být zachován. Výsledná stromová struktura aplikace se dvěma Controllers může vypadat jako na obrázku 3.2.

Z obrázku 3.2 je intuitivně vidět, jak by se aplikace mohla rozdělit do dvou modulů. V prvním modulu by se nacházel ControllerA a ve druhém ControllerB. ControllerA navíc potřebuje ServiceA, ServiceB a ty potřebují ModelA, ModelB, DAO_A a DAO_B. Všechny tyto soubory by se nacházely v prvním modulu. To samé bychom aplikovali na druhý modul, takže by nakonec obsahoval soubory: ControllerB, ServiceC, ModelB, ModelC, DAO_B a DAO_C.



Obrázek 3.2 Příklad stromové struktury aplikace v technologii Java EE.

Teď už máme přehled, jak obecně vypadá aplikace v technologii Java EE, a jak se aplikace může rozdělit například do dvou modulů.

3.2 Struktura rozhraní a třídy v Javě

Nyní se zaměříme na strukturu rozhraní a třídy v Javě. Rozhraní se používá jako abstraktní vrstva a zamezuje tzv. principu tight coupling[1], definuje konstanty a veřejné metody. Třída se skládá z členských proměnných a metod, může dále dědit z jiné třídy nebo implementovat rozhraní. Pro účely rozdělení aplikace, rozhraní a třídu, dále pouze objekt, můžeme rozdělit do čtyř bloků[7]:

- balíček (packages) - nepovinný, může chybět
- importy
- deklarace objektu
- členské proměnné a metody

Nalezení importů z objektu je velmi důležité, protože z importů zjistíme, které další objekty se musejí přidat do modulu. Importovat se může jeden objekt:

```
1 import org.jboss.as.quickstarts.service.AuthorRegistration;
```

nebo můžeme přidat všechny objekty z daného balíčku:

```
1 import org.jboss.as.quickstarts.service.*;
```

hvězdička na konci importu značí, že jsou přidány veškeré objekty z balíčku **service**. V tuto chvíli je důležité znát u objektu balíček, ve kterém se nachází, protože podle balíčku zjistíme nutnost přídání objektu do modulu.

[1] tight coupling – používá se pro označení softwaru, který funguje v určité části systému a je závislý na ostatních softwarech[6]. Což znamená, že například při změně jedné komponenty v systému se zpravidla musí změnit i komponenta, která je k ní úzce spojená.

3.3 Požadavky

3.3.1 Cílová aplikace se nezmění

Prototyp nesmí nijak změnit rozdělovanou aplikaci, ať už se jedná o adresářovou strukturu aplikace nebo o samotný zásah do jednotlivých souborů.

Prototyp musí vygenerovat novou adresářovou strukturu identickou s rozdělovanou aplikací, pouze s tím rozdílem, že aplikace bude rozdělená do jednotlivých modulů.

3.3.2 Flexibilita prototypu

Prototyp musí být schopný rozdělit cílovou aplikaci do dvou nebo tří nezávislých modulů. Dále musí umožnit generovat stejný Controller do více modulů, se stejnými nebo různými metodami.

3.3.3 Vygenerování konfiguračního souboru

Prototyp vygeneruje konfigurační soubor, který bude obsahovat důležité informace o transformaci cílové aplikace do modulů. Navíc bude prototyp jednoduše rozšiřitelný o nové implementace vytvářející konfigurační soubor.

3.3.4 Platformní nezávislost prototypu

Prototyp musí být platformě nezávislý. To znamená, že prototyp musí být spustitelný a korektně fungovat na operačním systému windows, linux a ios.

3.3.5 Otestování výkonu monolit vs. distribuované verze

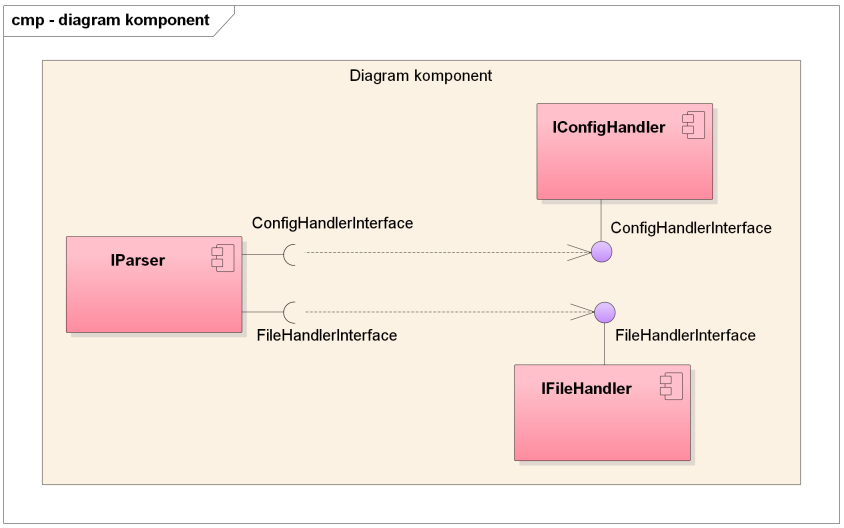
Otestování cílové aplikace v monolitu a distribuované verzi je velmi důležité, abychom zjistili výkonnostní rozdíl mezi dvěma formami cílové aplikace. Zjistíme, jestli má rozdělení cílové aplikace nějaký vedlejší efekt, co se týče například přístupu k databázi, rychlosti, atd.

3.4 Struktura prototypu

Prototyp má za úkol transformovat aplikaci v technologii Java EE do menších spustitelných modulů. Víme, že prototyp nesmí žádným způsobem změnit transformovanou aplikaci, takže budeme potřebovat vygenerovat podobnou strukturu aplikace s tím rozdílem, že bude transformovaná do modulů. Navíc k tomu také musíme vytvořit konfigurační soubor. Je patrné, že tedy můžeme prototyp rozdělit do tří komponent podle funkcionality. Rozdělení prototypu je zachycené na obrázku 3.3, který představuje diagram komponent prototypu.

Hlavní komponentou prototypu je **IParser**, která má za úkol transformování aplikace do modulů. Ke správnému fungování potřebuje komponenty **IFileHandler** a **IConfigHandler**. **IFileHandler** má na starost zapsání transformované aplikace do souborů. Poslední komponentou je **IConfigHandler**, která vytváří konfigurační soubor.

Z diagramu si také můžeme všimnout, že komponenta **IParser** nemusí znát přesnou implementaci komponent **IFileHandler** a **IConfigHandler**, pouze stačí znát jejich



Obrázek 3.3 Diagram komponent prototypu.

rozhraní. To je velmi přínosné, protože můžeme jednoduše zaměnit implementace například **IConfigHandleru** bez zásahu do komponenty **IParser**.

4 Návrh

V této kapitole navrhneme řešení prototypu pro flexibilní modularizaci zdrojového kódu. Navrhneme způsob transformace aplikace v jazyce Java EE do menších nezávislých modulů. Také se zde zaměříme na požadavky z kapitoly 3 – Analýza, které musí prototyp splňovat.

Z kapitoly 3 víme, jak obecně vypadá struktura aplikace v technologii Java EE. To je velmi důležité, abychom věděli, jak správně prototyp navrhnout.

4.1 Obecné fungování prototypu

Prototyp má za úkol transformovat aplikaci v technologii Java EE do menších nezávislých modulů[8][9]. Zatím jsme se ale nezabývali jak. Prototypu také musíme zajistit, aby věděl, jakým způsobem má cílovou aplikaci rozdělit.

Jazyk Java, popřípadě jazyk Java EE, má velmi dobrou podporu anotací, ať už se jedná například o testování pomocí unit testů (jednotkové testy) nebo v případě Javy EE pro označení Java Beans apod. Proto jsme se rozhodli použít anotace pro označení cílové aplikace tak, aby prototyp zjistil, jak se má transformace provést. Z obrázku 3.2, která ukazuje stromovou strukturu aplikace, vidíme, že stačí anotací označit tzv. Controllers a prototyp už je poté schopen provést správnou transformaci cílové aplikace do modulů.

4.2 Vstup prototypu

Vstupem do prototypu bude cesta ke zdrojovým souborům cílové aplikace. Nezáleží, jestli bude cesta k cílové aplikaci relativní například:

```
1 ./BookSystem
```

nebo absolutní

```
1 /Users/admin/Desktop/BookSystem
```

Také víme, že každý operační systém může mít v cestě jiný oddělovač adresářů a souborů. V linuxu se používá „/“ (lomítko) namísto „\“ (zpětné lomítko), které se používá v DOSu (MS windows apod.). Z požadavku 3.3.4 „Platformní nezávislost“ víme, že prototyp musí být platformě nezávislý. Tudíž prototyp musí podporovat oba tyto zápisy cesty.

