PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

# DIPLOMOVÁ PRÁCE

Repozitář výsledků výkonnostních testů

2013

Bc. Adam Dohnal

## Anotace

*Testování je důležitou a často i nezbytnou součástí vývoje softwaru, od tzv. unit testů, které se vyskytují i v těch nejmenších projektech, až po ty, co se zabývají správností po výkonnostní stránce. V této diplomové práci navrhuji repozitář, který nad výsledky testů vytváří jednotný pohled a usnadňuje uživatelům jejich analýzu. Důraz především kladu na jeho nezávislost a snadnou rozšiřitelnost pomocí rozdělení na moduly, které definují jednoduché API. Součástí práce jsou výkonnostní testy, které dokazují, že je možné repozitář provozovat i v náročnějších podmínkách. Aplikace je napsána pomocí technologií dostupných na platformě Java Enteprise Edition a sestavována pomocí nástroje Apache Maven.*

Děkuji vedoucímu této práce Janu Konečnému za rady, které mi při vypracování této práce poskytl. Také bych chtěl poděkovat firmě RedHat a panu Liboru Zoubkovi za spolupráci a užitečné konzultace.

# Obsah

**1. Úvod** **8**

**2. Základy testování** **9**
2.1. Úrovně testování . . . . . 9
2.1.1. Unit testování . . . . . 9
2.1.2. Integrační testy . . . . . 10
2.1.3. Výkonnostní testy . . . . . 12
2.2. Shrnutí . . . . . 13

**3. Analýza dostupných nástrojů** **14**
3.1. Continuous integration servery . . . . . 14
3.2. Repozitáře . . . . . 14
3.3. Monitorovací nástroje . . . . . 14
3.4. Shrnutí . . . . . 15

**4. Požadavky a vlastnosti repozitáře** **16**
4.1. Datové schéma . . . . . 16
4.2. Služba . . . . . 17
4.3. Klient . . . . . 18

**5. Návrh repozitáře** **19**
5.1. Apache Maven . . . . . 19
5.2. Struktura projektu . . . . . 20
5.2.1. Rozhraní . . . . . 21
5.2.2. Klient . . . . . 21
5.2.3. Server . . . . . 22

**6. Uživatelská dokumentace** **28**
6.1. Vývojář testů . . . . . 28
6.1.1. Rozšíření o nový formát . . . . . 28
6.1.2. Vkládání dat do repozitáře . . . . . 29
6.2. Běžný uživatel . . . . . 33
6.2.1. Průběh suite runu . . . . . 33
6.2.2. Průměr výsledků suite runů . . . . . 33
6.2.3. Srovnání suite runů . . . . . 33
6.2.4. Statistický pohled na suite runy . . . . . 34

**7. Výkonnostní testy** **39**
7.1. Výsledky . . . . . 40
7.2. Shrnutí . . . . . 41

**8. Možnosti rozšíření** **42**

**Závěr** **45**

**Conclusions** **46**

**Reference** **47**

**A. REST API** **49**

**B. Obsah přiloženého CD** **52**

**C. Instrukce pro spuštění repozitáře** **53**

# Seznam obrázků

1. Architektura maven projektu . . . . . . . . . . . . . . . . . . . . . . 20
2. Návrh vrstev v serverové části . . . . . . . . . . . . . . . . . . . . . 22
3. Pohled na průběh suite runu . . . . . . . . . . . . . . . . . . . . . . 35
4. Pohled na průměr suite runů z daného buildu . . . . . . . . . . . . 36
5. Srovnání suite runů mezi sebou . . . . . . . . . . . . . . . . . . . . 37
6. Pohled na statistické vlastnosti suite runů . . . . . . . . . . . . . . 38

# Seznam tabulek

1. Výsledky testu na zápis . . . . . . . . . . . . . . . . . . . . . . . . . . 40
2. Výsledky testu na čtení . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

# 1. Úvod

Téměř každý programátor během vývoje softwaru používá různé testovací techniky, ať už se například jedná pouze o kontrolu zobrazení výsledného HTML ve webovém prohlížeči nebo ověření správnosti služby, která provádí převod mezi bankovními účty. V dnešní době existuje obrovské množství nástrojů a technologií, které se snaží vývoj a správu testů co nejvíce usnadnit a zautomatizovat. Každý takový nástroj je navržen a specializován pouze na určitou množinu problémů, se kterými se můžeme ve světě testovaní setkat. Takový počet různých testovacích nástrojů může občas zapříčinit komplikace ve vlastním vývoji. Část týmu se musí seznámit s dokumentací, integrováním a schopností zpracovávat výstupy těchto nástrojů. Nastává otázka, jestli nelze tento postup zjednodušit návrhem jednotného rozhraní. Nejde přitom o to nahradit všechny výše zmíněné nástroje pouze jedním univerzálním, ale mít možnost vkládat výstupy těchto nástrojů do rozhraní bez ztráty informace a umožnit jednotný pohled nezávislý na použitých technologiích.

Tato práce je zpracována ve spolupráci s firmou RedHat a jeden z požadavků byl, aby výsledný repozitář běžel na libovolném Java EE aplikačním serveru. Práce je rozdělena do několika kapitol, ve kterých se zabývám návrhem a řešením problémů, které mě při vývoji potkaly.

V první kapitole jsou rozebrány nejdůležitější typy testů. Je popsáno, kde se s nimi můžeme setkat a jaké informace o nich by nás mohly z hlediska repozitáře zajímat. Dále je z každé kategorie uvedeno pár nástrojů rozšířených na platformě Java EE.

Druhá kapitola se snaží shrnout a vysvětlit všechny požadavky, které jsou na repozitář kladeny jak z hlediska vývojáře testů tak i z hlediska koncového uživatele, který bude výsledky testů analyzovat. Důraz je především kladen na flexibilitu a rozšiřitelnost aplikace.

Ve třetí kapitole se zabývám vlastním návrhem aplikace splňující předdefinované požadavky. Především jde o architekturu, použité technologie a popis řešení problémů, se kterými jsem se setkal.

Čtvrtá kapitola slouží jako dokumentace uživatelům, kteří repozitář budou používat. Z počátku jsou detailně popsány možnosti rozšíření, které vývojáři testů budou často využívat. Jde především o možnost přizpůsobení aplikace na jiný formát, přidání vlastních měření ve výkonnostních testech atd. V druhé části kapitoly je popsáno rozhraní z pohledu uživatele, který pracuje nebo analyzuje výsledky testů.

Poslední kapitola se zabývá ukázkou aplikace v praktičtějším slova smyslu. Součástí práce jsou výkonnostní testy, které slouží k otestování a uložení výsledků do sebe sama.

# 2. Základy testování

Podle [1] je testování proces ověřování, zda software splňuje požadavky, které byly stanoveny klientem během návrhu a vývoje. Z hlediska této diplomové práce nás ani tak nezajímá návrh a vytváření testů, jako spíš zpracovávání a analýza jejich výsledků. Nicméně protože je výsledná aplikace mířena jak pro vývojáře testů, tak pro koncové uživatele, kteří analyzují výstup těchto testů, je důležité zjistit jak s aplikací budou vlastně pracovat. V průběhu této kapitoly se budu zabývat nejčastějšími typy testování, se kterými můžeme během vývoje přijít do styku a jaké důležité informace je třeba do repozitáře ukládat.

## 2.1. Úrovně testování

Jednotlivé kusy kódu se dají testovat různými způsoby. Můžeme je testovat nezávisle na sobě, jako celek nebo v závislosti na prostředí (operační systém, kontejner, připojení do databáze, ...). Tyto způsoby můžeme označovat jako úrovně testování z hlediska jejich závislosti na prostředí. Následuje stručný rozbor těchto úrovní.

### 2.1.1. Unit testování

Při unit testování nám jde o testování malých kusů kódu, které samostatně vykonávají nějakou funkci. Hledáme tedy nejmenší možné testovatelné celky a testujeme je v naprosté izolaci. Je vidět, že jde o nejnižší úroveň, kde neuvažujeme ani interakci mezi těmito celky ani interakci s vnějším prostředím. Unit testy bývají zpravidla velmi rychlé a dokáží při ověřování správnosti nalézt chyby velmi brzo.

#### Nástroje na platformě Java EE

**JUnit, TestNG** Testovací frameworky, které umožňují spouštět unit testy a produkovat reporty o výsledcích testů v XML formátu. Způsob generování reportů se dá ovlivnit vytvořením a zaregistrováním vlastního listeneru. Tyto frameworky také do výsledných reportů vkládají informace o prostředí, ve kterém byly testy spuštěni vypsáním všech systémových proměnných.

**Mockito, EasyMock** Frameworky, které usnadňují testování nezávisle na prostředí. Pokud například testujeme objekt $A$, který je závislý na objektu $B$, můžeme pomocí tohoto frameworku místo pravého objektu $B$ vytvořit pouze jeho napodobeninu tak, že to objekt $A$ nepozná. Pak testujeme objekt $A$ izolovaně.

#### Informace v repozitáři

Následující seznam obsahuje informace, které je možné v repozitáři uchovávat. Informace se budou získávat přímo z reportů některého použitého testovacího frameworku.

- název suity (zařazení testu do skupiny)
- název testu
- výsledek testu (jestli byl test úspěšný nebo neúspěšný a popřípadě proč)
- celková doba trvání testu
- čas začátku testu
- čas konce testu

Je vidět, že tyto informace nejsou z hlediska uživatele, který provádí analýzu moc užitečné. Smysl unit testů je spíše ten, že mají co nejdříve odhalit vzniklé chyby. Na druhou stranu je možné využít repozitář například k archivaci výsledků nebo třeba analyzovat dobu trvání testu během vývoje.

#### Generování výsledků

Jak bylo řečeno, pro generování výsledků testů lze použít některý z existujících testovacích frameworků. Pokud si blíže nastudujeme formáty, které jednotlivé frameworky generují, můžeme si všimnout velké nejednotnosti. Například u Junit frameworku nelze nalézt informace o začátku nebo konci testu. Nicméně tyto informace jsou například nepovinné a nemusí mít pro koncového uživatele příliš význam. Důležité je, že téměř každý framework umožňuje návrh generování vlastního formátu, kde si tyto informace můžeme dopsat, pokud je potřebujeme.

### 2.1.2. Integrační testy

U integračních testů se snažíme z již otestovaným malých celků sestavit systém a ten dále testovat. Na rozdíl od unit testů nás nezajímá funkčnost jednotlivých malých kousků aplikace, ale spíše jejich komunikace. Vetšinou je sestavený systém závislý na vnějším prostředí (kontejner, databáze, . . . ). Ve většině případu se snažíme toto prostředí uměle vytvořit. Například vytvoření speciální testovací databáze, vyhrazení testovacího aplikačního serveru apod.

