VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SPRÁVA PŘEPÍNÁNÍ PROCESŮ V APLIKACI JENKINS

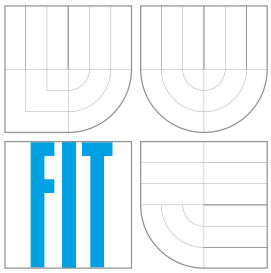
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LIBOR ONDRUŠEK

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SPRÁVA PŘEPÍNÁNÍ PROCESŮ V APLIKACI JENKINS

SUSPENDING JENKINS JOB EXECUTION IN FAVOR OF HIGH-PRIORITY JOBS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LIBOR ONDRUŠEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. RADEK KOČÍ, Ph.D.
SUPERVISOR

BRNO 2012

Abstrakt

Projekt se zabývá návrhem a řešením zásuvného modulu pro *Jenkins server*, umožňujícího řízení běžících úkolů na uzlech v clusteru serverů pro systém průběžné integrace. Modul by měl umožnit pozastavení probíhajícího dlouhotrvajícího procesu, vykonání procesu s vyšší prioritou a následného obnovení původního procesu v místě, kde byl přerušen.

Abstract

The project deals with design and solution of plug-in for a *Jenkins server*, which allows management of jobs running on the nodes in the cluster of continuous integration system servers. This plug-in should allow pause the long time job to, done process with higher priority and restore original job in state where it was interrupted.

Klíčová slova

Průběžná integrace, Jenkins, Cluster, Java, JVM, Procesy

Keywords

Continous integration, Jenkins, Cluster, Java, JVM, Processes

Citace

Libor Ondrušek: Správa přepínání procesů v aplikaci Jenkins, diplomová práce, Brno, FIT VUT v Brně, 2012

Správa přepínání procesů v aplikaci Jenkins

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Radka Kočího Ph.D. a Ing. Vojtěcha Juránka Ph.D.

.....
Libor Ondrušek
21. května 2012

Poděkování

Děkuji za pomoc a vstřícnost Ing. Juránkovi Ph.D. z firmy Red Hat Česká republika, vedoucímu práce Ing. Kočímu Ph.D. za výborný přístup k problematice a všem kteří mi během vytváření práce pomáhali.
V neposlední řadě bych také rád poděkoval své manželce Martině za trpělivost a pochopení mé zaneprázdněnosti během psaní této práce.

Obsah

Seznam obrázků

Seznam zkratek

SCM Source Control Management – systém pro správu verzí software

HTML Hyper Text Markup Language – hypertextový značkovací jazyk

API Application Programming Interface – rozhraní pro programování aplikací

JVM Java Virtual Machime – virtuální stroj, kde lze spouštět Java aplikace

GUI Graphical User Interface – grafické uživatelské prostředí

FIFO First In First Out neboli dřív z fronty odchází to co přišlo do fronty dřív

JNLP Java Network Launching Protocol

JNI Java Native Api – sada nástrojů pro volání funkcí implementovaných pomocí hostitelského systému, nakterém je spuštěna JVM

JDK Java Development Kit – Sada nástrojů pro kompilaci a úpravy Java zdrojů

POM Project Object Model – XML deklarace sestavení zdrojových kódů

XML Extensible Markup Language – rozšiřitelný značkovací jazyk

OS Operační systém

MVC Model View Controller – Architekrura zobrazovací vrstvy, kde jsou odděleny komponenty pro strukturu dat, vytváření cílového zobrazení a kontrolu zobrazení

JSON Datově orientovaný strukturovaný formát dat ve formě textu

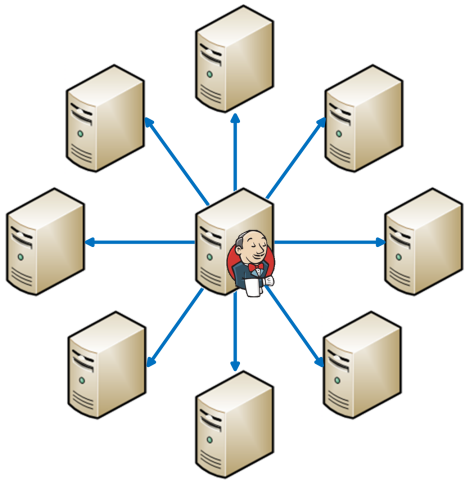
Seznam ukázek kódu

Kapitola 1

Úvod

Tato práce se zabývá řešením problematiky přepínání procesů v *Jenkins serveru*, který je jednou z nejpopulárnějších a nejpoužívanějších implementací metodologie zvané průběžná integrace.

Zadání pochází od firmy Red Hat Česká republika se sídlem v Brně. Tato firma používá pro sestavování svých produktů implementovaných převážně v Javě ze skupiny *JBoss Enterprise Middleware* právě tento systém průběžné integrace, čítající obsáhlý cluster serverů, jenž se starají o kompilaci, testy, sestavení a vydávání desítek projektů patřících právě do JBoss Enterprise Middleware.



Obrázek 1.1: Cluster sestavovacích serverů

I přes použití velkého, ale omezeného množství výkonných uzlů v clusteru, nastávají situace, kdy probíhá dlouhotrvající úkol, který zaměstnává více uzlů, ale je nutné sestavit prioritnější projekt. Zpravidla jsou to komplexní testy, které se spouštějí periodicky a běží i několik dní. V této situaci pak není možné sestavit projekt, který musel být kupříkladu lehce poupraven, a poté v krátkém čase otestován, sestaven a vydán.

Řešením by sice mohlo být vytvoření nové farmy serverů, která by obsluhovala pouze prioritní a nenáročné úkoly, avšak to by znamenalo administrovat další systém, který je v podstatě stejný jako ten původní a udržování dvou systémy ve stejné konfiguraci by přinášelo jen problémy, časovou a finanční zátěž.

1.1 Cíl práce

Proto přichází na mysl řešení vytvořit zásuvný modul do Jenkins serveru, který bude moci řídit úkoly předávané na výkonné uzly clusteru. Řízení by mělo probíhat tak, že pokud budou zaměstnány všechny uzly clusteru, zásuvný modul vyhodnotí podle nastavení priority jednotlivých úkolů, který uzel se zastaví a spustí se vykonávání úkolu s vyšší prioritou. Po úspěšném či neúspěšném dokončení prioritního úkolu se původní úkol a spustí ze stavu, ve kterém byl přerušen.

Cílem této práce tedy bude vytvořit takový zásuvný modul, který poskytne výše uvedenou funkcionalitu, popřípadě upravit stávající serverovou část, tak aby bylo možné procesy řídit díky prioritě, nastavené, každému typu úkolu zvlášť.

Vyřešení popsaného problému přinese do Jenkins serveru novou možnost jak využívat efektivněji výkonné uzly během sestavování časově náročných projektů, což rozšíří jeho škálovatelnost o další úroveň. Zároveň pomůže firmě Red Hat v efektivním využívání jejich sestavy serverů, používaných systémem průběžné integrace, k sestavování produktů.

Takto vytvořené řešení se stane součástí projektu Jenkins, které je svobodným softwarovým produktem o jehož vývoj se stará komunita vývojářů. Tím bude toto řešení dostupné nejen zadavateli, ale i široké veřejnosti, kde může najít dobré uplatnění.

1.2 Navržený postup

Dalším postupem práce bude nutně provedení podrobné analýzy funkčnosti serveru. Důležité bude soustředit se na proces zpracování příchozího požadavku na sestavení, jeho podrobný průběh v závislostech na typu projektu, použitých zásuvných modulech a dalších nezbytných okolnostech, ovlivňujících průběh tohoto zpracování.

K pochopení funkcionality serveru bude navíc nutné pochopení základních principů metodologie průběžné integrace a její začlenění do vývojového procesu software.

Dále bude nutné pečlivě zvážit co bude na serveru rozšiřováno pomocí zásuvného modulu a co bude případně nutné doplnit nebo změnit ve stávajícím řešení serverové části. To ve zkratce znamená najít vhodný způsob a místo pro pozastavení prováděného úkolu a způsob, jak tuto skutečnost zobrazit ve webové aplikaci, případně ji promítnout do dalších rozhraní, kterými lze k serveru přistupovat.

Implementace zásuvného modulu bude zahrnovat doplnění nastavení priority úkolu, nastavení globálního přepínače a veškerou výkonnou logiku pro rozhodování, zda běžící procesy zastavovat nebo zůstat čekat ve frontě. Také bude vytvořena nová globální konfigurace, jenž bude řídit zda prioritizaci používat nebo ne (toto bude sloužit k dočasnému vyřazení funkcionality z provozu).

Implementované řešení bude také nutné otestovat jak automatickými testy, tak ručně pomocí spuštění úkolu v různých situacích a následné vyhodnocení chování zásuvného modulu a serveru jako celku.

Zhodnoceno v bodech:

- Analýza současné verze serveru se zaměřením na proces sestavení požadovaného úkolu.
- Nalezení vhodného místa k zastavené probíhajícího úkolu.
- Implementace zásuvného modulu.
- Testování na reálném systému s více než 20 projekty.

- Zhodnocení výsledků.

1.3 Členění dokumentu

Dokument je rozdělen do dvou logických hlavních celků. První (jedná se o kapitolu 1 až 3) se zabývá teoretickými poznatky metodologie průběžné integrace a analýzou architektury a funkcionality Jenkins serveru.

Druhá část (kapitoly 4 až 5) naproti tomu obsahuje popis a postup řešení zadaného problému. Lze zde nalézt podrobnou analýzu činnosti serveru během sestavování projektu (úkolu), popis navrženého řešení a v neposlední řadě také popis implementace řešení spolu s vysvětlením proč některé navržené způsoby řešení nebyly vhodné.

Závěrečné zhodnocení práce a popis dalších možných rozšíření jsou v poslední kapitole 6.

Přílohy jsou navíc doplněny o licenci, pod kterou je celý server vyvíjen, návod k použití implementovaného řešení a další nezbytné součásti.

1.4 Konvence

U této práce se můžeme setkat s různými styly textů a grafických úprav. Každý tento styl má určitý význam a popisuje jiný typ informace. Jedná se především o tyto:

- Krátké a věcné poznatky

Zdrojové kódy se mohou vyskytnout v několika formách. Pokud je krátký kód začleněn přímo do textu, bude vypadat takto: `int i = 0;`

Větší logické celky a části kódu jsou zvýrazněny v samostatném rámečku:

```
1 // komentar zdrojoveho kodu
2 int a = 15;
3 if(a < 5){
4   a = a + 1;
5 }
```

Ukázka 1.1: Ukázka zdrojového kódu

Popis konvencí by měl usnadnit orientaci v této práci a pomohou k rychlejšímu zapamatování si o čem pojednává.

Kapitola 2

Průběžná integrace

V dnešní době, kdy jsou kladeny vysoké nároky na spolehlivost softwarových produktů a rychlost vývoje, se nemohou vývojáři, kteří chtějí dosahovat komerčního úspěchu při vývoji software, spoléhat na staré metodiky sdílení zdrojových kódů a způsobů integrace, jako je například sestavování výsledného produktu na vývojových stanicích jednotlivých vývojářů.

V moderních softwarových společnostech tyto staré metodiky nahrazuje právě proces průběžné integrace neboli *CI*. Zjednodušeně řečeno, jedná se o proces používaný při vývoji softwarových produktů k dosažení nejvyšší možné spolehlivosti a konzistence výsledného produktu. Martin Fowler ji definuje jako metodu vývoje software v článku uvedeném na jeho blogu [6]. Fowler je také autorem oficiální definice průběžné integrace.

Tato technika vývoje software je vhodná pro integraci práce členů týmu, kteří dělají mnoho změn v kódu produktu. Tyto změny jsou do výsledného společného produktu zanášeny v krátkých časových intervalech a dosti často.

Pokud by se tato technika nepoužívala, bylo by nutné po dokončení všech částí *seskládat* projekt do jednoho celku, otestovat všechny části, zda spolu správně komunikují a zda celý produkt správně funguje vzhledem k okolním systémům a vstupům.

2.1 Proces integrace změn do celku

2.1.1 Vysvětlení pojmů

Průběžná integrace nejčastěji operuje s pojmem *build*. Jedná se o jedno konečné sestavení výsledného produktu. V podstatě jde o produkt samotný, jak bude dodávaný ke koncovému zákazníkovi. Každé takové sestavení musí být jednoznačně a unikátně označeno verzí, kvůli pozdější identifikaci ze strany zákazníka.

Ruku v ruce jde průběžná integrace s *jednotkovým a integračním testováním*. Tímto způsobem bývá obvykle ověřována základní funkcionalita a kvalita kódu jako takového. Jednotkové testování přitom kontroluje co nejmenší oddělitelné a samostatně testovatelné jednotky zdrojového kódu, jako jsou například metody nebo celé třídy. Tohoto testování se využívá hlavně pro ověření správnosti algoritmů různých pomocných tříd a ostatních bloků, které zpravidla transformují konkrétní vstupy na konkrétní výstupy. Naproti tomu integrační testování zahrnuje ověření vyvíjeného produktu při začlenění do systémů, ve kterých bude po dodání fungovat. Kontroluje se integrace a komunikace s okolními systémy, zdroji dat, případně dalšími zdroji. Při vývoji obvykle nebývá k dispozici funkční testovací prostředí, a také z hlediska konzistence tohoto případného prostředí, se při integračním testování používá technika zvaná *Mocking*, což je nahrazení konkrétních okolních zdrojů

proxy objekty, u kterých lze explicitně definovat chování a reakce na požadované akce a dotazy vyvíjeného produktu. Tuto techniku lze také s výhodou využít, pokud je produkt logicky dělen do vrstev, na sobě nezávislých, k nahrazení nižších vrstev.

Dalším důležitým prvkem je *verzovací systém*, který se stará o uchování změn v kódu a dalších zdrojích projektu. Data do tohoto systému vkládají jednotliví vývojáři, podle potřeb projektu. Jedno takové uložení práce se nazývá *commit*. Commit by měl splňovat několik požadavků. Měl by být logicky strukturovaný a po sloučení změn by měl být produkt sestavitelný v stejné nebo větší míře než před sloučením změn. Často se stává, že nové změny *rozbijí* výsledný produkt nebo ho změní tak, že neprojdou některé testy. V tomto případě systém průběžné integrace informuje o této skutečnosti nastaveným způsobem.

Opakem commitu je *checkout*, kde se získává zdrojový z verzovacího systému, většinou i s časovou známkou a hashem jednotlivých souborů pro správné rozpoznání konkurentního přístupu k datům.

Pokud se nepodaří vytvořit sestavení správně, systém průběžné integrace by měl toto oznámit původci změn v kódu a dalším pověřeným osobám. To se ale odvíjí podle nastavení procesu a řízení kvality v konkrétní společnosti.

2.1.2 Popis integračního procesu

Jak lze vidět na obrázku 2.1, je celý proces rozdělen do dvou asynchronních částí.

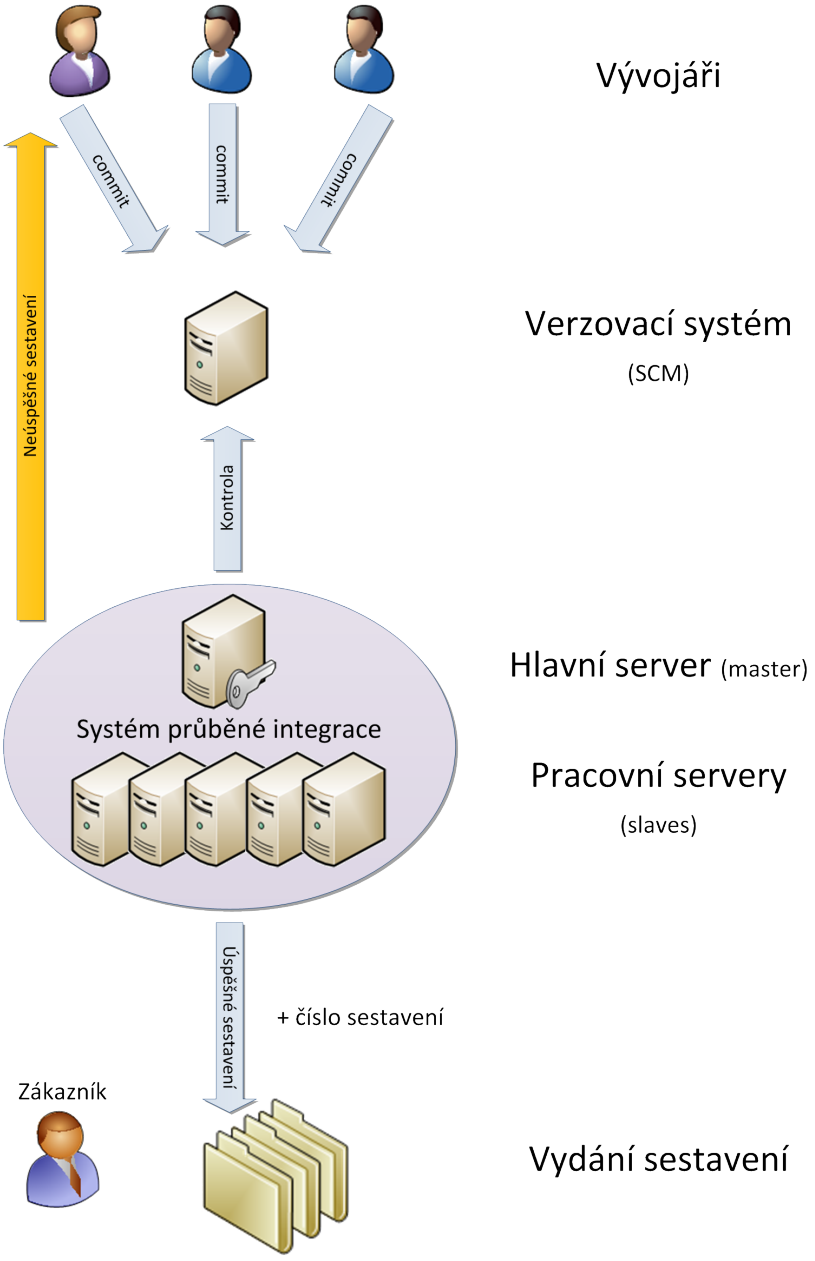
První část zahrnuje klasický postup při vývoji software, kdy vývojáři pravidelně slučují své dílčí změny pomocí verzovacího systému v jeden celek. Ukládání změn přichází paralelně. Pokud nejsou konfliktní, tzn. pokud je změna v jednom souboru provedena pouze jedním vývojářem a během změny nebyl pozměněn původní soubor jiným vývojářem, jsou přijaty do verzovacího systému.

