FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Scheduler podporující multiplatformní prostředí
Student:	Ondřej Závodný
Vedoucí:	RNDr. Ondřej Zýka
Studijní program:	Informatika
Studijní obor:	Informační systémy a management
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Cílem bakalářské práce je analýza a návrh scheduleru, který umožňuje jednotným způsobem řídit technické procesy organizace na více platformách (WIN, Unix, Cloud, …), a vytvoření prototypů napojení na tyto platformy.

Postupujte v následujících krocích:

1. Analyzujte a identifikujte požadavky z pohledu procesů, scheduleru i jednotlivých cílových platforem.
2. Navrhněte unifikované API mezi jádrem scheduleru a konektory na jednotlivá prostředí.
3. Po diskusi s vedoucím práce zvolte tří technologie (např. Hadoop, Oracle, AWS Cloud, …) a pro ně navrhněte a prototypově implementujte specifické konektory.
4. Navrhněte a formou prototypu implementujte API na straně scheduleru.
5. Zhodnoťte zkušenosti z návrhu a prototypové implementace.
6. Vyčíslete náklady na reálnou implementaci prototypu a pokuste se zhodnotit ekonomický přínos jednotného přístupu k realizaci plánovače.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 13. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Scheduler podporující multiplatformní prostředí

Ondřej Závodný

Katedra softwarového inženýrství
Vedoucí práce: RNDr. Ondřej Zýka

4. června 2020

Poděkování

Nejvíce bych chtěl poděkovat vedoucímu bakalářské práce RNDr. Ondřeji Zýkovi za jeho cenné rady a velmi profesionální vedení práce.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 4. června 2020 . . . . . .

Odkaz na tuto práci

Závodný, Ondřej. *Scheduler podporující multiplatformní prostředí.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato bakalářská práce se zabývá spouštěním a monitorováním úloh v kontextu plánovačů. Úlohy jsou ovládány z jazyka Java a mohou být spouštěny na více platformách. Je vytvořen návrh jednotného rozhraní, které umožňuje práci s těmito úlohami. Také jsou pomocí návrhu implementovány tři typy úloh, a to pro platformy Linux, Windows a Oracle. Dále jsou analyzovány náklady na implementaci a také jsou posouzeny přínosy jednotného centrálního plánování.

Klíčová slova vzdálené spuštění, enterprise plánovače, automatizace pracovních postupů, procesní řízení, multiplatformní plánování

Abstract

This bachelor's thesis is focused on executing and monitoring jobs in the context of schedulers. Jobs are managed with Java programming language and can be executed on multiple platforms. A unified interface that enables working with these jobs was created. Using this interface, three types of jobs were implemented for different platforms, Linux, Windows and Oracle. The costs of the implementation were evaluated, and the advantages and disadvantages of unified central planning were assessed.

Keywords remote execution, enterprise schedulers, workflow automation, bussiness process managemenet, multiplatform scheduling

Obsah

Seznam obrázků

Úvod

V moderním světě plném různorodých technologií a dostupného internetu vzniká nepřeberné množství dat, která je potřeba zpracovávat a ukládat s dosud nevídanou efektivitou. Mnoho firem buduje obrovské datové sklady, které jsou velmi náročné na údržbu, protože samotné uchovávání dat není ani zdaleka vše. Při práci s daty se řeší spousta opakujících se monotónních úloh, které je potřeba dělat pravidelně a jejich automatizace se stává nezbytnou součástí firemních procesů.

Tento problém nejlépe řeší plánovače. Ty za předem určených podmínek spouští jasně definované úlohy. Právě proto jsou velmi užitečné v procesním řízení, které se snaží dosáhnout maximální výkonnosti firemních procesů. Jejich vyžitím můžeme eliminovat nutnost tyto úlohy provádět manuálně. Tím se nejen ušetří čas zaměstnanců, kteří budou mít možnost věnovat se výdělečným činnostem v rámci firmy, ale také se sníží chybovost při výkonu těchto úloh, protože monotónní opakované činnosti vedou k časté chybovosti.

Téma je zpracováváno pro firmu Profinit EU, s.r.o., která má vlastní metodiku DATA_FRAME Concept. Ta určuje jak vytvářet a spravovat datové sklady a obdobná řešení efektivně, na základě ověřených vzorů a opakovatelného a pragmatického přístupu s maximálním využitím automatizace. Zároveň firma Profinit vyvíjí svůj nástroj DATA_FRAME Application pro podporu vývoje datových skladů, který je plně v souladu s metodikou DATA_FRAME Concept. Jeden z modulů této aplikace je plánovač DF_SCHEDULER, do kterého bude rozhraní zaintegrováno.

Samotná práce se zabývá návrhem rozhraní, které umožňuje plánovači jednotným způsobem řídit úlohy na různých typech vzdálených prostředí. Obecnost implementace rozhraní snadno umožňuje rozšíření o další připojení na rozmanitá prostředí. Jedním z cílů práce je také vyčíslit ekonomické přínosy při volbě jednotného přístupu. Rozhraní bude následně úzce zaintegrováno do modulu DF_SCHEDULER, ale tím už se tato práce nezabývá.

V první kapitole se nejdříve zabývám samotným procesním řízením a plá-

novači. Poté jsem si vybral tři konkrétní open-source plánovače, které dobře demonstrují, jak se plánovače implementují v praxi. Každý z vybraných plánovačů je vhodný pro odlišné účely a detailně zpracovává některé z požadavků. Na závěr kapitoly se věnuji implementaci vzdáleného spouštění úloh přes SSH, která slouží jako inspirace k formulaci obecného přístupu.

V druhé a třetí kapitole se věnuji návrhu a implementaci samotného rozhraní pro práci s úlohami. Popisuji použití tohoto rozhraní a stavový automat úlohy. Také prototypuji připojení na tři různá prostředí pro spouštění úloh.

V poslední kapitole se zaměřuji na měřitelné ukazatele centralizovaného plánování. Analzyuji náklady na implementaci a nasazení do firemního prostředí. Také se věnuji zhodnocení ekonomického přínosu.

Kapitola 1

Cíl práce

Hlavním cílem práce je navrhnout jednotné rozhraní plánovače pro spouštění a monitoring úloh na více prostředích. Pomocí tohoto rozhraní lze jednoduše implementovat připojení na jednotlivé platformy. Tři z těchto platforem jsou implementovány, konkrétně Linux prostřednictvím SSH, Oracle a Windows prostřednictvím Windows Remote Management.

Práce se také věnuje analýze plánovačů. Kromě obecného popisu jsou vybrány tři reprezentativní příklady open-source plánovačů. S hlavním cílem práce také úzce souvisí analýza programu Ansible, který umožňuje spouštění úloh na jednom typu vzdálených prostředí.

V neposlední řadě je cílem zpracovat ekonomické přínosy při využití jednotného centrálního přístupu k plánování v rámci procesního řízení. V rámci toho jsou také analyzovány náklady, které se pojí s implementací a nasazením.