### Nástroje na platformě Java EE

#### Arquillian

Výborný framework, který velmi usnadňuje vývoj integračních testů. Arquillian je schopen programově vytvořit testovací archiv, který je následně nahrán na cílový aplikační server. Sám vývojář testu si může určit, co bude archiv obsahovat a co ne. Tento framework je navíc integrovatelný s nástrojem Apache Maven 5.1., takže celý proces vytváření archivu, nahrání na aplikační server a následné spuštění testů je zautomatizovaný. Navíc spuštění testů pomocí Arquillianu je založeno na JUnit frameworku, takže výstupy jsou velmi podobné.

### Informace v repozitáři

Informace, které je vhodné uchovávat v repozitáři jsou velmi podobné těm v případě unit testů. Důležité mohou být u tohoto typu testů například informace o prostředí, ve kterém testy běžely.

- název suity (zařazení testu do skupiny)
- název testu
- výsledek testu (jestli byl test úspěšný nebo neúspěšný a popřípadě proč)
- celková doba trvání testu
- čas začátku testu
- čas konce testu
- informace o prostředí

Tyto informace můžou opět sloužit k archivaci nebo zkoumání délky testu během vývoje systému. Zajímavější pro koncového uživatele může být analýza výsledku testů v závislosti na prostředí, ve kterém byl spuštěn. V úvahu můžeme například brát testování webové aplikace v různých prohlížečích, kdy v jednom prohlížeči integrační test může projít a v jiném ne. Nabízí se otázka, že u těchto testů je možné sledovat délku testu v závislosti na prostředí a tím měřit výkon. Je to sice možné, ale připomeňme, že integrační testy testuji funkčnost systémů, ne výkon.

#### Generování výsledků

Generování výsledků je velmi podobné jako tomu bylo u unit testů, především pokud použijeme Arquillian, který je založen na Junit. V opačném případě si sami musíme zajistit vytváření prostředí a měli bychom být schopni generovat vlastní reporty, ve kterých budou informace o vytvořeném prostředí obsaženy. Samotný repozitář není omezen pouze na některé formáty, ale mělo by být jednoduché přizpůsobit repozitář na uchovávání našich vlastně vytvořených formátů.

### 2.1.3. Výkonnostní testy

Nakonec se dostáváme k výkonnostním testům, u kterých nám nejde o testování systému z hlediska funkčnosti, ale zajímá nás stabilita a škálovatelnost v různých podmínkách. Představme si, že jedním z požadavků je, aby systém dokázal zpracovat požadavky od tisíců uživatelů zároveň. Stabilitou se myslí, že při náporu tohoto počtu uživatelů systém dokáže stále zpracovávat požadavky v rozumném čase. Většina takových systémů používá nějaké zdroje, například paměť, disk, databázové připojení atd. Obvykle platí, že pod větší zátěží se náročnost na tyto zdroje zvyšuje. Analýza toho, jak se spotřeba těchto zdrojů mění v závislosti na zatížení systému se označuje škálovatelnost. Škálovatelnost systému je především důležitá u systémů, které jsou vystavovány obrovskému počtu uživatelů.

Z hlediska vývojáře takových výkonnostních testů je nutné tyto zdroje v průběhu testu měřit. Může jít například o využití procesoru, spotřebě paměti, zatížení disku atd. Zdrojů, které lze takto měřit existuje celá řada. Způsob, jakým lze vlastnosti těchto zdrojů změřit je úkol vývojáře testů. Existuje mnoho externích nástrojů, které taková měření umožňují a úkolem repozitáře je schopnost výstupy z těchto nástrojů zpracovat. V následujícím seznamu je uvedeno pár příkladů, pomocí kterých je možno monitorovat tyto zdroje.

- nástroje a API operačního systému – k monitorování zdrojů můžeme využít existující nástroje a funkce operačního systému, na kterém je aplikace testována, například příkaz top na unixových systémech.

- JMX api – umožňuje monitorování virtuálního stroje Javy a je součástí platformy Java SE od verze 5. Pomocí JMX je možné se připojit k JVM, na kterém aplikace běží a přímo monitorovat zdroje, které aplikace využívá.

- JConsole – grafický nástroj založen na výše zmíněném JMX. Stačí znát identifikátor procesu nebo URL JMX služby a můžeme se snadno k cílovému JVM připojit a začít s monitorováním.

- Apache JMeter – pomocí toho nástroje je možné simulovat zatížení systému a zkoumat dobu odezvy. Pokud chceme například otestovat zatížení webové

aplikace, je možné místo programování jednotlivých testů použít JMeter pro odesílání jednotlivých požadavků.

Samozřejmě takových nástrojů a technologií existuje celá řada (Ganglia, Nagios, New relic ...). Výše uvedený seznam měl shrnout základní možnosti pomocí, kterých se dá monitorování zdrojů docílit. Jak bude dále uvedeno v kapitole zabývající se testováním zátěže repozitáře, bylo pro monitorování zdrojů využito JMX.

#### Informace v repozitáři

Informace, které budeme uchovávat v repozitáři budou vycházet z integračních testů. Navíc zde budou obsaženy údaje o průběžném měření určitých zdrojů.

- název suity (zařazení testu do skupiny)
- název testu
- výsledek testu (jestli byl test úspěšný nebo neúspěšný a popřípadě proč)
- celková doba trvání testu
- čas začátku testu
- čas konce testu
- informace o prostředí
- informace o zdrojích (čas a jednotlivé naměřené hodnoty)

Tyto údaje jsou z hlediska uživatele, který nad nimi bude provádět analýzu důležité. Je možné například analyzovat využití specifického zdroje v závislosti na zatížení systému i na prostředí, ve kterém byl systém testován. Jednotlivé naměřené hodnoty se mohou porovnávat mezi jednotlivými verzemi projektu během jeho vývoje a tím zamezit vzniku větších performance chyb před nasazením systému do provozu.

## 2.2. Shrnutí

V této kapitole jsme se seznámili se základy testování. Uvedli jsme typy a úrovně testování, se kterými se můžeme v praxi nejčastěji setkat. U každé úrovně jsme si stanovili, jaké informace by jsme měli být schopni v repozitáři uchovávat a jakým způsobem můžeme tyto informace získat. Nakonec jsme ukázali, jakým způsobem budou koncoví uživatelé tyto informace analyzovat a měli bychom jim k tomu repozitář přizpůsobit.

# 3. Analýza dostupných nástrojů

Předtím než se pustím do formulace požadavků, které bude výsledný repozitář splňovat, uvedl bych některé existující nástroje. Nástrojů zabývajících se monitorováním a analyzováním je celá řada. Jejich možnosti a architektura mě významně ovlivnila při návrhu. V následující části jednotlivé nástroje popíšu ve skupinách z hlediska funkčnosti.

## 3.1. Continuous integration servery

Tento typ serverů sice neslouží přímo k monitorování a analýze testů, ale umožňuje jejich kontinuální spouštění a archivaci. U těchto nástrojů mě ovlivnil především způsob jakým jednotlivé výsledky přehledně zobrazují. Určitě je dobré, že nedělají rozdíl mezi jednotlivými typy testů a snaží se je sjednotit. Nevýhodou těchto nástrojů je, že chybí analýza jednotlivých výsledků testů. Jako příklad je možné uvést Jenkins a Team city.

## 3.2. Repozitáře

Repozitář je služba poskytující API pro import dat. Mezi nejrozšířenější repozitář využívaný v praxi bych uvedl [2] Graphite. Graphite obsahuje tři základní komponenty:

**carbon** služba, která přijímá data,

**whisper** úložiště dat,

**graphite webapp** webový klient.

Výhodou tohoto rozdělení je určitě škálovatelnost. Graphite definuje vlastní protokol pomocí, kterého lze odesílat jednotlivé naměřené hodnoty na carbon. Vytvoření vlastního protokolu způsobilo vznik velkého množství malých klientských nástrojů, které tento protokol implementují. Webový klient funguje na bázi HTTP protokolu, kdy uživatel pomocí query parametrů definuje dotaz a klient mu vrátí graf. Tento způsob umožňuje jednoduché zobrazení grafů na vlastních stránkách. Hlavní nevýhodou tohoto nástroje je datové schéma, které nepodporuje data dále seskupovat podle dalších parametrů jako projekt, test nebo skupina.

## 3.3. Monitorovací nástroje

Úkolem monitorovacích nástrojů je snímání stavu zdrojů. Takových nástrojů existuje celá řada (RRDTool, Nagios, New relic . . . ) a liší se pouze podle platformy. Zajímavým nástrojem z této skupiny je Ganglia. Jde o aplikaci navrženou

pro monitorování distribuovaných systémů. Ganglia pak definuje vlastní distribuovaný algoritmus, který monitoruje data na celém clusteru a archivuje je. Součástí nástroje je PHP webový klient, který umožňuje analýzu výsledků.

## 3.4. Shrnutí

V této kapitole jsem se snažil uvést hlavní nástroje, které se používají pro monitorování a analyzování. Je vidět, že nástroje lze podle funkčnosti rozdělit do tří hlavních kategorií. Repozitář, který je navržen v této práci by měl podporovat všechny tyto funkčnosti. Celkový seznam základních požadavků a vlastní návrh repozitáře je popsán v následujících kapitolách.

# 4. Požadavky a vlastnosti repozitáře

V této kapitole se pokusím shrnout veškeré informace, které jsme si doposud stanovili a formuluji základní požadavky a vlastnosti, které by měl výsledný repozitář mít. V předchozí kapitole se ukázalo, že repozitář budou používat dva druhy uživatelů. Prvním druhem jsou vývojáři výkonnostních testů, kteří budou do repozitáře jednotlivé informace o výsledcích testů odesílat. Navíc by měli mít možnost repozitář nějakým způsobem rozšířit, například schopnost importovat vlastní formát výstupu, přidání dodatečných informací u výsledků atd. Druhým typem uživatelů jsou ti, kteří data uložené v repozitáři pouze čtou a analyzují. Je vhodné tedy celkovou aplikaci rozdělit na dvě části. První část bude sloužit jako služba, která bude schopná přijímat data a ukládat je do repozitáře. Druhá část bude klientská aplikace (web, desktop, ...), která bude komunikovat se službou, ze které bude data číst a zobrazovat je koncovému uživateli.

## 4.1. Datové schéma

Předtím něž se pustím do popisu požadavků na výše zmíněné části repozitáře, je třeba formulovat obecně s jakými daty se v repozitáři bude pracovat.

### Projekt

Základní datovou entitou v repozitáři bude projekt. Ať už píšeme, spouštíme nebo analyzujeme testy a jejich výsledky, tak vždy v kontextu nějakého projektu. Repozitář by měl být schopen udržovat informace o výsledcích celé řady projektů.

