Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Rozšíření a propojení nástroje UrbanCode Deploy

Plzeň 2018 Patrik Harag

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 29. dubna 2018

Patrik Harag

Poděkování

Tímto bych chtěl poděkovat Ing. Lukáši Holému, Ph.D. za rady a připomínky v průběhu zpracování této práce.

Abstract

This bachelor's thesis deals with the IBM UrbanCode Deploy plugin development. The theoretical part provides introduction to application deployments and describes UrbanCode Deploy deployment automation tool. It also shows UrbanCode Deploy plugins creation. The practical part deals with technical questions related to plugins creation as well as plugin creation for integration with ServiceNow and SAG webMethods Integration Server. It demonstrates deployment automation processes consisting of ServiceNow items creation and webMethods Integration Server package deployments.

Abstrakt

Tato bakalářská práce se zabývá vývojem rozšíření pro IBM UrbanCode Deploy. Teoretická část poskytuje úvod do problematiky nasazování softwarových produktů a popisuje nástroj pro automatizaci nasazování UrbanCode Deploy. Popisuje také tvorbu rozšíření pro UrbanCode Deploy. Praktická část se zabývá technickými záležitostmi souvisejícími s tvorbou rozšíření a dále pak tvorbou rozšíření pro integraci s nástroji ServiceNow a SAG webMethods Integration Server. Demonstruje procesy automatizace nasazování s vytvářením položek v ServiceNow a nasazováním balíčku na webMethods Integration Server.

Obsah

1 Úvod **8**

2 Nasazování softwarových produktů **9**
2.1 DevOps 9
2.1.1 Charakteristika 9
2.1.2 Techniky DevOps 10
2.2 Cílové prostředí 11
2.3 Nástroje pro automatizaci nasazování 12

3 Nástroj UrbanCode Deploy **14**
3.1 Představení 14
3.1.1 Architektura 15
3.1.2 Přehled elementů 15
3.1.3 Parametry 18
3.2 Rozšíření 19
3.3 Vývoj rozšíření 19
3.3.1 Soubor plugin.xml 20
3.3.2 Soubor info.xml 23
3.3.3 Soubor upgrade.xml 24

4 Výběr problému **27**

5 Příprava na vývoj **28**
5.1 Instalace UrbanCode Deploy 28
5.2 Výběr programovacího jazyka 30
5.2.1 Programovací jazyk Groovy 30
5.3 Sestavení rozšíření 32
5.3.1 Rozšíření pro Eclipse IDE 32
5.3.2 Výběr nástroje pro automatizaci sestavování 32
5.3.3 Tvorba skriptu pro automatizaci sestavování 33

6 Rozšíření pro webMethods Integration Server **36**
6.1 WebMethods 36
6.2 Analýza 36
6.2.1 Postup nasazování s webMethods Deployer 36
6.2.2 Popis CLI 37

6.3 Návrh kroků 38
6.4 Implementace 39
6.5 Testování 39

7 Rozšíření pro ServiceNow **42**
7.1 ServiceNow 42
7.2 Analýza 42
7.2.1 Životní cyklus požadavku na změnu 43
7.2.2 Organizace dat v ServiceNow 44
7.2.3 Dostupná rozhraní ServiceNow 44
7.3 Návrh 47
7.3.1 Návrh kroků k realizaci 47
7.3.2 Komunikace se serverem ServiceNow 48
7.4 Implementace 48
7.5 Testování 50
7.5.1 Testování klienta 50
7.5.2 Testování rozšíření 51

8 Závěr **54**

Přehled zkratek **55**

Literatura **56**

Přílohy **57**

1 Úvod

Trendem ve vývoji softwaru je snaha o zkrácení vývojového cyklu produktu. Kratší vývojový cyklus umožňuje rychleji získat zpětnou vazbu na vyvíjený produkt, což umožní cíleně usměrňovat jeho vývoj a minimalizovat tak ekonomická rizika. Jednou z částí vývojového cyklu, kterou se můžeme pokusit zkrátit je nasazování – proces, při kterém se nová verze produktu dostává do rukou uživatelů. Zkrácení spočívá v automatizaci, při které by mělo dojít k odstranění co možná největšího počtu manuálních kroků. Poměrně nový koncept, který se v této oblasti prosazuje se nazývá DevOps.

Jedním z nástrojů, které nám automatizaci nasazování zjednoduší je IBM UrbanCode Deploy. Umožňuje vizuálně modelovat komplexní procesy nasazování zahrnující integraci s dalšími službami a nástroji. Podporu pro další nástroje je možné přidat pomocí rozšíření. Výrobce UrbanCode Deploy poskytuje rozšíření pro mnoho často používaných nástrojů, ale pochopitelně nemůže vyhovět všem požadavkům, a tak dává k dispozici rozhraní, které mohou uživatelé využít pro tvorbu vlastních rozšíření.

Cílem této práce je poskytnout úvod do problematiky nasazování softwarových produktů, seznámit čtenáře s nástrojem UrbanCode Deploy, popsat způsob tvorby rozšíření pro tento nástroj, identifikovat nástroje, pro které by bylo užitečné vytvořit rozšíření a vybraná rozšíření implementovat a otestovat.

2 Nasazování softwarových produktů

2.1 DevOps

2.1.1 Charakteristika

Každá větší společnost zabývající se vývojem software rozlišuje oddělení vývoje a oddělení pro podporu provozu.

Oddělení vývoje (Nebo jen *vývoj*, anglicky *development*.) Tvoří jej vývojáři, jejichž úkolem je vyvinout produkt a následně ho udržovat. Každou novou verzi předává oddělení pro podporu provozu k nasazení na produkci.

Oddělení pro podporu provozu (Nebo jen *provoz*, někdy *servis*, anglicky *IT operations*.) Je zodpovědné za hladké fungování infrastruktury. To zahrnuje především správu počítačových sítí a serverů, často i poskytování technické podpory.

Toto striktní rozdělení ovšem typicky přináší následující problémy:

- Potřebuje-li *vývoj* nečekaně nasadit novou verzi produktu (například kvůli opravě bezpečnostní chyby), nemusí se ze strany servisu dočkat okamžité reakce. Nasazení nové verze tak může být zbytečně odloženo.
- *Provoz* nemá hlubší znalosti o vnitřním fungování software, který provozuje a pokud dojde k netriviálnímu problému, pravděpodobně jej nedokáže vyřešit. Následné hledání znalé osoby a komunikace může stát mnoho času.
- Technologie používané při vývoji se často liší od technologií používaných v produkčním prostředí. Vývojáři mohou například vyvíjet na OS Windows a produkt je poté nasazen na OS Linux. Pravděpodobně tak bude lépe odladěn na jiném než produkčním prostředí.
- *Vývoj* musí detailně popsat veškeré závislosti a požadavky na běhové prostředí, včetně podrobných instrukcí, jak produkt nasadit. Ty se

navíc mohou často měnit. Kromě vynaložené práce *vývoje*, který nesouvisí s jeho hlavní činností může být dalším problémem případná chybná interpretace či opomenutí ze strany *provozu*.

Podle [4] a [8] však skutečný problém leží mnohem hlouběji – stojí za ním odlišné cíle těchto oddělení. *Vývoj* se snaží rychle reagovat na změny na trhu, vydávat nové verze a co nejrychleji je dostat na produkci. *Provoz* na sebe bere zodpovědnost za poskytování služeb, které jsou stabilní, spolehlivé a bezpečné. Přirozeně se tak brání nasazování nových verzí, které by mohly ohrozit produkci.

DevOps, se snaží všechny tyto problémy řešit tím, že zmenšuje pomyslnou mezeru mezi těmito odděleními. Rivalitu se snaží proměnit ve spolupráci. Samotné slovo DevOps je složené z anglických slov *development* a *operations*. Netýká se však pouze poměrů mezi *vývojem* a *provozem*, i když v tom je jeho jádro. Jedná se o koncept, který zasahuje do celého vývoje, od návrhu, přes testování, až po monitoring. Nesmíme proto zapomenout ani na QA oddělení a získání podpory u managementu. [4] [8]

2.1.2 Techniky DevOps

DevOps využívá celou řadu dobře známých technik, ale zavádí i některé zcela nové. Dále jsou popsány ty nejdůležitější z nich.

Continuous integration Je metoda vývoje software, kde každý člen týmu integruje svoji část práce průběžně, obvykle alespoň jednou za den. Každá integrace je ověřena automatickým sestavením, po kterém následují automatické testy, které včas odhalí případné problémy s integrací. [2]

Prakticky to znamená, že si vývojář stáhne aktuální zdrojové kódy ze sdíleného úložiště. Zatímco pracuje, stav úložiště se může měnit. Po dokončení své práce zkontroluje stav úložiště, ihned vyřeší všechny kolize a otestuje funkčnost. Poté mohou být provedené změny odeslány na úložiště.

Continuous delivery Je schopnost dodat změny všeho druhu (nové vlastnosti, změny v konfiguraci, opravy chyb) do produkce nebo do rukou uživatelů, a to bezpečně, rychle a udržitelným způsobem. [6]

Klíčové je vytvoření spolehlivého automatizovaného procesu pro nasazení produktu. Nasazení by mělo být maximálně jednoduché a nemělo by představovat mimořádnou událost. [5]

Za extrémní formu *Continuous delivery* lze považovat *Continuous deployment* [5]. Při tomto přístupu jsou veškeré změny, které projdou automatickými testy rovnou nasazeny na produkčním prostředí. Od odeslání změn na

sdílené úložiště jsou tedy veškeré další kroky automatizované. Tento přístup však s sebou přináší určitá rizika a nemusí být vždy vhodný. Ne vždy je například možné zcela vynechat manuální testy.

Continuous testing Continuous testing znamená začít s testováním dříve a testovat průběžně napříč vývojovým cyklem. Získaná průběžná zpětná vazba umožní cílenější usměrňování vývoje. [11]

Continuous monitoring V produkci *provoz* monitoruje a zajišťuje správné fungování aplikace. Na rozdíl od běžného monitoringu, který sleduje pouze systém, jsou monitorovány navíc také samotné aplikace. Díky spolupráci s *vývojem* může být s monitoringem počítáno již od samotného návrhu aplikace, a přímo do aplikace tak mohou být zabudovány nástroje pro její vlastní monitorování. [11]

Infrastructure as code Je paradigma, při kterém se na infrastrukturu hledí stejně, jako by se jednalo o zdrojový kód programu [1]. To umožňuje využít nástroje a techniky běžné pro vývoj software – například verzování, automatické testy a *Continuous delivery* [1]. *Infrastructure as code* se prosazuje zejména ve spojení s cloudovými technologiemi.

2.2 Cílové prostředí

Cílové prostředí je místo, kam nasazujeme náš softwarový produkt. V závislosti na charakteru produktu a technologiích, které využívá, máme několik možností, kam jej nasadit.

Fyzický stroj Produkt je nasazován a provozován jako aplikace přímo v operačním systému.

Virtuální stroj Stejně jako u fyzického stroje je produkt nasazován a provozován jako aplikace přímo v operačním systému, s tím rozdílem, že je tento stroj virtualizovaný. Virtualizace přináší řadu výhod, ať už je to vyšší bezpečnost, snadná přenositelnost nebo lepší využití fyzických serverů, díky skutečnosti, že je možné provozovat hned několik virtuálních strojů na jednom fyzickém stroji. V oblasti cloud computingu se můžeme setkat se službami typu *Virtual Private Server* (VPS) a *Infrastructure as a service* (IaaS), pro pronájem virtuálních strojů.

Kontejner Kontejnery vytvářejí prostředí, které je izolované od ostatních aplikací hostitelského systému. Podobně jako virtuální stroje, i kontejnery využívají virtualizaci, ale na rozdíl od nich neobsahují operační systém, jsou vázány na ten hostitelský. Spuštění a provoz kontejneru tedy vyžaduje méně prostředků než celý virtuální stroj, ale přináší obdobné výhody. [3]

Je možné se setkat s termínem *Containers as a Service* (CaaS), kdy poskytovatelé cloudových služeb prodávají výpočetní výkon svých datacenter pro provoz kontejnerů.

Platforma Platformou může být běhové prostředí (například *Java Virtual Machine*), aplikační server, webový server, databáze a podobně. Můžeme se setkat s termínem *Platform as a Service* (PaaS), kdy je uživateli poskytnuta taková platforma připravená k použití.

Serverless Ještě o krok dále v odstiňování od infrastruktury jde bezserverová architektura (*Serverless Architecture*), která se snaží oprostit od standardního serverového paradigmatu. Aplikace se skládá z takzvaných funkcí, které připomínají mikroslužby, jejichž instance mohou s minimální režií vznikat a zanikat podle aktuální zátěže. Zde se jedná o téměř výhradně cloudová řešení, někdy nazývaná *Function as a service* (FaaS). [9]

Rozmach cloud computingu a s tím souvisejících řešení typu IaaS, CaaS, PaaS a jako poslední FaaS přinesly nové možnosti v oblasti nasazování softwarových produktů. Nasazení již nemusí znamenat vysokou počáteční investici do hardware a flexibilní platební modely umožňují dynamicky optimalizovat počet běžících instancí se schopností reagovat na prudké změny v poptávce.