Kapitola 2

Analýza

2.1 Analýza požadavků

V této části jsou uvedeny požadavky na rozhraní pro spouštění a monitorování úloh:

- Rozhraní umožní spustit úlohu s požadovanými parametry
- Rozhraní poskytne informace o stavu úlohy
- Rozhraní umožní úlohu v kterýkoliv moment zastavit
- Rozhraní zpracuje chyby, které nastanou
- Rozhraní umožní pracovat souběžně na více úlohách.
- Rozhraní bude snadno rozšiřitelné o další implementace

2.2 Co jsou plánovače

Plánovač je nástroj, který umožňuje automatizaci úloh. [1] Nejzákladnější formou plánovače je program, který v daný den a danou hodinu spustí nějakou úlohu. To se dá potom rozšiřovat o mnoho dalších funkcí. Pro příklad uvádím některé z nich:

- Různé formy uživatelských rozhraní
- Možnost vytvářet úlohy závislé na jiných úlohách
- Závislost úloh na nepředvídatelných událostech
- Zasílání oznámení a upozornění
- Konfigurace uživatelských práv

2.3 Procesní řízení

Pro pochopení role plánovačů ve firmě je nejprve potřeba vysvětlit jejich vztah k procesnímu řízení. K tomu využijeme definici Filipa Šmídy: *„Procesní řízení (management) představuje systémy, postupy, metody a nástroje trvalého zajištění maximální výkonnosti a neustálého zlepšování podnikových i mezipodnikových procesů, které vycházejí zjasně definované strategie organizace a jejichž cílem je naplnit stanovené strategické cíle.“* [2]

V rámci procesního řízení jsou ve firmě některé úlohy, které mají vždy stejný postup, a které jsou prováděny periodicky. Právě tyto úlohy jsou ideálním kandidátem na automatizaci pomocí plánovačů. Tato automatizace přináší několik výhod:

- Postup úlohy je jasně předepsán a pokaždé je aplikován stejně
- Vyloučí se lidské chyby dané monotónním opakováním stejných úloh
- Vyloučením lidské práce se zvýší produktivita
- Plánovač poskytuje přehled všech naplánovaných úloh

2.4 Analýza existujících řešení

V této sekci se věnuji analýze existujících řešení plánovačů. Pro účely práce byly vybrány tři implementace, které reprezentují některé z přístupů k problému plánování úloh.

2.4.1 Quartz Enterprise Scheduler

První z rozebíraných plánovačů je Quartz Enterprise Scheduler. Všechny informace o tomto plánovači jsem čerpal z knihy Quartz Job Scheduling Framework. [3]

Quartz Enterprise Scheduler je nejjednodušší z rozebíraných plánovačů. Jeho implementace je v Javě a neposkytuje žádné uživatelské rozhraní pro vytváření úloh, ale vše se píše přímo do zdrojového kódu.

2.4.1.1 Komponenty

Středobodem implementace je úloha (*Job*). Tou může být libovolná třída, která implementuje *JobInterface*. Každá úloha obsahuje metodu *execute*, která se zavolá v moment, kdy se má úloha spustit.

Další nedílnou součástí je spouštěč (*Trigger*). Spouštěče se plánovači předají spolu s úlohami a udávají, kdy se má daná úloha spustit. Nejjednodušším příkladem je předdefinovaný *CronTrigger*, který se spustí v zadaný čas. Jde vytvořit i spouštěč, který zaregistruje dokončení úlohy a na základě toho spustí další.

2.4.1.2 Další funkcionality

Velmi pokročilá funkcionalita programu Quartz Enterprise Scheduler je možnost vytvářet několikaserverové clustery. To je výhodné v moment, kdy máme velké množství spouštěčů a potřebujeme minimalizovat čekací dobu na spuštění jednotlivých úloh. Také to může být vhodné, pokud potřebujeme spustit více výpočetně náročných úloh najednou, protože lze využít výpočetního výkonu více strojů najednou.

2.4.2 JAZ (Job Arranger for Zabbix)

JAZ je plánovač, který je velmi úzce integrován do monitorovacího nástroje Zabbix. Umí skládat úlohy (*Jobs*) do jednoduchých diagramů (*Flowcharts*), ve kterých lze definovat typy úloh a závislosti mezi nimi. Diagramy jsou velmi jednoduchý způsob, jak na uživatelské úrovni definovat, které úlohy se spustí za kterých okolností, co se stane když některá z úloh skončí chybou, a další. Diagramy mohou být i vnořené. To umožňuje přehledně definovat velmi komplexní stromové struktury úloh.

2.4.2.1 Komponenty

Středobodem implementace je *Job Server*, který se stará o spouštění úloh a zpracovávání výsledků jejich běhu.

Uživatelské rozhraní je v komponentě *Job Manager*, která slouží pro nastavování jednotlivých úloh, zobrazování stavu a definici diagramů. *Job Manager* je nainstalován na počítači uživatele, který k plánovači přistupuje.

Pro přístup na vzdálené servery slouží *Job Agent*. Ten se nainstaluje na server, na kterém chceme spouštět úlohy plánovače. *Job Server* potom komunikuje přímo s tímto agentem. Agent je volitelná komponenta, protože plánovač umožňuje úlohy spouštět i v tkzv. agentless režimu, kdy použije pro připojení protokol SSH.

2.4.2.2 Další funkcionality

Nejdůležitější funkcionalitou je integrace do monitorovacího nástroje Zabbix, který umožňuje monitorovat další důležité části ekosystému, do kterého je plánovač nasazen. Výstupy z plánovače mohou být automaticky exportovány do Zabbixu a některé výstupy Zabbixu mohou sloužit v podmínkách v diagramech plánovače.

Plánovač umožňuje úlohy spouštět i paralelně. K tomu existuje značka v diagramu, která rozdělí průběh do dvou větví a každou z nich vykoná zvlášť. Mezi další značky, které lze použít při definici diagramu jsou například podmínka, cyklus, přenos souboru a další.

2.4.3 Schedulix

Schedulix je ze všech zde uvedených plánovačů zdaleka nejkomplexnější. Základním stavebním nástrojem je úloha (*Job*). U úloh se spolu s podmínkami definují počáteční a koncové stavy, které mohou být předávány i mezi navazujícími úlohami. Důležitou funkcionalitou je velmi podrobné nastavování práv k jednotlivým úlohám.

2.4.3.1 Komponenty

Centrem celého ekosystému je *Job Scheduling Server*, který se stará o spouštění úloh, logování a všechny logické funkce v rámci plánovače. Všechna data včetně konfigurace, logování a stavů úloh má uložené v relační databázi, do které se připojuje přes JDBC.

Na strojích, ke kterým Schedulix přistupuje, je nainstalován *Job Server Agent*, který podobně jako u *JAZ* komunikuje přímo se serverem a spouští jednotlivé úlohy.

Pro práci s plánovačem slouží webové uživatelské rozhraní (*Schedulix Web Frontend*) nebo nástroje příkazové řádky (*Schedulix Commandline Utilities*).

2.4.3.2 Další funkcionality

Nejrozsáhlejší funkcionalitou plánovače Schedulix takzvaný *Exit State Model*. Pomocí něho je možné univerzálním způsobem reagovat na skončení úlohy. Uživatel definuje model, který udává, jaká úloha se spustí na základě toho, v jakém stavu skončila předchozí úloha. Také je možné díky tomuto modelu předávat mezi jednotlivými úlohami informace.

Schedulix také obsahuje rozsáhlé API, pomocí kterého je možné přistupovat k stavům a výsledkům jednotlivých úloh. Toto API může být použito pro propsání údajů do různých jiných informačních systémů.

Jednou z dalších funkcionalit je také rozsáhlá možnost konfigurace uživatelských práv. Schedulix umožňuje vytvoření více uživatelských účtů a je možné každému uživateli nastavit, do jakých částí plánovače má přístup.

2.4.4 Shrnutí

Existuje mnoho různých úhlů pohledu na řešení plánování úloh a spoustu implementací plánovačů s různými funkcionalitami. Každá implementace je vhodná pro nasazení do různých prostředí a neexistuje univerzálně nejlepší řešení plánovače.

Jedna z funkcí, ve kterých se plánovače liší, je možnost reakce na různé události. Nejzákladnější událostí je časová, neboli spuštění úlohy v daný čas. Některé plánovače umožňují reagovat i na nepředvídatelné události, například vyvolané mimo samotný plánovač, ale není to nutností.

Velkým faktorem při volbě plánovače jsou také typy úloh, které plánovač umí spouštět. Cílem této práce je právě navrhnout rozhraní, které plánovači umožní spustit v podstatě libovolný typ úlohy, který lze implementovat v jazyce Java.

Dalším zásadním rozdílem mezi jednotlivými implementacemi plánovačů je přístup k uživatelskému rozhraní. Ty mohou mít různé úrovně komplexity, od jednoduchého konzolového rozhraní po složitý webový dashboard.

