ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

Тема: Пакетен мениджър за платформата Cloud Foundry

Дипломант:
Рангел Иванов

Научен ръководител:
инж. Димитър Дончев

СОФИЯ

2018

Темата на дипломната работа изискваше от Рангел още от самото начало да учи и опита на практика много нови за него концепции и технологии, за да може да ги използва в реализацията. Това включваше научаване на нов програмен език Go, работа с платформата Cloud Foundry, нови шаблони за програмиране, планиране и изпълнение на проект за изработка на продукт и др. Той анализира проблемната област, планира обхвата от функционалности, обема работа и сложността ѝ и спазвайки предварително уговорена итеративна инкрементална методология изработи продукта в очаквания срок и с очакваното качество. Добро впечатление остави начина му на работа и усилията, които положи през последните 3 месеца преди датата на предаване. Кодът, разработен от него спазва добри стандарти за качество и в него той е приложил успешно и правилно шаблони за програмиране. Рангел проявяваше интерес към процеса на разработване на софтуерни продукти, като приложи някои от практиките в реализацията на продукта. Писането на дипломната работа беше прекрасна възможност за него да вникне и опита реалистично производството на софтуер и Рангел я използва пълноценно!

Дипломанта се справи успешно не само с оформяне на концепцията и разработката на софтуерния продукт, но и със самата дипломна работа!

Научен ръководител:
инж. Димитър Дончев

Увод

В съвременните операционни системи пакетните мениджъри са незаменима част от потребителската практика. Настоящата дипломна работа ще представи реализацията на пакетен мениджър за платформата Cloud Foundry.

По дефиниция, пакетен мениджър, или система за управление на пакети, е съвкупност от софтуерни приложения, автоматизиращи процеса на инсталиране, обновяване, конфигуриране и премахване на софтуерни приложения по консистентен начин.

Пакетният мениджър работи с *пакети,* дистрибуции на софтуер и информация в архиви. Пакетите съдържат мета информация (данни, доставящи информация за други данни), като име на софтуера, описание на неговата цел, номер на версия, доставчик на софтуера, контролна проверка[1] и описание на зависимостите, нужни на приложението да работи. След инсталация, мета информация бива запазена в локална база данни за пакети. Пакетните мениджъри обикновено държат бази данни за софтуерните зависимости и данни за версиите, за да се предотврати несъответствие в приложенията и липсващи зависимости. Системите за управляване на пакети работят в съответствие със софтуерни хранилища[2], мениджъри на бинарни хранилища[3] и дигитални магазини за приложения[4].

Пакетните мениджъри имат за цел елиминиране на нуждата от ръчно инсталиране и обновяване на софтуер. Те са най-полезни при големи предприятия, чиито операционни системи са основани на Linux и други Unix-базирани системи, обикновено съдържащи се от стотици или дори десетки хиляди отделни софтуерни пакети.

Cloud Foundry[5] е платформа като услуга (Platform as a Service - PaaS[6]) с отворен код[7], определена от Фондацията Cloud Foundry. Софтуерът е разработен първоначално от VMware[8], преди да бъде прехвърлен на Pivotal Software[9], съвместно предприятие на EMC[10], VMware и General Electric[11].

Cloud Foundry използва продължително доставяне[12] на приложения, като поддържа пълния цикъл на разработване на приложение, от начална разработка, през всички тестови фази, до пускането в експлоатация. Архитектурата на CF, използваща контейнери поддържа приложения на всеки програмен език чрез множество доставчици на облачни услуги.

Целта на дипломната работа е:

- Да се реализира система за управление на зависимостите между облачни (Cloud Foundry) софтуерни приложения и автоматизиране на процеса на инсталация, обновяване, конфигуриране и премахване на програми в облачна система по консистентен начин.
- Възможност за достъпване на програмни библиотеки от различни по тип ресурси.
- Потребителски интерфейс, отговарящ на стандартите на Cloud Foundry.

Първа Глава

1.1 Обзор на съществуващи пакетни мениджъри

1.1.1 Пакетни мениджъри за Linux-базирани системи

1.1.1.1 Advanced Packaging Tool

Advanced Packaging Tool, или APT, е безплатен потребителски интерфейс, който използва библиотеки за инсталиране и премахване на софтуер на Debian и други Linux дистрибуции. APT опростява процеса на управление на софтуера на Unix-базирани системи чрез автоматизиране на достъпа и изтеглянето на различни ресурси, конфигурацията и инсталацията на софтуерни пакети от компилирани файлове или компилиране на програмен код.

APT е първоначално направен като потребителски интерфейс за dpkg[13], работейки с пакети с файлов формат ".deb" на Debian. В последствие е модифициран за работа и с RPM (т. 1.1.1.2) системата чрез APT-RPM[14]. Написан е на C++.

APT е съвкупност от средства дистрибутирани в пакета "apt". Съществена част от apt е дефинирана в C++ библиотека от функции; apt също включва конзолни програми (такива, използвани в Linux терминал) за работа с пакети, използващи библиотеката. Три от тези програми са apt, apt-get и apt-cache. Те са често използвани в примери на apt, защото са прости и са срещани навсякъде. Пакетът apt е важен за всички Debian дистрибуции, защото цикъла на живот на всички пакети в дистрибуцията се управляват през него и затова е включен в Debian инсталацията по подразбиране. APT може да бъде разглеждан като интерфейс на dpkg, по-удобен за потребителя от по-стария dselect. Докато dpkg изпълнява действия върху единични пакети, apt управлява взаимоотношенията (особено зависимостите) между тях.

Важно свойство на APT е начина, по който използва dpkg - той прави топологично сортиране[15] на пакетите за инсталация или премахване и извиква dpkg във възможно най-добрата последователност. В някои случаи използва “--force” опцията в dpkg.

1.1.1.2 RPM Package Manager

RPM Package Manager (оригинално Red Hat Package Manager) е система за управление на пакети. Името RPM се отнася до: файловия формат “.rpm”, файлове във формата “.rpm”, софтуер, пакетиран в такива файлове и самата програма за управляване на пакети. RPM е предназначен главно за Linux дистрибуции. Файловият формат е базовия пакетен формат за Стандартната Linux база[16] .

Първоначално направен за използване на Red Hat Linux[17], сега RPM се използва на много Linux дистрибуции. RPM е също и пренесен на други операционни системи, като Novell NetWare[18] и AIX[19] на IBM.

RPM пакет може да съдържа произволен брой файлове. По-голямата част от RPM файловете са в бинарен формат (BRPM), съдържайки компилираната версия на някакъв софтуер. Има също и “source RPM” (SRPM) файлове, които съдържат изходния код нужен за да се направи един пакет. Те имат съответна маркировка в header на файла, различавайки ги от нормалните (B)RPM файлове, което причинява тяхното извличане в “/usr/src” директорията при инсталация. SRPM файловете имат формата “.src.rpm” по конвенция (.spm при файлови системи с ограничение на файловите формати до 3 символа).

1.1.1.3 Yellowdog Updater, Modified

Yellowdog Updater, Modified (YUM) е конзолна система с отворен код за управляване на пакети за операционни системи Linux, използващи пакетния мениджър RPM. Съществуват различни приложения доставящи графични потребителски интерфейси за YUM, въпреки че има конзолен интерфейс.

YUM позволява автоматично обновяване, управляване на пакети и зависимости между тях на RPM-базирани дистрибуции. Подобно на APT за Debian, YUM работи със софтуерни хранилища, които могат да се достъпват локално или чрез връзка с интернет мрежа.

Зад потребителският интерфейс YUM зависи от RPM, който е пакетен стандарт в дигиталното дистрибутиране на софтуер, използващ MD5 хеш кодове[20] и digisigs[21] за да потвърди авторството и устойчивостта на софтуера. YUM е имплементиран като библиотеки за програмния език Python, с малък брой програми, доставящи конзолен интерфейс. Съществуват също и обвивки, базирани на GUI[22] като YUM Extender (yumex).

1.1.1.4 Dandified Packaging Tool

Dandified Packaging Tool, или dnf е следващото поколение на yum, пакетен мениджър за RPM-базирани дистрибуции. DNF е въведено с Fedora 18 и е било пакетния мениджър по подразбиране за Fedora от версия 22.

Възприеманите недостатъци на yum, с които DNF е направен да се справи са ниската производителност, използването на голямо количество памет и ниската скорост на неговото итеративно използване на взаимотношения между пакети[15]. DNF използва libsolv, външна система за управляване на взаимотношения между пакети.

DNF е първоначално написано на Python, но се полагат усилия да се пренапише на C и да се преместят повечето функционалности от Python кода към библиотеката libdnf.

1.1.1.5 ZYpp

ZYpp (или libzypp) е ядро за пакетни мениджъри, който захранва Linux приложения като YaST[23], Zypper и имплементацията на PackageKit[24] за openSUSE и SUSE Linux Enterprise[25]. За разлика от някои по-базови пакетни мениджъри, ZYpp предоставя "satisfiability solver"[26] за да изчисли и да се справи с пакетните зависимости. ZYpp е безплатен софтуерен проект с отворен код, спонсориран от Novell[27] и лицензиран под условията на GNU General Public License[28] версия 2 или по-късно. ZYpp е имплементиран главно на C++.

Zypper е конзолният интерфейс по подразбиране на пакетния мениджър ZYpp за инсталиране, премахване, обновяване и търсене на софтуерни пакети от локална или отдалечена медия. Графичният му еквивалент е модулът YaST[23]. Използва се в openSUSE от версия 10.2 beta1.

1.1.1.6 pacman

Пакетният мениджър pacman е една от главните отличителни характеристики на Arch Linux. Той комбинира бинарния пакетен формат с лесна за използване система на асемблиране[29]. Целта на pacman е да позволи лесно да се управляват пакети, независимо дали са от официални хранилища или от самия потребител.

Pacman поддържа системата актуална чрез синхронизиране на списъците с пакети с главния сървър. Този модел сървър-клиент също

позволява на потребителя да инсталира пакети с проста команда, заедно с всички нужни зависимости.

Pacman е написан на програмния език C и използва “.tar” формата за пакетиране.

Pacman е смятан за един от най-бързите системи за управляване на пакети.