Druhá část zahrnuje pravidelné nebo jiné kontroly verzovacího systému pomocí *hlavního serveru*. Pokud se od poslední kontroly udály nějaké změny, bude projekt (v prostředí průběžné integrace se tyto projekty nazývají anglicky *Job*) zpracován podle nastavení projektu. Zpravidla to obsahuje kompilaci, testování, kontrolu kódu a vystavení reportů, popřípadě ohlášení chyby. Zpracování může provést jeden hlavní server nebo tento jenom řídí proces a veškerou práci obstarávají *pracovní servery*. Generování souhrnných zpráv v tomto případě provádí také hlavní server.

Pokud bylo sestavení neúspěšné, to znamená, že některý z předchozích kroků selhal, je na základě nastavení projektu provedeno generování souhrnných zpráv a zaslání těchto zpráv některou z běžných cest elektronické komunikace. Zpravidla to bývá elektronickou poštou konkrétnímu vývojáři, který provedl poslední uložení změny do verzovacího systému a vedoucímu projektu. Moderní systémy průběžné integrace nabízejí většinou také rozšířené možnosti upozornění na problémy, jako je například krátká textová zpráva (SMS) nebo rozblikání výstražného majáku napojeného na systém průběžné integrace.

Velkou výhodou systémů průběžné integrace je to, že dokáží podle posledního přispěvatele do zdrojového kódu produktu rozlišit právě tohoto vývojáře a upozornit na případné chyby v sestavení právě jej, čímž nedochází ke zbytečnému obtěžování ostatních vývojářů. To bývá oceněno hlavně v případě rozsáhlého projektu, kdy by jinak byly rozesílány desítky mailů v případě chyby.

Každé sestavení, ať úspěšné nebo neúspěšné, je vedeno pod unikátním *číslem sestavení*, které přiděluje systém průběžné integrace a na základě kterého lze bezpečně rozlišovat, které sestavení bylo například naposledy dodáno koncovému zákazníkovi. Sestavení většinou obsahuje navíc unikátní verzi z verzovacího systému, což podporuje také systém průběžné



Obrázek 2.1: Schéma vytvoření sestavení pomocí systému průběžné integrace

integrace.

Na konci sestavení bývá zpravidla samostatný úkol, který vystaví finální sestavení na místo, odkud ho pak může získat koncový *zákazník*.

V současné době se vzrůstající popularitou *Cloud* služeb, které reflektují filosofii *Software as Service*, roste také popularita průběžného dodávání (*Continuous delivery*). Jelikož při tomto přístupu zákazník neodebírá produkt jako takový, ale služby produktem poskytované, může být každé úspěšné sestavení rovnou nasazeno na cílový server a nemusí se "ručně" dodávat zákazníkovi.

Kapitola 3

Jenkins server

Jednou z implementací výše popsaného přístupu průběžné integrace je *Jenkins server*. Původně byl určen pro integraci Java projektů, ale postupem času s přibývající popularitou rozšířil, díky systému zásuvných modulů, působnost i na projekty v jiných jazycích. Navíc ho lze díky jeho univerzalitě použít prakticky na jakoukoliv integrovatelnou úlohu, třeba až na úrovni nástrojů operačního systému. Více o zásuvných modulech v sekci 3.3.2

3.1 Historie

Jenkins server vznikl pod hlavičkou firmy *Sun Microsystems* původně jako *Hudson server*. Hlavním vývojářem byl Kohsuke Kawaguchi a téměř ihned po akvizici firmy Sun Microsystems firmou Oracle Kawaguchi odešel i s komunitou kolem Hudson serveru a založil vlastní vývojovou větev, která dnes čítá něco přes 120 přispěvatelů[1]. V tomto okamžiku získává projekt nové jméno *Jenkins server* i logo, protože firma Oracle odmítla věnovat název Hudson a jeho logo komunitě, tehdy vyvíjející ještě Hudson server.

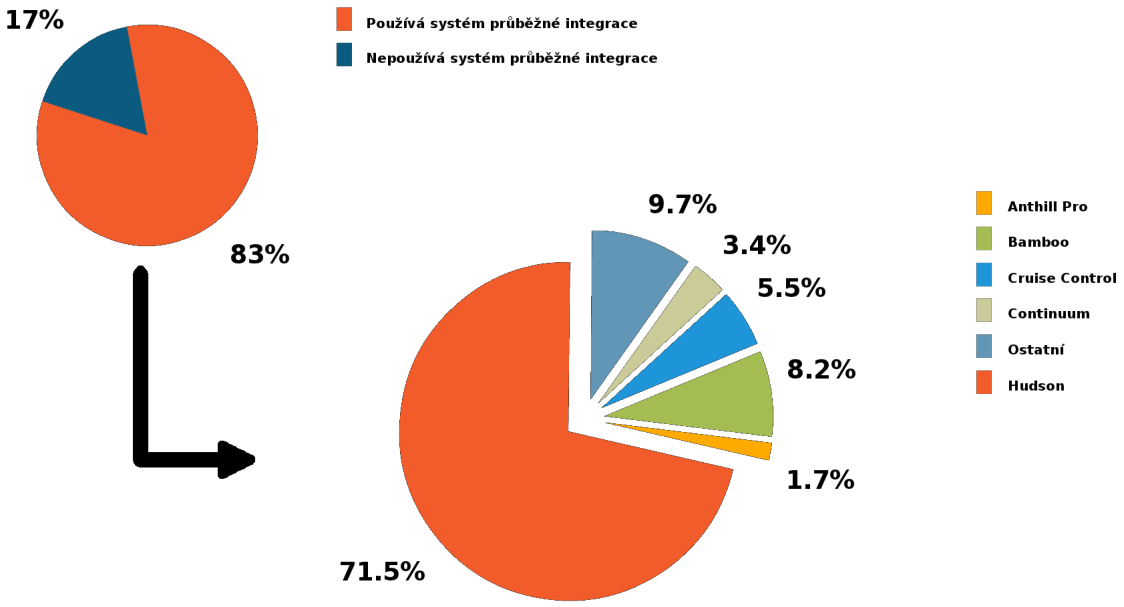
Projekt Hudson server je dnes udržován firmou Oracle, ale jeho rozvoj není zdaleka tak dynamický jako u Jenkins serveru. Přesné porovnání těchto parametrů mezi Hudsonem a Jenkinsem nejspíš neexistuje, jelikož firma Oracle nezveřejňuje statistiky o zdrojovém kódu Hudson serveru. V současné době přechází projekt Hudson pod vedení *Eclipse foundation*. Tato komunita bude dále rozvíjet Hudson server za pomoci současných přispěvatelů, mezi které mimo jiné patří i společnost *Sonatype, Inc.*

3.2 Současnost

V současné době je Jenkins server nejoblíbenějším a nejznámějším systémem průběžné integrace, jak zveřejnila společnost Sonatype v průzkumu z roku 2011. Průzkum se mimo jiné týkal vývojářů a používání systémů průběžné integrace. Z dotázaných vývojářů používá nějaký systém průběžné integrace 83 % respondentů a z toho 71.5 % používá právě Hudson server jak je vidět z obrázku 3.1, který byl převzat z výsledků výzkumu zveřejněných na internetu [7].

Tehdy to byl ještě Hudson, ale kolik subjektů zůstalo skutečně u Hudsonu není zcela zřejmé, ale má se za to, že většina přešla na Jenkins větev. Více v prezentaci výsledků průzkumu zveřejněného na internetu. [7]

[1] Aktuální údaj z úložiště zdrojových kódů https://github.com/jenkinsci/jenkins/contributors



Obrázek 3.1: Výsledky průzkumu společnosti Sonatype, Inc. z roku 2011

3.3 Architektura serveru

Jedná se o distribuované řešení, kdy můžeme použít jeden server jako hlavní *Master* a další servery jako jeho výkonné uzly *Slave*. Veškeré řízení sestavování a kontroly verzovacích systémů obstarává výhradně hlavní server, který v případě potřeby spouští pracovní servery, aby vytvořili sestavení. Generování souhrnných zpráv výsledků těchto sestavení má pak opět na starost hlavní server.

Je to dobře škálovatelné řešení, kde navíc všechny výkonné uzly nemusí pracovat nad stejným operačním systémem. Můžeme se tak setkat s hybridními systémy průběžné integrace, kde se sestavují Java projekty společně s .NET projekty.

Pokud je potřeba sestavit nějaký projekt, hlavní uzel vybere volný výkonný uzel a předá mu požadavek na sestavení projektu. Jakmile je sestavení dokončeno, ať úspěšně nebo neúspěšně, je toto oznámeno hlavnímu uzlu, který na základě výsledku sestavení provede nastavené akce.

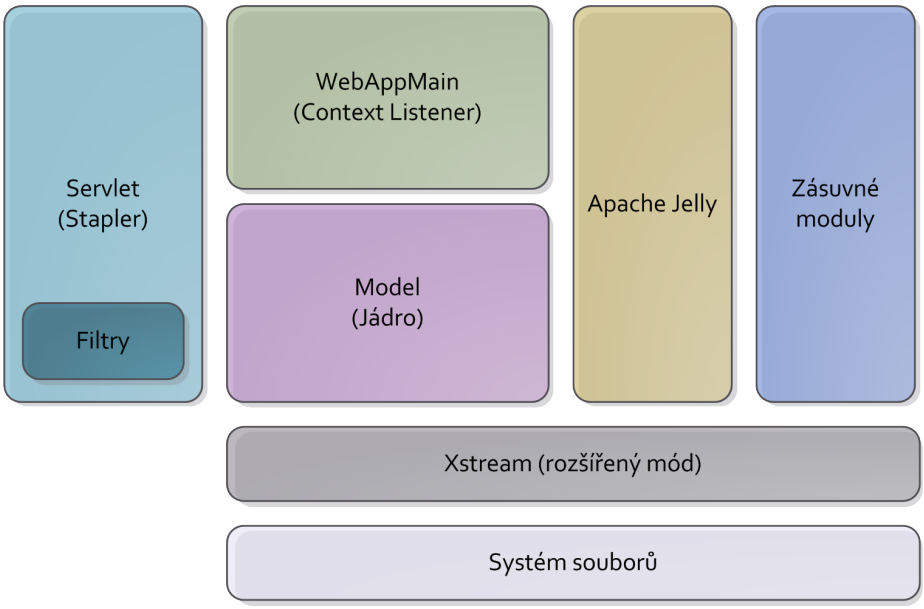
Komunikace mezi uzly probíhá na síťové vrstvě pomocí TCP protokolu. Na výkonný uzel se neinstaluje Jenkins server, jak by se mohlo zdát, ale pouze se nastaví, jakým způsobem je tento server fyzicky dosažitelný a hlavní uzel na něj odešle potřebné knihovny a nakonfiguruje agenta, starajícího se o spouštění potřebných úkolů. Pomocí nastavení lze dosáhnout různých způsobů využívaní těchto výkonných uzlů. Lze zvolit strategii "vše na výkonných uzlech" nebo využít také hlavní uzel pro samotné sestavování.

Pomocí zásuvných modulů může Jenkins, na základě požadavku o sestavení a nedostatku volných kapacit na současných uzlech, vytvořit nový výkonný uzel. Typ uzlu a přístup k němu se odvíjí podle použitého zásuvného modulu. Jelikož lze takto vytvářet uzly i na komerčních cloudových službách jako je například Amazon EC2 [2] dává to Jenkins serveru velkou flexibilitu co se týká distribuovaných sestavování a možnosti testování na různých

[2] Amazon Elastic Compute Cloud viz. http://aws.amazon.com/ec2/

operačních systémech.

Z hlediska architektury je Jenkins server standardní servletový Java EE server, jak je patrné z obrázku 3.2. Je koncipován jako jednoduchý server, obsahující výkonné jádro, rozhraní pro připojení zásuvných modulů a perzistentní vrstvu operující na souborovým systémem.



Obrázek 3.2: Architektura Jenkins serveru

Servlet je implementován třídou `Stapler` a jedná se o implementaci servletu podle standardu Java EE. Hlavním cílem tohoto servletu je mapování struktury Java balíků a tříd do jednoznačné URL adresy. Tento servlet je načítán při spuštění serveru do webového kontejneru a je vytvořen jeho kontext.

WebApplicationMain je *Context Listener*, pro výše zmíněný kontext servletu a obstarává vytvoření a spuštění jádra aplikace. Po startu provede několik nezbytných kontrol důležitých komponent a vytvoří instanci třídy `Jenkins`.

Model tvoří výkonné jádro serveru. Obsahuje důležité třídy zajišťující hlavní funkcionalitu serveru. Podrobnější popis je v sekci 3.3.1.

Apache Jelly se stará o šablonování pohledů HTML stránek. Více k Jelly lze nalézt na stránkách Apache komunity [2].

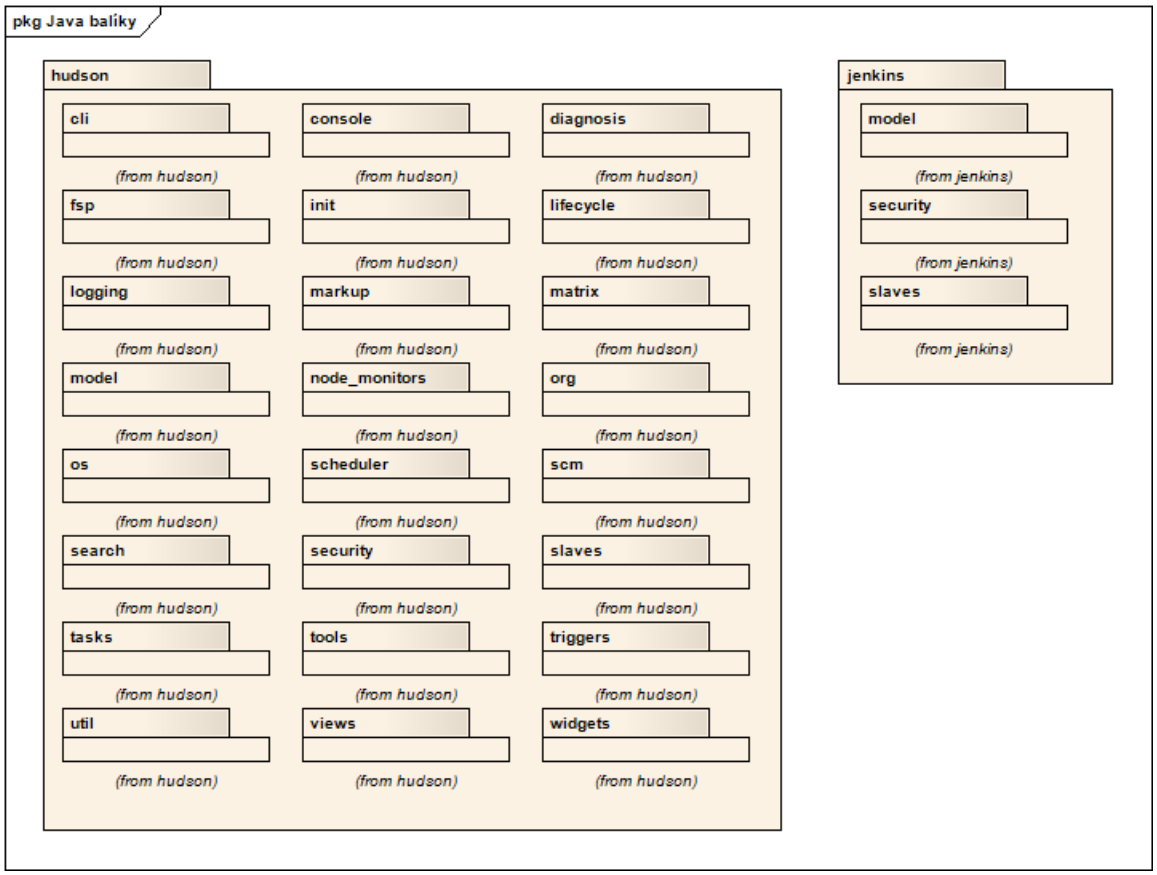
Zásuvné moduly rozšiřují funkcionalitu jádra serveru o nové sestavovací pravidla, nastavení, spouštěče sestavení, monitorovací nástroje, reportovací nástroje a jiné.

XStream Tato komponenta se stará o perzistenci potřebných dat na souborový systém pomocí *XStream* knihoven. Jde o ukládání dat do XML souborů pomocí datových toků používající *Java reflection API* pro rozpoznávání parametrů objektů pro perzistenci. Některá data jsou ale i přes to ukládána jako prostý text. Jedná se především o výstupy z konzolí a výstupy logovacích komponent.

3.3.1 Model serveru

Při podrobnějším pohledu na komponentu modelu serveru je patrné, že hlavní složku zde tvoří třída `Jenkins`. Od této třídy dědí třída `Hudson`, která se zde nachází kvůli zachování zpětné kompatibility zásuvných modulů.

Rozčlenění kódu do Java balíků je podobné jako u hlavních tříd. Na nejvyšší úrovni se nacházejí balíky jenkins a hudson viz. obrázek 3.3.