Plánovače se také liší svým přístupem k autentizaci. Existuje více způsobů, jak zajistit, aby data z plánovače byla zabezpečena proti neoprávněnému vniknutí. Také je potřeba určit, kdo bude mít přístup k definici jednotlivých úloh, komu bude umožněno úlohy spouštět, a kdo bude mít přístup k informacím o průběhu úloh.

Také je velmi důležité, jaké vazby mezi jednotlivými úlohami plánovač poskytuje. Například jestli je možné spustit navazující úlohu podmíněně, nebo jestli lze spustit více úloh paralelně a s další navazující úlohou počkat na dokončení všech paralelních běhů.

Některé plánovače také umožňují vytvářet cykly, kdy se jedna úloha spustí n-krát.

2.5 Vzdálené spouštění úloh pomocí Ansible

Práce z části vychází z nástroje Ansible, který umožňuje pomocí protokolu SSH automaticky konfigurovat a spouštět software na vzdálených serverech. Jeho jednoduchost, přehlednost a bezpečnost mohou jít příkladem i ostatním softwarům s podobným účelem.

2.5.1 Moduly

Největší inspirací pro tuto práci jsou takzvané moduly. Každý modul obsluhuje na vzdáleném serveru jednu službu. Příklady takového modulu jsou například kopírování souborů, ovládání systémových služeb, nebo konfigurace a spouštění služeb třetích stran. Obecná modulová architektura umožňuje snadným způsobem rozšiřovat program o další typy úloh. V této práci je při návrhu rozhraní řešen obdobný problém – vykonávání různých činností přes jednotné rozhraní.

2.5.2 Porovnání s navrhovaným rozhraním

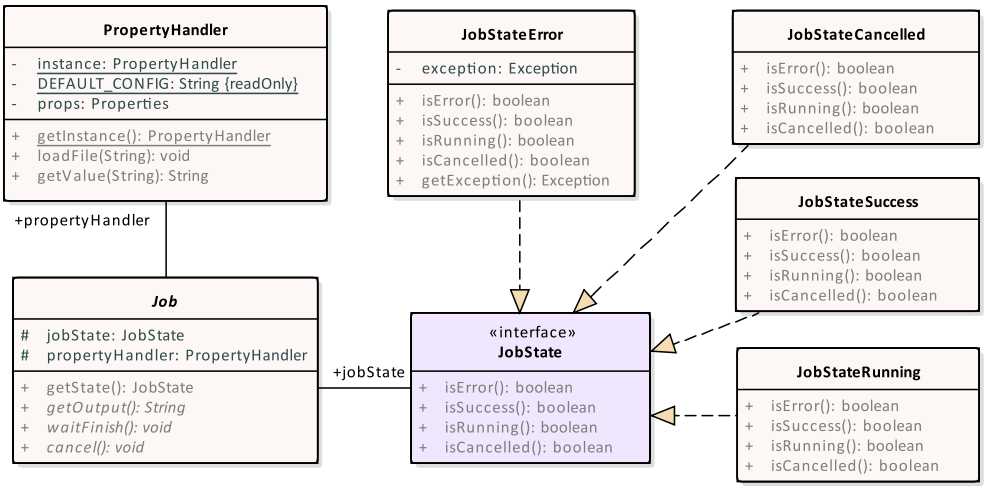
Na rozdíl od Ansible, který vše vykonává přes SSH a koncovým prostředím je v drtivé většině Linux, může být koncovým prostředím plánovaného rozhraní velké spektrum různých platforem.

Kapitola 3

Návrh

Tato kapitola se zabývá návrhem rozhraní v Javě, které jasně definuje metody pro spouštění a monitorování úloh na více prostředích. Rozhraní splňuje všechny požadavky vymezené v předchozí kapitole.

Jednotlivé třídy a jejich provázanost jsou vyobrazeny na diagramu tříd, obrázkek 3.1.



Obrázek 3.1: Diagram tříd, Job.

Základním stavebním kamenem je abstraktní třída *Job*. Ta obsahuje všechny požadované metody pro práci s úlohou.

3.1 Metody abstraktní třídy Job

První metodou třídy je její konstruktor. Ten u abstraktní třídy *Job* zařizuje, aby úloha měla přístup ke konfiguračním proměnným. Každá implementace,

která dědí tuto třídu musí mít svůj vlastní konstruktor, který úlohu spustí.

Další metoda *getState* je implementována jako getter pro instanční proměnnou *jobState*, ale může být implementací konkrétní úlohy nahrazena komplexnější logikou.

Třetí metodou je metoda *getOutput*, která vrací výstup úlohy ze vzdáleného prostředí v textové podobě. Pokud to vzdálené prostředí umožňuje, tak by u běžící úlohy měla metoda vracet i částečný výstup.

V neposlední řadě je metoda *waitFinish*. Ta má smysl pouze pro běžící úlohy, pokud úloha skončila, tak tato metoda neudělá nic. Pokud ale úloha běží, tak tato metoda zablokuje hlavní vlákno aplikace, dokud úloha neskončí. Metoda nemá žádnou návratovou hodnotu.

Poslední je metoda *cancel*. Ta má za úkol zastavit běžící úlohu. Pokud úloha neběží, tak metoda neudělá nic. Také nemá žádnou návratovou hodnotu.

3.2 Konfigurace

Rozhraní umožňuje dva způsoby předávání parametrů. Všechny parametry, které jsou nutné pro spuštění úlohy, musí být předány konstruktorem. Nepovinné parametry jsou uloženy v souboru *config.properties* a jsou úloze předány prostřednictvím třídy *PropertyHandler*.

PropertyHandler je singleton, což znamená, že v celé aplikaci může existovat pouze jedna instance této třídy. To je v tomto případě vhodné, protože není žádoucí, aby se soubor s konfiguračními parametry načítal vícekrát.

Implementace PropertyHandler využívá pro reprezentaci i načítání konfiguračních proměnných třídu přímo ze standardní Java knihovny, *Properties*. Tato třída umožňuje snadné načtení proměnných ze souboru a potom poskytuje metody pro vyhledání proměnné podle klíče.

Metoda *loadFile* třídy *PropertyHandler* přijímá jako argument cestu k souboru, který je umístěn ve složce resources v projektu. Z toho souboru jsou potom načteny proměnné. Metodu je možné zavolat vícekrát. Pokud se stejný klíč vyskytuje dvakrát, tak zůstane pouze poslední načtená hodnota. Pomocí metody *getValue* je potom možné najít hodnotu, která odpovídá klíči předanému jako argument.