### Verze projektu

Projekt se postupem času vyvíjí a existuje v jednotlivých verzích. Spuštění určitého testu může mít v různých verzích projektu jiné výsledky. Je tedy nutné uchovávat informaci o verzích projektu a výsledky testů přímo s těmito verzemi spojovat. To dovoluje analyzovat výsledky testů mezi jednotlivými verzemi.

### Výsledek testu

Tato datová entita bude uchovávat veškeré informace o výsledku spuštěného testu nad určitou verzí projektu. Testy se většinou spouštějí ve skupinách (suitách), které testy logicky seskupují. Může se stát, že výsledek určitého testu bude závislý i na skupině, ve které se pouštěl.

## 4.2. Služba

První logickou část aplikace označím obecně jako službu. Její úkol spočívá ve zpracování dat a jejich následnému ukládání do repozitáře. Navíc služba definuje jediný možný přístup k datům uloženým v repozitáři. Následuje popis jednotlivých požadavků a vlastností této částí repozitáře.

### Platformová nezávislost

I přesto, že bude aplikace napsána na platformě Java EE, neměla by být dostupnost služby jakkoliv závislá na této nebo jiných platformách. Součástí repozitáře by tedy měl být jednoznačně definovaný protokol, pomocí kterého bude komunikace s rozhraním služby probíhat. Protokol by měl být samozřejmě nezávislý na operačním systému, platformě i jazyce.

### Správa projektů a jejich verzí

Součástí služby by měla být možnost spravovat projekty a jejich verze. Pod správou se myslí možnost vytváření, mazání, upravování a hledání. Tyto informace nebudou většinou obsaženy ve výstupech testů a sami uživatelé by měli být schopni tyto informace předdefinovat nebo v průběhu měnit.

### Import výsledků testů

Základní vlastností repozitáře je schopnost importovat jednotlivé výsledky testů. Importování může probíhat různými způsoby. Nejčastějším z nich může být například importování konečného reportu, který se vygeneroval po dokončení všech testů dané skupiny. Tento způsob může v případě velkých velikostí reportů zatížit repozitář. Dalším způsobem může být postupné importování jednotlivých informací po malých kouscích hned jak budou k dispozici. Tento způsob zatížení repozitáře zmenšuje. Služba by měla podporovat oba tyto způsoby.

### Rozšíření o vlastní formát

Služba by měla být schopná přijímat a zpracovávat informace různých formátu. Způsob jakým daný formát služba zpracuje je funkční celek, který součást služby samotné. Formátů, do kterých jsou výsledky testů generovány je obrovské množství. Vývojář by měl být schopen jednoduše službu rozšířit o funkci zpracovávání vlastního formátu.

### Rozšíření o dodatečné informace

Další možností rozšíření se týká datového schématu. Může se například jednat o dodatečné informace o projektech a jejich verzích. Častěji budou vývojáři potřebovat přidávat vlastní informace do schématu u výsledků testů. Vezměme si například výkonnostní testy, u kterých se můžeme setkat s obrovským množstvím měřitelných veličin. To samé se týká u popisu prostředí třeba u integračních testů. Repozitář by měl uživateli jednoduchým způsobem umožnit tyto dodatečné informace integrovat.

### Integrita dat

Posledním a velice důležitým požadavkem je integrita dat. Jelikož je služba zodpovědná jak za sběr a uchování dat, tak za jejich přístup, měla by kontrolovat jejich integritu. V žádném případě by nemělo dojít k poruše dat v repozitáři jak při jejich importu, tak při jejich manipulaci.

## 4.3. Klient

Klientská část aplikace by měla uživateli umožňovat snadnou analýzu dat uložených v repozitáři. Vlastní klient může být implementován například jako webová aplikace, desktopová aplikace nebo i dalšími způsoby. Repozitář by měl co nejvíce usnadňovat implementaci takového klienta. Následuje obecný popis požadavků a vlastností klientské části.

### Komunikace se službou

Cílem klienta je poskytnout uživateli vizualizaci a správu dat uložených v repozitáři. Toto obstarávání dat bude zajištěno výhradně přímou komunikací se službou repozitáře pomocí jejího protokolu.

### Vizualizace dat

Nejdůležitější částí klienta bude zobrazování dat uživateli, tak aby jejich analýza byla co nejsnadnější. Jedná se například o grafové zobrazení naměřených hodnot během výkonnostního testu, sledování změn atributů během vývoje systému atd.

# 5. Návrh repozitáře

V této kapitole si blíže popíšeme postup při implementaci repozitáře splňující požadavky definované v předchozí kapitole. V první části rozeberu návrh systému z obecného hlediska a ve zbytku kapitoly se budu zabývat jednotlivými částmi a popisu použitých technologií. Tato kapitola by neměla sloužit jako programátorská dokumentace obsahující úryvky kódu, ale popisovat význam jednotlivých částí systému. Budu se snažit popsat jednotlivé technologie a jejich význam při implementaci a naznačit řešení problémů, které se při návrhu nebo implementaci vyskytly. Uživatel, který si tuto kapitolu přečte by měl být schopen celý systém implementovat na jakékoliv platformě bez znalosti dalších implementačních detailů.

## 5.1. Apache Maven

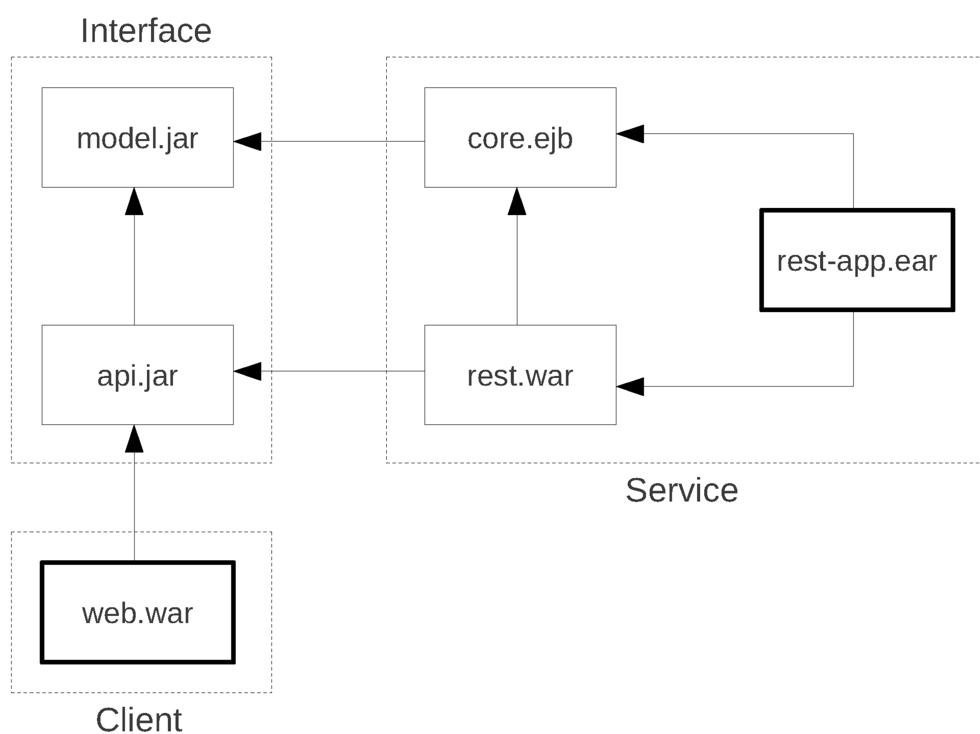
Apache maven je nástroj sloužící pro správu projektů. Usnadňuje proces sestavování projektu, správu závislostí atd. Mimo hlavní centrální repozitář existuje spousta dalších firemních repozitářů (JBoss), ze kterých je možné závislosti do projektu přidávat. Následující seznam shrnuje přednosti tohoto nástroje.

- jednoduchá správa multimodulových projektů
- verzování a uchovávání jednotlivých modulů v lokálním nebo externím repozitáři
- správa závislostí na další moduly, jejich automatické vyhledání, stažení a přidání k aktuálnímu modulu
- velké množství pluginů rozšiřující možnosti při sestavování projektu, například plugin pro spouštění testů, nahrání archivu na aplikační server, sestavení archivu atd.

Více informací o tomto nástroji lze nalézt na jeho domovské stránce http://maven.apache.org/. Repozitář bude sestavován pomocí mavenu a popis jednotlivých příkazů je popsán v příloze.

## 5.2. Struktura projektu

Na obrázku 1. jsou uvedeny jednotlivé moduly maven projektu. Celý projekt je rozdělen do tří hlavních částí podle specifikace požadavků v předchozí kapitole. Šipkami jsou vyjádřeny závislosti a tučně ohraničené moduly jsou určeny pro nasazení na aplikační server. V dalších sekcích bude následovat obecná charakteristika a význam jednotlivých částí systému a modulů, které obsahují.




Obrázek 1. Architektura maven projektu

### 5.2.1. Rozhraní

Tento modul slouží k propojení klienta se serverem. Výhodu tohoto společného rozhraní pocítíme až v případě kdy budeme chtít vytvořit nového klienta pro komunikaci se serverem na platformě Java. Pak nám stačí si pouze pomocí mavenu přidat závislost na tento modul a rovnou použít vytvořené objekty ke komunikaci se serverem.

Modul se skládá ze dvou hlavních částí:

#### Model

Model obsahuje definici dat, která se přenášejí mezi serverem a klientem. Datový model tvoří obyčejné POJO (Plain old java objects) objekty. Tyto objekty by měli být serializovatelné a uložitelné do databáze.

#### Api

Definuje komunikační protokol mezi serverem a klientem. Komunikační protokol by měl poskytovat tyto operace

- dotahování dat z repozitáře
- vkládání dat do repozitáře
- manipulace s daty (editace, mazání, . . . )

Z předpokladů v minulé kapitole vyplývá, že protokol by měl být nezávislý. K tomuto účelu se nejlépe hodí použití architektury REST, která funguje na bázi HTTP protokolu. Blíže se k implementaci protokolu dostanu v serverové části. V příloze A. naleznete kompletní výčet REST operací, které rozhraní podporuje.