2.3 Nástroje pro automatizaci nasazování

DevOps přináší mnohé výhody, ale jeho zavedením vývojářům v podstatě přidáváme práci navíc, protože kromě vývoje musejí zvládat i některé činnosti, které dříve zastával *servis*. Úspěšnost adopce tedy do značné míry závisí na volbě vhodného nástroje nebo kombinace nástrojů pro automatizaci nasazování. Identifikoval jsem několik tříd nástrojů s rozdílným použitím. Nástroje z různých tříd na sebe mohou být navázány.

Skripty Shell skripty, například Bash (Linux) a PowerShell (Windows), nebo skripty ve vyšším programovacím jazyce. Tento přístup je jednoduchý

a přímý, ale s rostoucím projektem neudržitelný. Mezi nevýhody patří především absence management úrovně. Například přehled o uživatelích, prostředích a spuštění nasazení – tyto informace je třeba uchovávat na jiném místě. Vyžaduje přímý přístup na cílové prostředí.

Nástroje pro automatizaci sestavování Nástroje jako *Make*, *Ant*, *Maven*, *Gradle*, apod. jsou používány především pro správu projektů a automatizaci sestavování, mohou však být použity i na navazující záležitosti.

Na platformě Java je jedním z nejčastějších použití automatizace odeslání sestaveného balíčku do sdíleného Maven úložiště, které lze s dostupnými rozšířeními pro Maven nebo Gradle realizovat velmi snadno. Rozšíření jsou dostupná také pro aplikační servery (*Tomcat*, *WebLogic*, *WebSphere*, *JBoss*,...) a pro většinu významných cloudových poskytovatelů (*Amazon AWS*, *Google Cloud Platform*, *Microsoft Azure*, *Heroku*,...).

Nástroje zaměřené na Continuous integration Existuje několik nástrojů pro zajištění technické stránky *Continuous integration*, jako *Jenkins*, *Travis CI*, *CircleCI* nebo *Hudson*. Jedná se o servery, které sledují systémy pro správu verzí a s každou novou změnou (nebo po určité době) provedou sestavení a spustí automatické testy.

Tyto nástroje obvykle také zvládají integraci s různými aplikačními servery a cloudovými poskytovateli. Pokud tedy projdou testy, je možné produkt rovnou nasadit.

Nástroje zaměřené na modelování nasazování Nástroje umožňující modelování komplexních procesů nasazování, které vyžadují přípravu prostředí a integraci s dalšími službami. Slouží zároveň pro správu nasazených produktů – na jakém prostředí je nasazeno, v jaké verzi atd. Patří mezi ně například *UrbanCode Deploy* a *Jenkins* (*Pipeline*).

Procesy vytvořené v těchto nástrojích mohou stát více času než jednorázové skripty, ale výhodou potom je standardizovanost, robustnost a vyšší míra znovupoužitelnosti takových řešení.

Nástroje pro práci s kontejnery Aplikace zabalená do kontejneru, včetně veškeré konfigurace a závislostí, může být s minimálním úsilím nasazena téměř kdekoliv [3]. V této oblasti jasně dominuje nástroj zvaný *Docker*. Existují také nástroje pro správu a nasazování kontejnerů, jako *Kubernetes* a na něm založené *OpenShift* a *IBM Cloud Private*.

3 Nástroj UrbanCode Deploy

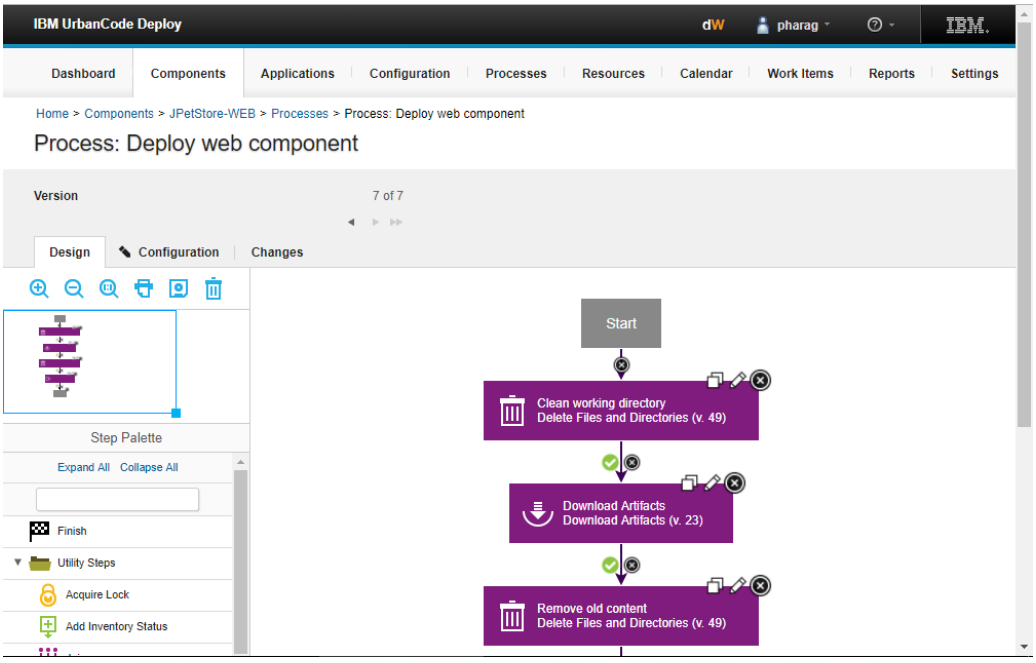
3.1 Představení

IBM UrbanCode Deploy, zkráceně UCD, je nástroj pro automatizaci nasazování aplikací, databází a konfigurací do vývojového, testovacího a produkčního prostředí [7]. Veškerá konfigurace a vytvořené procesy jsou automaticky verzovány. UCD je možné obsluhovat prostřednictvím webového rozhraní (viz obrázek 3.1), *command-line interface* (CLI) nebo pomocí REST API.

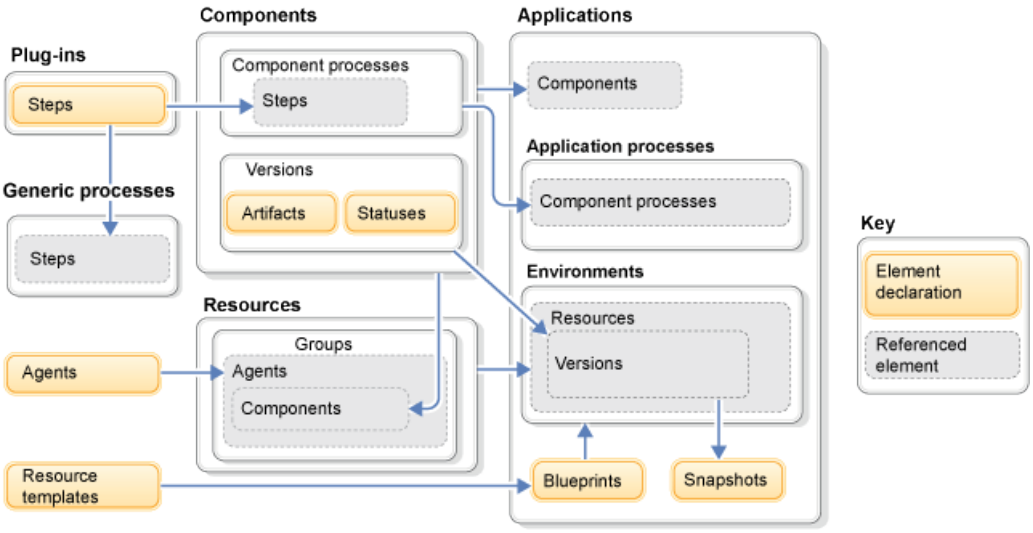
Vyvinula ho společnost UrbanCode Inc., která byla v roce 2013 odkoupena společností IBM Corporation. Vedle UrbanCode Deploy existují také příbuzné nástroje UrbanCode Build a UrbanCode Release, které se specializují na jiné aspekty vývoje software.

UCD obvykle pracuje již se sestavenými programy (*buildy*). Sice by bylo možné za pomoci rozšíření do procesu v UCD zahrnout i sestavení a spuštění testů, ale na to již existují jiné nástroje zaměřené na *Continuous integration*.

Verze, se kterou pracuje tato práce je 6.1.3. Prakticky jediným zdrojem informací o UCD je oficiální dokumentace [7], která byla zároveň použita jako hlavní zdroj pro zpracování této kapitoly.



Obrázek 3.1: Webové rozhraní UCD.



Obrázek 3.2: Závislosti mezi různými typy elementů. Zdroj: [7]

3.1.1 Architektura

Ve své základní podobě se UCD skládá ze serveru, agentů a licenčního serveru. Všechny tyto části mohou běžet na jednom stroji nebo každá na svém.

Server Poskytuje webové rozhraní a REST API. Právě zde jsou spouštěny procesy pro automatizované nasazování aplikací. Obsahuje databázi komponent, procesů a dalších elementů.

Agent Agent je program, který je nainstalován na stroji, ať už virtuálním nebo fyzickém, na který se bude nasazovat. UCD server posílá příkazy a agent je vykonává. UCD potřebuje logicky alespoň jednoho agenta aby bylo možné spustit nějaký proces.

Licenční server Poskytuje licence nejenom UCD serveru, ale i dalším produktům od IBM.

3.1.2 Přehled elementů

UCD obsahuje několik druhů elementů, ze kterých se modelují automatizované procesy nasazování aplikací. Schéma na obrázku 3.2 ukazuje závislosti meze nimi.

Components

Komponenty reprezentují nasaditelné položky, někdy se také nazývají artefakty. Mohou to být aplikace, obrázky, konfigurační soubory nebo cokoliv jiného, podle toho, co projekt vyžaduje.

Zdrojem artefaktů může být v tom nejjednodušším případě adresář, který je umístěn na stejném systému jako UCD server. Dále to může být některý z podporovaných systémů pro správu verzí, jako Git nebo SVN. Podporováno je také několik systémů pro správu *buildů* jako Maven nebo Artifactory.

Applications

Aplikace je logické seskupení komponent, které jsou nasazovány společně.

Resources

Prostředek je logická položka, která určuje kam se bude nasazovat (cílové prostředí). Prostředek může být agent, skupina agentů (*agent pool*), komponenta nebo skupina prostředků. Prostředky mohou vytvářet hierarchie. V nejjednodušší konfiguraci může prostředek odkazovat na fyzický server. Může také odkazovat na specifický cíl uvnitř systému, například na aplikační server nebo na databázi.

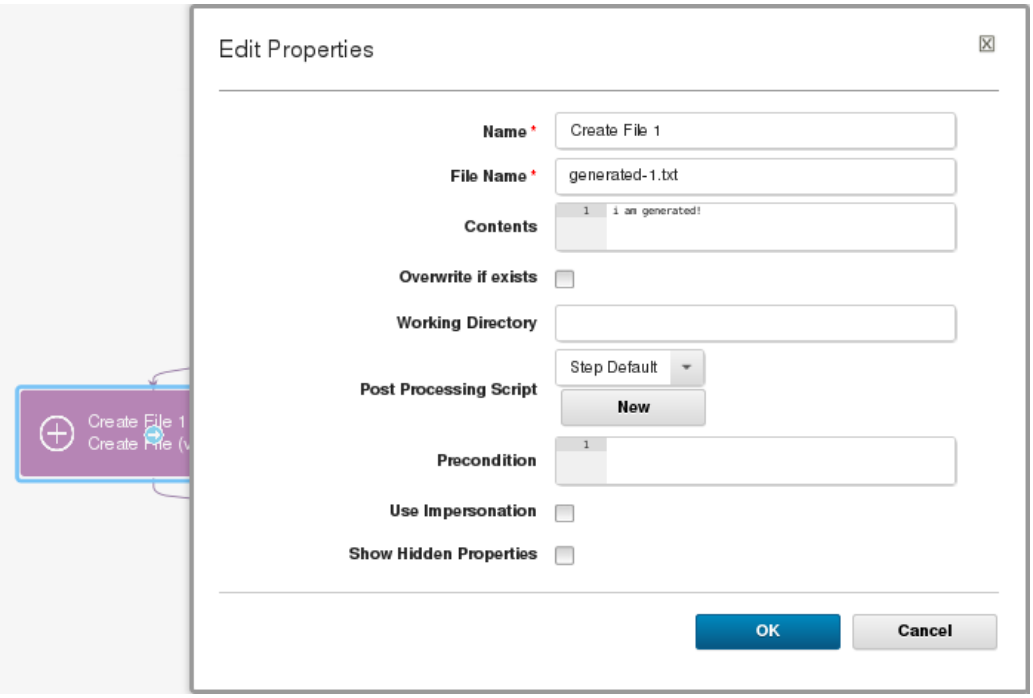
Environments

Prostředí je množina prostředků, na které je nasazena určitá aplikace. Jedna aplikace může mít více prostředí. Můžeme mít například aplikaci X, která je nasazena v prostředí *PRODUCTION* ve verzi n, v prostředí *UAT* ve verzi $n + 1$ a v prostředí *TEST* ve verzi $n + 2$.