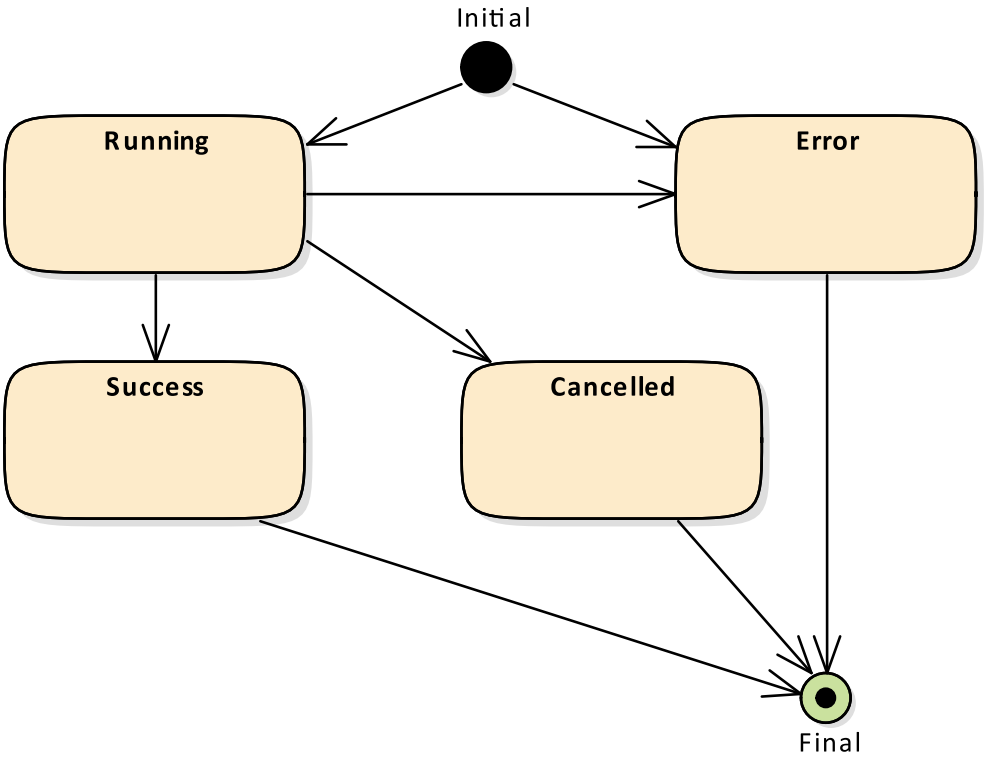
3.3 Stavový automat

Úloha může procházet celkem čtyřmi stavy. Přechody mezi nimi jsou znázorněny obrázkem 3.2. Spouštění úlohy začne zavoláním konstruktoru. Pokud během spouštění dojde k chybě, úloha se nespustí, ale rovnou přejde do stavu *Error*. Pokud se úloha úspěšně spustí, přejde do stavu *Running*.

Pokud je v průběhu zavolána metoda *cancel*, tak se úloha ukončí a přejde do stavu *Cancelled*. Když metoda zavolána není, tak úloha doběhne. Pokud

úloha skončila chybou, tak přejde do stavu *Error*, jinak přejde do stavu *Success*.

Pro úplnost je u chybového stavu také potřeba rozlišit, z jakého důvodu k chybovému stavu došlo. Proto má chybový stav navíc instanční proměnou *error* typu *Exception*, která obsahuje konkrétní výjimku, ke které při práci s úlohou došlo.



Obrázek 3.2: Stavový diagram, Job.

Kapitola 4

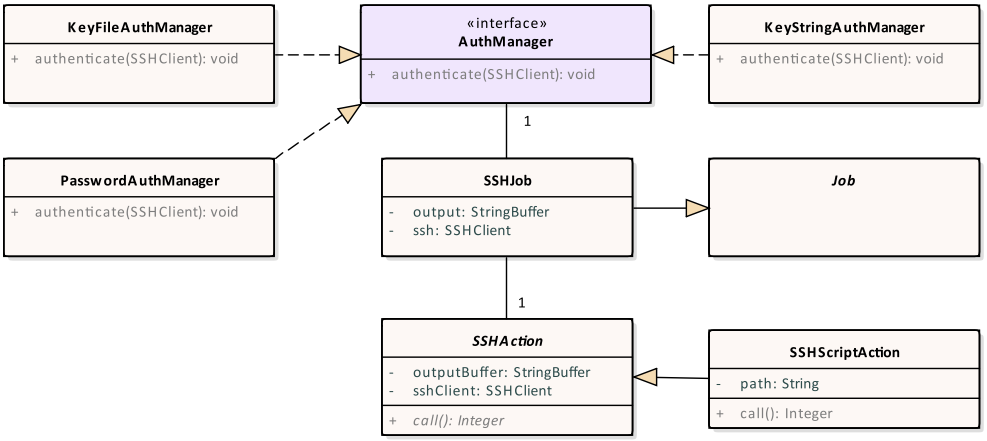
Prototypová implementace

Tato kapitola se věnuje prototypové implementaci navrženého rozhraní. Každá implementace dědí ze třídy *Job* a implementuje všechny její metody. Byly vybrány tři různá prostředí, která byla implementována.

4.1 Linux

Připojení na vzdálená prostředí s operačním systémem Linux je možné pomocí protokolu SSHv2. Na straně serveru je protokol implementován pod balíčkem OpenSSH. V Javě existuje více knihoven, které tento protokol implementují. V této práci je použita knihovna sshj, která podporuje velké množství aktuálně používaných šifer a algoritmů pro výměnu klíčů. [4] Oproti ostatním knihovnám má sshj lépe zdokumentované API.

Rozvržení tříd v prototypu napojení na Linux je popsáno obrázkem 4.1.



Obrázek 4.1: Diagram tříd, SSHJob.

4.1.1 Konfigurace

Všechny informace nutné pro spuštění úlohy, jsou předávány konstruktorem, který má tuto definici:

```
public SSHJob(String hostname,
              int port,
              AuthManager authManager,
              SSHAction action)
```

První dva parametry jsou *hostname* a *port*, které určují server, na který se bude připojení uskutečňovat. Dalšími parametry jsou *AuthManager* a *SSHAction*, které jsou rozebrány v dalších sekcích.

Úloha má také několik nepovinných parametrů, které jsou uloženy v konfiguračním souboru. Pokud tyto proměnné nejsou vyplněny, nebo soubor neexistuje, tak úloha pořád funguje.

job.ssh.script.prepend
: Tato proměnná umožňuje úloze předat příkaz, který se zavolá na vzdáleném prostředí před spuštěním samotného skriptu. Vhodné je například použití příkazu echo pro vypsání některých systémových proměnných.

job.ssh.append
: Obsah této proměnné se přidá jako poslední řádek do výstupu úlohy. Řetězec *%exitCode* bude nahrazen návratovým kódem posledního spuštěného příkazu.

4.1.2 Autentizace

Je potřeba zohlednit fakt, že SSH umožňuje více různých způsobů přihlášení. Každý způsob je implementován do vlastní třídy, která rozšiřuje rozhraní *AuthManager*. Stěžejní součástí tohoto rozhraní je metoda *authenticate*, která vypadá takto:

```
void authenticate(SSHClient client)
    throws IOException
```

Metoda dostane jako parametr objekt klienta z knihovny sshj a na něm zavolá metody pro autentizaci. Pokud se přihlášení na vzdálený server nepodaří, tak metoda *authenticate* vyhazuje výjimku. Celkem jsou v této práci implementovány tři různé metody přihlašování.

První a nejednodušší metoda přihlášení je implementována třídou *PasswordAuthManager*.

```
public PasswordAuthManager(String username,
                           String password)
```

Její konstruktor má dva parametry, uživatelské jméno a heslo.

Další možností pro přihlášení je třída *KeyStringAuthManager*.

```
public KeyStringAuthManager(String username,
                            String privKey,
                            String pubKey)
```

Ta v konstruktoru přijímá uživatelské jméno a dvojici klíčů v textové podobě.

Poslední způsob přihlášení je implementován třídou *KeyFileAuthManager*.

```
public KeyFileAuthManager(String username,
                          String location)

public KeyFileAuthManager(String username,
                          String location,
                          String passphrase)
```

Parametry jejího konstruktoru jsou uživatelské jméno a cesta k privátnímu klíči. Je zapotřebí, aby byl ve stejném adresáři jako privátní klíč umístěn i odpovídající veřejný klíč s příponou .pub. Konstruktor má navíc také nepovinný parametr, heslovou frázi k privátnímu klíči.

Instance jedné z této tříd je předána v konstruktoru třídě *SSHJob*, která se postará o zavolání metody *authenticate* i o zpracování případných výjimek.

4.1.3 Akce

Přes SSH je možné vykonávat více různých operací. Kromě spouštění nějakého příkazu je možné např. také překopírovat lokální soubor. Informace o tom, která operace se má vykonat, je předávána pomocí akce.