### 5.2.2. Klient

Jak již bylo řečeno, se serverem lze komunikovat pomocí REST API, které je definováno v Interface modulu. Díky tomu je možné přistupovat k serveru například pomocí linuxového nástroje Curl, z Javascriptu nebo jakéhokoliv jiného nástroje, který umí komunikovat pomocí HTTP protokolu. Například pro vkládání dat do repozitáře budeme určitě chtít, aby klient byl co nejjednodušší a nezatěžoval nám zbytečně aplikační server. K tomuto účelu můžeme využít například výše uvedený Curl a podobné programy dostupné na naší platformě. Zajímavější způsob využití klienta je v možnosti analýzy a vizualizaci dat v repozitáři. Tento modul vytváří webovou aplikaci, která umožňuje uživateli provádět následující operace

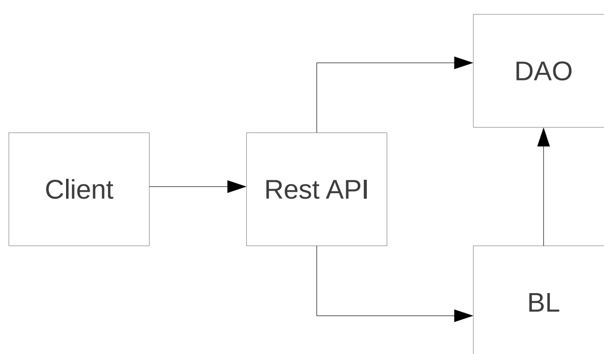
- správa projektů a jejich verzí

- import výsledků výkonnostních testů přímo z uživatelského rozhraní
- vizualizaci dat v repozitáři
- provádět různé statistické analýzy nad daty

Způsob použití webového klienta je podrobněji rozepsán v následující kapitole. Pro vytvoření webového klienta jsem vybral technologii [3] JSF (Java server faces), která je na platformě Java EE velice rozšířená a taky jsem se chtěl naučit i jiný framework pro vytváření webových aplikací. Java server faces je standardní komponentový MVC (Model view controller) framework, nad kterým komunita vytváří vlastní komponentové knihovny jako jsou [4] Richfaces, [5] Primefaces atd. Tyto knihovny pak nabízejí bohaté množství nejrůznějších komponent, takže vlastní vývoj uživatelského rozhraní je velmi efektivní. Poslední technologií, kterou jsem se rozhodl z Java EE stacku vyzkoušet je [6] Jboss Seam.

### 5.2.3. Server

Základní částí repozitáře je určitě server. Jeho úkolem je správa dat a jejich zpřístupnění pomocí komunikačního protokolu. V této části bych rád ukázal jakým způsobem jsem tuto část implementovat, s jakými problémy jsem se setkal a jaké výhody/nevýhody tento návrh má. Samotný návrh jakékoliv aplikace je jednou z nejdůležitější věcí, protože na tom závisí její budoucí použitelnost a škálovatelnost. Při navrhovaní repozitáře jsem se snažil splnit podmínky, které jsem uvedl na začátku práce. Konečná implementace a seznam vrstev v serverové části je zobrazena na následujícím obrázku.




Obrázek 2. Návrh vrstev v serverové části

Tento návrh obsahuje tři základní vrstvy. Ač se zdá velmi jednoduchý tak má velký vliv na škálovatelnost a robustnost celé aplikace. Popis jednotlivých vrstev a jejich výhody/nevýhody jsou popsány v následující části kapitoly.

### DAO – Data access object

Tato vrstva nebo návrhový zdroj se snaží odstínit celou aplikaci od fyzického uložení dat, například v databázi. Ve zbytku aplikace se už nemusíme trápit se skládáním SQL dotazů a využíváme přímo DAO Api. Tento návrh má tu výhodu, že veškerá manipulace s daty probíhá výhradně přes tuto vrstvu, což usnadňuje vývoj, logování a zabezpečení. Dlouhou dobu jsem hledal a zjišťoval na různých Java EE fórech jak často je tato vrstva v aplikacích využívána. Bohužel zjištění bylo spíše negativní. Většina aplikací psaných na této platformě využívá [7] JPA (Java persistence API), které většina odborníku bere jako samotnou DAO vrstvu. Tento názor mě bohužel nevyhovoval protože JPA je sice specifikace a má mnoho implementací, ale není jednoduché najít například implementaci JPA pro NoSQL databáze. Po velkých studiích této specifikace a kódu [8] Hibernatu (především jeho Criteria API) jsem se rozhodl, že jsi pro vlastní účely napíši vlastní Criteria API. Nejedná se o celkové přepsání veškerého kódu, ale pouze částí, které jsou pro repozitář důležité. Výhodu, kterou mi to přineslo je to, že je repozitář absolutně nezávislý na úložišti dat. Současná implementace sice používá JPA a hibernate, ale není nejmenší problém napsat další implementaci například pro MongoDB.

### Validace

Další součástí DAO vrstvy je validace, která zaručuje integritu dat na nejnižší možné úrovni. K tomuto účelu jsem použil rozšířenou specifikaci JSR 303 a její implementaci [9] Bean Validation od Hibernatu. Jedná se o anotací řízenou validaci, která se dá libovolně rozšířit. Jedinou věcí, která mi v této specifikaci chyběla byla validace závislá na databázi. Jedná se například o unikátnost verze projektu atd. Pro tyto účely jsem si specifikaci rozšířil a napsal vlastní validace, které využívaly výše zmíněné Criteria Api.

### BL – Business Logic

Na této vrstvě je implementovaná veškerá logika aplikace. Pro manipulaci s daty tato vrstva využívá rozhraní, které definuje DAO. Výhodou takto oddělené logiky je jeho testovatelnost. V tomto modulu je napsáno přes 100 unit testů, které se snaží otestovat co největší část této vrstvy. Z vlastní zkušenosti můžu říct, že mi tyto testy pomohly ušetřit hodiny hledání chyb. Tato vrstva je implementována pomocí specifikace [10] EJB 3.1 (Enterprise Java Beans). Důvod, proč jsem si tuto technologii vybral byl kvůli škálovatelnosti. I když pro tento typ aplikace nebude nejvíc vytěžována vrstva s logikou, je dobré mít aplikaci navrženou do budoucna tak, aby se tomuto problému dalo vyhnout. Určitě největší zátěž na této vrstvě bude deserializace datového schématu

z importovaných dat uživatelem, což může být při velkém počtu uživatelů procesorově náročně. Modul se sestavuje do speciálního EJB archívu, který je řízen EJB kontejnerem. Není problém mít v clusteru velké množství aplikačních serverů s EJB kontejnery. Následně při volání metody na BL vrstvě dojde v kontejneru k přesměrování na nejnevytíženější server a ten požadavek zpracuje.

### REST API

Zbývá vyřešit problém jak DAO a BL vrstvu zpřístupnit přes komunikační protokol. Tomuto účelu nejlépe vyhovuje architektura zvaná REST (Representational State Transfer). Je to architektura založená na protokolu HTTP. Základním principem je, že máme v aplikaci nějaké zdroje, které mají svůj stav. V našem případě můžeme brát jako zdroj entitu datového schématu. Každý zdroj má svoje unikátní URI (Uniform Resource Identifier) pomocí, kterého lze zdroj identifikovat. Pro manipulaci s těmito zdroji se používají HTTP metody GET, POST, PUT a DELETE. Když to všechno shrneme dohromady máme univerzální protokol založený na REST architektuře pomocí, kterého můžeme zpřístupnit datové schéma.

Asi nejtěžším problémem, se kterým jsem se při zpracování této práce setkal byl způsob jak navrhnout REST rozhraní. Jak bylo řečeno, tak veškerá manipulace s daty půjde právě přes tohle rozhraní. Můj cíl nebyl jenom vytvořit rozhraní, které bude splňovat požadavky repozitáře, ale je nutné aby nad tímto rozhraním bylo možné implementoval libovolného klienta. V praxi jsem se setkal z mnoha různými implementacemi REST a popravdě se jim ani tak říkat nedá. Zastávám názoru "Špatné frontend API, nepoužitelná aplikace." Snažil jsem se najít nějakou odbornou literaturu, která se návrhem správného API zabývá a nakonec jsem narazil na knihy [11] a [12]. Tato literatura mi velice pomohla při návrhu a řešení velkého množství problémů. V následujícím seznamu bych chtěl sepsat ty nejdůležitější pravidla pro tvorbu REST Api:

#### 1) Jednoduchá a srozumitelná URI

Každý zdroj by měl mít přiřazené dvě unikátní URI.

```
\projects
\projects\id
```

První z nich označuje kolekci zdrojů, druhé identifikuje konkrétní zdroj. Doporučuje se přitom dávat zdrojům v URI jména v množném čísle.

#### 2) Pro manipulaci se zdroji používat HTTP operace

Význam jednotlivých operací nad dvěma možnými typy URI je popsán níže.

```
GET    /projects    - vrátí všechny projekty
POST   /projects    - vytvoří nový projekt
PUT    /projects    - aktualizuje určitou množinu projektů
DELETE /projects    - smaže všechny projekty

GET    /projects/id    - vrátí daný projekt pokud existuje
POST   /projects/id    - chyba
PUT    /projects/id    - aktualizuje daný projekt pokud existuje
DELETE /projects/id    - smaže daný projekt
```

**3) Asociace mezi zdroji**

Velmi často jsou jednotlivé zdroje mezi sebou provázané a chceme nějakým způsobem tyto vazby v api modelovat.

```
GET    /projects/id/builds    - vrátí všechny verze daného
                                projektu
POST   /projects/id/builds    - vytvoří novou verzi do daného
                                projektu
PUT    /projects/id/builds    - aktualizuje určitou množinu
                                verzí daného
                                projektu
DELETE /projects/id/builds    - smaže všechny verze daného
                                projektu
```

Doporučuje se mít hloubku asociací pouze do první úrovně, protože to je srozumitelnější.

**4) Projekce/selekce pomocí query parametrů**

Pokud potřebujeme kolekci zdrojů filtrovat například pomocí některých predikátů, výrazně se doporučuje tyto informace zahrnout za otazník v URI a reprezentovat je pomocí query parametrů. V žádném případě by neměly tyto informace měnit základní URI například `\projectsWithName\Project1` . Správný je níže uvedený způsob.

```
GET \projects?name="Project1"
```

**5) Zpracování chyb**

Pokud programátor experimentuje s neznámým rozhraním často se stává, že formuluje požadavek špatně. Od správného API se předpokládá, že chybu rozumným způsobem zpracuje a odešle klientovi zpět data, která mu pomůžou zabránit opakovanému volání chybného požadavku. Způsob, jakým server informuje o stavu zpracování požadavku je pomocí HTTP status kódů. V repozitáři používám tyto kódy

```
200 - OK
201 - Created
400 - Bad request
401 - Unauthorized
404 - Not Found
500 - Internal Server Error
```

V případě chybového zadání dotazu repozitář klientovi odešle navíc zprávu, ve které je detailněji popsáno proč k chybě došlo. Příklad chybové zprávy je uveden níže.


```
{"statusCode":400,
 "createdResource":false,
 "resourceId":null,
 "error":{"code":101,
          "type":"Validation error",
          "messages":[{"body":"Project with that name already
                               exists."}]}}
```


### 6) Verzování API

Pokud je api nasazeno v produkčním módu a existuje mnoho klientů, kteří rozhraní využívají, je často nutné podporovat verzované API. Některé api například Facebooku, Twitteru atd. udržují několik verzí, přičemž dají dopředu vědět, kdy starší verze přestanou podporovat. Klienti pak mají čas k přechodu na nejnovější verzi. Repozitář podporuje verzování a verze je obsažena v URI jako prefix před URI zdroje.

```
\v1\projects
\v2\projects
```

### 7) Požadavek, který nevrací zdroj

Můžou nastat případy, kdy pomocí rozhraní chceme provést nějakou operaci a výsledek této operace není zdroj. Pro tento způsob se používají slovesa, která jsou za URI zdroje.

```
GET /measurements/compute?function="max"&type=10
```

Tento požadavek spočítá maximální hodnotu ze všech měření daného typu.

