Mikhail Timofeev

CONTINUOUS INTEGRATION – THE ENTERPRISE STRATEGY

Development of Maven plugin, promotion of an algorithm

Bachelor's thesis
Information Technology

2018

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

Author	Degree	Time
Mikhail Timofeev	Bachelor of Engineering	May 2018

Title

Continuous integration – the enterprise strategy
Development of Maven plugin, promotion of an algorithm

86 pages
54 pages of appendices

Commissioned by

PJSC Rosbank

Supervisor

Timo Mynttinen

Abstract

The objective of this thesis was to prove that it would be possible to minimise the risk of error getting to the production environment when compiling the application made by several independent developers. This way of development is called continuous integration and is extensively used in large business environments.

The method used for identifying the right way to approach the task was programming. To make it happen the “middleman” layer of software was added to replace the manual work of assembling sources into packages. The programming language was Java and the build system was Maven.

This experiment showed that filtering out most of the errors during the phases of assembly and tests was achievable. It was also proven that distributing smaller task parts among different task executors was not only safe but it also greatly improved the performance and coherence of process overall.

Keywords

continuous integration, Java, Maven, server, thesis

CONTENTS

1 INTRODUCTION 5

2 THEORY 6

2.1 Roles in software development 7

2.2 DevOps 7

2.3 Agile software development, Scrum 8

2.4 Markup languages, XML 8

2.5 Java, JAR files 9

2.6 Software design, modularity 10

2.7 Recommended Java programming practices 10

2.8 Technical specification 12

2.9 Outsourcing 12

2.10 Continuous Integration 12

2.10.1 Integration Layers 13

2.10.2 Version control, GIT 13

2.10.3 Bug tracking system 14

2.10.4 Repository manager 15

2.10.5 Automation server 15

2.10.6 Build system 16

2.10.7 Deployment tool 16

3 EXECUTION 16

3.1 Technologies used 17

3.2 Distribution structure formation 17

3.2.1 Property files for integration layers 18

3.2.2 MQ scripts 19

3.2.3 XSLT files 19

3.2.4 Installation scripts 20
3.2.5 DB scripts 20
3.3 Maven 21
3.3.1 Workspace 21
3.3.2 Package, artifact, distribution 22
3.3.3 IBM's BAR file 22
3.3.4 Maven's custom POM file 22
3.4 The workflow execution 24
3.4.1 P0 – Vendor 25
3.4.2 P1 – GIT 25
3.4.3 P2 – Build 25
3.4.4 P3 – Deployment 26
3.4.5 P4 – QA 26
3.4.6 P5 – Release 27
3.5 Automation Maven plugin 27
4 CONCLUSION 28
REFERENCES 30
APPENDICES
Appendix 1. Maven lifecycle phases
Appendix 2. Example POM file
Appendix 3. Install.sh execution
Appendix 4. Plugin's pom.xml
Appendix 5. Plugin's sources

1 INTRODUCTION

Yesterday's future is today's present where software plays key role in many areas of our lives. From games to complex systems controlling traffic lights and medical equipment – everywhere the code produced by programmers rolls the wheel and "makes things work". Tomorrow programmes may move even closer to us which means that errors made by programmers may turn out disastrous or even fatal.

The successfulness of software production depends on various diverse factors starting from the programmers' experience, including the number of programmers, deadlines, working ethics, software tests etc. To lower the risks we must be more careful in the development process, and to do so we shall improve the equation and replace the insecure variables wherever we can. Human nature leads to unexpected errors in unexpected places, because we tend to get tired, lose concentration, get distracted and so on – the human factor is the main source of problems. According to Wikipedia (2018), prevention of human error would majorly contribute to safety and reliability of complex systems as this kind of behaviour often cannot be foreseen and may easily fall outside acceptable limits and regulations.

The process of software development is complex and consists of massive number of steps. The most commonly used are planning, analysis, design, documentation & implementation, testing, maintenance, according to SynapseIndia’s (2018) article “6 Stages of Software Development Process”. There is a bunch of methodologies (development models) defining the order of these five steps. The traditional single-threaded pattern was rather complex and had multiple points of failure.

Luckily, there is an advanced strategy for application development that systemises the approach and removes most of the unnecessary complexity from a development process called Continuous Integration (CI). An architectural algorithm of CI allows developers to concentrate on the quality of source code without the real need to manually assemble sources, test and conduct other aspects of development automatically processed by the CI tool. In case of failure,

tracker notifications reach all interested parties including the last person to make changes. These principles are rather universal and can be applied to a vast majority of enterprise projects with few to no individual adjustments. As Fowler (2006) suggests, applying concepts of TDD (test driven development) would make it even more profitable because the code would test itself too.

Although, using the CI approach helps a lot in the development, not everyone is aware of its advantages and not all will benefit from it as setting up the system for the first time can be time consuming and may require some level of expertise. Theoretically, it is possible to build and deploy software absolutely without any kind of CI systems (traditional approach) and in many cases that is so – with smaller players. Enterprise businesses, however, tend to develop continuously and thus cannot take a risk with how they do their job. People are not machines and sometimes make mistakes which often result in build failures. Using CI tools eliminates the humane factor that is higher the larger the enterprise is. That is precisely why CI exists – to ensure consistent builds and avoid unpleasant surprises.

Previously the Continuous Integration system at PJSC Rosbank (Societe Generale Group Russia) did not work effectively enough as it was missing a core element at the build phase and continuous delivery. Using custom formats for packaging system required manual work for compilation and setup. The human error factor was a stumbling block for this huge enterprise project for a lengthy period. No doubt that replacing the manual work wherever rational is a way to speed up the development and at the same time make it safer and much more stable. In order to achieve a fully automatic build system, Maven plugin for automatic assembly tasks was created. Then the integration bus was configured for the automated build and deployment of applications and libraries.

2 THEORY

Applying the principles in-place is hardly possible without a good theoretical background. I have compiled a list of relevant topics to deeply understand the nature of the system I am working with. There are several essential IT terms to

get familiar with before studying the full picture. Knowing the exact meaning and usage patterns of each of them ensures the elimination of misunderstanding and, possibly, sets an appropriate climate for the adoption of new ideas. This part is a briefing that allows one to acquire the right context to further follow the narration in the practical part.

2.1 Roles in software development

A *vendor* is a provider of some sort of product. In an IT industry vendor is usually a company that supplies software for specific tasks and supports it technically. Following the technical specification provided by the client vendor meets its obligations in a task.

Testers are the engineers who apply real-life usage scenarios of the distribution to identify all the possible pitfalls before this distribution gets to release stage. They intend to find bugs and verify that the distribution is fit to use.

Quality assurance (QA) is the approach as well as personnel behind the testing process organization. They control the most important areas of distribution testing, communicate with testers and report to the bug tracking system.

2.2 DevOps

Along the duration of this experiment I had an opportunity of trying myself as a DevOps intern. Created as an acronym for “development and operations”, the role of DevOps is to actively participate, help and be involved in the process of development from almost every possible direction.

DevOps has to create and control the cycle of development, the use and deployment of software. This way the organisation is able to lay connected tasks on the shoulders of one person, thus avoiding collisions and disagreements. According to Knupp’s (2014) article “How ‘DevOps’ is killing the developer” the DevOps is meant to unite development, operations and QA roles.

2.3 Agile software development, Scrum

Before IT companies started using smarter approaches to optimise the development, it all went by the traditional "waterfall" (cascade) methodology as of “Software development methodologies” (Association of Modern Technologies Professionals 2018). This linear process required the completion of previous steps before the next steps could begin. As of today, everything is fast-paced, and so should the process of development be in order for the client to get served in time. The waterfall does not work that well anymore.

Currently, many companies attempt to think “short”. Instead of making long-term plans which may or may not come to reality, they utilise the concept of iterative development where each iteration lasts for one to four weeks and requires realistic goals to be accomplished in that time. These are Agile methods of development.

One of the Agile approaches is Scrum. According to the Association of Modern Technologies Professionals (2018) Scrum was developed by Ken Schwaber who suggested having short sprints and brief daily meetings for feedback on the progress and problems. Scrum forces everyone to be productive full-time (always).

2.4 Markup languages, XML

A markup language is a language organised in a way easy for computer to handle. Mixing the human-readable text with markup tags denoted in a certain, identifiable way (for example, by “<” and “>” in XML family) one can obtain the valid markup, following specific rules. Despite being really old XML markup family stays one of the most widely used nowadays. Its common applications are configuration files of every sort, web pages (HTML, XHTML) etc. Since February 1998 XML was a W3C Recommendation. (W3Schools 2018.)

2.5 Java, JAR files

In an ideal world there would be no such term as “cross-platform”, as there would only be one platform suiting everyone’s needs perfectly. However, the current reality proves us the impossible union of thought and feel, and that is where the cross-platform software appears to satisfy everybody, regardless of the OS.

Java is among the most popular programming languages. It is object-oriented, class based, concurrent general-purpose computer programming language that is designed to utilise only the needed functions from dependencies making the programmes as lightweight as possible according to Wikipedia (2018).

JVM, JRE and JDK are the three concepts to get familiar with to get to know Java. Java Virtual Machine is the abstract environment for Java bytecode execution in charge of these four tasks: code loading, code verification, code execution and providing runtime environment. Java Runtime Environment (JRE) is the physical set of libraries and other files making up the implementation of JVM, it is what a user will have to install in order to run Java applications. Java Development Kit (JDK) contains JRE and development tools. (JavaTpoint.com 2018.)

Java *JAR* files, according to “yet another insignificant programming notes...” (2012), are easily distributable platform-independent ZIP archives containing a bundle of files composing a Java application. These archives can be digitally signed by their authors to ensure authenticity. There is no need for extraction as the Java Runtime (JRE) is capable of reading classes directly from JAR.

Naming conventions in Java can be shown with one table (Figure 1), according to JavaTpoint.com (2018).

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

Figure 1. Java naming conventions

2.6 Software design, modularity

According to “Modular programming” (Wikipedia 2018), there is a software design practice that is concerned in separating the functionality of a program into several independent parts each of which is capable of executing only one desired aspect of functionality.

It is convenient when independent developers are able to include functions that are missing from otherwise suitable products without the urgency to invent the wheel again.

2.7 Recommended Java programming practices

There are certain recommendations for how to write a code the best way. One may think of it as unnecessary, on account of the programs written without these considerations still working. Nevertheless, knowing the basic syntax is not

everything it takes to write a good program. Following the suggested guidelines will improve everything from the code readability to general reliability of inlying logical algorithms. Most of the programming languages share similar principles for good coding. The following pieces of advice, however, are Java (object-oriented programming) specific, hence, it is important to keep in mind all the nuances in between of different programming languages prior to blindly applying them. The Java coding practices from the “IBM developerWorks” (2016) follow.