1.1.2 Пакетни мениджъри за macOS

1.1.2.1 MacPorts

MacPorts, с предишното название DarwinPorts, е система за управление на пакети, която опростява инсталирането на софтуер за macOS и Darwin операционните системи. Проектът е с отворен код и е предназначен да опрости инсталацията на друг софтуер с отворен код. Подобно на Fink[30] и ports collections[31] на BSD, DarwinPorts стартира през 2002 г. като част от проектът OpenDarwin, с участието на няколко служители на Apple.

MacPorts позволява инсталацията на голям брой пакети чрез използване на командата `sudo port install <packagename>` в терминал, което ще изтегли, компилира, ако е нужно, и инсталира искания софтуер, докато, в същото време, се инсталират автоматично всякакви нужни зависимости. Инсталираните пакети могат да бъдат обновени чрез командата `sudo port upgrade outdated`.
На 28 Април 2005 г. проектът пуснал първата версия на софтуер си.

1.1.2.2 Homebrew

Homebrew е безплатна система за управление на пакети с отворен код, която опростява инсталирането на софтуер за macOS операционната система. Оригинално написан от Max Howell, пакетният мениджър е спечелил

популярност в Ruby on Rails[32] общността и е хвален за възможността си да бъде разширяван. Homebrew е препоръчван за лекотата си на употреба, както и за интеграцията си с конзола.

Homebrew е разчитало до голяма степен на GitHub[33], за да разшири разработката на някои пакети, чрез помощта на потребителски принос. През 2010 г. Homebrew е било третото най-сваляно софтуерно хранилище на GitHub. През 2012 г. то е имало най-голямото количество потребитески принос в GitHub. През 2013 г. е имало както и най-голям брой сътрудници, така и най-голям брой проблеми, решени от всеки проект на GitHub.

Homebrew е породило множество подпроекти като Linuxbrew, който е порт за Linux, Homebrew Cask, който разширява Homebrew и се фокусира върху инсталацията на GUI[22] приложения и специфични среди или програмни езици като PHP.

Написан е на Ruby, от Max Howell през 2009 г.

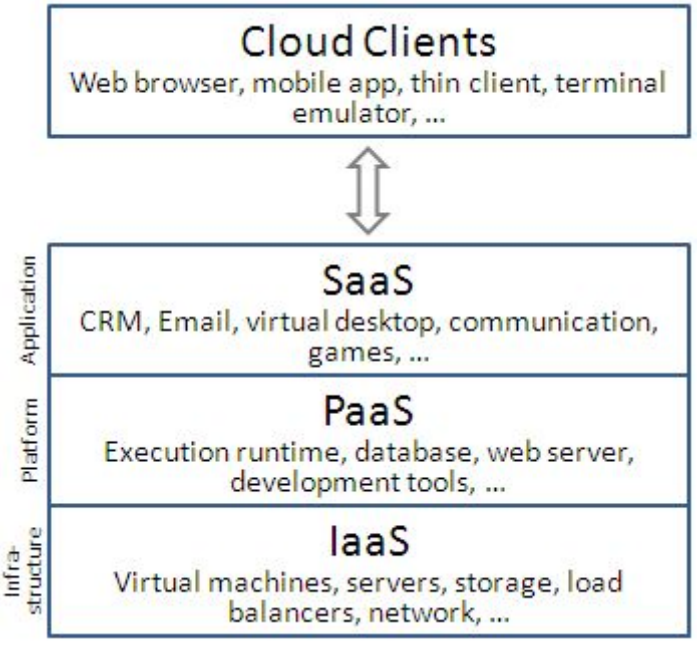
1.2 Аспекти на Cloud Foundry

Разгледаните системи предоставят управляване на пакети и приложения в локална среда. Те имат индивидуални предимства и недостатъци, но най-големия общ недостатък е, че не съществува пакетен мениджър за платформата Cloud Foundry.

За да се разберат част от проблемите, срещнати при реализацията на проекта, сега ще бъдат разгледани някои аспекти на моделите на услуги при облачното разработване и дистрибутиране на софтуер и онлайн платформата Cloud Foundry.

1.2.1 Модели на облачни услуги

Докато ориентираната към услуги архитектура предполага използването на "всичко като услуга" (Everything as a Service - EaaS), доставчиците на облачни изчислителни услуги предлагат техните продукти, спазвайки различни модели, от които трите стандартни модела дефинирани от NIST[34] са Инфраструктура като услуга (Infrastructure as a Service - IaaS), Платформа като услуга (Platform as a Service - PaaS) и Софтуер като услуга (Software as a Service - SaaS). Тези модели предлагат нарастваща абстракция на предлаганите услуги и ресурси и затова са често представяни като слоеве в архитектурата на софтуерни решения. Но не е задължително те да са свързани. Например, е възможно да се предостави SaaS имплементиран на физически машини без да се използват PaaS или IaaS, стоящи по-ниско в структурата, и обратно, може да се изпълнява и достъпва директно програма на IaaS без да бъде обвита като SaaS.



Фиг. 1.1 Моделите на услуги за cloud computing [35]

При IaaS, доставчик предоставя базовата инфраструктура за изчисление, съхранение на информация и достъпване на мрежа заедно с виртуализационния слой (hypervisor). След това потребителите създават виртуални машини, инсталират операционни системи, поддържат приложения и данни и конфигурират и управляват всичко свързано с тези операции.

При PaaS, доставчик предоставя по-голяма част от програмния стек от колкото IaaS доставчици, добавяйки операционни системи, бази данни и други среди за изпълнение (runtimes) в облачното пространство.

При SaaS, доставчик предоставя целият програмен стек. Потребителите само се регистрират и използват приложението, което работи напълно на инфраструктурата на доставчика.

Фиг. 1.2 Графично представяне на моделите на услуги [36]

1.2.1.1 Инфраструктура като услуга

Инфраструктура като услуга (IaaS) се отнася до услуги, доставящи програмни интерфейси от високо ниво, използвани за дереферириране на различни детайли на ниско ниво от мрежовата инфраструктура като физически компютърни ресурси, местоположение, разпределяне на данни, мащабиране, осигуряване на сигурност, елиминиране на загуба на данни, т.н. Hypervisor, като Xen[37], Oracle VirtualBox[38], Oracle VM[39], VMware

ESX/ESXi[40] или Hyper-V[41] осигуряват виртуалните машини. Множество от виртуализационни слоеве в облачната операционна система поддържат голям брой виртуални машини и имат способността да мащабират услугите в зависимост от нуждите на потребителите.

Linux контейнерите работят в изолирани дялове на едно Linux ядро, работещо на физическия хардуер. Базовите технологии на Linux ядрото “cgroups” и “namespaces” са използвани за изолиране, осигуряване на сигурността и управляване на контейнерите. Използването на контейнери предлага по-голяма производителност от виртуализация, защото няма hypervisor. Още една причина за използване е, че капацитета на контейнерите се мащабира автоматично с изчислителния товар, елиминирайки проблема на заделяне на прекалено големи изчислителни ресурси и позволява заплащане само за количеството използвани ресурси.

1.2.1.2 Платформа като услуга

При PaaS моделът, облачните доставчици доставят компютърна платформа, типично включваща операционна система, среда за разработване и изпълняване на приложения, база данни и уеб сървър. Разработчици на приложения имат възможност да разработват и тестват софтуерните си решения на облачна платформа без цената и сложността на купуване и управляване на хардуера и софтуера, на който работят. При някои PaaS предложения като Microsoft Azure[42], Oracle Cloud Platform[43] и Google App Engine[44] изчислителните и дискови ресурси се мащабират автоматично, за да напаснат нуждите на приложението. Това позволява на потребителя да не трябва да се грижи за ръчното управление на ресурсите.

Един PaaS доставчик създава и доставя гъвкава и оптимизирана среда, на която потребителя инсталира и използва приложения. Потребителите се

фокусират върху създаването и управлението на приложения, вместо да създават и поддържат инфраструктурата и услугите.

Създадени са много PaaS продукти за софтуерно разработване. Тези платформи предоставят инфраструктура за изчисление и съхранение на информация, както и редактиране на текст, управляване на версии, компилиране и тестване на услуги, които помагат на разработчици да създават нов софтуер по-бързо и ефикасно. Един PaaS продукт може също да позволи на разработващите екипи да си сътрудничат и да работят заедно, независимо от физическото им местоположение.

PaaS може да бъде доставяно по три начина: като публична облачна услуга от доставчик, където потребителя управлява изпълнението на софтуер с минимални конфигурационни опции, а доставчикът предоставя мрежите, услугите, дисковото пространство, операционните системи, средите за изпълнение, бази данни и други неща за да работи приложението на потребителя; като частна услуга (софтуер или хардуер, комбиниран със софтуер) в частна мрежа; или като софтуер, пуснат на публична инфраструктура като услуга.

1.2.1.3 Софтуер като услуга

При SaaS моделът, потребителите имат достъп до софтуер на приложения и бази данни. Облачните доставчици управляват инфраструктурата, платформата и самите приложения работещи на тях. Софтуер като услуга, понякога като софтуер при поискване, при възникнала необходимост ("on-demand software"), обикновено е заплащан при употреба или чрез абонаментна такса. В SaaS моделът, облачните доставчици инсталират и управляват софтуера, а потребителите достъпват този софтуер директно от облака. Потребителите не управляват облачната инфраструктура

и платформа, където работят приложенията. Това премахва нуждата да се инсталират и пускат приложенията на личните компютри на потребителите, което опростява поддръжката. Облачните приложения се различават от други приложения с тяхната мащабируемост - което може да бъде постигнато чрез клониране на задачи на множество виртуални машини по време на работа за да спазят нуждите на променящият се обем на работа. Изравнители на товара разпределят работата на поредица виртуални машини. Потребителят няма информация за този процес, като само вижда една точка за достъп. За да се справят с голям брой потребители, облачните приложения могат да са multitenant[45], означавайки че всяка машина може да поддържа повече от една организация на потребител.