# 6. Uživatelská dokumentace

V této kapitole bych rád shrnul jak repozitář efektivně používat. Jak jsem uvedl na začátku, s aplikací budou pracovat dva různé typy uživatelů. Prvním typem budou určitě vývojáři výkonnostních testů, jejichž cílem bude snaha dostat efektivně výsledky testů do repozitáře. Na druhé straně jsou uživatelé, kteří posuzují kvalitu produktu a pomocí repozitáře by měli být schopni zjistit výkonnost jejich aplikace. Kapitola obsahuje dvě části. V každé části se budu snažit uvést postup jakým každý typ těchto uživatelů bude s repozitářem pracovat tak, aby dosáhl svého cíle.

## 6.1. Vývojář testů

Jak jsem již uvedl, cílem vývojáře testů je přizpůsobit si repozitář tak, aby do něj šli vkládat výsledky testů. Jedna z možností jak vkládat data do repozitáře je využít vlastní REST API. Díky tomu, že je toto api nezávisle na platformě a jazyce, je jednoduché ho využít kdekoliv. Na druhou stranu musíme počítat se situací, kde nám výkonnostní testy provádí software třetí strany. Takovýto software většinou po skončení všech testů vrátí nějaký report o průběhu testů v nějakém specifickém formátu. Nastává otázka jak data v tomto formátu dostat do repozitáře. V následujících částech jsou rozebrány oba způsoby.

### 6.1.1. Rozšíření o nový formát

Uvažujme situaci, kdy chceme do repozitáře dostat data v libovolném formátu, který doposud není v repozitáři implementovaný. Jak bylo na začátku práce řečeno, nástrojů provádějících výkonnostní testy je velká spousta a neexistuje žádný standardizovaný formát, do kterého by tyto nástroje svůj výstup ukládaly. V repozitáři jsou defaultně implementovány formáty Junit, TestNG a můj vlastní PTR formát. Postup pro implementaci nového formátu je zhruba takový:

1. implementujeme rozhraní `EntityConverter<SuiteResult>`, ve kterém říkáme jak ze streamu vytvořit novou instanci typu `SuiteResult`.
2. do `ImportFormat` přidáme vlastní specifickou konstantu pro náš formát
3. nakonec v metodě `EntityConverterConfigurationImpl.initialize` zaregistrujeme nás konvertor pomocí metody `registerConverter`

Jak je vidět rozšíření repozitáře o nový formát je jednoduché. Nejsložitější je samozřejmě způsob jak implementovat deserializaci ze streamu. I v tomto směru nabízí repozitář mnoho pomocných funkcí, pokud je soubor typu XML nebo Json.

### 6.1.2. Vkládání dat do repozitáře

V této části bych rádu podal malou ukázku jak data importovat do repozitáře pomocí REST API. Příklady jsou prováděny pomocí Unixového programu Curl. Jsem si jistý, že podobný nástroj bude dostupný i na ostatních platformách, kde si můžete práci s repozitářem vyzkoušet. Příklady předpokládají, že je repozitář nahrán na aplikačním serveru a je dostupný na adrese **http://localhost:8080**

#### Vytvoření nového projektu

Pro vytvoření nového projektu s názvem **Project 1** a popisem **This is new project** použijeme následující požadavek

```
curl -H "Accept: application/json"
     -H "Content-type: application/json"
     -X POST -d '{"name":"Project 1",
                "description":"This is new project"}'
     http://localhost:8080/rest/v1/projects

->

{"statusCode":201, "createdResource":true, "resourceId":1, "error":null}
```

Vidíme, že vytvoření projektu proběhlo v pořádku a projekt je dostupný pod identifikátorem 1.

#### Vytvoření verzí projektu

Každý projekt se během vývoje mění - má nějaké verze. Vytvoříme v projektu dvě verze.

```
curl -H "Accept: application/json"
     -H "Content-type: application/json"
     -X POST -d '{"version":"1.0"}'
     http://localhost:8080/rest/v1/projects/1/builds

->

{"statusCode":201, "createdResource":true, "resourceId":1, "error":null}
```

```
curl -H "Accept: application/json"
     -H "Content-type: application/json"
     -X POST -d '{"version":"2.0"}'
     http://localhost:8080/rest/v1/projects/1/builds

->

{"statusCode":201, "createdResource":true, "resourceId":2, "error":null}
```

**Vytvoření suite runu**

Pokud spouštíme nějakou skupinu testů nad určitou verzí projektu, můžeme použít tento příkaz.

```
curl -H "Accept: application/json"
     -H "Content-type: application/json"
     -X POST -d '{"name":"Suite 1"}'
     http://localhost:8080/rest/v1/builds/1/suite-results

->

{"statusCode":201,"createdResource":true,"resourceId":1,"error":null}
```

**Vložení výsledku testu**

Jakmile doběhne test, měli bychom ho vložit do repozitáře pod daný suite run.

```
curl -H "Accept: application/json"
     -H "Content-type: application/json"
     -X POST -d '{"name":"Test 1",
                 "result":"SUCCEED",
                 "startedAt":1375347579000,
                 "finishedAt":1375347591000}'
     http://localhost:8080/rest/v1/suite-results/1/test-results

->

{"statusCode":201,"createdResource":true,"resourceId":1,"error":null}
```

### Vložení naměřených hodnot

Během suite runu můžeme měřit hodnoty nějakých veličin a postupně je do repozitáře vkládat.

```
curl -H "Accept: application/json"
     -H "Content-type: application/json"
     -X POST -d '{"value": 16.5,
                  "type":{"name":"cpu load",
                          "unit":"%",
                          "scalar":false},
                  "time":1375347589000}'
      http://localhost:8080/rest/v1/suite-results/1/measurements

->

{"statusCode":201, "createdResource":true, "resourceId":1, "error":null}
```

### Zobrazení informací o proběhlém suite runu

Jakmile suite run proběhne, můžeme si pro jistotu zobrazit jeho informace.

```
curl http://localhost:8080/rest/v1/suite-results/1

->

{"id":1,
 "name":"Suite 1",
 "createdAt":1375443149873,
 "startedAt":1375347579000,
 "finishedAt":1375347591000,
 "succeed":1,
 "skipped":0,
 "failed":0,
 "buildId":1,
 "buildVersion":"1.0",
 "suiteId":1,
 "testResultsCount":1}
```

### Import souboru s výsledky testů

V případě, že nám nějaký nástroj vrátil výsledky testů ve formátu, který je v repozitáři implementovaný, můžeme jej importovat pod verzi projektu následovně.

```
curl -H "Accept: application/json"
     -X POST -d @report.xml
     http://localhost:8080/rest/v1/builds/2/import.PTR

->

{"statusCode":201, "createdResource":true, "resourceId":2, "error":null}
```

a následně vypsat informace o importovaném suite runu

```
curl http://localhost:8080/rest/v1/suite-results/2

->

{"id":2,
 "name":"Suite 2",
 "createdAt":1375444815649,
 "startedAt":1375346250000,
 "finishedAt":1375346302000,
 "succeed":2,
 "skipped":0,
 "failed":0,
 "buildId":2,
 "buildVersion":"2.0",
 "suiteId":2,
 "testResultsCount":2}
```

## 6.2. Běžný uživatel

Tento typ uživatelů přistupují ke klientské části repozitáře a analyzují výsledky testů. Jak bylo uvedeno v popisu repozitáře je k dispozici webový klient, který umožňuje správu projektů a jejich verzí. Nejdůležitější části budou však různé pohledy na výsledky testů, které budou usnadňovat analýzu. V této části práce bych rád popsal pohledy, které jsem ve webovém klientovi implementoval a informace, které z nich lze vyčíst.

### 6.2.1. Průběh suite runu

Prvním standardním pohledem bude určitě detail proběhlého suite runu. Uvažujme případ, kdy se testy pravidelně spouštějí přes noc a my ráno chceme zjistit jak si produkt v testech vedl. V tomto pohledu máme k dispozici hodnoty jednotlivých veličin, které byly v průběhu suite runu naměřeny. V tomto pohledu můžeme pomoci výběrového seznamu měnit veličiny a testy, které nás zajímají. Tento test může být vhodný pokud chceme zjistit například detailní pohled na stav paměti během testu a zjistit jestli v některých momentech nebyl nad určitou hranicí.

### 6.2.2. Průměr výsledků suite runů

Předchozí pohled měl jednu nevýhodu, neměl dobrou vypovídací hodnotu. Mám tím teď na mysli fakt, že jednotlivá měření mohla být ovlivněná mnoha okolními věcmi. Pokud například zjistíme, že během suite runu se v jednom úseku zvýšila doba odezvy o desetinásobek, nemusí to být problém v samotné aplikaci. Nejjednodušší řešení spočívá v tom pouštět testy pravidelně a zjistit jestli se daný problém neopakuje. Přesně k tomuto účelu slouží další pohled. Princip spočívá v tom, že pravidelně spouštíme testy pro danou verzi a zajímá nás průměrný pohled nad těmito suite runy pro danou verzi. Jak je vidět na následujícím obrázků, pohled se od minulého téměř neliší. Přibyl tam navíc seznam, ze kterého je možné vybrat verzi, pro kterou chceme zjistit průměrný pohled. Server pak vezme veškeré naměřené hodnoty ve všech suite runech dané verze projektu a provede tzv. interpolaci hodnot. Výsledek by měl vystihovat průměrný průběh veličiny pro danou verzi projektu.

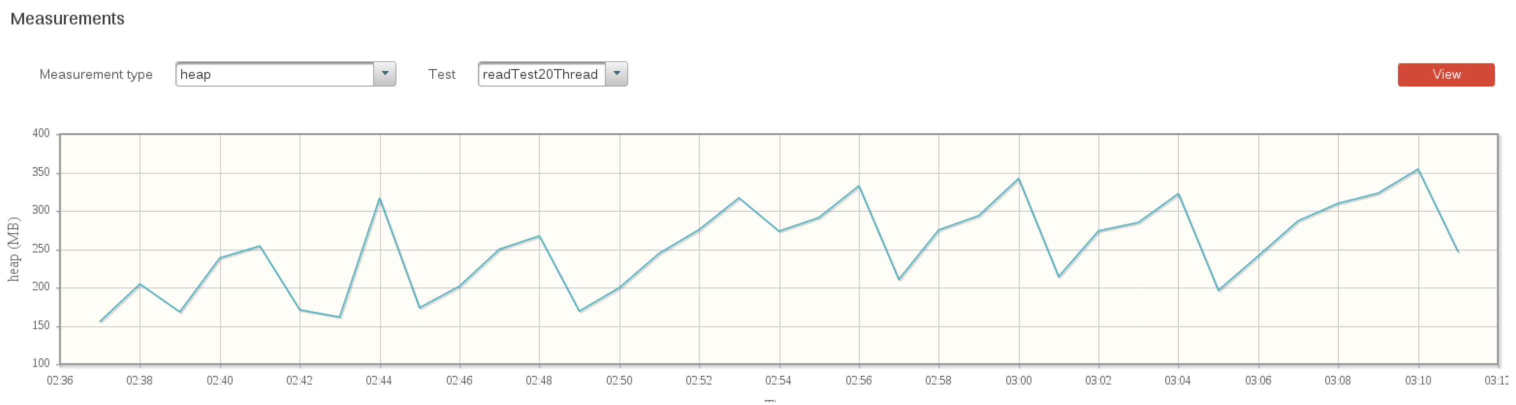
### 6.2.3. Srovnání suite runů

Dalším důležitým pohledem je srovnávání. Předchozí dva pohledy nám ukazovaly hodnoty pro daný suite run nebo build. Nás ovšem může zajímat, jak si tento nový suite run nebo build vede oproti předchozím. Zajímá nás, jestli se produkt zlepšil nebo naopak zhoršil po výkonnostní stránce. K tomuto účelu slouží třetí pohled, který nám pomocí svého rozhraní umožňuje srovnávat libovolné suite runy a buildy mezi sebou. Za zmínku stojí tlačítko agregovat, které

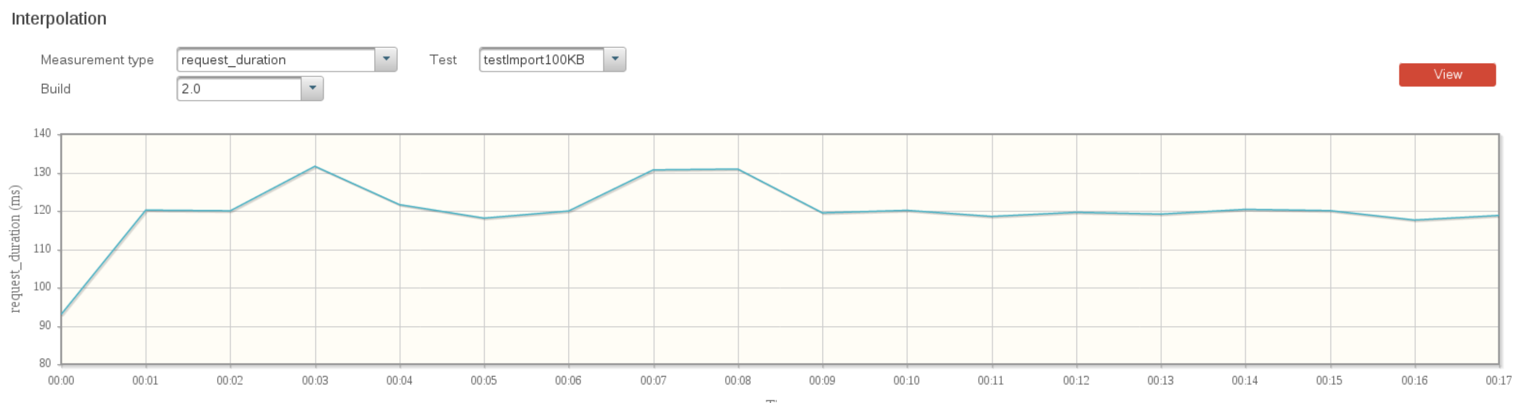
právě přepne pohled do módu srovnání jednotlivých buildů mezi sebou. V tomto případě pak server jako v minulém případě sjednotí všechny suite runy z buildů, vytvoří interpolaci a tu následně zobrazí.

### 6.2.4. Statistický pohled na suite runy

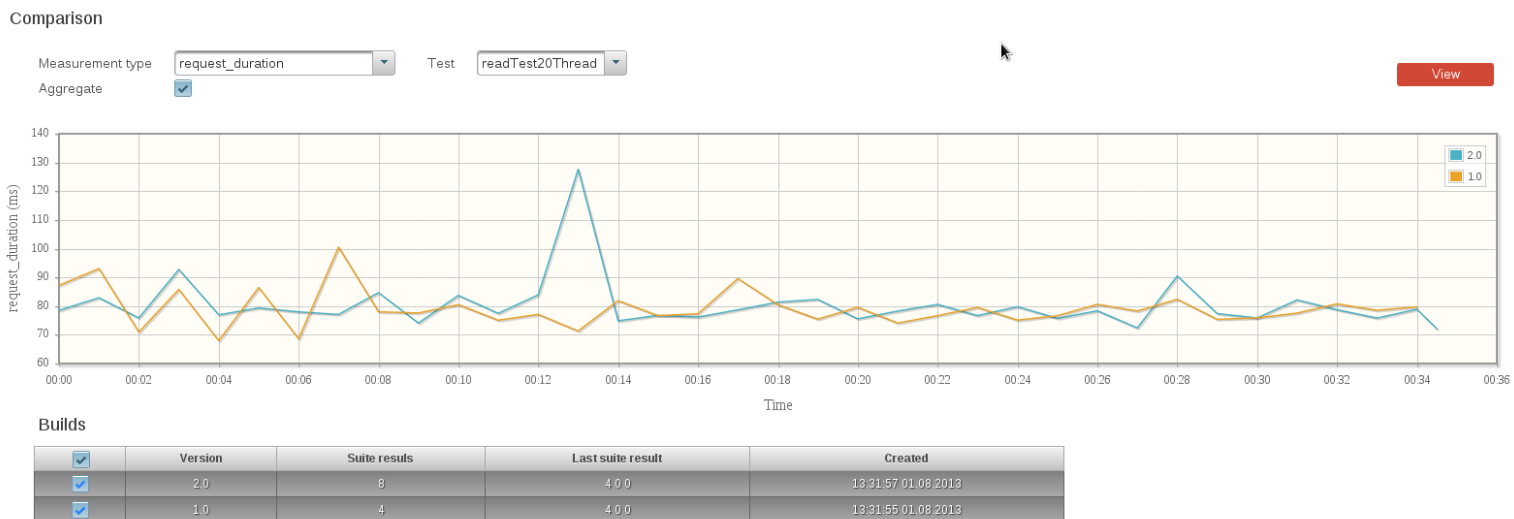
Předchozí srovnávací pohled se hodí jen pro malý počet suite runů nebo buildů, které srovnáváme. Při větším počtu se stává nepřehledným. Navíc nás ani tak nemusí zajímat přesné hodnoty suite runu v závislosti na čase, ale chceme spočítat nějakou jednu hodnotu, která určitým způsobem dokáže vývoj suite runu charakterizovat. Toho docílíme pomocí statistických funkcí. Můžeme například chtít vědět jakou mel webový server průměrnou odezvu během testu, jakou minimální popřípadě maximální. Nezajímají nás všechny jednotlivé hodnoty, které byly naměřeny během testů. Tyto statistické hodnoty navíc chceme porovnávat s ostatními verzemi projektu a zjišťovat odchylky. K tomuto slouží poslední pohled. V rozhraní si standardně navolíme veličinu a test, který chceme analyzovat a ze seznamu vybereme statistickou funkci, která se na měření aplikuje. Pohled má rovněž možnost jednotlivé suite runy agregovat. Server nejprve vezme jednotlivé suite runy a aplikuje na ně statistickou funkci. Pokud je zaškrtlá volba agregovat, server sjednotí tyto vypočítané hodnoty podle buildu a na hodnoty v těchto skupinách opakovaně aplikuje statistickou funkci. Ve výsledku můžeme jednoduše zjistit průměrnou dobu odezvy webového serveru během testu pře celý build a porovnat ho s ostatními.



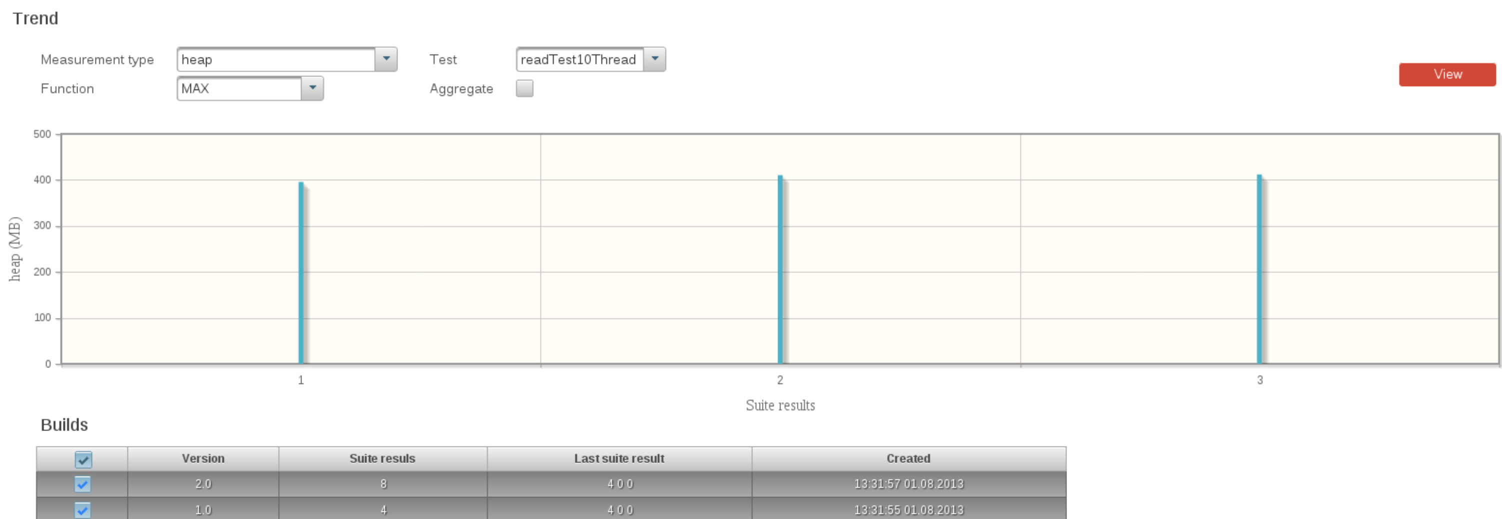

Obrázek 3. Pohled na průběh suite runu




Obrázek 4. Pohled na průměr suite runů z daného buildu




Obrázek 5. Srovnání suite runů mezi sebou




Obrázek 6. Pohled na statistické vlastnosti suite runů

# 7. Výkonnostní testy

Z vlastností repozitáře vyplývá, že jeho výkonnost bude velice závislá na rychlosti vkládání naměřených hodnot. Uvažujme, že je repozitář nasazen v produkčním módu a uchovává data desítek různých projektů. Každý projekt pravidelně spouští testy, jejichž výsledky jsou okamžitě importovány do repozitáře. Největší váhu co se výkonnosti týče bude mít rychlost jednoho importovaného záznamu. REST api repozitáře dále podporuje možnost importovat naměřené hodnoty po kusech, což znamená, že nás bude určitě zajímat vliv rychlosti zpracování požadavku v závislosti na velikosti importovaných dat. Na druhou stranu nesmíme zapomenout na klienty repozitáře, kteří data analyzují a zpracovávají. Bude nás tedy zajímat rychlost čtení dat v závislosti na počtu uživatelů, kteří s repozitařem pracují.