4.3 Výstup prototypu

Z požadavku 3.3.1 „Cílová aplikace se nezmění“ víme, že při transformaci cílové aplikace do modulů nesmíme změnit původní aplikaci. To znamená, že prototyp musí vygenerovat kopii adresářové struktury cílové aplikace se soubory a tu poté transformovat do modulů.

Výstupem prototypu tedy bude kopie cílové aplikace transformované do modulů, navíc prototyp vygeneruje konfigurační soubor obsahující informace o rozdělení cílové aplikace.

4.4 Konfigurační soubor

Prototyp při transformaci cílové aplikace do modulů má také za úkol vygenerovat konfigurační soubor, který bude tyto informace obsahovat.

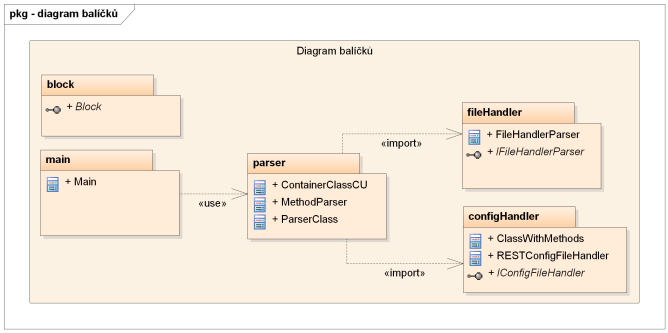
Nyní se zaměříme na strukturu dat. Na začátku této kapitoly jsme se dozvěděli, že pro správné fungování prototypu musíme anotacemi označit poslední vrstvu (Controllers) cílové aplikace, která představuje API rozhraní. Tudíž konfigurační soubor by měl obsahovat názvy Controllers s příznakem, do jakého modulu byl přiřazen, popřípadně může obsahovat jeho metody atd.

Problémů s vygenerováním konfiguračního souboru je hned několik. Nevíme, pro jaké účely bude konfigurační soubor používán. Bude využíván pouze pro informování o rozdělení nebo bude mít další využití. Nevíme, jaké API rozhraní cílová aplikace používá, jedná-li se o REST nebo jiné. Také nevíme, jakou přesnou strukturu a formát mají data obsahovat. Máme na výběr z formátu JSON, xml, csv nebo jiné.

Je vidět, že nelze vytvořit pouze jeden univerzální konfigurační soubor, který by vyhověl všem. Z tohoto důvodu je velmi důležité, aby byl prototyp nezávislý na konkrétní implementaci, která vytváří konfigurační soubor.

4.5 Struktura prototypu

Prototyp můžeme rozdělit do pěti balíčků podle logické odpovědnosti za jednotlivě vykonávané bloky funkcí, strukturu lze vidět na obrázku 4.1.



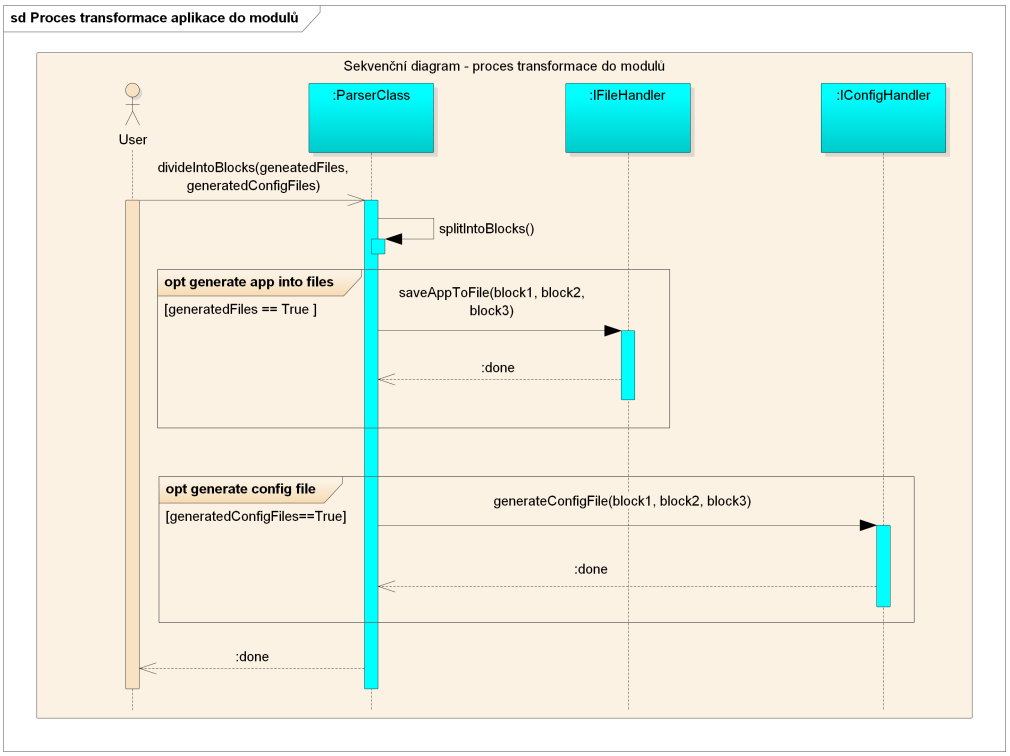
Obrázek 4.1 Struktura balíčků prototypu

Balíček **block** se v prototypu nevyužívá, ale nachází se zde rozhraní **Block**, které je potřeba přidat do struktury transformované aplikace pro používání anotací **@Block**. Z toho důvodu je v prototypu obsažený, aby byl jednoduše dostupný.

4.6 Fáze prototypu

Transformování cílové aplikace do modulů můžeme rozdělit do tří hlavních fází:

1. fáze rozdělení
2. fáze vygenerování transformované aplikace
3. fáze vygenerování konfiguračního souboru



Obrázek 4.2 Sekvenční diagram – proces transformace do modulů

Ze sekvenčního diagramu z obrázku 4.2, který popisuje celý proces transformace, lze vidět jednotlivé fáze prototypu.

V první fázi je zavolaná metoda **divideIntoBlocks()** se dvěma parametry, které určují zda se vykoná druhá a třetí fáze. Třída **ParserClass** zavolá metodu **splitIntoBlocks()**, která provede samotnou transformaci aplikace do modulů, ale zatím se nic do souborů nezapisuje.

Provedení druhé fáze záleží na parametru **generatedFiles**, při hodnotě **true** se vykoná zápis transformované aplikace do souborů metodou **saveAppToFile()** se třemi parametry, které představují rozdělené moduly aplikace.

I ve třetí fázi záleží na vykonání na parametru, tentokrát na parametru **generatedConfigFiles**, kdy se při hodnotě **true** zavolá metoda **generateConfigFile()** se stejnými třemi parametry jako v metodě **saveAppToFiles()**, která vytvoří konfigu-

rační soubor.
Návratové hodnoty **done** představují výpis v logu, které informují o ukončení jednotlivých fázích.

5 Související práce

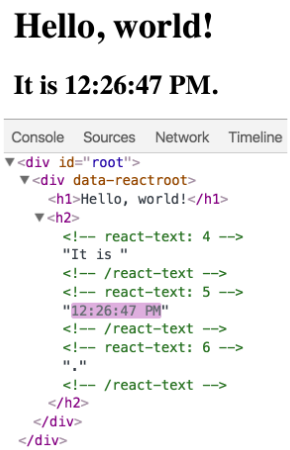
V této kapitole si vysvětlíme technologie použité v kapitole 6 – Demonstrační aplikace a v kapitole 7 – Prototyp aplikace.

5.1 React.js

Knihovna React.js je deklarativní, efektivní a flexibilní JavaScriptová knihovna pro vytváření uživatelského rozhraní[10].

Knihovna React.js poskytuje pro vytváření DOM[1] modelu komponentu ReactDOM, která obsahuje metodu **.render()**. Tato metoda zajišťuje vytváření DOM modelu v příslušném kontejneru a vrací referenci na kontejner nebo ukazatel nastaví na null[10]. Ukázka použití komponenty ReactDOM, která vypíše na stránku „Hello world.“ a aktuální čas [10]:

```
1 function tick() {
2   const element = (<div>
3     <h1>Hello, world!</h1>
4     <h2>It is {new Date().toLocaleTimeString()}.</h2>
5   </div>);
6   ReactDOM.render(element, document.getElementById('root'));
7 }
8 setIntervat(tick, 1000);
```



Obrázek 5.1 Ukázka webové stránky, ve které se každou sekundu změní hodnota času. Z obrázku je vidět, že se vždy načte pouze hodnota času, podbarvená růžově. Vše ostatní na stránce zůstává původní. Obrázek převzat z [10].