Základem je abstraktní třída *SSHAction*.

```
public abstract class SSHAction
    implements Callable<Integer> {

    protected SSHClient sshClient;
    protected StringBuffer outputBuilder;
}
```

Třída implementuje rozhraní *Callable<Integer>*, které umožňuje akci, aby byla spuštěna ve vedlejším vlákně. Rozhraní obsahuje metodu *call*. Návratovou hodnotou metody je návratový kód posledního spuštěného příkazu na vzdáleném serveru.

Třída obsahuje dvě instanční proměnné. První je instance objektu *SSHClient* z knihovny sshj. Druhá instanční proměnná je *output* typu *StringBuffer*, který slouží k uložení průběžného výstupu. Objekt *StringBuffer* je vláknově bezpečným, to znamená že nehrozí chyby vzniklé přístupem z více vláken najednou. [5]

V této práci je implementována jedna akce, *SSHScriptAction*.

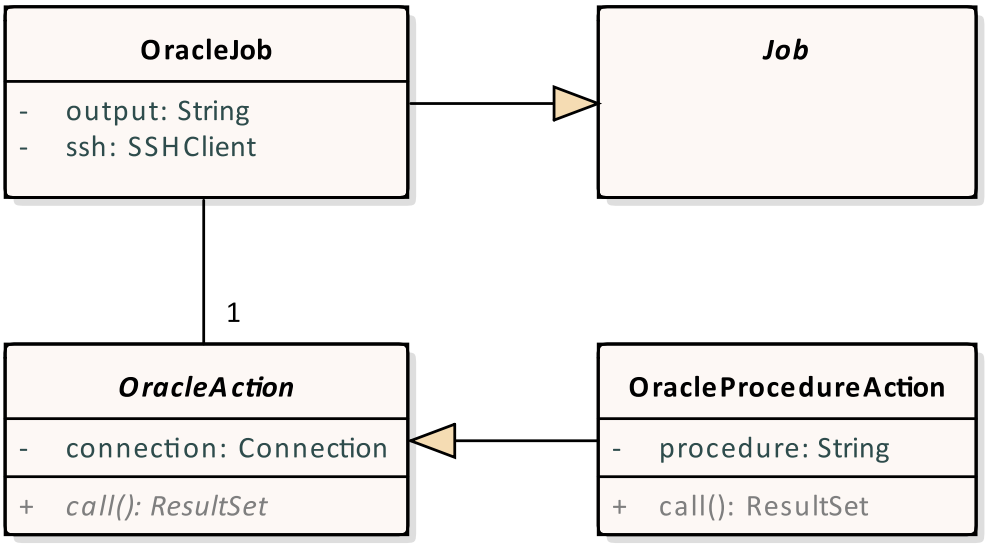
Ta slouží pro spuštění bashového skriptu, který je umístěn na vzdáleném serveru. Cesta k tomuto skriptu je akci předána konstruktorem. Před spuštěním skriptu je nejprve spuštěn příkaz, který je obsahem konfigurační proměnné *job.ssh.script.prepend*. Pokud tato proměnná není vyplněna, spustí se pouze samotný skript.

4.1.4 Výjimky

Většina chyb, které mohou během připojování na vzdálené prostředí nastat, je vyřešena v knihovně sshj a tyto výjimky jsou přímo uloženy do chybového stavu. Nicméně situace, kdy připojení i spuštění proběhlo úspěšně, ale skript skončil nenulovým návratovým kódem v knihovně sshj žádnou výjimku nevyhodí. Pro tento účel je vytvořena třída *ScriptException*. Součástí chybové zprávy je návratový kód skriptu.

4.2 Oracle

K připojení k Oracle databázím z Javy se standardně používá JDBC Driver. Ten vydává a udržuje přímo Oracle Corporation. V této práci je použita verze 8. Rozvržení tříd je reprezentováno obrázkem 4.2.



Obrázek 4.2: Diagram tříd, OracleJob

4.2.1 Konfigurace

Veškerá konfigurace úlohy je předávaná konstruktorem.

```
public OracleJob(String url,
```

```
        String user,
        String password,
        OracleAction action)
```

Prvním argumentem je databázové url, které udává k jaké databázi se úloha připojuje. Následující dva argumenty jsou uživatelské jméno a heslo k databázi. Posledním argumentem je rozhraní *OracleAction*. To určuje, jaká akce se nad databází bude vykonávat.

4.2.2 Akce

Podobně jako u připojení do Linuxu je také u Oracle vykonávání operací na vzdáleném prostředí implementováno abstraktní třídou *OracleAction*.

```
public abstract class OracleAction
    implements Callable<ResultSet> {

    Connection connection;
}
```

Třída má jednu instanční proměnnou, *Connection*. Tato proměnná reprezentuje spojení s databází. O správné přiřazení této proměnné se stará samotná úloha, *OracleJob*. Při implementaci konkrétní akce je nejdůležitější metoda *call*, která se při spuštění úlohy spustí v paralelním vlákně. Tato metoda vrací *ResultSet*, který slouží pro reprezentaci návratové hodnoty z databáze.

Třída *OracleAction* je rozšířena jednou implementovanou akcí, *OracleProcedureAction*. Ta dostane v konstruktoru název procedury, kterou nad databází spustí. Akce neumožňuje předávat proceduře žádné vstupní parametry, ani nedokáže získat žádné výstupní parametry, ale pozná, pokud došlo během vykonávání procedury k chybě.

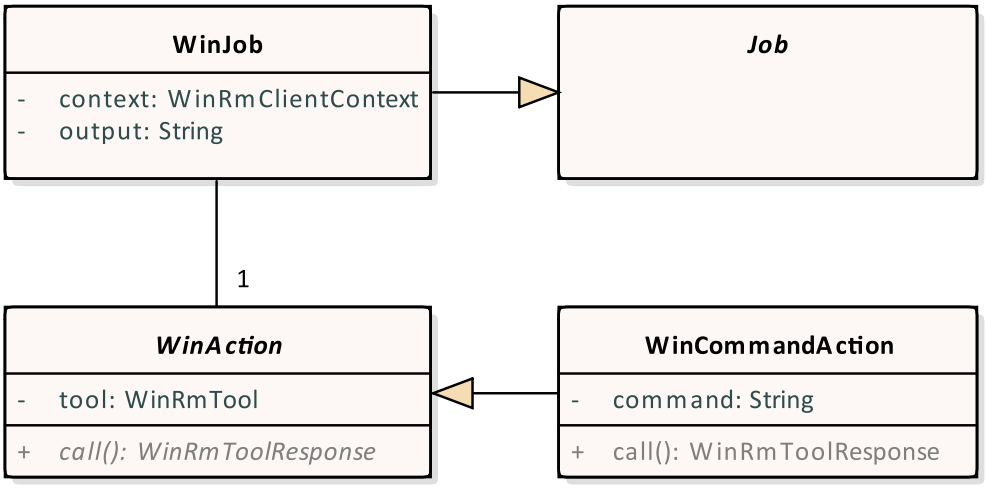
4.3 Windows

Připojení na operační systém Windows je uskutečněno pomocí službou Windows Remote Management, která je součástí standardní instalace operačního systému. Jedná se v podstatě o webový server, který umožňuje komunikaci pomocí SOAP protokolu. Na straně Javy byla pro vytvoření spojení použita knihovna winrm4j, která tento protokol implementuje.

Všechny třídy, které jsou součástí implementace jsou popsány obrázkem 4.3.

4.3.1 Konfigurace

Všechny konfigurační parametry, které jsou nutné pro spuštění úlohy jsou předávány konstruktorem. Ten vypadá takto:



Obrázek 4.3: Diagram tříd, WinJob

```
public class WinJob
    extends Job {

    public WinJob(String hostname,
                  int port,
                  String username,
                  String password,
                  String authScheme,
                  WinAction action)
}
```

První dva argumenty jsou *hostname* a *port*, které identifikují server, na který se úloha připojí. Parametry *username* a *password* slouží pro identifikaci uživatele, který bude použit pro přihlášení do systému. Parametr *authScheme* určuje, jakým způsobem se bude ke vzdálenému serveru přihlašovat. Hodnotou tohoto parametru může být kterákoliv z konstant ve třídě *org.apache.http.client.config.AuthSchemes*. Poslední parametr je *action*, který určuje jaká akce se na serveru vykoná.