Součásti aplikace je modul speciálně určený pro výkonnostní testování repozitáře. Jsou zde napsány testy pro testování zátěže jak pro čtení tak pro zápis. Následuje popis a charakteristika jednotlivých testů.

**RestReadPerformanceTest**

Jak už název tohoto testu napovídá, tento test se snaží repozitář zatížit čtením jednotlivých REST zdrojů. Test se skládá z několika částí, kde v každé části se server zatěžuje více vlákny (1, 5, 10 a 20). Tento test spočívá ve čtení všech REST zdrojů v náhodném pořadí tak, aby se zabránilo nacachování dotazů. Největší roli u tohoto testu hraje závislost doby zpracování požadavku na počtu uživatelů. Nesmíme opomenout ani vytížení procesorů a využití paměti, což by výrazně zabránilo škálovatelnosti.

**RestWritePerformanceTest**

Cílem tohoto testu je co nejvíce repozitář zatížit zapisováním. Zapisováním mám na mysli vytváření nových zdrojů a vkládání naměřených hodnot. Jelikož tenhle typ zátěže bude nejčastější, je důležité optimalizovat schopnost repozitáře pojmout co nejvíce zápisu za jednotku času. Test se opět skládá z několika částí, ve kterých se repozitář zatěžuje z více vláken.

**RestReadWritePerformanceTest**

Tento test se snaží zatížit repozitár jak čtením, tak i zapisováním zároveň. Ačkoliv se může tento test zdát zbytečný, víceméně simuluje reálnou činnost, když se do repozitáře zapisuje a čte zároveň. U tohoto testu nemůžeme jednoznačně určit jakou veličinu chceme minimalizovat, protože jednotlivé

požadavky jsou vybrány náhodně. Tento test je spíše informativní a chceme znát přibližně kolik uživatelů repozitář zvládne.

**RestImportPerformanceTest**

Nakonec zřejmě nejdůležitějším testem je importování naměřených hodnot po částech. Zajímá nás u tohohle testu tedy jak dlouho bude trvat import dat o určité velikosti a jak bude tato doba na velikosti závislá. Test je opět rozdělen na několik částí, ve kterých se do repozitáře importuji data o různých velikostech (50 KB, 100 KB, 200 KB, 1 MB, 5 MB).

## 7.1. Výsledky

V této části bych uvedl výsledky testů pro zápis a čtení, protože přímo na těchto operacích výkonnost repozitáře závisí. Na začátek bych ještě uvedl poznámky o prostředí, ve kterém byly testy spuštěny.

**Aplikační server** Jboss AS 7.1.1 Final

**Databáze** H2 in memory databáze

**Počet opakování** 100

| Velikost dat | Délka zpracování požadavku | Stav cpu | Využití paměti |
|---|---|---|---|
| 50 KB | 62 ms | 16 % | 187 MB |
| 100 KB | 120 ms | 14 % | 154 MB |
| 200 KB | 349 ms | 21 % | 160 MB |
| 1 MB | 1967 ms | 13 % | 183 MB |
| 1 MB | 27000 ms | 14 % | 187 MB |

Tabulka 1. Výsledky testu na zápis

| Počet vláken | Délka zpracování požadavku | Stav cpu | Využití paměti |
|---|---|---|---|
| 1 | 10 ms | 11 % | 275 MB |
| 5 | 20 ms | 20 % | 257 MB |
| 10 | 40 ms | 21 % | 305 MB |
| 20 | 79 ms | 20 % | 261 MB |

Tabulka 2. Výsledky testu na čtení

## 7.2. Shrnutí

Při pohledu na výše zmíněné výsledky jsou zřejmé následující věci.

- Doba čtení je celkem rychlá (10 ms) a jak se dalo čekat lineárně závisí na počtu vláken, protože byly testy prováděny na jednom serveru. Vytížení procesoru se drží průměrně na 20 %, což při plném zatížení také není špatné. Velice důležitým parametrem je nárůst paměti s počtem uživatelů. Mnoho aplikací, které si ukládají různé informace do HTTP sessiony mají s tímto parametrem veliký problém, který způsobuje nemožnost aplikaci dobře škálovat. Díky tomu, že je repozitář založen na REST architektuře, která nedrží žádný stav mezi libovolnými dvěma požadavky je tento problém vyřešen. I kdyby repozitář podlehl obrovskému nárůstu uživatelů, stav paměti by to neovlivnilo. V ideálním stavu, kdybychom měli k dispozici velké množství serverů zapojených do clusteru, můžeme dosáhnout výše zmíněné odezvy 10 ms.

- U zápisu jsme na tom podstatně hůř. Na první pohled je zřejmé, že s vyšší velikostí importovaných dat roste doba zpracování nelineárně. Je to způsobeno více faktory. Za prvé při větší velikost dat roste doba uploadování souboru na server. Dále bude určitě delší doba zpracování dat do datového modelu repozitáře. Bohužel faktorem, který nejvíce ovlivňuje celkovou dobu zpracování požadavku je databáze. Pro dosažení lepších výsledků by se musely provést další optimalizace na úrovni databáze. Určitě by pomohlo mít k dispozici databázový cluster, který by zápis rovnoměrně rozprostřel mezi databázové servery. Dále by mohlo stát za to integrovat repozitář na některou z NoSQL databází, které jsou navržené speciálně pro problémy s častým zápisem. Stav procesoru a paměti je jako u čtení nezávislý na velikosti importovaných dat, což je určitě příznivé pro škálovatelnost aplikace.

# 8. Možnosti rozšíření

V poslední části této práce se věnuji možnými rozšířeními repozitáře. Napadlo mě velké množství rozšíření, které by se v budoucnu dala implementovat. V následujícím textu uvedu ty nejvýznamnější. Velká část těchto rozšíření nemá významný dopad na funkčnost repozitáře, ale snaží se zvýšit produktivitu uživatelům. V případě rozšířenějšího použití repozitáře by bylo určitě dobré některá z rozšíření dodělat.

### Vykreslování grafů

Jednou ze základních funkcí je vykreslování grafů uživateli. Způsob jakým jsou uživateli data zobrazeny velice výrazně ovlivňují efektivnost jeho práce. V současné době je zobrazování grafů implementováno pomocí komponent, které nabízí knihovna Primefaces. Jako uživatel repozitáře bych určitě uvítal další možnosti jako jsou:

- přibližování/oddalování částí grafů a automatické dotahování dat, které jsou vidět
- skinování (změna vzhledu, velikosti, ...)
- export dat do různých formátů (PNG, CSV, ...)

K implementaci těchto požadavků by se musela použít nějaká Javascriptová knihovna, například Highcharts JS [21].

### Vytvoření nových pohledů na data

V repozitáři jsou implementovány čtyři základní pohledy na výsledky testů. Určitě se v budoucnu vyskytne potřeba vytvoření nových pohledů. API je navrhnuto tak aby se tyto pohledy daly vytvořit bez zásahu do kódu serveru. Někteří uživatelé mohou být odrazení tím, že si pohledy musí doimplementovat v klientovi. Napadlo mě rozšíření podobné tomu co jsem viděl u softwaru [22] JIRA. Stejný problém řeší pomocí widgetů. Widget je nějaká obecná webová komponenta (například tabulka), která definuje možnosti jak s ní pracovat pomocí API (třízení, selekce, filtrování atd.). V aplikaci se pak vyskytuje obrovské množství konkrétních widgetů (seznam projektů, poslední suite run v projektu atd.) a uživatel si přímo může jejich chování přizpůsobit a uložit na serveru. Nevýhodou tohoto způsobu je velká časová náročnost na implementaci. Musela by se vytvořit velká knihovna základních widgetů, napojit se na REST API a vytvořit bohaté uživatelské rozhraní, které by umožňovalo widgety přidávat a přizpůsobovat potřebám uživatelů.

### Integrace s nejrozšířenějšími platformami

Základní otázku, jenž si budou uživatelé před použitím repozitáře pokládat, bude, jak dovnitř nahrnout data. Ačkoliv repozitář podporuje základní formáty a nabízí jednoduchý způsob registrování nových formátů, může to některé uživatele odradit. Chtělo by prozkoumat nejrozšířenější platformy a doimplementovat celou řadu nových formátů tak, aby se práce uživatelům odlehčila. Podobný problém nastává, pokud uživatelé používají monitorovací nástroj, který data neumí exportovat. V této situaci bude muset uživatel do své aplikace dopsat napojení na REST API repozitáře. Klientský modul obsahuje monitorovací nástroj založený na protokolu JMX. Pokud uživatel pracuje na platformě Java, jediné co musí udělat je, aby si tento modul zaregistroval a předal mu JXM URL. Během testů se modul automaticky připojí pomocí JMX k danému virtuálnímu stroji a naměřené hodnoty bude rovnou odesílat do repozitáře. Tento přístup by byl určitě vhodný i pro ostatní platformy jako jsou například .NET nebo Node.js.

### Integrace s verzovacími systémy

Tohle rozšíření se snaží usnadnit uživateli zjištění zdroje výkonnostních chyb. Pokud uživatel zjistí, že v některé verzi projektu došlo k výraznému zhoršení výkonu, bude chtít zjistit příčinu. Podle verze projektu a času jednotlivých testů bude ve verzovacích systémech pátrat po zdroji chyb. Mělo by smysl repozitář integrovat s nejrozšířenějšími verzovacími systémy jakou jsou Git a Subversion a mít podporu hledání změn přímo v repozitáři.

### Notifikace

Dalším užitečným rozšířením můžou být notifikace. Uživatelé by si mohli nastavit emailová upozornění na různé události, které se v repozitáři vykonávají. Příkladem takové notifikace může být zaslání emailu vedoucímu projektu při neúspěšném testu, při zhoršení výkonu nebo při překročení určité hranice využití zdroje. Při častém spouštění výkonnostních testů by uživatelé byli schopni řešit tyto problémy dříve a předejít tak dlouhému hledání zdroje chyb.