SaaS премахва нуждата организациите да инсталират и пускат приложения на техните компютри или в техните информационни центрове. Това премахва разхода от хардуер, първоначалното инсталиране и поддръжка, както и лицензиране на софтуер, инсталация и поддръжка. Други предимства на модела SaaS включват:

Гъвкаво плащане: Вместо закупуването на софтуер за инсталиране или допълнителен хардуер, който трябва да се поддържа, клиентите се абонират за SaaS услуга. Обикновено, те плащат за тази услуга месечно, използвайки метода на заплащане при използване. Този метод на плащане позволява на много компании да предвиждат по-добре бюджета си. Потребителите могат и да прекратят използваните услуги по всяко време за да спрат разходите.

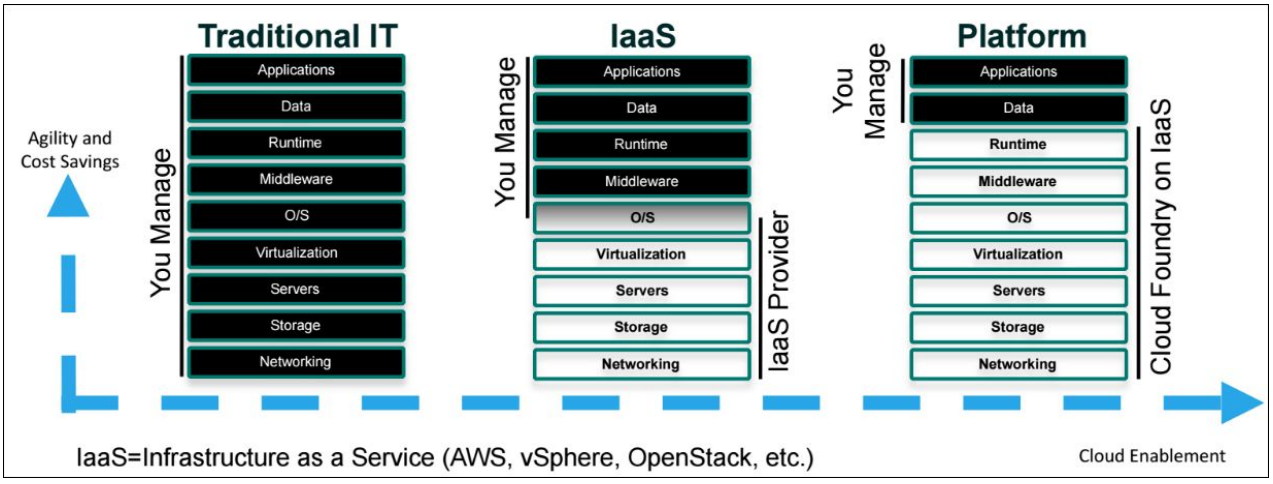
Мащабируемо използване: Облачните услуги като SaaS предлагат голяма мащабируемост, което позволява на потребителите да използват повече, или по-малко, услуги или функционалности при възникнала необходимост.

Автоматично обновяване: Вместо да закупуват нов софтуер, потребителите могат да разчитат на SaaS доставчик за автоматично обновяване и управляване на версиите.

Достъпност: Тъй като SaaS приложенията са предоставени през интернет, потребителите могат да ги досъпят от всяко устройство и местоположение с интернет връзка.

1.2.2 Общ преглед на Cloud Foundry

Облачните платформи позволяват на потребителите да разработват приложения и услуги, които да бъдат пускани в употреба за броени минути. Когато приложение стане популярно, платформата лесно го мащабира да поема повече трафик, изпълнявайки с няколко команди нещо, на което по старите начини с миграции би отнело месеци. Облачните платформи представляват следващата стъпка в развитието на интернет технолгиите, позволявайки на потребителите да се фокусират изцяло върху приложенията и данните си без да мислят за инфраструктурата, на която работят.



Фиг. 1.2 Причината да бъде използвано PaaS [46]

Не всички облачни платформи са еднакви. Някои имат ограничена поддръжка за програмни езици и помощни библиотеки, други са лишени от ключови услуги за приложения или ограничават пускането в експлоатация в

конкретен облак. Cloud Foundry (CF) се е превърнал в стандарт за индустрията. То е платформа с отворен код, която може да се използва за пускане на приложения на собствена компютърна инфраструктура или на IaaS като AWS[47], vSphere[48] или OpenStack[49]. Голяма общност работи по развитието и поддръжката на Cloud Foundry. Характеристиката на платформата да бъде с отворен код и разшируема позволява на потребителите ѝ да използват всякакви допълнителни библиотеки, програмни услуги или облачен доставчик. Cloud Foundry е най-добрият избор за хора, интересуващи се от премахване на цената и сложността от конфигуриране на инфраструктурата за приложенията си. Разработчиците могат да стартират и развиват приложенията си на Cloud Foundry, използвайки съществуващите развойни среди без никаква модификация по техния код.

"Buildpacks" доставят библиотеки и среда за изпълнение на приложения. Те типично анализират приложенията за да определят какви зависимости са нужни за инсталиране и как да се конфигурират тези приложения за да си комуникират със свързани услуги.

Когато едно приложение се качи на Cloud Foundry, платформата автоматично избира подходящия buildpack за него. Този buildpack се използва за компилиране, пакетиране и подготовка за пускане на приложението.

Инстанциите на платформата обикновено имат ограничен достъп до зависимости. Това ограничение става, когато те са в частни мрежи или когато администраторите използват локални сървъри. В тези случаи, Cloud Foundry осигурява приложение Buildpack Packager.

Втора Глава

2.1 Функционални изисквания

Изискванията към програмния продукт са такива, каквито се очакват от всеки един качествен пакетен мениджър. Тези изисквания включват напълно автоматизирано изпълнение на операции върху приложения/пакети, консистентно репродуцируеми резултати, значителна разлика във времето на изпълнение на някоя операция, подържана от пакетен мениджър (инсталиране на приложение, обновяване, изтриване, т.н.) в сравнение с ръчното ѝ изпълнение, минимална до никаква възможност от допускане на грешки при операции и възможност за конфигуриране на налични приложения/пакети.

Автоматизираното изпълнение на операции върху приложения/пакети предполага изпълнението на пълния набор от събития (операции), нужни за инсталирането, обновяването и изтриването на приложение. Този набор започва от преглеждането на софтуерните хранилища, съдържащи пакетите, искани за инсталация/обновяване, техните зависимости и други конфигурационни файлове, нужни за правилното функциониране на инсталираното приложение. След намирането на всички файлове, нужни за изпълнение на операцията, пакетните мениджъри изтеглят тези файлове от съответните хранилища, компилират ги, ако не са изпълними файлове, и започват процеса по инсталиране/обновяване. Конфигурирането на приложенията става по различно време при различните пакетни мениджъри. При представеният програмен продукт, това става по време на инсталация. При някои съществуващи решения - по желание на потребителя, във време, след инсталацията. Изтриването на приложения/пакети е по-прост процес и предполага по-малък брой операции, нужни за автоматизиране от пакетния мениджър. Тези операции включват валидиране за съществуващо

приложение, намиране на неговите файлове във файловата система или на мястото, където е инсталирано, и изтриването им.

Консистентно репродуцируеми резултати предполага един пакетен мениджър да предоставя едни и същи резултати при еднакви входни данни. Независимо от броят пъти, който една или няколко операции бъдат извикани от потребителя, системата трябва да продуцира винаги консистентни резултати. Провал при представяне на едни и същи резултати може да бъде породен от системни грешки, неправилна конфигурация или некачествено реализирана система.

Значително по-малко време за изпълнение на операции, сравнено с извършването им ръчно, предполага процесите, които човек би могъл да изпълни за една единица време, системата да изпълни за една десета, стотна (и т.н.) от времето. Всичките отделни гореспоменати операции при инсталиране, обновяване и изтриване на пакети/приложения отнемат различни периоди от време и имат вариращ шанс за грешка при ръчно изпълнение. Освен премахването (или намаляването) на коефициента на грешка, разликата във времето, което отнема на човек и на машина да изпълнят дори една команда е значителна. Натрупването на тази разлика води до едно от главните предимства на пакетните мениджъри и именно главното, разглеждано тук.

2.2 Избор на технологии

2.2.1 Java SE

Java Standard Edition е спецификация за език и библиотеки, както и платформа (езиков интерпретатор и имплементация на библиотеки). Езикът и платформата Java са разработени в началото на 90’ от компанията Sun Microsystems[50]. Първоначално предназначена за мобилни устройства, тя се оказва изключително гъвкава и се използва с голям спектър на приложение. Основна черта и качество на платформата и езика е, че тя е интерпретаторна.

Това означава, че потребителският код, който бива изпълнен, не е компилиран до машинен код, а до език от по-високо ниво (byte code), който по време на изпълнение бива превеждан на машинен език, в зависимост от архитектурата на която бива изпълнен. По този начин се постига едно от най-големите предимства на Java, а именно възможността един и същ код да бъде изпълнен върху различни по архитектура машини. Други характерни черти на езика са, че е обектно ориентиран. Предоставя автоматично управление на паметта. Той има изключително гъвкав синтаксис, позволяващ описването на сложни алгоритми. Синтаксисът до голяма степен е взаимстван от езика C++, с малки промени и ограничения. Езикът е строго типизиран и предразполага към писане на обектно ориентиран код със стриктни програмни интерфейси. Изходният код се компилира до byte code, което позволява извършване на оптимизации. В настоящето Java намира приложение от микро устройства, роботи, подвижни устройства и настолни компютри до корпоративни системи обработващи големи обеми от данни и приложения с мрежови интерфейс.

Причините да бъде избран за дипломната работа са, че е доказано най-използваният език за разработка на облачни приложения, съвместим е с различни платформи, има голяма общност, която не спира да го развива и е популярна технология за разработка на софтуер с отворен код.

2.2.2 Maven

Maven е система за автоматизация на асемблирането главно на Java проекти. Maven адресира два аспекта на асемблиране на софтуер: описване как софтуер се асемблира и описание на зависимостите му. За разлика от по-ранни системи като Apache Ant[51], Maven използва конвенции за процедурата по асемблиране и само изключения са нужни да бъдат написани. Един XML[52] файл описва софтуерният проект, който се асемблира, зависимостите му - външни модули и компоненти, реда на асемблиране,

файловата структура и нужните приставки. Maven има предварително определени цели за изпълняване на предефинирани задачи като компилацията на код и пакетирането му.

Maven динамично изтегля приставки и Java библиотеки от едно или повече хранилища, като Maven 2 Central Repository[53], и ги запазва в локалната кеш памет. Тези изтеглени артефакти (така се наричат библиотеките, изтеглени от Maven) в локалният кеш могат да бъдат обновени с артефакти, направени от локални проекти.

Maven може да се използва за асемблиране и управляване на проекти, написани на C#, Ruby, Scala и други езици. Проектът Maven е част от Apache Software Foundation[54], което преди е било част от Jakarta Project[55]. Системата използва архтектура, базирана на приставки, позволявайки да се използва от всяко приложение, контролируемо чрез стандартния вход. На теория, това би позволило на всеки да напише приставки за да си взаимодейства с инструменти за асемблиране (като компилатори) за всеки програмен език. В действителност, поддръжката и използването на езици, различни от Java е минимално. Понастоящем съществуват приставки за .NET[56] и C/C++, поддържани за Maven 2.

Алтернативни технологии като Gradle[57] и sbt[58] не използват XML, но се придържат към ключовите концепции, въведени от Maven. При Apache Ivy[59] е разработена система за управление на зависимостите ексклузивно за него, което поддържа и Maven хранилища.

Системата Maven е избрана, защото е една от най-често използваните системи за асемблиране на Java проекти.

2.2.3 Уеб услуга REST чрез Jersey (Jax-RS)

Representational State Transfer (REST) е архитектурен стил, специфицирайки ограничения, като уеднаквен интерфейс, който при

прилагане в уеб услуга доставя желани свойства, като висока производителност, мащабируемост и способността да бъде модифициран. В архитектурният стил REST данни и функционалностти са приемани като ресурси и са достъпвани чрез Uniform Resource Identifiers (URIs), обикновено линкове в интернет. Тези ресурси са обработвани чрез използването на прости, добре дефинирани операции. Архитектурният стил REST ограничава една архитектура до тип клиент-сървър и е направен да използва комуникационен протокол без състояние (stateless), обикновено HTTP. При този стил клиентите и сървърите си обменят репрезентации на ресурси, използвайки стандартизиран интерфейс и протокол. Тези принципи помагат на REST приложенията да са прости, леки за поддръжка и да имат висока производителност.

Уеб услугите REST обикновено имплементират четирите главни HTTP метода към операциите, които изпълняват: създаване, вземане на информация, обновяване, изтриване. Следната таблица показва съпоставката на HTTP методите с операциите, които изпълняват.

HTTP Метод	**Изпълнени операции**
GET	Вземане на ресурс
POST	Създаване на ресурс (или други операции)
PUT	Създаване или обновяване на ресурс
DELETE	Изтриване на ресурс

Фиг. 2.1 REST операциите съответстващи с HTTP заявките

Jersey RESTful Web Services е имплементация на стандарта за разработване на уеб услуги REST на Java с отворен код.

С цел да се опрости разработването на REST уеб услуги и клиенти на Java е направен стандартен JAX-RS програмен интерфейс. Jersey изпълнява

ролята на референтна имплементация на JAX-RS (JSR 311[60] и JSR 339[61]). Освен това, Jersey има собствен програмен интерфейс, който разширява функционалността на JAX-RS с допълнителни функции за още повече опростяване на разработването на REST услуги и клиенти.

Избрана е, защото е най-използваната библиотека за REST уеб услуги за Java.

2.2.4 JUnit u Mockito

JUnit е библиотека за тестване за програмният език Java. JUnit е важен в разработването на приложения, използвайки Test-driven Development (TDD), и е една от библиотеките, образуващи xUnit. Библиотеката се намира в пакетът junit.framework за JUnit 3.8 и по-ранни версии, и в org.junit за JUnit 4 и по-късни. Тя е най-често използваната библиотека за Java тестове.

Mockito е библиотека за тестване с отворен код за Java, лицензиран под MIT лицензa. Библиотеката позволява създаването на заменящи (mock) обекти (такива, наподобяващи поведението на обектите, които се тестват) в автоматизирани тестове. Използва се при Test-driven Development или Behavior Driven Development (BDD).

В разработването на софтуер има възможност за осигуряването на очакваното поведение на обектите. Един начин това да се постигне е да се разработи библиотека за автоматизация на тестове, която тества всяко поведение и проверява дали се изпълнява както трябва, дори след като е променено. Условията за разработване на цяла библиотека за тестване, обаче, са обикновено същите, каквито при разработването на самите обекти, които трябва да бъдат тествани. Поради тази причина, разработчиците са създали библиотеки за тестване със заменящи обекти. Те заменят някои външни зависимости, така че тестваният обект да има консистентно взаимодействие

със зависимостите си. Mockito има за цел да рационализира използването на тези външни зависимости, които не подлежат на тестване.

При разработването на JUnit тестове в големи проекти се налага да се подменят функционалности, от които тестваният код зависи. Това може да се направи като за тестовите цели се имплементират тези функционалностти, така че да се изолира тестваният код от друг програмен код, на който зависи.

2.2.5 Go

Go, често наричан “golang”, е програмен език създаден от Google през 2009 г. Езикът се компилира и е статично типизиран като C, има функции като асинхронно освобождаване на памет, безопасно управление на паметта и функционалност за асинхронно изпълнение на задачи в стил CSP (Communicating sequential processes - комуникиращи си последователни процеси). Компилаторът и всички стандартни библиотеки първоначанлно разработени от Google са безплатни и с отворен код.

Go има сходности със C, но има много промени за подобряване на простотата на синтаксиса и безопасността.

Go се състои от:

- Синтаксис и среда, взаимстващи шаблони, по-често срещани в динамичните езици за програмиране:
    - Съкратена декларация и инициализация на променливи чрез подразбиране на тип -

      `x := 0` вместо `int x = 0;` или `var x = 0;`
    - Кратко време за компилация.
    - Локално управление на пакети (чрез командата “go get”) и онлайн документация на пакетите.

- Характерни решения на специфични проблеми:
    - Вградени примитиви за асинхронно програмиране: олекотени процеси (goroutine), канали (channel - елементите, свързващи успоредно изпълняващите се олекотени процеси) и израз “select” (позволява изчакването на няколко операции от канали).
    - Начин на компилиране, който, по подразбиране, произвежда статично свързани изпълними файлове без външни зависимости.
- Желанието за спецификациите на езика да са достатъчно прости да се запомнят лесно от програмистите, от части чрез пропускането на функционалности, често срещани в подобни програмни езици.

Приставките за командният интерфейс на Cloud Foundry се пишат на Go. Това е основната причина за избор на езика като част от технологийте, използвани в настоящата дипломна работа.

2.3 Ориентирана към услуги архитектура

Ориентираната към услуги архитектура (Service-oriented architecture - SOA) е начин за асемблиране на софтуерни приложения, инженерни и бизнес процеси чрез свързване на софтуерни услуги. Аритектурата бива разработвана, използвайки традиционни езици като Java, C, C++, C#, Python, Ruby или PHP. Ориентираната към услуги архитектура представлява начин на дизайн на софтуер, при който услуги се предоставят на други компоненти, чрез портове за комуникация в мрежата. Основният принцип на SOA е независимост от конкретен доставчик, продукт или технология. Услугата е малка част, предоставяща функционалност, която може да се достъпи от разстояние, да се използва и да се обновява независимо от останалите.

Една услуга има четири свойства според една от многото дефиниции на SOA:

1. Услугата представлява бизнес активност със специален изход/резултат.
2. Услугата е самостоятелна, самосъдържаща се.
3. Услугата е черна кутия за потребителите.
4. Услугата може да се състои от други, скрити отдолу, услуги.

Различни услуги могат да се използват заедно, за да се предостави функционалност на голямо софтуерно приложение. Ориентираната към услуги архитектура е свързана не толкова с правенето на модуларно приложение, колкото с това, как да се състави приложение чрез интеграция на отдалечени, отделно поддържащи се и отделно инсталирани компоненти. Това се предлага от технологии и стандарти, които правят комуникацията между отделните компоненти в мрежата лесна, особено в IP адресирана мрежа.

В Ориентираната към услуги архитектура, услугите използват протоколи, които описват как те предават и оформят съобщения, използващи описателни метаданни. Тези метаданни описват както функционалните характеристики на услугата, така и качествата на услугата. Целта на архитектурата е да позволи на потребителите да комбинират големи парчета от функционалности за да оформят приложения, които са избрани части и само от вече съществуващи услуги и комбинирайки ги по специален начин.

Понятието "ориентирани към услуги" (service-orientation) цели малка свързаност (loose coupling) между услугите. SOA разделя функционалностите на отдалечени елементи, или услуги, които разработчиците ги правят достъпни в мрежата за да позволят на ползвателите на услугите да ги комбинират и преизползват, разработвайки приложенията. Тези услуги и техните потребители комуникират един с друг като предават данни в добре

структуриран и разпространен формат, или като координират активностите между две или повече услуги.

Всяка SOA може да играе една от следните три роли в изграждането на приложението:

Доставчик на услуги

Той създава уеб услугата и предоставя информацията свързана с услугата на регистъра с услуги. Всеки доставчик има несигурности, и се дебатира върху многото "защо" и "как", коя услуга да се предостави за използване и на коя да се даде по голяма важност: сигурност или лесна достъпност, на каква цена да се предоставя услугата и още много. Доставчикът на услугата също трябва да реши в коя категория попада услугата в списъка на конкретните брокери на услуги, а също и какви са условията за ползване на услугата.

Брокер на услуги, регистър на услуги или хранилище на услуги

Техните функции са свързани с това да направят информацията (без значение от услугата) на разположение за всеки потенциален потребител. Този, който имплементира брокера решава и какъв да е обхвата на този брокер. Публичните брокери са на разположение навсякъде, а частните брокери са ограничени само до определен брой потребители. UDDI[62] е ранният, но вече не поддържан опит да се предостави откриване на уеб услуги (Web services discovery).

Потребител на услуги

Той локализира услугите в регистъра на брокерите, използвайки голям набор от търсещи операции и след това ги свързва с доставчика на услугите, за да може да извика някоя от неговите уеб услуги. Която и услуга да търсят

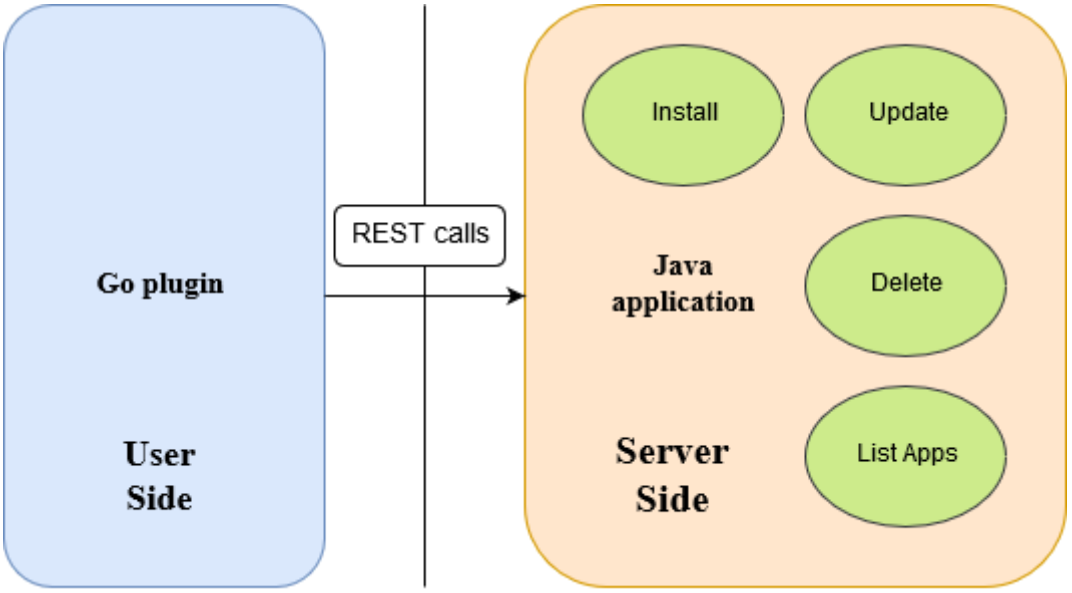
потребителите, те първо трябва да я потърсят при брокерите, да се свържат с конкретната услуга и след това да я използват. Те могат да достъпват множество услуги, стига доставчика да предоставя повече от една услуга.

Връзката между потребител и доставчик се определя от договор на услуги (service contract), който има бизнес част, функционална част и техническа част.

Трета Глава

3.1 Програмна реализация на Cloud Foundry базиран пакетен мениджър

Реализацията на пакетния мениджър за платформата Cloud Foundry съдържа приставка за CF написана на Go, която е потребителският интерфейс и Java приложение, обработващо заявките на потребителя и извършващо операциите с файлове (пакети). Комуникацията между приставката и приложението се извършва чрез REST (Representational State Tranfer) заявки.



Фиг. 3.1 Обобщена диаграма на проекта

3.1.1 Потребителска част - приставка на Go

Реализацията на приставката включва имплементирането на шаблона за програмиране Command. В главният файл на приставката се намира интерфейса command, който има функция execute(). Освен това, има функция за създаването на асоциативен контейнер (map), съдържащ командите като референции към обектите, с ключове - съответните имена на командите като символни низове.

В главната функция на приставката, дефинирана от шаблона на Cloud Foundry, Run(CliConnection, []string), се създава клиентът, обработващ командите (неговата реализация е в client.go файла), проверява се за грешка при тази операция, взима се map с командите от предишно споменатата функция и се итерира през него. Чрез итерирането се проверява дали на входа на приложението клиентът не е написал някоя команда, поддържана от пакетния мениджър. Ако това е така, се извиква методът execute() на тази команда, проверява се за грешка и изпълнението на програмата се прекратява.

```
func (c *ApmPlugin) Run(cliConnection plugin.CliConnection, args
[]string) {
	client, err := getClient(cliConnection)
	if err != nil {...}
	commands := getCommands(*client)

	for key, command := range commands {
		if args[0] == key {
			if err := command.execute(args); err != nil {...}
		}
	}
}
```

Фиг. 3.2 Главната функция на приставката

Втората функция, дефинирана от шаблона за приставки на Cloud Foundry е GetMetadata() plugin.PluginMetadata. Тя връща мета информация за приставката, като версия и списък от команди. Всяка команда може да има псевдоним (alias), например:

```
{
	Name:     "install",
	Alias:    "i",
	HelpText: "Command for installing apps",
	UsageDetails: plugin.Usage{
		Usage: "cf install <app_name>",
	},
}
```

Фиг. 3.3 JSON информация за команда

Структурата client, имплементирана във файла client.go, съдържа информация за приложението в Cloud Foundry, заедно с името на организацията и пространството, в което е инсталирано приложението и Access Token за достъп до този профил. Тази структура има метод за изпълняване на команди, която приема името на комадната, URL, на който да се направи REST заявка и името на приложението, което да се обработи. Допълнително, тази функция може да приема променлив брой аргументи от тип “string”, които да са query параметри за заявката. Този параметър бе предназначен за способност за избиране на размер на памет и диск при инсталация.

Методът за изпълняване на заявките на пакетния мениджър построява пътя (URL) на заявката и извиква функция, която прави HTTP заявка, дефинирана в файла httpRequest.go.

```
func (c *client) manageApmCalls(apmCall string, httpVerb
string, appName string,
     query ...string) (string, error) {

     uri := "https://" + ...

     resp, err := httpCall(httpVerb, uri, c.token)
     if err != nil {...}

     return resp, nil
}
```

Фиг. 3.4 Функцията, изпълняваща операциите на пакетния мениджър

Изпълняването на HTTP заявката се прави от метода httpCall(string, string, string) (string, error), намиращ се в файла httpRequest.go. Този метод създава заявка посредством аргументите, подадени на функцията. Тези аргументи са

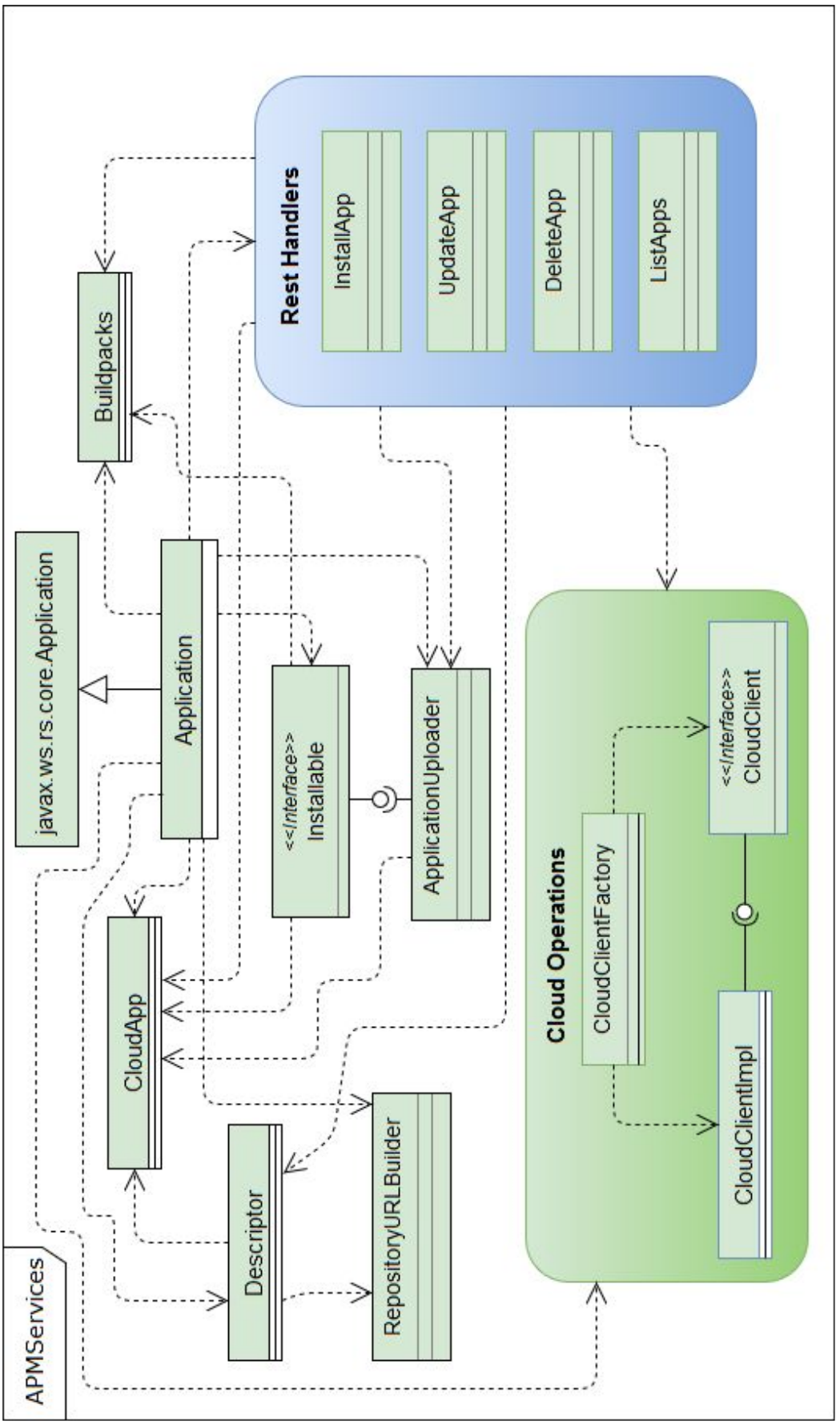
типа на заявката (GET / POST / PUT / DELETE), пътя, който да се извика и кода за достъп до профила в Cloud Foundry (Access Token). След създаването на заявката кода за достъп се слага като header параметър. Заявката се изпълнява и отговорът се обработва и връща от функцията.

```
func httpCall(method string, uri string, token string) (string, error)
{
	client := &http.Client{}

	req, err := http.NewRequest(method, uri, nil)
	if err != nil {...}

	req.Header.Set("access-token", strings.Split(token, " ")[1])

	resp, err := client.Do(req)
	if err != nil {...}

	defer resp.Body.Close()

	bs, err := ioutil.ReadAll(resp.Body)
	if err != nil {...}

	return string(bs), nil
}
```

Фиг. 3.5 Функцията, обработваща HTTP заявките

Причината, поради която кода за достъп се разделя на “ ” преди да се зададе като header параметър, е, че символния низ, който се подава на функцията е във формат “bearer <код за достъп>”.

Самите операции на пакетния мениджър са имплементирани във файлове със същите имена като командите. Те са install, update, remove, listInstalled и listRepo. Тези функционалностти са реализирани чрез структури, ипмлентиращи интерфейсът command. Структурите дефинират метода execute() със съответната функционалност на командата. Всяка дефиниция на метода е присъща за съответните команди, въпреки че има прилика между тях.



Фиг. 3.6 Клас диаграма на Java приложението

3.1.2 Сървърна част - Java приложение

Реализацията на Java приложението включва имплементирането на шаблоните за програмиране Factory, Singleton, Adapter и Builder.

В базовият пакет на приложението, org.elsys.apm, се намират класовете ApplicationUploader, CloudClientImpl и CloudClientFactory, и инерфейсите CloudClient и Installable. Класът CloudClientImpl имплементира CloudClient и използва шаблонът за програмиране Adapter. Този шаблон представлява енкапсулация (един основен принцип при обекто-ориентираното програмиране) на някой клас от друг, изискван от приложението, за да не може да се достъпват полетата или извикват методите на енкапсулираният клас директно. CloudClientImpl обгражда класът CloudControllerClient, на който се извикват методите за работа с Cloud Foundry. CloudClientImpl не съдържа всички методи на класа, който използва, а само тези, нужни на приложението. Те са дефинирани в интерфейсът CloudClient. Класът CloudClientFactory имплементира шаблонът за програмиране Factory. Factory е един от най-често използваните шаблони в Java. При него, когато се създава обект, се скрива логиката на създаване от клиента. CloudClientFactory има два метода, които създават обекти от тип CloudClient с различни аргументи за конструиране (единият метод е с код за достъп, а другият - с потребителско име и парола). Класът ApplicationUploader имплементира Installable и дефинира методите install() и checkDependencies(). В методът install() се създава и качва приложение в Cloud Foundry чрез “default” методите на интерфейсът Installable. Методи с модификаторът “default” представят базова (по подразбиране) имплементация на някаква функционалност. Те са въведени в Java 8 с цел съвместимостта на стари интерфейси с ламбда синтаксиса в Java 8. Освен инсталирането на подаденото приложение, install() инсталира рекурсивно и зависимостите на това приложение, ако не съществуват.

Методът checkDependencies() проверява дали са налични всички зависимости на едно приложение.

Класът Application в пакета org.elsys.apm.model наследява javax.ws.rs.core.Application и задава начален път на цялото приложение чрез анотацията “@ApplicationPath(“...”)”. В този пакет има също и списък от пътища (URL) от тип "enum", чрез които е възможна инсталацията на различни по програмен език приложения. Последният клас в пакета model е CloudApp, който е модел за облачните приложения, с които Java приложението работи.

Пакетът org.elsys.apm.descriptor съдържа класът Descriptor, който служи като връзка на приложението с хранилището за пакети. Този клас имплементира шаблона за програмиране Singleton. Този шаблон е много често използван и представлява клас, който не може да бъде инстанциран повече от веднъж. Това е реализирано тук, за да се избегне допълнителното време за изпълняването на заявка през интернет за да се прегледа хранилището, защото този клас е използван в 3 от главните функционалности на пакетния мениджър. Descriptor има няколко публични метода, които са за вземане на приложение като JSON обект от описателния файл на хранилището, съдържащ метаданни за приложенията; за вземане на множество от ключовете на този JSON файл за преглеждане на всички налични приложения; за проверяване дали някое приложение съществува в хранилището, при което в противен случай възниква ClassNotFoundException.

Пакетът org.elsys.apm.repository съдържа класът RepositoryURLBuilder, който имплементира шаблонът за програмиране Builder. Този шаблон конструира сложен обект, използвайки множество прости обекти. При представената реализация, класът RepositoryURLBuilder създава URL обект като добавя различни символни низове в StringBuilder обект в методите си и

връщайки обект от тип URL при извикването на метода build(). Методите на класът са repoRoot(), който добавя началния път на хранилището; repoDescriptor(), който добавя описателния файл на хранилището; target(String), който приема символен низ, име на файл, проверява дали този низ съдържа символи, невалидни за файловете, поддържани от пакетния мениджър чрез регулярен израз, и го добавя в символния низ на класа; build(), който създава обект от тип URL със символния низ конструиран до сега.

Следващите точки ще разгледат имплементациите на класовете, реализиращи функциите на пакетния мениджър. Разглежданите файлове се намират в пакета org.elsys.apm.rest.

3.1.2.1 List Apps

Функционалността за преглеждане на приложения е реализирана чрез клас ListApps, който съдържа два метода в себе си - за преглеждане на инсталираните приложения в профила на регистрирания потребител и за преглеждане на наличните приложения в хранилището.

Тези функционалностти, реализирани чрез методите, имат частни пътища (URL), наследяващи пътя, с който се извикват list-apps командите:

```
@Path("{org}/{space}/list_apps")
public class ListApps {
      @GET
      @Path("/repo")
      @Produces(...)
      public Response getRepoApps() {...}

      @GET
      @Path("/installed")
      @Produces(...)
      public Response getInstalledApps() {...}
```

Фиг. 3.7 Анотациите на методите в класа ListApps

Преглеждането на приложенията в хранилището става чрез вземането на описателния файл в него, който съдържа в себе си информация за наличните приложения под форма на JSON файл. Този файл бива прочетен и обработен в работоспособен за приложението формат в класа Descriptor. След като се направи обект от тип Descriptor във функцията getRepoApps(), се преглеждат всички JSON ключове, записват се в обект от тип StringBuilder и се връщат от функцията под формата на символен низ. Освен това има проверка за IOException и ParseException при евентуални грешки на приложението, в които случаи се връща статус код 500.

Преглеждането на инсталираните приложения става чрез извикването на функция на CloudControllerClient посредством класа CloudClient. Конструирането на обект от тип CloudClient става чрез CloudClientFactory класа. Извиква се метод getApps(), който връща списък с обекти CloudApplication, които са наличните приложения в организацията и пространството, асоциирани с профила на регистрираният потребител в Cloud Foundry. Имената на тези приложения се записват в обект от тип StringBuilder и се връщат от функцията под тип String.

3.1.2.2 Install

Функционалността за инсталиране на приложения е реализирана чрез класът InstallApp. Той има един публичен метод за реализиране на REST заявки за инсталиране на приложения, getInstallResult(), който връща резултата от операцията.

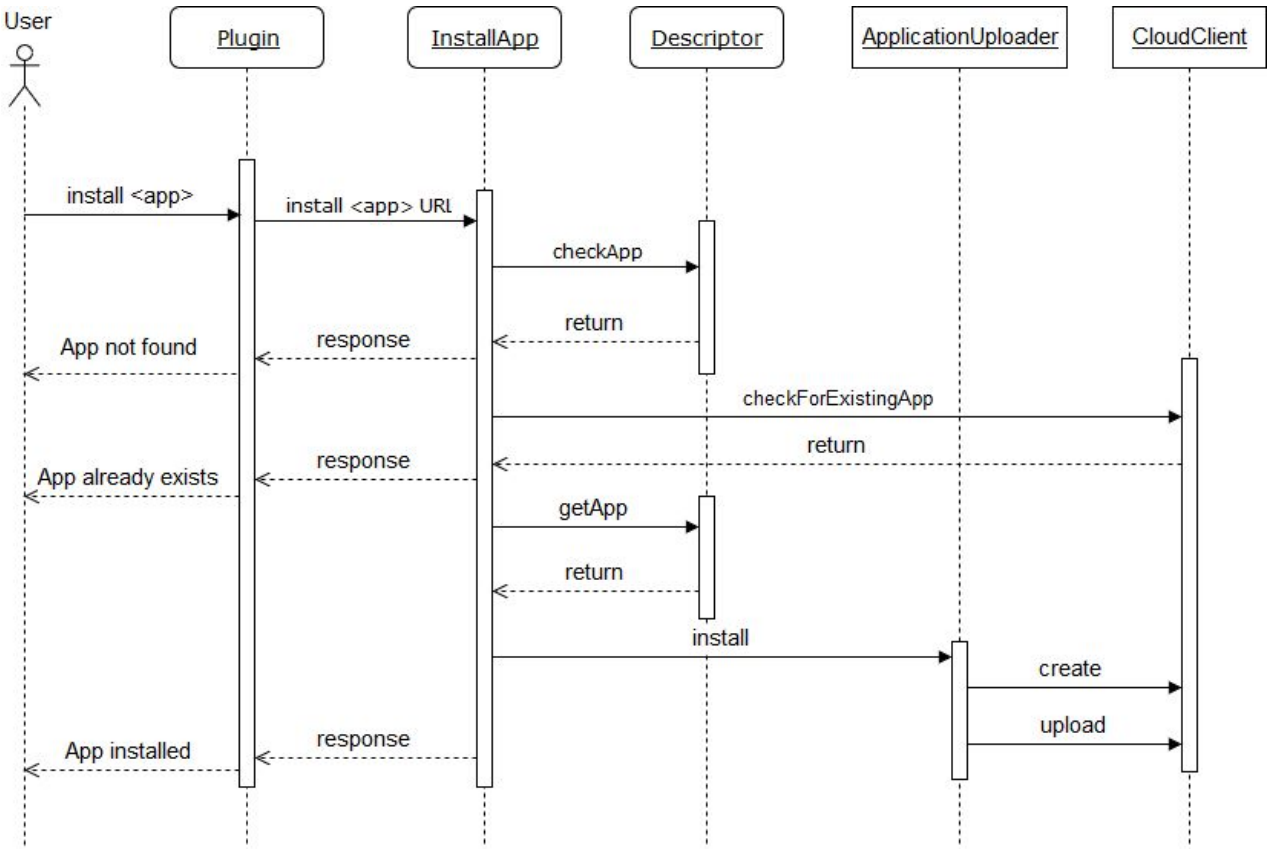
При ранен етап на развитие на приложението, операциите за работа с Cloud Foundry за създаване на ресурс (дисково пространство и памет) и URL за приложение бяха извиквани директно от CloudControllerClient, който е

класът, извършващ самата връзка със услугата Cloud Foundry да създаде приложение и да го качи в регистрирания профил.

При по-късни етапи от реализацията, тези операциите се извикват върху обект от тип CloudClient, който се създава посредством CloudClientFactory. Функцията проверява дали подаденото приложение съществува в хранилището, при което в противен случай връща статус код 404 и съответното съобщение. След това се проверява дали то не е вече инсталирано в Cloud Foundry и ако е, се връща код 400 и съобщение, че приложението е налично. Ако то съществува в хранилището, но не в Cloud Foundry, се построява ApplicationUploader обект и се извиква метода install(), като се подава като аргумент това приложение.

Ако програмният език на исканото приложение не се поддържа, getInstallResult() връща статус код 415 и съответното съобщение.

При успешна инсталация се връща код 201 и съответното съобщение, потвърждаващо резултата.



Фиг. 3.8 Диаграма на последователността на инсталирането

3.1.2.3 Update

Функционалността за обновяване на приложения е реализирана чрез класът UpdateApp. Той има един публичен метод, който се извиква при постъпване на заявка на пътя на класът (“/org/space/update/appName”) и връща резултата от операцията по обновяване. Този метод, getUpdateResult(), създава CloudClient посредством Factory класът за клиента, проверява за съществуващо приложение и го обновява, ако версията му в хранилището е по-нова от тази, присъща на наличното приложение. Проверяването на приложението става чрез метода checkApp() на класа Descriptor. Ако то не съществува в Cloud Foundry, getUpdateResult() връща статус код 404 и съответното съобщение. Ако приложението е налично в Cloud Foundry, но не съществува в хранилището, се връща статус код 410 и съобщение, че приложението повече не се поддържа. Проверяването на версията на

наличното приложение и това, в хранилището става чрез рекурсивна функция, която гледа “major”, “minor” и “build” версиите.

```
private boolean checkVer(List<Integer> currentVers,
List<Integer> repoVers, int depth) {
    if (currentVers.get(depth) < repoVers.get(depth)) {
        return true;
    } else if (depth == 2) {
        return false;
    } else {
        return checkVer(currentVers, repoVers, depth + 1);
    }
}
```

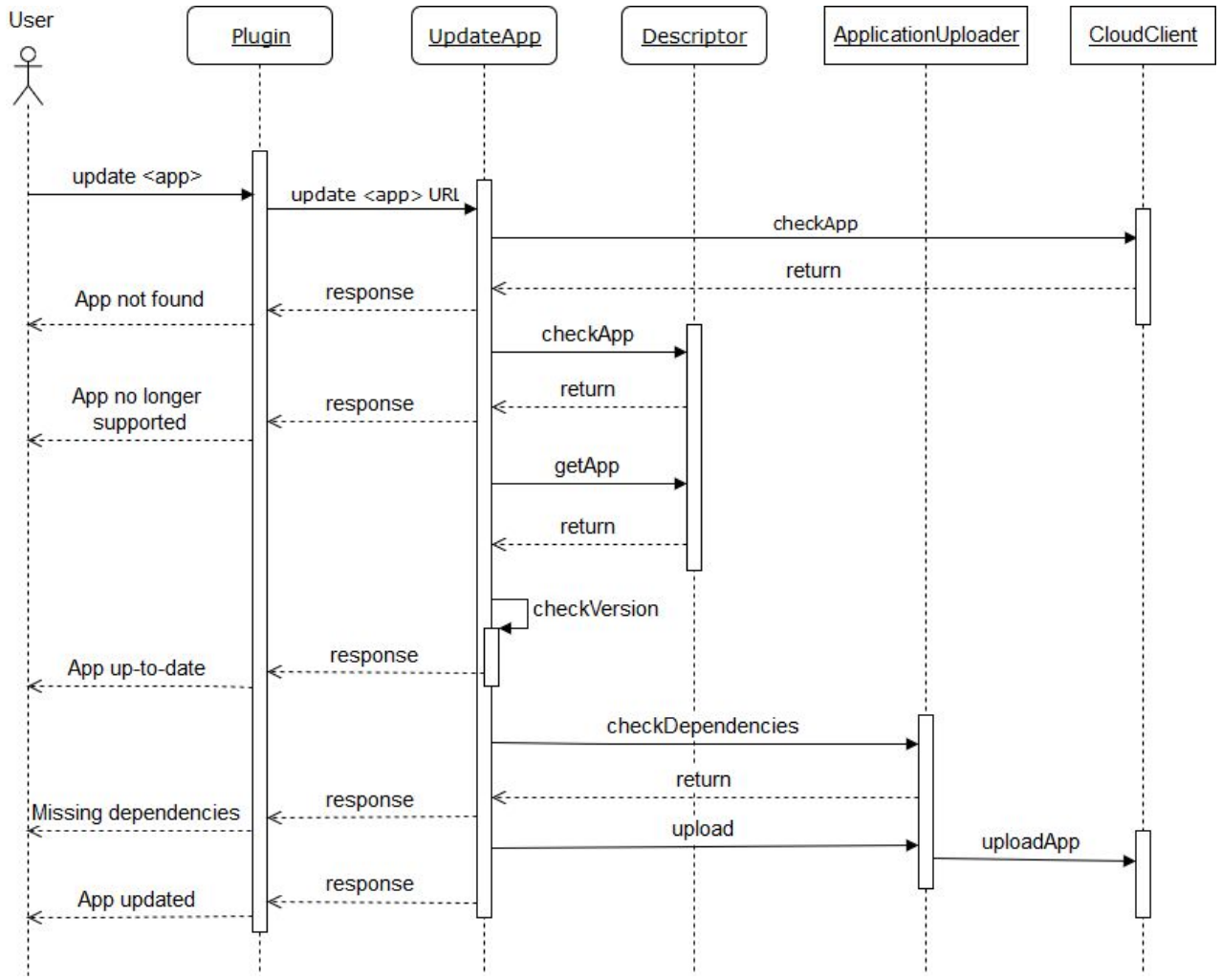
Фиг. 3.9 Функцията за проверяване на версии

Ако версиите са еднакви, функцията прекратява действието си, връщайки статус код 200 и съобщение, че приложението е най-нова версия. Ако версията в хранилището е по-нова, то тогава се конструира обект от тип ApplicationUploader, проверява се дали зависимостите на приложението са налични и се обновява, ако са.

Ако някои зависимости липсват възниква MissingResourceException и getUpdateResult() връща резултат 424 и съответното съобщение.

Метода, който се извиква за обновяване на приложения е upload() на ApplicationUploader. При Cloud Foundry обновяването на приложения става чрез качване на файл на адреса на приложението, без нужда то да се изтрива преди това или да се инсталира наново след това. Качването на приложения е възможно чрез File обект или входен поток от данни, но второто е избрано за да се избегне нуждата от създаване на локален файл.

Другото, което прави upload() е да обнови променливата за версия в метаданните на приложението в Cloud Foundry.



Фиг. 3.10 Диаграма на последователността на обновяването

3.1.2.4 Delete

Функционалността за изтриване на приложения е реализирана чрез класът DeleteApp. Той има една публична функция, която извършва действието на командата и връща резултатът от операцията. Тази функция, getDeleteResult() създава CloudClient обект посредством CloudClientFactory, проверява дали съществува подаденото приложение и ако е така, го изтрива от Cloud Foundry и връща статус код 200 и съобщение за успешна операция. При несъществуващо приложение връща статус код 404 и съответното съобщение.

3.1.2.5 JUnit тестове

Тестването на софтуерни продукти с автоматизиран процес (или части от него) е много важно. Това става с отделни софтуерни продукти или платформи, които предоставят възможности за разработването, изпълнението на тестовете и сравняването на резултатите. Самите автоматизирани тестове могат да следват както клиентски сценарии, така и вътрешни за продукта функционалности, които може да са трудни за ръчно тестване. При ръчното тестване е трудно да се тестват всички детайли по функционалността и поддържането ѝ в работещо състояние може да не е ефективно без автоматизация.

В представената реализация на пакетен мениджър са имплементирани множество тестове за проверка на правилната функционалност на кода. Тези тестове обхващат класовете ApplicationUploader, Descriptor, Buildpacks, CloudApp, RepositoryURLBuilder и CloudClientFactory. Тестовите класове покриват голяма част от пълната реализация на проекта.

Четвърта Глава

4.1 Ръководство на потребителя

4.1.1 Изисквания към компютърната конфигурация

Едно от главните предимства на използването на облачни приложения е, че не натоварват хардуера, на който са използвани толкова, колкото локално инсталираните продукти. Представеният пакетен мениджър попада в тази категория от приложения. Характеристиката му като облачно приложение означава, че потребителите нямат причини за променяне на хардуера си за да напаснат изискванията на продукта.

Пакетният мениджър може да работи безпроблемно на всички хардуерни и софтуерни платформи поддържащи Cloud Foundry. Поддържаните хардуерни платформи включват Intel и AMD процесори, x86 архитектурата, всякакво количество оперативна памет и хард-дисково пространство. Поддържаните софтуерни платформи са 32 и 64 битовите операционни системи Windows, macOS и Linux дистрибуциите базирани на Ubuntu, Debian и RedHat.

Преди инсталация няма нужда от допълнителна конфигурация на хардуера или софтуера.

4.1.2 Инсталиране

Преди да може да се инсталира пакетния мениджър, трябва да бъде инсталиран конзолният интерфейс на Cloud Foundry (CF Command Line Interface). Това става чрез вграден пакетен мениджър като Homebrew за macOS или apt-get за Linux или инсталатор за Windows, Linux или macOS.

Командите за инсталиране на Mac чрез пакетен мениджър са:

```
brew tap cloudfoundry/tap
brew install cf-cli
```

Командите за инсталиране на Ubuntu и Debian-базирани Linux дистрибуции са:

```
wget -q -O -
https://packages.cloudfoundry.org/debian/cli.cloudfoundry.org.key |
sudo apt-key add -

sudo tee /etc/apt/sources.list.d/cloudfoundry-cli.list
sudo apt-get update
sudo apt-get install cf-cli
```

Командите за инсталиране на Enterprise Linux дистрибуции и Fedora са:

```
sudo wget -O /etc/yum.repos.d/cloudfoundry-cli.repo
https://packages.cloudfoundry.org/fedora/cloudfoundry-cli.repo

sudo yum install cf-cli
```

След инсталация на конзолният интерфейс трябва да се регистрира потребител в системата на някоя от сертифицираните платформи на Cloud Foundry. Те включват Atos Cloud Foundry[63], Fujitsu Cloud Service K5[64], Huawei FusionStage[65], IBM Cloud Foundry[66], Pivotal Cloud Foundry[67], SAP Cloud Platform[68] и Swisscom Application Cloud[69]. Следният пример е с регистрация в Pivotal. Това може да се направи през уеб браузър.
При съществуващ и валидиран профил се изпълнява следната команда да се влезне в профила на потребителя през конзолата:

```
cf login
```

При първоначално влизане ще се изпише поле за посочване на програмен интерфейс. Въвежда се “api.run.pivotal.io”. След това се въвеждат потребителския имейл и парола.

След като потребител е регистриран и влезнал в профила си от конзола, може да се инсталира продукта. Това става с готовите скриптове в папката scripts, намираща се в главната директория на продукта. Проверява се дали тези скриптове са изпълними файлове, което при изтегляне на хранилището от GitHub би трябвало да са. Примерно изпълнение от Linux конзола представлява така:

```
./installApp
./uploadPlugin
```

Първата команда отнема сравнително време - около минута, заради начина, по който Cloud Foundry качва и инсталира приложенията си. Втората команда ще пита за потвърждаване на инсталацията на приставката. Трябва да се напише ‘y’ или ‘Y’ в конзолата и инсталирането ще завърши.

4.1.3 Използване на системата

За да се използва пакетният мениджър, потребителят трябва да има валидиран профил в Pivotal (или някоя от платформите, поддържащи Cloud Foundry) и да е влезнал в този профил през конзолния интерфейс на Cloud Foundry.

Поддържаните команди могат да се прегледат чрез въвеждане на “cf plugins” в конзолата. При тази команда ще се изведе следният екран:

```
plugin      version   command name          command help
apmPlugin   7.1.1     install, i            Command for installing apps
apmPlugin   7.1.1     list-installed, lsi   Command for listing installed apps
apmPlugin   7.1.1     list-repo, lsr        Command for listing repo apps
apmPlugin   7.1.1     remove, rm            Command for deleting apps
apmPlugin   7.1.1     update, u             Command for updating apps
```

Фиг. 4.1 Поддържаните команди от пакетния мениджър

4.1.3.1 Install

Инсталиране на приложения става чрез командата install. Примерно ползване на тази команда е както следва:

```
[rivanov@workstation ~]$ cf install todo-list-elsys
App installed successfully
```

Фиг. 4.2 Използване на командата за инсталиране на приложения

Освен този начин на използване може да се използва и псевдонима на командата “i”. Примерна такава операция е:

```
cf i <app_name>
```

4.1.3.2 Update

Обновяване на приложения става чрез командата update. Примерно ползване на тази команда при съществуващо приложение е както следва:

```
[rivanov@workstation ~]$ cf update test-app-two
App up-to-date
```

Фиг. 4.3 Използване на командата за обновяване на приложения

Освен този начин на използване може да се използва и псевдонима на командата “u”. Примерна такава операция е:

```
cf u <app_name>
```

4.1.3.3 Remove

Изтриване на приложения става чрез командата remove. Примерно използване на тази команда е както следва:

```
[rivanov@workstation ~]$ cf remove test-app-one
App deleted
```

Фиг. 4.4 Използване на командата за изтриване на приложения

Освен този начин на използване може да се използва и псевдонима на командата “rm”.

Примерна такава операция е:

```
cf rm <app_name>
```

4.1.3.4 List Apps

Командите за преглеждане на приложения са две. Те са за преглеждане на приложенията, налични в профила на ползващия потребител и за преглеждане на наличните приложения в хранилището. Първата команда се извиква по следния начин:

```
cf list-installed
```

Или с псевдонима си “lsi”:

```
cf lsi
```

Втората команда се извиква по следния начин:

```
cf list-repo
```

Или с псевдонима си “lsr”:

```
cf lsr
```

4.1.3.5 Добавяне на приложение за използване

Продуктът предоставя системата за управление на пакети. Но тъй като не съществува друг пакетен мениджър за тази платформа, няма официално хранилище за пакети/приложения. Съответно няма готови приложения, които да се инсталират и използват.

За да се добави ново приложение за използване от пакетния мениджър е нужно да се копира хранилището (https://github.com/radito3/gradProject), да се добави изпълнимия файл, или папката на приложението, в папката packages и да се направи заявка за приемане на промените.

Заключение

С настоящата дипломна работа беше реализиран пакетен мениджър за платформата Cloud Foundry, който дава решение на липсата на система за управление на облачни приложения и зависимостите между тях. Беше направен преглед на съществуващи решения за Unix-базирани системи и за macOS. Бяха разгледани аспектите на облачните модели на услуги и на Cloud Foundry.

Реализираното приложение се придържа към изискванията на стандартен пакетен мениджър като функционалността му бе пригодена за работа с облачни приложения. Това дава възможност за улесняване и ускоряване на работата на разработчици на приложения за Cloud Foundry.

Спазването на стандартите на Cloud Foundry за потребитески интерфейс позволява на потребителите на пакетния мениджър да започнат работата си с него без забавяне от нужда за научаване и пригаждане към нов интерфейс.

За гарантиране на качеството беше направено ръчно тестване и тестване с автоматични тестове JUnit. При създаването на JUnit тестовете бяха спазени най-добрите практики описани от Robert C. Martin в книгата Clean Code[70].

По време на работа, разработчика на приложението придоби опит и опозна библиотеката Jersey, програмния език Go и платформата Cloud Foundry.

Следващи стъпки при разработката на продукта биха били добавянето на способността за избиране на размер на памет и диск при инсталация на приложение, добавяне, обновяване и изтриване на хранилища за приложения, и търсене на отделни пакети по име или по регулярен израз. Също така област за разширение би била управляването на зависимостите между приложенията при добавяне на версии на нужните зависимости.

Използвана литература

Erich Gamma, Richard Helm, Ralph Johnson, John Vissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994

1. https://www.lifewire.com/what-does-checksum-mean-2625825

2. https://www.manageengine.com/products/desktop-central/software-repository.html

3. https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Binary_repository_manager.html

4. https://www.techopedia.com/definition/27519/app-store

5. https://cloudacademy.com/blog/cloud-foundry-benefits/

6. https://azure.microsoft.com/en-us/overview/what-is-paas/

7. https://opensource.com/resources/what-open-source

8. https://www.vmware.com/

9. https://pivotal.io/about

10. https://www.dellemc.com/en-us/index.htm

11. https://www.ge.com/

12. https://continuousdelivery.com/

13. https://help.ubuntu.com/lts/serverguide/dpkg.html

14. https://docs-old.fedoraproject.org/ro/Fedora_Draft_Documentation/0.1/html/RPM_Guide/ch07s03s06.html

15. https://www.geeksforgeeks.org/topological-sorting/

16. https://wiki.linuxfoundation.org/lsb/start

17. https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

18. http://www.operating-system.org/betriebssystem/_english/bs-netware.htm

19. https://www.ibm.com/power/operating-systems/aix

20. http://www.iusmentis.com/technology/hashfunctions/md5/

21. https://www.gnupg.org/

22. https://www.computerhope.com/jargon/g/gui.htm

23. https://en.opensuse.org/Portal:YaST

24. https://www.freedesktop.org/software/PackageKit/pk-intro.html

25. https://www.suse.com/products/server/highlights/

26. https://embedded.eecs.berkeley.edu/eecsx44/fall2011/lectures/SATSolving.pdf

27. https://www.microfocus.com/novell/

28. https://www.gnu.org/licenses/gpl-3.0.en.html

29. https://wiki.archlinux.org/index.php/Arch_Build_System

30. http://www.finkproject.org/about.php?phpLang=en

31. https://www.freebsd.org/ports/

32. http://guides.rubyonrails.org/getting_started.html

33. https://github.com/

34. https://www.commerce.gov/doc/national-institute-standards-and-technology#5/37.546/-91.103

35. https://en.wikipedia.org/wiki/Cloud_computing#Service_models

36. https://www.engineyard.com/platform-as-a-service-cloud

37. https://www.xenproject.org/

38. https://www.virtualbox.org/

39. https://www.oracle.com/virtualization/vm-server-for-x86/index.html

40. https://www.vmware.com/products/esxi-and-esx.html

41. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/

42. https://azure.microsoft.com/en-us/

43. https://cloud.oracle.com/paas

44. https://cloud.google.com/appengine/

45. https://www.gartner.com/it-glossary/multitenancy

46. https://docs.cloudfoundry.org/concepts/overview.html

47. https://aws.amazon.com/

48. https://www.vmware.com/products/vsphere.html

49. https://www.openstack.org/

50. https://www.britannica.com/topic/Sun-Microsystems-Inc

51. https://ant.apache.org/

52. https://www.lifewire.com/what-is-an-xml-file-2622560

53. https://www.mvnrepository.com/repos/central

54. https://www.apache.org/

55. https://jakarta.apache.org/

56. https://www.microsoft.com/net/

57. https://gradle.org/

58. https://www.scala-sbt.org/

59. https://ant.apache.org/ivy/

60. https://www.jcp.org/en/jsr/detail?id=311

61. https://www.jcp.org/en/jsr/detail?id=339

62. http://searchmicroservices.techtarget.com/definition/UDDI-Universal-Description-Discovery-and-Integration

63. https://atos.net/en/solutions/application-cloud-enablement-devops/multi-cloud-application-platform

64. http://jp.fujitsu.com/solutions/cloud/k5/function/paas/cf/

65. http://developer.huawei.com/ict/en/site-paas

66. https://www.ibm.com/cloud/cloud-foundry

67. https://pivotal.io/platform

68. https://cloudplatform.sap.com/index.html

69. https://www.swisscom.ch/en/business/enterprise/offer/cloud-data-center-services/paas/application-cloud.html

70. Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 2008

Съдържание