ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kratochvíl** Jméno: **Ondřej** Osobní číslo: **466267**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Continuous Delivery on the Salesforce Platform

Název bakalářské práce anglicky:

Pokyny pro vypracování:

Salesforce system and its platforms are widely used by many corporate and non-profit organizations. Applications on these platforms can get large and with standard Application Lifecycle Management it is time consuming to develop, test, deploy and maintain these applications. It is common that Salesforce apps are not integrated into any version control system, because the source of truth is designed to be production org [3]. Another frightening fact is that all of the migrations are done manually by developers, who need to track somehow all the changes. [1,2]
Aims of the thesis:
Study the metadata source on the Salesforce Platform.
Analyse methods of metadata migration between Salesforce instances.
Analyse requirements of application lifecycle management on this platform.
Research existing solutions for Continuous Delivery on this platform.
Study the new technology Salesforce DX.
Implement application lifecycle management on the Salesforce Platform, including version control integration and automatize testing.
Evaluate the usability of the process and suggest improvements.

Seznam doporučené literatury:

1. David Farley, Jez Humble: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0321601919
2. Michael J. Kavis, Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS), 1st Edition, John Wiley & Sons, Inc., 2014, ISBN-13: 978-1118617618
3. Apex Developer Guide. Salesforce, 2019, [cit. 2019-01-08]. Available at: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_dev_guide.htm

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jiří Šebek, kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2019** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **20.09.2020**

[redacted] [redacted] [redacted]

Ing. Jiří Šebek
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

9.5.2019 [redacted]

Datum převzetí zadání

Podpis studenta

FACULTY
OF ELECTRICAL
ENGINEERING
CTU IN PRAGUE

Bachelor's thesis

Continuous Delivery on the Salesforce Platform

Ondřej Kratochvíl

Department of Computer Science
Supervisor: Ing. Jiří Šebek

May 23, 2019

Acknowledgements

I would like to express my gratitude and appreciation to my advisor Ing. Jiří Šebek especially for his patience and motivation.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 23, 2019 .....

Czech Technical University in Prague
Faculty of Electrical Engineering

Citation of this thesis

Kratochvíl, Ondřej. *Continuous Delivery on the Salesforce Platform.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2019.

Abstrakt

Tato práce analyzuje současné procesy vývoje na cloudové platformě Salesforce, popisuje co to jsou metadata a jaké jsou možné způsoby jejich migrace. V další části se tato práce zabývá novou technologií Salesforce DX a jak těchto změn využít v rámci návrhu aplikačního životního cycklu. Významnou součástí práce je implementace skriptů a Java archivu, které umožňují automatizaci vývoje. Na závěr hodnotí navrhnutý řešení a předkládá možná rozšíření.

Klíčová slova Salesforce, SFDX, PaaS, SaaS, vývoj, CI, CD

Abstract

This thesis analyses the current development processes on the Salesforce cloud platform, describes what metadata are and what are the options for their migration. Further, this thesis deals with the new technology Salesforce DX and how to use these changes in the application lifecycle design. A significant part of this work is an implementation of scripts and Java archive, which allows development automation. Lastly, it comments the proposed solution and provides possible future extensions.

Keywords Salesforce, SFDX, PaaS, SaaS, development, CI, CD

Contents

List of Figures

Introduction

Salesforce platform is a cloud computing platform as a service developed by Salesforce, a company that specialises in customer relationship management. Its growth and success lead us to many questions about development improvements and simplifications on this platform. A platform as a service, in general, brings many advantages; a developer does not need to deal with servers, infrastructure and its security, and therefore can focus directly on his assignment [1, 2].

Salesforce made a significant step forward with the Salesforce platform providing stable development tools and apps built on the top of it; however, the development operations are far behind other modern technologies. Convenient integration with any version control system is missing, and also to achieve a mechanism of deploying changed and newly created metadata is impossible with standard tools provided by Salesforce. Usually, all developers must remember the metadata they created or modified during development so they can later deploy them, and this fact causes severe problems and significantly prolongs deployment phases.

Aims of this thesis

This thesis aims to analyse development, testing and deployment processes on the Salesforce platform, including analysis of instance types, metadata and their migration. The goal is to integrate the testing, deployment and release activities into the development process. There must be little to no risk when deploying to production, and therefore, not only the software must be tested automatically and adequately but also the deployment itself must be tested, before releasing it to production [3]. A significant part of this thesis should be a solution which provides a practical solution that will simplify the whole development process.

Structure

Firstly, I would like to summarise the current development approach and deduce difficulties it brings. This part will also contain a general description of a project structure, metadata, instance types and deployment options. Then follows an analysis of the solution. In this part, I would like to describe a new technology called Salesforce DX and what it brings to us, put together requirements for the solution and provide its design. In another part, I will focus on the solution's implementation and its configuration. Lastly, I will review the solution's advantages and disadvantages and list other problems to solve to achieve a fully automatised development environment.

Chapter 1

Development on the Salesforce Platform

There are several disadvantages to the current approach for the development process, and the most severe is there is no integration with any version control because the source of truth lives in production. Omitted version control is very painful for both, admins and developers, and leads to inconvenience. Every declarative change made directly in the production is not traceable anymore. Moreover, if there is some serious bug deployed, there is no easy and fast solution to revert. And many other advantages as pull requests and automatised testing that version control brings are missing.

1.1 The Salesforce Platform

The Salesforce platform is the core of Salesforce services. The platform is powered my metadata and other services and APIs such as data services, artificial intelligence, and robust APIs for development. Salesforce provides applications, such as Sales Cloud, Service Cloud and Marketing Cloud, that sit on top of the platform. These apps are consistent with apps you build on the platform. See visualization of the Salesforce platform and applications built on it in figure 1.1.

1.2 Sandbox types and their roles

Sandbox is a Salesforce instance isolated from production org. Salesforce provides several sandbox types with each for different purposes. Some of them are lightweight to be fast and some of them contain a specified amount and type of data for a different kind of testing. Every sandbox can be refreshed, that means that the sandbox's metadata are updated from its source org. Use cases for each type can be seen in table 1.1.

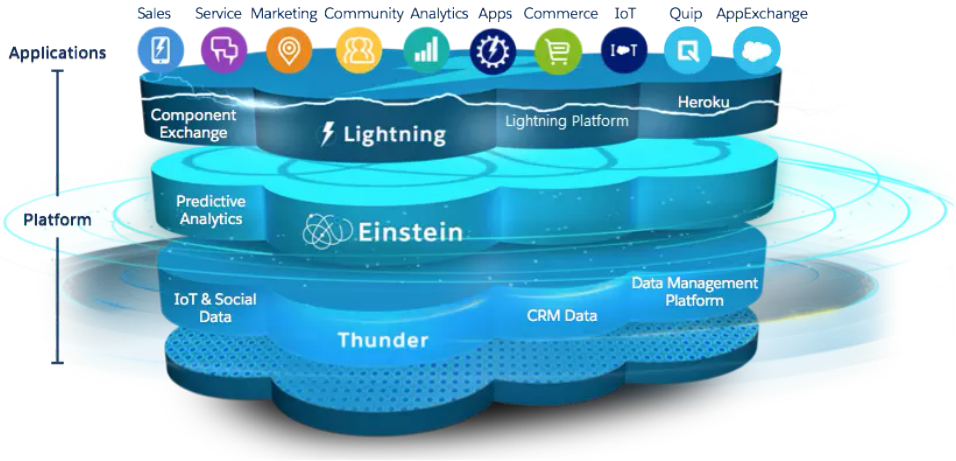


Figure 1.1: Salesforce platform overview [4]

- Developer Sandbox

  The main purpose of this sandbox is isolated development and testing. This sandbox contains a copy of all metadata from production.

- Developer Pro

  This sandbox has a similar purpose as a basic Developer sandbox. Developer Pro sandbox has a higher limit on data sets, so it can also be used for integration and user testing.

- Partial Copy

  Used mainly as a testing environment. Similar to Developer Pro sandbox, but with a sample of production org's data defined by a sandbox template.

- Full Copy

  This sandbox is used as a testing environment because it is a full copy of a sandbox containing all the metadata and all the data, so it is suitable for performance testing, load testing, and staging [5].

Table 1.1: Sandbox Use Cases [5]

Use Case	**Developer**	**Developer Pro**	**Partial Copy**	**Full Copy**
Develop	yes	yes	yes	
QA	yes	yes	yes	
Integration Test			yes	yes

Table 1.1: Sandbox Uses

Use Case	**Developer**	**Developer Pro**	**Partial Copy**	**Full Copy**
Batch Data Test			yes	yes
Training			yes	yes
UAT			yes	yes
Performance and Load Testing				yes
Staging				yes

1.3 Metadata

Metadata on the Salesforce system represents every customisation, every metadata is of some type, and there are precisely 242 metadata types (Spring '19 release) [6]. For example, there are metadata types such as CustomObject and CustomField representing database objects, ApexClass and ApexTrigger used for programmatical backend logic, Flow and Workflow which are produced by declarative tools. For more examples see table 1.2.

Table 1.2: Metadata Types examples

Metadata Type	**Description**
ApexClass	Represents an Apex class, compiled after deployment.
ApexPage	Represents a Visualforce page.
ApexTrigger	Represents an Apex trigger. Apex code executed after specific DML event, e.g. before/after, insert/update/delete/undelete events.
ApprovalProcess	An approval process automates how records are approved in Salesforce, contains all approval steps.
AuraDefinitionBundle	A bundle containing Aura definition and its related resources.
CustomField	Represents the metadata associated with a field.
CustomLabel	Represents label used for translations.
CustomObject	Represents object entity definition
Dashboard	Represents visualization of reports.
FlexiPage	Represents the metadata associated with a Lightning page, customizable screen made up Lightning components.

Table 1.2: Metadata Types examples

Metadata Type	Description
Flow	Definition of a flow, which can navigate users through series of screens with capability to query and update records.
Profile	A user profile to manage permissions.
Report	Represents a custom report with specified objects, report type, filter logic, grouping.
ValidationRule	Represents rules to validate records before they are saved.
Workflow	Actions executed immediately or on a specific day when a record meets specified criteria.

1.3.1 Metedata migration

Creation of metadata of specific metadata types directly in production is forbidden for good reasons. There are few exceptions, for example, you can create reports directly in production. Most of the metadata has to be created either in a sandbox or a Developer Edition org. Then these metadata are migrated to the production.

Before completing a metadata deploy, package install, or package upgrade each Apex code is automatically recompiled.

You have many options how to migrate your metadata, each has its advantages and disadvantages listed below. For each metadata migration approach, you can see which metadata types are supported in the metadata coverage report [6].

Change Set

The most straightforward approach to send customisations from one org to another is Change Set. A Change Set is created using a declarative tool accessible through the setup without a need for a local file system. The main advantage of Change Sets is no knowledge is needed, using point and click method selects all the desired metadata, another benefit is you can upload the same Change Set into multiple orgs [5].

After the publishing org creates an outbound Change Set and uploads the Change Set to subscriber org, this Change Set can be validated and deployed using inbound Change Set again through the setup.

Metadata API

Every Salesforce org provides Metadata API, which can be used for retrieving, deploying, creating, updating and deleting metadata. This API is intended for implementation of migration tools such as Ant Migration tool [7]. Almost every customisation in Salesforce can be migrated through the Metadata API. For customisations not available in Metadata API one must write down all changes and recreate them in a target org manually.

Ant Migration Tool

Ant Migration tool provides file-based deployment of metadata changes that is scriptable. When deploying a more massive amount of metadata then the Change Sets can take a long time. A typical development process requires iterative building, testing, and staging before releasing to a production environment. Scripted retrieval and deployment of components can make this process much more efficient [7, 8].

All you have to do is modified file build.properties with sf.user and sf.password so the migration tool can connect to a target org. Once you setup build.properties you can implement targets. Example of a target that deploys metadata from a package called codepkg and runs only one test called SampleDeployClass is in the listing 1.1.

If you need to repeat this process, it's as simple as calling the same deployment target again [9].

Code 1.1: Ant target example

```
<target name="deployCode">
   <sf:deploy
      username="${sf.username}"
      password="${sf.password}"
      serverurl="${sf.serverurl}"
      deployroot="codepkg"
   >
      <runTest>SampleDeployClass</runTest>
   </sf:deploy>
</target>
```

Unmanaged Package

An unmanaged package is simply a container of metadata that can be deployed to any org instance. In comparison with the managed package, the unmanaged package is not upgradable and is usually used to distribute open-source projects or base functionality. The unmanaged package should not be

used to migrate metadata from sandbox to production [10]. Once the package is installed, all the metadata are part of the org, so all the metadata are visible and can be updated, however, can't be upgraded to a newer version.

Managed Package

As a Salesforce partner or an independent software vendor, you can create a package called Manage Package, which can be distributed and sold to customers. In comparison with the unmanaged package, the managed package can be versioned and upgraded in a subscriber org [10]. Another benefit is that the source code is hidden for subscribers. The code from the managed package visible and available to call directly for subscribers are classes, methods and attributes with the global access modifier [7].

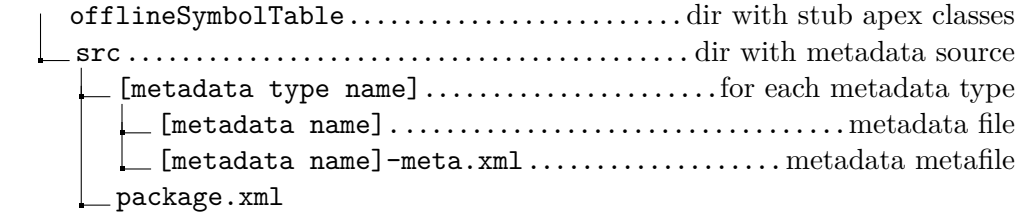
1.3.2 Unsupported Metadata Types

Not all customisations you make in a Salesforce org is available through the Metadata API. Components such as Console Layouts or Outlook Configurations can't be retrieved or deployed with the Metadata API, and changes to them must be made manually in each of your organisations [11].

1.4 Project structure

The project structure is rather simple. Every project contains at least some metadata files, package.xml and offline symbol table. See figure 1.2 for directory tree visualization.

Figure 1.2: Project structure of the current process



1.4.1 Generated supportive classes for local development

The Offline symbol table are supportive Apex classes for programmatical development in IDE, mainly for code completion, navigation and cross-referencing, and integrated API documentation features. These stub classes are system classes, entities called SObjects and all global classes with their declarations of global methods and attributes from installed managed packages [12]. The

offline symbol table is in general generated using some plugin, e.g. IlluminatedCloud. For a successful generation, a developer must have active connection to some Salesforce org from which the classes are retrieved.

1.4.2 Metadata files

All the metadata files usually live in a directory conventionally named src, and every metadata lives in a subdirectory named by its metadata type. This directory structure is inconvenient because that means all metadata of one type, for example, every file representing metadata type ApexClass lives in one directory, i.e. every test class, service class, UI controller class are placed in the same directory [13, 8]. This is caused by the fact that the project is not versioned. At the beginning of every development, the metadata are retrieved from the Salesforce org instance, and the org itself does not structure the project metadata the way we do it on a local machine. So even if you create a project structure at the beginning of a new project, new developers who retrieve the metadata do not have this information.

What can be a little confusing is that some of the metadata files have associated its metafile, for example, a metadata file of the type ApexClass named ProductService.cls exists with a metafile ProductService.cls-meta.xml. From the listing 1.2 we can see that metadata of type ApexClass has an XML element apiVersion, specifying the Salesforce API version, and status whether the class is active or not. Another example a static resource metafile shown in the listing 1.3 has contentType specifying type of the content, e.g. application/javascript, text/plain, image/png, application/zip.

File package.xml belongs to the metadata source so in our project structure is located in the src directory. This file is a project manifest specifying what to retrieve or deploy. Sample package.xml can be seen in the listing 1.4. In this example, we can see that metadata are grouped by metadata types in elements <types> specified with an element <name>. To list concrete metadata add another <member> element into specific type with its name as an inner text. Some metadata types such as CustomObject allows wildcard * (asterisk) for all metadata instead of listing each one.

Code 1.2: Example of an apex class meta file

```
<?xml version="1.0" encoding="UTF-8"?>
<ApexClass xmlns="http://soap.sforce.com/2006/04/metadata">
   <apiVersion>45.0</apiVersion>
   <status>Active</status>
<ApexClass>
```

Code 1.3: Example of a static resource meta file

```
<?xml version="1.0" encoding="UTF-8"?>
<StaticResource xmlns="http://soap.sforce.com/2006/04/metadata">
   <cacheControl>Public</cacheControl>
   <contentType>application/javascript</contentType>
</StaticResource>
```

Code 1.4: Sample package.xml manifest

```
<?xml version="1.0" encoding="UTF-8"?>
<Package xmlns="http://soap.sforce.com/2006/04/metadata">
   <types>
      <members>*</members>
      <name>CustomObject</name>
   </types>
   <types>
      <members>MyCustomObject__c.MyCustomField__c</members>
      <name>CustomField</name>
   </types>
   <types>
      <members>Case.samplerule</members>
      <members>Lead.newrule</members>
      <name>AssignmentRule</name>
   </types>
   <version>45.0</version>
</Package>
```

1.5 Current application lifecycle

Applications on Salesforce platforms can get large, and with standard Application Lifecycle Management, it is time-consuming to develop, test, deploy and maintain these applications. It is common that Salesforce apps are not integrated into any version control system or are integrated, but the content of the system differs from production org, because it is hard to maintain them to be equal. The production org was designed to be the source of truth and that is the key problem. Another frightening fact is that all of the migrations are done manually by developers, who need to track somehow all the changes. This process significantly prolongs the deployment and furthermore, it is impossible to test the deployment itself.

Current development cycle has usually four phases. The first phase begins with the creation of isolated Developer sandbox for each developer for implementation and unit testing. This sandbox has no production data. The second phase is a build release phase. Every customisation is migrated to a

shared Developer Pro sandbox for integration. This sandbox also does not have any production data, but testing data can be imported if necessary. Another phase is user acceptance testing in a Full Copy sandbox with production data. After successful testing, all metadata from the previous phase are migrated to production. The figure 1.3 illustrates all the phases [5, 14]. Note that this cycle is simplified for demonstrative purpose only and can differ from project to project, because every project has different need for testing.

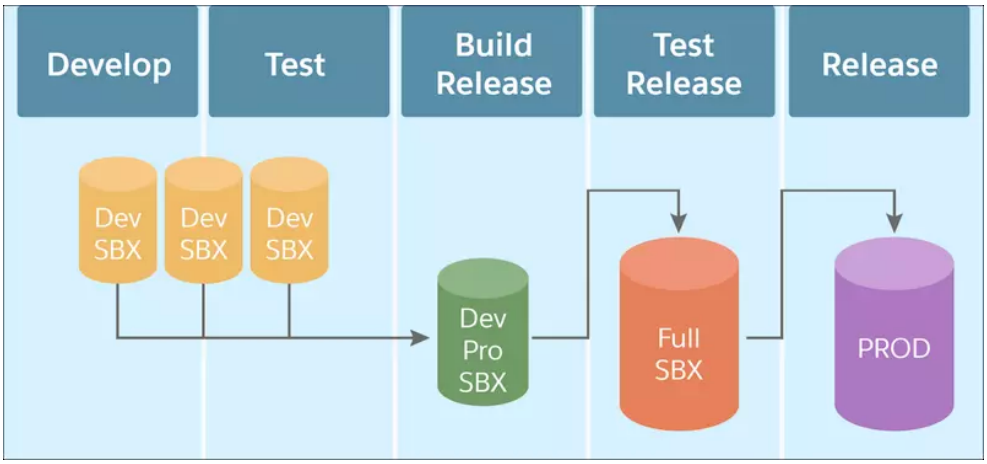


Figure 1.3: Current application cycle [15]

Chapter 2

Analysis

This chapter examines a recent technology Salesforce DX, what are the changes it brings and what it means for our continuous delivery process. Later, I will move on to explain what are our goals, how a possible solution to achieve them could look like, analyse the solution and provide its design.

The solution is generally useful for every developer and administrator on the Salesforce platform, who is responsible for source quality, its maintenance and deployments. The main goals are to achieve a reliable process of development, testing, deployment and maintenance.

2.1 Salesforce DX

Salesforce DX is a relatively new technology. It is a released product; however, a lot of its features are still under development. This technology by Salesforce is changing the game of continuous delivery on the Salesforce platform. Many benefits come with Salesforce DX, especially changing the source of truth from production org to a version control system, Scratch Orgs and CLI.

2.1.1 Version control system

The main distinction between standard development and development with Salesforce DX is changing the source of truth. Salesforce developers usually pull metadata from a Salesforce org, and that is the project and metadata with which they work. It is not common to use a version control system for collaboration and maintaining a project, because, with time, the content of a version control system sooner or later differs from real metadata in production.

With Salesforce DX, the source of truth is in a version control system, and this fact opens new options. One of the favourable possibilities is to enhance the project directory structure. From section 1.4.2, we know that metadata

files are grouped by their type; this can now be changed because the metadata are versioned with their structure.

Also, many metadata types are now parsed into separate files. Take a look, for example, on the SObject metadata file, in the standard format, this file contains all the SObject's information, including fields, list views, record types and many others. Now, all these children metadata types are parsed into separate files that makes it much simpler to automatise builds of these types, because they are easily visible in the git diff-tree result.

The more important benefit is that the whole project can be reverted to some point in its history in case of an application failure. This rollback is not possible in standard development, because the Salesforce org does not make any versions. We can configure continuous delivery with some automatisation server and webhooks to go even further. The automatisation server can provide many benefits such as automated build, test and deploy.

2.1.2 Scratch Org

A Scratch Org is a new type of Salesforce Org. Till now, we did not have any fully isolated and lightweight environment. A scratch org is entirely configurable, from choosing Salesforce edition to allowing different features and preferences. Every scratch org is defined by its configuration file, which can be shared even through a version control system [16].

A user needs a connected DevHub and a scratch definition file to create a scratch org. A DevHub is an ordinary Salesforce Org with activated DevHub mode and limits for active scratch orgs and daily scratch org creation.

A scratch org is useful for many purposes. In our process, we will use scratch orgs for two use cases, as an isolated development environment and for automatised testing.

2.1.3 Command line interface

Salesforce comes up with the command line interface to add some scripting options for automatisation. This interface has many features, from creating scratch org, authorise an org for use with the Salesforce CLI, manipulate records in an org, retrieve and deploy metadata using Metadata API, synchronise a project with scratch org and perform user-related admin tasks.

2.2 Continuous integration and delivery

Continuous integration and delivery are modern development practices. Together, these approaches provide a better system that find bugs quicker, improve quality, automatise testing and reduce the time taken to deliver the software.

2.2.1 Continuous integration

Continuous integration is useful for many reasons. Without continuous integration, the developers implement new features in an isolated environment for some time which leads to complicated and time-consuming merging and also bugs are usually detected late. To solve this problem, developers merge their work into a central repository more often, and integration service is being configured to built and run unit tests automatically [17]. This service is usually triggered by a version control service, for example by the BitBucket through a webhook, which execute jobs to built and run tests. The continuous integration is visualised in figure 2.1.

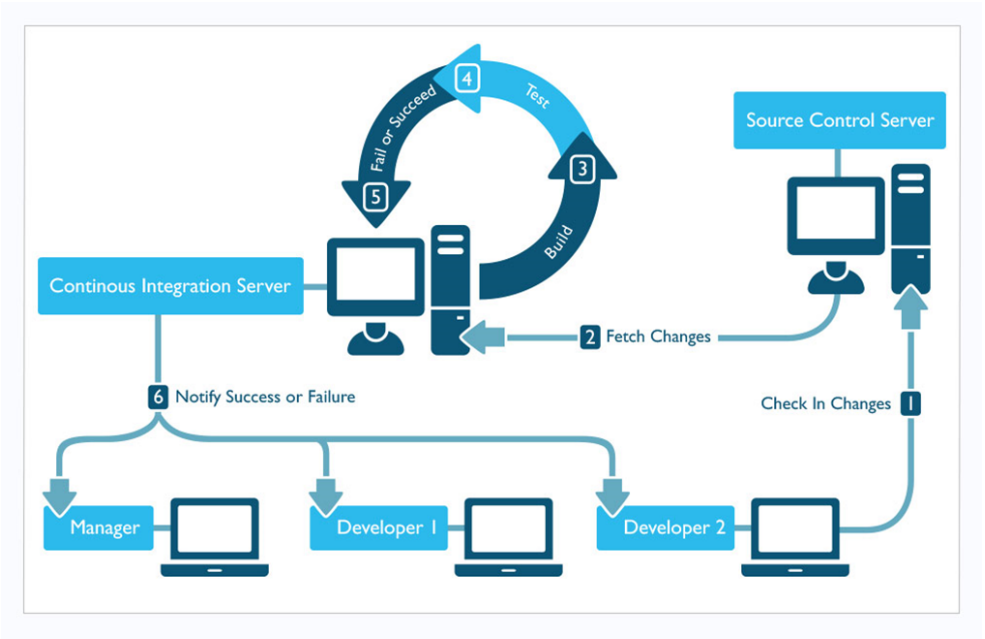


Figure 2.1: Continuous integration [18]

2.2.2 Continuous delivery

Continuous delivery is built on the top of continuous integration. To configure continuous delivery, we already need to be sure that continuous integration works as expected. When properly configured, every code change is built, tested and pushed to a staging environment. The code is deployed to production after a human approves it.

If continuous delivery works excellently and human interaction is, in most cases, not needed at all, we can go even further and implement continuous deployment. The only difference between continuous delivery and deployment

is need of human interaction; the former requires someone to approve the deploy, the latter deploys code to production automatically [19].

2.3 Application lifecycle

In this section, I will focus on how possible application lifecycle and branching strategy could look like when developing on the Salesforce platform with Salesforce DX. As always, I will try to propose the best possible solution, that can be used for the majority of projects.

Scratch orgs are great, but we do not forget traditional sandboxes. Both of these orgs have their purpose. Scratch orgs are great for development, unit testing, automatised testing and for quality assurance, as explained in section 2.1.2. On the other hand, traditional sandboxes such as Developer Pro, Partial Copy or Full Copy sandboxes are great for staging testing, user acceptance testing and training. In figure 2.2, we can see each purpose associated with some org type. This designation is not final for sandboxes, because of the projects differences. For example, think about a project that needs performance testing, the Full Copy sandbox is eligible for this purpose so we would see another testing before production.

If we compare this flow with the cycle from section 1.5, we will see that the most significant distinction is mainly in development and QA. That is because we have used the new scratch orgs for this purpose, which is fully isolated, disposable and created from source.

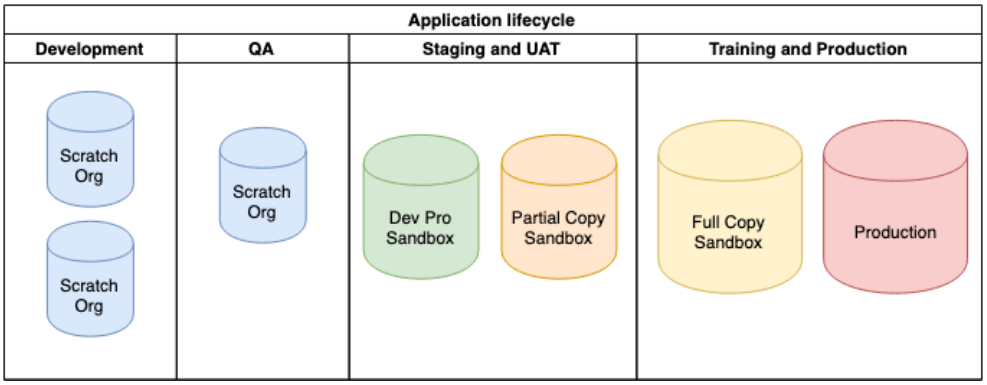


Figure 2.2: Application lifecycle

2.3.1 Branching strategy

Salesforce and Salesforce developers came with a lot of branching strategies, and every single one of the has some positives and negatives. The branching

strategy can differ from project to project. Different projects also require different types of testing, such as QA, UAT, SIT, staging. So there is not a universal solution to every project.

One of the conventional approaches is to have a master branch, which reflects production, a develop branch, which reflects a persistent integrated test sandbox and feature branches, which are then merged into the develop branch before releasing [14]. This approach could be convenient for many developers, but it has a significant disadvantage; every feature merged into the develop goes into the master, so you cannot choose a feature, and everything in develop goes to master. That is almost in every project unacceptable.

To avoid this problem, I came up with a strategy that every sandbox would also be reflected as a separate branch in VCS. This method has another benefit; a developer can clone a branch associated with the desired sandbox and instantly see what the org contains. The only thing developers need to bear in mind is, that when the sandbox is refreshed, he also need to update the branch with content from the production branch.

2.3.2 Development flow

In this section, I will go step by step how the developer should proceed. This guideline contains instructions from the creation of a feature branch to its merging to QA branch.

1. The developer creates a new temporary branch out of the master branch. This branch servers only for one purpose only; it can be feature, bugfix, hotfix, and should exist only for a few hours.
2. The developer creates a scratch org for development and works with metadata from that branch.
3. The developer runs unit tests in the scratch org.
4. The developer commits the change and publishes his branch in the central repository.
5. The developer raises a pull request to a new branch prefixed QA.
6. The project responsible person does a code review.
7. If the review is in order, the pull request is merged into the new QA branch.

2.3.3 Automatised flow

After a pull request is approved and merged to a branch prefixed with QA, the hosting service for version control fires a preconfigured webhook. The webhook is simple event containing information about the triggered repository,

branch and commit. This webhook is configured to send an HTTP request that triggers a job on the automation server, which pulls the source, creates a new scratch org and pushes the source into the scratch org for quality assurance testing. This step is identical for every project, and the following steps can differ depending on testing processes. Also, we need to be aware of the fact that the QA environment is a disposable scratch org and other testing environments are shared sandboxes. The same applies to their branches, QA branches are disposable and for one-time use only, on the other hand, the sandbox branch lives as long as the sandbox is not refreshed.

To deploy changes to a sandbox, we merge the feature branch to the sandbox associated branch. This merge also triggers a job on the automation server. Now, because the sandbox already contains metadata, the jobs functionality differs from the job for QA org. The automation server remembers the last commit of a source that was successfully deployed to the desired org and retrieves difference between this commit and head commit. From this difference prepares a deployment package. The job can also automatically and immediately deploy these changes, but for the beginning, it would not be ideal. A person that is responsible for this deployment looks at the job's result and if everything is all right, manually clicks on the deployment job for this sandbox.

2.4 Automation server user roles

For our automation server, we need three user roles with different permissions and access. The roles are hierarchical, i.e. a user with role higher in the hierarchy has at least the same permissions as a user with role beneath him. The three categories are developer, deployment responsible person and admin, see figure 2.3.

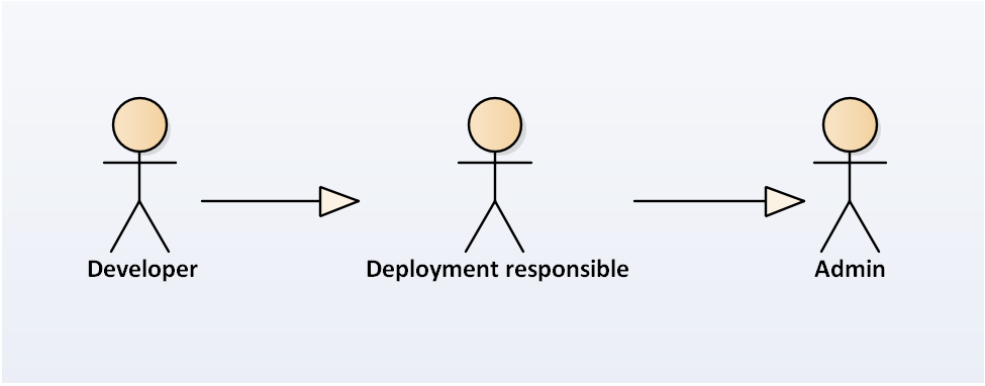


Figure 2.3: Automation server user roles

2.4.1 Developer

The developer is a role with fewest rights. This role allows access to the automation server and does not have any create or edit rights. The purpose of this role is mainly for viewing the job results and their log to see what happened if something went wrong during build or deployment.

2.4.2 Deployment responsible person

The deployment responsible person role is an extension to the developer role. It gives a user additional create and edit rights. This role is designated for a user, that has more experience with Salesforce platform and metadata. He is a supervisor of given builds and deployments and is the one who can add new jobs for new projects, modify the current jobs and run deployment jobs.

2.4.3 Admin

Admin has full access to the automation server. He can modify other users permissions and add new users.

2.5 Solution deployment

We need an automation server to automatise jobs such as a package build of metadata for deployment, deployment itself and automatised testing. There are many choices for this server, and I chose Jenkins, the explanation behind this choice is in chapter 4. This automation server must be active every time any action is needed. To achieve its availability, the Jenkins runs in a cloud. Again, there are many options for choosing a cloud server. For demonstration purposes, I decided to use AWS EC2, which is well documented and easy to use, but there are no limitations to use any other cloud server.

The solution itself is divided into two parts; bash scripts and the Java archive. Bash scripts serve as an interface for the automation jobs and depend on the Salesforce DX CLI and the Java archive. The archive provides more complex functionality, that would be impossible to implement in bash. Another important fact is that Jenkins requires JRE version 8 and because the Java archive should be implemented using modern Java, it requires JRE version 11.

The whole deployment is visualised in diagram 2.4. From the diagram, we can see that the scripts are dependent on many Salesforce orgs. The scripts need access to a DevHub so that the automation server can create scratch orgs. Then they need access to the created scratch orgs to deploy metadata and data for testing, and also they need access to the deployment target orgs. The target org is nothing else than a standard Salesforce org, e.g. production or sandbox. The access to the DevHub can be established manually once through the Salesforce DX CLI because the CLI itself remembers the refresh token.

To grant access to any new target org, we need to deploy the connected app, which will generate a new client id. How to use this id and the JWT key file is explained in chapter 4.3.1.

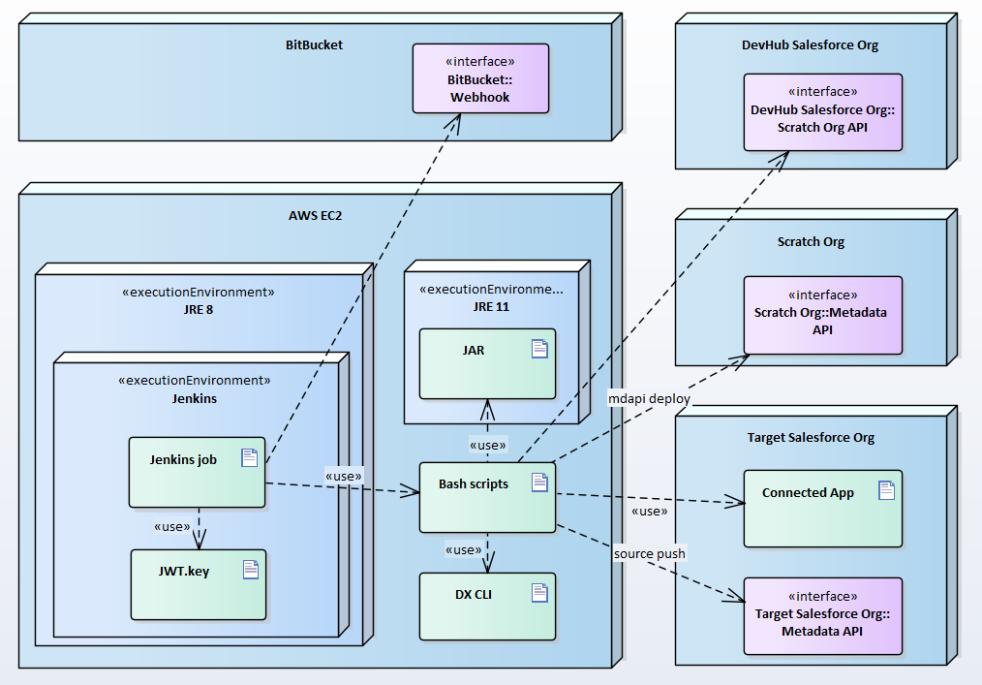


Figure 2.4: Deployment diagram

Chapter 3

Realisation

I decided to divide the solution into two projects. One project wraps the functionality, and the other provides complex utilities. The former is written in bash, so it offers easy to use scripts for CI/CD jobs. The latter is Java project, whose result is java archive used by the bash scripts.

3.1 Salesforce CI project

Salesforce CI project is a core application for dynamic package creation, Scratch Org creation, deployment and test automatisation. This project is mainly written in bash and for more complicated tasks uses java archive analysed in another section.

To use Salesforce CI, run a script called *sfci* that wraps the entire application. This script supplies usage and recalls subscripts placed in the *libs* directory. Each subscript is named with prefix *sfci* so all the scripts are independent on other executable files. As mentioned all the scripts are called through *sfci*. Usage is as simple as running *sfci <command> [<args>]*.

3.1.1 Dynamic package build

The most essential script is *sfci-package-build*. This script is responsible for dynamic package creation using git diff tree. It runs git diff-tree and filters metadata files, so it will be possible for us to deploy only changed and new metadata and destruct deleted metadata from any org. The result of this operation is a directory with files to deploy, and their corresponding deployment manifests.

Parameters to run this script are listed below:

- API version

  Optionally specify Salesforce API version for package manifest. By default, API version is parsed from the DX configuration file *sfdx-project.json*.

- Output directory

  Build specific name, same build name is used for deployment. All package metadata are saved in a directory *"SFCI_HOME/builds/buildname"*

- Two commit hashes

  These two hashes are used for the difference in the project.

In the beginning, it converts all the DX source to Metadata API format usgin *sfci source-convert* command in a temporary directory *sfci-all-metadata*.

Then is called *git diff-tree* which output is pipelined into our java archive execution. The diff-tree runs recursively with parameters *–no-commit-id* and *–name-status* so on each line is only a diff type and a path to a file [20]. We call the jar with the *package-build* positional argument to specify that we want to create a package. Also, we have to specify a metadata directory with our all metadata, which is the temporary directory, an build name used as a directory where the metadata and their manifests to be deployed will be saved.

To provide more flexibility we have to call the jar with parameters specifying two git diff filters [20], one for deployment and one for destructive changes.

In the end, the temporary *sfci-all-metadata* directory is deleted.

Note that you must be in a valid DX project, have installed JRE 11 and have Salesforce CI java archive to run this script.

Run *sfci package-build –help* for help.

3.1.2 Deployment

Deployment functionality is located in the script called *sfci-deploy*. To successfully run deployment you must provide few parameters:

- Target username

  The Salesforce org username or alias specifies against which org and under which user the deployment is run.

- Build name

  A path *"SFCI_HOME/builds/buildname"* is a directory with metadata files and their manifests to be deployed.

- Client Id

  Client Id string that is configured for the target org.

- JWT key file path

  Path to a JWT key file paired with the Client Id.

- Destructive changes strategy

  Specify whether the script should deploy the destructive changes pre or post.

  By default, the script will omit destructive changes.

Firstly, it tries to authorise the target user with the Client Id and the JWT key file. If it fails, no further changes are made. Otherwise, the script prepares destructive changes by specified pre or post strategy or defaults to no destructive changes. Then metadata are deployed using Salesforce DX CLI, and every while pulls report status and parses results. When the deployment is done, the report is printed to standard output in a human-readable format.

Run *sfci deploy –help* for help.

3.1.3 Last Commit hash manipulation

The *sfci-last-commit* script is responsible for manipulating last successful build git commit hashes. It stores or get these commits located in a directory *"SFCI_HOME/builds/buildname"*.

- Commit to save

  When saving a commit, you must implicitly specify which one.

- Build name

  A unique identifier of the build. Usually a combination of a project name and target org type, e.g. "My project UAT".

Run *sfci last-commit –help* for help.

3.1.4 Convert source

A script called *sfci-source-convert* encapsulates functionality to convert a project from DX format to Metadata API format. Is searches the DX configuration file *sfdx-project.json*, parses all package directories paths and each directory converts. Note that you must be in a valid DX project to run this script.

Provide these parameters to run this script:

- Output directory

  A path to a directory where the converted files will be saved.

Run *sfci source-convert –help* for help.

3.2 Salesforce CI jar extension project

The main bash scripts use the resulting jar from this project as a plugin. I chose to avoid scripting languages and use Java programming language instead because it is strongly typed and well-known for many programmers, so to add support for other metadata is simple.

The primary purpose of this application is to provide more complex utilities. One of the tasks is to consume git diff-tree result and prepare metadata and its manifest for automatised deployment, which is vital for Continuous delivery on Salesforce.

I focused on implementing a solution, that would be easily maintainable and adding support for other metadata as straightforward as possible.

To ensure flexibility the main programme parses first positional argument specifying which task to run. Every task is mapped to a class which implements a Job interface, so the main thread has still control over execution. I am using JCommander framework by Cédric Beust to parse parameters, "because life is too short to parse command line parameters" [21]. See JCommander in a PackageBuilderJob class in listing 3.1.

3.2.1 Resources

This project contains two necessary resources, one is a JSON configuration file, and the other is a mapping of metadata file path to its type. The configuration file describes all the metadata supported by this app, specifying metadata type, directory name, file's suffix, child types and optionally an element name if it is a child of another metadata. To create this file, you need to consume your org's Metadata WSDL. The mapping file contains on each line a mapping rule. Each mapping rule is a pair of a Java Regex pattern and a metadata type joined with the equal sign character.

3.2.2 Package builder job

This job is responsible for git diff-tree consumption, filtering metadata for deployment and creating deployment manifests, package.xml and destructiveChanges.xml. The app reads git diff-tree result from standard input and iterates over it for both, deployment and destructive changes. For every changed file the method converts the file into the package members and decides whether the metadata file assigns to deployment or destructive changes.

This operation can result in these exceptions:

- MetadataNotSupportedException

  This exception is thrown when the metadata is not supported by this app.

- NoSuchMetadataConverterException

Code 3.1: JCommander example in PackageBuilderJob class

```
@Parameters(separators = "=")
public class PackageBuilderJob implements Job {

   @Parameter(names = {"--version"}, converter =
       VersionConverter.class, required = true)
   private Version version;

   // directory containing all metadata
   @Parameter(names = {"--metadata-dir"}, converter =
       FileConverter.class, validateWith =
       ExistingDirectoryValidator.class, required = true)
   private File metadataDirectory;

   // directory in which all the metadata
   // with its manifests will be placed
   @Parameter(names = {"--output-dir"}, converter =
       FileConverter.class, required = true)
   private File outputDirectory;

   //  A for Added, M for Modified (see git diff-filter)
   @Parameter(names = {"--package-filter"}, listConverter =
       FilterTypeConverter.class, required = true)
   private List<FilterType> packageFilter;

   @Parameter(names = {"--destructive-filter"}, listConverter =
       FilterTypeConverter.class, required = true)
   private List<FilterType> destructiveChangesFilter;

   @Parameter
   private List<String> args;

   ...
}
```

Every metadata type requires its converter, and if the metadata type is recognised and no converter for it exists, this exception is thrown.

- MetadataFilesCopierException

  If deployment changes are being processed, the app must copy its corresponding file to the deployment directory. This exception handles this operation failure.

- MetadataConversionException

  This exception handles unexpected failures, that can be the wrong implementation of a converter.

Furthermore, if currently deployment changes are being processed, corresponding metadata files are copied from the directory with all metadata to the one designated for deployment. Finally, both manifests are saved to the deployment directory and printed to standard output so they can be read in console output on the automation server.

3.2.3 Metadata Converters

The package builder job currently supports many essentials metadata for development. New metadata will be released sooner or later, so the app was developed in a way that supporting new metadata is as simple as possible. That's where metadata converters take place.

All these converters extend an abstract class called MetadataConverter. All the attributes of this class are listed bellow:

- protected @NotNull String dxFileName

  Dx file name is the file name with its project relative path as returned from git diff-tree.

- protected @NotNull String baseNameWithoutExtension

  This is the base name of the file without its extension. This base name can be parsed from the dxFileName; however, it is a separate attribute to reduce code repetition because it is used by almost any converter.

- protected @NotNull Metadata metadata

  This attribute is just a reference to the immutable metadata instance of the file that is being converted.

- protected @NotNull File metadataDirectory

  A path to a directory with all metadata of the project. This is for example used by a WorkflowConverter, so that the converter can parse all its children, e.g. WorkflowAlert, WorkflowFieldUpdate, WorkflowRule from the metadata file.

To package members method

To answer a question why the converters are vital. That is because they contain two similar methods; one that converts a metadata file to a package member for deployment and the other that converts it to a destructive changes member. To fulfil these requirements, every converter must implement two methods:

- toPackageMembers
- toDestructiveChangesMembers

Metadata support

To add support of another metadata is simple. Just create a new class named as metadata type suffixed with *Converter* in the converters package that will inherit from the MetadataConverter class and implement its abstract methods, and you are ready to go.

That is because all the converters are instantiated dynamically via *MetadataConverterFactory*. The newInstance factory method concatenates metadata type with *Converter* and instantiates the converter as *"(MetadataConverter) constructor.newInstance(dxFileName, metadata, metadataDirectory)"*

Another feature of this factory is that the constructors are cached for each metadata type. So if for the same metadata type were a constructor once created, the method gets the constructor from the cache and does not have to call *Class.forName* and *getDeclaredConstructor*. If the converter is not found, an exception *NoSuchMetadataConverterException* is thrown.

All the converters are placed in a metadata.converters package.

Default metadata converter

A lot of metadata are represented just by a single file, additionally another associated metafile. These metadata are for example ApexClass, ApexTrigger, ApexPage, Layout, FlexiPage and many others. To reduce the code repetition and need to implement the same code over and over, this solution provides an abstract default metadata converter called *DefaultMetadataConverter* that solves this problem.

To use this feature, you need to extend the *DefaultMetadataConverter* class and override its abstract method *getMetadataFileNames* which returns a list of strings. This collection contains file names of the metadata files.

An implementation of an ApexClass converter that extend the *DefaultMetadataConverter* is available in the listing 3.2.

Code 3.2: Apex class converter

```
public class ApexClassConverter extends DefaultMetadataConverter {

    public ApexClassConverter(@NotNull String dxFileName, @NotNull
        Metadata metadata, @NotNull File metadataDirectory) {
        super(dxFileName, metadata, metadataDirectory);
    }

    @Override
    protected List<String> getMetadataFileNames() {
        String name = "classes/" + baseNameWithoutExtension + ".cls";
        return List.of(name, name + "-meta.xml");
    }
}
```

Custom object child converter

If you look on a dx object directory structure, you can see that every object now is parsed into multiple files. That is a great feature compared to the standard model because now we can track using git every addition and deletion of an object's child types, e.g. fields, list views, record types.

We can see that all these types differ only in their path, so to implement these converters, you have to create a class that will extend the *CustomObjectChildConverter* and provide two regex patterns that will parse the object name and the metadata name. The main feature of the *CustomObjectChildConverter* is a method called *toPackageMembersFromPattern* that accepts the two regex patterns and returns the collection of package members.

3.2.4 Logger

Logging is another significant component of the implementation. Not only it is beneficial during development and testing, but it is also crucial for further development. The whole process is logged, and the most significant part is that logs all the converted metadata, so the user can see what is going on and what will be built and deployed. For this purpose, I chose the log4j by Apache. Logger customisation is simple through a modification of a file log4j.properties in the resources folder.

3.2.5 Building java archive

To build jar, modify your configuration file *toolchains.xml*, so it reflects a path to your Java 11. Then run *mvn clean compile assembly:single* which builds a jar with its dependencies.

Chapter 4

Configuration

There are many options to choose an automation server, such as Jenkins, Bamboo or TeamCity. The list would get much longer, but there is no need to list all of them. All of them has some differences; some of them are open-source, others are paid, they also differ in the way they are installed and configured. For our purpose, there are not many differences, so I chose Jenkins, because it is well documented, can be easily set up and configured and has a large community, but again, many other automation servers would be sufficient.

As an extensible automation server, Jenkins has many advantages including easy configuration via its web interface, on-the-fly error checks, built-in help and supports hundreds of plugins, that integrates with it [22]. Jenkins is a client-server tool that can be deployed on the cloud. That is a great advantage for our purpose and brings another dilemma to solve; which cloud should we use? There is again no straightforward answer to this question. Before we vote for a specific cloud, we should weight up a price, security and its reliability. Amazon claims that theirs Amazon EC2 is a highly reliable environment, the service runs within a proven network infrastructure and data centres. On their website they also state that cloud security is at AWS the highest priority [23]. These statements are something I can hardly test, but what eventually decided to give the Amazon EC2 a try is that you pay for the compute capacity you consume. Note that all steps in these sections are demonstrated using bash.

4.1 Launch an AWS EC2 instance

Amazon provides many ways how to begin with their products. I find the most easier the AWS Management Console. A user must go through a few steps and he is all set up. In this section I will go through each of the steps and explain them.

First of all, we log into the AWS Management Console at https://aws.amazon.com/console/ and create an account if we already do not have one. Next, we will need to launch an Amazon EC2 instance. Go to the

Amazon EC2 Dashboard and choose "Launch instance". This will guide us through a configuration wizard [24].

In the wizard, we will need to choose an Amazon Machine Image (AMI). The Amazon recommends the Amazon Linux AMI, but I selected an Ubuntu Server LTS. For this decision, there is no specific reason but personal. You can also choose for example Windows, but that is not possible for our purpose, because the significant part of the implementation is written using bash scripts.

Next, we have to choose an instance type. This step is dependent on how many projects we will be building on this server. The instance types differ in the number of CPUs, memory size, instance storage and network performance. We can see a list view of instance types in figure 4.1.

Family	Type	vCPUs	Memory (GiB)
General purpose	t2.nano	1	0.5
General purpose	t2.micro Free tier eligible	1	1
General purpose	t2.small	1	2
General purpose	t2.medium	2	4
General purpose	t2.large	2	8
General purpose	t2.xlarge	4	16
General purpose	t2.2xlarge	8	32
General purpose	t3a.nano	2	0.5

Figure 4.1: EC2 instance type overview

In another step, we can create or select a security group which defines our virtual firewall. This step is optional but is always a good idea to go through it. For a demonstrative purpose only I set up a simple security group. We have to make up a self-descriptive name for this group, I chose Salesforce CI. Then we define inbound and outbound rules. Every rule contains type, protocol, port range and source. Types are for examples HTTP, HTTPS, SSH or custom TCP. Source filters IP ranges. We can see the configured inbound rules in figure 4.2. I did allow all IP addresses which is not best practice and you should allow only the IP addresses your users have, but for our purpose it is sufficient. Also, we can see that I add a rule with custom TCP on port 8080, this rule is for our Jenkins server, and SSH on port 22, so we can connect through terminal to our instance.

We can see complete configuration in figure 4.2. If everything is all right,

we select "launch" and proceed.

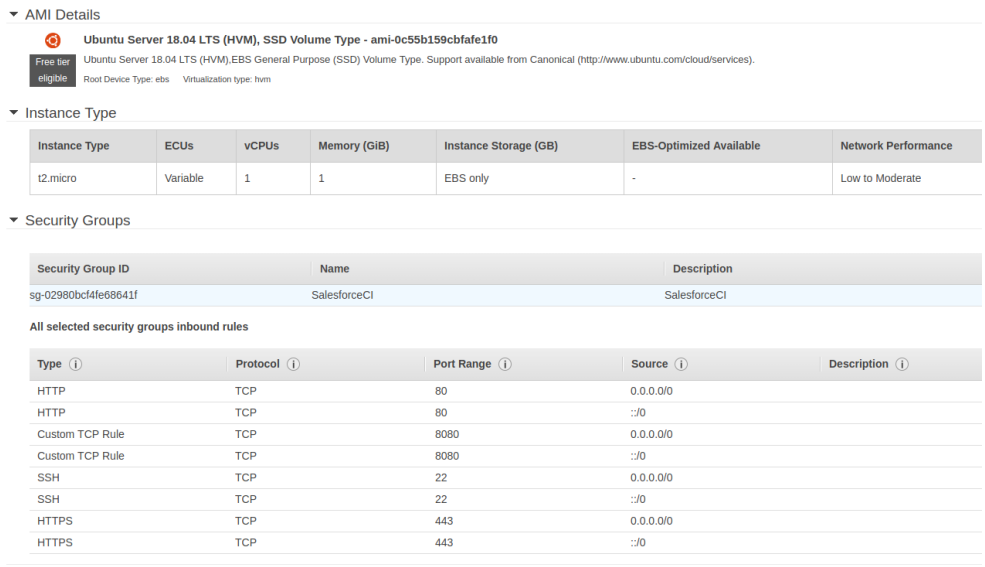


Figure 4.2: Create an EC2 instance overview

In the last step, we select "Create a new key pair" and assign its name, again, Salesforce CI will do it. This will automatically generate and download an RSA private key associated with your instance. The key will be used to connect to our instance.

4.2 Configure Salesforce CI

After we successfully launched an EC2 instance, we can connect to it through ssh and configure our CI/CD jobs. To achieve this, we need the RSA decryption key generated in the previous section. If we try to use the key directly, it will fail, because default permissions are too open and it is required that the private key files are not accessible by others. So before we use the key, we set its permissions to 400 using chmod.

Now we can connect to our instance using ssh as *"ssh -i path/to/SalesforceCI.pem ubuntu@your-domain-name.com"*, where the SalesforceCI.pem is you private key and *your-domain-name.com* is an instace domain name visible in the instance description. Note that after every reboot of your instance the domain name changes and also if you have selected different instance AMI, your user name can differ.

With an opened connection to our instance, we install all necessary, that is JRE and Jenkins. We need JRE because both, Jenkins and my solution run on Java Virtual Machine. Need to specify that Jenkins requires Java 8 and my

solution is built on a newer version, so we need to install JRE 8 and JRE 11. After we install Java, we also install Jenkins that includes the automatic start of the Jenkins service. We can ensure the service is running using command *systemctl status jenkins*. Also install *jq* that is used by the custom scripts.

To use the implementation, we need to copy the source code to the instance. That is also possible through our secure connection and remote file copy program scp (secure copy). For this purpose we create a directory called *"sfci"* in */var/lib/jenkins/workspace/* and make it executable using *chmod*. The last thing we need before closing the connection is to cat the initial admin password using *"sudo cat /var/lib/jenkins/secrets/initialAdminPassword"*. After this point, we do no longer need the ssh connection, and we can close it.

Jenkins configuration is simple; we only need a web browser, our domain name and the initial admin password. Open a web browser and enter your domain name with port 8080. This will get you to a Jenkins wizard, where you need to enter the initial password. Now that the Jenkins is unlocked, we can customise it. We do not install suggested plugins and select them manually. In this step, the wizard will preselect some recommended plugins and a lot of them we do not need, e.g. Ant, Gradle, Subversion. Be sure to install these plugins: Folders, Credentials Binding, Git and some plugin that will manage webhooks, if you are using BitBucket, I recommend Bitbucket Push And Pull Request Plugin [25]. Once the plugins are installed, we will be prompted to set up the first amind user, so we fill a user name, password, full name and e-mail address. After this, the Jenkins is configured and ready to use and the wizard will open the Jenkins console.

Next, we will configure global properties. To do that, we will go to Manage Jenkins in the Jenkins console and click on the Configure System button. In section Global properties add the environment variables list below.

- SFCI_HOME

  This variable is used by the implementation. It is, for example, a directory where last successful build commits are stored and where the scripts are placed. Set this variable to *"/var/lib/jenkins/workspace/sfci"*

- PATH

  We add our scripts to the PATH so we can execute our scripts with just their name. Set this variable to *"$PATH:/var/lib/jenkins/workspace/sfci"* We cannot use the SFCI_HOME variable, because the evaluation order of the variables is undefined.

Also set shell executable to point to *"/bin/bash"*.

4.3 Configure jobs

In this section, I will focus on the jobs configuration, especially on package build and deploy jobs to sandbox and production orgs. The scripts are written in a way, that you can combine the flow as you wish, but I will demonstrate the flow shown in the analytical section.

4.3.1 Grant access to org

The automatisation scripts need access to our orgs without human interaction. There are many ways to achieve this and one of them is to use JWT [26], which is also a method recommended by Salesforce [27].

To implement a JWT flow on the Salesforce platform, we need two things; self-signed certificate and Connected App with this certificate.

"A connected app integrates an application with Salesforce using APIs. Connected apps use standard SAML and OAuth protocols to authenticate, provide single sign-on, and provide tokens for use with Salesforce APIs. In addition to standard OAuth capabilities, connected apps allow Salesforce admins to set various security policies and have explicit control over who can use the corresponding apps." [28]

Firstly, we need to sign our certificate. For this purpose, we need some tool, e.g. OpenSSL, and generate a private key and its associated self-signed digital certificate. The certificate will be uploaded to our org to which we want to grant access, and the private key will be used by our jobs to connect to this org.

Now any org that we need in the automatisation process needs to contain a specific Connected App. To do that, we once create the connected app through setup, retrieve its metadata and this metadata push to any other org. To create the connected app, we select App Manager in one of the org's setup and click New Connected App.

We choose our connected app's name to be self-describing, that could be Salesforce CI, fill contact email and check "Enable OAuth Settings". Then, we need to set a callback URL to *"http://localhost:1717/OauthRedirect"*, upload our digital certificate generated earlier and select few OAuth scopes. Then we select these scopes: Access and manage your data (api), Perform requests on your behalf at any time (refresh_token, offline_access) and Provide access to your data via the Web (web). Note that the changes to take effect can last a few minutes. The last steps to configure the connected app is to edit policies and pre-authorise admin approved users, and lastly manage profiles, so at least the Admin profile is associated with the connected app.

The created connected app will generate consumer key and consumer secret. For our JWT flow, we will later need the consumer key, which serves as client id.

Also, we need to create Global credentials in our Jenkins instance of type *Secret file* and upload the private key associated with the self-signed certificate generated earlier. This credential file will be used together with the consumer key from the connected app to grant access to the org.

4.3.2 Build job

Firstly, we configure webhook that will trigger our job in BitBucket settings of the desired project. We set the webhook URL to *"<aws domain>:8080/bitbucket-hook/"* and choose the Repository push trigger.

Now, we can get back to Jenkins and configure the job. We create a new Freestyle project and name it, so it reflects the project name and instance name, e.g. "Project 1 - Build UAT".

We pair the job with our project using source code management and specify a particular branch that matches one of the orgs, in our example UAT.

Because we want to trigger this job on push, we select "Build with BitBucket Push and Pull Request Plugin" build trigger and set it to push action. Lastly, we populate execute shell field as in the example 4.1. In the example, we build a package using a specific build name, last successful commit and head commit.

Code 4.1: Example of execute shell in build job

```
build_name="Project 1 - UAT"
last_commit=$(sfci last-commit --get -n "$build_name")
head_commit=$(git rev-parse HEAD)

sfci package-build \
    --buildname "$build_name" \
    $last_commit $head_commit
```

4.3.3 Deploy job

In this job, we will use our connection to Salesforce org using the client id and the secret file. We configure source code management as in the build job and the execute shell as in the example 4.2. We can see that we need to specify a user name for the target org, the client id, and a build name as in the build job, so the deployment will know which build to deploy. Then we deploy the changes, and if the deployment is successful, we update the last commit.

Code 4.2: Example of execute shell in deploy job

```
target_username= % username to target org
client_id= % consumer key from the connected app
build_name="Project 1 - UAT" % same as in the build

head_commit=$(git rev-parse HEAD)

sfci deploy \
    --targetusername $target_username \
    --buildname "$build_name" \
    --jwtkeyfile "$JWT_KEY" \
    --clientid $client_id \

if [ $? -eq 0 ]; then
    sfci last-commit \
        --save $head_commit \
        --buildname "$build_name"
fi
```

Chapter 5

Testing

In this chapter, I will focus on testing. Because the implementation contains Java code, part of it are also unit tests because the proper testing is essential for a functional product. Another part of testing is user testing on which I will move on later.

5.1 Unit testing

I mixed two approaches for unit testing. During the development, I usually was implementing the code using test-driven development, and after the development, the rest of the classes was tested using black-box testing. For this purpose, I used jupiter JUnit 5 and wrote tests for almost every single class in the project. In figure 5.1, we can see the code coverage overview for metadata converters.

Element	Class, %	Method, %	Line, %
converters	71% (20/28)	69% (48/69)	75% (148/196)
DescribeMetadata	100% (1/1)	100% (2/2)	80% (8/10)
Metadata	100% (1/1)	75% (3/4)	69% (9/13)
MetadataNotSupported	100% (1/1)	100% (1/1)	100% (2/2)
MetadataType	100% (1/1)	100% (2/2)	100% (264/264)

Figure 5.1: Converters code coverage

5.2 User testing

User testing was performed on a real person to test the overall functionality and usability of the scripts. I chose two crucial use cases to be tested; package

build and deployment.

5.2.1 Test scenarios

In this section are listed test scenarios.

Building deployment package

- Task:

  Create a simple SObject and one its field and push these changes into the remote repository. Verify that the build job finished as expected. Delete the field, push changes and again verify the build job.

- Requirements:

  Test scenario presumes that an empty DX project exists in VCS and the project has configured build job in Jenkins.

- Test cases:

  1. Pull a DX project from the remote repositoy.
  2. Create a Scratch Org.
  3. Create an SObject and one field through the org's setup.
  4. Pull changes from the org.
  5. Commit the whole project, including pulled metadata.
  6. Push changes into the remote repository.
  7. Verify the build job in Jenkins, check its log.
  8. Delete the field through the org's setup.
  9. Pull changes from the org.
  10. Commit and push changes into the remote repository.
  11. Verify the build job in Jenkins, check its log.

- Expected result:

  The jobs finish successfully and the user sees his changes in both logs.

Deploying package

- Task: Create a simple SObject and one field, push these changes into the remote repository. Verify that the build job finished as expected. Find a deployment build associated with the build job, run the job and verify the deployment; check the jobs log and the target org.

- Requirements:

  Test scenario presumes that an empty DX project exists in VCS, the project has configured build and deploy jobs in Jenkins and that the deploy job is associated with an existing Salesforce org.

- Test cases:

  1. Pull a DX project from the remote repository.
  2. Create a Scratch Org.
  3. Create an SObject and one field through the org's setup.
  4. Pull changes from the org.
  5. Commit the whole project, including pulled metadata.
  6. Push changes into the remote repository.
  7. Verify the build in Jenkins, check its log.
  8. Find the deployment job.
  9. Run the deployment job.
  10. Verify the deployment job in Jenkins, check its log.
  11. Verify the deployment in the target org.

- Expected result:

  The jobs finish successfully, the user sees his changes in both logs and can find the deployed metadata in the target org.

5.2.2 Test evaluation

A user performed the test scenarios, and the whole test process was monitored. The results of the monitoring were compared against the expected results.

User characteristics

- 25 years old
- Salesforce developer
- familiar with Salesforce technologies and metadata deployment

First scenario

The user completed his task as expected. He did not have any significant difficulties, even though he is not familiar with Jenkins. The user recommends a few changes in logs to improve readability.

Second scenario

The test was completed as expected. The user was already more confident about Jenkins and again mentions how he would improve the logs to enhance readability.

5.3 Test conclusion

At first, I was sceptical to the test-driven development approach but wanted to try it, because sometimes I was decided what the classes should do, but was not sure about their implementation. To summarise if it was helpful or not, it depends on the situation; when I knew what the class interface should look like, the test-driven development was beneficial, on the other hand, if I was not sure, it was counterproductive, because I had to rewrite the test class.

Both user testing scenarios finished as expected. The user did not face any significant difficulties. Furthermore, the user recommended a few stylistic changes in the logs to improve readability.

Conclusion

The goals of this thesis were to examine possibilities for continuous integration and delivery on the Salesforce platform and implement a universal solution. In the first theoretical part of this thesis, I focused on current development on the Salesforce platform, its structure and pointed out its flaws. Later in this part, I am analysing new technology Salesforce DX, its benefits and options it brings for continuous delivery. Readers also find there how continuous delivery could be implemented on the Salesforce platform and analysis of a possible solution. In another part, I move on to the implementation based on the analysis from previous chapters. I am describing there how the solution is implemented, how it works and how to use it.

Implementation review

Salesforce has its imperfections and lacks a lot of functionality that should be part of the official product. The Salesforce platform was not designed for package development and version control system integration is almost impossible in any reasonable way. Salesforce provides many options for package development. However, it rather complicates the whole process.

I believe I successfully managed to provide a solution that significantly improves and simplifies development on the Salesforce platform and fully integrates a version control system into the development process, which rejects the whole view on the Salesforce org as a source of the truth. The implementation is generic, so anyone can change their process to fit their needs and still use the application. Even though the analysis counts on automatised testing in an isolated environment, the implementation is missing this functionality. Another benefit of the solution is that whenever Salesforce introduces a new metadata type, it is simple to add its support to the app by implementing just a single class and the application will dynamically instantiate it by its predefined name and use it for recognised metadata files.

The most crucial about the whole idea of using a version control system as a source of truth for Salesforce is that the developers must be able to read the metadata files and merge them.

Goals for the future

As mentioned in the implementation review, the solution is missing automatised testing in isolated environments before any deployment is allowed to execute. This desired functionality is analysed in the solution and would be appreciated in further implementation.

One of the biggest goals is to implement continuous deployment. Continuous deployment is built on the top of continuous delivery and goes a little bit further with the whole automatisation. These two differs in the way the deployment to production is executed. In continuous deployment, there is no manual approval needed for this and the code is automatically deployed to production [19]. Before this can become a reality, the metadata must be supported entirely.

Bibliography

[1] What is Software as a Service (SaaS). *Salesfroce*, 2018, [cit. 2019-01-08]. Available at: `https://www.salesforce.com/saas/`

[2] PaaS: Platform as a Service Definition. *Salesfroce*, 2018, [cit. 2019-01-08]. Available at: `https://www.salesforce.com/paas/overview/`

[3] Farley, D.; Humble, J.: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, ISBN 978-0321601919.

[4] Understand the Salesforce Architecture. *Salesforce*, [cit. 2019-01-08]. Available at: `https://trailhead.salesforce.com/en/content/learn/modules/starting_force_com/starting_understanding_arch`

[5] Deploy Enhancements from Sandboxes. *Salesforce*, 2019, [cit. 2019-01-10]. Available at: `https://help.salesforce.com/apex/HTViewHelpDoc?id=deploy_sandboxes_parent.htm&language=en_us`

[6] Metadata Coverage. *Salesforce*, 2019, [cit. 2019-01-10]. Available at: `https://developer.salesforce.com/docs/metadata-coverage/44`

[7] Apex Developer Guide. *Salesforce*, 2019, [cit. 2019-01-08]. Available at: `https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_dev_guide.htm`

[8] Ant Migration Tool Guide. *Salesforce*, 2019, [cit. 2019-01-08]. Available at: `https://developer.salesforce.com/docs/atlas.en-us.daas.meta/daas/meta_development.htm`

[9] Apache Ant™ 1.10.5 Manual. *Apache*, 2018, [cit. 2019-01-11]. Available at: `https://ant.apache.org/manual/index.html`

[10] Package and Distribute Your Apps. *Salesforce*, 2019, [cit. 2019-01-15]. Available at: https://help.salesforce.com/articleView?id=package_distribute_apps_overview.htm&type=5

[11] Unsupported Metadata Types. *Salesforce*, 2019, [cit. 2019-05-05]. Available at: https://developer.salesforce.com/docs/atlas.en-us.api_meta.meta/api_meta/meta_unsupported_types.htm

[12] An SDK for Salesforce development. 2015, [cit. 2019-01-11]. Available at: http://www.illuminatedcloud.com/home/offlinesymboltable

[13] Metadata API Developer Guide. *Salesforce*, 2019, [cit. 2019-01-08]. Available at: https://developer.salesforce.com/docs/atlas.en-us.api_meta.meta/api_meta/meta_intro.htm

[14] Next Generation Agility with Salesforce DX. *David Dawson*, Sep. 2018, [cit. 2019-01-10]. Available at: https://www.rea-group.com/blog/next-generation-agility-with-salesforce-dx/

[15] Plan for Changes to Your Org Unit. *Salesforce*, 2019, [cit. 2019-01-10]. Available at: https://trailhead.salesforce.com/en/content/learn/modules/declarative-change-set-development/plan-for-changes-to-your-org

[16] Scratch Orgs. *Salesforce DX Developer Guide*, [cit 2019-02-10]. Available at: https://developer.salesforce.com/docs/atlas.en-us.sfdx_dev.meta/sfdx_dev/sfdx_dev_scratch_orgs.htm

[17] What is Continuous Integration? *Amazon Web Services*, [cit. 2019-05-05]. Available at: https://aws.amazon.com/devops/continuous-integration/

[18] Kabir, A.: Continuous Integration: A "Typical" Process. *Red Hat Developers*, Sep. 2017, [cit. 2019-05-05]. Available at: https://developers.redhat.com/blog/2017/09/06/continuous-integration-a-typical-process/

[19] What is Continuous Delivery? *Amazon Web Services*, [cit. 2019-05-05]. Available at: https://aws.amazon.com/devops/continuous-delivery/

[20] Git - git-diff-tree Documentation. *Git SCM*, 2019, [cit. 2019-05-15]. Available at: https://git-scm.com/docs/git-diff-tree

[21] Grow faster with Salesforce. *Salesforce*, 2019, [cit. 2018-05-05]. Available at: http://jcommander.org/

[22] Jenkins. *Jenkins*, [cit. 2019-04-22]. Available at: https://jenkins.io/

[23] Amazon EC2. *Amazon Web Services*, [cit. 2019-04-22]. Available at: https://aws.amazon.com/ec2/

[24] Getting Started with Amazon EC2. *Amazon*, [cit. 2019-05-10]. Available at: https://aws.amazon.com/ec2/getting-started/

[25] Bitbucket Push And Pull Request Plugin. *Jenkins*, [cit. 2019-05-13]. Available at: https://wiki.jenkins.io/display/JENKINS/Bitbucket+Push+And+Pull+Request+Plugin

[26] Michael B. Jones, N. S., John Bradley: JSON Web Token (JWT). [cit 2019-05-12]. Available at: https://tools.ietf.org/html/rfc7519

[27] Authorize an Org Using the JWT-Based Flow. *Salesforce*, [cit 2019-05-12]. Available at: https://developer.salesforce.com/docs/atlas.en-us.sfdx_dev.meta/sfdx_dev/sfdx_dev_auth_jwt_flow.htm

[28] Connected Apps. *Salesforce*, [cit 2019-05-12]. Available at: https://help.salesforce.com/articleView?id=connected_app_about.htm

Appendix A

List of Abbreviations

AMI Amazon Machine Image

CI/CD Continuous Integration/Delivery

CLI Command-line interface

CRM Customer relationship management

DML Data manipulation language

EC2 Elastic Compute Cloud

IDE Integrated development environment

JRE Java runtime environment

JSON JavaScript Object Notation

JWT JSON Web Token

QA Quality Assurance

RSA Rivest–Shamir–Adleman

SFDX Salesforce Developer Experience

SIT System Integration Testing

SSH Secure Shell

TCP Transmission Control Protocol

UAT User Acceptance Testing

URL Uniform Resource Locator

XML Extensible Markup Language

Appendix B

Contents of the attached CD

- `readme.txt` ..... CD content description
- `src`
  - `impl` ..... source code
  - `thesis` ..... thesis in LaTeX
- `text`
  - `thesis.pdf` ..... thesis in pdf