### Administrace a zabezpečení

V současné verzi je obecně REST API repozitáře nezapezpečeno. Očekává se, že uživatelé budou používat repozitář uvnitř sítě nebo si zabezpečení zajistí sami. Nebylo by špatné implementovat do repozitáře zabezpečení pomocí protokolu [23] OAuth 2.0. Má smysl uvažovat i oddělené zabezpečení na straně klienta pomocí rolí. V aktuální verzi webový klient definuje dvě základní role,

z nichž jedna nemá právo na zápis. Bylo by vhodné tento systém rolí rozšířit o další role a přidat administraci uživatelů. Například role vedoucí projektu by viděl veškeré informace o daném projektu a spravovala by práva svému týmu. Výkonnostní analytik by naopak viděl jen výsledky testů a neměl by právo na zápis.

### Výkon v clusterových prostředích

Repozitář byl od začátku navržen tak, aby byl bez větších problémů nasazen na cluster serverů. Bohužel jsem neměl takovýto serverový cluster k dispozici, takže jsem nemohl provést kompletní testování v těchto podmínkách. Určitě by před nasazením do takového prostředí chtělo analyzovat chování repozitáře v těchto podmínkách a zjistit, zda je opravdu škálovatelný na všech vrstvách.

### Uživatelsky definované seskupování testů

Posledním rozšířením, které v této části uvedu je seskupování testů a definování vlastních filtrů. Toto rozšíření je především užitečné pro analytiky. Takovýto uživatel by měl mít možnost si definovat vlastní pohledy na data podle toho jak s nimi chce pracovat. Příkladem takového pohledu může být filtr na suite runy za poslední týden. Uživatel by si mohl vytvořit vlastní skupinu, do které by si přidal jednotlivé testy, o které má zájem a ty analyzoval v izolaci. Možností je opravdu hodně a určitě by chtělo tohle rozšíření implementovat pomocí widgetů jako u softwaru JIRA.

# Závěr

Při zpracování této diplomové práce jsem se seznámil s různými způsoby a nástroji, které umožňují testovat software. Tato analýza mi podstatně pomohla pochopit podstatu testování a potřebu vyvinout aplikaci, která vytvoří jednotný pohled nad výstupy těchto testů. Výsledkem práce je snadno rozšiřitelný repozitář, který lze použít na libovolné platformě. Jeho API je navrženo tak, aby ho uživatelé byli schopni jednoduše a efektivně využívat bez potřeby implementace nebo použití nějakého dalšího softwaru. Velká výhoda oproti ostatním nástrojům je sjednocení všech potřebných funkcí (monitorování, archivace a analýza) do jednoho nástroje. Tento repozitář je navržen jako opensource framework, který je možno libovolně využívat a rozšiřovat. Výkonnostní testy, které měly otestovat, zda je repozitář možné používat v praxi byly úspěšné. Již delší dobu pracuji ve firmě, která vyvíjí webové aplikace, takže mi tohle téma bylo velice blízké a chtěl jsem ho poznat detailněji. I když se u nás výkonnostní testy skoro nepíší, po zpracování této práce se budu snažit repozitář u nás nasadit a snažit se ho i nadále rozšiřovat.

# Conclusions

In this thesis, i learn different ways and tools, which can test a software. This analysis helps me understand testing philosophy and need to develop application, which can unify view on result of these tests. The result of this thesis is an extensible repository, which can be used on every platform. Its API is designed, so users can use it simply and effectively withou any external software. A big advantage agains existing tools is that this application unify of all functions (monitoring, archiving and analysis). This repository is designed as opensource framework, which can be used and extended. Performance test, which should test, whether is repository usable in production mode was successful. I am working in company which develop web applications for a long time, so this topic is very close to me. Although there are no performance tests in our company, after developing this application I will try to use this repository in our company and still work on it.

# Reference

[1] Gelperin, Dave - Hetzel, C. William *Software testing*

[2] Graphite http://graphite.readthedocs.org/

[3] Java server faces technology http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

[4] Primefaces http://www.jboss.org/richfaces

[5] Richfaces http://primefaces.org/

[6] Jboss Seam http://www.seamframework.org/

[7] Java persistence API specification http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[8] Hibernate http://www.hibernate.org/

[9] Bean Validation specification http://beanvalidation.org/

[10] Enteprise JavaBeans technology - http://www.oracle.com/technetwork/java/javaee/ejb/index.html

[11] Brian Mulloy, Apigee *Web API Design - Crafting Interfaces that Developers Love*

[12] Brian Mulloy, Apigee *API Facade Pattern - A simple Inteface to a Complex System*

[13] Maven building tool http://maven.apache.org/

[14] Jboss application server https://www.jboss.org/jbossas/

[15] Oracle glassfish application server http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html

[16] Arquillian http://arquillian.org/

[17] JUnit http://junit.org/

[18] TestNG http://testng.org/doc/index.html

[19] Apache JMeter http://jmeter.apache.org/

[20] Apigee http://apigee.com/about/

[21] Highcharts JS http://www.highcharts.com/

[22] JIRA https://www.atlassian.com/software/jira

[23] OAuth 2.0 protocol http://oauth.net/2/
http://oauth.net/2/

# A. REST API

V této příloze přikládám popis jednotlivých REST zdrojů a operace, které s nimi lze provádět. Tato dokumentace usnadní budoucí implementaci klienta a dá přehled o tom co rozhraní podporuje. Jak jsem již uvedl, REST architektura je postavená nad HTTP protokolem, tudíž lze se zdroji pracovat pomoci standardních HTTP metod jako jsou GET, POST a DELETE. Níže je popsán seznam zdrojů, které se vyskytují v aplikaci. Ke každému zdroji je uveden seznam operací, které lze nad zdroji provádět včetně jejich popisu.

### Project

`GET /projects - vrátí všechny projekty`

`GET /projects/{id} - vrátí daný projekt`

`DELETE /projects/{id} - smaže daný projekt`

`POST /projects - vytvoří nový projekt`

### Build

`GET /builds - vrátí všechny buildy`

`GET /builds/{id} - vrátí daný build`

`DELETE /builds/{id} - smaže daný build`

`GET /projects/:id/builds - vrátí všechny buildy daného projektu`

`POST /projects/:id/builds - vytvoří nový build do projektu`

`DELETE /projects/:id/builds - smaže všechny buildy v projektu`

### Suite

`GET - vrátí všechny suity`

`GET /suites/{id} - vrátí danou suitu`

`GET /projects/{id}/suites - vrátí všechny suity z daného projektu`

`GET /tests/{id}/suites - vrátí všechny suity, ve kterých je daný test`

### Test

```
GET /tests - vrátí všechny testy
GET /tests/{id} - vrátí daný test
GET /projects/{id}/tests - vrátí všechny testy v daném projektu
GET /suites/{id}/tests - vrátí všechny testy v dané suitě
```

### SuiteResult

```
GET /suite-results - vrátí všechny výsledky suit
GET /suite-results/{id} - vrátí daný výsledek
GET /builds/{id}/suite-results - vrátí všechny výsledky v daném buildu
GET /suites/{id}/suite-results - vrátí všechny výsledky pro danou suitu
DELETE /suite-results/{id} - smaže daný výsledek
DELETE /builds/{id}/suite-results - smaže všechny výsledky v buildu
DELETE /suites/{id}/suite-results - smaže všechny výsledky v suitě
POST /suite-results/import.{format} - importuje nový výsledek v
                                       daném formátu
POST /builds/{id}/import.{format} - importuje nový výsledek do
                                   daného buildu v daném formátu
POST /builds/{id}/suite-results - vytvoří nový výsledek do daného
                                     buildu
```

### TestResult

```
GET /test-results - vrátí všechny výsledky testů
GET /test-ressults/{id} - vrátí daný výsledek
GET /tests/{id}/test-results - vrátí všechny výsledky daného testu
GET /suite-results/{id}/test-results - vrátí všechny výsledky
                                               daného suite runu
POST /suite-results/{id}/test-results - vytvoří nový výsledek testu do
                                                daného suite runu
```

### Measurement

```
GET /measurements - vrátí všechny měření

GET /measurements/compute - spočítá statistickou funkci nad měřeními

GET /suite-results/{id}/measurements - vrátí měření v daném suite runu

POST /suite-results/{id}/measurements/import.{format} - importuje měření
                                        do daného suite runu v daném formátu

POST /suite-results/{id}/measurements - vytvoří nové měření v daném
                                        suite runu
```

### MeasurementType

```
GET /measurement-types - vrátí všechny použité veličiny

GET /measurement-types/{id} - vrátí danou veličinu
```

# B. Obsah přiloženého CD

V této příloze je popsán obsah CD, který je přiložen jako součást práce.

`bin/`
: V tomto adresáři se nacházejí přeložené a sestavené archivy, které je možné ihned nahrát na libovolný aplikační server.

`src/`
: Tento adresář obsahuje veškeré zdrojové kódy aplikace tak, aby z nich bylo možné sestavit archivy kompatibilní s aplikačními servery. Aplikace je rozdělena do několika modulů, které jsou spravovány pomocí mavenu.

`doc/`
: Zde jsou přiloženy veškeré soubory nutné pro vygenerování textové verze práce pomocí systému latex.

`data/`
: Pro potřeby ukázky jsou zde obsažena testovací data, která je možná nahrát do repozitáře.

`install/`
: Tento adresář obsahuje programy, které jsou nutné pro spuštění repozitáře. Jedná se o aplikační server Jboss AS 7.1.1 a program pro správu projektu maven.

`readme.txt`
: Zde jsou popsány jednotlivé kroky, dle kterých lze repozitář spustit na aplikačním serveru Jboss AS 7.1.1 pomocí mavenu.

# C. Instrukce pro spuštění repozitáře

Pro úspěšné spuštění repozitáře je nutné mít na lokálním PC dostupné tyto věci:

- Java EE 6+ JDK a nastavenou systémovou proměnnou `JAVA_HOME`
- Maven 2+ a nastavenou systémovou proměnnou `M2_HOME`
- JBOSS AS 7.x.x a nastavenou systémovou proměnnou `JBOSS_HOME`

### Instalace repozitáře

Po provedení následujícího příkazu se provede kompilace a sestavení výsledných archivů a jejich nakopírování do lokálního repozitáře mavenu.

```
mvn clean install -DskipTests
```

### Spuštění unit a integračních testů

```
mvn test
```

### Spuštění výkonnostních testů

Pro spuštění výkonnostních testů je potřeba nahrát na aplikační server REST modul.

```
sh $JBOSS_HOME/bin/standalone.sh
cd rest-app
mvn jboss-as:deploy
cd ../test
mvn test
```

#### Spuštění repozitáře

```
sh $JBOSS_HOME/bin/standalone.sh
cd rest-app
mvn jboss-as:deploy
cd ../web
mvn jboss-as:deploy
```

- REST API - `http://localhost:8080/rest/v1`
- Web klient - `http://localhost:8080/web`