1. First of all, it is important to keep classes as small as possible. While it is definitely possible to tightly pack your Java classes with dozens or even hundreds or methods, it is generally recommended against. Instead, if the structure as such is required at all, it is better to “refactor”. Refactoring is altering the code design without changing the functionality. Each class should “do a small number of things and do them well”.
2. Method names should denote the methods’ purposes. Rather than using some non-descriptive generic letter sequences, one shall have something that shows more meaning. For instance, having “checkValidity()” instead of “c()”. This example also demonstrates the Java naming convention (check Figure 1, part 2.1.4) applied to method naming.
3. Same as keeping classes as small as possible it is profitable to keep methods “maintainable”. For instance, if the method’s length exceeds one page, it may be wise to refactor it. When the method is limited to a single job, it usually does not take more than 30 lines of code and is easy to manage.
4. Code commenting is crucial as it will help anyone reviewing the code later (maybe even the programmer himself) to understand what was meant to happen.
5. Consistency of the selected style may prevent confusion, keep the code clean and help not to get lost in the code. Retaining the habit of having opening braces start right after the method name on the same line and putting closing braces afterwards at the same indentation level every time will make the code much more readable.
6. Last but not least it is highly recommended to use Java’s built-in logger or Log4j library for logging instead of a canonical way (System.out.println()).

2.8 Technical specification

Technical specification (TS) is a formal document explaining the task in detail. It consists of

1) project description;
2) purpose and aims;
3) requirements;
4) description of workflow (tasks and deadlines);
5) the order of control and acceptance;
6) sketches and drafts.

The fees are usually mentioned in separate attachments. However, there are exceptions – the fees are included in the TS. The complexity of making TS is in the fact that the client has to recreate the image of the final result, disassemble it and describe each part. (Arefiev 2017.)

2.9 Outsourcing

In the business world it is common to hire developers who are not permanent employees to solve single-time big tasks. This practice is called outsourcing. Outsourcing specific tasks often reduces overheads as the company does not need, for instance, to buy software and equipment for own developers (who are not needed as well). (Investopedia 2018.)

In order for the company to achieve a certain goal, it is not required to stop the production in progress, giving employees who already have tasks a different TS, nor does the company have to wait for the employees to get ready for a new TS. Instead the company gives the unusual TS to the outsource vendor.

2.10 Continuous Integration

The practice of continuous integration is the business strategy targeted at minimising the gaps between releases by building and testing the packages after every commit, this way eliminating the need to dig too deep into the code in

search of errors. According to Fowler (2006), team members integrate their work frequently, integration is verified by the automation server and errors become revealed quickly, if there are any.

Furthermore, Fowler (2006) implies that having the CI ensures employees will collaborate. Every change made by a team member becomes available to other team members. CI server's web pages (Cruise or Jenkins) show information on builds' success, who made changes, which changes were made, calendar with history.

2.10.1 Integration Layers

Integration Layers are the preconfigured working environments for either testing or running distributions in production. ILs are usually located at different machines physically, thus they have different IPs. Other configurations may differ as well (logins, passwords etc.). Test layers being configured the same way as a production one make sense – errors and successes from test environment would be repeated in the real application.

2.10.2 Version control, GIT

GIT is a source control system that allows the developers to safely experiment with adding new functionality without the risk of breaking what already exists. Controlling source is done by tracking changes.

GIT's mechanisms of versioning allow developers to safely experiment with editing the code, being able to roll back in case it is needed – the process of collaborative development becomes easier. It is also possible that the developers decide to take different approaches by splitting the project tree into different branches: different versions based on the same "parent" version.

When there are changes to source code, it is important to save them. In terms of GIT the save process is called COMMIT. This is a trigger we invoke to save changes we made in a local repository. COMMIT produces a PUSH system

notification. It makes sure all interested parties get to know about changes. Invoking a PUSH event applies our committed changes to the remote repository. At best the meanings of COMMIT and PUSH can be illustrated by the illustration from Oliver Steele (2008) (Figure 2).

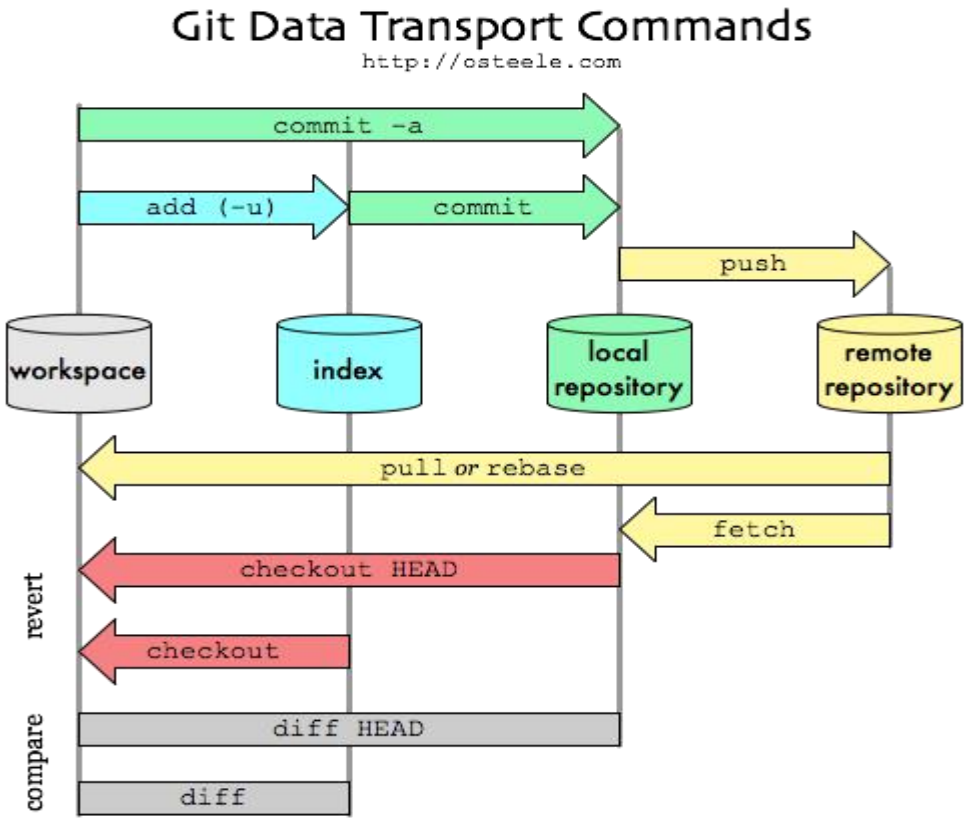


Figure 2. Git data transport commands (Steele 2008)

2.10.3 Bug tracking system

Bug tracking system (or simply – bug tracker, task manager) is an advanced planning and bug tracking tool. It is needed for the coders to get feedback from the testing group. Not only does it keep bug reports organized but also allows making requests and planning new releases, and provides a more transparent view of current situation. Following Kuzmich (2012), there are many possible uses of the tracker. Some of the less obvious uses are listed below (Figure 3).

Possible tracker uses:

- help desk of tech support – it is handy to tie request tickets to bugs, not as “native” as a separate help desk but, nevertheless, commonly used
- bureaucracy – tracking document edits, purchases, contracts etc.
- batch mailing – after changes are made, subscribers get notified
- resources distribution – to check availability and book (eg. to check if the server, tablet computer or the car is available or someone borrowed it already)
- voting – subscribers vote for project “version” – option and the votes are counted
- test management – subroutines of test successes/failures allow report generation

Figure 3. List of non-obvious tracker uses

Moreover, remote access interfaces (or CLIs) allow automating posts on events which opens unlimited horizons.

2.10.4 Repository manager

Repository manager is a tool for storing package releases. It provides reliable access to resources; the upload can be secured by authentication. According to OBrien (2009), the repository manager has two main goals: being a proxy for remote repositories for artifact caching and hosting internal artifacts.

2.10.5 Automation server

When it is required to run some sort of a programmable function for other applications without human interaction, an automation server comes to play. According to Microsoft (2018), the automation server exposes automation objects (programmable objects) to automation clients (other applications). This allows the direct calls to foreign functions and greatly increases the number of possibilities for development.

2.10.6 Build system

In a nutshell, a build system is a smart programme that executes a certain predefined compilation scenario every time the input is changed. The build system is idempotent which means it does not recompile output if the same input was already present once. This makes it an extremely efficient tool for compilation from sources. (Williams 2009.)

2.10.7 Deployment tool

A deployment tool is needed for installing distributions to the Integration Layer. The deployment tool also keeps a record of deployment. Also called the “automated deployment software”, these tools make it easy for team members to deploy safely many times per day because there is a rollback feature (roll out healthy deployment), according to Mitchell (2017).

3 EXECUTION

The understanding of the system work’s general principles is more crucial than knowing how exactly each part operates. That knowledge is the most important part of this experiment. The algorithm is the foundation without which the application of technology to the appropriate parts is dull and worthless. However, to get a deeper understanding of the specific case we must know a few details.

The assembly and deployment of packages happens on top of IBM’s Integration Bus with the help of IBM MQ (messaging and queueing middleware) and UrbanCode Deploy (deployment system). There is a total of four ILs (integration layers): debug, test, cert and prod in this experiment’s environment. The source code is outsourced.

At the beginning of this experiment, the bank’s integration system was not fully automatic and heavily relied on an operator’s assistance for deployment. There was a need for software to do “negotiations” with deployment tool: providing ready-made packages and triggering deployment event. That is precisely why I created the automation plugin.

3.1 Technologies used

BitBucket is an open source and free to use GIT system, according to Atlassian, Inc. (2018). Source code is saved to the project IIBS GIT where the structure is “one repository – one package”. There are three types of packages: applications, individual application libraries (used solely by one application) and common libraries (in use by multiple applications).

Jira is an advanced bug tracking system. It assists in finding and recording bugs, tracking bug resolution and change history. Originally, released as a solely bug tracking system in 2003 it was later expanded with additional functions to help teams with all aspects of development cycle, as of “Bug tracking done right” (Atlassian 2018).

NEXUS OSS is the repository manager in our case. It has advantages of being free and supporting the most popular formats according to the official page “Nexus Repository OSS” (Sonatype 2017).