Steps

Krok představuje příkaz, který bude spuštěn na cílovém stroji (agentovi). Kroky mohou například řídit tok procesu (podmínky), manipulovat se soubory, spouštět příkazy příkazové řádky nebo volat vzdálené služby. Každý krok má určitou množinu vstupních parametrů, které mohou nebo musejí být při jeho použití nastaveny. Další kroky, které umožní integraci s dalšími službami je možné přidat prostřednictvím rozšíření (*plug-ins*). Vývojem těchto rozšíření se zabývají další kapitoly této práce.

Na obr. 3.3 je ukázán příklad dialogového okna pro úpravu kroku ve webovém rozhraní UCD. První čtyři parametry jsou specifické pro tento krok, zbytek je pro všechny kroky shodný.



Obrázek 3.3: Dialogové okno s úpravou kroku

Component processes

Komponentový proces je posloupnost kroků, která pracuje s artefakty jedné komponenty. Jedna komponenta může mít více procesů. Typicky se jedná o úkony typu instalace, odinstalace nebo konfigurace. Ve webovém rozhraní se procesy modelují metodou *táhni a pusť*. Příklad je vidět na obr. 3.1.

Generic processes

Generické procesy se velmi podobají komponentovým procesům, ale nevztahují se k žádné konkrétní komponentě nebo aplikaci. Mohou v nich být definované obecné administrační úkony, jako příprava serveru.

Application processes

Aplikační procesy se na první pohled podobají komponentovým a generickým procesům, ale pracují na vyšší úrovni. Mají k dispozici jinou sadu kroků, která se skládá mimo jiné z *Install Component* a *Uninstall Component*. Definují způsob, jakým se má aplikace složená z komponent, nainstalovat nebo odinstalovat. Velmi často, když není potřeba žádná další logika, může takový proces vypadat jako: *Start* → *Install Component A* → *Install Component B* → *Finish*.

3.1.3 Parametry

Parametry jsou proměnné, které jsou buď globálního charakteru, nebo se váží k elementům, jako komponenta, prostředí, aplikace nebo proces. Tabulka 3.1 obsahuje přehled rámců.

Mohou být definovány uživatelem, který je může využít při vytváření kroků, kde je možné do vstupních polí nastavit, místo napevno zadané hodnoty, referenci na parametr. Existuje také několik výchozích parametrů, které poskytuje systém.

Referencování parametrů

Parametry mohou být kdekoliv v textu referencovány s rámcem i bez něj:

- `${p:scope/name}`
- `${p:name}`

V případě, že parametr neexistuje dojde k běhové chybě. Přidáním otazníku můžeme změnit toto chování. Výraz poté vrátí prázdný řetězec, pokud uvedený parametr neexistuje.

- `${p?:scope/name}`
- `${p?:name}`

Výstupní parametry

Speciální úlohu hrají výstupní parametry kroků, s jejichž pomocí je možné předávat data mezi jednotlivými kroky. Záleží na konkrétním kroku, co bude poskytovat jako své výstupní parametry.

Na výstupní parametry se odkazuje pomocí názvu kroku (přesněji názvu konkrétní instance kroku, který je určen uživatelem) a názvu parametru:

- `${p:step_name/name}`

Tabulka 3.1: Přehled rámců.

Rámec	Formát reference
Globální	`${p:system/name}`
Application	`${p:application/name}`
Component	`${p:component/name}`
Environment	`${p:environment/name}`
Resource	`${p:resource/name}`
Process	`${p:name}`, `${p:step_name/name}`

3.2 Rozšíření

Rozšíření (*plug-ins*) v UCD rozšiřují možnosti vytváření komponentových procesů a integrací. Výrobce poskytuje oficiální rozšíření pro celou řadu služeb a nástrojů, a další vytváří komunita uživatelů. Všechna tato rozšíření jsou dostupná na jednom portálu[1].

Typy rozšíření

Existují dva typy rozšíření:

- *Source-type plug-ins* – umožňují při vytváření komponent importovat artefakty z dalších zdrojů.
- *Automation-type plug-ins* – přidávají další kroky, které je možné využít v komponentových procesech. Tato práce se dále zabývá pouze tímto typem rozšíření.

Oba dva typy rozšíření se vyvíjejí velmi podobným způsobem, rozdíly jsou pouze v detailech. Dalo by se říci, že se jedná především o logické členění pro uživatele.

Instalace rozšíření

Instalace rozšíření probíhá za běhu, skrze webové rozhraní UCD. V záložce *Settings* výběrem *Automation Plugins* nebo *Source Config Plugins* a kliknutím na tlačítko *Load Plugin*.

3.3 Vývoj rozšíření

Rozšíření jsou v podstatě jednoduché souboru typu ZIP, které obsahují:

- konfigurační soubory;
  - `plugin.xml` obsahuje základní údaje a definuje kroky;
  - `info.xml` obsahuje různé další informace;
  - `upgrade.xml` poskytuje informace o změnách, aby bylo možné provést automatickou migraci kroků, použitých ve stávajících procesech, na novou verzi;
- skripty vykonávající logiku kroků referencované z `plugin.xml`, případně knihovny a další závislosti.

[1] `https://developer.ibm.com/urbancode/plugins/ibm-urbancode-deploy`

3.3.1 Soubor plugin.xml

Tento soubor především definuje kroky pro použití v komponentových a generických procesech. Dále obsahuje hlavičku se základními informacemi o rozšíření, jako název a popis. Základní strukturu ukazuje výpis 3.1.

Výpis 3.1: Struktura souboru *plugin.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
    xmlns="http://www.urbancode.com/PluginXMLSchema_v1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <!-- Základní informace -->
  <header>
    <identifier id="..." version="..." name="..."/>
    <description>...</description>
    <tag>...</tag>
  </header>

  <!-- Definice kroků -->
  <step-type name="Step 1">
    <description>...<description>
    <properties>
      <property name="input_1" required="true">
        ...
      </property>
      ...
    </properties>
    <command program="...">...</command>
    <post-processing>...</post-processing>
  </step-type>
  ...

</plugin>
```

Základní informace

Informace uváděné v elementu *<header>* slouží k identifikaci rozšíření.

<identifier> obsahuje atributy:

- *id* – jednoznačná identifikace rozšíření; obvykle se používá název balíčku (např. `com.urbancode.air.plugin.helloworld`)
- *version* – aktuální verze rozšíření
- *name* – název

<description> nastavuje popis, tem se zobrazuje například v seznamu nainstalovaných rozšíření.

<tag> udává kategorii, pod kterou se budou zobrazovat kroky v editoru procesů (nabídka po levé straně). Kategorie jsou hierarchické, pro oddělení slouží lomítko.

Definice kroků

Krok je definován elementem *<step-type>*, který v sobě obsahuje následující elementy:

<description> nastavuje popis kroku.

<properties> definuje vstupní parametry kroku. Krok může mít libovolný počet vstupních parametrů, které se zadávají skrze dialog pro úpravu nastavení kroku v návrháři procesů.

Vstupní parametr je definován elementem *<property>* s atributy *name* a *required*. Atribut *name* není zobrazován uživatelům, jedná se o interní název. Atribut *required*, který je volitelný (s výchozí hodnotou *true*), určuje, zda musí mít nastavenou hodnotu.

Uvnitř elementu *<property>* se nachází element *<property-ui>*, který určuje, jakým způsobem se parametr zobrazí v dialogu pro úpravu nastavení kroku. Má atributy *label*, *description*, *default-value* a *type*. Atribut *type* může nabývat následujících hodnot:

- `textBox` – jednořádkové textové pole
- `textAreaBox` – víceřádkové textové pole
- `secureBox` – jednořádkové textové pole se skrytým obsahem
- `checkBox` – zaškrtávací políčko; je-li zaškrtnuto, hodnota je *true*, v opačném případě není hodnota nastavena
- `selectBox` – rozbalovací seznam

Při použití rozbalovacího seznamu je nutné nadefinovat alespoň jednu hodnotu pro výběr. K tomu slouží elementy *<value>* patřící pod element *<property>*. Atributem *label* se nastavuje text, který se zobrazí uživateli a obsah elementu představuje hodnotu, která bude při výběru přiřazena parametru. Hodnoty jsou ve webovém rozhraní zobrazeny v pořadí, v jakém byly definovány. Použití ukazuje výpis 3.2.

Výpis 3.2: Definice parametru s rozbalovacím seznamem

```
<property name="state" required="true">
  <property-ui type="selectBox" label="State"
      default-value="10" description="Select state"/>
  <value label="Draft">10</value>
  <value label="Suspended">15</value>
  <value label="Closed">110</value>
</property>
```

<command> sestavuje příkaz, kterým bude spuštěn program vykonávající logiku daného kroku. Program je obvykle skript v jazyce Groovy, případně jiném skriptovacím jazyce, zabalený v souboru s rozšířením. Může být ale spuštěn zcela libovolný program, který se nachází na stroji s agentem.

Výpis 3.3 ukazuje typické použití. Spuštění skriptu `hello.groovy` se všemi dalšími třídami v jazyce Groovy ve složce `classes` a s knihovnou `groovy-plugin-utils-1.0.jar` (která je ve složce `lib`) na *class path*. Řetězec `${PLUGIN_INPUT_PROPS}` bude nahrazen cestou k souboru se vstupními parametry kroku. Podobně `${PLUGIN_OUTPUT_PROPS}`.

Výpis 3.3: Příklad elementu *command*

```
<command program="${GROOVY_HOME}/bin/groovy">
  <arg value="-cp"/>
  <arg path="classes:lib/groovy-plugin-utils-1.0.jar"/>
  <arg file="hello.groovy"/>
  <arg file="${PLUGIN_INPUT_PROPS}"/>
  <arg file="${PLUGIN_OUTPUT_PROPS}"/>
</command>
```

<post-processing> obsahuje skript v jazyce JavaScript verze 1.7, který je spouštěn po skončení vlastní činnosti rozšíření. Jeho úkolem je podle návratového kódu a výstupu programu nastavit výstupní parametry.

Skript má přístup k následujícím objektům:

- `properties` je instance typu `java.util.Properties` do které jsou ukládány výstupní parametry. Ve výchozím stavu již nějaké parametry obsahuje, jedním z nich je `exitCode` obsahující návratový kód programu. Bude také obsahovat výstupní parametry nastavené v rámci programu.
- `commandOut` je instance typu `java.io.PrintStream`, kterou je možné použít pro logování.

- `scanner` je objekt, který umožňuje prohledávat výstup programu. Má metodu `register(String regex, Function callback)`, která přebírá regulární výraz a funkci, která je volána při nalezení řádky. A dále metodu `scan()`, která zahájí prohledávání.

Skript musí minimálně nastavit výstupní parametr `Status`, a tedy rozhodnout, zda bylo vykonání kroku úspěšné. Ukázka viz výpis 3.4.

Výpis 3.4: Příklad skriptu

```
<post-processing>
  <![CDATA[
    if (properties.get("exitCode") != 0)
        properties.put("Status", "Failure");
    else
        properties.put("Status", "Success");
  ]]>
</post-processing>
```

3.3.2 Soubor info.xml

Obsahuje různé další informace o rozšíření – autor, verze, popis změn k jednotlivým verzím, atd. Základní strukturu souboru ukazuje výpis 3.5.

Povinné elementy

<author> s atributem *name* nastavuje autora rozšíření. Volitelně může dále obsahovat elementy *<organization>*, *<email>*, *<website>* a *<bio>* s textovým obsahem.

<integration> atributem *type* určuje obecný typ rozšíření pro jeho kategorizaci. Hodnota *Source* je nastavována, pokud se jedná o *source-type* rozšíření (viz sekce 3.2), *Artifact* pro rozšíření, které slouží pro získávání artefaktů z nějakého úložiště během nasazování, *Deploy* pokud nasazuje program na middleware server a *Automation* pro ostatní.

<release-notes> obsahuje pro každou verzi element *<release-note>* s atributem *plugin-version* a popisem změn v dané verzi.

Výpis 3.5: Struktura souboru *info.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<pluginInfo
    xmlns="http://www.urbancode.com/InfoXMLSchema_v1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
  <!-- Jméno autora, volitelně další informace o něm -->
  <author name="John Doe">...</author>

  <!-- Specifikace typu rozšíření -->
  <integration type="Deploy"/>

  <!-- Popis změn pro každou verzi -->
  <release-notes>
    <release-note plugin-version="1">...</release-note>
    ...
  </release-notes>

  <!-- Následují nepovinné položky -->

  <!-- Verze sestavení -->
  <release-version>2</release-version>

  <!-- Popis nástroje, se kterým toto rozšíření pracuje -->
  <tool-description>...</tool-description>

  <!-- Licence -->
  <licenses>
    <license type="MIT"/>
  </licenses>
</pluginInfo>
```

3.3.3 Soubor upgrade.xml

Poskytuje informace o změnách, aby bylo možné provést automatickou migraci kroků při načtení nové verze rozšíření. Základní struktura viz výpis 3.6.