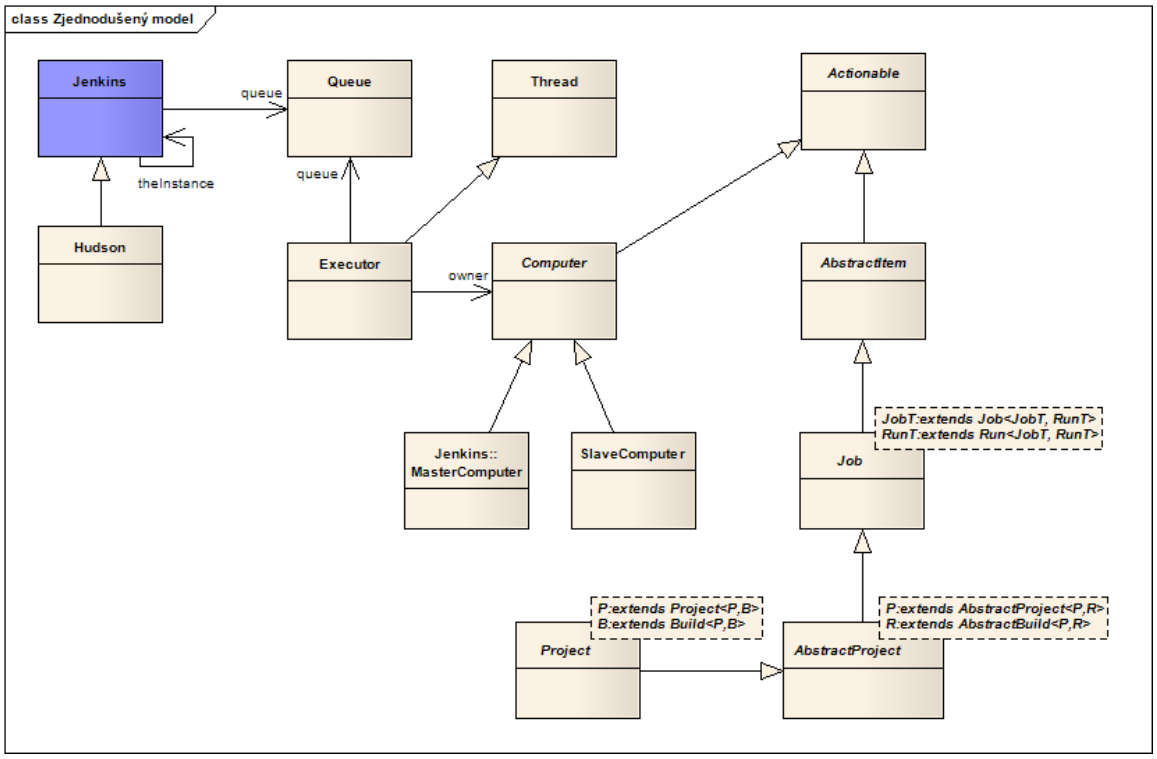


Obrázek 3.3: Rozdělení do Java balíků

Jak je vidět z obrázku, *Jenkins server* je poskládán ze značného množství různých balíků a komponent. To ho dělá dosti rozsáhlým, ale díky pečlivosti při návrhu, také srozumitelným.

Při podrobnějším pohledu na infrastrukturu jádra lze nalézt podobnosti mezi prvními dvěma balíky, což je také známka snahy o zachování zpětné kompatibility. Na tu nicméně Jenkins server dbá důrazně, jak je popsáno v dokumentu o Jenkins serveru [4]. Snaží se o zpětnou kompatibilitu co nejdále do minulosti Jenkins i Hudson serveru.

Dále se budeme věnovat popisu balíku `hudson`, jelikož balík `jenkins` v současnosti obsahuje jen zlomek komponent, které nejsou zásadní pro pochopení stavby a struktury serveru.



Obrázek 3.4: Zjednodušený model serveru

3.3.2 Zásuvné moduly

Objektový model Jenkins serveru je, jak už bylo zmíněno, jednoduše rozšiřitelný, pomocí zásuvných modulů, které umožňují po připojení k jádru serveru rozšiřovat nebo upravovat možnosti serveru.

Aby se předešlo konfliktům při zavádění modulů, protože vývojářů těchto modulů je mnoho, jednotlivé zásuvné moduly se nahrávají do oddělených zavaděčů tříd. Zavedené moduly mají pak přístup do jádra serveru, díky výše popsané třídě `Jenkins`, k perzistentní vrstvě. Podporují také generování HTML stránek pomocí integrované technologie Apache Jelly, přístup ke zdrojům webové prezentační vrstvy, jako jsou například obrázky nebo *JavaScript*.

Tento přístup je výhodný v tom, že uživatel nepozná rozdíl, mezi tím co je implementováno v jádru serveru nebo v zásuvném modulu, což je cílem techniky používající rozšíření implementované právě systémem zásuvných modulů.

Typy zásuvných modulů

Jenkins definuje abstraktní a rozhraní, která mohou být rozšířena zásuvnými moduly. Tyto entity definují co všechno lze rozšířit nebo nahradit. Všechny tyto entity spojuje nutnost implementace rozhraní `ExtensionPoint`. Takto rozšiřitelných entit je v jádru serveru kolem stovky. Navíc lze rozšiřovat i zavedená rozšíření a to ať moduly dodávané s jádrem, tak moduly od ostatních vývojářů.

Konkrétní registrovaná místa, které lze rozšiřovat jsou popsána podrobněji na Jenkins

wiki [3] stránkách.[3]

Díky použití moderních technik pro registraci rozšíření a možnosti registraci míst, která lze rozšiřovat, dává Jenkins server velmi širokou základnu pro úpravy sestavovacích procesů, přesně podle požadavků speciálních projektů. Většina projektů sice asi vystačí s tím co poskytuje základní dodávaná verze serveru spolu se současnou množinou zveřejněných rozšíření, ale takto široká možnost úprav z tohoto produktu dělá výborný nástroj pro použití ve velkých softwarových společnostech, které mívají ve většině případů dost netradiční požadavky na vývojový proces a tudíž i na sestavování svých výsledných produktů.

3.4 Podrobná analýza serveru

Pro vyřešení problému je nutné pochopit do detailu jak Jenkins server funguje a jakým způsobem probíhá požadavek skrze celý server a také jeho komponenty, které nemusí být nutně vždy na stejném fyzickém zařízení jako běžící server. Tento průběh je zajímavý od vzniku požadavku až po jeho interpretaci v JVM nebo v procesu operačního systému.

Ačkoliv se může zdát průběh požadavku jasným a jednoduchým, opak je pravdou, jelikož server dovoluje mnoho rozšíření. Z toho plyne fakt, že celá architektura serveru musí být postavena tak, aby toto bylo možné, což může zanášet do objektového modelu více abstrakce, nutnost striktního používání rozhraní při předávání parametrů a obalování modelu různými objekty.

Dalším důležitým aspektem je, že server vyvíjí komunita, což znamená že do kódu přispívá více programátorů a ne každý dokáže přesně držet původní myšlenky architekta tohoto sytému. Mimo jiné některé problémy nelze vyřešit jinak než porušením zavedených konvencí i za cenu toho že se kód může zdát poněkud nečitelný. Dokonce v kódu lze najít podobné *nepěkné* způsoby řešení i od hlavního autora, vždy s komentářem: “Bohužel jinak to nešlo.”

Jenkins server jako takový je komplexní řešení, postavené na moderních technologiích a přístupech k řešení problémů. Rozvoj řešení má na starosti hlavní vývojář *Kohsuke Kawaguchi*, který udává styl a trend ve vývoji serveru. Jeho přístup je v dnešní době trochu zvláštní, jelikož se snaží používat co nejméně knihoven třetích stran a raději si i rutinní záležitosti píše *zcela sám*. Nutno podotknout, že se mu to daří docela úspěšně a díky tomu je jeho kód dobře čitelný, nezbytně komentovaný a snadno pochopitelný.

Server celý se skládá z několika samostatných projektů, které jsou na sobě závislé, jen v nejmenší nutné míře.

Z těch nejdůležitějších projektů vyvíjených Jenkins komunitou jsou to:

jenkins-core Jádro serveru.

jenkins-war Kompletace výsledného `jenkins.war` souboru.

cli Řádková utilita pro ovládání serveru.

maven-plugin Sestavování projektů pomocí *Apache Maven 2* a *Apache Maven 3*.

test Testy částí serveru.

remoting Komunikace s výkonnými uzly a sebou samým. souboru.

[3] Konkrétně na https://wiki.jenkins-ci.org/display/JENKINS/Extension+points

Z pohledu funkcionality lze server rozdělit na dvě větší části. První část je konfigurační, uživatelská a skládá se ze schématu úkolů, které si může uživatel nakonfigurovat na serveru. Druhá část je výkonná a jde o abstrakce výkonných jednotek serveru. Tato část je pro řešení zadaného problému velmi důležitá, neboť pro správný chod se musí doplnit a zásadně upravit. Původní záměr na vytvoření pouze čistého zásuvného modulu proto nemůže být naplněn, jelikož je nutné obohatit základní logiku celého serveru na dosti nízké úrovni abstrakce, což zásuvný modul nedovolí.

Tento fakt plyne z dosud zjištěných informací o možnostech rozšiřování serveru. To je omezeno pouze na objekty, které jsou implementací rozhraní `ExtensionPoint` nebo se používají způsobem který dovoluje registraci rozšíření. Je jasné že bude nutné upravit minimálně třídu `Executor` a ta toto rozhraní neimplementuje. Navíc zřejmě dojde i na změny v komunikaci mezi JVM při vykonávání jednotlivých úkolů, tudíž lze předpokládat že takovéto změny nelze postihnout pouze pomocí zásuvných modulů.

3.4.1 Konfigurační část serveru

Tato část Jenkins serveru je v podstatě to co vidí uživatel. Jeden úkol odpovídá jednomu typu požadavku uživatele na činnost serveru. Úkoly jsou konfigurovatelné a možnosti konfigurace jsou přímo závislé na nainstalovaných zásuvných modulech. Úkol lze tedy chápat také jako konfiguraci jednotky činnosti serveru.

Třída `Jenkins`

Třída `Jenkins` je implementována podle návrhového vzoru singleton a v systému tudíž existuje pouze jedna instance této třídy. Tak ji mohou lehce využívat ostatní komponenty a hlavně zásuvné moduly, jelikož tato instance poskytuje přístup k ostatním komponentám a objektům celého serveru.

Třída `Project`

Představuje nastavení a agregátora objektů, které nesou informaci o vykonání určité činnosti vedoucí k splnění kompletního procesu sestavení. Tyto objekty jsou instance třídy `Builder` a implementují rozhraní `BuildStep`, představující jeden abstraktní krok sestavení (*build*) výsledného produktu. Z pohledu vývojového procesu produktu tuto fázi lze přirovnat ke kompilaci zdrojového kódu a linkování výsledných objektů do výsledné *fyzické* podoby produktu.

Objekty třídy Project dále vlastní kolekci instancí třídy `Publisher`, které se starají o post sestavovací kroky. Jsou taktéž jako builder implementací rozhraní `BuildStep`, tudíž z pohledu logiky serveru jsou rovnými objekty.

Další důležitou součástí jsou pomocné objekty, pro nastavení potřebných interních i externích parametrů pro každé sestavení. Jsou to objekty instanciované z třídy `BuildWrapper`. Jejich metody pro nastavování parametrů jsou volány vždy po uložení projektu uživatelem, aby se zajistilo správné nastavení prostředí pro následné sestavení produktu. Navíc se tyto objekty volají vždy před zahájením činnosti a po ukončení práce builderu, se kterým jsou svázány. Pokud dojde k selhání příslušného builderu, tento pomocný objekt se postará o uvedení prostředí zpět do konzistentního stavu, jaký byl těsně před spuštěním konkrétního builderu.

Po spuštění projektu se postupně spustí všechny výše uvedené objekty definované v kolekcích. Nejprve *wrappery*, pak *buildery* a nakonec *publishery*. Tím se provede jedno sestavení, které tvoří jednotku na nejvyšší úrovni.

Projekty lze navíc sdružovat do skupin pomocí objektů `ItemGroup`. To je vhodné pro seskupování větších sestavitelných celků, jako jsou například POM soubory pro projekty sestavované pomocí *Apache Maven*[1] nástroje, které mohou tvořit strom, pokud se jedná o více-modulový projekt.

Třída `AbstractBuild`

Tato abstraktní třída je předkem všech různých typů sestavení. Pod pojmem sestavení si lze představit sadu dílčích příkazů a úkonů, jež je nutné vykonat ve správném pořadí tak, aby byl naplněn cíl takového sestavení.

Sestavení pak může být v serveru celá řada. Od nejjednoduššího, které je tzv. *ve volném stylu*, kde se v podstatě veškerá činnost konfiguruje ručně pomocí volání operací a nástrojů operačního systému, až po složité typy sestavení jako je `MavenBuildSet`. V něm je naopak seznam úkonů již dopředu určen v závislosti na nastavení životního cyklu, jenž je třeba dosáhnout (viz. 4.3.2) a dalších parametrů, které jsou nastaveny přímo v definičních souborech daného projektu.

Třída `Command a rozhraní Callable`

Tyto dvě třídy spolu úzce souvisí. V podstatě se jedná o podobnou funkcionalitu s tím rozdílem že třída `Command` zahrnuje příkazy spíše systémového charakteru, kdežto třída `Callabale` uživatelského charakteru.

V obou případech se ale jedná o nejmenší možnou jednotku komunikace serveru, na kterou se rozkládají jednotlivé úkony sestavení. Takovýchto příkazů je velké množství, protože každé rozšíření operuje většinou se svojí vlastní sadou příkazů a z důvodů dobré oddělitelnosti těchto rozšíření se nepoužívají z jiných než standardních rozšíření.

Třída `CommandTransport`

Objekty toho typu jsou důležité z hlediska vytváření komunikačního kanálu. Jedná se v jednoduchosti o obal spojení. Pro představu lze o těchto objektech uvažovat jako o obalech paketů na úrovni TCP spojení s tím rozdílem, že nepotřebují znát cílovou adresu, ta je určena nastavením a při zakládání spojení je již známa objektům, které spojení iniciují.

Zjednodušeně jde o *přepravní* jednotku pro posílání příkazů a objektů implementující rozhraní `Callable`.

3.4.2 Výkonná část serveru

Jak je popsáno výše, Jenkins server má možnost pracovat nad skupinou fyzických zařízení.

Toto jsou ve valné většině servery nebo osobní počítače. Model serveru se snaží tuto skutečnost reflektovat v největší možné míře, tak aby se co nejvíce přibližoval skutečnosti a tím poskytoval dobrou možnost pro pochopení funkcionality jednotlivých komponent modelu.

Níže popsané komponenty jsou výtahem toho nejdůležitějšího z obsahu *fyzické* části serveru.

Třída `Computer`

Modeluje jeden fyzický uzel. Pro *Master* uzel je společnou abstrakcí, která se drží v objektu `Jenkins`. Pokud se jedná fyzicky o výkonný uzel, neboli *Slave*, je tento model o něco přesnější, neboť se jedná o skutečné vzdálené fyzické zařízení.

Tento model sebou nese informace o nastavení zařízení, jako je pracovní prostor, instalované JDK, způsob použití daného zařízení, přístupovou metodu, pomocí které se s tímto zařízením spojuje master uzel, plánování kdy má být zařízení dostupné a v neposlední řadě počet instancí třídy `Executor`, vykonávající vlastní sestavení konkrétního úkolu.

Způsob použití zařízení je důležitý pro rozložení výkonu celé farmy zařízení, jelikož ve standardní sestavě zásuvných modulů lze používat konkrétní uzel buďto jako arbiter pro rozdělování jednotlivých úkolů nebo jako vlastní výkonný uzel. Pokud je uzel nastaven jako arbiter, tak se na něm žádný úkol nebude spouštět, ale bude delegovat všechny úkoly na ostatní uzly. V opačném případě se jedná o obyčejné zařízení, které pouze vykonává úkoly předávané mu přímo hlavním uzlem nebo sebou samým, pokud se jedná přímo o master uzel. Nutno podotknout, že hlavní uzel se zapojuje až tehdy, když všechny výkonné uzly jsou naplno zaneprázdněny.

Přístupových metod, jak se spojit se zařízením je ve základu serveru několik a existuje velké množství rozšíření, které tuto množinu zvětšují. Mezi základní metodu patří spojení pomocí JNLP tak že se na výkonném uzlu spustí aplikace uložená v souboru `slave.jar` (po spuštění serveru je přístupný na adrese `<server>/jnlpJars/slave.jar`) s příslušnými parametry, které ukazují ma master uzel a tím se vytvoří spojení. Tento příkaz je možné spustit také z master uzlu bez dalších argumentů.

Důležitým parametrem objektů typu `Computer` je počet instancí tříd `Executor`. Tento nastavitelný počet umožňuje využívat plný potenciál výkonného uzlu nebo naopak omezit využívané prostředky na tomto zařízení, tak aby mohlo být využíváno i pro jiné účely.

Třída `Executor`

V modelu se jedná o vlastní výkonnou jednotku, jejíž hlavní účel spočívá v získání čekajícího úkolu z fronty požadavků a vykonání všech dílčích činností, které jsou k tomuto úkolu aktuálně nakonfigurovány.

Z procesního hlediska se jedná o samostatné vlákno, které se vytvoří při startu serveru a *zaparkuje* se do stavu, kdy neprovádí žádnou činnost. Seznam zaparkovaných vláken se udržuje v objektu `Queue`, což je fronta úkolů právě pro vykonání pomocí objektů typu `Executor`. Z pohledu obsluhy takto vytvořených vláken se jedná o tzv. *Thread Pool*[9], což je návrhový vzor, definující jak se mají chovat vlákna ve vícevláknových aplikacích. Především se jedná o obsluhu spouštění z řízení množství spustitelných a běžících vláken, jejich životní cyklus, uspávání a probouzení k činnosti.

Pokud je nutné provést úkol z fronty, úkol se vyzvedne, z *poolu* se uvolní příslušné vlákno a úkol se provede. Nakonec se vlákno vrátí do *poolu*, kde se uspí. O nutnosti provedení úkolu informuje fronta objekt, který se stará o rozdělování požadavků mezi jednotlivé exekutory a v Jenkins serveru je implementován třídou `LoadBalancer`. Ten vybere prvního volného exekutora, pomocí implementovaného algoritmu a předá mu informaci, že ve frontě čeká úkol na provedení.