*Jenkins wa*s the chosen automation server. It can easily be used as a continuous integration shell. That simplifies the process of software development by automating tasks that can be executed without human interaction. It is an open source technology accompanied with hundreds of plugins (Jenkins 2018).

UrbanCode Deploy it is a tool for “continuous delivery in agile development”, according to IBM Community (2017). Surprisingly, it has very simple and intuitive GUI that allowed me to test deployment to the target (test) integration layer in a few dozens of clicks, drags and drops. One more possible alternative the DevOps team seriously considered was Ansible, which is a free utility, slightly harder to master.

3.2 Distribution structure formation

The file “build.cmd” is an accessory to creating distribution and storing it to Nexus OSS. To properly use this tool we should provide four arguments for execution:

NAME – artefact id (eg. Acs.Bis.FindCustomer), GROUP – execution group (eg. acs.find), TYPE – application type (eg. acs, adp etc.) and VER – bar file version included in the distribution (ex. 1.01). The resulting archive contains BAR, MQ, Install and XSLT folders (Figure 4).

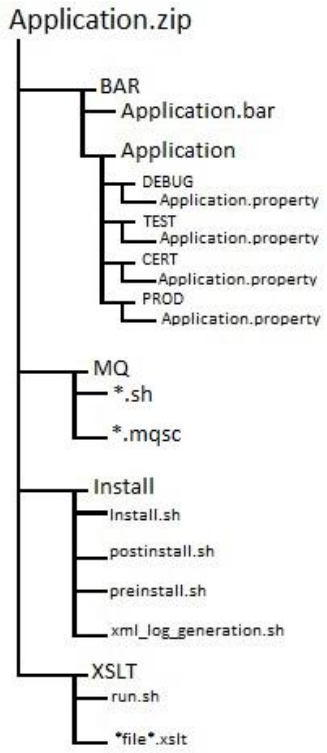


Figure 4. Distribution structure

3.2.1 Property files for integration layers

These property files describe installation configurations for each of the Integration Layers. They are configured individually for each IL and distribution (Figure 5). Settings are formed by the parameters provided by the person in charge. In case the property files are not provided, the default settings are used – they are identical for all ILs. Configuration files are originally found in the source code as "*_template_min.properties" files. All of the parameters are stated as "thread_name#configuration_parameter=value".

```
<execution group name>
   |--- <file bar name>.bar
   |--- <file bar name>
      |--- debug
         |--- <application name>.properties
      |--- test
         |--- <application name>.properties
      |--- cert
         |--- <application name>.properties
      |--- prod
         |--- <application name>.properties
```

Figure 5. BAR file and deployment properties

3.2.2 MQ scripts

These scripts contain IBM's commands and create queues. In the distribution they are placed in a separate "MQ" folder. When provided with a "REPLACE+" parameter the scripts make a simple recreation of the existing queues possible or, if there are none, a creation of new ones. They exist in every distribution (Figure 6). Also, if necessary, scripts for queue automation must exist for specific users. Files with the "mqsc" extension are scripts for queue creation. There can be several files containing JDBC Provider, permission approval for users etc.

```
…
   |---MQ
      |--- *.sh
      |--- createQueues.mqsc
      |--- …
      |--- *.mqsc
```

Figure 6. MQ catalogue structure

3.2.3 XSLT files

These files describe message conversion. The rules must be in a distribution in a separate "xslt" folder. The property file must state a path to these files in the settings as "…#stylesheetPath=<path_to_file>". Additionally, the catalogue must contain bash script for copying files. The compulsory requirement of scripts of that kind is containing a target copying path and stating file extensions. The catalogue tree of target destination must be as similar as possible to used IBM IB's execution group's tree (Figure 7).

```
…
|---XSLT
   |--- run.sh →
   |--- *.XSLT
   |--- *.XSL
```

```
#!/bin/bash
export
XSLT_INSTALL_DIR=<File_copy_path(ex:/mqshared/project/
H2H/XSLT/)>
if [ ! -d "$XSLT_INSTALL_DIR" ]; then
   mkdir -p $XSLT_INSTALL_DIR
fi
cp *.xsl $XSLT_INSTALL_DIR
```

Figure 7. XSLT catalogue structure and copy script

3.2.4 Installation scripts

These scripts are needed to install the distribution to the target IL (Debug, Test, Cert and Prod). They are simple Linux Bash files with instructions for execution before, during and after the installation. The exact algorithms can be seen in appendix 2.

3.2.5 DB scripts

These scripts create message routing rules and database components. Currently, they are shipped as a separate routing distribution which is not a good practice. Embedding it to the distribution is a better way. Some application types are reused which leads to the DB scripts being reused as well. Also, if additional objects at the DB level are necessary, additional scripts for creation, alteration or deletion may be added. Scripts for rollback should also be included. Currently, when rules change, a separate distribution is formed (Figure 8).

- Existing rules are taken from the production IL.
- Changes are manually taken from new/altered scripts gained from developers and added to the rules.
- Check happens, test on the Debug IL and HASH-sum calculation to avoid unauthorized rules' edits.
- Distribution is passed to testing.

Figure 8. Distribution formation

3.3 Maven

According to Apache (2018), Maven is software comprehension and project management tool. It is build system that acts according to the instructions defined in a POM (Project Object Model) file. By default Maven is able to produce JAR, WAR, EAR and POM files which are called artifacts. Artifact production settings are set in the POM file which must be placed into the directory with source code.

The core concept Maven is based around is a build lifecycle. The artifact building and distributing process is clearly defined. There is a total of three build lifecycles by default in Maven: “clean”, “default”, and “site”. (Apache 2018.) The full list of default Maven phases can be illustrated by a table (Appendix 1) from Apache (2018).

Straightforward as it is, Maven follows the instructions from the POM file of the project and returns the resulting file. The problem was, however, in the unsupported format of the output file. Thus, the need to create a Maven plugin arose.

3.3.1 Workspace

Logically workspace is the environment where we will work with our builds. In case of Maven within Jenkins a workspace is a directory and its subdirectories for the package build. It contains all the required files for the build process: sources + configuration files. The directory name is the same as a package name. Any additional libraries should exist in the workspace’s root (Figure 9). The sources’ folder contains the POM file required by Maven. That folder is also called appspace.

```
<workspace>
  |---<package_name>
      |--- <source_code>
      |--- pom.xml
      |--- <dependencies>
```

Figure 9. Typical workspace

3.3.2 Package, artifact, distribution

There are several stages the application or library goes through before getting deployed to the integration layer:

- “Package” is the first stage. This is a source code compiled.
- “Artifact” is the second stage. It is an instance created by Maven.
- Distribution is a union of files meant to be a product for a deployment tool (delivery system). It consists of an executable file, property files and accessory scripts to set up the environment.

3.3.3 IBM’s BAR file

This is the distribution’s executable file. Packages used by the company are incompatible with the default Maven’s configuration, as it was mentioned above. There are two types of packages: applications and libraries. The application (BAR file / broker archive) is an executable containing everything that is needed for the deployment to the target environment as of IBM Community (2017). Library (ZIP file) is an archive that ensures the correct functioning of the executable (application library) or group of executables (common library).

3.3.4 Maven’s custom POM file

Automation Maven plugin is the final (previously missing) core element for the automatic package (ZIP library or BAR application) creation with Maven. The plugin is written in Java. It is a JAR file that contains Java classes. However, to handle the proprietary IBM’s format, a custom crafted Maven’s POM file is required.

POM is a subset of the XML document that is written in an object-oriented style, defining the properties of a project, build parameters, dependencies etc. within the appropriate tags. These tags that are used by Maven to generate packages (Maven artifacts). To create a correct POM file we must rely on the information from the ".project" file that is made by developers (vendor).

- <*project*> is a root/parent tag that contains all the other tags.
- <*groupId*> tag contains the group ID from the NEXUS OSS, where the artifact will be placed, defines the type of package. This project's products (BAR applications and ZIP libraries) always start with "ru.rsb.esb.". This is a typical Java naming convention's real life example for marking the products of the same family. Library archives (ZIP), on the one hand, will always have "ru.rsb.esb.**lib**" groupId where "lib" stands for library. Applications (BAR), on the other hand, will only have a part "ru.rsb.esb.**app**." that is constant where "app" part stands for "application" followed by TYPE. The type part, however, differs for applications different by type: the "adapter" applications will have type "adp", services – "acs" etc. In this example we are building application ("app") package with type "adp" – adapter, hence, the groupId is "ru.rsb.esb.app.adp".
- <*artifactId*> tag stands for artifact ID – the unique package name. In this example the package name is "Adp.H2H".
- <*version*> tag contains the package version name. Versions are numerical strings with dots at the release stage. Pre-release packages, however, have a "-SNAPSHOT" addition after the version number (eg. "<version>1.01**-SNAPSHOT**</version>"). They are placed to different repository branch for snapshots. In this example package the name is "Adp.H2H". Forgetting to change the version name (deploying the same version twice) will cause the task to fail.
- <*barDep*> tag highlights the library required for the BAR build. In this example for successful result the library Message.Utils.Java needs to exist in the workspace. If we are creating a library, we must remove the tag or leave it empty. Otherwise, the task fails.

- In the <*dependencies*> tag we define dependencies that are needed to build our package. The dependencies need to be placed into the workspace. This is done automatically by the automation plugin. Dependencies contain individual application libraries and common libraries (including BAR compilational dependency from the <barDep> tag, if we are building application).
- Each individual dependency line is placed within the <*dependency*> tag that also contains several tags: <groupId>, <artifactId>, <version>, <type> and <scope> (Figure 10). If the dependencies are libraries, the type is typically "zip". The scope depends on at what phase the dependency is required.

```
<dependency>
        <groupId>ru.rsb.esb.lib</groupId>
        <artifactId>Esb.Lib</artifactId>
        <version>[1,)</version>
        <type>zip</type>
        <scope>compile</scope>
</dependency>
```

Figure 10. Typical dependency

The example POM file can be seen in Appendix 1. It describes the artifact with an id "Adp.H2H" from the "ru.rsb.esb.app.adp" group – the BAR application.

3.4 The workflow execution

Sounding like a vague and inconsistent process, in fact, CI has a strict and well defined structure. The simple definition of workflow would be "Get code (phase 1)> Build and store package (phase 2)> Tests and deployments (phases 3, 4) > Release (phases 3, 5)" (Figure 11). The part I work on is located at step 04. The final goal is to provide a working packages compiled from vendor's file storage to the bank (IL Prod). There is a total of five phases: P0 – Vendor, P1 – GIT, P2 – Build, P3 – Deployment, P4 – QA and P5 – Release.