[1] DOM – Document Object Model

Hlavní výhoda použití knihovny React.js je šetrná aktualizace DOM modelu. Jakmile je DOM model jednou vytvořen pomocí komponenty ReactDOM, při změně obsahu v DOM modelu se modifikují pouze potřebné prvky tak, aby byl vždy aktuální. To znamená, že ReactDOM při změně nepřepisuje celý kontejner s DOM modelem, ale pouze jeho potomky. Tudíž se v prohlížeči poprvé načte celý DOM model, a poté se načítá pouze modifikovaný obsah (lze vidět na obrázku 5.1).

To má velký dopad ve zlepšení výkonnosti aplikace. Navíc ReactDOM reaguje na změnu, při které upozorní modifikované potomky a zavolá tzv. callback funkce.

5.2 Gatling.io

Gatling je velmi výkonný nástroj pro testování zatížení. Je navržen pro jednoduché používání, údržbu a vysoký výkon. Také má mimořádnou podporu pro HTTP protokol, který z něj činí nástroj pro libovolné testování HTTP serveru. Hlavní engine je agnostický protokol, takže je možné naimplementovat podporu pro další protokoly[11].

5.3 Architektura REST

Pojem REST, neboli **Re**presentation **S**tate **T**ransfer, byl poprvé představen Royem Fieldingem v jeho disertační práci v roce 2000. Architektura REST je popsána šesti omezeními[12]:

1. Klient - server
2. Jednotné rozhraní
3. Vrstvený systém
4. Mezipaměť
5. Bezestavovost
6. Code-on demand

5.3.1 Klient – Server

Klientská a serverová část je oddělená a mohou být implementovány nezávisle, každá v jiném jazyce a technologii, pouze se musí dodržet jednotné rozhraní[12].

5.3.2 Jednotné rozhraní

Jednotné rozhraní definuje rozhraní mezi klientskou a serverovou částí. Zjednodušuje vývoj aplikace, umožňuje každou část implementovat nezávisle.
Jednotné rozhraní se řídí čtyřmi pravidly[12]:

- **Identifikace zdroje**
  Každý zdroj je v požadavku identifikován pomocí URI[2] adresy.

- **Manipulace zdrojů přes reprezentaci**
  Klientská část manipuluje se zdroji pomocí reprezentací. To znamená, že zdroj může bát reprezentován různými způsoby, různým klientům. Například pro webový prohlížeč může být dokument reprezentován jako HTML stránka a pro automatický program jako JSON struktura. Tento koncept zabezpečuje, že samotný

[2] URI – Unified Resource Identifier

zdroj není měněn, pouze se mění jeho reprezentace[12].

- **Self-descriptive message**
  Každá zpráva poskytuje dostatek informací ke zpracování požadavku.

- **Hypermedia as the engine of application state(HATEAOS)**
  Reprezentace stavu zdroje poskytuje odkaz k souvisejícím zdrojům. Odkazy poskytují uživateli procházení webu ve smysluplném sledu[12].

5.3.3 Vrstvený systém

Požadavek na vrstvený systém zabezpečuje webovým zprostředkovatelům jake je proxy nebo gateway (brána) se transparentně připojit mezi klientskou a serverovou část pomocí jednotného rozhraní. Weboví zprostředkovatelé jsou běžně používané například pro vynucení bezpečnosti[12].

5.3.4 Mezipaměť (CACHE)

Ukládání odpovědi do mezipaměti je velmi důležité omezení webové architektury. Pomáhá redukovat klientem vnímanou latency, zvyšuje spolehlivost aplikace a kontroluje zatížení webového serveru.
Ukládání do mezipaměti snižuje celkové náklady na web[12].

5.3.5 Bezestavovost

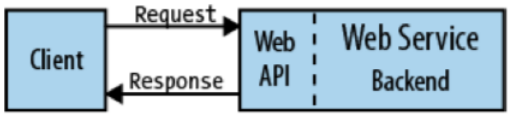
Omezení na bezestavovost zaručuje, že webový server není povinný si pamatovat stav svých klientských aplikací. To má za důsledek,že klient musí při každé interakci se serverem přikládat k požadavku veškeré relevantní informace[12].

5.3.6 Code-on demand

Tzv. „kód na vyžádání“ je jediné volitelné omezení takové, že dočasně umožňuje serveru odesílat spustitelný kód klientovi. Má to své nevýhody, klient musí být schopný porozumět a spustit kód, který byl stáhnutý ze serveru na požádání. Příkladem „kódu na požádání“ jsou například Java applety a JavaScripty[12].

5.4 REST API

Architektura REST je běžně používaná při návrhu API[3] rozhraní pro moderní webové služby. Webové API, které je založeno na architektuře REST se nazývá REST API. Webové aplikace, které podporují REST API nazýváme „RESTfull“[12].



Obrázek 5.2 Znázorněná komunikace mezi klientem a webovým serverem prostřednictvím webového API, převzato z [12].

[3] API – Application Programming Interface

REST API má jednotný přístup ke zdrojům prostřednictvím URI adresy a implementuje čtyři metody, které pokrývají základní funkce pro přístup k datům. Tyto funkce jsou také označovány pod názvem CRUD, tedy vytvoření dat(create), získání požadovaných dat(read), změu dat(update) a smazání dat(delete).
Metody, které REST API implementuje:

- POST(create) - metoda pro vytvoření dat
- GET(read) - metoda pro získání dat
- PUT(update) - metoda pro aktualizaci dat
- DELETE - metoda pro smazání dat

6 Demonstrační aplikace

Pro demonstrační účely rozdělení Java EE aplikace do modulů byla vybrána semestrální práce z předmětu A4B33SI.

Aplikace byla navržena jako knihovní systém, ve kterém se dají vytvářet autoři, nakladatelství, knihy a knihovny. Také se dají vkládat knihy do knihoven, uzavírat smlouvy mezi autorem a nakladatelstvím, atd. Semestrální práce byla značně předělána a rozdělena na dvě části. Nyní se skládá z frontendové a backendové části serveru. Dřívější uživatelské rozhraní JavaServer Faces bylo nahrazené frontendovou částí serveru napsanou pomocí JavaScriptové knihovny React. Pro backendovou část serveru zůstala samotná aplikace, business logika se nezměnila, jen bylo přidáno rozhraní REST (sekce 5.4).

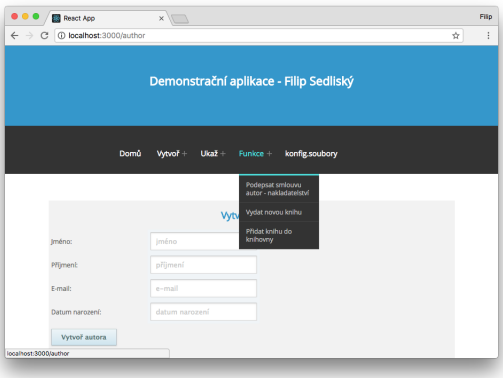
6.1 Frontendová část serveru

Frontend[1] je pojem, který označuje část webu viditelné pro uživatele, to znamená definuje uživatelské rozhraní serveru.

Jak bylo zmíněno, frontend byl napsán pomocí JavaScriptové knihovny React (sekce 5.1) a je pojmenován BookSystemFrontend. Spouští se v terminálu v adresáři serveru příkazem:

```
1 npm start
```

Uživatelské rozhraní je převážne statické. Obsahuje stránky s formuláři pro vytváření autorů, knih, apod. Dále stránky pro jejich výpisy a stránku s výpisem dvou konfiguračních souborů.



Obrázek 6.1 Ukázka uživatelského rozhraní v prohlížeči Chrome.

[1] frontend – přeložené jako „přední část“

6.2 Backendová část serveru

Backend[2] je část serveru, která slouží k administraci a ke zpracování dat, je v něm obsažena veškerá business logika a přístup k databázi.

Backend je pojmenován jako BookSystem a je napsán v jazyce Java EE, ke správě a sestavování serveru se používá nástroj Apache Maven a spouští se na aplikačním serveru WildFly. Server se připojuje k databázi PostgreSQL, ve které uchovává veškerá data. Serveru bylo přidáno API[3] rozhraní REST pro komunikaci s okolními aplikacemi. Veškerá odesílaná data ze serveru jsou ve formátu JSON, který je také očekáván při příjmu dat.
Příklad RESTového požadavku GET na server pro obdržení všech autorů vypadá následovně:

```
1 GET http://localhost:8080/BookSystem/rest/authors
```

a obdržená odpověď s HTTP hlavičkou a dvěma autory:

```
1  HTTP/1.1 200 OK
2  X-Powered-By: Undertow/1
3  Access-Control-Allow-Headers: origin, content-type,
4    accept, authorization
5  Server: WildFly/11
6  Date: Wed, 11 Apr 2018 13:46:53 GMT
7  Connection: keep-alive
8  Access-Control-Allow-Origin: *
9  Access-Control-Allow-Credentials: true
10 Content-Type: application/json
11 Content-Length: 4650
12 Access-Control-Allow-Methods: GET, POST, PUT,
13   DELETE, OPTIONS, HEAD
14
15 [{"id":1,"name":"John","surname":"Smith","email":
16 "johnSmith@seznam.cz","dateOfBirth":"12.12.1999",
17 "publishers":[],"books":[]},{"id"2,"name":"Petr",
18 "surname":"Novak","email":"novak@gmail.com",
19 "dateOfBirth":"1.2.1969","publishers":[], "books":[]}]
```

Na prvním řádku s odpovědí si můžete všimnout stavového kódu 200, který značí úspěšný HTTP požadavek.

6.2.1 Struktura serveru

Struktura serveru se skládá ze tří vrstev[13]:

- Prezentační vrstva
- Aplikační vrstva
- Vrstva perzistence dat

[2]backend – přeložené jako „zadní část“
[3]API – Application Programming Interface, označuje rozhraní pro programování aplikací.

Prezentační vrstva

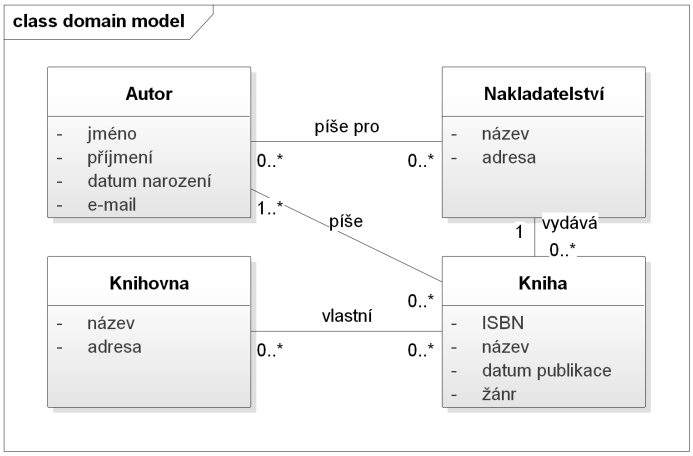
Prezentační vrstva je zodpovědná za zpřístupnění funkcionalit aplikace uživateli. V našem případě server poskytuje rozhraní REST API a nachází se v balíčku **rest**.

Aplikační vrstva

Aplikační vrstva zajišťuje vlastní funkcionalitu programu. Nachází se v balíčcích **service** a **serviceImpl**.

Vrstva perzistence dat

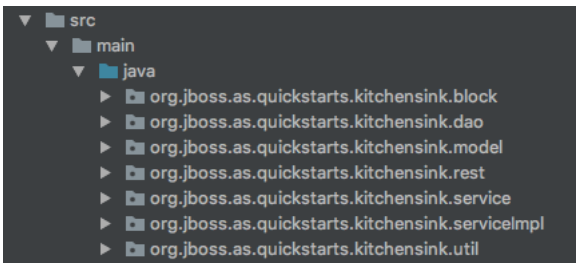
Vrstva perzistence dat slouží pro ukládání dat do perzistentního úložiště. Server používá jako úložiště relační databázi PostgreSQL a pro komunikaci s ní používá PostgreSQL JDBC driver. Tato vrstva je rozdělena do balíčku **model**, který je relačně mapován na databázi (ORM[4]), a balíčku **dao**, který implementuje návrhový vzor Data Access Object (DAO)[15] pro perzistentní přístup do databáze. Doménový model se skládá ze čtyř entit: autor, nakladatelství, kniha a knihovna, kde jejich vazby jsou vidět na obrázku 6.2.



Obrázek 6.2 Doménový model demonstrační aplikace.

6.2.2 Adresářová struktura

Adresářová struktura backendu je zobrazena na obrázku 6.3.



Obrázek 6.3 Adresářová struktura aplikace.

[4]ORM – objektově relační mapování, technika pro automatickou konverzi mezi objektově orientovaným programováním a relační databází[14].

6.3 Propojení Frontend - Backend

Frontend je navržen tak, aby byl schopný komunikovat s backendem v monolitu či distribuované verzi. To znamená, že se frontend připojí buď k jednomu serveru obsahující celé rozhraní REST, nebo se připojí k několika serverům, kde záleží jakou část frontend u rozhraní REST vyžaduje. Pro tyto účely se ve frontendu používají dva konfigurační soubory. Konfigurační soubor **dataRESTControllers.json** se nemění a je přímo mapovaný na Controllers a na jejich metody z backendové části. Takže soubor obsahuje všechny jména Controllers s názvy metod a jejich relativními URL cestami ve formátu JSON. Druhý konfigurační soubor je generován prototypem a používá se pouze pro distribuovanou verzi backendu. Obsahuje názvy Controllers s jejich metodami a příznak, ve kterém modulu se Controller nachází.

Rozhodnutí zda je frontend připojován k monolitu či distribuované verzi závisí na proměnné typu boolean s názvem **isMonolit**, která se nachází v adresáři **configFile/** v souboru **configFunctions.js**. Rozhodnutí se musí provést před spuštěním frontendu, protože proměnná **isMonolit** je konstantní a nemůže se za běhu serveru změnit.

Vzhled uživatelského rozhraní se nijak nemění, ať se server připojí k monolitu anebo k distribuované verzi. Tudíž uživatel není schopen rozeznat rozdíl.

7 Prototyp flexibilní modularizace

7.1 Shrnutí aplikace

Prototyp je aplikace pro transformaci middleware modulu na menší nezávislé moduly v technologii Java EE. Skládá se ze dvou částí, první část je samotná aplikace (prototyp), která je zodpovědná za zanalyzování cílové aplikace, transformace aplikace do nezávislých modulů tak, aby se v modulech nacházely pouze potřebné soubory. Druhá část prototypu se skládá z rozhraní anotace **@Block**, které se vkládá do cílové aplikace a používá se pro označení Java objektů. Podle této anotace se prototyp rozhoduje, do jakého modulu objekt patří.

Prototyp žádným způsobem nemodifikuje cílovou aplikaci při transformaci do modulů. Pouze využívá znalosti adresářové struktury a vytváří kopie souborů z cílové aplikace, které poté rozdělí do modulů.

7.2 Výběr technologie

Pro implementaci prototypu jsem zvolil jazyk Java s použitím knihovny JavaParser a nástroje Maven pro správu a sestavení aplikace. Knihovna JavaParser poskytuje AST[1] objektu, který obsahuje veškeré potřebné informace, za to zásadní nevýhodou použití reflexe v Javě je neznalost importů daného objektu. Při výběru nástroje Maven stačí pro používání knihovny JavaParser přidat dependency do souboru **pom.xml**:

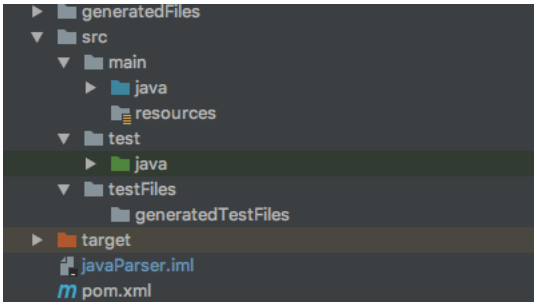
```
<dependency>
  <groupId>com.github.javaparser</groupId>
  <artifactId>javaparser-core</artifactId>
  <version>3.0.0-RC.3</version>
</dependency>
```

kde tato verze JavaParseru byla použita v prototypu a její nejnovější verze se může časem změnit.

7.3 Implementace

Pomocí nástroje Maven byla vygenerována hlavní adresářová struktura prototypu, která je zobrazena na obrázku 7.1. V adresáři **main/** se nacházejí veškeré zdrojové soubory, které implementují prototyp. V adresáři **test/** se nacházejí unit (jednotkové) testy, které nám otestují korektnost vyvíjené aplikace a k tomu využívají pomocné soubory, které se nacházejí v adresáři **testFiles/**. Do adresáře **generatedFiles/** prototyp defaultně ukládá veškeré generované soubory.

[1] AST – Abstract syntax tree



Obrázek 7.1 Hlavní adresářová struktura prototypu

Z kapitoly 3 – Analýza víme, že se prototyp skládá ze tří komponent (obrázek 3.3), které si tu rozsáhleji představíme, ale před tím se koukneme na třídu **Main** z balíčku **main**.

7.3.1 třída Main

Třída Main obsahuje pouze statickou metodu, která spouští celou aplikaci a přijímá veškeré argumenty na vstupu. Mění se zde například cesta k cílové aplikaci, která má být rozdělená do více modulů. Veškeré možnosti nastavení si probereme v sekci 7.7 **Použití prototypu**.