Výpis 3.6: Základní struktura souboru *upgrade.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin-upgrade
    xmlns="http://www.urbancode.com/UpgradeXMLSchema_v1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <migrate to-version="2">
    <migrate-command name="Step Name">
      ...
    </migrate-command>
    ...
  </migrate>
  ...
</plugin-upgrade>
```

Při každém vydávání nové verze je vytvářen element *<migrate>* s atributem *to-version*, který má hodnotu rovnu číslu nové verze.

Propagace kroku do další verze

Autoři UCD v dokumentaci [7] doporučují, aby byl pro každou novou verzi vytvořen element *<migrate-command>* pro každý krok, i pokud u něj nedošlo k žádné změně. Prakticky to ovšem není nutné dělat, k propagaci do další verze dojde i bez něj. Ukázka viz výpis 3.7.

Výpis 3.7: Migrace kroku do další verze

```
<migrate to-version="2">
  <migrate-command name="Step Name"/>
</migrate>
```

Odstranění kroku

Informace o odstranění kroku se v *upgrade.xml* neuvádí.

Přidání kroku

V souboru *upgrade.xml* se nic neuvádí.

Přejmenování kroku

Při přejmenování kroku je do elementu *<migrate-command>* přidáván parametr *old* s předchozím názvem. Viz výpis 3.8.

Výpis 3.8: Přejmenování kroku

```
<migrate to-version="2">
  <migrate-command name="New Step Name" old="Step Name"/>
</migrate>
```

Přidání parametru kroku

Pokud nový parametr nemá výchozí hodnotu, není potřeba provádět žádné speciální akce. Všem existujícím krokům přibude nový parametr, který nebude mít nastavenu žádnou hodnotu. V případě, že je hodnota vyžadována ale nebude možné takové procesy spustit.

Pokud má nový parametr výchozí hodnotu a cílem je ji nastavit všem krokům, je nutné přidat element *<migrate-properties>*. Viz výpis 3.9.

Výpis 3.9: Přidání parametru kroku

```
<migrate to-version="2">
  <migrate-command name="Step Name">
    <migrate-properties>
      <migrate-property name="Property Name" default="..."/>
    </migrate-properties>
  </migrate-command>
</migrate>
```

Odstranění parametru kroku

Podobně jako v případě odstranění kroku není potřeba dělat nic.

Přejmenování parametru kroku

Při přejmenování parametru kroku se do elementu *<migrate-property>* přidává atribut *old* s předchozím názvem. Viz výpis 3.10.

Výpis 3.10: Přejmenování parametru kroku

```
<migrate to-version="2">
  <migrate-command name="Step Name">
    <migrate-properties>
      <migrate-property name="New Property Name"
                        old="Property Name"/>
    </migrate-properties>
  </migrate-command>
</migrate>
```

4 Výběr problému

Bylo identifikováno několik nástrojů, se kterými by bylo užitečné UCD integrovat a vytvořit pro ně rozšíření.

SAG webMethods Integration Server zahrnuje nástroj pojmenovaný Deployer, určený pro nasazování produktů na servery webMethods. Cílem by bylo vytvořit rozšíření, které by umožnilo pomocí tohoto nástroje nasazovat balíčky.

ServiceNow je nástroj pro podporu řízení IT služeb. Použití vytvořeného rozšíření by bylo takové, že by se během nasazování aplikace automaticky přenášeli informace o jeho průběhu přímo do ServiceNow prostřednictvím REST nebo SOAP API. Úspěch či neúspěch nasazení by se mohl promítnout do stavu příslušného tiketu a podobně. Další možné použití by mohlo být v získání informací z tiketu, pro potřeby procesu nasazování.

Již existuje oficiální rozšíření pro integraci s tímto nástrojem, které ovšem ve spojení s danou konkrétní instancí ServiceNow, pro kterou bude rozšíření primárně vyvíjeno, nelze použít. Je to především kvůli API, které toto rozšíření používá. Mimo to je existující rozšíření pro běžné uživatele složité na použití, protože poskytuje pouze kroky obecnějšího charakteru, které vyžadují znalost interně používaných názvů. Nové rozšíření by bylo specializované na správu změn.

IBM Rational Team Concert je nástroj pro podporu týmového vývoje software. Obsahuje správu tiketů, správu verzí, řízení práce a plánování. I pro tento nástroj existují rozšíření, které se ovšem zabývají pouze některými aspekty tohoto komplexního nástroje.

VMware vRealize Automation je nástroj, prostřednictvím kterého si mohou týmy vyžádat a spravovat virtuální servery včetně síťové architektury na cloudové platformě VMware. Vytvořené rozšíření by bylo využíváno pro automatizaci alokace virtuálních serverů.

Nakonec bylo rozhodnuto o realizaci rozšíření pro integraci s webMethods Deployer a ServiceNow, protože jejich vytvoření má nejvyšší prioritu. Zároveň se jedná o představitele dvou nejčastějších forem rozšíření, a to rozšíření „obalující“ CLI a rozšíření komunikující se vzdáleným serverem.

5 Příprava na vývoj

5.1 Instalace UrbanCode Deploy

Pro testovací účely bude UCD, kvůli svým nárokům na operační systém, nainstalován do virtuálního stroje, a to včetně agenta. Samotný vývoj rozšíření poté může po přesměrování portů, na kterých poslouchá server, probíhat na hostitelském stroji.

Trial verzi UCD je možné zdarma získat na stránkách IBM[1]. Je k tomu ovšem nutná registrace.

Instalace

1. Výběr a instalace virtualizačního nástroje. Byl zvolen volně dostupný VirtualBox[2].

2. Výběr operačního systému. UCD nelze nainstalovat na desktopové operační systémy Windows (pouze na Windows Server) a z Linuxových distribucí je oficiálně podporován pouze Red Hat Enterprise Linux (RHEL). Byl tedy zvolen RHEL, který je po registraci dostupný zdarma k vyzkoušení[3].

3. Vytvoření virtuálního stroje a instalace operačního systému. Pro naše potřeby stačí alokovat disk o velikosti 10 GB a 2 GB RAM.

4. Instalace JDK 8[4] a nastavení proměnné prostředí `JAVA_HOME`.

5. Instalace UCD podle návodu[5]. Instalátor se bude dotazovat na různá nastavení. Kde to jde, je možné nechat výchozí hodnoty.

6. Instalace agenta podle návodu[6].

7. Instalace licenčního serveru. U trial verze UCD lze vynechat.

[1] `https://developer.ibm.com/urbancode/products/urbancode-deploy`
[2] `https://www.virtualbox.org`
[3] `https://developers.redhat.com/products/rhel/download`
[4] `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
[5] `https://www.ibm.com/support/knowledgecenter/SS4GSP_6.1.3/com.ibm.udeploy.install.doc/topics/server_install_interactive.html`
[6] `https://www.ibm.com/support/knowledgecenter/SS4GSP_6.1.3/com.ibm.udeploy.install.doc/topics/agent_install_ov.html`

Přesměrování portů

Pro přístup k serveru z hostitelského systému, je nutné odblokovat a přesměrovat porty. Ve výchozím nastavení je webový server na portu 8080. RHEL má poměrně restriktivní nastavení firewallu. Port 8080 se odblokuje následovně:

```
sudo firewall-cmd --zone=public --add-port=8080/tcp
    --permanent
sudo firewall-cmd --reload
```

Dále je nutné ve VirtualBoxu pro daný virtuální stroj vytvořit pravidlo pro přesměrování portů (Nastavení → Síť → Pokročilé → Předávání portů). Po těchto změnách bude zřejmě nutné virtuální stroj restartovat.

Spuštění

- UCD server lze spustit dvěma způsoby. S parametrem *run* bude blokovat příkazový řádek a vypisovat logy. Při zadání parametru *start* bude spuštěn bez blokování.

  ```
  # způsob 1
  sudo /opt/ibm-ucd/server/bin/server run
  # vypnout: ctrl+c

  # způsob 2
  sudo /opt/ibm-ucd/server/bin/server start
  sudo /opt/ibm-ucd/server/bin/server stop
  ```

  Cesty se mohou samozřejmě lišit.

- Agenta lze spustit a vypnout podobným způsobem.

  ```
  # způsob 1
  sudo /opt/ibm-ucd/agent/bin/agent run
  # vypnout: ctrl+c

  # způsob 2
  sudo /opt/ibm-ucd/agent/bin/agent start
  sudo /opt/ibm-ucd/agent/bin/agent stop
  ```

  Agenta není nutné spouštět, pokud není potřeba nic nasazovat.

- Webové rozhraní UCD je dostupné na adrese: `http://localhost:8080`

5.2 Výběr programovacího jazyka

Teoreticky je možné zvolit libovolný programovací jazyk, ale je nutné počítat s tím, že UCD agenti, na kterých se kroky spouštějí mohou mít různou konfiguraci, a dokonce i různé operační systémy. UCD garantuje pouze dostupnost JRE a Groovy. Pokud je tedy cílem vytvořit rozšíření, které bude fungovat na všech agentech, musíme se omezit na platformu Java.

Drtivá většina stávajících rozšíření s otevřeným zdrojovým kódem používá Groovy. Dalším argumentem pro výběr Groovy je, že autoři UCD dávají k dispozici pomocnou knihovnu[7] usnadňující vývoj rozšíření v tomto jazyce. Knihovna je minimalistická, pomůže při načtení vstupních a uložení výstupních parametrů nebo při spouštění příkazů příkazové řádky. Není ale nutné ji využít.

Pro výše uvedené důvody jsem se rozhodl pro použití jazyka Groovy.

5.2.1 Programovací jazyk Groovy

Groovy je dynamický, objektově orientovaný programovací jazyk postavený na platformě Java. Program v Groovy je možné zkompilovat do Java bajtkódu nebo interpretovat přímo.

Je úzce spjatý s jazykem Java, v mnohém z něj vychází, sdílí jeho standardní knihovnu a často se používá jako skriptovací jazyk u programů v něm vyvíjených. Proto Groovy krátce popíšeme tím způsobem, že tyto dva jazyky porovnáme. Více informací lze nalézt na oficiálních webových stránkách.[8]

Syntaxe

Syntaxe vychází z jazyka Java. Většinu z ní zachovává a zavádí celou řadu nových konstruktů a zjednodušení, které mohou výrazně zkrátit kód. Někdy se proto používá pro tvorbu doménově specifických jazyků.

Hlavní rozdíly:

- možnost vynechat středníky a v některých případech závorky;
- možnost vynechat definici třídy a metodu `main` (u skriptů);
- příkaz *return* je volitelný;
- podpora pro přetěžování operátorů;

[7] `https://github.com/IBM-UrbanCode/groovy-plugin-utils`
[8] `http://groovy-lang.org`

- jiný zápis pro inicializaci pole – `[1, 2]` místo `{1, 2}`, přidává také zápis pro inicializaci seznamu, mapy a množiny;
- do mapy je možné přistupovat také pomocí tečkové notace;
- přidává nová klíčová slova: `as`, `def`, `in`, `trait`;
- podpora pro interpolované řetězce;
- používá jednoduché uvozovky pro řetězec, literál pro znak chybí;
- výchozí modifikátor je `public`;
- generování *getterů* a *setterů*, implicitní volání *getterů* a *setterů* při práci s vlastnostmi objektu.

Sémantika

Rozdíly v sémantice:

- vše je objekt;
- volitelné typování;
- výběr volaných metod je prováděn až za běhu, nikoli při kompilaci, říká se tomu dynamický výběr (*dynamic dispatch*);
- existence tzv. meta-tříd, které umožňují například přidání metody do existující třídy – tímto způsobem jsou rozšířeny mnohé třídy ze standardní knihovny Javy;
- místo lambda výrazů má uzávěry (closures), které se liší v tom, že z nich lze modifikovat lokální proměnné;
- operátor `==` je ekvivalentní volání metody `equals`.

Pomocí anotace `@ComplileStatic` lze v rámci třídy či metody nařídit statickou kompilaci, která bude vynucovat statické typování a provádět statický výběr. To se často využívá pro urychlení kritických částí kódu nebo pro zajištění větší typové bezpečnosti.

Interoperabilita s jazykem Java

Program v Groovy může bez problémů využívat knihovny napsané v Javě. Naopak to lze také, ale nebudou fungovat některé dynamické vlastnosti, například metody definované pomocí meta-tříd.

5.3 Sestavení rozšíření

Struktura rozšíření se dost liší od běžných distribučních formátů, jako je JAR. Proces sestavování zahrnující kompilaci, spuštění testů a tvorbu ZIP souboru se skripty, konfiguračními soubory a knihovnami je nutné nějakým způsobem automatizovat.

5.3.1 Rozšíření pro Eclipse IDE

IBM poskytuje rozšíření pro vývojové prostředí Eclipse (*Plug-in Development Kit*), které usnadní tvorbu rozšíření pro UCD. Zahrnuje v sobě průvodce pro vytvoření projektu, editory konfiguračních souborů (umožní snadnou úpravu i bez znalosti jejich struktury), nástroj pro sestavení rozšíření a instalaci do UCD, a integrovanou nápovědu. Interně používá Apache Ant.

Použití tohoto rozšíření je tedy asi nejjednodušší způsob, jak začít. Problém však může být, jak jej získat – není volně ke stažení. Dalším problémem pro někoho může být samotné Eclipse nebo filozofie rozšíření, které programátorovi ubírá kontrolu.

Nakonec jsem se rozhodl toto rozšíření nepoužít a vydat se cestou vlastního build skriptu a vývojového prostředí dle své volby.

5.3.2 Výběr nástroje pro automatizaci sestavování

Typické nástroje pro automatizaci sestavování na platformě Java jsou Apache Ant, Apache Ivy a Gradle.

Apache Ant + Apache Ivy

Postup sestavení (*build skript*) se definuje procedurálně pomocí XML. Z pohledu přístupu se podobá nástroji Make. U větších projektů *build skript* může postupně narůst do velkých rozměrů a stát se nepřehledným. Kvůli tomu se často stává terčem kritiky. Pro zajištění správy závislostí lze kombinovat s nástrojem Apache Ivy.

Apache Maven

Maven je další nástroj pro automatizaci sestavování, který používá konfigurační soubory definované v XML. Na rozdíl od Apache Ant, Apache Maven standardizuje strukturu projektu a životní cyklus. Stará se i o správu závislostí. Projekty jsou popisovány deklarativně.

Ze své osobní zkušenosti si dovolím tvrdit, že při nestandardní úkonech bývá pro svoji deklarativnost až příliš těžkopádný.

Gradle

Gradle je, co se týče přístupu, někde mezi výše uvedenými nástroji. V mnoha ohledech na nich přímo staví – využívá Maven úložiště, má stejnou strukturu projektu, podobný životní cyklus, umožňuje přímo vykonávat Ant skripty atd.

Postup sestavení se definuje v doménově specifickém jazyce (DSL) založeném na Groovy. Kombinuje deklarativní a procedurální přístup. Díky Groovy, který dokáže být velmi expresivní, jsou výsledné skripty v porovnání s předchozími nástroji výrazně kratší.

Závěr

Rozhodl jsem se použít Gradle, protože v něm, podle mého názoru, bude implementace požadované logiky snazší.

5.3.3 Tvorba skriptu pro automatizaci sestavování

Kompilaci a spuštění testů zajistí Groovy Plugin, který je ve výchozím stavu již součástí Gradle. Pro vytvoření distribučního ZIP archivu s rozšířením lze použít task odvozený od typu `Zip`.

Generování class path

Při definování kroků v souboru `plugin.xml` sestavujeme příkaz, který spouští program vykonávající samotnou logiku kroku. V případě programu v Groovy je součástí tohoto příkazu *class path* (`-cp`). Tento parametr předávaný virtuálnímu stroji určuje, kde se mají případně hledat použité třídy, pokud nebyly nalezeny v samotném programu ani ve standardní knihovně. Jedná se o seznam cest oddělených dvojtečkami.

Parametr by bylo určitě výhodné sestavovat automaticky, abychom ho při každé změně v závislostech (stačila by změna verze) nemuseli na mnoha místech ručně měnit. Pokud vyjdeme z předpokladu, že má každý krok stejné závislosti (v distribučním ZIP archivu umístěné v adresáři `/lib`), můžeme ho sestavit třeba jako ve výpisu 5.1.

Výpis 5.1: Automatické sestavení *class path*

```
def libs = configurations.runtime.files.findAll {
    // vynecháme knihovnu Groovy
    def name = it.getName().toString()
    return !name.matches('groovy-all-.+\\.jar')
}

def classpath = libs.collect { 'lib/' + it.name }.join(':')
```

Generování souboru plugin.xml

Zjistil jsem, že je nepohodlné udržovat odděleně definici kroků v souboru `plugin.xml` od skriptů, které implementují jejich logiku. Často se mi například stávalo, že jsem při větších úpravách změnil název parametru v definici kroku a zapomněl ho změnit ve skriptu (nebo naopak) nebo při vytváření kroku referencoval jiný skript. Přecházení mezi dvěma místy mě vyčerpávalo a kontrola byla obtížná. Kromě toho, je XML dle mého názoru příliš „upovídaný“ formát, vhodný spíše pro strojovou výměnu dat. U větších rozšíření navíc narůstá do velkých rozměrů.

Další věc, která mi vadí na struktuře `plugin.xml` je, že přímo obsahuje skripty zajišťující post-processing. Takové míchání dvou různých jazyků mi přijde nevhodné. Kromě toho obvykle dochází k opakování stejného kódu pro každý krok.

Hledal jsem proto způsob, jak rozumně definovat krok celý na jednom místě, ideálně každý v samostatném souboru. Nakonec jsem to vyřešil pomocí anotací, které se umístí k definici třídy s implementací logiky kroku. Použití ukazují výpisy 5.2 a 5.3.

Výpis 5.2: Definice kroku s použitím anotací (importy vynechány)

```
@Step(name='Example', description='Some text',
        postProcessing='/script.js', properties=[
    @Property(name='note', type=TEXT_BOX, label='Note'),
    @Property(name='state', type=SELECT_BOX, label='State',
            defaultValue='10', values=[
        @Value(label='Draft', value='10'),
        @Value(label='Suspended', value='15'),
        @Value(label='Closed', value='110')
    ]),
])
class StepCreateDeploymentCandidate {
    static main(String[] args) {
        // logika kroku
    }
}
```

Výpis 5.3: Část z `plugin.xml` odpovídající kroku z Ukázky 5.2

```
<step-type name='Example'>
  <description>Some text</description>
  <properties>
    <property name='note' required='true'>
      <property-ui type='textBox' label='Note'/>
    </property>
    <property name='state' required='true'>
```

```
      <property-ui type='selectBox' label='State'
          default-value='10'/>
      <value label='Draft'>10</value>
      <value label='Suspended'>15</value>
      <value label='Closed'>110</value>
    </property>
  </properties>
  <post-processing>
    <![CDATA[
      // obsah souboru script.js
    ]]>
  </post-processing>
  <command program='${GROOVY_HOME}/bin/groovy'>
    <arg value='-cp'/>
    <arg path='classes:lib/ucd-plugin-builder-1.0.0.jar:lib/
       groovy-plugin-utils-1.0.jar:lib/gson-2.3.1.jar'/>
    <arg file='ExampleStep.groovy'/>
    <arg file='${PLUGIN_INPUT_PROPS}'/>
    <arg file='${PLUGIN_OUTPUT_PROPS}'/>
  </command>
</step-type>
```

Při realizaci jsem vytvořil samostatný projekt `ucd-plugin-builder`, jenž obsahuje program v Groovy, který na vstupu přebírá adresář s definicemi kroků (skripty), *class path* pro sestavení spouštěcího příkazu, obecné údaje o rozšíření do hlavičky a cestu k výstupnímu souboru. Program prochází všechny skripty, kompiluje je a s použitím reflexe z anotací vygeneruje konfigurační soubor `plugin.xml`.

Program se do cílového projektu zavede podobně jako každá jiná závislost. Předtím však musí být nainstalován na lokálním Maven úložišti příkazem `gradle install`. Z Gradle tento program spustí task odvozený od typu `JavaExec`.

Závěr

Vytvořil jsem projekt s názvem `ucd-plugin-template`. Ten lze použít jako šablonu pro tvorbu dalších rozšíření. Řešení má plnou podporu ve vývojovém prostředí IntelliJ IDEA. Pro zprovoznění v IntelliJ IDEA stačí tento projekt importovat jako Gradle projekt a nechat jej synchronizovat (*View* → *Tool Windows* → *Gradle* → tlačítko pro synchronizaci).

Tvorba vlastního řešení si sice vyžádala nějaký čas navíc, především navržení vlastního způsobu pro definici kroků, ale věřím, že se mi tato investice vrátí.

6 Rozšíření pro webMethods Integration Server

6.1 WebMethods

WebMethods je platforma pro integraci IT služeb vyvíjená společností Software AG. Jádrem této platformy je webMethods Integration Server. Ten umožňuje integraci různých služeb tím, že zajišťuje mapování dat mezi formáty a komunikaci mezi systémy.

Součást webMethods Integration Server, je nástroj zvaný Deployer. Ten slouží pro nasazování prostředků různého charakteru, nacházejících se na serverech webMethods na cílový server webMethods [12]. Příkladem může být nasazení balíčku, který byl vyvinut na serveru ve vývojovém prostředí, na servery v testovacím nebo produkčním prostředí [12]. Nástroj nabízí webové rozhraní a CLI. Hlavním zdrojem informací o tomto nástroji je oficiální dokumentace [12], podle níž byla také zpracována analýza.

6.2 Analýza

Cílem je vytvořit rozšíření poskytující množinu kroků, které umožní prostřednictvím nástroje webMethods Deployer nasazovat balíčky. Je potřeba si uvědomit, že rozšíření nemá sloužit k prvotní konfiguraci projektů, ale k nasazování nových verzí na testovací nebo produkční prostředí.

6.2.1 Postup nasazování s webMethods Deployer

1. Vytvoření projektu. Zahrnuje výběr jména, nastavení parametrů a přidělení rolí dalším uživatelům.

2. Identifikace zdrojů, které patří do projektu a mají být nasazovány. Tyto zdroje se musejí nacházet na serverech webMethods. Například Integration Server pracuje s již zmíněnými balíčky, ale na platformě webMethods existují i další typy serverů s různými typy zdrojů.

3. Sestavení projektu – vytvoří soubor se všemi zdroji, tzv. *build*.

4. Mapování zdrojů na cílové servery. Vytváří se tzv. *deployment map*.

5. Nasazení projektu na cílové servery. Při nasazování se prvně vytváří *deployment candidate*, což je kombinace *build* a *deployment map*. Volitelně může být spuštěna simulace nasazení, která může pomoci zachytit např. problémy v konfiguraci. A nakonec samotné nasazení.

6.2.2 Popis CLI

Základní struktura příkazu vypadá následovně:

```
Deployer.<sh|bat> --<command>
    -host <url> -port <port> -user <user> -pwd <password>
```

Podle typu příkazu (výše jako `<command>`) může obsahovat další parametry.

Přehled příkazů

Jsou vynechány příkazy, které slouží pouze ke zjišťování informací a výpisu parametrů. Vynechány jsou rovněž některé konfigurační příkazy. Je vždy uvedena pouze specifická část příkazu.

`--create -build <B> -project <P>`
: Vytvoří *build* s názvem `<B>` v projektu `<P>`.

`--export -build <B> -project <P> -overwrite <true|false>`
: Exportuje *build* s názvem `<B>` z projektu `<P>` a uloží jej do adresáře s exporty.

`--import -buildFile <BF> -project <P> -overwrite <true|false>`
: Importuje soubor s *buildem*, daný úplnou cestou `<BF>`, do projektu `<P>`. Pokud nebude název projektu zadán, tak bude vytvořen nový projekt. Importovat *build* je možné pouze z adresáře určeného pro import.

`--export -map <M> -project <P>`
: Exportuje *deployment map* s názvem `<M>` z projektu `<P>` a uloží jej do adresáře s exporty.

`--import -mapFile <MF> -project <P>`
: Importuje soubor s *deployment map*, daný úplnou cestou `<MF>`, do projektu `<P>`. Importovat *build* je možné pouze z adresáře určeného pro import.

`--create -dc <C> -build <B> -map <M> -project <P>`
: Vytvoří *deployment candidate* s názvem `<C>` v projektu `<P>` a použije k tomu *build* `<B>` a *deployment map* `<M>`.

`--checkpoint -dc <C> -project <P>`
Vytvoří *checkpoint* pro daný *deployment candidate* `<C>` projektu `<P>`.

`--simulate -dc <C> -project <P>`
Provede simulaci nasazení daného *deployment candidate* `<C>` projektu `<P>` a uloží záznam.

`--deploy -dc <C> -project <P>`
Provede nasazení daného *deployment candidate* `<C>` projektu `<P>`.

`--rollback -dc <C> -project <P>`
Zvrátí nasazení daného *deployment candidate* `<C>` projektu `<P>`. Odebere všechny prostředky nasazené s `<C>`.

`--delete -<map|dc> <X> -project <P>`
Odstraní *deployment map* nebo *deployment candidate* s názvem `<X>` z projektu `<P>`.

`--delete -project <P>`
Odstraní projekt `<P>`.

6.3 Návrh kroků

Z přehledu příkazů v sekci 6.2.2 jsem vybral několik příkazů, o kterých předpokládám, že budou potřeba při automatizaci nasazování. Pro tyto příkazy budou vytvořeny kroky:

- *Import Build* (resp. *Create Project*),
- *Import Map*,
- *Create Deployment Candidate*,
- *Create Checkpoint at Target Server*,
- *Simulate Deployment to Target Server*,
- *Deploy Project to Target Server*.

Pomocí těchto kroků je možné automatizovat celý proces nasazení nové verze balíčku. Předpokladem je vytvořený *build* a *deployment map*.

6.4 Implementace

Implementace s použitím připraveného šablonového projektu (jeho tvorba je popsána v sekci 5.3.3) byla jednoduchá. Kromě tříd definující kroky projekt obsahuje dvě další třídy:

- `Deployer` ze vstupních parametrů vytváří příkazy příkazové řádky a spouští je. Tuto třídu používají obsluhy kroků.
- `CommandBuilder` je pomocná třída pro sestavování příkazů příkazové řádky. Tuto třídu interně používá třída `Deployer`.

Zabýval jsem se tím, jak vyřešit opakující se vstupní parametry pro identifikaci serveru a autentizaci uživatele. Zjistil jsem, že zřejmě neexistuje lepší způsob, než pro ně vytvořit vstupní pole. Neznamená to ale, že by uživatel musel tyto údaje vyplňovat stále dokola u všech kroků. Kroky mají vstupní pole ve výchozím stavu nastavené na parametry, např.: `${p:wm.user}`. Uživatel pouze definuje parametry se stejnými jmény například v rámci procesu nebo komponenty a tím se tyto údaje dostanou do všech vytvořených kroků. Tento přístup zároveň nevylučuje složitější použití, kdy by se v rámci jednoho procesu pracovalo s více servery a podobně.

Použité knihovny

Knihovny jsou uvedeny ve formátu `<group id> / <artifact id>`.

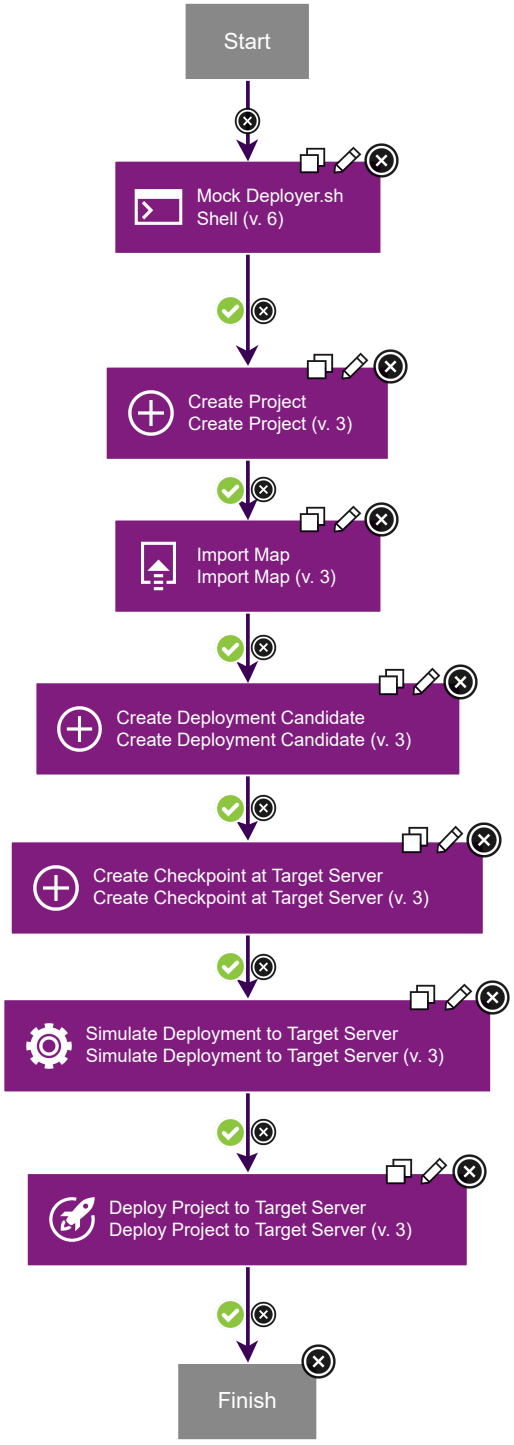
- `org.codehaus.groovy` / `groovy-all` – Kompilátor a standardní knihovna Groovy v jednom.
- `com.ibm.urbancode.plugins` / `groovy-plugin-utils` – Knihovna od tvůrců UCD usnadňující tvorbu rozšíření v Groovy.
- `junit` / `junit` – Populární knihovna JUnit pro testování.

6.5 Testování

Obecně můžeme testování každého rozšíření do UCD rozdělit na dvě části. Funkčnost jednotlivých tříd lze otestovat jednotkovými testy. Rozšíření samotné je vhodné otestovat jako celek tak, že se z kroků vytvoří procesy, které se při vydání nové verzi spustí. Tímto se zároveň testuje jeho zpětná kompatibilita – po nahrání nové verze by nemělo dojít k „rozbití“ stávajících procesů.

Sestavování příkazů příkazové řádky je otestováno jednotkovými testy. Jako celek je rozšíření otestováno v UCD generickým procesem, který používá jednoduchý mock nástroje webMethods Deployer (viz obr. 6.1). Jsou v něm využity všechny kroky a demonstruje celý proces nasazování balíčku na webMethods Integration Server. Zároveň může uživatelům rozšíření posloužit jako vzor. Zmíněný proces je přiložen na DVD ve formátu umožňujícím import do UCD, spolu se snímky webového rozhraní UCD zachycující jeho úspěšné vykonání. Rozšíření bylo jednorázově otestováno i při skutečném nasazení balíčku, ale příprava je příliš zdlouhavá, a proto je pro běžné testování vhodnější použít mock nástroje.

Ukázalo se, že je pro testování lepší používat generické procesy, než se snažit vytvářet testovací aplikace, prostředí, komponenty a podobně. Vhodně připravené generické procesy lze na rozdíl od komponentových procesů opakovaně spouštět v podstatě na jedno kliknutí. Další výhodou generických procesů je, že jejich příprava je mnohem rychlejší.



Obrázek 6.1: Proces *webMethods Deployer Test*

7 Rozšíření pro ServiceNow

7.1 ServiceNow

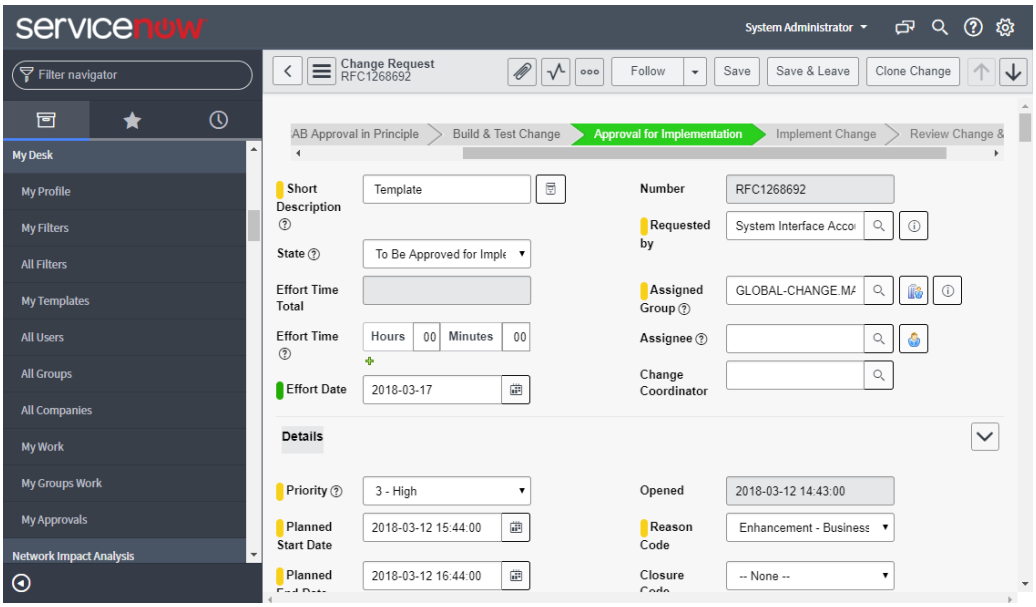
ServiceNow je nástroj pro podporu poskytování IT služeb, od stejnojmenné společnosti. Následuje rámec ITIL (*Information Technology Infrastructure Library*), který představuje standard pro řízení služeb v oblasti informačních technologií. Uživatelé ServiceNow jsou především velké společnosti. Jedná se o cloudové řešení prodávané jako služba (SaaS – *Software as a Service*). Webové rozhraní ServiceNow je ukázáno na obrázku 7.1. Hlavním zdrojem informací o tomto nástroji je oficiální dokumentace: [10].

Některé oblasti použití:

- Správa aktiv (*Asset Management*) – Správa hmotného i nehmotného majetku v držení společnosti, dohled nad jeho efektivním užíváním. [10]
- Řízení změn (*Change Management*) – Kontrolované zavádění změn do produkčního prostředí. Změna je jakákoliv aktivita, který může mít vliv na kvalitu poskytovaných služeb [10]. Požadavek na změnu (*Request for Change* – RFC), má svůj životní cyklus a obsahuje veškeré informace o dané změně. Jedná se v podstatě o zúžení obecnějšího pojmu tiket. Může v sobě obsahovat ještě úkoly (*Change Tasks*).
- Správa incidentů (*Incident management*) – Cílem je rychlé odstranění problému a obnovení provozu služby [10]. Incident (druh tiketu) popisující problém může vytvořit zaměstnanec dané společnosti nebo koncový uživatel prostřednictvím technické podpory. Nemusí se však jednou pouze o technický problém, incidentem může být také žádost o službu či o informace.

7.2 Analýza

Rozšíření se bude používat především pro práci s požadavky na změnu (oblast řízení změn). Aby bylo možné navrhnout kroky, je nutné porozumět životnímu cyklu RFC. Implementace navržených kroků poté bude vyžadovat seznámení se s rozhraním ServiceNow.



Obrázek 7.1: Webové rozhraní ServiceNow

7.2.1 Životní cyklus požadavku na změnu

RFC je vytvořen se stavem *Draft*. Po jeho dodefinování a případném vytvoření souvisejících úkolů se přepne do stavu *Registered*. Po zhodnocení rizika a vyplnění dalších polí se přepne do stavu *To Be Approved for Implementation*. Pokud RFC není předschválený, musí pro něj být vytvořen tzv. *Group Approval*, který musí schválit alespoň jeden uživatel patřící do skupiny, ke které byl RFC přiřazen. Po schválení dojde k automatickému přepnutí do stavu *Approved for Implementation*. Následuje implementace a uzavření všech úkolů. Po dokončení se přepne stav na *Implemented*. Následuje zhodnocení a uzavření, tj. změna stavu na *Closed*.

Úkoly se dělí na dva typy: *Build And Test Tasks* a *Implementation Tasks*. Mají také svůj životní cyklus se stavy *Draft* → *Registered* → *Assigned* → *In Progress* → *Completed*/*Cancelled*/*Suspended*.

RFC a jejich podřízené úkoly obsahují povinná a nepovinná pole. Ve webovém rozhraní jsou nevyplněná povinná pole označena červeně a dokud nejsou vyplněna, není možné tiket uložit. V počátečním stavu (*Draft*) je jich jen několik a v dalších stavech přibývají. Někdy mohou být pole nepovinná a po vyplnění jiného pole se povinnými stát (např. pokud úkol chceme uzavřít jako *Failed*, je nutné vyplnit proč). Podobně mohou být některá pole skryta a po nastavení specifické hodnoty do pole, na kterém závisí mohou být následně zobrazena.

Tento popis životního cyklu RFC je do určité míry zjednodušený. Některé

typy RFC mohou požadovat průchod dalšími stavy, které se jinak vynechávají nebo mít složitější proces schvalování apod. Například RFC pro stavbu budovy bude mít jiné procesní nároky než RFC pro změnou v softwarovém produktu.

7.2.2 Organizace dat v ServiceNow

Záznam (např. RFC) se skládá z polí (např. číslo, stav,...). Záznamy jsou organizovány v tabulkách. Na záznamy se také můžeme dívat jako na řádky tabulky a na pole jako na sloupce tabulky.

Pole má popisek, který se uživateli zobrazuje ve webovém rozhraní, formu vstupu (textové pole, rozbalovací seznam,...) a název, se kterým se interně pracuje.

Každá tabulka obsahuje pole s názvem `sys_id`, které slouží jako unikátní identifikátor záznamu napříč databází. Jedná se o řetězec dlouhý 32 znaků. Můžeme jej najít v URL při prohlížení záznamu, ale jinak je běžnému uživateli skryt.

Je potřeba vzít v potaz, že administrátor ServiceNow může dle libosti tabulky měnit a přizpůsobit tak nástroj potřebám své společnosti. Nemáme tedy jistotu, že určitá tabulka nebo pole bude na dané instanci k dispozici.

7.2.3 Dostupná rozhraní ServiceNow

ServiceNow poskytuje REST API a SOAP API pro čtení, modifikaci a vytváření záznamů. Tato rozhraní mohou být navíc rozšířena o další funkcionalitu.

SOAP API

SOAP (*Simple Object Access Protocol*) je formát pro výměnu strukturovaných dat založený na XML. Zprávy se posílají přes HTTP, metodou POST, pod typem XML, na adresu ve formátu:

`http://<doména>/<tabulka>.do?SOAP`

Dostupné funkce ServiceNow má nad všemi tabulkami definované následující funkce:

- `get` – Vrátí záznam podle `sys_id`. Použití viz výpis 7.1.
- `getKeys` – Vrátí seznam `sys_id` záznamů, které mají pro daná pole stejné hodnoty.
- `getRecords` – Vrátí záznamy, které mají pro daná pole stejné hodnoty.

- `aggregate` – Jako `getRecords`, ale s možností použití agregační funkce `SUM`, `COUNT`, `MIN`, `MAX` a `AVG`.
- `insert` – Vytvoří nový záznam.
- `insertMultiple` – Vytvoří více záznamů.
- `update` – Upraví záznam, který ja zadán pomocí `sys_id`.
- `deleteRecord` – Smaže záznam, daný `sys_id`.
- `deleteMultiple` – Smaže záznamy, které mají pro daná pole stejné hodnoty.

Výpis 7.1: SOAP požadavek na položku v tabulce `change_request`

```
<soap:Envelope
    xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
    xmlns:ns='http://www.service-now.com/change_request'>
  <soap:Header/>
  <soap:Body>
    <ns:get>
      <sys_id>026ce682dbf283c0d6dff1c51d96193c</sys_id>
    </ns:get>
  </soap:Body>
</soap:Envelope>
```

Výpis 7.2: SOAP odpověď na požadavek z výpisu 7.1

```
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <getResponse
        xmlns="http://www.service-now.com/change_request">
      <sys_id>026ce682dbf283c0d6dff1c51d96193c</sys_id>
      <short_description>TEST</short_description>
      <state>60</state>
      <!-- + další stovky polí -->
    </getResponse>
  </soap:Body>
</soap:Envelope>
```

WSDL S technologií SOAP souvisí technologie zvaná WSDL (*Web Service Description Language*), která slouží k popisu rozhraní pomocí XML. ServiceNow automaticky generuje WSDL dokumenty pro všechny tabulky[1].

[1]Pod URL ve formátu: `http://<doména>/<tabulka>.do?WSDL`

Můžeme z nich zjistit, jaké jsou k dispozici funkce a jakou strukturu bude mít odpověď.

REST API

REST (*REpresentational State Transfer*) je styl tvorby architektury rozhraní, která funguje obvykle nad HTTP. Zajišťuje snadný a jednotný přístup ke zdrojům.

Popis API ServiceNow nad tabulkami umožňuje následující operace:

- Operace nad tabulkami:
  `http://<doména>/api/now/v1/table/<tabulka>`
  - GET vrátí záznamy v tabulce. Záznamy můžeme filtrovat parametrem `sysparm_query`. Podmínky se zapisují ve formátu: `<název pole>=<hodnota>`. Více podmínek oddělíme znakem ^.
  - POST vytvoří nový záznam se zadanými hodnotami.
- Operace nad záznamy:
  `http://<doména>/api/now/v1/table/<tabulka>/<sys_id>`
  - GET vrátí záznam. Použití viz výpis 7.3 (ekvivalentní 7.1).
  - PUT a PATCH záznamu nastaví zadané hodnoty.
  - DELETE odstraní daný záznam.

Výpis 7.3: Odpověď ve formátu JSON na požadavek zaslaný na URL: `http://<doména>/api/now/v1/table/change_request/026ce6...`