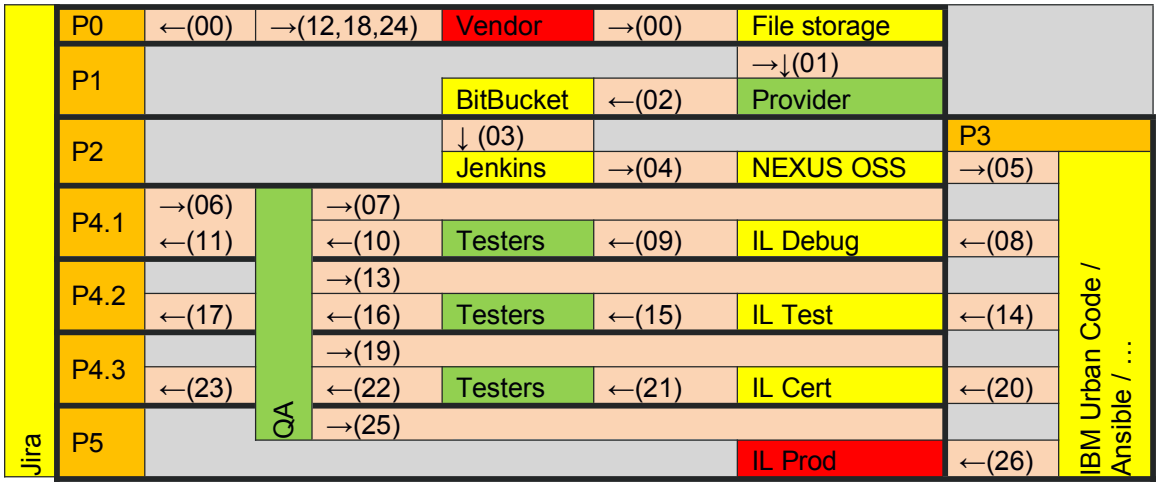


Figure 11. Workflow schema

3.4.1 P0 – Vendor

Goals: read reports from Jira, prepare sources and report to Jira.

12, 18, 24) Vendor reads reports and decides if patches are needed;
00) Vendor gets a task's technical specification and creates application submitting the source code to Storage and writes submission notes to Jira.

3.4.2 P1 – GIT

Goals: take sources from the vendor's storage, commit sources to BitBucket.

The provider is a bank employee in charge of retrieving source code from vendor's storage to bank's GIT.

01) The provider gets notification from Jira, takes the vendor's code from File storage (vendor's GIT);
02) The provider places source code to BitBucket (bank's GIT), the commit of changes takes place.

3.4.3 P2 – Build

Goals: download sources from BitBucket, build and store package to NEXUS OSS.

03) BitBucket makes a PUSH notification about a new commit. After getting the PUSH notification Jenkins downloads the changes of sources;
04) Jenkins builds a package with Maven:
   a. the workspace for package compilation is created:
      i. directory structure is formed,
      ii. required artifacts (ZIP libraries) are downloaded from NEXUS OSS,
      iii. these artifacts are extracted according to auxiliary POM file,
   b. Jenkins builds a working package,
   c. Jenkins saves a new package to NEXUS OSS to a target branch with the version specified in the POM file.

3.4.4 P3 – Deployment

Goals: prepare distribution for deployment to integration layer, deploy to IL on QA's command.

05) Deployment tool downloads a package from the NEXUS OSS repository, four property files are manually added to the UrbanCode project;
08, 14, 20, 26) Deployment tool deploys package to Integration Layer X.

3.4.5 P4 – QA

Goals: QA reads Jira and decides to start deployment, testers test IL package, testers report to QA, QA reports to Jira.

06, 99, 99) Quality Assurance (QA) reads release notes from Jira;
07, 13, 19) QA allows deployment to IL;
08, 14, 20 – P3;
09, 15, 21) Integration Layer (IL) X out system runs packages and Testers read logs from there;
10, 16, 22) Testers report to QA;
11, 17, 23) QA reports success or failure to Jira;
12, 18, 24 – P0.

3.4.6 P5 – Release

Goals: Check Jira, if package is OK – deploy.

25) QA allows deployment;
26) Production Integration Layer runs packages.

3.5 Automation Maven plugin

Each of the classes in the plugin's JAR plays a specific role such as communicating with Maven, handling command execution etc. Once the execution starts, the plugin selects the right goal (create, deploy etc.) and then starts the appropriate task. The plugin itself was build with Maven as well (Appendix 3).

3.5.1.1 Command, Clean, Package, Install and Deploy

This class executes Maven and other commands in the target (UNIX or Windows) environment as well as provides logging functionality, functions to retrieve running and error states etc.

Clean.class, Package.class, Install.class and Deploy.class are the placeholder classes for Maven that point to the "clean", "create" and "deploy" methods in the App.class class respectively. They are executed by Maven only.

3.5.1.2 Zip and App

Zip.class contains methods to make a proper and clean library ZIP archive, leaving out unnecessary files (left by IDEs) that would otherwise enlarge archives with no reason. Together with the App.class this class can deploy libraries (ZIP is created by this class).

Broker archive packages' creation method differs. BAR files are compiled by a different command and do not require Zip.class class. Broker archives are made

by *query()* method of App.class. Basically, App.class is the most important class in the plugin as it contains most of the subroutines for creation and deployment of the broker archive or library package as well as all Maven-related tasks.

3.5.1.3 Standalone

It is possible to run the plugin as an ordinary Java (JAR) file with Standalone.class. While for the typical Maven plugin it is not required to have the main executable class execution is started using classes that contain @goal defined – each of them is invoked at its own workflow phase. Running this JAR file is suitable for debugging purposes from the console (terminal, PowerShell etc.). It takes one optional argument: appspace (sources folder within workspace). Alternatively, it is run in the current working directory.

4 CONCLUSION

This system is now successfully applied at PJSC Rosbank (Societe Generale Group Russia). This experiment was a successful demonstration of replacing unreliable chain elements with solid ones. And business is the place where unnecessary risks are better avoided in favour of reliability.

The true algorithm of continuous integration can be tricky to achieve in an environment deeply rooted based on the different idea. However, rebuilding a similar structure or starting fresh is a solid foundation for that. The algorithm proves to be worthy both by cost efficiency (outsourcing) and from time perspective (no need to rebuild everything in panic as errors are found immediately).

Being an addon for the extremely versatile build system my plugin is not only relevant in bank environments. Indeed, the algorithm behind it can be used in various setups. Talking solely about IBM compatible build systems, we can assume that by altering a few parameters we would easily replace manual labour in every workflow of IBM’s partners. In the future the plugin written by me will possibly be extended to fully replace proprietary deployment solution (UrbanCode

Deploy), to handle more formats, adapting to everyday changes of business world.

REFERENCES

Apache. 2018. Introduction to the build lifecycle. WWW document. Available at: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html [Accesses 18 May 2018].

Apache. 2018. Welcome to Apache Maven. WWW document. Available at: https://maven.apache.org/ [Accesses 18 May 2018].

Arefiev A. 2017. Technical specification. WWW document. Available at: http://www.alexcouncil.com/tehnicheskoe-zadanie/ [Accessed 6 January 2018].

Association of Modern Technologies Professionals. 2018. Software Development Methodologies. WWW document. Available at: http://www.itinfo.am/eng/software-development-methodologies/ [Accessed 8 January 2018].

Atlassian. 2018. Bitbucket. WWW document. Available at: https://bitbucket.org/ [Accessed 18 May 2018].

Atlassian. 2018. Bug tracking done right. WWW document. Available at: https://www.atlassian.com/software/jira/bug-tracking [Accessed 4 May 2018].

Fowler M. 2006. Continuous integration. PDF document. Available at: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf [Accessed 18 May 2018].

IBM Community. 2017. IIB Integration With UrbanCode Deploy. WWW document. Available at: https://developer.ibm.com/urbancode/2017/05/18/iib-integration-urbancode-deploy/ [Accessed 14 November 2017].

IBM Community. 2017. UrbanCode Deloy – Deployment Automation. WWW document. Available at:

https://developer.ibm.com/urbancode/products/urbancode-deploy/ [Accessed 4 May 2018].

IBM developerWorks. 2016. Unit 12: Writing good Java code. WWW document. Available at: https://www.ibm.com/developerworks/library/j-perry-writing-good-java-code/index.html [Accessed 17 May 2018].

Investopedia. 2018. Outsourcing. WWW document. Available at: https://www.investopedia.com/terms/o/outsourcing.asp [Accessed 18 May 2018].

JavaTpoint.com. 2018. Difference between JDK, JRE and JVM. WWW document. Available at: https://www.javatpoint.com/difference-between-jdk-jre-and-jvm [Accessed 18 May 2018].

JavaTpoint.com. 2018. Java Naming conventions. WWW document. Available at: https://www.javatpoint.com/java-naming-conventions [Accessed 17 May 2018].

Jenkins. 2018. Jenkins. WWW document. Available at: https://jenkins.io/ [Accessed 18 May 2018].

Kharchuk O. 2016. Everyday releases are not so scary. WWW document. Available at: https://habrahabr.ru/company/dataart/blog/276901/ [Accessed 4 May 2018].

Knupp J. 2014. How 'DevOps' is killing the developer. WWW document. Available at: https://jeffknupp.com/blog/2014/04/15/how-devops-is-killing-the-developer/ [Accessed 4 May 2018].

Kuzmich M. 2012. Not a bug tracker, but… WWW document. Available at: https://habrahabr.ru/post/144122/ [Accessed 30 November 2017].

Microsoft. 2018. Automation Servers. WWW document. Available at: https://msdn.microsoft.com/en-us/library/6wx53dax.aspx [Accessed 4 May 2018].

Mitchell. 2017. Six essential software deployment tools for error-free applications. WWW document. Available at: https://raygun.com/blog/software-deployment-tools/ [Accessed 18 May 2018].

OBrien T. 2009. What is a repository manager. WWW document. Available at: https://blog.sonatype.com/2009/04/what-is-a-repository-manager/ [Accessed 18 May 2018].

Sonatype. 2017. Nexus Repository OSS. WWW document. Available at: https://www.sonatype.com/nexus-repository-oss [Accessed 14 November 2017].

Steele O. 2008. My git workflow. WWW document. Available at: https://blog.osteele.com/2008/05/my-git-workflow/ [Accessed 18 May 2018].