7.3.2 komponenta IParser

Komponenta IParser je nejdůležitější komponentou v prototypu. Nachází se v balíčku **parser** a obsahuje tři soubory. Je zodpovědná za rozdělení cílové aplikace do nezávislých modulů.

Třída ParserClass

Ve třídě ParserClass dochází k rozdělení cílové aplikace do modulů. Rozdělení se skládá ze tří fází.

V první fázi nalezneme všechny Java soubory ze zadané cesty (adresáře) a jejich podadresářů, uložíme si je do mapy, jako klíč je použita absolutní cestu k souboru a jako hodnota je třída **ContainerClassCU**. Při naleznutí anotace @Block u deklarace třídy nebo rozhraní se také přidá do mapy daného modulu podle parametru z anotace. Tudíž máme čtyři mapy: jednu, kde se nachází všechny soubory a poté postupně mapa pro první, druhý a třetí modul.

Nyní máme v každém modulu pouze objekty, které měli v deklaraci třídy nebo rozhraní anotaci @Block. Ve druhé fázi se postupně pro každý modul začnou analyzovat objekty, ve kterých se zjišťuje, jestli není potřeba některý objekt dopřidat. Nutnost přidání objektu se prohledává z:

- importů
- implementujících rozhraní (implements)
- děděných tříd (extends)
- členských proměnných

Při zjištění chybějícího objektu v modulu se daný objekt do modulu přidá a také se u něj zkoumá, jestli není potřeba některý objekt přidat. Tudíž se potřebné objekty rekurzivně přidávají do modulu, dokud se neprozkoumají všechny objekty z modulu a

nejsou všechny chybějící objekty (ze zkoumaných čtyř oblastí objektu) přidány.

To stále nestačí, protože tímto způsobem se například nepřidaly děděné třídy a třídy, které implementují rozhraní. Ve třetí fázi procházíme mapu, ve které se nacházejí všechny nalezené objekty ze zadané cesty, a zkoumáme zda objekt neimplementuje rozhraní nebo nedědí od jiné třídy. V případě že ano, v každém modulu prohledáváme, jestli se v něm nenachází právě implementující rozhraní nebo děděná třída. Při shodě se objekt dodatečně přidá do modulu a poté se objekt prozkoumá jako ve druhé fázi.

Po skončení třetí fáze se v modulech nacházejí veškeré potřebné objekty a třída **ParserClass** s transformací do modulů končí. Nakonec je při nutnosti zavolaná metoda **saveAppToFile()** z komponenty IFileHandler a metoda **generateConfigFile()** z komponenty IConfigHandler.

Třída MethodParser

Třída **MethodParser** má za úkol správné zachování metod u tříd a rozhraní označené anotací **@Block**.

Třída obsahuje jednu veřejnou metodu se dvěma parametry. V prvním parametru získá mapu (HashMap) obsahující objekty jednoho modulu a ve druhém parametru příznak, o jaký modul se jedná. Z mapy poté vyfiltruje třídy a rozhraní, které nejsou označené anotocí **@Block**. Ze zbylých tříd a rozhraní zkoumá anotace u jednotlivých deklarací metod. Při nalezení anotace **@Block** porovnává její parametr s příznakem modulu. Při shodě se metoda v objektu nechá, v opačném případě se z objektu odstraní. Ve chvíli, kdy se odstraněná metoda nenachází v žádném modulu, prototyp vypíše do logu varování (warning) o špatně zvoleném parametru.

Třída ContainerClassCU

Třída **ContainerClassCU** vytváří CompilationUnit (kořen Abstract Syntax Tree) ze zadané cesty pomocí knihovny JavaParser. Dále poskytuje veškeré metody potřebné pro rozdělení do modulů, které má CompilationUnit implementovány. To znamená, že kdykoliv potřebujeme přistoupit k metodě z CompilationUnit, nezavoláme ji přímo, ale skrz třídu **ContainerClassCU**. Hlavní výhodou tohoto přístupu je, že **ContainerClassCU** zavolá metodu z CompilationUnit pouze jednou a výsledek si uloží do členské proměnné. Navíc tyto metody jsou vyhodnocované tzv. líným (lazy)[16] přístupem, což znamená, že se metoda nevykoná do té doby, dokud se poprvé nezavolá.

7.3.3 Komponenta IFileHandler

Komponeta se skládá z rozhraní **IFileHandler** s jednou veřejnou metodou **saveAppToFile()**, která ukládá rozdělené moduly do souborů, a třídy, která rozhraní implementuje.

V defaultním nastavení se transformovaná aplikace ukládá do adresáře **generatedFiles/**, která se nachází v adresářové struktuře prototypu. Pro transformovanou aplikaci se vygeneruje hlavní adresář, který je pojmenován podle transformované aplikace a k tomu se přidává časové razítko jako přípona, kdy byla transformace provedena. V hlavním adresáři se nachází dva nebo tři spustitelné moduly pojmenované podle parametru z anotace **@Block**, ve kterých se nachází podmnožina adresářové struktury rozdělené

aplikace. Tudíž se adresářová struktura rozdělené aplikace nezmění. Navíc se při zjištění souboru **pom.xml** v modulu provede přejmenování hodnoty v artifactId, která určuje název aplikace. K původní hodnotě se přidá přípona „Block“ s číslem, do kterého modulu patří. Takže například v naší demonstrační aplikaci se v souboru **pom.xml** z druhého modulu přejmenuje hodnota artifactId z „BookSystem“ na „BookSystemBlock2“.

7.3.4 Komponenta IConfigHandler

Komponenta **IConfigHandler** má také pouze jednu veřejnou metodu **generateConfigFile()**, která generuje konfigurační soubor. Z požadavku 3.3.3 víme, že prototyp nesmí být přímo závislý na jedné implementaci vytvářející konfigurační soubor. Toho jsme dosáhli vytvořením rozhraní, na které se prototyp odkazuje, místo odkazování přímo na danou implementaci.

Byl implementován **RESTConfigFileHandler**, který generovaný soubor ukládá v defaultní nastavení do adresáře **generatedFiles/** podobně jako komponeta **IFileHandler** a je pojmenován **config.json**. Jakmile se zjistí, že se už v adresáři soubor **config.json** nachází, soubor není znovu vygenerován a původní soubor se nepřepíše. To je z důvodu bezpečnosti, abychom nemohli nechtěně přepsat už existující soubor.

Konfigurační soubor je generován ve formátu JSON, který obsahuje jedno pole. Prvky pole poté tvoří název třídy nebo rozhraní, příznak v jakém modulu se nachází a list s názvy jeho metod, které implementují rozhraní REST (jsou označené anotací @GET, @POST, @PUT nebo @DELETE).

7.3.5 Anotace @Block

V jazyce Java můžeme vytvořit vlastní anotace, které se mohou libovolně vkládat do třídy nebo rozhraní. Prototyp využívá vytvořenou anotaci **@Block**[17] pro zjištění, do jakého modulu třída nebo rozhraní patří. Celé rozhraní vypadá následovně:

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ ElementType.TYPE, ElementType.METHOD })
3 public @interface Block {
4
5     String[] value();
6 }
```

Ze zdrojového kódu si můžeme všimnout, že anotace @Block se přidává k deklaraci rozhraní, třídy nebo metody. Anotace se používá spolu s jedním až třemi parametry key1, . . . , key3 k určení, do jakého modulu rozhraní, třída, případně metoda patří.

7.4 Flexibilita

Prototyp je platformě nezávislý.

Máme úplnou kontrolu nad transformací cílové aplikace do modulů, díky řešení s přidáváním anotací do cílové aplikace. Anotace **@Block** jsou přidávány manuálně, takže si sami určíme, jak bude výsledná transformace aplikace vypadat.

Také můžeme označit anotacemi metody u tříd a rozhraní pro větší flexibilitu. V tu chvíli se transformovaná třída nebo rozhraní generuje pouze s některými metodami (detailnější používání anotací si ukážeme v sekci – 7.7 Použití prototypu). Prototyp umožňuje transformovat cílovou aplikaci do dvou nebo tří spustitelných modulů, vše záleží na použití parametrů u anotací.

7.5 Omezení

Omezení prototypu je ve způsobu transformace cílové aplikace do modulů. Prototyp u každé třídy nebo rozhraní přidané do modulu prohledá všechny importy, členské proměnné, implementující rozhraní, popřípadě děděné třídy. Z toho prototyp zjišťuje, jestli je nutné některou třídu nebo rozhraní do modulu přidat. Tudíž prototyp neprochází u každé třídy a rozhraní řádek po řádku.
Mohlo by se tedy stát, že by prototyp za jistých okolností transformoval aplikaci špatně. Například kdybychom měli dvě třídy, třídu A a třídu B ze stejného balíčku, a třída A používala třídu B pouze v těle některých z metod, tak by třída A nepotřebovala import řídy B (nacházejí se ve stejném balíčku) a ani by se neobjevila v dalších prohledávaných možnostech. V tuto chvíli by prototyp nebyl schopný vygenerovat moduly správně.
Řešením by mohlo být prohledávání i metod tříd. To by bohužel mělo velmi negativní dopad na výkonnost, složitost a pracnost při transformaci prototypem. Navíc toto omezení je velmi malé a ve většině případů nijak limitující. Z tohoto důvodu jsme se rozhodli od této možnosti upustit.

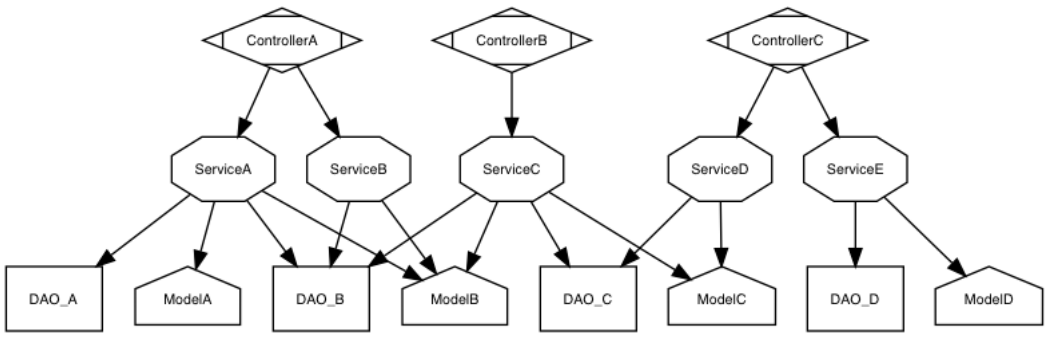
7.6 Použití anotace @Block u cílové aplikace

Prototyp není schopný bez přidání anotace @Block do cílové aplikace správně transformovat aplikaci do modulů. Proto si ukážeme jednoduché příklady, jak správně anotaci @Block používat.

Pro připomenutí anotace @Block se používá u deklarace rozhraní, třídy nebo metod s jedním až třemi parametry, kde platnými parametry jsou:

- key1 - pro první modul
- key2 - pro druhý modul
- key3 - pro třetí modul

Následující příklady transformace do modulů budou prováděny na aplikaci, která má stejnou strukturu jako na obrázku 7.2.



Obrázek 7.2 Stromová struktura aplikace pro příklady č.1 – 3.

7.6.1 Příklad č.1 – transformace do tří modulů

Na obrázku 7.2 máme tři Controllers, každý Controller rozdělíme do jiného modulu. Anotaci @Block přidáme do deklarace třídy s parametrem patřící do daného modulu:

```
1 @Block("key1")
2 public class ControllerA { ... }
3
4 @Block("key2")
5 public class ControllerB { ... }
6
7 @Block("key3")
8 public class ControllerC { ... }
```

ve výsledku se nám v prvním modulu vytvoří soubory: ControllerA, ServiceA, ServiceB, ModelA, ModelB, DAO_A a DAO_B. Druhý modul bude obsahovat: ControllerB, ServiceC, ModelB, ModelC, DAO_B a DAO_C. U třetího modulu bychom postupovali úplně stejně jako u prvních dvou modulů.

7.6.2 Příklad č.2 – více parametrů

U anotace @Block můžeme používat jeden až tři parametry. Pro přidání ControllerA do prvního a druhého modulu :

```
1 @Block({"key1", "key2"})
2 public class ControllerA { ... }
```

kde na pořadí parametrů nezáleží, takže bychom mohli napsat:

```
1 @Block({"key2", "key1"})
2 public class ControllerA { ... }
```

Použití s třemi parametry bude ukázáno v následujícím příkladě.

7.6.3 Příklad č.3 – transformace s různými metodami

Jak již bylo zmíněno rozhraní nebo třídu můžeme rozdělit do modulů s různými metodami. To zajistíme přidáním anotace @Block do deklarace metod. Představme si třídu, která bude vypadat následovně:

```
1  @Block({"key1", "key2"})
2  public class ControllerA {
3
4    @Block({"key1", "key2", "key3"})
5    public void push(int x){ ... }
6
7    @Block({"key1", "key2"})
8    public void pop(){ ... }
9
10   @Block("key2")
```

```
11    public boolean isEmpty(){ ... }
12
13    public void print(){ ... }
14
15    @Block("key3")
16    public void clear(){ ... }
17  }
```

v deklaraci třídy jsou dva parametry key1 a key2, takže se ControllerA vygeneruje do prvního a druhého modulu. V prvním modulu bude mít ControllerA metody:

- push(int x)
- pop()
- print()

zatímco ve druhém modulu bude mít navíc metodu isEmpty(). Parametr key3 v metodě push(int x) bude ignorován, protože parametr key3 není v deklaraci třídy. To samé se stane s metodou clear(), která má stejný parametr key3. Ten je zvolen špatně a metoda nebude nikde vygenerována. V tomto případě bude prototyp v logu varovat, že metoda clear() není dostupná ze žádného modulu.

7.7 Použití prototypu

Použití prototypu můžeme rozdělit do dvou fází. V první fázi zkopírujeme balíček **block**, který se nachází v prototypu, do cílové aplikace, kterou chceme nechat rozdělit. Poté libovolně umístíme anotaci @Block do deklarací třídy, rozhraní, pořípadně metod, které se nacházejí v poslední vrstvě cílové aplikace. Nezáleží, jestli prototypu předáme cestu k cílové aplikaci relativní nebo absolutní. Máme dvě možnosti buď změnit třídní proměnnou **FILE_APP_PATH** ve třídě **Main** a spustit prototyp příkazem:

```
1  mvn exec:java
```

nebo spustíme prototyp s jedním argumentem:

```
1  mvn exec:java -Dexec.args="argument"
```

kde za „argument“ dosadíme cestu k cílové aplikaci. Prototyp transformuje do modulů pouze Java soubory, jedinou výjimkou je soubor **pom.xml**, který je při nalezení automaticky přidán do všech modulů. Pokud tedy chceme do modulů přidat i jiné pomocné soubory, musíme je zvlášť deklarovat v poli členské proměnné **files**. Kdybychom například chtěli přidat do všech modulů soubor data.sql, vypadalo by to následovně:

```
1  private static final String[] files = {"data.sql"};
```

V defaultním nastavení prototyp generuje veškeré soubory do adresáře **generatedFiles/**, který se nachází v kořenovém adresáři prototypu. Toto nastavení můžeme změnit přidáním druhého a třetího argumentu při spuštění prototypu. Druhým argumentem se mění cesta, kam se ukládá konfigurační soubor, třetím argumentem se mění cesta umístění vygenerované transformované aplikace.

Prototyp tedy můžeme spustit s jedním až třemi argumenty. Prvním argumentem určujeme umístění cílové aplikace. Druhý argument slouží pro změnu umístění konfiguračního souboru, v tu chvíli můžeme třetí argument vynechat. To bohužel není možné, když chceme změnit umístění vygenerované aplikace. V tom případě musíme doplnit první i druhý argument.

7.8 Testování

Testování prototypu bylo provedeno ve dvou fázích. V první fázi bylo testování realizováno jednotkovými testy, které byly implementovány pomocí frameworku JUnit. Ve druhé fázi bylo testování realizováno na demonstrační aplikaci z kapitoly 6.

7.8.1 Jednotkové testy

JUnit je framework pro jednotkové testy psané v programovacím jazyce Java[18].

Jednotkové testy se nacházejí v adresáři **test/**, které využívají pomocné soubory z adresáře **testFiles/**. Pro každou třídu byla vytvořena testovací třída, která byla pojmenována podle testované třídy s přidáním přípony **Test**. Výjimkou je třída **Main**, která obsahuje pouze jednu statickou metodu **main**, která spouští prototyp.

Implementace jednotkových testů

V jednotkových testech byly pro porovnávání použity funkce **assertBool** pro funkce, které vrací návratovou hodnotu **Boolean**, a funkce **assertEquals** pro ostatní.
Funkce **assertBool** se používá se dvěma vstupnímy parametry: popisek chybové hlášky a výraz, který chceme otestovat. Funkce **assertTrue** považuje za správnou hodnotu **true**, funkce **asertFalse** poté hodnotu **false**. V jiném případě test selže a vypíše chybovou hlášku.
Funkce **assertEquals** se používá se třemi parametry: popisek chybové hlášky, předpokládaný výraz a testovaný výraz. Funkce otestuje, jestli se předpokládaný a testovaný výraz rovná, pokud ne, test selže a vypíše chybovou hlášku.

Popisek chybové hlášky u funkcí **assertBool** a **assertEquals** je nepovinný a může být vynechán, ovšem je dobré popisek doplňovat pro jednodušší orientaci při selhání testu.

Každý jednotkový test byl otestován minimálně jednou porovnávací funkcí **assertBool** anebo **assertEquals**. Pro ukázku byl zvolen test metody **getImportsImplementsExtendedFromClass()** ze třídy **ContainerClassCU**, která má za úkol vrátit všechny importy, dále implementující rozhraní a děděné třídy:

```
1 @Test
2 public void getImportsImplementsExtendedFromClass() throws
     FileNotFoundException {
3
4 ContainerClassCU classCU = new
     ContainerClassCU("$pathToOneSpecificFile");
5
6 list.forEach(l -> assertTrue("list␣does␣not␣contain␣" + l,
     classCU.getImportsImplementsExtendedFromClass().contains(l)));
```

```
7 assertEquals("Wrong␣size␣of␣list!!!", list.size(),
      classCU.getImportsImplementsExtendedFromClass().size());
8 }
```

kdy jsme si nejdříve do listu přidali všechny prvky, které se nachází v souboru $pathToOneSpecificFile (kvůli délce zde neuvádím přesnou cestu k souboru). Poté jsme postupně procházely list a zjišťovali jsme, jestli se daný prvek také nachází v listu, který vrací zkoumaná metoda. Nakonec jsme otestovali, jestli se počet prvků v listu a počet prvků ze zkoumané metody shodují, tím jsme měli zaručeno, že list, který vracela zkoumaná metoda, obsahoval pouze nutné prvky.

Pokrytí jednotkových testů

Pokrytí jednotkových testů bylo automaticky vygenerováno frameworkem JUnit. Vygenerovaná data jsou obsažena v tabulce 7.1.

Název	Třídy [%]	Metody [%]	řádky [%]
ParserClass	100 (1/1)	96 (25/26)	88 (176/199)
MethodParser	66 (2/3)	100 (7/7)	93 (42/45)
ContainerClassCU	85 (6/7)	100 (25/25)	98 (89/90)
FileHandlerParser	100 (1/1)	90 (9/10)	76 (70/91)
RESTConfigFileHandler	75 (3/4)	92 (12/13)	82 (56/68)
ClassWithMethods	100 (1/1)	66 (4/6)	47 (10/21)

Tabulka 7.1 Tabulka pokrytí jednotkových testů.

Tabulka obsahuje čtyři sloupce. První sloupec obsahuje název testované třídy. Poté obsahuje popořadě procentuální pokrytí tříd (respektive využitých konstruktorů pro testování), metod a řádků.

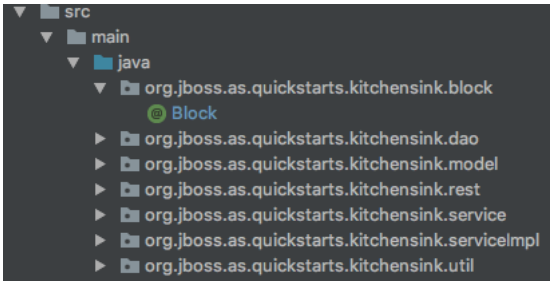
7.8.2 Testování na demonstrační aplikaci

Testování na demonstrační aplikaci bylo prováděno manuálně, ve kterém jsme vyzkoušeli celý proces od přidání anotací **@Block** do demonstrační aplikace a její transformace, spuštění frontendu a vygenerovaných modulů až po vyzkoušení, že se frontend dokáže připojit a získat data z distribuované verze demonstrační aplikace, tedy ze spuštěných modulů.

Nejdříve bylo potřeba připravit demonstrační aplikaci na transformaci do modulů. Pro používání anotací **@Block** v aplikaci jsme museli přidat balíček **block** do struktury aplikace. Nachází se ve struktuře prototypu a obsahuje pouze rozhraní **Block**. Ukázka přidání balíčku **block** do struktury je vidět na obrázku 7.3.

Poté jsme umístili anotace **@Block** s libovolnými parametry (key1, . . . , key3) do deklarací **Controllers**, které představují poslední vrstvu aplikace a implementují rozhraní REST. Také jsme vyzkoušeli přidat anotaci **@Block** do deklarace metod v **Controllers** pro otestování správného rozdělení metod.

Nyní je demonstrační aplikace připravená pro transformaci a zaměříme se na nastavení prototypu. Pro spuštění prototypu použijeme příkazovou řádku. Prototyp se



Obrázek 7.3 Ukázka přidání balíčku **block** do demonstrační aplikace.

spouští minimálně s jedním argumentem, který určuje cestu k demonstrační aplikaci. Víme, že prototyp transformuje do modulů pouze Java soubory, proto musíme explicitně vyjmenovat pomocné soubory, které chceme do modulů přidat. V našem případě se jedná o tři soubory **kitchensink-quickstart-ds.xml**, **persistence.xml** a **beans.xml**, ve kterých se nachází datasource a driver k připojení k databázi. Přidáme je tedy do členské proměnné ve třídě **Main**:

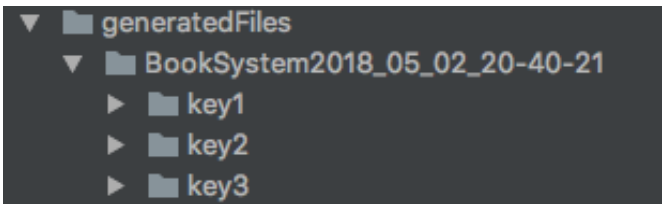
```
1 private static final String[] files =
    {"kitchensink-quickstart-ds-xml", "persistence.xml",
    "beans.xml"};
```

V defaultním nastavení prototyp generuje všechny soubory do adresáře **generatedFiles/**. To nám nevadí, ale konfigurační soubor chceme použít ve frontendové části, proto při spuštění prototypu přidáme i druhý argument určující adresář, kam se konfigurační soubor vygeneruje. V tomto případě bude spuštění prototypu s argumenty vypadat:

```
1 mvn exec:java -Dexec.args="$demoAppFile␣
    $pathToGeneratedConfigFile"
```

kde za **$demoAppFile** dosadíme cestu k demonstrační aplikaci a za **$pathToGeneratedConfigFile** dosadíme cestu, kam se má konfigurační soubor vygenerovat.

V této chvíli se nám vygenerovala transformovaná demonstrační aplikaci do adresáře **generatedFiles/**, kde se adresářová struktura nachází na obrázku 7.4.



Obrázek 7.4 Adresářová struktura transformované aplikace do modulů.

Z obrázku 7.4 si můžeme všimnout, že prototyp pojmenovává hlavní adresář jménem transformované aplikace a přidává časové razítko, kdy byla transformovaná aplikace vygenerována, jako příponu. Jednotlivé moduly jsou poté pojmenovány podle parametru z anotace **@Block**, ve kterých se nachází podmnožina adresářové struktury demon-

stračním aplikace.

Spustíme frontendovou část serveru příkazem:

```
1 npm start
```

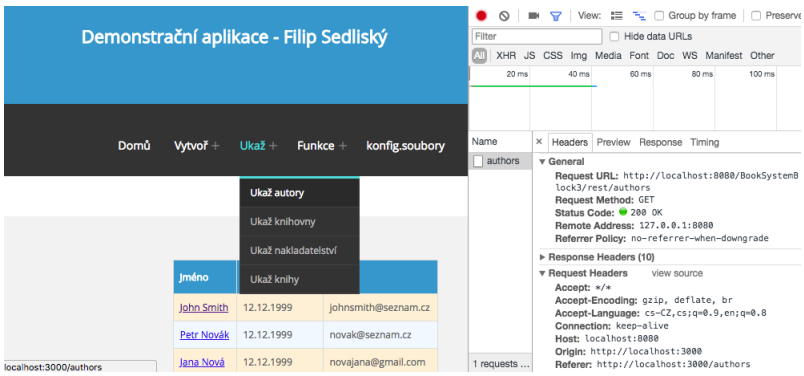
nezapomeňme, že musíme nastavit ve frontendu proměnnou **isMonolit** na hodnotu **false**, aby se připojoval k distribuovaným verzím. Poté spustíme aplikační server WildFly, na kterém poběží všechny backendové servery, ať už jako monolit nebo distribuované verze. Postupně zkusíme spustit jednotlivě vygenerované moduly a zjistíme, jestli vše proběhlo vpořádku. Pokud ano, prototyp správně transformoval demonstrační aplikaci a pokud ne, jsou dva různé důvody. Prototyp špatně transformoval aplikaci, například při chybějícím Java souboru v modulu nebo jsme špatně nastavili rozdělení v prototypu, například jsme zapomněli specifikovat pomocný soubor aplikace.
Moduly můžeme spustit pomocí vývojového prostředí nebo pomocí příkazové řádky v adresářové struktuře modulu příkazem:

```
1 mvn clean package wildfly:deploy
```

V našem případě vše proběhlo vpořádku a povedlo se nám spustit všechny tři transformované moduly, které postupně poslouchají na adrese:

- http://localhost:8080/BookSystemBlock1/rest/
- http://localhost:8080/BookSystemBlock2/rest/
- http://localhost:8080/BookSystemBlock3/rest/

V tuto chvíli máme vše připravené a můžeme vyzkoušet, že se frontend dokáže připojit k různým spuštěným modulům, tedy k různým backendovým serverům. Frontend je spuštěn na adrese: **http://localhost:3000** a můžeme ji otevřít v libovolném webovém prohlížeči. Otevře se nám hlavní stránka s menu, se kterým můžeme procházet aplikaci. Poté otevřeme „Nástroje pro vývojáře“ (například ve webovém prohlížeči Chrome) a zvolíme záložku **Network**. V této záložce jsou všechny síťové přenosy, které byly uskutečněny. Kdybychom procházely jednotlivé požadavky, zjistili bychom, že všechny byly provedeny úspěšně. Obrázek 7.5 demonstruje jeden požadavek ze strany frontendu, který se připojil ke třetímu backendovému serveru.



Obrázek 7.5 Snímek obrazovky ukazující vykonaný požadavek při stisknutí tlačítka „Ukaž autory“.

Na levé straně obrázku 7.5 je ukázka uživatelského rozhraní frontendu. Při stisknutí tlačítka „Ukaž autory“ se provede požadavek na server **http://localhost:8080/BookSystemBlock3/rest/authors**, který vrátí všechny autory ve formátu JSON. Frontend přijme odpověď a vypíše všechny autory do tabulky. Z požadavku je vidět, že se frontend připojil ke třetímu modulu a ze stavového kódu **200 OK** zjstíme, že požadavek proběhl úspěšně.

Důkladnější automatické testování demonstrační aplikace proběhlo v následující kapitole.

8 Zátěžové testy Monolit vs. Distribuované verze

V této kapitole se podíváme na výkonnostní rozdíly demonstrační aplikace v monolitu a distribuované verze z kapitoly 6. K automatickému testování byl vybrán nástroj Gatling.io, který byl představen v sekci 5.2. K podrobení testu byl vybrán jeden use case, který testoval aplikaci v monolitu a v distribuovaných verzích s paralelní vytížeností:

- 1 uživatel
- 5 uživatelů
- 20 uživatelů
- 50 uživatelů
- 300 uživatelů

Ke snížení měřící chyby byl každý test 5x zopakován a z toho vypočítán průměr. Výsledná data se nacházejí v tabulce 8.1.

8.1 Use case

K otestování výkonnosti demonstrační aplikace byl vybrán use case, který testoval načítání dat s občasným zápisem do databáze. V testování aplikace nebyly vytvářené objekty, protože demonstrační aplikace nemá implementované smazání objektů a docházelo by k nárustu dat v databázi a tím pádem k různým naměřeným hodnotám.
Use case, kterým byla otestována demonstrační aplikace:

1. ukaž všechny autory
2. ukaž detaily o prvním autorovi
3. ukaž všechny knihovny
4. ukaž detaily o první knihovně
5. ukaž všechny nakladatelství
6. ukaž detaily o třetím nakladateli
7. ukaž všechny nakladatelství
8. ukaž detaily o druhém nakladateli
9. ukaž všechny knihy
10. ukaž detaily o čtvrté knize
11. ukaž detaily o druhém autorovi
12. ukaž detaily o čtvrté knize
13. ukaž všechny knihy
14. ukaž všechny nakladatelství
15. první nakladatelství vydá první knihu

8.2 Data měření

Data měření představují vždy dvojici celkový průměr doby odezvy (response time) a její standardní odchylku. Dále bylo každé měření 5x zopakováno pro snížení měřící chyby.

Monolit vs. Distribuovaná verze – čas odpovědi						
	Monolit		Dva moduly		Tři moduly	
Počet uživatelů	Průměr [ms]	Standardní odchylka[ms]	Průměr [ms]	Standardní odchylka[ms]	Průměr [ms]	Standardní odchylka[ms]
1	27,8	19,2	28	17,8	40,2	31
5	35,4	32,4	38	43,4	44,2	46,2
20	52,6	85,4	56	93,2	58	96,4
50	74,8	163,6	75,4	160,2	75,2	159,2
300	817,2	1000,2	804,6	981,6	754,7	923

Tabulka 8.1 Naměřené hodnoty z měření Monolit vs. Distribuovaná verze.

8.3 Závěr měření

Z výsledných dat v tabulce 8.1 můžeme vypozorovat, že monolit má lehce lepší response time (čas odpovědi) při nižší zátěži. Na druhou stranu distribuované verze dosahovaly lepších výsledky u vyšší zátěže.

9 Závěr

V bakalářské práci byla prozkoumána obecná struktura aplikace v technologii Java EE, podle které byla navrhnuta implementace prototypu flexibilní modularizace.

Prototyp byl implementován v jazyce Java a využívá knihovny JavaParser. Byl navržen způsob označení rozdělované aplikace pomocí anotace @Block, která se používá s jedním až třemi parametry. Podle anotace prototyp transformuje aplikaci do dvou nebo tří spustitelných modulů.

Dále byla představena demonstrační aplikace, na které byla otestována funkčnost vyvíjeného prototypu. Demonstrační aplikace byla podrobena zátěžovému testu, ve kterém jsem porovnával výkonnost monolitu proti distribuovaným verzím, které byly rozděleny nejdříve do dvou a poté do tří modulů.

Prototypem jsme chtěli dokázat, že je možné automaticky rozdělit aplikaci v Javě EE bez většího úsilí. Bohužel prototyp je schopný transformovat pouze určitý vzorek aplikací, které mají obdobnou strukturu jako například naše demonstrační aplikace.

Literatura

[1] Nicholas Smith, Danny van Bruggen a Federico Tomassetti. *JavaParser:Visited*. LeanPub, 2016 - 2017.

[2] Ira R. Forman a Nate Forman. *Java Reflection in Action (In Action Series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[3] Victor Winter, Carl Reinke a Jonathan Guerrero. "Certifying a java type resolution function using program transformation, annotation, and reflection". In: *Springer Science+Business Media New York* (2014).

[4] Niklas Boije a Kristoffer Borg. "Semi-automatic code-to-code Semi-automatic code-to-code transformer for Java". Dipl. Linköping University, Department of Computer Science, 2016.

[5] M.Fowler et al. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002.

[6] Martin Fowler. "Reducing Coupling". In: *design* (2001).

[7] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.

[8] Cerny T. a Donahoo M.J. "On separation of platform-independent particles in user interfaces". In: *Cluster Comput* (2015).

[9] Spring. *T.: 3.2 million servers vulnerable to jboss attack (2017)*. 2017. URL: `https://threatpost.com/3-2-million-servers-vulnerable-to-jboss-attack/117465/`.

[10] URL: `https://reactjs.org`.

[11] URL: `https://gatling.io/`.

[12] Mark Massé. *REST API Design Rulebook*. Ed. Simon St. Laurent. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2012.

[13] Cerny T. a Donahoo M.J. "Survey on Compromise-defensive System Design". In: *Information Science and Applications* (2018).

[14] 2018. URL: `http://hibernate.org/orm/`.

[15] URL: `http://www.oracle.com/technetwork/java/dataaccessobject-138824.html`.

[16] 2017. URL: `https://cw.fel.cvut.cz/old/_media/courses/a4b77ass/ass2_2017.pdf`.

[17] URL: `http://tutorials.jenkov.com/java/annotations.html`.

[18] URL: `https://junit.org/junit5/`.

Příloha A

Seznam použitých zkratek

API	Application Programming Interface
AST	Abstract Syntax Tree
CRUD	Create, Read, Update, Delete
DAO	Data Access Object
DOM	Document Object Model
DOS	Disk Operating System
HATEAOS	Hypermedia as the engine of application state
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
Java EE	Java Platform, Enterprise Edition
JDBC driver	Java Database Connectivity driver
JSON	JavaScript Object Notation
MS windows	Microsoft Windows
ORM	Object-relational mapping
REST	Representation State Transfer
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

Příloha B

Obsah CD

- thesis.pdf – elektronická verze textu bakalářské práce
- thesisFlexMod.zip – zdrojové soubory bakalářské práce
- javaparser.zip – zdrojové soubory prototypu flexibilní modularizace
- BookSystemFrontend.zip – zdrojové soubory frontendové části serveru demonstrační aplikace
- BookSystemBackend.zip – zdrojové soubory backendové části serveru demonstrační aplikace