Executor pak podle konfigurace serveru vytvoří komunikační kanál na konkrétní fyzické zařízení. Pokud se jedná o jediné fyzické zařízení, kanál se vytváří také, ale jde o implementaci lokálního kanálu. To dává dobré možnosti pro rozšiřování funkcionality, protože

z tohoto pohledu se chová Jenkins stejně ke vzdáleným uzlům tak i sám k sobě jako master uzlu.

Třída `Channel`

Komunikační kanál je implementován třídou `Channel` a jedná se o spojení mezi dvěma JVM. Kanál je obousměrný, což znamená že objekty na koncích kanálu jsou zároveň konzumenti i producenti. Tím se zjednodušuje komunikace, jelikož obě strany jsou tak v rovnoprávném postavení a tento kanál lze využívat pro zasílání požadavků i čtení odpovědí.

Samotný kanál je tvořen dvěma nezávislými vlákny, z nichž jedno slouží ke zasílání požadavků a druhé slouží ke čtení požadavků producenta z druhé strany kanálu. Pro čtecí vlákno je to objekt typu `ReaderThread` a pro vlákno určené k zapisování je to `ExecutorService`.

Fyzicky se zasílání zpráv provádí pomocí objektového proudu, tedy se jedná o objekty typů `InputSteam` a `OutputStream`. Těmito proudy lze posílat všechny objekty, které implementují rozhraní `Callable`, protože pro provedení takto přeneseného objektu se využívají funkce tohoto rozhraní. Jmenovitě jde o metodu `call()` z tohoto rozhraní.

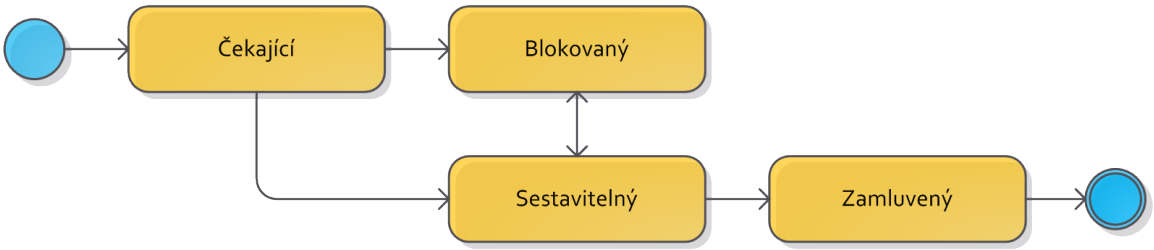
Třída `Queue`

Architektonicky stojí třída `Queue` nad objekty typu `Computer` i `Executor`, jelikož je v celém serveru vždy jenom jedna a v posloupnosti provádění jednotlivých úkolů je také před těmito komponentami. Jedná se o implementaci FIFO fronty bez priorit. Objekty, které do této fronty přichází jsou obaleny objektem `Item` a jeho deriváty pokud bude objekt prováděn ihned nebo pokud bude čekat ve frontě.

Fronta prošla za dobu vývoje serveru pár změnami, z nichž nejpodstatnější je asi změna přidávání úkolů. Zatímco dříve se jednalo opravdu čistě o frontu, do které se přidávají úkoly, dnes plní takříkajíc funkci plánovače, jelikož k přidávaným úkolům přibyly ještě čekací periody mezi jednotlivými úkoly.

Dalším důležitým faktem je že fronta nepřijímá duplicitní úkoly. To znamená, že pokud je ve frontě úkol určitého typu reprezentovaný projektem, při pokusu o naplánování dalšího úkolu stejného typu, nebude tento naplánován a bude serverem zahozen.

Úkol se dostává při průchodu frontou do několika různých stavů. Po naplánování se dostane do stavu čekání a je uložen v seznamu `waitingList`. Odsud se může dostat dál do stavu, kdy je sestavitelný nebo do stavu, kde se zablokuje další provádění z důvodů nedostatku zdrojů pro sestavení projektu. Pokud se zdroje po čase uvolní, přejde do sestavitelného stavu a pokračuje dál. Odsud putuje úkol do stavu kde je vyzvednut exekutorem, ale ještě se nezapočalo provádění. Jakmile je exekutor plně připravený, provede daný úkol, respektive všechny akce s ním spojené.



Obrázek 3.5: Stavy úkolu ve frontě

3.4.3 Průběh sestavení jednoho projektu

Zpracování projektu pomocí Jenkins serveru může být zahájeno z různých míst. Jeden ze způsobů může to být http požadavek přímo z webové vrstvy, který zpracovává abstraktní třída `AbstractProject` pomocí metody `doBuild(...)`.

Dalším možným způsobem jak začít sestavovat projekt je změna v SCM a následná periodická kontrola tohoto úložiště. Pokud v SCM nastala nějaká změna, která je akceptována Jenkins serverem, naplánuje se nový úkol.

V tuto chvíli Jenkins pracuje s objektem typu `Project`, ve kterém má podrobný popis toho co bude nutné vykonat pro správné sestavení projektu.

Jak bylo popsáno výše, naplánování sestavení probíhá pomocí fronty. Fronta přijímá projekt a pracuje s ním skrze rozhraní `Task`. Jelikož třída `Project` implementuje toto rozhraní, je možné s projektem takto dál pracovat aniž by se ztratila jakákoliv informace o projektu.

Ve frontě prochází úkol stavy, které jsou zobrazeny na obrázku 3.5. Během přechodu ze stavu sestavitelný do stavu zamluvený rozhodne komponenta `LoadBalancer` o tom který exekutor bude nejvhodnější pro sestavení právě obsluhovaného projektu.

V této fázi se také projekt obaluje objektem `JobOffer`, který má za úkol sloužit jako prostředník mezi frontou a exekutorem a vytváří se pro každého exekutora zvlášť. `JobOffer` obsahuje důležitou komponentu, která je typu `OneShotEvent` a má na starosti synchronizaci vláken respektive exekutorů. Je-li nutné vlákno zastavit volá se metoda `block()`, pro návrat k činnosti metoda `signal()`. Předání úkolu exekutorovi probíhá tak, že exekutor je informován o skutečnosti že ve frontě čeká úkol na sestavení a může se pokusit si ho vyzvednout.

Vyzvednutí exekutorem z fronty probíhá pomocí metody `pop()` nad frontou. `JobOffer` zabalí celý úkol do objektu `WorkUnit` a takto ho připraví k vyzvednutí z fronty.

Na řadě je zpracování exekutorem. Ten již byl díky volání fronty vytažen ze *zaparkovaného* stavu do činnosti a může zpracovat zadaný úkol. Mimo jiné si zaznamená čas startu provádění pro podrobné statistiky a převede úkol nejprve pomocí metod ve třídě `AbstractProject` na potomka třídy `AbstractBuild` a posléze na objekt typu `Executable`, který sám o sobě implementuje rozhraní `Runnable`, které definuje něco co bude interpretováno v samostatném vlákně. Tím se dostáváme na úroveň skutečného požadavku, který bude proveden pomocí exekutora. V kódu je doslovně napsáno: "Represents the real meat of the computation run by Executor"[5].

Odsud je úkol ve formě rozhraní `Runnable` předán zpět objektu typu `Queue`, respektive jejímu rodiči `ResourceController` k provedení takto předzpracovaného úkolu. Ačkoliv se může zdát že se tento úkol vrací zpět do fronty, není tomu tak. `ResourceController` se v tomto případě pouze využívá k řízení zdrojů a k zanesení informace že konkrétní exekutor se používá. Zavoláním metody `run()` úkolu se započne vlastní provádění úkolu.

V tento moment se řízení přenáší do konkrétní implementace úkolu. Přesněji řečeno do implementace metody `run()` abstraktní třídy `AbstractBuild`, kterou jsou povinni implementovat všichni potomci této třídy. Implementace této metody obsahuje ve většině případů pouze delegaci na metodu `run()` z objektu typu `Run`, úzce spojeného s objektem typu `Job`. Tato volaná metoda potřebuje jako jediný parametr objekt typu `Runner`, deklarovaný abstraktně jako vnitřní třída třídy `Run`. Jelikož se jedná o abstraktní třídu, je nutné ji implementovat přímo v konkrétním builderu.

Nyní je řízení zanořeno až do objektu `Run` s konkrétní implementací `Runner` objektu, který má implementovanou metodu `run(...)`, `post(...)` a `cleanUp(...)`. Tyto metody

se postupně volají v pořadí jak byly uvedeny. Pro další běh je nejvíce důležitá metoda `run(...)`.

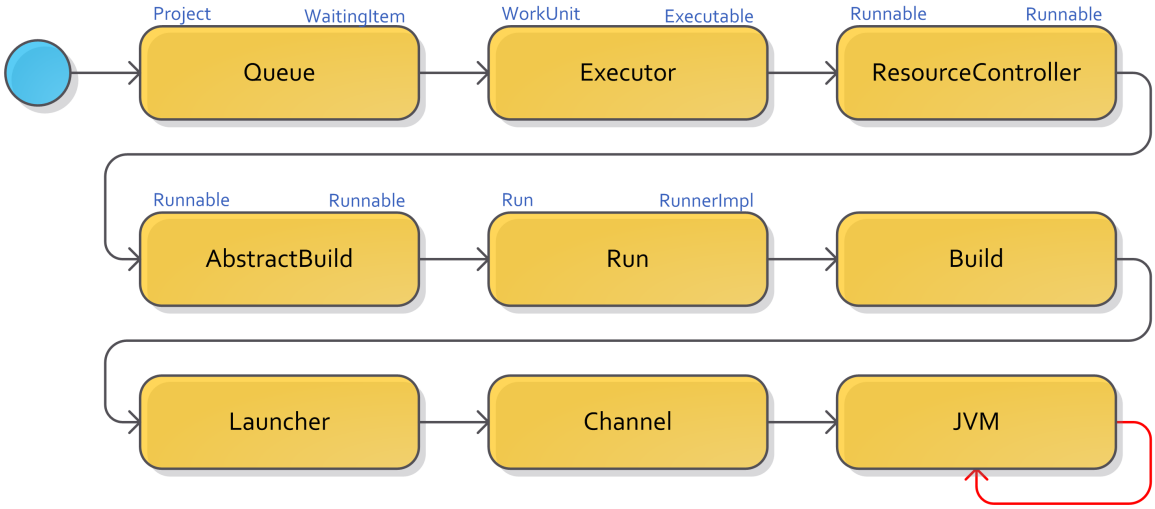
Ve své implementaci zjistí aktuální fyzický stroj, kde se právě spouští (`Node`) a na základě tohot vytvoří třídu `Launcher`, která obaluje rozdíly mezi lokálním spouštěním a vzdáleným spouštěním. Nakonec se volá metoda `doRun(...)`, která vrací řízení zpět do konkrétního builderu, jelikož musí být nutně překryta v objektu typu `Runner`. V této metodě pak probíhá vlastní sestavení všech potřebných kroků k tomu aby se mohl projekt jako celek sestavit na lokálním nebo vzdáleném stroji. Konkrétně například u *Maven* projektu se zjišťuje zda se jedná o strom projektů, kde je nastaveno lokální maven úložiště, nastavení systému, verze nativního programu Maven a podobně. Tyto příkazy se vykonávají skrze objekt `Launcher`, proto je jedno zda jsou určeny pro lokální nebo vzdálený stroj.

V praktickém pohledu vykonávání příkazů skrze objekt `Launcher` znamená, že se pro vykonání příkazu použije objekt třídy `Channel`, který je vytvořen jak pro všechna vzdálená spojení, tak pro všechna lokální spojení. V kódu je tento rozdíl smazán použitím společného rozhraní `VirtualChannel`, které deklaruje metody `call(...)` a `asyncCall(...)`. Tyto metody přijímají jediný argument typu `Callable`, což je konkrétně rozhraní popisující jak má vypadat objekt, jenž lze volat pomocí komunikačního kanálu a spustit jej na cílové JVM.

Komunikace skrze vytvořený a otevřený kanál probíhá pomocí objektů, jak bylo popsáno výše. Princip toho je velmi jednoduchý a je standardně implementován přímo v Javě. Jedná se o obyčejnou serializaci a deserializaci objektů. Serializace ve svém smyslu znamená převod objektu na text tak, aby se dal následně sestavit původní objekt. Zpětné sestavení je právě zmiňovaná deserializace. Tudíž pokud bereme v potaz že se komunikace skrze kanál provádí pomocí objektů, pak se ty to objekty serializují do textu, přesněji řečeno do pole bytů a toto se pak následně pomocí síťové nebo jiné vrstvy přenáší na vzdálený stroj. Při pohledu na surové přenášené data se jedná o konkrétní Java *bytekód*, do něhož je daná třída přeložena.

Jakmile dorazí požadavek na druhou stranu kanálu, deserializuje se zpět na objekt typu `Callable` a zavolá se jeho metoda `call()`. Po provedení této metody se vrací výsledek volání zpět skrze komunikační kanál. Tento princip se výrazně podobá principům používaných pro RPC a není se čemu divit, jelikož z něj vychází.

Vše je názorně shrnuto na následujícím obrázku i s vyznačením typů vstupujících a vystupujících do entit. na obrázku je pochopitelně pouze cesta požadavku jedním směrem. Jakmile se požadavek na cílové JVM provede, vrací se výsledek zpět. To naznačuje ona červená šipka. Pro vyšší přehlednost nejsou návratové přechody zakresleny, ale jedná se o klasické volání metod, tudíž je jasné že se výsledky vrací zpět k objektu, jenž původní dotaz vyvolal.



Obrázek 3.6: Schématický průběh sestavení požadavku

Kapitola 4

Analýza problému

Jak již z bylo popsáno v úvodní kapitole 1, hlavní ohnisko problému spočívá v zastavení procesu, běžícího pod JVM , spuštění nového procesu a následné obnovení původního procesu s ošetřením všech variant chyb, které se mohou vyskytnout.

Řešení musí být natolik univerzální, aby nebylo nutné upravovat existující zásuvné moduly a pokud možno ani serverovu část. Pokud to nutné bude, musí být úpravy v co nejmenší možné míře. Důležité je aby zůstala zachována původní architektura a stabilita serveru a řešení se příliš neodklonilo od původní myšlenky serveru.

4.1 Dekompozice problému

Celý problém by se dal rozdělit do několika důležitých rovin. Hlavní část problému nespočívá v zastavení běžících procesů, ale v nalezení správného místa, kde bude toto zastavení možné. Pro zastavení procesu bude nutné znát dobře problematiku vláken v Javě a obecně problematiku konkurentního přístupu.

Pro nalezení tohoto *správného* místa bude potřeba dobře chápat životní cyklus projektu uvnitř serveru a to dopodrobna od stavu, kdy se projekt nakonfiguruje, až do místa kde se provádí vlastní operace definované konfigurací projektu. pro tyto účely se musí také provést analýza nejdůležitějšího a nejpoužívanějšího zásuvného modulu, a sice *Maven modulu*. Je to z toho důvodu, že je natolik komplexní, aby postihnul problematiku většiny ostatních zásuvných modulů a navíc je typ projektu *Maven* nejvíce používaný co se týče typů projektů v Jenkins serveru.

Dalším velmi důležitým ohniskem problémů jsou procesy operačního systému, které může uzel spouštět a u nichž také může čekat na dokončení a navrácení hodnoty výsledku. Z tohoto důvodu bude nutné zjistit jak se k procesům OS přistupuje v Jenkins serveru a chápat jak lze s těmito procesy operovat v rámci JVM.

V neposlední řadě bude nutné vyřešit změny prezentační vrstvy serveru, neboli webového uživatelského rozhraní, které je implementováno pomocí šablonovací technologie *Apache Jelly*.

4.2 Java Virtual Machine

Jelikož celý *Jenkins server*, je postaven nad jazykem a běhovým prostředím Java, je nutné dobře chápat, jak se chová toto prostředí k procesům a vláknům, které v něm vznikají.

Jak již název napovídá, JVM je virtuální prostředí, které poskytuje běhovou podporu pro aplikace vytvořené programovacím jazykem Java a dalšími jazyky na Javě založenými. V podstatě může v JVM běžet cokoliv, co je schopné převést svůj zdrojový kód na tzv. *bytekód*. Ten je definovaný přesně ve specifikaci popisující JVM, napsané hlavními pracovníky firmy Sun Microsystems, která jazyk Java vyvinula. [8]

4.2.1 Běžící procesy

Procesy v JVM jsou členěny do vláken, které mohou běžet paralelně. Zda jde o čistý paralelní běh nebo pseudo paralelismus záleží na hardware, na němž běží JVM, ale JVM je vytvořena tak, aby podporovala čistý paralelní běh více vláken v závislosti na počtu jader procesoru.

Ve spuštěné JVM běží standardně několik vláken, které mají na starosti udržovat JVM v dobré kondici, co se týče paměti a monitorují další spouštěné vlákna. Jedná se zejména o *Garbage collector*, což je automatický správce paměti, jenž má na starost monitorovat objekty, na které už není žádná reference z ostatních udržovaných objektů a tyto pak uvolňovat z paměti. V nových JVM bývá klasicky několik typů těchto správců a každý lze použít optimalizovaně pro specifické úkoly tak, aby příliš nezatěžoval JVM, jelikož monitorování a odstraňování nepotřebných objektů z paměti je poměrně náročná záležitost co se týče strojového času a tudíž i výkonu.

4.2.2 Vlákna

Vlákno v Javě je reprezentováno třídou `Thread` nebo přesněji rozhraním Runnable. Pro použití objektu typu `Thread` a objektů od něj odvozených je nutné implementovat metodu `run()`. Při vytvoření nového objektu, který bude spuštěn jako samostatné vlákno se ale zavolá metoda `start()`, která je implementována ve třídě `Thread`, jelikož je nutné provést několik důležitých operací před spuštěním vlastního vlákna.

Mezi tu nejdůležitější patří například vytvoření prostředí pro běh vlákna. V JVM jsou totiž vlákna implementována pomocí vláken hostitelského systému a to se vytváří právě operací `private native void start0()` volanou z metody `start()`. Nativní metoda `start0()` je pak implementována pomocí jazyka C podle konkrétní implementace příslušné použitému OS.