Úloha má také některé nepovinné parametry, které jsou součástí konfiguračního souboru.

job.win.command.prepend
: Tento parametr umožňuje úloze předat příkaz, který se spustí před samotným příkazem úlohy.

job.win.append
: Obsah této proměnné se přidá jako poslední řádek do výstupu úlohy.

Řetězec *%exitCode* bude nahrazen návratovým kódem spuštěného příkazu.

job.win.use_https
Tento parametr udává, jestli se pro připojení použije šifrovaný protokol HTTPS, místo nešifrovaného HTTP.

4.3.2 Akce

Podobně jako u předchozích implementací je u připojení k operačnímu systému Windows možné předat informaci o tom, jaká akce se má v rámci úlohy vykonat. To je implementováno abstraktní třídou *WinAction*, která má následující definici:

```
public abstract class WinAction
    implements Callable<WinRmToolResponse> {

    protected WinRmTool tool;
}
```

Třída má jednu instančí proměnnou, která je typu *WinRmTool*. Tento typ je z knihovny winrm4j a reprezentuje připojení k vzdálenému serveru.

Každý potomek třídy WinAction také musí implementovat metodu call. Tato metoda vykoná akci na vzdáleném serveru a vrátí typ *WinRmToolResponse*, který slouží pro obalení návratové hodnoty ze serveru.

V této práci je implementována pouze jedna akce, *WinCommandAction*. Ta dostane jako argument konstruktoru příkaz, který na serveru spustí v programu PowerShell.

4.3.3 Výjimky

Většinu chyb, které při vykonávání příkazu mohou nastat, ošetřuje samotná knihovna winrm4j. Nicméně situace, kdy PowerShell příkaz skončil s nenulovým návratovým kódem, který standardně znamená, že došlo k chybě, není v této knihovně vyřešena. Proto byla v práci implementována výjimka *CommandException*, která slouží pro naznačení právě této situace. Součástí chybové hlášky je také návratový kód příkazu.

Kapitola 5

Testy

Důležitou součástí implementace prototypu je otestování jeho správného fungování. To je zajištěno pomocí unit testů. Každá implementace obsahuje testy v samostatné třídě, která rozšiřuje třídu *JobTest*. Testy se spouští přímo nástrojem Maven, pomocí příkazu *mvn test*.

5.1 Konfigurace

Pro konfiguraci testovacích proměnných je použitá stejná třída jako u úlohy, *PropertyHandler*. Konfigurační proměnné testovacího prostředí se nachází ve složce *src/test/resources* v souboru *test.properties*. Pro fungování testů je potřeba všechny konfigurační proměnné vyplnit příslušnými hodnotami.

5.2 Testovací prostředí

Tato sekce se věnuje popisu prostředí, které byly pro testování použity. Jelikož je testováno spouštění úloh na různých platformách, tak je potřeba nejen prostředí, ze kterého jsou úlohy spouštěny, ale také prostředí, ke kterým se úlohy připojují. První část této sekce se věnuje lokálnímu prostředí, zatímco druhá část sekce se věnuje vzdáleným prostředím.

5.2.1 Lokální prostředí

Testy byly spouštěny z operačního systému Windows 10 Enterprise LTSC, verze 1809. Na něm byla nainstalována Java, verze 1.8.0_222-2-ojdkbuild, a program Apache Maven, verze 3.6.2.

Program má také několik závislostí na knihovny třetích stran (winrm4j, sshj, ojdbc, junit, junit-jupiter, slf4j, jzlib). Ty jsou definovány pomocí POM (Project Object Model), což je reprezentace Maven projektu v souboru pom.xml.

Ten v sobě obsahuje informace o jednotlivých částech projektu a jeho závislostí. [6] U závislostí jsou uvedeny i jejich verze.

5.2.2 Vzdálená prostředí

5.2.2.1 Windows

Pro testování tohoto typu úloh byl použit operační systém Windows 10 Pro, verze 1909. Na počítači byl povolen Windows Remote Management, verze 3.0. Příkazy byly spouštěny v PowerShell, verze 5.1.18362.752.

5.2.2.2 Linux

Připojení na Linux bylo testováno na operačním systému Arch Linux ARM armv7l, s verzí kernelu 4.19.97-1-ARCH. Součástí instalace byl ssh server, verze OpenSSH_8.1p1.

5.2.2.3 Oracle

Jako databázový server byl použit Oracle Database 19c Enterprise Edition, verze 19.3.0.0.0.

5.3 Testovací scénáře

Tato sekce se věnuje konkrétním testovacím scénářům, které byly prováděny. Sekce je rozdělena podle jednotlivých prototypů. Pro správné provedení testů bylo potřeba splnit předpoklady, které jsou u každého prototypu popsány.

5.3.1 Windows

5.3.1.1 Předpoklady

Pro testování úloh na operačním systému Windows byl vytvořen testovací uživatel, kterému byl do domovského adresáře nahrán testovací skript, *test_-script.bat*. Ten se nachází v projektové struktuře ve složce *test/win*. Jeho obsah je následující:

```
for /l %%x in (1, 2, 3, 4, 5) do (
    echo %%x
    ping -n 1 127.0.0.1 > NUL
)
```

Skript pomocí cyklu for a příkazu *echo* vypíše čísla od jedné do pěti. Pomocí příkazu *ping* je vytvořena jednosekundová prodleva mezi vypsáním každého čísla.[7]

5.3.1.2 Scénáře

SuccessTest testuje základní funkčnost. Připojí pomocí uživatelského jména a hesla a spustí testovací skript.

InvalidPasswordTest testuje správné odchycení výjimky při zadání špatných přihlašovacích údajů tím, že se pokusí připojit pomocí neplatného hesla.

InvalidCommandTest testuje správné odchycení výjimky při zavolání neplatného příkazu na vzdáleném serveru.

CancelTest testuje, jestli jde běžící úloha zrušit.

5.3.2 Linux

5.3.2.1 Předpoklady

Při testování na operačním systému Linux byl vytvořen testovací uživatel. Tomuto uživateli byly do souboru *authorized_keys* přidány klíče *key.pub* a *key_passphrase.pub*, které jsou umístěny ve složce *test/ssh* v projektu. Tím bylo uživateli umožněno přihlašování privátními klíči umístěnými ve stejné složce. Zároveň byl uživateli do domovského adresáře nahrán skript *test_script.sh*, který se rovněž nachází ve složce *test/ssh*. Obsah tohoto skriptu je následující:

```
#!/bin/bash
for i in {1..5} ; do
  echo \$i
  sleep 1
done
```

Skript obsahuje stejnou funkcionalitu jako u testování operačního systému Windows. Vypíše čísla od jedné do pěti, s tím že mezi každým výpisem pomocí příkazu *sleep* počká jednu vteřinu.

5.3.2.2 Scénáře

PasswordTest testuje přihlášení uživatele pomocí přihlašovacího jména a hesla. Spustí testovací skript a počká na jeho dokončení.

KeyStringTest je stejný jako *PasswordTest*, pouze přihlášení probíhá pomocí dvojice RSA klíčů, které jsou v předány jako *String*.

KeyFileTest je stejný jako *KeyFileTest*, ale místo klíčů je předána cesta k privátnímu klíči.

KeyFilePassphraseTest je stejný jako *KeyFileTest*, ale je použit klíč s nastavenou heslovou frází.

InvalidPasswordTest testuje správné odchycení výjimky v případě pokusu o připojení s nesprávným heslem.

InvalidScriptTest testuje správné odchycení výjimky v případě zavolání s neplatnou cestou ke skriptu.

CancelTest testuje, jestli se úloha zruší při zavolání metody *cancel*.