SynapseIndia. 2018. 6 Stages of Software Development Process. WWW document. Available at: https://www.synapseindia.com/6-stages-of-software-development-process/141 [Accessed 18 May 2018].

W3Schools. 2018. Introduction to XML. WWW document. Available at: https://www.w3schools.com/xml/xml_whatis.asp [Accessed 18 May 2018].

Wikipedia. 2018. Human error. WWW document. Available at: https://en.wikipedia.org/wiki/Human_error [Accessed 18 May 2018].
Wikipedia. 2018. Java (programming language). WWW document. Available at: https://en.wikipedia.org/wiki/Java_(programming_language) [Accessed 4 May 2018].

Williams D. 2009. Build System Philosophy. WWW document. Available at: https://www.cs.virginia.edu/~dww4s/articles/build_systems.html#id6 [Accessed 4 May 2018].

Yet another insignificant programming notes... 2012. WWW document. Available at: https://www.ntu.edu.sg/home/ehchua/programming/java/J9d_Jar.html [Accessed 18 May 2018].

MAVEN LIFECYCLE PHASES

Clean Lifecycle

pre-clean	execute processes needed prior to the actual project cleaning
clean	remove all files generated by the previous build
post-clean	execute processes needed to finalize the project cleaning

Default Lifecycle

validate	validate the project is correct and all necessary information is available.
initialize	initialize build state, e.g. set properties or create directories.
generate-sources	generate any source code for inclusion in compilation.
process-sources	process the source code, for example to filter any values.
generate-resources	generate resources for inclusion in the package.
process-resources	copy and process the resources into the destination directory, ready for packaging.
compile	compile the source code of the project.
process-classes	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
generate-test-sources	generate any test source code for inclusion in compilation.
process-test-sources	process the test source code, for example to filter any values.
generate-test-resources	create resources for testing.
process-test-	copy and process the resources into the test destination

resources	directory.
test-compile	compile the test source code into the test destination directory
process-test-classes	post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes. For Maven 2.0.5 and above.
test	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
prepare-package	perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. (Maven 2.1 and above)
package	take the compiled code and package it in its distributable format, such as a JAR.
pre-integration-test	perform actions required before integration tests are executed. This may involve things such as setting up the required environment.
integration-test	process and deploy the package if necessary into an environment where integration tests can be run.
post-integration-test	perform actions required after integration tests have been executed. This may including cleaning up the environment.
verify	run any checks to verify the package is valid and meets quality criteria.
install	install the package into the local repository, for use as a dependency in other projects locally.
deploy	done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Site Lifecycle

pre-site	execute processes needed prior to the actual project site generation

site	generate the project's site documentation
post-site	execute processes needed to finalize the site generation, and to prepare for site deployment
site-deploy	deploy the generated site documentation to the specified web server

EXAMPLE POM FILE

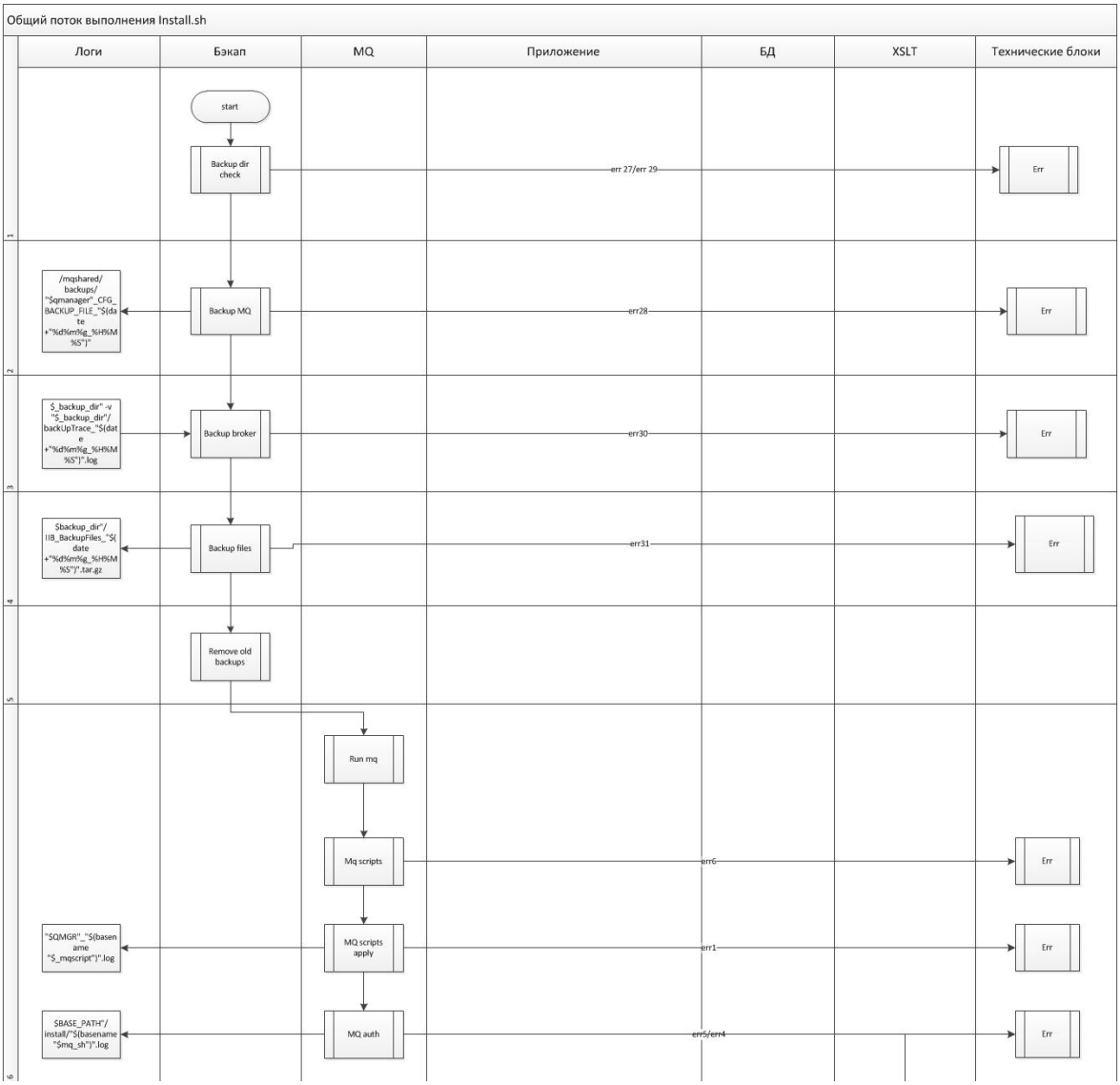
```
<project
      xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
> <!-- Standard project declaration -->
   <modelVersion>4.0.0</modelVersion> <!-- Standard model version -->
   <groupId>ru.rsb.esb.app.adp</groupId> <!-- Artifact's group -->
   <artifactId>Adp.H2H</artifactId> <!-- Artifact's ID -->
   <version>1.00.05</version> <!-- Artifact's version -->
   <packaging>pom</packaging> <!-- Package format (to prevent Maven from
messing with compilation business) -->
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding> <!--
Encoding: Unicode -->
      <barDep>Message.Utils.Java</barDep> <!-- Library for BAR build. If the ZIP
library is build, leave the tag blank. -->
   </properties>
   <build>
      <plugins>
         <plugin> <!-- Enabling automation plugin -->
            <groupId>ru.rsb.esb.mvn</groupId>
            <artifactId>automation-maven-plugin</artifactId>
            <version>2.2</version>
         </plugin>
         <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-deploy-plugin</artifactId>
            <version>2.6</version>
         </plugin>
         <plugin> <!-- Disabling default JAR build plugin -->
```

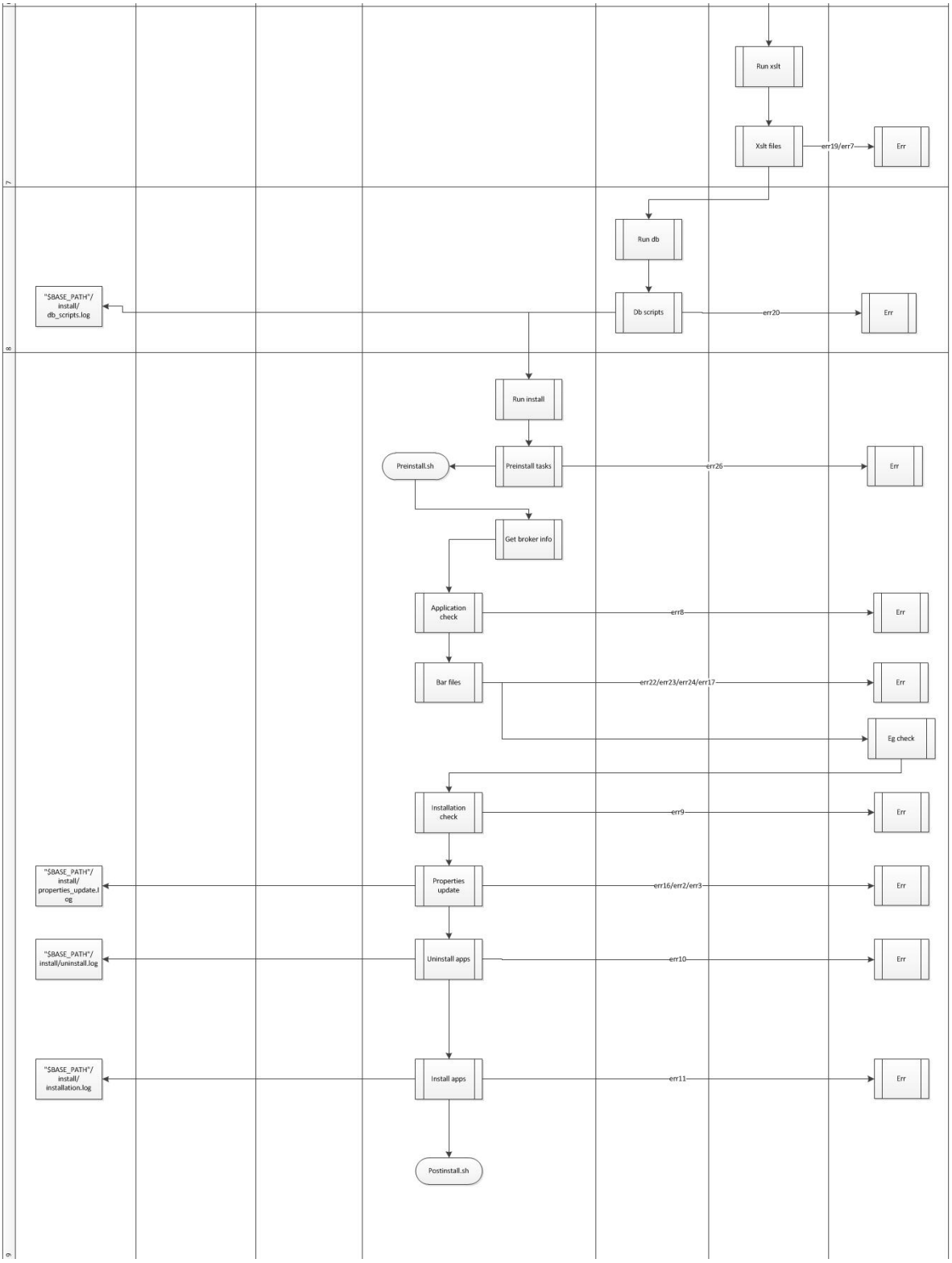
```
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.4</version>
            <executions>
               <execution>
                  <id>default-jar</id>
                  <phase/>
               </execution>
            </executions>
         </plugin>
      </plugins>
      <pluginManagement>
         <plugins>
            <plugin> <!-- Skipping default deploy plugin -->
               <groupId>org.apache.maven.plugins</groupId>
               <artifactId>maven-deploy-plugin</artifactId>
               <configuration>
                  <skip>true</skip>
               </configuration>
            </plugin>
            <plugin> <!-- Starting automation package deploy -->
               <groupId>ru.rsb.esb.mvn</groupId>
               <artifactId>automation-maven-plugin</artifactId>
               <version>2.2</version>
               <executions>
              <execution> <!-- Clean build (optional) -->
                     <id>clean</id>
                     <goals>
                        <goal>clean</goal>
                     </goals>
                     <phase>clean</phase>
                  </execution>
                  <execution> <!-- Make package -->
```

```
                <id>package</id>
                <goals>
                    <goal>package</goal>
                </goals>
                <phase>package</phase>
            </execution>
            <execution> <!-- Deploy package to local -->
                <id>install</id>
                <goals>
                    <goal>install</goal>
                </goals>
                <phase>install</phase>
            </execution>
            <execution> <!-- Deploy package to remote -->
                <id>deploy</id>
                <goals>
                    <goal>deploy</goal>
                </goals>
                <phase>deploy</phase>
            </execution>
        </executions>
    </plugin>
</plugins>
</pluginManagement>
</build>
<dependencies> <!-- Pointing out dependencies for package build (including
the main library if we build BAR!) -->
    <dependency>
        <groupId>ru.rsb.esb.lib</groupId>
        <artifactId>Esb.Lib</artifactId>
        <version>[1,)</version>
        <type>zip</type>
        <scope>compile</scope>
```

```
        </dependency>
        <dependency>
            <groupId>ru.rsb.esb.lib</groupId>
            <artifactId>Adp.H2H.Java</artifactId>
            <version>[1,)</version>
            <type>zip</type>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>ru.rsb.esb.lib</groupId>
            <artifactId>Message.Utils.Java</artifactId>
            <version>[1,)</version>
            <type>zip</type>
            <scope>compile</scope>
        </dependency>
    </dependencies>
    <distributionManagement> <!-- Target repository for deploy -->
        <repository>
            <id>distribution.repository</id>
            <name>distribution.repository</name>
            <url>http://10.45.35.204:8000/nexus/content/repositories/releases/</url>
        </repository>
    </distributionManagement>
</project>
```

INSTALL.SH EXECUTION




Run xslt
Xslt files
err19/err7
Err
Run db
"$BASE_PATH"/
install/
db_scripts.log
Db scripts
err20
Err
Run install
Preinstall.sh
Preinstall tasks
err26
Err
Get broker info
Application
check
err8
Err
Bar files
err22/err23/err24/err17
Err
Eg check
Installation
check
err9
Err
"$BASE_PATH"/
install/
properties_update.l
og
Properties
update
err16/err2/err3
Err
"$BASE_PATH"/
install/uninstall.log
Uninstall apps
err10
Err
"$BASE_PATH"/
install/
installation.log
Install apps
err11
Err
Postinstall.sh

POM.XML

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd"
>
  <modelVersion>4.0.0</modelVersion>
  <groupId>ru.rsb.esb.mvn</groupId>
  <artifactId>automation-maven-plugin</artifactId>
  <packaging>maven-plugin</packaging>
  <version>2.2</version>
  <name>automation-maven-plugin Maven Mojo</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <version>3.0.2</version>
          <configuration>
            <archive>
              <manifest>
                <mainClass>com.mastertimmi.Standalone</mainClass>
              </manifest>
            </archive>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
    <plugins>
      <plugin>
```

```
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <!-- Dependencies to compile this plugin -->
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-plugin-api</artifactId>
      <version>3.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.maven.plugin-tools</groupId>
      <artifactId>maven-plugin-annotations</artifactId>
      <version>3.4</version>
      <scope>provided</scope>
    </dependency>
    <!-- Dependencies for building other projects -->
    <dependency>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-help-plugin</artifactId>
      <version>2.1.1</version>
      <scope>provided</scope>
    </dependency>
```

```
    <dependency>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.8</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <pluginRepositories>
    <pluginRepository>
      <id>distribution.repository</id>
      <url><URL/to/git>.git</url>
    </pluginRepository>
  </pluginRepositories>
  <repositories>
    <repository>
      <id>distribution.repository</id>
      <releases>
        <enabled>true</enabled>
      </releases>
      <url><URL/to/git>.git</url>
    </repository>
  </repositories>

  <distributionManagement>
    <repository>
      <id>distribution.repository</id>
      <name>distribution.repository</name>
      <url><URL/to/nexus-oss>/content/repositories/releases/</url>
    </repository>
  </distributionManagement>
</project>
```

PLUGIN SOURCE CODE

Clean.java

```
package com.mastertimmi;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoFailureException;

/**
 * @author masterTimMi
 * @since 2.0.6
 * @goal clean
 */
public class Clean
   extends AbstractMojo
{
   // Main execution for Maven
   public void execute ()
      throws MojoFailureException
   {
      if (!App. prepare (0))
      {
         throw new MojoFailureException ("FAILURE");
      }
   }
}
```

Package.java

```
package com.mastertimmi;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoFailureException;
```

```
/**
 * @author masterTimMi
 * @since 2.0.6
 * @goal package
 */
public class Package
    extends AbstractMojo
{
    // Main execution for Maven
    public void execute ()
        throws MojoFailureException
    {
        if (!(App. create (true)))
        {
            throw new MojoFailureException ("FAILURE");
        }
    }
}
```

Install.java

```
package com.mastertimmi;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoFailureException;

/**
 * @author masterTimMi
 * @since 2.0.6
 * @goal install
 */
public class Install
    extends AbstractMojo
{
    // Main execution for Maven
    public void execute ()
```

```
        throws MojoFailureException
    {
        if (!(App. save (true, true)))
        {
            throw new MojoFailureException ("FAILURE");
        }
    }
}
```

Deploy.java

```
package com.mastertimmi;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoFailureException;

/**
 * @author masterTimMi
 * @since 2.0.6
 * @goal deploy
 */
public class Deploy
    extends AbstractMojo
{
    // Main execution for Maven
    public void execute ()
        throws MojoFailureException
    {
        if (!(App. save (true, false)))
        {
            throw new MojoFailureException ("FAILURE");
        }
    }
}
```

Command.java

```
package com.mastertimmi;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * Command execution engine
 */
public class Command
{
   static boolean running;          // status: stopped (false), running (true)
   static boolean hasErrors;        // if there were errors

   private static String outLine = "";        // string generated instead of logs

   /**
    * Command constructor
    * @param cmdQuery      command to execute
    * @param errorPattern  error pattern to identify problem
    * @param makeLogs      return logs as String (no visible output)
    */
   Command
   (
      String cmdQuery,
      String errorPattern,
      boolean makeLogs
   )
   {
      // setting variables
      outLine = "";
      running = true;
      hasErrors = false;
```

```
int linesOfOutput = 0;
Process p;

try
{
   p = Runtime. getRuntime (). exec
   (
      App. makeStr
      (
         App. windows?         // if Windows,
            "cmd /c \" chcp 1251 & ":      //  we change Windows encoding,     < command for Windows
            "",                        //  otherwise, we are OK,          < command for UNIXes

         "cd ", App. appSpace,          // change directory to the appspace
         " && ", cmdQuery,              // and finally execute command
         App. windows?                  // if Windows,
            "\"":                       //  close quotation mark,          < command for Windows
            ""                         //  otherwise, we are OK,          < command for UNIXes
      )
   );

   BufferedReader reader = new BufferedReader
   (
      new InputStreamReader
      (
         p. getInputStream ()
      )
   );

   String line;
   while
   (
      (line = reader. readLine ()) != null
```

```
    )
    {
        linesOfOutput++;
        if (linesOfOutput > 1) // skip the first line of output ("Active code page: 1251")
        {
            line = line. trim ();

            if (makeLogs) // last valid line of output to string
            {
                outLine = line;
            }
            else // output to logs (visible)
            {
                App. step (line);
            }
            if (getErrors (line, errorPattern))
            {
                hasErrors = true;
            }
        }
    }

    if (p. waitFor () == 0) // End code
    {
        hasErrors = false;
        running = false;
    }

    while (running)
    {
        if (reader. readLine () == null)
        {
            running = false;
        }
    }
}
```

```
    catch (Exception e)
    {
        hasErrors = true;
        running = false;
    }
}

/**
 * Return log line when command finishes
 * @return      logs' string
 */
static String logs() // get output of command to string
{
    while (true)
    {
        if (!running)
        {
            return outLine. trim ();
        }
    }
}

/**
 * Scans string for errors (pattern matches)
 * @param line              string that may contain error code
 * @param cmdErrorPattern   error code format
 * @return                  if error was found
 */
private static boolean getErrors
(
    String line,            // line to search for error in
    String cmdErrorPattern  // error pattern to search for
)
{
    Pattern pattern = Pattern. compile (cmdErrorPattern); // error code pattern
    Matcher matcher = pattern. matcher (line);
```

```
        return (matcher. find ()); //return matcher.group(1);
    }
}
```

Zip.java

```
package com.mastertimmi;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;

/**
 * Zipping engine
 */
class Zip
{
    List <String> fileList;

    Zip ()
    {
        fileList = new ArrayList <>();
    }

    /**
     * Package files into archive
     * @return      if files were zipped successfully
     */
    boolean zipIt ()
    {
```

```
String output = App. makeStr
(
   App. outSpace,
   App. appName,
   "-",
   App. version,
   ".zip"
);

byte [] buffer = new byte [1024];
new File (output). getParentFile (). mkdirs ();

try
{
   ZipOutputStream zos = new ZipOutputStream (new FileOutputStream (output));

   App. step
   (
      App. makeStr
      (
         "Zipping to ",
         output
      )
   );

   for (String file: this. fileList)
   {
      App. step
      (
         App. makeStr
         (
            "File Added : ",
            file
         )
      );
```

```
            ZipEntry ze = new ZipEntry (file);
            zos. putNextEntry (ze);

            FileInputStream in = new FileInputStream
            (
               App. makeStr
               (
                  App. workSpace,
                  File. separator,
                  file
               )
            );

            int len;
            while
            (
               (len = in. read (buffer)) > 0
            )
            {
               zos. write (buffer, 0, len);
            }

            in. close ();
         }
         zos. closeEntry ();

         zos. close (); //remember close it

         return true;
      }
      catch (IOException ex)
      {
         return false; //System.out.println("!!! error"); //ex.printStackTrace();
      }
   }
```

```
/**
 * Traverse a directory and get all files, and add the file into fileList
 * @param node      folder to search in
 * @param excludes  excluded folders filter
 */
void generateFileList
(
   File node,
   String [] excludes
)
{
   if (!stringContainsItemFromList (node. toString (), excludes))
   {
      if (node. isFile ()) // add files only
      {
         String file = node. getAbsoluteFile (). toString ();
         fileList. add (file. substring (App. workSpace. length () + 1, file. length ()));
      }

      if (node. isDirectory ())
      {
         String [] subNode = node. list ();
         if (subNode != null)
         {
            for (String filename: subNode)
            {
               generateFileList (new File (node, filename), excludes);
            }
         }
      }
   }
}

/**
 * Check if file is from one of folders we need in ZIP archive
 * @param inputStr  string to search in
```

```
     * @param items     elements to look for in a string
     * @return          if elements were found in a string
     */
    private boolean stringContainsItemFromList
    (
        String inputStr,
        String [] items
    )
    {
        for (String item: items)
        {
            if (inputStr. contains (item))
            {
                return true;
            }
        }
        return false;
    }
}
```

App.java

```
package com.mastertimmi;

import java.io.File;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * Main engine
 */
class App
{
        static boolean windows = true;          // by default, we expect Maven in
Windows
```

```
        static String appSpace = "";            // application folder inside workspace
        static String workSpace = "";           // workspace with application folder,
dependencies etc.
        static String outSpace = "";            // directory for compiled package
        static String appName = "";             // application artifact id
        static String version = "";             // application version

    private static String groupName = "";           // application group id
    private static String cleanBuild = "";          // if workspace should be sanitised prior
to building
    private static String barDep = "";              // bar build dependency (if empty, we are
building library)

    private static String localRepository = "";       // maven remote repository ID
    private static String remoteRepositoryId = "";    // maven remote repository ID
    private static String remoteRepositoryUrl = "";   // maven remote repository URL

    /**
     * Logically calling "array" element containing a command string
     * @param request   the command ID
     * @return          the command query
     */
    private static String query
    (
        String request,
        String variant
    )
    {
        switch (request)
        {
            case "helpEvaluate":
                return makeStr
                (
                    "mvn -o help:evaluate -Dexpression=",
                    variant,
                    ( // display some - parse
```

```
                windows?
                    " | findstr /V \\[":   // windows
                    " | grep -v \\["      // UNIX
                )
            );

        case "unpackDependencies":
            return makeStr
            (
                "mvn dependency:unpack-dependencies -DmarkersDirectory=\"",
                workSpace,
                File. separator,
                ".deps-",
                appName,
                "-",
                version,
                "\" -DoutputDirectory=\"",
                workSpace,
                "\" -DincludeScope=compile -Dencoding=UTF-8",
                (
                    windows?        // so that all logs are shown
                        " | findstr /B Downloaded":   // windows
                        " | grep Downloaded"
                )
            );
        case "packageBar":
            return makeStr
            (
                windows? // set profile or continue if it is set already
                    "CALL mqsiprofile.cmd & ": // windows
                    "unsetopt function_argzero && echo $MQSI_WORKPATH &&
mqsiprofile && ", // UNIX (untested!)
                "mqsicreatebar -data \"",
                workSpace,
                "\" -a \"",
                appName,
```

```
            "\" -b \"",
            outSpace,
            appName,
            "-",
            version,
            ".bar\" -p \"",
            barDep,
            "\" -cleanBuild -trace && ", // create bar
            "mqsiapplybaroverride -b \"",
            outSpace,
            appName,
            "-",
            version,
            ".bar\" -k \"",
            appName,
            "\" -m \"javaIsolation=true\""  // enable Java isolation;
        );

    case "install":
    case "deploy":
        return makeStr
        (
            "mvn ",
            request,
            ":",
            request,
            "-file -Dpackaging=",
            variant,
            " -Dfile=\"",
            outSpace,
            appName,
            "-",
            version,
            ".",
            variant,
            "\" -DrepositoryId=\"",
```

```
                remoteRepositoryId,
                "\" -Durl=\"",
                remoteRepositoryUrl,
                (
                    variant. equals ("bar")?
                        (makeStr ("\" -DgeneratePom=false -DgroupId=\"", groupName, "\" -
DartifactId=\"", appName, "\" -Dversion=\"", version, "\"")):    // bar
                        (makeStr ("\" -DpomFile=\"", appSpace, File. separator,
"pom.xml\""))                                                    // zip
                ),
                (
                    windows?        // so that all logs are shown
                        " | findstr /B Uploaded":   // windows
                        " | grep Uploaded"
                )
            );

        default:
            return "";
    }
}

/**
 * Get properties of project, Maven and system, clean and prepare environment
 * @param goalId    0 - clean, 1 - create, 2 - install, 3 - deploy; scenario
 * @return          if the properties were extracted successfully, and we can go on
 */
static boolean prepare
(
    int goalId    // 0 - clean, 1 - create, 2 - install, 3 - deploy; scenario
)
{
    windows = System. getProperty ("os.name"). contains ("Windows");

    state
    (
```

```
            makeStr
            (
               "Detected OS family: ",
               windows?
                  "Windows":
                  "UNIX"
            ),
            true
         );

         if (appSpace. equals ("")) // first we get appSpace
         {
            appSpace = System. getProperty ("user.dir"); // the initial step
         }

         File f = new File
         (
            makeStr
            (
               appSpace,
               File. separator,
               "pom.xml"
            )
         );

         if
         (
            f. exists ()
            && !f. isDirectory ()
         )
         { // Getting all the important environment properties:
            workSpace = appSpace. substring (0, appSpace. lastIndexOf (File. separator));
// workspace is actually a parent directory for appSpace

            appName = appSpace. substring (appSpace. lastIndexOf (File. separator) + 1,
appSpace. length ());
```

```
Command cleanBuildCmd = new Command // Cleaning workspace
(
   query
   (
      "helpEvaluate",
      "cleanBuild"
   ),
   "\\b(ERROR)\\b",
   true
);
cleanBuild = cleanBuildCmd. logs ();

if (goalId >= 1) // create
{
   outSpace = makeStr // directory for compiled package
   (
      workSpace,
      File. separator,
      "target",
      File. separator
   );

   Command groupNameCmd = new Command // get group name of artifact
   (
      query
      (
         "helpEvaluate",
         "project.groupId"
      ),
      "\\b(ERROR)\\b",
      true
   );
   groupName = groupNameCmd. logs ();

   Command versionCmd = new Command
```

```
    (
        query
        (
            "helpEvaluate",
            "project.version"
        ),
        "\\b(ERROR)\\b",
        true
    );
    version = versionCmd. logs ();

    Command barDepCmd = new Command // Getting BAR build dependency
    (
        query
        (
            "helpEvaluate",
            "barDep"
        ),
        "\\b(ERROR)\\b",
        true
    );
    barDep = barDepCmd. logs ();

    if
    (
        barDep. equals ("")
        || barDep. contains (" ")
    )
    { // we are working with ZIP file
        barDep = "";
    }
}

if (goalId >= 2) // install
{
    Command localRepCmd = new Command // Getting local repository
```

```
    (
        query
        (
            "helpEvaluate",
            "settings.localRepository"
        ),
        "\\b(ERROR)\\b",
        true
    );
    localRepository = localRepCmd. logs ();
}

if (goalId == 3) // deploy
{ // deploy - we only need folder and remote repository
    Command remoteRepIdCmd = new Command // Getting remote repository
    (
        query
        (
            "helpEvaluate",
            "project.distributionManagement.repository.id"
        ),
        "\\b(ERROR)\\b",
        true
    );
    remoteRepositoryId = remoteRepIdCmd. logs ();

    Command remoteRepUrlCmd = new Command // Getting remote repository
    (
        query
        (
            "helpEvaluate",
            "project.distributionManagement.repository.url"
        ),
        "\\b(ERROR)\\b",
        true
    );
```

```
    remoteRepositoryUrl = remoteRepUrlCmd. logs ();
}

stage
(
    cleanBuild. equals ("true")?
        "Cleaning workspace": // then we are building clean
        "Dirty build"        // then we are building dirty
);
sanitise ();

stage
(
    barDep. equals ("")?
        "Building ZIP lib":
        "Building BAR app"
);

step
(
    makeStr
    (
        "workspace      : ",
        workSpace
    )
);

if (goalId >= 1)
{
    if (!groupName. equals (""))
    {
        step
        (
            makeStr
            (
                "groupId        : ",
```

```
            groupName
        )
    );
}

if (!appName. equals (""))
{
    step
    (
        makeStr
        (
            "artifactId     : ",
            appName
        )
    );
}

if (!version. equals (""))
{
    step
    (
        makeStr
        (
            "version        : ",
            version
        )
    );
}

if (!barDep. equals (""))
{
    step
    (
        makeStr
        (
            "bar dependency  : ",
```

```
            barDep
        )
    );
}

if (goalId == 2)
{
    step
    (
        makeStr
        (
            "local repo     : ",
            localRepository
        )
    );
}

if (goalId == 3)
{
    step
    (
        makeStr
        (
            "remote repo ID  : ",
            remoteRepositoryId
        )
    );

    step
    (
        makeStr
        (
            "remote repo URL : ",
            remoteRepositoryUrl
        )
    );
```

```
            }
        }

        return state
        (
            "Arguments set",
            true
        );
    }
    else
    {
        return state
        (
            "POM file was not found",
            false
        );
    }
}

/**
 * Removes directory and all of its contents
 * @param folder    folder to remove
 */
private static void deleteFolder
(
    File folder
)
{
    File [] files = folder. listFiles ();
    if (files != null) // some JVMs return null for empty dirs
    {
        for (File f: files)
        {
            if (f. isDirectory ())
            {
                deleteFolder (f);
```

```
            }
            else
            {
               f. delete ();
            }
         }
      }
      folder. delete ();
   }

   /**
    * Fast and memory efficient way of concatenating strings
    * @param strParts  strings
    * @return          resulting string
    */
   static String makeStr
   (
      String... strParts
   )
   {
      StringBuilder builder = new StringBuilder ();
      for (String part: strParts)
      {
         builder. append (part);
      }
      return String. valueOf (builder);
   }

   /* ----------Logging---------- */

   /**
    * Print simple log message
    * @param message   message itself
    */
   static void step // Simple output for usual messages // --- Logging --- // "trace",
"debug", "info", "warn", "error" or "off"
```

```
(
    String message // message to print
)
{
    if (message. trim (). length () > 1)
    { // simple message
        System. out. println (message);
    }
}

/**
 * Print message to let user know what's happening
 * @param message   info for user
 */
private static void stage // Simple output for starting messages for different stages
(
    String message // stage title to print
)
{
    System. out. println
    (
        makeStr
        (
            "[",
            time (),
            "] [INFO] > ",
            message,
            "..."
        )
    );
}

/**
 * Print current state for something
 * @param message   what is inspected
 * @param success   success: is it true
```

```
     * @return          also return success
     */
    private static boolean state // Simple output for finishing messages for different
stages
    (
        String message, // result to print
        boolean success // if result is successful
    )
    {
        step
        (
            success?
                (
                    makeStr
                    (
                        "[",
                        time (),
                        "] [INFO] ",
                        message,
                        " - SUCCESS"
                    )
                ):
                (
                    makeStr
                    (
                        "[",
                        time (),
                        "] [ERROR] ",
                        message,
                        " - FAILURE"
                    )
                )
        );
        return success;
    }
```

```
    /**
     * Time string for logger
     * @return time string
     */
    private static String time () // returns time - formatted string
    {
        Date now = new Date (); // java.util.Date, NOT java.sql.Date or java.sql.Timestamp!
        return new SimpleDateFormat ("yyyy/MM/dd/HH:mm:ss"). format (now);
    }

    /* ----------------Goals---------------- */

    /**
     * Clean workspace
     * @return if workspace was cleaned successfully
     */
    private static boolean sanitise ()
    {
        if (!cleanBuild. equals ("true"))
        {
            return true;
        }

        File [] list = (new File (workSpace)). listFiles ();
        if (list != null)
        {
            for (File item: list)
            {
                if (!item. getAbsolutePath (). equals (appSpace)) // do not delete foldder with
sources
                { // delete everything else
                    if (item. isDirectory ())
                    {
                        deleteFolder (item);
                    }
                    else
```

```
            {
                item. delete ();
            }
          }
        }
        return true;
    }
    else
    {
        return true;
    }
}

/**
 * Create BAR application or ZIP library - package
 * @param prepare   if preparation step is needed
 * @return          if package creation was successful
 */
static boolean create
(
    boolean prepare
)
{
    boolean success = false;

    if
    (
        prepare
        && !prepare (1)
    )
    {
        return false;
    }

    if (sanitise ()) // cleans workspace if needed
    {
```

```
if (barDep. equals ("")) // Zip
{
   stage ("Compressing");

   Zip archive = new Zip ();

   String [] excludes =
   {
      ".git",
      ".metadata",
      ".deps-",
      makeStr
      (
         appName,
         "-"
      )
   };

   archive. generateFileList
   (
      new File (appSpace),
      excludes
   );

   boolean filesGenerating = true;
   while (filesGenerating)
   {
      if (!archive. fileList. isEmpty ())
      {
         filesGenerating = false;
         success = state
         (
            "Compression finished",
            (
               archive. zipIt ()
            )
```

```
            );
        }
    }
}
else // Bar
{
    stage ("Checking dependencies");

    deleteFolder
    (
        new File
        (
            makeStr
            (
                workSpace,
                File. separator,
                ".deps-",
                appName,
                "-",
                version
            )
        )
    );

    new File
    (
        makeStr
        (
            workSpace,
            File. separator,
            ".deps-",
            appName,
            "-",
            version
        )
    ). mkdirs (); // create folder for dep markers
```

```
        Command checkDepCmd = new Command // unpack dependencies to current
workspace
        (
            query
            (
                "unpackDependencies",
                ""
            ),
            "\\b(ERROR)\\b",
            false
        );
        boolean checkingDepsNow = true;

        while (checkingDepsNow)
        {
            if (!checkDepCmd. running)
            {
                checkingDepsNow = false;
                success = state
                (
                    "Dependencies are satisfied",
                    !checkDepCmd. hasErrors
                );
            }
        }

        if (success) // if dependencies were satisfied, build BAR
        {
            stage ("Compiling");
            Command compileCmd = new Command // setting session profile and
compiling
            (
                query
                (
                    "packageBar",
```

```
                    ""
                ),                  // command to execute
                "\\b(BIP[0-9][0-9][0-9][0-9]E)\\b",    // error pattern to detect problems
(BIPxxxI - success, BIPxxxW - warning)
                false
            );
            boolean barBuilding = true;
            while (barBuilding)
            {
                if (!compileCmd. running)
                {
                    barBuilding = false;
                    success = state
                    (
                        "Build finished",
                        !compileCmd. hasErrors
                    );
                }
            }
        }
    }
    return success;
}

/**
 * Save package to local or remote repository - install or deploy
 * @param prepare   if preparation step is needed
 * @param local     if installing to local repository
 * @return          if package was saved successfully
 */
static boolean save
(
    boolean prepare,
    boolean local
)
```

```
{
    if
    (
        prepare
        && !prepare
        (
            local?
                2:
                3
        )
    )
    {
        return false;
    }

    if
    (
        local? // if we are installing to local repository,
            (
                create (false)
            ): // we only need to create package, prior to installation // install (local)
            (
                save (false, true)
            ) // otherwise (when - to remote), we first save it to local // deploy  (remote)
    )
    {
        String format = barDep. equals ("")?
            "zip":
            "bar";

        Command deployCmd = new Command // Deploying ZIP to repository
        (
            query
            (
                (
                    local?
```

```
                    "install":  // install to local
                    "deploy"   // deploy to remote
                ),
                format
            ),
            "\\b(ERROR)\\b", // error pattern
            false // log does not go to variable
        );

        return state
        (
            makeStr
            (
                format. toUpperCase (),
                " deployment to ",
                local?
                    "local":
                    "remote",
                " repository"
            ),
            !deployCmd. hasErrors
        );
    }
    return false;
  }
}
```

Standalone.java

```
package com.mastertimmi;

import java.io.File;
import java.util.Scanner;
import static java.lang.Integer.parseInt;

/**
```

```
 * @author masterTimMi
 */
public class Standalone
{
   public static void main (String [] args)
   {
      App. step ("{This is a \"standalone\" version of maven plugin.}");
      App. appSpace = System. getProperty ("user.dir");

      boolean showHelp = false;

      Scanner reader = new Scanner (System. in);  // Reading from System.in

      if (args. length > 0)
      {
         if
         (
            isDir (args [0])
         )
         {
            App. appSpace = args [0];
         }
         else
         {
            App. step
            (
               App. makeStr
               (
                  "Directory \"",
                  args [0],
                  "\" does not exist."
               )
            );
            showHelp = true;
         }
      }
```

```
else
{
   App. step ("Where should we run? [current directory]");
   String tryDir = ""; // Scans the next token of the input as a string

   if (reader. hasNextLine ())
   {
      tryDir = String. valueOf (reader. nextLine ()). trim ();

      if (tryDir. length () == 0)
      {
         App. step ("[current directory]");
      }
      else
      {
         if (isDir (tryDir))
         {
            App. appSpace = tryDir;
         }
         else
         {
            App. step
            (
               App. makeStr
               (
                  "Directory \"",
                  tryDir,
                  "\" does not exist."
               )
            );
            showHelp = true;
            reader. close (); //once finished
         }
      }
   }
}
```

```
        if (showHelp)
        {
            System. out. println ("--- Automation maven plugin ---
\nArguments:\n(directory) - Optional directory argument. Otherwise run in a
current one.");
        }
        else
        {
            App. step
            (
                App. makeStr
                (
                    "Running at ",
                    App. appSpace,
                    " ..."
                )
            );

            boolean optSetInProgress = true;

            int choice = 1;
            String goal = "package";

            while (optSetInProgress)
            {
                App. step ("What should we do?");
                App. step ("0 - clean workspace");
                App. step ("1 - create package");
                App. step ("2 - install package");
                App. step ("3 - deploy package");

                App. step ("Enter a number: [1]");

                String line;
                if (!(line = reader. nextLine ()). isEmpty ())
```

```
{
    try
    {
        choice = parseInt(line);
        if
        (
            choice >= 0
            && choice <= 3
        )
        {
            optSetInProgress = false;
            switch (choice)
            {
                case 0:
                    goal = "clean";
                    break;
                case 1:
                    goal = "package";
                    break;
                case 2:
                    goal = "install";
                    break;
                case 3:
                    goal = "deploy";
                    break;
            }
        }
        else
        {
            App. step ("Invalid option");
        }
    }
    catch (NumberFormatException e)
    {
        App. step ("Invalid input");
    }
```

```
            }
            else
            {
                optSetInProgress = false;
            }
        }
        reader. close (); //once finished

        App. step
        (
            App. makeStr
            (
                "Goal: ",
                goal
            )
        );

        new Command
        (
            App. makeStr
            (
                "mvn ru.rsb.esb.mvn:automation-maven-plugin:",
                goal
            ),
            "ERROR",
            false
        );
    }
}

/**
 * Checks whether the dir exists
 * @param dirName   name of directory
 * @return          directory exists
 */
private static boolean isDir (String dirName)
```

```
    {
        File f = new File (dirName);
        return
        (
            f. exists ()
            && f. isDirectory ()
        );
    }
}
```