Pro řešení problému bude tedy nutné takovéto vlákno najít a zastavit tak, aby neprovádělo žádnou činnost, ale aby neblokovalo zbytek systému, zejména pak procesor. Pro uvolnění paměti by bylo nutné toto vlákno uložit a celé zrušit, ale takovéto operace nejsou příliš žádoucí, protože by mohli snížit stabilitu serveru. Navíc by toto ukládání muselo být implementováno v nativním hostitelském systému pomocí JNI, což je daleko nad rámec této práce.

Dalším důležitým aspektem bude zastavit vlákno tak, aby se po dokončení operace mohlo znovu spustit. Toho lze docílit několika způsoby, avšak ne všechny jsou ty správné. Například bude nutné se vyvarovat tzv. *aktivnímu čekání*, což je primitivní způsob jak zastavit provádění běhu nějakého programu. V podstatě se jedná o vytvoření nekonečné smyčky a čekání na změnu proměnné uvnitř smyčky.

Tento způsob není vhodný z toho důvodu že pouze slepě drží proces v jednom místě, čímž ale spotřebovává výpočetní zdroje procesoru. Ačkoliv se může zdát, že to není v dnešní době tolik důležité, je toto řešení nekoncepční a existují mnohem lepší řešení, jak dosáhnout požadovaného efektu.

```
1  boolean run = false;
2  ...
3  while(run) {
4    if (run){
5      break;
6    }
7  }
8  ...
```

Ukázka 4.1: Aktivní čekání

Vzhledem k nevýhodám uvedených v předcházejícím odstavci by se mohlo jevit jako vhodné obohatit čekací smyčku na řádku 5 o uspání vlákna metodou `sleep()`, kterou každé vlákno disponuje. Pak by to vypadalo následovně.

```
1  boolean run = false;
2  final long WAIT_TIME = 5000L;
3  ...
4  while(run) {
5    if (run){
6      break;
7      Thread.sleep(WAIT_TIME);
8    }
9  }
10 ...
```

Ukázka 4.2: Aktivní čekání s uspáváním vlákna

Ani taková úprava ale nepřináší požadovaný efekt a prostředky jazylka Java a potažmo prostředky hostitelského systému nabízejí mnohem elegantnější řešení na úrovni volání signálů vláken a plánování procesů pomocí systémové plánovače.

Pro tyto účely lze použít následující konstrukci.

```
1  public final class MyThread extends Thread {
2
3  private volatile boolean paused = false;
4
5  @Override
6  public void run() {
7    while (canRun) {
8      synchronized (this) {
9        while (paused) {
10       wait();
11       }
12     // normalni cinnost vlakna
13     }
14   }
15 }
16 ...
17
18 public pause() {
```

```
19    this.pause = true;
20  }
21
22  public cont() {
23    synchronized (this) {
24      this.pause = false;
25      notify();
26    }
27  }
```

Ukázka 4.3: Správné pozastavení vlákna v Java kódu

Jedná se o běžný a klasický způsob jak nejjednodušeji implementovat pozastavení vykonávání vlákna. Funguje díky synchronizaci vláken pomocí monitoru. Na řádku 3 je definována řídící proměnná, pomocí níž se rozhoduje, zda bude vlákno běžet, nebo se pozastaví. Použití klíčového slova `volatile` v deklaraci je více než na místě, jelikož zamezuje optimalizátoru takto deklarovanou proměnnou optimalizovat. tak je zabezpečeno že se při volání takovéto proměnné vrátí vždy ta správná hodnota.

Zavoláme-li metodu `pause()` nastaví se tato hodnota na true a při příštím pokusu o průchod cyklem na řádku 10 se celé vlákno zastaví a bude čekat na *vzbuzení* od jiného vlákna. Celé volání cyklu musí nutně být obaleno synchronizovaným blokem, aby nedošlo ke kolizi, pokud by se proměnná nastavovala ze dvou jiných vláken současně.

Volání metody `cont()`[1] se řídící proměnná nastaví na hodnotu `false` a stojící vlákno se *vzbudí* zavoláním metody `notify()`. Toto volání musí být opět v synchronizovaném bloku, tentokráte z toho důvodu, aby bylo možně použít právě metodu `notify()`, protože ta dává signál právě monitoru, jímž se zamyká synchronizovaný blok, ve kterém se tato metoda volá.

Jako monitor pro zamykání synchronizovaných bloků zde slouží vlastní objekt vlákna, aby bylo možné ho odněkud *vzbudit*. Metoda `cont()` by mohla mít místo vloženého synchronizačního bloku ve své deklaraci použito přímo klíčové slovo `synchronize`, které vytváří taktéž synchronizovaný blok a jako monitor použije automaticky objekt, ve kterém je metoda definována, ale explicitní použití synchronizačního bloku je mnohem lépe čitelné, právě z toho důvodu že je zde explicitně definován monitor.

4.3 Současné řešení serveru

Jak už bylo uvedeno v předešlých kapitolách, změně implementace serverové části se nelze vyhnout. Celá nejdůležitější část bude muset být provedena právě změnou serverové části, konkrétně vláken starajících se o průběh zpracování požadovaných úkolů.

Důležitým výsledkem této analýzy je nalezení správného místa, kde se musí provádění úkolu pozastavit, tak aby byly splněny požadavky na řešení uvedené v předešlých kapitolách.

4.3.1 Zastavení provádění exekutora

Jako první možnost jak zadaný problém vyřešit se ihned nabízí pozastavení provádění činnosti na úrovni objektu typu `Executor`. Je to pochopitelné, jelikož se jedná o samostatné

[1] nazvat metodu `continue()` by bylo o dost logičtější, ale v Javě toto nelze jelikož `continue` je klíčové slovo jazyka Java

vlákno jak bylo popsáno v 3.4.2. Při bližším pohledu do kódu je pro to vhodné využít metody `run()`, která se implementuje pro definování činností za běhu vlákna.

V podstatě jde o doplnění funkcionality do objektu typu `Executor`. V tomto typu je provádění implementované v metodě `run()` obaleno `while` smyčkou, která je řízená výsledkem volání metody `shouldRun()`.

```
1 while(shouldRun()) {
2   // provadeni ukolu
3 }
```

Ukázka 4.4: Vykonávací smyčka ve třídě `Executor`

Tudíž by mělo stačit v tomto místě obalit celé tělo označené jako `// provadeni ukolu` na řádku 2 synchronizovaným blokem a řídící proměnnou jako je tomu v ukázce 4.3.

Tento způsob dodání funkcionality do třídy `Executor` je opravdu účinný a funkční. Zavoláním metody `pause()` se opravdu běh zastaví a po zavolání metody `cont()` vlákno pokračuje v činnosti. Problém ale nastává právě během sestavení *Maven* projektu.

Celý problém spočívá v tom, že činnosti popsané v 4.3.2 jsou na úrovni exekutora zapouzdřeny do jediného objektu, kterým je `MavenBuild`. To přináší zásadní omezení a sice takové, že pozastavení vlákna je možné pouze při vstupu do kontrolní smyčky `while`, jak je vidět v ukázce kódu 4.4 na řádku 1. Tudíž pokud by se začal projekt sestavovat, tak by exekutor čekal až se dokončí a potom by se pozastavil. Tím je jasné že toto řešení je nedostatečné vzhledem požadavkům zadavatele na *okamžité* zastavení provádění činnosti.

Z výše uvedených faktů plyne že pozastavení procesu sestavování na úrovni objektu typu `Executor` není vhodné a bude nutně rozebrat podrobněji sestavení projektu a hledat správné místo někde uvnitř funkcí volaných z metody `run()` třídy `Executor`.

4.3.2 Zastavení provádění buildu

Dalším možným místem je pozastavení provádění sestavení projektu pomocí změny sestavy příkazů vytvářené v objektech rozšiřujících abstraktní typ `AbstractBuilder`.

Díky důvodům popsaným níže je dobré toto analyzovat pomocí sestavování jednoho konkrétního typu projektu a sice *Maven* projektu, který je do Jenkins serveru dodáván ve formě zásuvného modulu.

Zásuvný modul Maven

Tento modul je pro pochopení činnosti serveru dosti zásadní, jelikož jeho složitost a komplexnost hravě pokryje většinu zásuvných modulů sloužícím k podobným účelům, tj. sestavování určitého druhu projektu. Taky na tomto modulu lze pochopit jakým způsobem se interpretují příkazy posílané skrze komunikační kanál, respektive jak vznikají a kde se obsluhuje jejich výsledek.

Apache Maven[1] je sestavovací systém pro Java projekty, založený na deklarativním popisu výsledného produktu pomocí XML. Tento popis neříká jak se má výsledný produkt sestavit, ale říká jak má výsledný produkt vypadat a co všechno má obsahovat. V souladu s agilními metodologiemi vývoje software zavedl *Maven* do svého sestavovacího cyklu také testování. To se z výhodou využívá právě v Jenkins serveru, jelikož projekt může být dekla-

rován pohodlně pomocí POM souborů při vývoji a každý vývojář si může výsledný produkt pohodlně sestavit na svém lokálním zařízení.

Nicméně je velmi žádoucí, aby sestavení produktu určené k dodání koncovému zákazníkovi bylo provedeno na konzistentním a co je důležitější referenčním prostředí. To poskytne Jenkins server, protože se požadovaný projekt jednoduše jednou nastaví a pak se vždy může sestavovat právě tady.

Z tohoto důvodu je typ projektu *Maven* asi nejčastějším typem sestavovaného projektu na Jenkins serveru.

Celé sestavení se skládá z několika kroků, v dokumentaci nazývaných životní cyklus. Jsou to tyto kroky:

1. **validate** – validace, zda je projekt správně nakonfigurován
2. **initialize** – inicializace proměnných
3. **generate-sources** – generování potřebných zdrojů pro kompilaci
4. **process-sources** – preprocesing zdrojů pro kompilaci
5. **generate-resources** – generování zdrojů, přikládaných k výslednému produktu
6. **process-resources** – umístění zdrojů na požadované místo
7. **compile** – kompilace zdrojových kódů
8. **process-classes** – postprocesing zkompilovaných zdrojů
9. **generate-test-sources** – generování potřebných zdrojů pro kompilaci testů
10. **process-test-sources** – preprocesing zdrojů pro kompilaci testů
11. **generate-test-resources** – generování externích zdrojů potřebných pro testy
12. **process-test-resources** – umístění zdrojů pro testy na požadované místo
13. **test-compile** – kompilace zdrojových kódů testů
14. **process-test-classes** – postprocesing zkompilovaných zdrojů
15. **test** – spuštění jednotkových testů
16. **prepare-package** – příprava na sestavení výsledného balíku
17. **package** – sestavení výsledného balíku
18. **pre-integration-test** – preprocesing integračních testů
19. **integration-test** – spuštění integračních testů
20. **post-integration-test** – postprocesing integračních testů
21. **verify** – verifikace výsledného balíku
22. **install** – instalace výsledného balíku do lokálního repositáře
23. **deploy** – nasazení výsledného balíku na cílové prostředí

Pokud chceme spustit například pouze kompilaci, definujeme explicitně že máme zájem pouze o tento krok životního cyklu. Maven provede všechny kroky předcházející tomuto definovanému kroku a krok samotný. Na příkladu je ukázáno jak se volá právě krok kompilace.

```
mvn compile
```

Ukázka 4.5: Spuštění sestavování Maven projektu

Jak se dá očekávat, tento krok lze nastavit také v konfiguraci projektu na Jenkins serveru. Zásuvný modul pro Maven projekty postupuje podobně, jako by postupoval vývojář při sestavování na lokálním zařízení. Nejprve zkontroluje, kde je nainstalován samotný Maven (jmenovitě nástroj `mvn`), pak sestaví potřebné parametry pro tento nástroj podle nastavení projektu a zavolá nástroj samotný s připravenými parametry.

Ze závěru analýzy zásuvného modulu vyvstává možnost na řešení problému pomocí vytvoření speciálního příkazu, zařaditelného do seznamu vykonávaných příkazů, který by dal signál koncové JVM, aby přerušila činnost a uvolnila zdroje pro prioritnější úkol.

Podpora pro takovéto chování sice v Jenkins serveru není implementována, ale je zde podpora pro speciální úkoly, které značí poslední příkaz v sekvenci požadovaných příkazů, tudíž by nebyl problém požadovanou podporu doimplementovat.

Celé řešení by mělo vypadat asi takto:

1. Úkol s nižší prioritou běží ne jednom z exekutorů, žádný jiný exekutor není volný.
2. Nově příchozí prioritnější úkol zjišťuje obsazenost a vybere exekutora podle priorit.
3. Nový úkol posílá skrze exekutora příkaz na přerušení vykonávání současného úkolu.
4. Nový proces vytváří nového exekutora a spouští skrze něj nový úkol.
5. Po dokončení nového úkolu tento maže vytvořeného exekutora a posílá příkaz k pokračování starého úkolu.

Tady ale narazíme na více problému, které toto řešení dělají nedostatečným a nepřípustným. V první řadě je to porušení podmínky, aby řešení neznamenalo úpravy stávajících zásuvných modulů. Při tomto řešení by to jinak udělat ale nešlo, jelikož autoři konkrétních zásuvných modulů při jejich vytváření s něčím takovým nepočítali a bylo by obtížné nutit je, aby doimplementovali takovou funkcionalitu, respektive její podporu do již stávajících a funkčních zásuvných modulů.

Dalším velkým problémem by byly úkoly, které pro svůj běh potřebují spouštět procesy operačního systému a jeho nástroje. Tady by nebylo zaručeno, zda se počká na dokončení tohoto úkolu a mohlo by se stávat, že by se výsledky takovéhoto úkolu zahodily. Je to způsobeno tím, že volání operačního systému a procesy na něm spouštěné jsou již ze své povahy asynchronního rázu.

Je tedy jasné že hledání správného místa se musí posunout ještě hlouběji do jádra serveru, přesněji do místa, které tvoří z hlediska zásuvných modulů univerzální *úzké místo* serveru. Jako takové přichází v úvahu místo, kde se všechnu příkazy interpretují do stejné podoby. Tímto místem je bezesporu komunikační kanál.

4.3.3 Zastavení provádění na úrovni komunikačního kanálu

Jak byl napsáno výše, komunikační kanál tvoří místo, kde se všechny příkazy převádějí na stejný typ objektu, jmenovitě se jedná o rozhraní `Callable`. Tím se stává dobrým místem k uvažování o přerušení vykonávání úkolu.

Pro správný návrh řešení je nutné chápat jak se kanál sestavuje, jak funguje a které objekty ho tvoří. Základ je popsán v 3.4.2. Velkou výhodou je, že kanál je vytvářen jak pro komunikaci se vzdálenými uzly, tak i pro komunikaci v rámci jediného serveru. Pokaždé se jedná i jinou implementaci, ale obě dvě mají společné rozhraní `VirtualChannel`.

Kanál jako takový se vytváří po zahájení vykonávání exekutora, těsně před prováděním prvního příkazu příslušného sestavení. To je patrné z obrázku 3.6 a popisu k němu patřícímu. Je vytvořen skrze objekt typu `Launcher`, který využívá právě implementace rozhraní `VirtualChannel` ke smazávání rozdílů mezi oběma typy kanálů.

Kanál je vytvořen za pomocí dvou samostatných vláken, což přináší další možnou úroveň zanoření se v logice do co nejnižší vrstvy komunikace. Jedno z těchto vláken logicky slouží ke čtení objektů z kanálu a druhý pak k zápisu do kanálu.

Jako možně řešení by tedy mohli být obě tyto vlákna. Při bližším pohledu na každé z nich je vidět, že vlákno pro zápis je mnohem složitější než to pro čtení. Z tohoto plyne jasný fakt, že nejvhodnějším místem pro přerušování komunikace bude právě vlákno pro čtení z kanálu. Má to další výhodu v tom, že pokud zastavíme čtení, nemusíme řešit co se děje s asynchronními voláními na úrovni operačního systému. Pokud čtecí vlákno stojí, všechny úkoly mohou v klidu doběhnout a jejich další cesta se zastaví díky pozastavenému čtecímu vláknu. Komunikační kanál se nepřeruší, a ani se nepřeruší vykonávání již započatých úkonů v rámci JVM. Jelikož jsem se dostali až na nejnižší úroveň, co se týká jednotek komunikace, není to žádným problémem, jelikož nedokončené úkoly budou natolik malé, že to řešení nikterak negativně neovlivní.

Jediný problém by mohl nastávat, pokud by některý z buildů prováděl v operačním systému nějakou dlouhotrvající operaci, pak by se prostředky koncového zařízení neuvolnily, ale zůstaly by blokované pro prováděnou operaci.

Po analýze několika nejvíce používaných zásuvných modulů, je ale jasné že nic takového se v těchto zkoumaných modulech neděje. Pokud autoři takovýchto sestavovacích zásuvných modulů dodrží základní myšlenku pro zpracovávání na vzdálených uzlech, nebude s tímto problém ani do budoucna.

Závěrem lze tedy s jistotou konstatovat, že ono hledané místo pro pozastavení provádění úkolu je právě ve čtecím vlákně komunikačního kanálu, které je reprezentováno třídou `ReaderThread`.

4.4 Prezentační vrstva

Pro to aby bylo možné upravený server ovládat i s nově dodanou funkcionalitou budou nezbytné také úpravy prezentační vrstvy, neboli webového uživatelského rozhraní.

To je v Jenkins serveru reprezentováno klasickým MVC přístupem, kde funkci kontroleru plní přímo komponenty serveru. Vrstva pohledů, neboli šablon je vytvořena pomocí nástroje *Apache Jelly*.

Jedná se o klasickou šablonovací techniku, kde se data připraví právě v modelu komponenty a pomocí těchto šablon se reprezentují do výsledného pohledu, kterým je v tomto případě HTML stránka. Tato technologie je založena na XML syntaxi šablon, což dovoluje kontrolu vytvořených šablon na dosti nízké úrovni pro to, aby bylo možné vývojáře

upozornit, kde přesně se vyskytuje chybná nebo neočekávaná hodnota.

Navíc *Apache Jelly* dovoluje strukturované šablonování, což je pro projekt typu Jenkins velmi vhodné. Princip spočívá v tom, že lze vytvořit základní osnovu podoby stránky a do této vkládat další dílčí šablony, jejichž může být například samostatný div element. Ve výsledném HTML se pak tento div jednoduše vloží na místo, kde bylo provedeno vložení dílčí šablony. Takto jsou pak vkládány rozšíření zobrazení pomocích zásuvných modulů nebo rozšíření zobrazení stávající funkcionality, například globální nastavení.

Celý běh požadavku z webového rozhraní má pak na starosti knihovna `Stapler`, které pochází z dílny hlavního vývojáře serveru. Jedná se o klasickou implementaci servletů podle standardu, obohacenou o některé zajímavé funkce jako například automatický výběr výsledného pohledu podle názvu kontroleru a jednoduchou implementaci mapování obsluhy požadavků pomocí názvů metod. Běžně stačí pokud se metoda v objektu anotovaném jako `ExportedBean` jmenuje například `doSomething()`, pak se dá z webové vrstvy zavolat jednoduše pomocí url skládající se z typu objektu a názvu metody bez prefixu `do`.

```
http://localhost:8080/jenkins/executor/build
```

Ukázka 4.6: Volání metody komponenty z webového rozhraní

Tento příklad konkrétně bude volat metodu `doBuild()`, třídy `Executor` bez předaných parametrů. Parametry se mimo jiné předávají pomocí standardních parametrů webového dotazu.

V Jenkins serveru to funguje tak, že pokud chceme vložit nějaké rozšíření, kupříkladu nastavení úkolu, musíme implementovat kontroler ve formě komponenty severu. V tomto konkrétním příkladu je to rozšíření třídy `JobProperty`. V této rozšířené třídě je pak nutné implementovat *deskriptor* objektu, který ja možné zobrazovat právě pomocí *Jelly* šablon.

Šablony jako takové jsou uloženy ve formě XML dokumentů ve stejné složce jako kontrolér. Pro přehlednost se šablony v projektech ukládají do jiné složky, které uvnitř obsahuje stejnou strukturu jako složka s kontrolerem. Po kompilaci se tyto dvě entity zkombinují do jedné složky ve výsledném souboru.

Pokud je potřeba vytvořit šablonu například pro výše zmíněný kontroler nastavení úkolu, pak se vytvoří ve správné složce soubor `config.jelly`, který bude obsahovat zhruba toto.

```
1  <j:jelly
2    xmlns:j="jelly:core"
3    xmlns:f="/lib/form">
4
5    <f:entry title="${%Title}"  field="objectField">
6      <f:textbox value="${instance.objectField}"
7        clazz="required number"
8        default="${descriptor.getSomeValue()}" />
9    </f:entry>
10
11 </j:jelly>
```

Ukázka 4.7: Šablona v Apache Jelly

Po nezbytných deklaracích prostorů jmen (2–3) je uveden tag označující jednu konfigurační položku (5), uvnitř které jsou pak různé prvky pro nastavení a ovládání vlastností

úkolu. Pro sestavení celého výsledného pohledu se proto projdou veškeré definice související s daným úkolem nebo úkoly obecně, vyhledají se veškeré prvky typu `entry` a tyto se vloží do výsledného pohledu seřazené alfabeticky podle titulu.

Položky ohraničené pomocí složených závorek začínající znakem dolaru jsou pak dynamicky naplněny podle hodnot z modelu, který připravil kontrolér, v našem případě komponenta serveru. Speciální označení takovéto položky znakem procenta těsně za otevírací složenou závorkou pak značí, že následující text může být internacionalizován. To znamená, že pokud zobrazovací vrstva aktuálně využívá lokální jazykové nastavení, bude se kontroler snažit nalézt soubor s překladem podle aktuální lokalizace. Nepodaří li se mu tento překlad najít, použije text přímo ze šablony. K výhodám tohoto přístupu patří možnost překládání takovýchto označených úseků přímo uživateli serveru, což je nesporně nejlepší cesta jak udělat překlad srozumitelný. K tomuto účelu Jenkins server obsahuje nástroj pro vytváření překladových souborů, použitelný v konzoli nebo přímo ve výsledném zobrazení. Popis je ale mimo rozsah této práce, proto se mu nebudeme nadále věnovat.

Kapitola 5

Implementované řešení

Jak už zde bylo několikrát uvedeno, původní myšlenku o zachování současného řešení serverové části nelze dodržet, proto se implementace rozdělila na dvě podstatné části.

Nejprve bylo nutné upravit serverovou část tak aby poskytovala podporu pro pozastavení některého probíhajícího úkolu a s tím související úpravy komunikačního kanálu.

Dále bylo nutné vyvinout zásuvný modul, který obohatí nastavení projektu o položku nesoucí informaci o prioritě a vytvořit rozhodovací logiku, které bude řídit spouštěné úkoly podle jejich priority.

5.1 Zajištění vývoje projektu

Pro začátek práce je nutné zajistit několik důležitých faktorů a potřebných zdrojů. Tím je myšleno správně naplánovat jak se bude práce postupně ukládat, kontrolovat a zveřejňovat. Také je logické že por práci je nutné mít upravovatelnou verzi zdrojových kódů, jelikož do současných repositářů smí změny ukládat pouze členové komunity s přidělenými právy zápisu.

5.1.1 Zdrojové kódy

Změna zdrojových kódů původní verze serveru není problém, jelikož se jedná o svobodný software, který je vydáván pod MIT licencí, jejíž znění je součástí přílohy C. pod touto licencí jsou vedeny všechny třídy ve zdrojových kódech Jenkins serveru. Jediný problém který bude nutné řešit pro další jednoduchý rozvoj tohoto zásuvného modulu je dostat tyto úpravy do hlavní větve serveru. To bude jistě vyžadovat, aby tyto úpravy neporušovaly původní architekturu serveru, nevnášely do kódu zmatek a aby obecně nezpůsobovali chyby v běhu serveru. Z této úvahy vyplývá, že bude nutné provádět během úprav testy, jenž jsou součástí aktuální verze serveru a kontrolovat, zda nedošlo k narušení původní funkcionality.

Veškeré zdrojové kódy jsou volně dostupné přes veřejnou službu sdílení zdrojových kódů *GitHub*. Jedná se o tzv. *Social coding*, kde lze udržovat verze svých projektů, ale také přispívat do cizích projektů.

Toto je přesně i případ celého *Jenkins serveru* a projektů s ním spojených. Konkrétně pro implementaci tohoto řešení byly důležité dva projekty.

- Jenkins server – `https://github.com/jenkinsci/jenkins.git`
- Remoting – `https://github.com/jenkinsci/remoting.git`

Jak název serveru napovídá, jedná se o službu podporující správu verzí pomocí *Git* SCM. Pokud se někdo rozhodne přispívat do kódu některé z komunit nebo obecně někoho jiného, pak si může založit vlastní účet, požadované kódy zi zdědit pod svůj účet, kde již bude mít právo zápisu. takto pak lze projekt obměnit nebo doplnit a posléze takovéto změny poslat na schválení původnímu vlastníkovi zdrojů nebo komunitě, které je udržuje. Tito pak mohou rozhodnout o tom, zda se změny do hlavní větve přidají nebo ne.

V případě řešení této práce jsem postupoval přesně jak bylo popsáno v předcházejícím odstavci. Navíc jsem založil nový projekt právě pro vyvíjený zásuvný modul. V těchto repozitářích je tedy uskladněna celá práce na řešeném problému i se všemi verzemi vývoje. Jmenovitě jsou to tyto.

- Upravený Jenkins server – `https://librucha@github.com/librucha/jenkins.git`
- Upravený remoting – `https://librucha@github.com/librucha/remoting.git`
- Zásuvný modul – `https://librucha@github.com/librucha/priority-plugin.git`

Ve výše zmíněných repozitářích zůstane projekt do doby, než se úpravy podaří dostat do hlavní větve. Pak zde zůstane pouze zásuvný modul a odkazy na potřebné projekty.

5.1.2 Sestavení projektů a spuštění serveru

Získané zdrojové kódy bylo nutné také nějakým způsobem zkompilovat do spustitelné podoby. Podrobný návod je v dokumentaci přímo na stánkách komunity. V krátkosti zde uvedu jen nejnutnější příkazy potřebné ke kompilaci a sestavení zdrojových kódů. podrobnější postup pak popisuje manuál k řešení, který je součástí práce a je v příloze B.

Celý projekt jenkins je postaven na sestavovacím nástroji *Apache Maven*. proto je sestavení výsledného serveru, zásuvného modulu nebo jakékoliv jiné části projektu jednoduché a přímočaré. Stačí mít nainstalováno běhové prostředí JVM, právě zmiňovaný *Apache Maven* a mít aktivní přístup k internetu nebo předem staženou část Maven repozitáře.

Pro kompilaci serverové části nebo zásuvného modulu je příkaz uvedené níže.

```
mvn clean install
```

Ukázka 5.1: Sestavení Jenkins serveru ze zdrojů

Pro spustění takto sestaveného serveru pak slouží příkaz

```
mvn hudson-dev:run -DskipTests=true
```

Ukázka 5.2: Spuštění serverové části Jenkins serveru

Pro spuštění takto sestaveného zásuvného modulu slouží naopak příkaz

```
mvn hpi:run -DskipTests=true
```

Ukázka 5.3: Spuštění zásuvného modulu

Poslední uvedený příkaz spustí opět celý Jenkins server ale v jeho seznamu zásuvných modulů již bude nainstalován právě sestavený zásuvný modul.

5.1.3 Informace

Dále bylo nutné vyřešit kde budou shromažďovány informace o vývoji projektu a získané data z analýz.

Původně jsem se zabýval myšlenkou zveřejňovat veškeré informace o průběhu projektu, ale nakonec jsem od toho upustil z důvodu pracné údržby těchto informací, pokud jsou poskytovány veřejně.

Nakonec jsem zvolil svoji soukromou wiki stránku, kam jsem zaznamenával veškeré důležité poznatky a fakta o analýze a nástrojích potřebných pro správný vývoj projektu. Navíc jsou zde zaznamenány veškeré poznámky od vedoucích této práce a označeno zda byly zapracovány či ne.

5.2 Implementace změn serverové části

Základem mého řešení byl postup, který jsem na začátku práce určil. Při sestavovaní tohoto postupu jsem vycházel z výsledků analýzy která vznikla pro semestrální projekt, analýzy serveru (3.4) a v neposlední řadě také analýzy problému (4.3).

Při implementaci bylo vhodné postupovat po menších úsecích, vytvářet funkční celky a tím minimalizovat možnosti zanesení chyb do stávajícího řešení. Z této úvahy vzešel pracovní postup pro změnu současného řešení. Tímto výčtem by měly být postiženy veškeré nutné úpravy serverové části.

1. Přidat do komunikačního kanálu metody pro pozastavení čtecího vlákna.
2. Přenést informaci o aktuálně otevřeném kanálu do exekutora.
3. Upravit webové uživatelské rozhraní, tak aby bylo možné nějakým požadavkem pozastavit jednotlivé exekutory.

Přidat metodu pro pozastavení čtecího vlákna bylo velmi jednoduché, jelikož postup pro pozastavení jsem vyzkoušel na příkladech během analýzy. V tuto chvíli bylo nutné pouze přidat dvě ovládací metody a jednu informační do třídy `ReaderThread`, reprezentující čtecí vlákno komunikačního kanálu. Jmenovitě se jedná o metody `pause()`, `cont()` a `isPaused()`. Poslední jmenovaná je zde proto aby bylo možné zjistit, zda je daný kanál zastaven či ne. Toho se pak dá využít pro optimalizaci prováděných příkazů a pro správné zjišťování stavu který se zobrazuje ve webovém uživatelském rozhraní.

Tím byl splněn bod jedna z plánu implementace a mohl jsem přejít na realizaci bodu číslo dva a sice přenesení informace do vyšších vrstev.

Aby se dalo pozastavování ovládat, by nutné přenést informaci o aktuálním otevřeném kanálu zpět do konkrétního objektu typu `Executor`. Jelikož objekty typu Executor jsou vlastně vlákna, není těžké kdekoliv v tomto běžícím vlákně získat odkaz na jeho instanci a ověřit zda se jedná opravdu o vlákno typu `Executor`. Tímto způsobem k tomu přistupuje právě i statická metoda `currentExecutor()` z právě jmenované třídy. Tak by se mohlo zdát jednoduché po vytvoření komunikačního kanálu, který vzniká právě v konkrétním vlákně, zjistit aktuální odkaz na vlákno exekutora a předat mu informaci o právě vznikajícím komunikačním kanálu. Já jsem se rozhodl umístnit toto přiřazení na jiné místo popsané

v analýze 4.3.3. Bylo to z důvodů vyšší kompatibility a také možnosti ovlivnit toto přiřazení pomocí jiného rozšíření.

Nicméně ve třídě Channel musely vzniknout také nové metody, které delegují provedení požadavku do čtecího vlákna. Pojmenoval jsem se stejně jako metody ve čtecím vlákně samotném a to z důvodu lepší čitelnosti řešení. Navíc jsem se snažil nazývat metody *samopopisně*, aby nebylo nutné vytvářet k nim zbytečné komentáře. Tedy z názvu metody je na první pohled jasné co metoda dělá a jaký je její význam.

Nyní měl objekt typu exekutor informaci o svém komunikačním kanálu a byl tedy schopný ovládat zastavení a znovu spuštění tohoto kanálu. Jelikož jsou exekutoři přístupní přímo z prezentační vrstvy pomocí HTTP operací a metod, bylo jednoduché doplnit tuto vrstvu o ovládání pozastavení konkrétního exekutora.

Jelikož se běžící exekutor v prezentační vrstvě zobrazen jako informace o běžícím úkolu s možností předčasného ukončení úkolu, bylo logické že bude vhodné doplnit tuto reprezentaci právě o tlačítko, které dovolí ručně jednotlivé běžící exekutory pozastavit anebo znovu spustit. Zdálo se vhodné použití pouze jednoho tlačítka, které bude měnit vzhled a nastavenou akci, jelikož každé kliknutí se na cokoliv se přenáší na server ve formě nového HTTP dotazu.

Z tohoto důvodu jsem do třídy `Executor` doplnil navíc metodu `doPause()`, `doCont()`, `supportPausing()` a `isPaused()`, přesně podle signatury servletového řešení, aby je bylo možné volat přímo z webového rozhraní. tyto metody v podstatě jen delegují činnost na stejnojmenné metody bez prefixu `do`.

Teď už jen zbývalo upravit prezentační šablonu, které má na starosti zobrazování exekutorů. Ta se logicky jmenuje `executors.jelly` a obsahuje jednoduchou tabulku pro zobrazení všech nakonfigurovaných exekutorů. to této tabulky stačilo přidat HTML odkaz s nastavenou akcí na pozastavení exekutora, nebo jeho znovuspuštění. Které tlačítko se zobrazuje řídí právě metoda `isPaused()`, jejíž výsledek se přenáší do modelu při zobrazení.

Jelikož jsou šablony *Apache Jelly* v podstatě programovatelné jako většina šablonovacích nástrojů, bylo vhodné doplnit rozhodování o zobrazení přímo do šablony. trochu si to sice odporuje s návrhovým vzorem MVC, kde by se měla dodávat pouze relevantní data a o jejich množině by měl rozhodovat kontrolér, ale v tomto případě jsem se musel přizpůsobit stávajícímu stylu kódu, abych do něj nezanášel zbytečné zmatky. Upravený rozhodovací segment vypadá následně.

```
<j:if test="${e.supportPausing()}">
     
  <j:if test="${e.isPaused()}">
    <a href="${rootURL}/${c.url}${url}/cont">
      <img src="${imagesURL}/16x16/cont.png"
           alt="${%pause this build}"
           height="16" width="16" />
    </a>
  </j:if>
  <j:if test="${!e.isPaused()}">
    <a href="${rootURL}/${c.url}${url}/pause">
      <img src="${imagesURL}/16x16/pause.png"
           alt="${%continue this build}"
           height="16" width="16" />
    </a>
  </j:if>
```

```
17 </j:if>
```

Ukázka 5.4: Část upravené šablony pro exekutory

Nejprve se ověří zda exekutor reprezentovaný objektem s názvem `e` podporuje zastavování provádění (řádek 1). tento krok je nezbytný, jelikož kanále se nesestavuje ihned při zahájení provádění, ale až po prvním požadavku na provedení nějakého úkolu v JVM. Tudíž po spuštění se žádné tlačítko nepřidá, ale objeví se až tehdy, když ze sestaví komunikační kanál. Pak se udělá nezlomitelná mezera v XML reprezentovaná přesně kódem z řádku 2. Následně se zjistí zda není konkrétní exekutor již pozastaven a pokud ano vytvoří se url pro invokaci příkazu pro znovuspuštění společně s příslušnou ikonou. Šablonovací nástroj nedisponuje rozhodovacím blokem *if-else*, proto musí být šablona doplněna navíc o další kontrolu pro případ že daný exekutor zastaven nebyl. tato kontrola se děje na řádku 10. Pokud není exekutor pozastaven, sestaví se jiná url s příslušným obrázkem.

Url pro možnou akci se skládá z několika prvků. První je `rootURL`, který odkazuje na kontext celého servletu. Dalším je `c.url`, což je konkrétní objekt typu `Computer`, kterému tento exekutor patří. Následuje `url`, což je odkaz na konkrétního exekutora v poli exekutorů daného objektu typu `Computer`. Výsledný odkaz může pak vypadat například takto.

```
http://localhost:8080/computer/Slave_01/executors/0/pause
```

Ukázka 5.5: Generovaný odkaz na novou akci

Celý výsledek úprav serverové části je pro uživatele patrný právě možností pozastavení jím vybraného exekutora v místech kde se exekutoři zobrazují. V zobrazení je to sice pouze nová *klikatelná* ikona, ale skrývá se za ní užitečná funkcionalita, použitelná dále novým zásuvným modulem pro řízení priorit úkolů. Výsledek úprav je na obrázku 5.1.

Obrázek 5.1: Upravené zobrazení exekutorů

5.2.1 Testování

Testování této fáze úprav bylo velmi jednoduché. Jednak stačilo spustit stávající jednotkové testy, které jsou součástí zdrojových kódů a pak dopsat dva jednoduché testy na ověření pozastavení a znovu spuštění jednoho exekutora.

Dalším důležitým testem byl uživatelský test, který jsem provedl sám, dle scénářů definovaných v následující tabulce.

Scénář	Stav exekutorů	Akce	Očekávané chování
001	Všichni exekutoři jsou nečiní	Pause	Nelze provést
002	Všichni exekutoři jsou nečiní	Continue	Nelze provést
003	Exekutor pracuje, zbytek je nečinný	Pause	Pozastaví exekutora
004	Exekutor pracuje, zbytek je nečinný	Continue	Nelze provést
005	Exekutor je pozastaven, zbytek je nečinný	Pause	Nelze provést
006	Exekutor je pozastaven, zbytek je nečinný	Continue	Znovu spustí provádění

Tabulka 5.1: Testovací scénáře pro úpravy serverové části

Tímto krokem byly dokončeny úpravy serverové části do takového stavu, aby bylo možné vytvořit zásuvný modul, který bude využívat této nové funkcionality pro řízení procesů podle zadaných priorit.

5.3 Implementace zásuvného modulu

Jelikož jsou zásuvné moduly do Jenkins serveru poměrně specializovanou záležitostí bylo vhodné začít jednoduchým zásuvným modulem, který je tzv. *Hello World* aplikací. Takový zásuvný modul je popsán v tutorialu přímo na stránkách s dokumentací k rozšiřování serveru. Abych dobře porozuměl architektuře zásuvných modulů vytvořil jsem nejprve testovací modul, kde jsem všechnu funkcionalitu, u které jsem usoudil že se bude pro implementaci výsledného modulu hodit, vyzkoušel a odladil jsem si zde rutinní činnosti jako například propojení s prezentační vrstvou.

Pro implementaci zásuvného modulu jsem si stanovil podobný seznam nutných kroků stejně jako u úprav serverové části.

1. Vytvoření nastavení priority do nastavení projektu.
2. Doplnění sloupce s přehledem priorit do hlavního pohledu.
3. Vytvoření nastavení globálních konfiguračních proměnných pro zásuvný modul.
4. Implementace rozhodovací logiky podle priorit.
5. Vytvoření nového typu exekutora pro úkoly s vyšší prioritou.

5.3.1 Konfigurace úkolu

Jako první byl logicky na řadě krok vytvoření nové konfigurační položky na úrovni jednoho projektu. Po krátkém bádání v dokumentaci jsem zjistil že taková rozšíření se dělají pomocí implementace potomka třídy `JobProperty`. jedná se o datovou strukturu, podobnou mapě, kde se různé hodnoty mohou ukrývat pod textovým klíčem. Tímto klíčem je jméno atributu nově vytvořené třídy, které je pak přístupné v objektech, jenž jsou potomky třídy `AbstractProject`.

Protože je nutné konfiguraci přidat i do prezentační vrstvy, musí se rovněž implementovat potomek vnitřní třídy `JobProperty` a sice `JobPropertyDescriptor`. Tento popisný objekt dovoluje přenášet data do prezentační vrstvy, tudíž se v MVC modelu jedná o kontrolér. Jde o potomka třídy `Descriptor`, která právě plní tuto roli i v dalších typech objektů, jenž

musí přenášet data (taktéž model) do šablony prezentační vrstvy. Všechny takové implementace musí být zaregistrovány jako rozšíření, což se provede pomocí anotace `@Extension`.

Tímto je vyřešen prostředek pro ukládání konfigurace projektu. Jelikož konfigurace jsou perzistentní, je navíc nutné umět takto uloženou konfiguraci správně zpětně načíst do daného objektu. O to se stará speciální konstruktor, označený anotací `@DataBoundConstructor`. Při zavolání takového konstruktoru se projde uložené nastavení projektu a položka se naváže na parametry konstruktoru. Tak lze s hodnotou z perzistentního úložiště dále pracovat nebo ji změnit. Změna se provede pro programátora transparentně po uložení nastavení ve webovém rozhraní.

Třída, kterou jsem do zásuvného modulu přidal se konkrétně jmenuje `PriorityConfigJobProperty` a splňuje výše popsané předpoklady pro nastavování konfigurace všech druhů projektů. Aby to bylo zcela funkční musel jsem vytvořit navíc ještě šablonu, která nově přidané pole zobrazí v konfiguraci. Ta se podle konvencí Jenkins serveru jmenuje `config.jelly` a leží ve stejném balíčku jako třída. Obsah šablony není složitý.

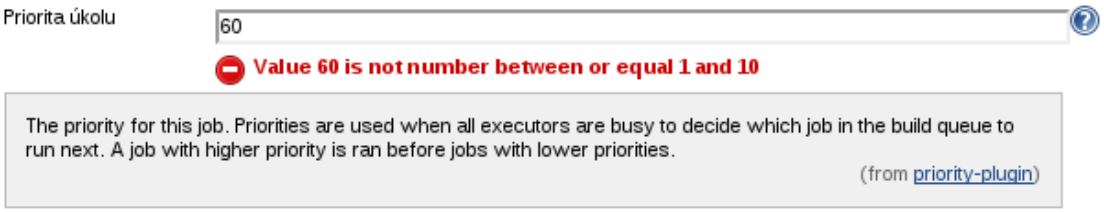
```
1  <j:jelly
2  xmlns:j="jelly:core"
3  xmlns:f="/lib/form">
4
5    %<f:entry title="${%Job Priority}"  field="jobPriority">
6      <f:textbox value="${instance.jobPriority}"
7        clazz="required number"
8        default="${descriptor.getDefaultPriority()}" />
9    </f:entry>
10
11 </j:jelly>
```

Ukázka 5.6: Část upravené šablony pro exekutory

jako jinde jsou nejprve provedeny deklarace jmenných prostorů (1-2), dále je přidána položka mapy nastavení (5), kde atribut `jobPriority`, je právě klíčem v mapě nastavení. Uvnitř této položky (6-8) je pak definován vlastní HTML prvek, zobrazující nastavení. Atribut tohoto prvku `class` je velmi důležitý, jelikož na základě něho se vytváří automaticky validační pravidlo pro položky vkládané do tohoto prvku. Zde to konkrétně musí být číslo. Bylo ale nutné přidat ještě další validaci a to na rozsah vloženého čísla. Toho jsem dosáhl pomocí implementace metody `doCheckJobPriority()` ve výše popsaném deskriptoru nastavení. V této metodě se jednoduše zkontroluje zda zadané číslo není mimo povolený rozsah a pokud je připraví se o tom hlášení, které se následně zobrazí uživateli, který daný projekt nastavuje. Ilustrace na obrázku 5.2.

Z obrázku 5.2 je navíc vidět i zobrazení nápovědy k danému konfiguračnímu poli. To se vytvoří jako jednoduchý HTML soubor obsahující pouze prvek typu `div`. Při správném pojmenování souboru, v tomto případě `help-jobPriority.html`, se takováto nápověda automaticky namapuje na konfigurační položku daného jména a u ní se pak zobrazí otazník, pod kterým lze po kliknutí danou nápovědu nalézt.

Po vytvoření této konfigurační položky bylo vhodné doplnit možnost zobrazení nastavených priorit také do přehledu nastavených projektů.

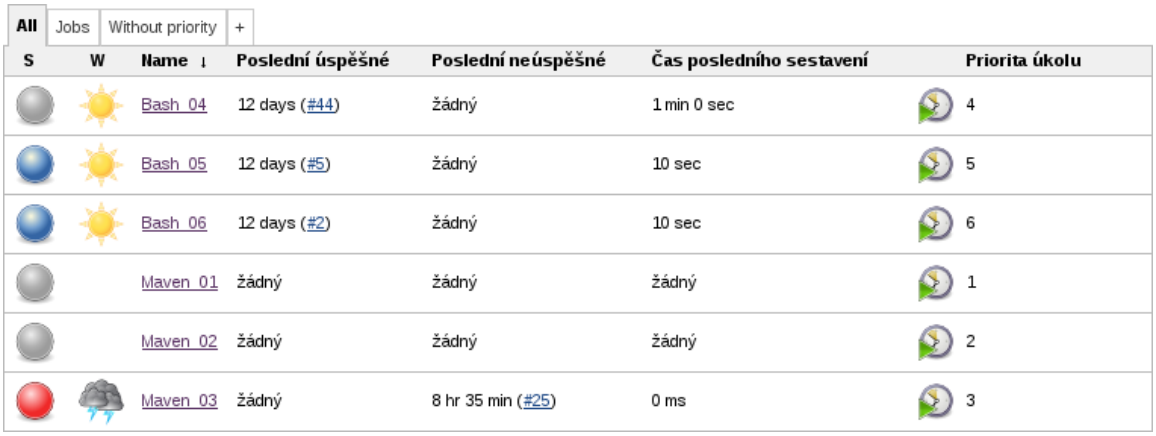


Obrázek 5.2: Chybná konfigurace priority

5.3.2 Obohacení hlavního pohledu

Hlavní pohled je seznam nastavených projektů, který se zobrazuje jako hlavní stránka *Jenkins serveru*. Tato tabulka má nastavitelné sloupce, co v ní bude zobrazeno. Proto bylo vhodné přidat takový sloupec, který by zpřehledňoval nastavení priorit u jednotlivých projektů.

Sloupce této tabulky jsou potomky třídy `ListViewColumn` a musejí mít deskriptor registrovaný jako rozšíření, stejně jak tomu bylo u konfigurační položky projektu. V tomto deskriptoru je pak nutné překrýt metodu `getDisplayName()`, která určuje jak se bude jmenovat sloupec a zároveň co bude v jeho hlavičce a také metodu `shownByDefault()`, která zase určuje zde se bude sloupec zobrazovat v základním nastavení. Sloupce lze totiž, jak bylo řečeno výše, libovolně zapínat nebo vypínat v různých pohledech. Obrázek 5.3 ukazuje jak potom takový sloupec vypadá.



Obrázek 5.3: Sloupec s přehledem nastavených priorit

Samozřejmě i tento sloupec musí mít svoji šablonu, ve které se říká, jak se bude hodnota ve sloupci interpretovat a co znamená.

5.3.3 Globální konfigurace

Během implementace zásuvného modulu se ukázalo vhodné mít možnost celou prioritizaci zapínat a vypínat podle přání uživatele.

Jelikož je globální konfigurace v *Jenkins serveru* šablonování jazykem groovy, který příliš neovládám, rozhodl jsem se ponechat globální konfigurace v šablonách *Apache Jelly*, jako

ostatní. Pro tento účel vzniklo registrované rozšíření `PriorityJobGlobalConfiguration`, které je zděděno z třídy starající se právě o globální nastavení `GlobalConfiguration`.

její implementace je vskutku jednoduchá. Jedná se o třídu s jedním atributem a metodami pro nastavení a získání tohoto atributu. Navíc je zde překryta metoda `configure()`, umožňující získání konfigurace z perzistentního úložiště, podobně jako tomu bylo u konfigurace projektu. Rozdíl je patrný při pohledu na parametry této metody, jelikož se globální konfigurace přenášejí pomocí datového protokolu JSON. Z něj je potřeba vybrat pouze položku požadovaného jména a převést ji na správný datový typ.

Po implementaci jednoduché šablony ve správném balíčku a se správným jménem se pak objeví v globální konfiguraci položka se zaškrtávacím políčkem , dovolující vypnout celou logiku prioritizace. Více v manuálu v příloze B.

5.3.4 Rozhodovací a řadící logika

Vymyslet logiku řazení prioritních úkolů a rozhodování o tom zda nově příchozí požadavek bude pozastavovat některý z běžících procesů bylo asi jádro implementace zásuvného modulu. Po pročtení dokumentace a správném pochopení procesu zahájení sestavování jsem usoudil že nejsprávnější místo, kam *vetknout* tuto logiku bude do místa kde se volají všechny objekty registrované jako rozšíření, jež jsou potomky třídy `QueueDecisionHandler`. tato třída se má starat o to zda daný požadavek bude zařazen do fronty, nebo nebude.

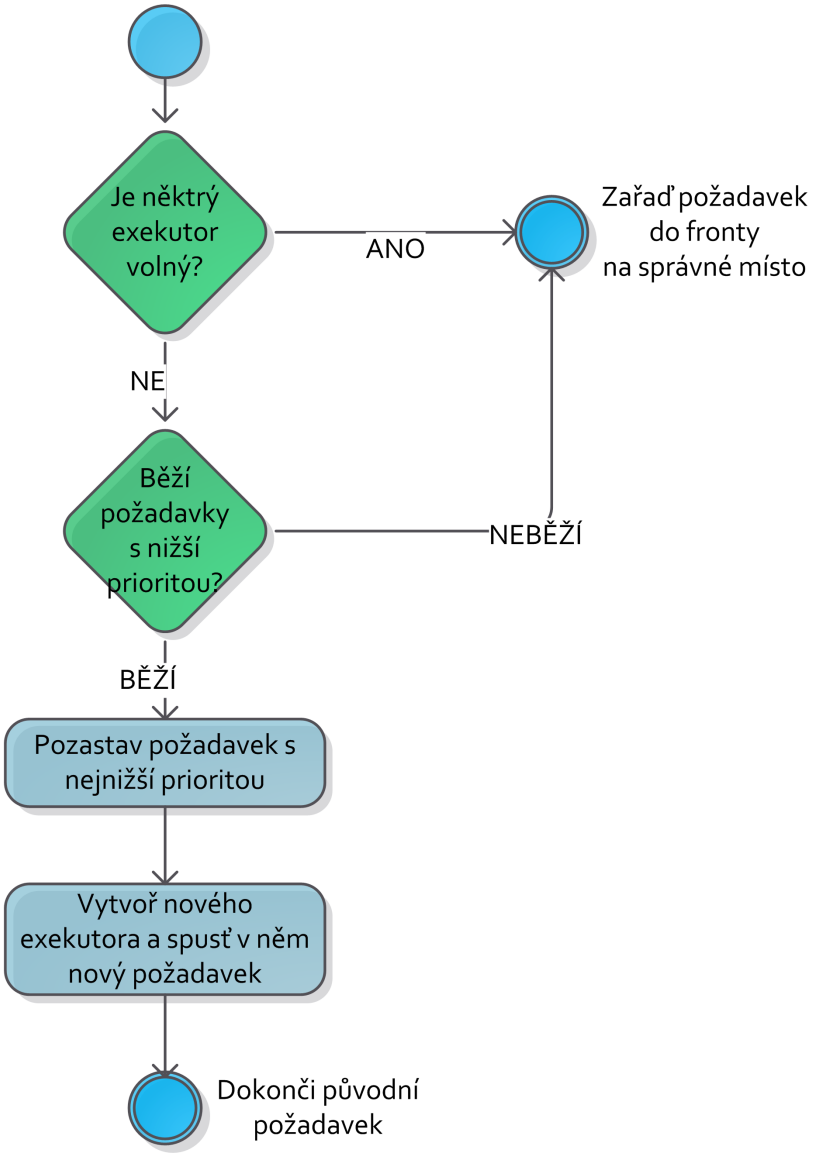
Z procesního hlediska se může na první pohled zdát že toto místo není to kde by se mělo o prioritizaci rozhodovat, protože úkoly do fronty zařazovat nechceme. Při bližším pohledu ale zjistíme, že zde máme všechny potřebné prostředky pro rozhodování.

Hlavní ložisko této nové třídy je v metodě `shouldSchedule()`, která je volána pro každý úkol, který je nutné sestavit. Díky tomu že je zde možné přistupovat k frontě, je možné získat seznam všech počítačů, aktivních v danou chvíli, seznam všech exekutorů, kteří pracují nebo čekají a to pro tento zásuvný modul a rozhodnutí o tom, zda nový úkol poběží prioritněji, stačí.

Celou rozhodovací logiku shrnuje vývojový diagram na obrázku 5.4. Jedná se o jednoduchou logiku. Při příchodu nového požadavku se nejprve zjistí, zda bude nutné vůbec prioritně spustit tento nový požadavek a zda je prioritizace povolena. Pokud nejsou splněny tyto podmínky, požadavek se naplánuje do fronty požadavků jako by se jednalo o běžný požadavek. Zjistí-li se naopak, že pracují všechny dostupné exekutory, bude se rozhodovat dále na základě priorit běžících požadavků. Běží li na některém z exekutorů požadavek s nižší prioritou než má nově příchozí, bude nutné vyhledat běžící požadavek s nejnižší prioritou a tento pozastavit. Pokud všechny běžící požadavky mají vyšší prioritu než požadavek příchozí, pak se tento zařadí do fronty tak, aby před ním nebyl žádný jiný frontovaný požadavek s nižší prioritou. to se docílí tím že se musí projít všechny čekající požadavky a přeplánovat je podle času právě řazeného požadavku s dodržením nastavené prodlevy mezi vykonáním jednotlivých požadavků.

5.3.5 Nový typ exekutora

Dojde-li na nutnost pozastavení některého z běžících požadavků a spuštění jiného, využije se k tomu nová funkcionalita doplněná do serverové části. Ta dovoluje pomocí konkrétního exekutora zastavit provádění požadavku a toto zobrazit ve webovém rozhraní. Nový požadavek se spustí na nově vytvořeném typu exekutora, speciálně pro tento zásuvný modul. Jedná se o jednoduchou implementaci, odvozenou od původní třídy `Executor`, jmenovitě



Obrázek 5.4: Rozhodovací diagram

je to `PriorityJobExecutor`. Při vytváření instance této třídy je požadován úkol, který se má vykonat a odkaz na původně zastaveného exekutora.

Překrytím původní metody `run()`, která si úkol bere z fronty požadavků jsem docílil přímého určení, jaký úkol bude ta tomto novém exekutorovi běžet. Pomocí bloku kódu `try-catch-finally` jsem zase dosáhl toho že se původně pozastavený exekutor vždy znovu spustí po jakémkoliv pádu tohoto nového exekutora nebo také po jeho správném dokončení.

Pro zvýšení přehlednosti o probíhajících úkolech bylo ještě nutné zobrazit nového exekutora ve webovém rozhraní. Toho jsem docílil jednoduchým způsobem a sice zaregistrováním tohoto exekutora do objektu typu `ResourceController`, který se stará mimo jiné o zobrazení probíhajících úkolů.

Tímto posledním objektem byla implementace zásuvného modulu hotova a stačilo ji jen pořádně a systematicky otestovat.

Scénář	Fronta	Exekutoři	Očekávaný výsledek
001	Prázdná	Nic neběží	Úkol se naplánuje do fronty a spustí na dostupném exekutoru
002	Prázdná	Úkoly s vyšší nebo stejnou prioritou	Nový úkol se přidá do fronty na správné místo, fronta se přeplánuje, bude-li to nutné
003	Úkoly s nižší nebo stejnou prioritou	Úkoly s vyšší nebo stejnou prioritou	Nový úkol se přidá do fronty před nejbližší úkol s nižší prioritou
004	Úkoly s vyšší prioritou	Úkoly s vyšší nebo stejnou prioritou	Nový úkol se přidá do fronty za nejbližší úkol s vyšší nebo rovnou prioritou

Tabulka 5.2: Testovací scénáře pro zásuvný modul

5.3.6 Testování

K tomuto účelu vznikl na začátku implementace seznam testů, které budou muset být splněny, aby se dala implementace považovat za funkční a úspěšnou. Přehled těchto testů je v následující tabulce.

Jelikož všechny tyto testy byly úspěšně provedeny, dala se implementace považovat za úplnou a funkční podle požadavků zadání.

Kapitola 6

Závěr

Tato práce stručně a jednoduše shrnula problematiku průběžné integrace, implementovanou Jenkins serverem a vysvětlila potřebu prioritizovat úkoly, prováděné na fyzických uzlech tohoto řešení.

Blíže se podívala také na správu vláken v JVM, která bylo pro řešení klíčovou. Navíc tato práce obsahuje dosti podrobnou analýzu průběhu požadavku skrze server od vzniku požadavku až po vykonání jeho dílčích činností v JVM.

Pro velké firmy, kde pracuje mnoho vývojářů se stal proces průběžné integrace nutností a většina z nich si nedokáže práci bez něj efektivně představit. Díky tomu že Jenkins server patří k otevřeným řešením, neboli svobodnému software, je velmi oblíben a používán ve velkém měřítku. Velké množství firem ale časem narazí na nevůli pořizovat další fyzická zařízení jenom kvůli tomu aby mohli vytvářet nové servery pro průběžnou integraci.

S tímto problémem jim pomohou změny provedené v serveru a nový zásuvný modul, jehož vývoj a problematika procesů, které umí řídit, je popsána právě v této práci, jmenovitě v kapitolách 3 až 5

Závěrem je nutné dodat, že tato práce se opírá o dokumentaci, o kterou se stará komunita vývojářů a je popsána na mnoha webových stránkách, jak je tomu u moderního software zvykem. Při prvním pohledu na kapitoly o implementaci se může zdát, že tato byla příliš jednoduchá. S tím se dá lehce souhlasit, jelikož většina úsilí stojící za touto prací je právě v podrobné analýze chování a funkcionality, bez které by toto jednoduché řešení nemohlo vzniknout nebo by vniklo, ale bylo by příliš neohrabané, naivní a neefektivní.

Navíc se dá s jistotou tvrdit že obsah práce inovuje něčí myšlenku, čímž se podílí na vzniku a odhalení nových informací nejen na poli informačních technologií. také obrací proud myšlenek dál a nabízí nové možnosti pro vylepšování ať už autorovi nebo dalším lidem, kteří jsou ochotni se podílet na rozvoji tohoto řešení a serveru obecně.

6.1 Možná rozšíření

Jelikož jsem poznal že je komunitní software důležitý pro rychlý rozvoj myšlenky, zdokonalování práce jiných a týmovou spolupráci, rozhodl jsem se pokračovat ve vylepšování mého řešení dokud mi to čas dovolí.

Na závěr se neodbytně vnucuje myšlenka jak toto dílo dále rozšiřovat. Možností je samozřejmě nepřeberné množství, ale dovolím si nastínit další zamýšlený vývoj zásuvného modulu a serveru v mém podání.

Určitě bude důležité vylepšit informovanost uživatele o tom že se nějaký proces poza-

stavil a jiný ho předbíhá. Toto bude dobré sjednotit se systémem notifikací, kterým server v současné době disponuje.

Dalším velkým celkem a zároveň dosti velkým oříškem bude možnost pozastavené procesy uložit z paměti do perzistentního úložiště, tak aby se uvolnily nejen procesory, ale i paměť. toto řešení nebude nikterak jednoduchým, protože se bude muset vyřešit nativní přístup do paměťi hostitelského operačního systému a co je nejdůležitější znovu rozběhnutí pozastavených procesů s nahráváním předchozích stavů do paměti. Představu jak na to již mám v hlavě a na řešení začnu pracovat jakmile to bude možné. Měla by vzniknout celá nová nezávislá knihovna umožňující ukládání *živých* objektů z paměti na libovolné perzistentní úložiště a jejich zpětné sestavení pro pokračování tam kde byly přerušeny.

Toto by mohlo přinést do světa Javy úplně nové možnosti jak se vypořádávat s pády serverů pomocí pravidelných obrazů aplikace na úrovni JVM. Bude ale zapotřebí ještě mnoho práce a času, než takové řešení spatří světlo světa.

Na úplný závěr bych rád podotknul, jak moc pro mě tato práce znamenala, jelikož mi otevřela cestu do komunity, která se stará o důležitý produkt a moc rád se budu také na tomto produktu podílet stejně jako odborný vedoucí této práce Ing. Juránek Ph.D.

Literatura

[1] Apache Software Foundation: Apache Maven. http://maven.apache.org.

[2] Apache Software Foundation: Jelly: Executable XML. http://commons.apache.org/jelly.

[3] Jenkins CI community: Jenkins Wiki. https://wiki.jenkins-ci.org.

[4] Jenkins CI comunity: Governance Document. https://wiki.jenkins-ci.org/display/JENKINS/Governance+Document#GovernanceDocument-Compatibilitymatters.

[5] Jenkins CI comunity: Jenkins Continuous Integration Server – Source codes. https://github.com/jenkinsci/jenkins.

[6] Martin Fowler: Continuous Integration. *http://martinfowler.com/articles/continuousIntegration.html*, květen 2006.

[7] Sonatype, Inc.: Sonatype Software Development Infrastructure Survey. http://go.sonatype.com/content/winter2011surveyresults.

[8] Tim Lindholm, Frank Yellin: The Java™Virtual Machine Specification. *http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html*, 1999.

[9] Wikipedia: Thread pool pattern. http://en.wikipedia.org/wiki/Thread_pool_pattern.

Příloha A

Obsah CD

Na přiloženém CD lze nalézt celé řešení problematiky popsané touto prací. Především se jedná o projekty odvozené od původní větve serveru, jmenovitě `jenkins` a `remoting` a pak nový projekt obsahující zásuvný modul, který vzniknul z výsledků analýzy serveru a implementační analýzy.

Struktura souborového systému:

- jenkins – složka obsahující upravenou verzi serverové části aplikace
  - cli – komponenty starající se o možnosti řízení serveru z příkazové řádky
  - core – jádro serveru obsahující důležité komponenty pro běh celého řešení
  - maven-plugin – nejdůležitější zásuvný modul pro sestavování Maven projektů
  - test – složka se testy serveru
  - ui-samples-plugin – ukázky jak lze používat různé prvky webového rozhraní a jak s nimi pracovat
  - war – složka pro sestavený výsledný balík serveru
  - work – složka s nakonfigurovanými úkoly, které byly používány během vývoje
- remoting – složka obsahující komponenty pro sestavování vzdálených spojení na jiné fyzické servery
- priority-plugin – složka s novým zásuvným modulem
  - .git – lokální repozitář obsahující všechny změny kódu, které vznikaly během vývoje
  - pom.xml – definiční soubor projektu pro sestavení pomocí nástroje Apache Maven (sestavováno verzí 3.0.4)
  - README – krátké info o projektu a autorovi
  - src – zdrojové kódy zásuvného modulu

Výše popsaný výčet je dostupný také na webové adrese s tímto projektem.

https://github.com/librucha

Jmenované zdroje budou veřejně přístupné, dokud bude služba GitHub veřejná a funkční.

Příloha B

Manuál

Součástí přílohy je i jednoduchý manuál na kompilaci, spuštění a nastavení Jenkins serveru s rozšířeními dovolující prioritizaci procesů.

B.1 Zdrojové kódy

Zdrojové kódy jsou dostupné na přiloženém CD, nebo na stránkách GitHub služby `https://github.com/librucha`. Po stažení je vhodné všechny tři projekty umístnit do stejné složky, například `jenkins-ci`. V dalším postupu se budeme vždy na tento adresář odvolávat jako na hlavní adresář, odkud budou psány všechny cesty a pokud nebude explicitně řečeno jinak, tak i příkazy.

B.2 Kompilace

Pro kompilaci je nutné mít nainstalováno:

- JDK od firmy SunMicrosystems Inc. ve verzi 1.6 (někdy také označováno jako verze 6)
- Apache Maven alespoň ve verzi 2 (Kompilováno a testováno bylo na verzi 3.0.4)

Vlastní kompilace musí proběhnout v tomto pořadí:

1. priority-plugin
2. remoting
3. jenkins

Vlastní příkazy musí být spouštěny v místě kde se nachází rodičovská definice projektu, tzv *parent pom* nebo je možné použít přepínač `-f` příkazu `mvn`, který nasměruje provádění k zadanému POM souboru.

Pro unixové systémy by mohl společný příkaz pro kompilaci spouštěná ze společného adresáře vypadat takto:

```
mvn clean install -f priority-plugin/pom.xml &&
mvn clean install -f remoting/pom.xml &&
mvn clean install -f jenkins/pom.xml
```

Ukázka B.1: Sdružený příkaz pro kompilaci projektů

Pokud by se vyskytnuly chyby v testech, je možné ještě použít přepínač `-DskipTests=true` pro každý jednotlivý `mvn` příkaz.

Po úspěšném provedení kompilace vznikne v adresáři `jenkins/war/target` soubor `jenkins.war`, do kterého je zabalena celá funkční aplikace upraveného serveru.

B.3 Spuštění

Takto vytvořený soubor `jenkins.war` je pak možně nahrát do libovolného servletového kontejneru, jakým je například `Apache Tomcat`. Pokud není přístupný takovýto servletový kontejner, je možné celý server spustit z příkazové řádky pomocí příkazu.

```
java -jar jenkins/war/target/jenkins.war
```

Ukázka B.2: Sdružený příkaz pro kompilaci projektů

B.4 Instalace zásuvného modulu

Po spuštění je možno nainstalovat nový zásuvný modul přes standardizované rozhraní pro instalaci těchto modulů. Stačí vybrat správný soubor obsahující veškerou zkompilovanou implementaci modulu. ten se nachází ve složce `priority-plugin/target` a má příponu `*.hpi`.

Tyto moduly se instalují v administraci serveru pod odkazem `Spravovat pluginy`. tam je nutné vybrat záložku `Pokročilé` a v části nadepsané jako `Upload Plugin` zadat cestu k výše zmíněnému souboru se zásuvným modulem. Ukázka je na obrázku B.1.

B.5 Ovládání serveru

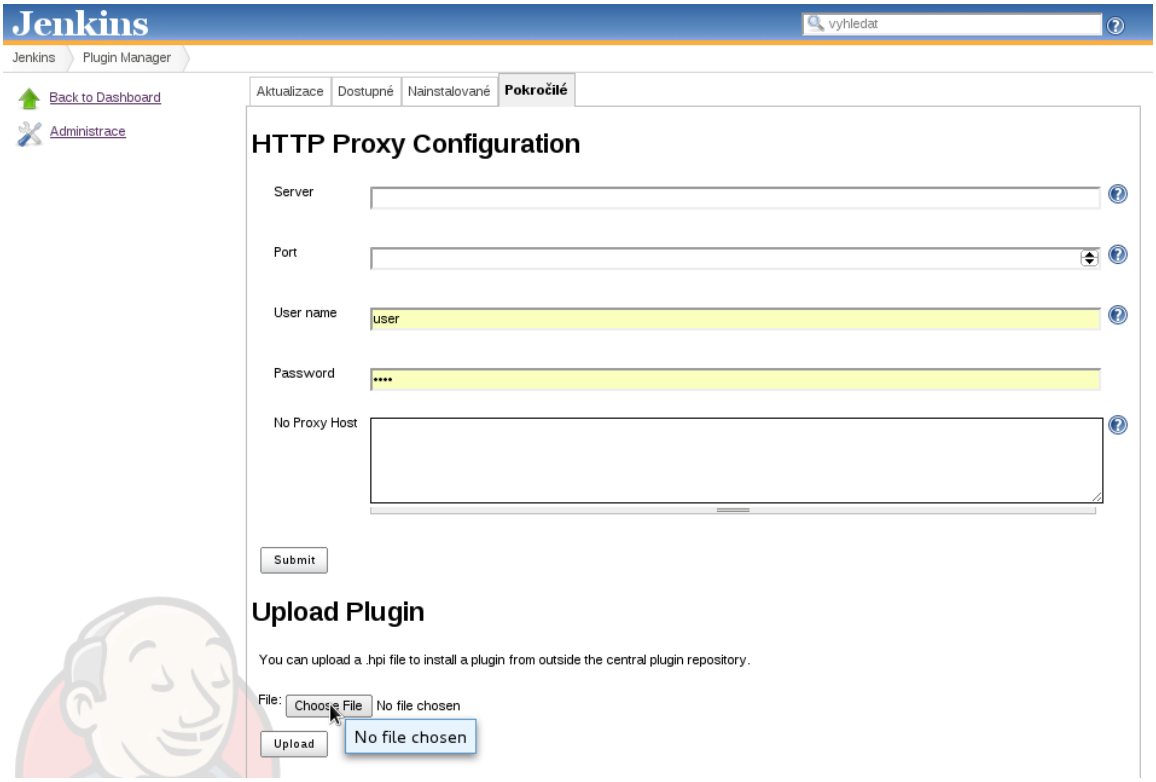
Změny a úpravy serveru nemají nijak vliv na standardní ovládání Jenkins serveru. Jinak řečeno ovládání se neliší, je pouze rozšířeno o určité položky, které budou popsány dále.

B.5.1 Globální konfigurace

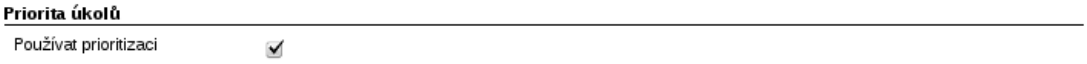
V globální konfiguraci lze ovládat, zda se bude vůbec brát zřetel na prioritizaci procesů. To lze přepínat pomocí přepínače v sekci `Administrace` – `Nastavit systém`. Přepínač se jmenuje `Priorita úkolů`. Pro větší názornost je toto zobrazeno na obrázku B.2.

B.5.2 Konfigurace úkolů

Aby bylo možně prioritu používat, je nutné nastavit každý úkol na požadovanou prioritu. Toho se dosáhne vyplněním prioryty v konfigurační sekci každého úkolu. Pokud nebude



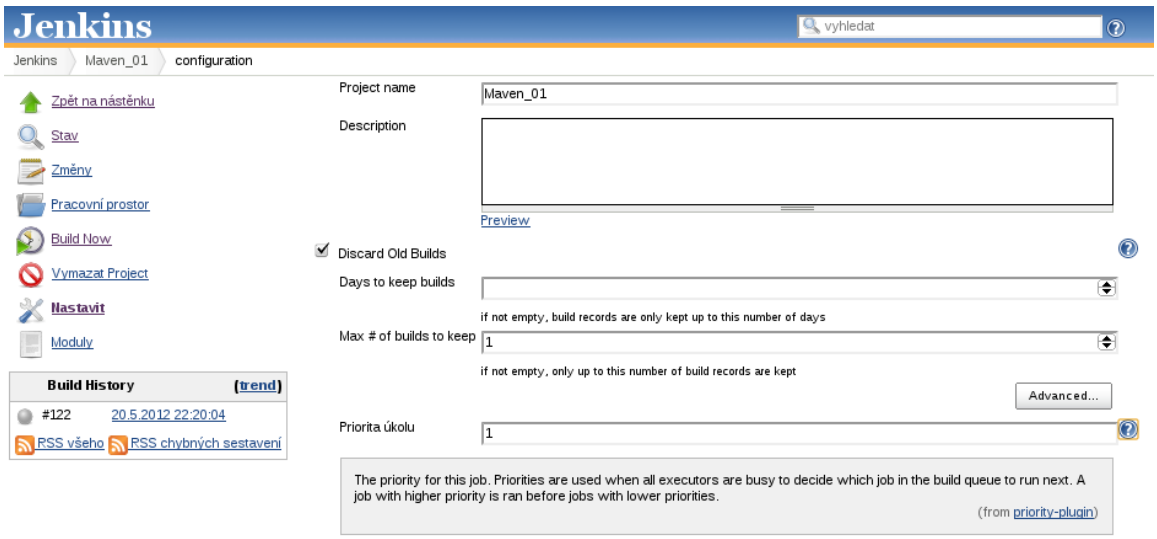
Obrázek B.1: Instalace zásuvného modulu



Obrázek B.2: Globální nastavení priorit

zadána žádná priorita (například u úkolů, které vznikly před instalací tohoto zásuvného modulu), bude takovém úkolu vždy přidělována výchozí priorita, čili nejnižší. Ilustrace (obr. B.3) nastavení ukazuje jak vypadá nová konfigurace.

Po nastavení všech projektů je možné začít používat novou funkcionalitu.



Obrázek B.3: Nastavení priority úkolu

Příloha C

Licence MIT

Licence pod kterou je celý Jenkins server vydáván. Je zde pouze anglická verze, jelikož překlad by musel být právně kvalifikovaný, aby mělo význam připojovat.

The MIT License

Copyright (c) 2004-2009, Sun Microsystems, Inc., Kohsuke Kawaguchi, Brian Westrich, Red Hat, Inc., Stephen Connolly, Tom Huybrechts

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.