5.3.3 Oracle

5.3.3.1 Předpoklady

Pro testování byl na testovacím databázovém serveru vytvořen testovací uživatel. Také byl na serveru spuštěn SQL skript *create_test_procedures.sql*, který má následující obsah:

```
create or replace procedure failure_procedure
as
begin
    raise_application_ERROR(-20002, 'Custom error');
end;
/

create or replace procedure success_procedure
as
begin
    sys.DBMS_SESSION.sleep(5);
end;
/
```

Skript obsahuje definici dvou testovacích procedur. První z nich je *success_procedure*, která slouží pro pozitivní testování. Jediný příkaz, který procedura obsahuje je *sys.DBMS_SESSION.sleep(5)*, který počká pět vteřin.

Druhá procedura je *failure_test*, která skončí chybou a slouží pro negativní testování. Pomocí této procedury se dá otestovat, jestli rozhraní správně zareaguje na vzniklou chybu.

5.3.3.2 Scénáře

SuccessTest otestuje, jestli připojení, přihlášení a spuštění testovací procedury proběhlo úspěšně.

FailureTest otestuje, jestli rozhraní při spuštění testovací procedury *failure_procedure* správně zareaguje na vzniklou chybu.

InvalidProcedureTest testuje správné odchycení výjimky v případě zavolání neexistující procedury.

InvalidPasswordTest testuje správné odchycení výjimky při použití neplatného hesla při přihlašování.

CancelTest testuje, jestli je možné úlohu zrušit zavoláním metody *cancel*.

Kapitola 6

Ekonomická analýza

Tato kapitola se věnuje ekonomické analýze a byla rozdělena na dvě části. V první části jsou popsány náklady na návrh rozhraní a implementaci prototypu. V druhé části jsou rozebrány výhody a nevýhody jednotného centrálního plánování.

6.1 Náklady

Tato sekce se věnuje popisu celkových nákladů na projekt. Náklady byly rozděleny do dvou kategorií, na implementaci a na údržbu. Implementace zahrnuje také návrh jednotného rozhraní. Údržba zahrnuje samotný provoz softwaru, ale také některé nepředvídatelné výdaje, které mohou v budoucnu vzniknout.

6.1.1 Náklady na implementaci

Náklady na implementaci přímo vyplývají z rozsahu daného projektu. Nejdříve se tato sekce věnuje jednomu z měřítek rozsahu, LOC (počet řádků zdrojového kódu). Dále se tato sekce věnuje rozdělení implementace na menší samostatné celky, u kterých se dá rozsah snadněji odhadnout.

6.1.1.1 LOC

Nejjednodušším měřítkem velikosti projektu je počet řádků zdrojového kódu. To sice není nejlepším měřítkem, jelikož závisí např. i na zvoleném programovacím jazyce, ale i tak poskytuje nějakou představu o rozsahu. [8] Následující tabulka znázorňuje počet řádků kódu jednotlivých částí:

Část	Počet řádků
Rozhraní Job	195
SSHJob	263
SSHJobTest	152
WinJob	154
WinJobTest	52
OracleJob	111
OracleJobTest	65

Z tabulky můžeme odhadnout, kolik řádků kódu budou mít případné implementace na další prostředí. Prototypová implementace neobsahuje všechny funkcionality pro nasazení do produkčního prostředí, takže celkový počet řádků neodpovídá přesně výslednému produktu.

Z řádků kódu se dá odhadnout kolik času programátor, který rozhraní implementuje, stráví jeho psaním. jelikož podle výzkumů programátor napíše přibližně mezi 325 a 750 řádky kódu za jeden měsíc. [9]

6.1.2 Rozdělení na samostatné funkcionality

Pro získání celkového přehledu o velikosti projektu je potřeba si ho rozdělit na jednotlivé funkcionality. Základní je rozdělení na tři části, jednotné rozhraní, prototypová implementace a testy. Tyto části se dále dělí na jednotlivé funkcionality, které se postupně implementují. Pomocí tohoto přehledu funkcionalit lze snadněji odhadnout celkové náklady na projekt.

6.1.2.1 Rozhraní

Hlavní součástí návrhu rozhraní je popsání metod, kterými lze úlohu ovládat. Také je potřeba zařídit, aby úloha dostala přístup ke všem konfiguračním proměnným projektu. K tomu je navrhnuta speciální třída.

Tato třída umožňuje načtení konfiguračních souborů do instančních proměnných. Také vystavuje metody, kterými lze k těmto proměnným přistupovat.

6.1.2.2 Implementace

V této části jsou popsány implementace jednotlivých typů úloh. Díky unifikovanému přístupu jsou tyto informace platné pro připojení na libovolnou platformu.

V první řadě je potřeba umožnit danou úlohu spustit. U toho se řeší několik problémů. Daná úloha může na prostředí spouštět různé typy akcí. Zároveň některé typy prostředí umožňují více různých způsobů přihlášení. Spuštění úlohy musí proběhnout na samostatném vlákně a je potřeba odchytit chyby, které při spuštění mohou vzniknout.

Úloha musí poskytovat aktuální informace o svém stavu. Pokud úloha skončila chybou, tak musí tato chyba být odchycena a informace o ní musí být dostupná přes rozhraní.

Implementace také musí umožnit zrušení aktuálně běžící úlohy.

6.1.2.3 Testy

Pro každou implementaci cílového prostředí je potřeba vytvořit testy, které pokryjí základní funkcionalitu. Základním testem je jednoduchý pozitivní test, který otestuje jestli úloha zafunguje.

Další je důležité otestování správného odchycení chyby. Na to by měly existovat minimálně dva testy. Jeden, u kterého se vůbec nepodaří připojit ke vzdálenému serveru, a druhý, u kterého se úloha úspěšně spustí, ale skončí chybou.

Poslední základní test otestuje, jestli je možné běžící úlohu zrušit.

6.1.3 Náklady na údržbu

Údržba začíná v moment, kdy je software nasazen do produkčního prostředí. Mezi hlavní činnosti údržby se většinou řadí tyto tři: [8]

Oprava chyb
: Náklady na opravu chyb jsou obecně velmi nepředvídatelné. Chyby mohou nastat v kterékoliv části programu. Je potřeba počítat s tím, že se i velmi závažná chyba může objevit opravdu kdykoliv.

Drobné úpravy a vylepšení
: Je potřeba počítat s tím, že se v produkčním prostředí narazí na nedostatky. Kromě vyřešení těchto nedostatků je také potřeba řešení znovu důkladně otestovat, aby se odhalily případné problémy, před nasazením do produkce. Samotné nasazení nové verze může přinášet komplikace, které se v nákladech promítnou.

Úpravy vynucené změnou prostředí
: Jednotné rozhraní umožní přístup k libovolnému prostředí, ale je nutné toto připojení implementovat. To přináší další náklady. A stejně jako drobné úpravy a vylepšení, pojí se i s těmito úpravami nutnost nasazení nové verze do produkčního prostředí.

Další složkou údržby je také správa serveru, na kterém řešení poběží. Existuje dvě možnosti, jak takový server provozovat. Buďto je možné využít cloudové služby, nebo zřídit vlastní fyzický server.

V případě fyzického serveru je v rámci nákladů důležitá pořizovací cena samotného serveru. Také je potřeba započítat životnost jednotlivých komponent, které je potřeba v případě rozbití vyměnit. U cloudového řešení není

potřeba řešit výpadky komponent, ale na druhou stranu se pojí s periodickým poplatkem za tyto služby.

V obou případech je potřeba spravovat operační systém, ve kterém řešení běží a vyřešit, jakým způsobem se budou nasazovat aktualizace.

6.2 Přínosy jednotného centrálního plánování

Tato kapitola se věnuje jednotlivým výhodám a nevýhodám jednotného centrálního plánování. Hlavním zdrojem informací pro tuto kapitolu byly konzultace s vedoucím práce.