```
{
  "result": {
    "sys_id": "026ce682dbf283c0d6dff1c51d96193c",
    "short_description": "TEST",
    "state": "60",
    /* + další stovky polí */
  }
}
```

Odpověď může být doručena ve formátu XML nebo JSON (možno nastavit v HTTP hlavičce parametrem `Accept`). Hodnoty polí pro vytvoření nebo upravení záznamu můžeme odesílat také ve formátu XML nebo JSON.

Scripted web services

Administrátor ServiceNow může SOAP a REST API rozšířit od další vstupní body pomocí tzv. *Scripted web services*, které se programují v jazyce JavaScript.

Autentizace

Všechny výše uvedená rozhraní používají základní HTTP autentizaci.

7.3 Návrh

7.3.1 Návrh kroků k realizaci

Cílem je vytvořit ucelenou množinu kroků, pro práci s RFC, která půjde použít pro co nejvíce možných scénářů použití, a to nejlépe pouze s uživatelskou znalostí ServiceNow. Podle sekce 7.2, popisující životní cyklus RFC, jsem navrhl následující kroky:

- *Create RFC* – Vytvoření RFC podle šablony.
- *Change RFC status* – Změna stavu RFC.
- *Close RFC* – Uzavření RFC. To zahrnuje změnu stavu na *Closed* a nastavení několika dalších polí.
- *Create Group Approval* – Vytvoření *Group Approval* pro daný RFC.
- *Approve RFC* – Schválení RFC za daného uživatele.
- *Create CTask* – Vytvoření úkolu k danému RFC podle šablony.
- *Create Build And Test Task* – Vytvoření úkolu k danému RFC typu *Build And Test Task*.
- *Create Implementation Task* – Vytvoření úkolu k danému RFC typu *Implementation Task*.
- *Close CTask* – Uzavření úkolu.
- *Close All CTasks* – Hromadné uzavření všech úkolů pro daný RFC.

Takto navržené kroky pokrývají celý životní cyklus RFC. Většina polí bude nastavena, při jeho vytváření, pomocí šablony. Reference na jiné záznamy bude možné zadávat jak ve formě `sys_id`, tak i ve formě čísla.

Kromě kroků zaměřených na RFC dáme k dispozici množinu obecnějších kroků, které však budou nevyhnutelně vyžadovat určité hlubší znalosti ServiceNow (např. interní názvy tabulek a polí). Tyto kroky umožní pokrýt zbylé scénáře použití, se kterými základní množina kroků nepočítá:

- *Raw Create* – Vytvoří nový záznam.
- *Raw Edit* – Upraví záznam.
- *Raw Check* – Zkontroluje, zda daný záznam obsahuje určité hodnoty.

Všechny kroky vytvářející nové záznamy budou poskytovat výstupní parametry s číslem a `sys_id` nově vytvořeného záznamu. Bez toho by jinak nebylo možné s nově vytvořeným záznamem provádět další operace.

7.3.2 Komunikace se serverem ServiceNow

Osobně bych preferoval REST API, protože je dle mého názoru lepší a jednodušší na použití. Ale protože je rozšíření vyvíjeno primárně pro integraci s konkrétní upravenou instancí ServiceNow, musel jsem zvolit SOAP API. Vedly k tomu problémy, které se objevily při testování existujícího rozšíření, které používá právě REST API. Jedná se spíše o záležitost na míru upravené instance ServiceNow, obecně není důvod REST API nepoužít.

Pro tvorbu HTTP požadavků využiji osvědčenou knihovnu *HttpClient* z projektu *Apache HttpComponents*.

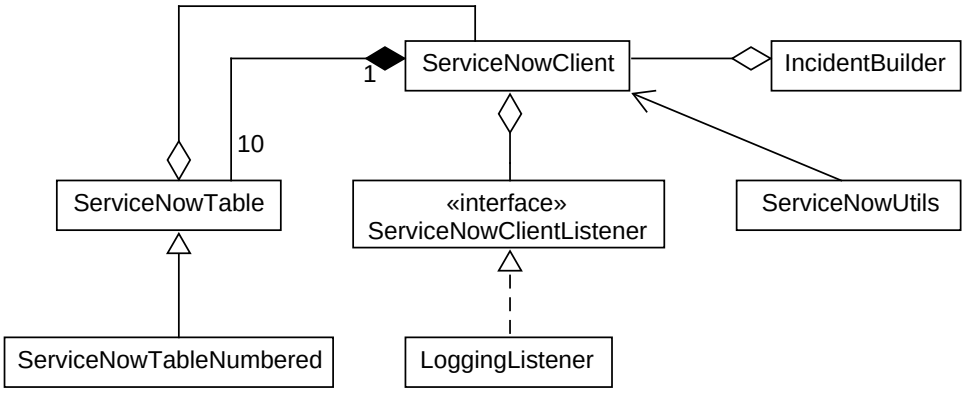
7.4 Implementace

Projekt jsem rozdělil na dva podprojekty – `sn-client` a `sn-plugin`.

Podprojekt sn-client

Obsahuje samostatně použitelnou knihovnu pro práci se ServiceNow. Je v plánu i její další použití mimo tento projekt. Obrázek 7.2 zobrazuje vztahy mezi nejdůležitějšími třídami. Přehled tříd:

- `ServiceNowClient` – Zajišťuje síťovou komunikaci. Poskytuje metody pro vyhledání, upravení a vytvoření záznamu.
- `ServiceNowTable` – Třída vytváří abstrakci pro tabulku. Poskytuje metody pro práci se záznamy nad danou tabulkou.



Obrázek 7.2: Diagram tříd

- `ServiceNowTableNumbered` – Speciální forma `ServiceNowTable` pro tabulky, které obsahují pole `number`. Existuje z důvodu, že v těchto tabulkách chceme dát možnost jednoduše vyhledávat i podle čísla.
- `ServiceNowClientListener` – Rozhraní pro třídu, která může být předána `ServiceNowClient` pro zachytávání požadavků a odpovědí. Slouží především pro potřeby logování.
- `LoggingListener` – Implementace `ServiceNowClientListener` určená pro produkční logování. Nevypisuje některá pole, která by mohla obsahovat citlivé informace.
- `IncidentBuilder` – Třída pro práci s incidenty.
- `ServiceNowUtils` – Poskytuje různé pomocné statické metody pro práci se ServiceNow.

Kromě těchto tříd existuje ještě několik výčtových typů.

Podprojekt sn-plugin

Projekt s rozšířením. Obsahuje konfigurační soubory a definice kroků, které využívají knihovnu `sn-client`. Rozšíření sestavuje s použitím skriptu navrženým v sekci 5.3.3.

Každý krok obsahuje vstupní pole s přihlašovacími údaji a URL k ServiceNow, které mají ve výchozím stavu hodnotu nastavenou na určitý parametr. Tyto parametry je nutné před použitím definovat.

Použité knihovny

Knihovny jsou uvedeny ve formátu `<group id> / <artifact id>`.

- `org.codehaus.groovy` / `groovy-all` – Kompilátor a standardní knihovna Groovy v jednom.
- `org.apache.httpcomponents` / `httpclient` – HTTP klient použitý při komunikaci se serverem ServiceNow.
- `commons-codec` / `commons-codec` – Obsahuje Base64 kóder/dekóder, použitý při implementaci HTTP autentizace.
- `com.ibm.urbancode.plugins` / `groovy-plugin-utils` – Knihovna od tvůrců UCD usnadňující tvorbu rozšíření v Groovy.
- `junit` / `junit` – Knihovna pro testování.

7.5 Testování

Testování knihovny pro práci se ServiceNow a rozšíření do UCD vyžadovalo odlišné přístupy.

7.5.1 Testování klienta

Při vývoji jsem řešil problém, jak ověřit základní funkčnost svého ServiceNow klienta. Chtěl jsem se přitom vyhnout komunikaci se skutečným serverem, a to z důvodu dlouhé odezvy a nestálosti. Jenou z možností bylo vytvořit mock serveru, například pomocí nástroje MockServer[2]. Zjistil jsem však, že stejného efektu lze mnohem snáze docílit vytvořením mocku rozhraní `HttpClient`.

Testy jsem připravil na testovacím serveru a uložil požadavky a odpovědi – to je možné provést automatizovaně vytvořením třídy implementující `ServiceNowClientListener`, která je následně předána `ServiceNowClient`. Z testovacích dat jsem ještě ručně odstranil některé řádky a nahradil několik údajů, jako například emailové adresy.

Při testování `ServiceNowClient` vytvoří požadavek, předá ho namockované instanci `HttpClient`, která ho strukturálně porovná s uloženým požadavkem. Pokud se shodují, vrátí uloženou odpověď a `ServiceNowClient` ji standardním způsobem zpracuje.

[2] `http://www.mock-server.com`

7.5.2 Testování rozšíření

Rozšíření má za sebou několik cyklů úprav a testování. Mnoho času bylo například stráveno diskusí o tom, jaká pole mají být nastavována jednotlivými kroky a jak přesně má fungovat proces schvalování RFC.

Při průběžném testování se ukázalo, jak důležité je u podobného rozšíření důkladné logování. Bez něho prakticky není možné zjistit, k jakému problému došlo a jak jej napravit.

Navržené procesy

Pro testovací a demonstrační účely bylo vytvořeno několik generických procesů. Jsou v nich použity všechny vytvořené kroky.

- *SN Test Workflow* – Testuje celý životní cyklus RFC s jedním úkolem a řeší i stav, kdy dojde k chybě při nasazování. Samotné nasazování je pouze naznačeno. Pracuje s předschváleným RFC – neobsahuje schvalování. Je zobrazen na obrázku 7.3.

  Proces může sloužit jako základ pro složitější procesy obsahující více úkolů, další mezistavy, schvalovací proces, komplexnější řešení situace v případě selhání nasazování, dynamické rozhodování na základě vyplněných polí a podobně.

  Proces má na vstupu 2 šablony – RFC a úkol. Šablonou může být obyčejný předvyplněný tiket nebo speciální tiket typu šablona. Tyto šablony musejí být manuálně vytvořeny v ServiceNow. Vyplněna musí být veškerá povinná pole. Výjimku tvoří pouze dynamicky nastavované *Planned Start/End Date*, *Customer RTP Date* (krok *Create RFC*) a pole nastavovaná při uzavírání RFC a úkolů (kroky *Close RFC* a *Close CTask*/*Close All CTasks*).

- *SN Test Change Tasks* – Testuje práci s úkoly. Paralelně vytvoří úkoly dvou různých typů, ověří obsah nastavených polí a oba úkoly uzavře. Poté ještě ověří, zda byl u úkolů skutečně změněn jejich stav a nastavena poznámka k uzavření.

  Na vstupu má libovolný RFC, který v procesu figuruje jako rodičovský RFC vytvářených úkolů.

- *SN Test Raw Steps* – Testuje kroky *Raw Create*, *Raw Edit* a *Raw Check*. Pomocí těchto obecných kroků vytvoří úkol, zkontroluje nastavená pole, uzavře ho a znovu zkontroluje jeho správné uzavření.

Na vstupu má libovolný RFC, který v procesu figuruje jako rodičovský RFC vytvářených úkolů.

- *SN Test Approvals (automated)* – Testuje schvalování tak, že RFC automatizovaně schválí jménem daného uživatele. Tento uživatel musí být ze skupiny, které byl RFC přiřazen (*assigned group*).

  Na vstupu má RFC před schválením a uživatele.

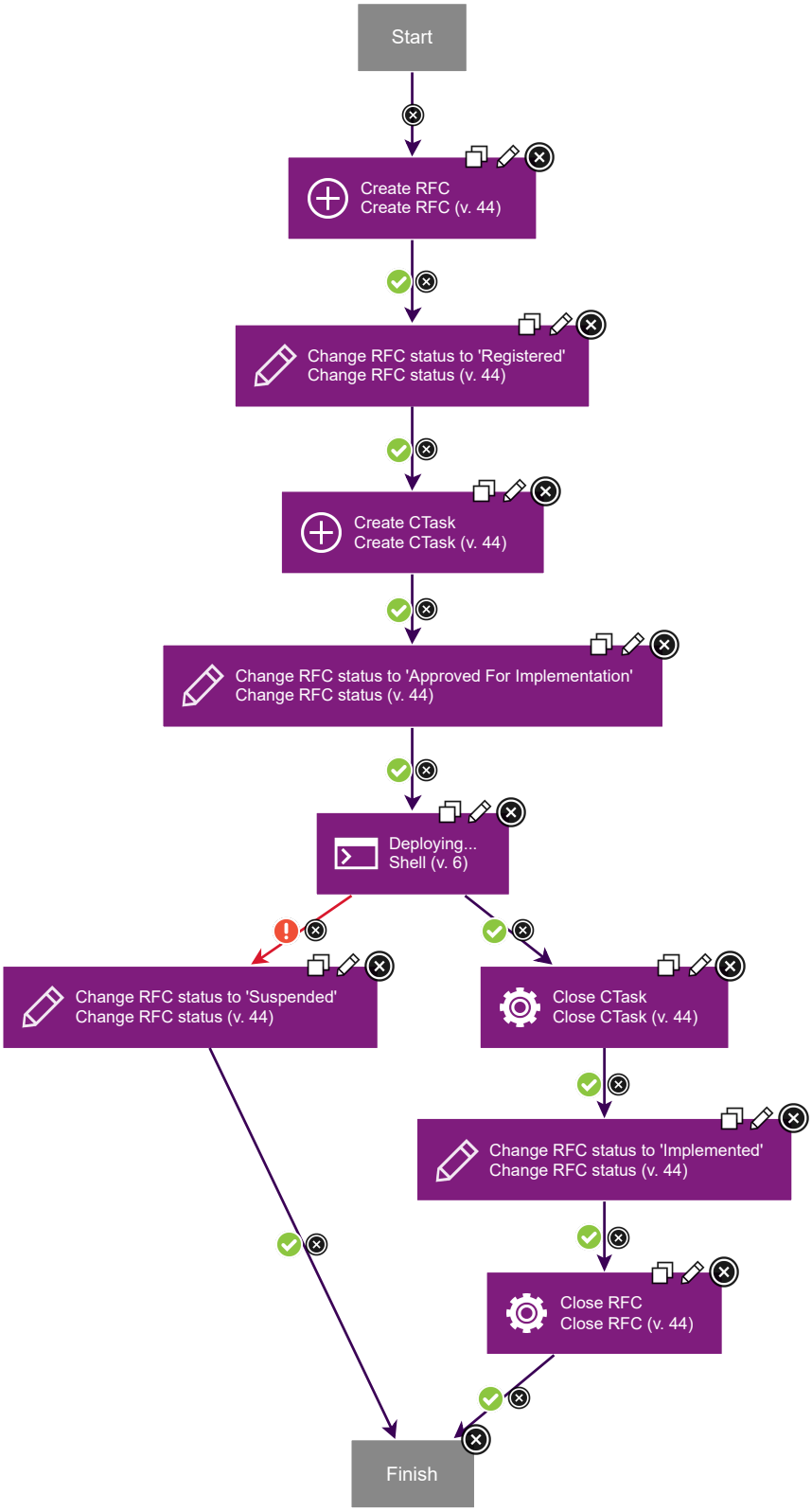
- *SN Test Approvals (automated, empty group)* – Stejně jako předchozí proces, i tento testuje schvalování RFC jménem daného uživatele. Využívá při tom ale generickou prázdnou skupinu, pomocí níž se vytvoří *group approval* pouze s tímto jedním uživatelem. Schvalující uživatel nemusí patřit do skupiny, které byl RFC přiřazen.

  Na vstupu má RFC před schválením, uživatele a skupinu.

- *SN Test Approvals (manual)* – Další proces testující schvalování vedené odlišným způsobem. Využívá vestavěný krok *Manual Task*. Po vytvoření *group approval* je vykonávání procesu nad tímto krokem pozastaveno a pověřený uživatel (nemusí to být nutně ten, kdo spustil proces) je vyzván ke schválení daného RFC. Následovat mohou dva různé scénáře. V prvním případě může RFC schválit manuálně ve webovém rozhraní ServiceNow a povolit pokračování. V druhém případě může povolit pokračování rovnou a RFC bude schváleno automatizovaně. V každém případě pověřený uživatel vyjádřil svoji vůli.

  Na vstupu má RFC před schválením, uživatele a volitelně skupinu.

Všechny uvedené procesy jsou přiloženy na DVD, a to jak ve formátu umožňujícím import do UCD, tak i jako obrázek pro snadné prohlédnutí. Přiloženy jsou také snímky z webového rozhraní UCD zachycující jejich úspěšný průběh.



Obrázek 7.3: Testovací proces *SN Test Workflow*

8 Závěr

První kapitola čtenáře seznamuje s problematikou nasazování softwarových produktů a přístupem DevOps. V další kapitole je popsán nástroj UrbanCode Deploy v rozsahu nutném pro vývoj rozšíření a následně pak samotný vývoj rozšíření. Ve čtvrté kapitole jsou vybírány nástroje, pro které by mohla být vytvořena rozšíření. Pátá kapitola se zabývá obecnými technickými záležitostmi, které bylo nutné vyřešit před samotným vývojem vybraných rozšíření. Konkrétně instalací UrbanCode Deploy, výběrem vhodného jazyka a automatizací sestavování distribučního archivu s rozšířením, spolu s výběrem vývojového prostředí. Nad rámec zadání byla vytvořena knihovna pro alternativní způsob definice kroků, která si bere za cíl ulehčit vývoj a která může pomoci i dalším vývojářům. V dalších kapitolách byla popsána tvorba rozšíření pro integraci s nástroji SAG webMethods Integration Server a ServiceNow.

Vytvořená rozšíření byla otestována jednotkovými testy a byly pro ně vytvořeny procesy v UrbanCode Deploy, které je testují jako celek. Rozšíření byla otestována nejdůkladněji pod UrbanCode Deploy ve verzi 6.1.3, ale vzhledem k tomu, že je API zpětně kompatibilní budou fungovat i na novějších verzích.

Při testování byli použiti pouze agenti nainstalovaní na strojích s operačním systémem Linux. Pokud by ale vznikl požadavek na podporu dalších operačních systémů, znamenalo by to pouze několik menších úprav. Všechny použité nástroje jsou multiplatformní.

Vzhledem k režii UrbanCode Deploy a povaze rozšíření nemělo smysl provádět měření rychlosti provedení vytvořených kroků, lze jen konstatovat, že rychlost jejich provedení je dostatečná.

ServiceNow bývá obvykle upraveno pro potřeby konkrétní společnosti. Při případném použití vytvořeného rozšíření pro integraci s jinou instancí ServiceNow by pravděpodobně byly nutné nějaké úpravy.

Přehled zkratek

API . . . . . Application Programming Interface

FaaS . . . . . Function as a Service

HTTP . . . . Hypertext Transfer Protocol

IaaS . . . . . Infrastructure as a Service

JAR . . . . . Java Archive

PaaS . . . . . Platform as a Service

QA . . . . . Quality Assurance

REST . . . . Representational State Transfer

RFC . . . . . Request for Change

SaaS . . . . . Software as a Service

SOAP . . . . Simple Object Access Protocol

UCD . . . . . UrbanCode Deploy

URL . . . . . Uniform Resource Locator

XML . . . . . Extensible Markup Language

Literatura

[1] Caum, C. *What is infrastructure as code?* [online]. 2017. [cit. 2018-04-08]. Dostupné z: https://puppet.com/blog/what-is-infrastructure-as-code.

[2] Fowler, M. *Continuous Integration* [online]. 2006. [cit. 2017-12-25]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html.

[3] Google. *CONTAINERS AT GOOGLE* [online]. 2018. [cit. 2018-03-28]. Dostupné z: https://cloud.google.com/containers/.

[4] Httermann, M. *DevOps for Developers.* Apress, 1st edition, 2012. ISBN 1430245697, 9781430245698.

[5] Humble, J. – Farley, D. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN 9780321670229.

[6] Humble, J. *What is Continuous Delivery?* [online]. [cit. 2017-12-25]. Dostupné z: https://continuousdelivery.com/.

[7] IBM. *IBM Knowledge Center / IBM UrbanCode Deploy* [online]. 2018. [cit. 2018-03-03]. Dostupné z: https://www.ibm.com/support/knowledgecenter/SS4GSP/ucd_welcome.html.

[8] Kim, G. et al. *The DevOps Handbook: How to Create World-Class Speed, Reliability, and Security in Technology Organizations.* G - Reference, Information and Interdisciplinary Subjects Series. IT Revolution Press, 2015. ISBN 9781942788003.

[9] Roberts, M. *Serverless Architectures* [online]. 2016. [cit. 2018-03-28]. Dostupné z: https://martinfowler.com/articles/serverless.html.

[10] ServiceNow. *Product Documentation* [online]. 2018. [cit. 2018-03-10]. Dostupné z: https://docs.servicenow.com.

[11] Sharma, S. *Understanding DevOps – Part 4: Continuous Testing and Continuous Monitoring* [online]. 2012. [cit. 2017-12-30]. Dostupné z: https://sdarchitect.blog/2012/10/30/understanding-devops-part-4-continuous-testing-and-continuous-monitoring/.

[12] Software AG. webMethods Deployer 7.1.1 – User's Guide, 2008.

Přílohy

Obsah DVD

Přiložené DVD obsahuje text práce a všechny vytvořené projekty. Každý projekt obsahuje soubor `README.txt` s popisem adresářové struktury daného projektu, instrukcemi pro sestavení a dalšími informacemi.

Adresářová struktura

- `/`
  - `projects` ..... Adresář s projekty
    - `ucd-plugin-builder` ..... Pomocná knihovna
    - `ucd-plugin-template` ..... Šablona pro rozšíření
    - `ucd-servicenow-plugin` ..... Rozšíření pro ServiceNow
    - `ucd-webmethods-plugin` ..... Rozšíření pro webMethods
  - `thesis` ..... Adresář s bakalářskou prací
    - `latex` ..... Zdrojové soubory bakalářské práce