6.2.1 Výhody centralizovaného přístupu

Důležitým předpokladem pro efektivní využití plánovače je jeho centralita. V ideálním případě by v celém ekosystému firmy byl jenom jeden centrální plánovač. To ale v podstatě není možné, protože většina softwaru ve firmě obsahuje nějaký způsob vlastního plánování. Nicméně pokud to je možné, tak by co největší množství úloh mělo být spouštěno z jednoho centrálního plánovače, a to hned z několika důvodů.

Zaměstnanci firmy mají kompletní přehled naplánovaných, ale také běžících a ukončených úloh. Pokud jsou informace o úlohách roztroušené po více informačních systémech, tak může být velmi obtížné zjistit jejich celkový stav. V tomto případě je potřeba vytvořit rozsáhlou dokumentaci, která popisuje provázanost úloh. Takováto dokumentace je velmi náročná na údržbu a je výrazně větší riziko chybovosti při jejím zanedbávání.

Díky centrálnímu přístupu je možné přehlednějším způsobem spravovat uživatelská práva. Díky tomu, že jsou všechna práva spravována z jednoho místa, tak se snižuje riziko nekonzistentních práv, než kdyby se pro každý informační systém řešily zvlášť.

Při nedodržení centrálního přístupu může dojít k duplicitnímu spouštění úloh, kdy je stejná úloha spuštěna ze dvou různých systémů naráz. To může vést k problémům s konzistencí dat. Takovéto chyby se v produkčním prostředí velmi špatně odhalují.

Dalším velkým přínosem centrálního přístupu je jeho průhlednost při hledání chyb. V případě, že někde dojde k chybě, tak je velká pravděpodobnost, že se informace o této chybě objeví přímo v centrálním plánovači. Pokud by bylo plánovačů více, tak se snižuje šance, že na chybu někdo narazí.

Využitím centralizovaného přístupu se také zjednodušuje administrace plánování v celé firmě. Každý zásah do firemního plánování se provádí v centrálním plánovači, takže odpadá nutnost učit se pracovat s více informačními systémy.

6.2.2 Nevýhody centralizovaného přístupu

Největší nevýhodou je nutná provázanost všech samostatných oddělení ve firmě. Pokud jednotlivá oddělení dostatečně nekomunikují mezi sebou, tak může dojít k problémům v centrálním plánovači, kdy dojde k podobné situaci, jako kdyby plánovačů bylo více. Každé oddělení spravuje jen jednu část a každá část je v podstatě samostatný plánovač, takže už se nejedná o centralizovaný přístup.

Další nevýhodou je zvýšená náročnost změnového procesu. Pokud dojde ke změně jednoho z řízených systému, tak je vždy potřeba provést zásah do centrálního plánovače. Tím se zvyšují náklady na změnu kterékoliv z platforem.

Jednou z nevýhod je také nutnost federalizovaného přístupu. Jelikož do plánovače potřebují přistupovat zaměstnanci napříč celou firmu, tak je nutné centrálně spravovat jejich přihlašovací údaje. Centrální správa údajů není jednoduché řešení a pokud ve firmě není implementováno, tak není možné nasadit centrální řešení plánovače.

6.2.3 Výhody jednotnosti

Jednotný přístup ke správě úloh má také svoje výhody. Při použití jednotného přístupu se snižují náklady na jednotlivá prostředí, jelikož není nutné spravovat pro každé prostředí samostatnou implementaci.

Také je výhodou, že při použití jednotného přístupu jsou sdíleny lidské zdroje a znalosti napříč více odděleními firmy, jelikož všechna oddělení používají stejný přístup k plánování úloh. Tím pádem není potřeba tolik lidských zdrojů na údržbu, jelikož je přístup k plánování v celé firmě stejný.

6.2.4 Nevýhody jednotnosti

Nevýhody jednotného přístupu žádné nejsou. Nicméně může nastat situace, kdy software neumožňuje použití centrálního plánovače a vyžaduje použití vlastního plánovacího řešení.

6.2.5 Shrnutí

Jednotný centrální přístup k plánování má svoje výhody, i svoje nevýhody. Pokud je centrální řešení implementováno správně, tak jeho výhody snadno převáží nevýhody a nasazení jednotného centrálního přístupu povede k celkově sníženým nákladům na plánování úloh ve firmě.

Závěr

Cílem práce bylo analyzovat, navrhnout, implementovat jednotné rozhraní pro spouštění a monitoring úloh na různých platformách a také zhodnotit ekonomický přínos jednotného přístupu.

První část práce je věnována analýze. V první řadě jsou vymezeny požadavky na vytvářené rozhraní a je přiblížena role plánovačů v procesním řízení. Jsou rozebrány některé existující plánovače a jejich funkcionality. Zároveň je analyzován software Ansible, který slouží k spouštění úloh na platformách Linux prostřednictvím SSH.

V druhé části je proveden návrh jednotného rozhraní pro spouštění a monitoring úloh, který odpovídá požadavkům vymezeným v analytické části. Pomocí tohoto návrhu je možné implementovat spouštění úloh na libovolném prostředí.

Třetí část se zabývá prototypovou implementací napojení na tři různá prostředí, kterými jsou Linux, Oracle a Windows. Napojení na prostředí Linux je uskutečněno pomocí SSH, které umožňuje spustit bashový skript. Implementace obsahuje několik způsobů autentizace. Napojení na Oracle umožňuje spustit nad databází libovolnou proceduru, a napojení na Windows se připojí pomocí Windows Remote Management a umožňuje interakci s příkazovou řádkou.

Poslední část analyzuje náklady na implementaci a údržbu vytvořeného řešení. Také se věnuje jednotlivým výhodám a nevýhodám jednotného centralizovaného přístupu. Bylo zjištěno, že výhody celkově převyšují nevýhody a že jednotný centrální přístup vede ke snížení nákladů.

Implementace splňuje požadavky vytyčené touto prací, ale existuje ještě prostor pro další zlepšení. Zřejmým rozšířením je implementace napojení na další platformy. Také je možné rozšířit možnosti konfigurace již existujících úloh a implementovat další akce. U napojení na Oracle by se mohla přidat podpora vstupních a výstupních parametrů.

Literatura

[1] Stamm, L.: What is Traditional Job Scheduling? And How did Enterprise Job Scheduling Evolve? [cit. 2020-03-07]. Dostupné z: `https://www.stonebranch.com/blog/what-is-job-scheduling/`

[2] Šmída, F.: *Zavádění a rozvoj procesního řízení ve firmě*. Praha 7: Grada Publishing a.s., 2007.

[3] Cavaness, C.: *Quartz Job Scheduling Framework*. Prentice Hall, 2006.

[4] Horn, M.: SSH implementation comparison. [online], [cit. 2020-01-05]. Dostupné z: `https://ssh-comparison.quendi.de/impls/sshj.html`

[5] Oracle: Class StringBuffer. [online], [cit. 2020-04-12]. Dostupné z: `https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html`

[6] The Apache Software Foundation: POM Reference. [online], [cit. 2020-01-20]. Dostupné z: `https://maven.apache.org/pom.html`

[7] Sheppard, S.: PING. [online], [cit. 2020-02-25]. Dostupné z: `https://ss64.com/nt/ping.html`

[8] Macinka, V.: *Techniky stanovení nákladů softwarových projektů*. Diplomová práce, Fakulta informatiky Masarykovy univerzity, Brno, 2009.

[9] Mahal, D.: The Programmer Productivity Paradox. [cit. 2020-03-15]. Dostupné z: `https://dzone.com/articles/programmer-productivity`

Příloha A

Seznam použitých zkratek

XML Extensible Markup Language

JDBC Java Database Connectivity

API Application Programming Interface

SSH Secure Shell

RSA Rivest–Shamir–Adleman

SOAP Simple Object Access Protocol

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

LOC Lines of Code

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `diagrams` ... zdrojová forma diagramů ve formátu Enterprise Architect Project
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF