TRAINING & CONSULTING

Java tooling

ABIS Training & Consulting
www.abis.be
training@abis.be

Document number: 1769_01.fm
10 September 2020

Address comments concerning the contents of this publication to:
ABIS Training & Consulting, Diestsevest 32 / 4b, B-3000 Leuven, Belgium
Tel.: (+32)-16-245610

TABLE OF CONTENTS

PREFACE

This course discusses a number of interesting tools, used in the Java environment, to control the building, testing and deployment of Java based applications.

- An introduction to the eXtended Markup Language (**XML**) is used as a starting point, to define and configure controlling structures for the application.
- Composing and building the application can be done via **Maven**, a software project management and comprehension tool. It is based on the concept of a project object model (POM) written in XML.
- A test driven development approach is enabled in Java via the **JUnit** framework. This test feature can be integrated in Maven, as an additional step.
- Next, source control and versioning support is provided via **Git**, a free distributed version control system, which can be accessed via native commands, or from within an IDE like Eclipse or IntelliJ. All application components, including the configuration and build resources, can be stored in the Git repository.
- Finally, the whole project can be continuously integrated, developed and deployed, using the features of **Jenkins**.

The different building blocks are introduced seperately, but will be integrated as soon as possible to demonstrate the power of CI/CD chain.

Further reference and details can be found at the appropriate tool sites on the internet.

XML

Objectives :

- **introduction to XML**
- **XML documents**
- **XML schema**
- **namespaces**

see 1006_09a.pdf

Java and XML

Objectives :

- **introduction**
- **handling XML in Java**
- **binding Java to XML**
- **example**

Introduction

1

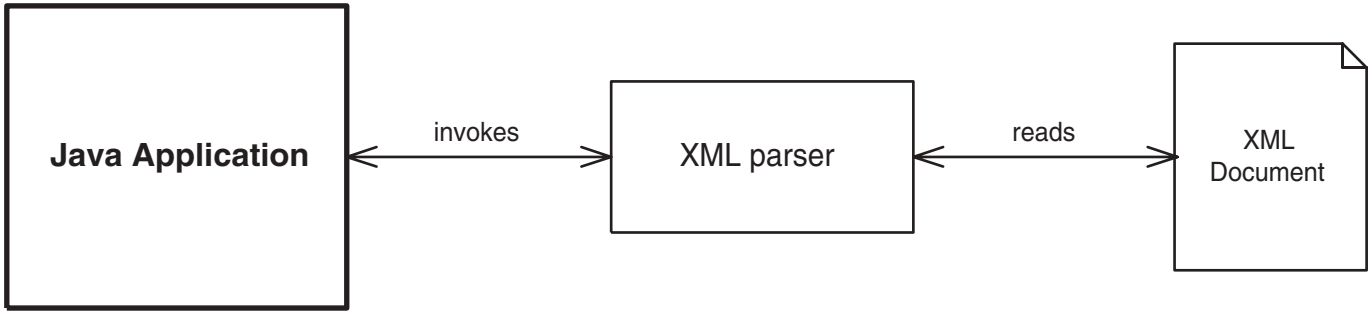
Importance of XML

- data definition (well formed)
- data validation (valid)
- data interchange
- hierarchical
- namespaces

Processing of XML

- data transformation
- document driven programming
- data storage
- data binding

Handling XML in Java



- Java API for XML Processing
- support for different parser types
  - Simple API for XML Parsing (SAX)
  - Document Object Model (DOM)
  - Streaming API for XML (StAX)
- supports Extensible Stylesheet Language Transformation (XSLT)
- namespace support
- XSL processor

JAXP - SAX example

```
public class TestXMLSax {
    public static void main(String[] args) {
        ArrayList<Person> persons = null;
        try {
            InputSource input =
                new InputSource(new FileReader("personList.xml"));
            SAXParserFactory spf = SAXParserFactory.newInstance();
            SAXParser sp = spf.newSAXParser();
            XMLReader mxr =sp.getXMLReader();
            mxr.setContentHandler(new MyContentHandler());
            mxr.parse(input);
            persons = MyContentHandler.getPersons();
            System.out.println(persons);
        } catch (SAXException | IOException | ParserConfigurationException e) {
            e.printStackTrace();
        }
    }
}
public class Person {
    private String firstName,lastName;
    //getters and setters
    public String toString(){
        return firstName+ " " + lastName;
    }
}
```

JAXP - SAX example (cont.)

```
public class MyContentHandler implements ContentHandler {
    private String tempVal;
    private Person p;
    private static ArrayList<Person> persons = new ArrayList<Person>();
    public static ArrayList<Person> getPersons() { return persons;}
    public void startDocument() throws SAXException {
        System.out.println("started parsing");
    }
    public void startElement(String uri, String localName, String name, Attributes atts)
                                throws SAXException {
        if (name.equals("Person")) { p = new Person();}
    }
    public void characters(char[] ch, int start, int length) throws SAXException {
        tempVal = new String(ch, start, length);
    }
    public void endElement(String uri, String localName, String name)
                    throws SAXException {
        if (name.equals("Person")){ persons.add(p);}
        else if(name.equals("FirstName")){p.setFirstName(tempVal);}
        else if(name.equals("LastName")){p.setLastName(tempVal);}
    }
    public void endDocument() throws SAXException {
        //System.out.println("finished parsing document");
    }
    // and the other methods...
}
```

Binding Java to/from XML

- Java Architecture for XML binding
- conversion of XML to Java types
- marshal/unmarshal XML content into/from Java representation
- access, update and validate Java representation against schema constraint
- javax.xml.bind package
- use annotations to map Objects to XML elements

Note: JAXB not include in JavaSE 11. Add library seperately

JAXB - example

4

```
public class TestJAXB {
    public static void main(String[] args) {
        try {
            File file = new File("personList.xml");
            JAXBContext jaxbContext = JAXBContext.newInstance(Persons.class);
            Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
            Persons persons = (Persons) jaxbUnmarshaller.unmarshal(file);
            for (Person p : persons.getPersons()) {
                System.out.println(p);
            }
        } catch (JAXBException e) {
            e.printStackTrace();
        }
    }
}
```

JAXB - example (cont.)

```
@XmlRootElement(name="persons")
@XmlAccessorType (XmlAccessType.FIELD)
public class Persons {
    @XmlElement(name="person")
    private ArrayList<Person> persons= new ArrayList<>();
    public ArrayList<Person> getPersons() {return persons;}
    public void setPersons(ArrayList<Person> persons) {this.persons = persons;}
}
@XmlAccessorType (XmlAccessType.FIELD)
public class Person {
    private String firstName;
    @XmlElement(name="lastname")
    private String lastName;
    public String getFirstName() {return firstName;}
    public void setFirstName(String firstName) {this.firstName = firstName;}
    public String getLastName() {return lastName;}
    public void setLastName(String lastName) {this.lastName = lastName;}
    public String toString(){
        return firstName+ " " + lastName;
    }
}
```

Maven

Objectives :

- core concepts
- building a Java project
- properties and resource filtering
- building enterprise projects
- support in IDE tools

Core concepts

Maven: build and dependency management tool for Java based application development

maven.apache.org

Software project management framework/tool

- builds
- configuration management
- versioning
- project reports
- documentation
- ...

Core concepts (cont.)

- Project Object Model POM
  - information about the project
  - configuration details (goals, dependencies, plugins, ...)
  - used to build (artifacts for) the project
- artifact
  - group id, version id, type
  - jar, war, ear, pom, ...
  - stored in repository
- repository
  - local vs remote
  - standardised directory structure and naming
- plugin
  - executed by Maven goals during build
- lifecycle

POM

1.1

Project Object Model

- information about the project and configuration details used by Maven to build the project
- project dependencies -> cornerstone of Maven !
- the plugins or goals that can be executed
- the build profiles
- (POM can inherit from another POM)
- ...

Defined in pom.xml

When executing goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, then executes the goal.

cf.: goal ~ ANT task

POM example

```
<?xml version="1.0" encoding="utf-8"?>
<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>be.abis.mvn</groupId>
    <artifactId>my-app</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>My first Maven POM</name>
```

POM elements

- project: top-level element
- modelVersion: version of the maven object model
- groupId: unique identifier of the organization or group that created the project.

  based on the fully qualified domain name of your organization
- artifactId: unique base name of the primary artifact (typically JAR) being generated by this project

  secondary artifacts, like source bundles, also use the artifactId as part of their final name

  <artifactId>-<version>.<extension> (`myapp-1.0.jar`)
- version: of the generated artifact
  - SNAPSHOT indicates project in development

    <version>1.0-SNAPSHOT</version>
  - nothing indicates (stable) released project

    <version>1.0</version>
- name: display name used for the project.

POM example (cont.)

- properties: placeholders accessible anywhere within a POM

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.release>11</maven.compiler.release>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>
```

- dependencies: to other projects

  Maven downloads (JAR) to local repository, and links the (transitive) dependencies on compilation, as well as on other goals that require them

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Dependency scope

determines when/where the dependency is available

- in the current build
- as a transitive dependency

possible values

- compile
- runtime
- test
- provided
- system
- import

More information about the dependencies

- **use the maven-dependency-plugin**

  mvn dependency:resolve

  shows a list of all artifacts that have been resolved

  mvn dependency:tree

  shows a tree of (transitive) dependencies

- **via site generated report**

  mvn site

POM example (cont.)

- **build: declare the project's directory structure and manage plugins**

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>3.1.0</version>
      </plugin>
    ........
    </plugins>
  </pluginManagement>
</build>

</project>
```

Artifact

1.2

Every artifact has

- group id
- artifact id
- version (1.0-SNAPSHOT, 2.1.0, ...)
- artifact type (pom, jar, war, ear, ...)
- (optionally) artifact classifier

Artifacts are stored in a repository

Repository

1.3

- standard directory layout

  <group id>/<artifact id>/<version>

  be/abis/my-app/1.0-SNAPSHOT

- standard naming conventions for artifacts

  <artifact-id>-<version>

  my-app-1.0-SNAPSHOT.pom

Local repository

- created automatically in `<user_home>/.m2/repository`
- all dependencies are loaded from this repository first

Remote repository

- used by Maven to download additional artifacts
- all downloads are copied to the local repo
- default central repo: `https://repo.maven.apache.org/maven2`
- (optionally) configure additional repos

Plugin

1.4

Maven is plugin execution framework, i.e. executes goals via plugins

- plugins have (collection of) goals
- goals are identified by `<plugin id>:<goal id>`
  clean:clean
  compiler:compile
- build your own plugins

(https://maven.apache.org/guides/mini/guide-configuring-plugins.html)

Common plugins (`https://maven.apache.org/plugins/`)

- maven-compiler-plugin: compiles Java sources
- maven-deploy-plugin: deploy an artifact into a remote repo
- maven-resources-plugin: copy resources to the output directory
- maven-surefire-plugin: runs JUnit (or TestNG) test
- ...

Archetype

Maven project templating toolkit

http://maven.apache.org/archetype/maven-archetype-plugin/usage.html

plugin, used to generate standard projects, based on model

mvn archetype:generate

configuration via additional (interactive) parameters

- archetypeArtifactId: maven-archetype-quickstart (1652)
- groupId: be.abis.mvn (based on organisation URL)
- artifactId: my-app (~main directory)
- archetypeVersion: 1.4
- (Java) package for sources: be.abis.mvn

Generate Maven project structure via archetype

```
mvn archetype:generate
    -DgroupId=be.abis.mvn
    -DartifactId=my-app
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DarchetypeVersion=1.4
    -DinteractiveMode=false
```

resulting directory structure (+ HelloWorld app !)

```
C:\....\my-app
pom.xml
+---src
   +---main
   ¦  +---java
   ¦     +---be
   ¦        +---abis
   ¦           +---mvn
   +---test
      +---java
         +---be
            +---abis
               +---mvn
```

Lifecycle

build lifecycle consists of (ordered sequence of) phases

- validate, compile, test, package, integrationtest, verify, install, deploy, ...

- in every build phase, plugin goals are executed
- goals are bound to the lifecycle phases

Example: for pom packaging...

install install:install-file

Maven will execute every phase in the sequence up to and including the one defined

Activation

- add a plugin
- configure in the POM

Example of lifecycle phase

mvn compile

Compile phase: the phases and goals that actually get executed are

- validate
- generate-sources
- process-sources
- generate-resources
- process-resources
- compile

Maven lifecycle phases

- validate: is the project correct? all necessary information available?
- compile: the source code
- test: with unit testing framework e.g. JUnit
- package: into distributable format, e.g. JAR
- integration-test: process and deploy the package into environment where integration tests can be run
- 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 local projects)
- deploy: to the remote repository
  (for sharing with other developers and projects)

Other

- clean: cleans up artifacts created by prior builds (target dir)
- site: generates site (HTML) documentation for this project

Building a Java project

2

Recommended directory structure

2.1

- sources: `/src/main/java`
- tests: `/src/test/java`
- target: `/target/classes`
- resources: `/src/main/resources`
- test resources: `/src/test/resources`
- test target: `/target/test-classes`

Note: use archetype for preparation

mvn archetype:generate

select maven-quickstart-archetype

typical Maven phases/goals

- compile
- test (includes compile): JUnit based test files
- test-compile: only compiles to test target
- package: generate .jar in `/target` dir
- install: copy .jar to local repository

Configuring plugins

2.3

Example: Java compiler plugin in POM

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8</version>
            <configuration>
                <source>1.9</source>
                <target>1.9</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

The configuration element applies the given parameters to the plugin

Configuring plugins (cont.)

optional: configure the goals to be executed in a phase

```
<plugin>
...
    <executions>
        <execution>
            <phase>process-resources</phase>
            <configuration>
                <tasks>....</tasks>
            </configuration>
            <goals>
                <goal>run</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Testing

Dependency, based on JUnit, in POM

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Run tests

mvn test

test files, based on (surefire plugin) JUnit:

- **/*Test.java
- **/Test*.java
- **/*TestCase.java

Properties

placeholders accessible anywhere within a POM

- every tag inside the <properties> tag becomes a property

Example

```
<properties>
    <compiler.release>11</compiler.release>
</properties>
```

- to specify Java version 11 for compilation

reference to the property inside the POM

- ${<property name>}

```
<version>${compiler.release}</version>
```

Resource filtering

4

allow filtering

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
  </resource>
</resources>
```

properties ${ } will be replaced when handling resources

Building enterprise projects

add required dependencies for

- supporting frameworks (e.g. Hibernate, Spring, Vaadin, ...)
- application servers (e.g. JBoss EAP, Tomcat, Glassfish, Liberty, ...)
- custom libraries (utilities, security, logging, ...)

  use central maven repository: `https://mvnrepository.com`

multi-module builds

- generation of multiple artifacts (JAR, WAR, EAR, ...)
- multi project development

POM inheritance

- super POM
- explicit reference

Multi module build

5.1

- project with packaging 'POM'
- for every module
  - subdirectory
  - <module> entry in the root project's POM
- referring to the subdirectory
  - modules can have dependencies on each other

Maven will determine the build order for the entire set of modules

Example: web project

The root project directory contains

- directories: web and core
- root pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
...
    <name>webapp</name>
    <modules>
        <module>web</module>
        <module>core</module>
    </modules>
</project>
```

Example: web project (cont.)

Subdirectory

- contains its own pom.xml
- suppose: web depends on core

```
<project>
    ...
    <packaging>war</packaging>
    <name>webapp :: web</name>
    ...
    <dependencies>
        <dependency>
            ...
            <artifactId>core</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
</project>
```

POM inheritance

5.2

Common definitions in parent POM

contains <parent/> element

Module POM optionally inherits from parent POM

POM without <parent> element, inherits from super POM

in maven-uber.jar

Maven support in IDE tools

6

Eclipse

6.1

m2eclipse

https://www.eclipse.org/m2e/

IntelliJ Idea

https://plugins.jetbrains.com/plugin/7179-maven-helper

- **for new project**
  - **File -> New -> Project**
  - **check "Maven" on the left**
  - **(optionally) specify archetype**
- **from "normal" Java module**
  - **right click -> add framework support -> Maven**
  - **change the group id in the POM to the base package name**
- **new Maven module**
  - **Project -> New -> Module**
  - **check "Maven" on the left**
  - **next**
  - **add new module to NONE!!! and watch out with naming**

Maven support for Java in IntelliJ

Specify Java version

```
<build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <source>1.11</source>
          <target>1.11</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

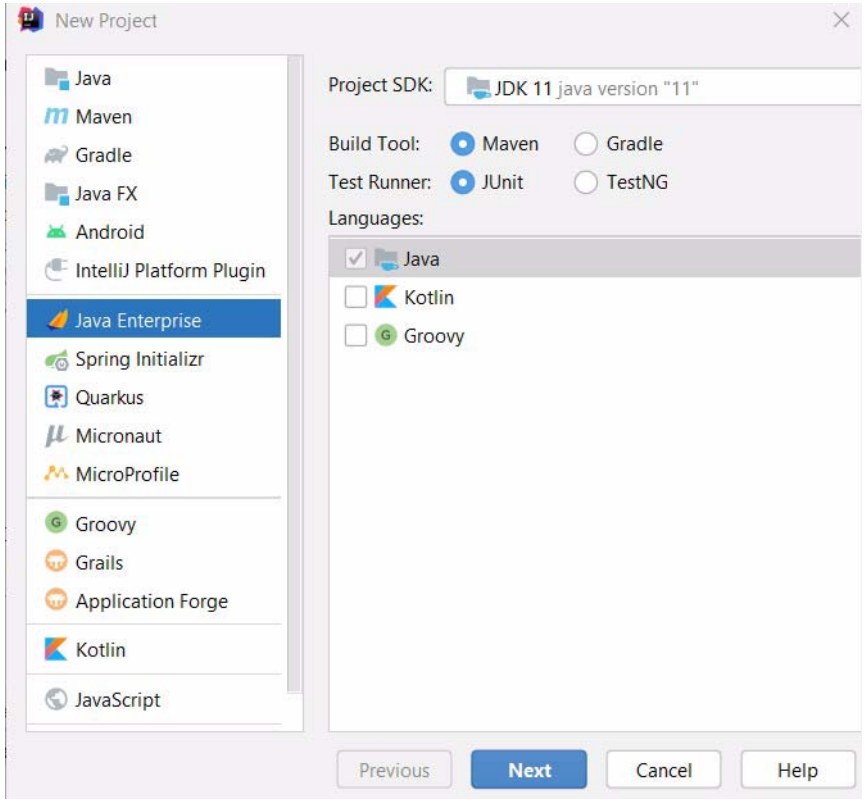
in case of <packaging>war</packaging>

```
  <properties>
   <maven.compiler.source>1.11</maven.compiler.source>
   <maven.compiler.target>1.11</maven.compiler.target>
  </properties>
```

Maven for Java EE project

for new Java EE project

- File -> New -> Project
- select Java Enterprise project on the left
- check "Maven"
- (optionally) specify additional libraries or frameworks



JUnit

Objectives :

- **testing principles and Unit testing**
- **JUnit (V4/5) framework**

Testing principles

1

types of tests:

- code testing
  - unit test
  - integration test
- regression testing
- performance testing
- defect tracking
- ...

Testing based on test plans/scenarios, derived from use cases

What is JUnit?

web site: www.junit.org

open source testing framework used to develop and execute unit tests in Java

What are unit tests?

- low-level
- investigate the behaviour of a single component (=unit) within a class, servlet, EJB,...
- based on component specification
- written before the component is developed (test-driven development)

Why use unit testing?

- improvement in productivity and overall code quality

Why use a unit testing framework?

Advantages

- easier to write tests
- easier to run tests
- easier to rerun tests after change

  +

  consistency, maintenance, ramp-up time, automation

JUnit overview

Instantiate object -> invoke method -> verify assertions

JUnit 4 (2006): refactored to take advantage of Java SE 5 features

- **annotations** (no inheritance, no naming conventions)
  - increase of flexibility
  - more lightweight
- **new functionality**
  - parameterized tests
  - simplified exception testing
  - timeout tests
  - flexible fixtures
  - easy way to ignore tests
  - new way to logically group tests

-> packages `org.junit.*`
(and `junit.framework.*` for compatibility with JUnit 3)

JUnit 5

- **released September 2017, requires Java 8 or higher**
- **JUnit 5 is composed of several different modules from three different sub-projects**
- **JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**
  - **JUnit Platform**
    - serves as a foundation for launching testing frameworks on the JVM
    - defines the TestEngine API for developing a testing framework that runs on the platform
    - provides a Console Launcher to launch the platform from the command line and build plugins for Gradle and Maven as well as a JUnit 4 based Runner for running any TestEngine on the platform
  - **JUnit Jupiter**
    - combination of the new programming model and extension model for writing tests and extensions
    - provides a TestEngine for running Jupiter based tests
  - **JUnit Vintage**
    - provides a TestEngine for running JUnit 3 and JUnit 4 based tests

JUnit 5 (..)

- new in JUnit 5
  - lambda support
  - test interfaces with default methods
  - nested unit tests
  - conditional test execution
  - parameterized tests
  - possibility to write custom extensions
  - repeated tests
  - dynamic tests
- watch out: some names of annotations have changed!
- available with Eclipse Oxygen.1a (4.7.1a)

JUnit 4 implementation

3.1

junit.jar in classpath

Most important classes of the `org.junit.*` framework

- **Assert** with several static test methods
- Write test methods in test classes with annotations

Execution of the test cases

- with the command line
  - `org.junit.runner.`**JUnitCore**
- or via IDE tooling (Eclipse, IntelliJ, Netbeans,...)

JUnit Assert

3.2

- The Assert class contains only static methods to be invoked in the test methods (use static import)
- General principle:
  - assert the condition
  - if the test fails ---> report the failure

Example

```
import static org.junit.Assert.*;
Person p = new Person("John","Travolta");
String firstName = p.getFirstName();
assertEquals(firstName,"Johan");
```

Two types of reports:

- Failures: failures of anticipated test conditions

  org.junit.ComparisonFailure: expected:<Joh[a]n> but was:<Joh[]n>
- Errors: unexpected error or exceptions

JUnit Assert (..)

Different tests

Type of test	Normal	Negated
condition returns true	assertTrue(boolean)	assertFalse(boolean)
Object does not exist	assertNull(Object)	assertNotNull(Object)
Both objects refer to the same instance	assertSame(Object, Object)	assertNotSame(Object, Object)
Both objects are equal	assertEquals(Object, Object)	-
Fail unconditionally	fail()	-

Notes:

- **assertEquals is overloaded to compare**

  Objects, booleans, longs, doubles,....

- **additional String parameter for message**

  assertEquals("First name incorrect", firstName,"Johan");

  output:

  org.junit.ComparisonFailure: First name incorrect expected:<Joh[a]n> but was:<Joh[]n>

JUnit test class

3.3

Create 1 test class for each class to be tested with

- annotation `@Test` before test method `testXxx()`
- initialize and finalize the fixture (test context)
  - defined via annotations `@Before` and `@After`

    `-> setUp()` / `tearDown()` methods
  - called before and after each test

  to make sure there are no side effects between test runs

Example

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import be.abis.demo.Person;

public class PersonTest {
  Person p;
  @Before
  public void setUp() throws Exception {
    p = new Person("John","Travolta");
  }
  @After
  public void tearDown() throws Exception {
   p = null;
  }
  @Test
  public void testGetFirstName() {
    String firstName = p.getFirstName();
    assertEquals("First name incorrect", "Johan", firstName);
  }
```

JUnit Run it!

Command line

- `java org.junit.runner.JUnitCore` contains main methods
- test classes as arguments

  --> include junit.jar in classpath

Programmatically

- use method
  `org.junit.runner.JUnitCore.runClasses(TestCls.class);`

Results

- success
- failure: a method from the `Assert` class failed (`AssertionFailedException` thrown)
- error: unexpected exception was raised by the test method

Example

with `org.junit.runner.JUnitCore` at command line

```
java -cp .;%JUNIT_HOME%/junit.jar org.junit.runner.JUnitCore PersonTest
JUnit version 4.8.2
.E
Time: 0,016
There was 1 failure:
1) testGetFirstName(PersonTest)
org.junit.ComparisonFailure: First name incorrect expected:<Joh[a]n> but was:<Joh[]n>
        at org.junit.Assert.assertEquals(Assert.java:123)
        at PersonTest.testGetFirstName(PersonTest.java:15)

FAILURES!!!
Tests run: 1,  Failures: 1
```

JUnit extensions

4

Annotations

4.1

The test class need not extend anything (vs TestCase in JUnit 3)

- **@Test** method annotation to define a new test

  instead of naming convention `testXxxx()`

  optional parameters:

  - **expected** to test for an exception to occur
  - **timeout** to fail after a certain timeout
- **@Ignore** to skip a test method or a complete test class
  - a documenting message can be passed

Example

```
import static org.junit.Assert.*;
import org.junit.Ignore;
import org.junit.Test;

public class CourseTest {
  @Test
  public void courseName() {
    Course c1 = new Course("Eclipse");
    assertEquals("Oclipse", c1.getCourseName());
  }
  @Test(expected = IllegalArgumentException.class)
  public void exceptionExpected() {
    Course course = new Course(null);
    course.getCourseName().length();
    fail();
   }
  @Test(timeout = 100)
  public void timed() {...}
  @Ignore("database not yet available")
  public void ignoreThis() {...}
}
```

Fixtures

4.2

more flexible than in JUnit 3.8

- @Before, @After annotation on a method
  - instead of `setUp()` and `tearDown()`
  - called before and after each test in a test case
- @BeforeClass, @AfterClass annotation on a method
  - “one-time fixtures”
  - called before the first and after the last test of a test case

JUnit

1. Testing principles
2. What is JUnit?
3. JUnit overview
4. JUnit extensions

Git

Objectives :

- **introduction**
- **locations**
- **commands**
- **handling versions**
- **integration with IDE**

ref. Book ProGit - Scott Chacon, Ben Straub - ISBN-13: 978-1484200773 Apress; 2nd ed. (November 9, 2014)

Introduction 1

What is a version control system (VCS)? 1.1

- keeps records of your changes
- allows for collaborative development
- allows you to know who made what changes and when
- allows you to revert any changes and go back to a previous state

Git - distributed version control system

created by Linus Torvalds in 2005

free and open source

(note SubVersioN is centralised VCS)

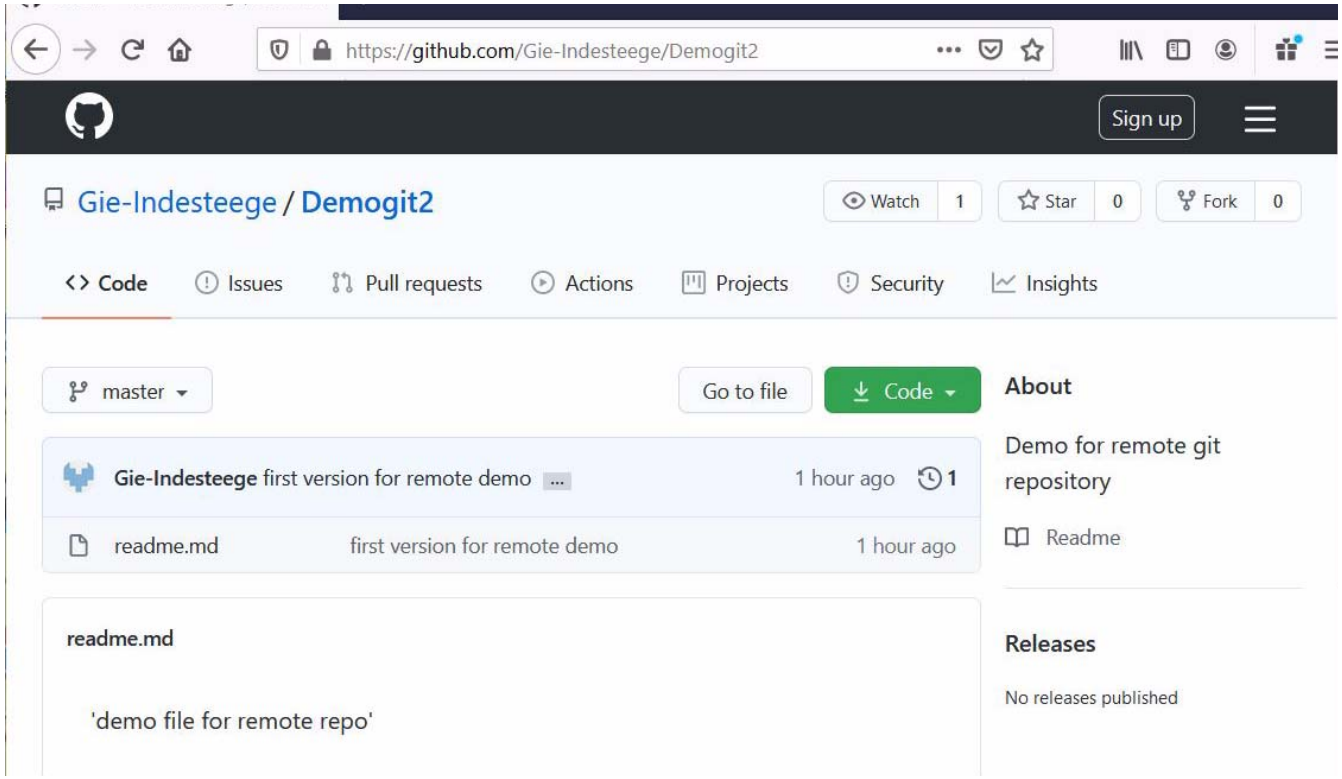
- reversibility
- concurrency
- annotation

every developer has a full local copy of the (central) **repository**

Central repository

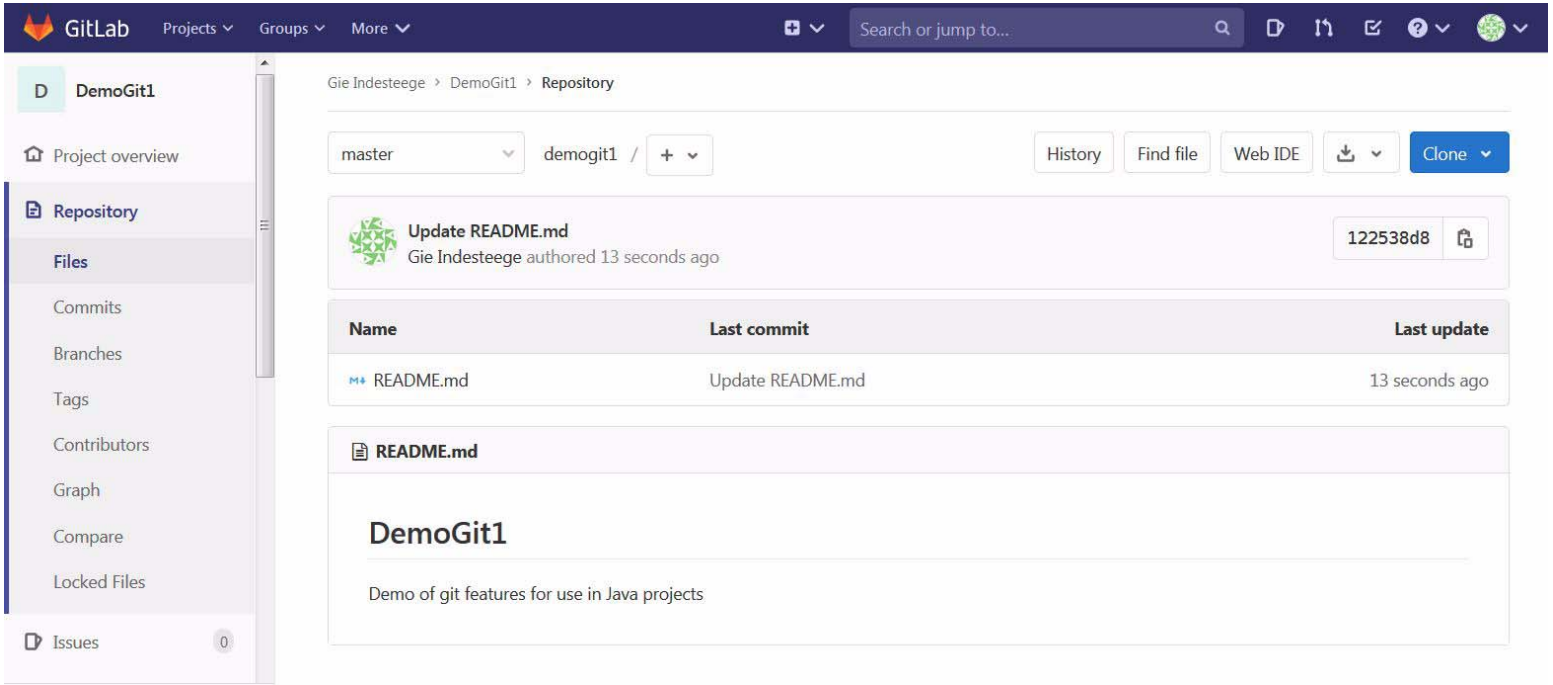
1.2

- **GitHub (MicroSoft)**



Central repository (cont.)

- **GitLab (GitLab)**



- **Bitbucket (Atlassian)**

Integration with CI/CD tools for continuous integration/deployment

Definitions

1.3

Repository (repo)

- contains (source) files
- directory .git with metadata and history
- local vs. remote

Branch

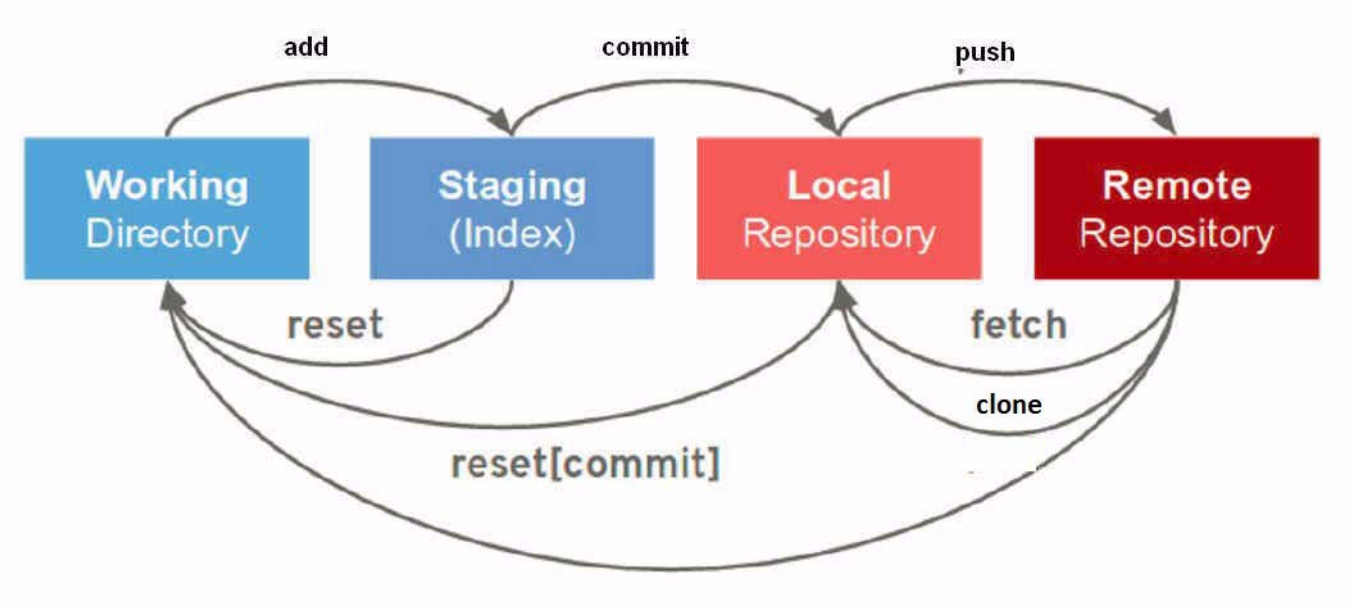
- group of commits that ‘live’ together
- master/main
- last commit in current branch, referenced/pointed to by HEAD

Snapshot

- to keep track of code history
- records what all your files look like at a given point in time
- user decides when to take a snapshot, and of what files
- created by **commit**

Overview of Git locations

Locations to store code and version information



- working directory: (source) files on local machine
- index/staging: files are ready to be ‘promoted’/added to snapshot
- local repository: projects under local control
- remote repository: projects under remote control

Commands to move information between locations

2.1

- clone: copy entire remote -> local (create **fork**)
- fetch: synchronise local (branches) with remote (branches)
- (modify resources)
- add: prepare/stage/index files for snapshot
- commit: create snapshot
- push: copy commits to remote
- merge: combine commits from branches
- checkout: switch to branch
- pull: merge fork into master

Commands

3

Local git repository creation

3.1

- **git init**

  creates .git folder for repository.
  This folder is the repository metadata, an embedded database

- **git config --global user.email "myname@abis.be"**

  set your account's default identity

or clone -> create fork/copy of remote repository

Remote (central) repository -> local

3.2

- **git clone https://github.com/username/repo.git**
- **git clone git@gitlab.com:username/repo.git**
  - **entire repository (all branches) copied to local machine (source/history)**

Commit

- **git add myfile**

  stage: add (untracked) files/changes to the index (temp storage)

  add . -> stages all unstaged changes

- **git commit**
  - **take (full) snapshot of the repository, that is saved in the database/current branch**
  - **information stored**
    - unique ID for each commit using SHA1 hash function of contents
    - about how the files changed from previously (delta info)
    - reference to the commit that came before it:

      "parent commit"

  make commit messages useful (reason to motivate the changes)

  git commit -m "my commit message"

  modification of commit information

  git commit --amend -m "other commit message" [--reset-author]

Branches

3.4

- git branch **feat1**
  - create new branch (**feat1**) off of **current** branch
- git checkout **feat1**
  - switches to **feat1** branch

the contents of the working directory will be those that belong to the snapshot the HEAD is pointing to -> master or branch

- git checkout -b feat1

  switches to and creates the branch feat1
- git branch

  check existing branches

Connecting local to remote repo

3.5

- git push origin **feat1**
  - pushes (sends) branch **feat1** to remote
  - “origin” is the default name given to the remote repo

  git push -u origin master
  - push (updates) from master to remote
- git remote add origin https://github.com/username/repo.git
- git remote add origin git@gitlab.com:username/repo.git
  - existing remote repo defined to local repo

Controlling commands

3.6

status of current git repo/project/branch

- **git status**

```
On branch master
nothing to commit, working tree clean
```

configuration of local repo

- **git config --global --edit**

history info

- **git log --all --graph --decorate --oneline**

Files not to store under git control

file .gitignore

contains names of files to be ‘ignored by git’

Merging

3.7

Switch (checkout) to master branch, and then

- git merge second-branch
  - merge with fast-forward (default)

    forget history -> common ancestors
  - merge without fast-forward

    git merge --no-ff second-branch

    no fast-forward mode should always be used to keep history info

merge with 'selection' of branch

git merge -X <ours|theirs> <branch-name>

Merging conflicts

merging conflicts have to be solved in the files

```
<<<<<<< HEAD
new text
=======
old text
>>>>>>> feat1
```

next: add file and commit!

Handling versions

3.8

tag version to current snapshot

- git tag -a <tag-name vx.y> -m 'message'

differences

- git diff <original>...<modified>

rebase current HEAD to other branch

- git rebase newbranch

  --abort

  --continue

view remote branches

- git branch -r

Undoing

unstage file

- git rm --cached file

  cached -> file removed from index, not from working directory!

restore (previous) version of file

- git restore file

delete commits

- git reset --hard HEAD~n
  - soft reset, the commit(s) will be removed, but the modifications saved in that/those commit(s) will remain
  - mixed (default) uncommit and unstage
  - hard reset, won't leave change made in the commit(s)

  ~n number of commits to be reset

Undoing (cont.)

revert specific commit (creates new commit!)

- git revert HEAD~n --no-edit

delete tag

- git tag -d <tag-name>

delete branch

- git branch -d second-branch

Undoing (cont.)

commands, based on `git push origin <source>:<destination>`

- **delete commit**

  git push origin HEAD~2:master --force

  git push origin +<badcommithash>^ : master

- **delete branch**

  git push origin :feat1

- **delete tag**

  git push origin --tags :v1.0

Synchronising local with remote

3.10

Fetching the remote repository means updating the reference of a local branch, to put it even with the remote branch

- git fetch origin branch

  git fetch --all

Merging

- git merge origin/master

Pull requests, to be confirmed by repo/project owner

- git pull origin rel1

  ~ git fetch + git merge

  - sends pull request for new changes from rel1 to master branch

Usage scenarios

3.11

Using Git without having a clear branching policy is a complete nonsense

Long running branch

key of this strategy is having a branch only for stable versions, where the releases are tagged, for which the default branch is used, master; and having other branches for development, where the features are developed, tested, and integrated.

One version, one branch

creating software that will be available and maintained for several versions

One branch for each bug

Integration of Git in IDE

Git support is provided in most actual IDEs

- Eclipse (EGit plugin)
- VisualStudio (Git extensions)
- Jetbrains IntelliJ IDEA
- SublimeText (GitSavvy)
- JetBrains WebStorm
- ...

IntelliJ IDEA Git support

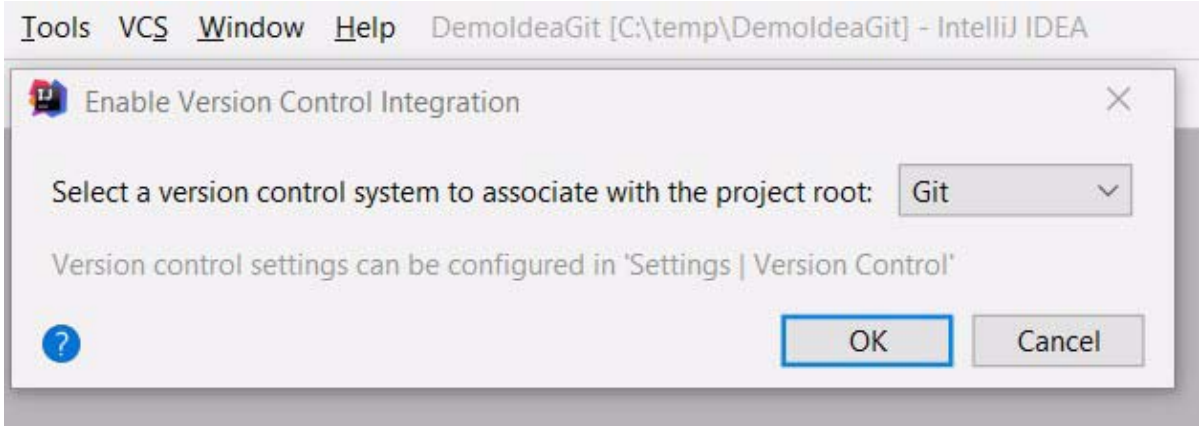
4.1

1. setup github/gitlab account + repository

   github.com or gitlab.com

2. main menu

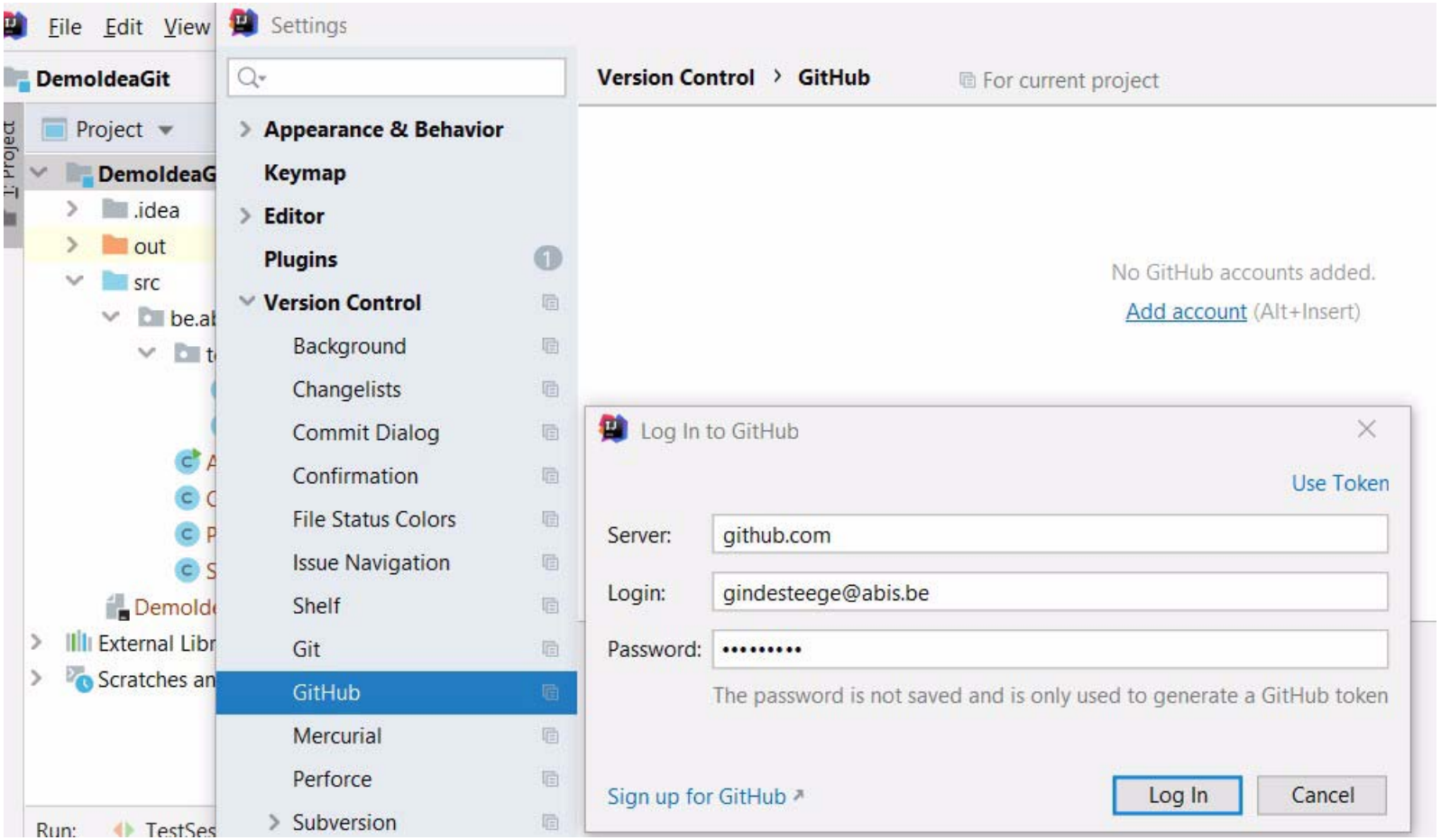
   VCS -> enable



IntelliJ IDEA Git support (cont.)

3. add github/gitlab account

File -> Settings -> Version Control -> Github/Gitlab



IntelliJ IDEA Git support (cont.)

4. checkout from git

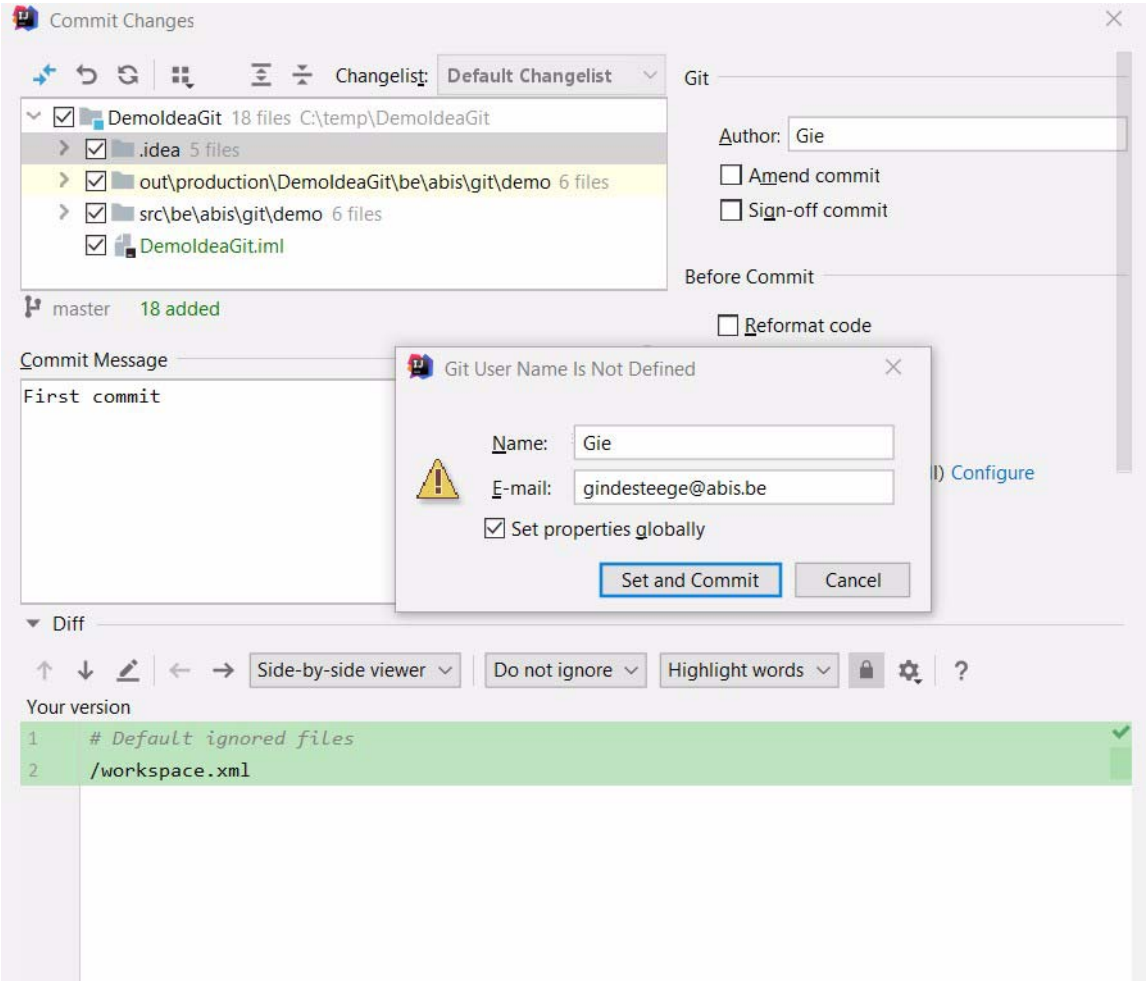
VCS -> get from version control -> git

******* project definition/modification

IntelliJ IDEA Git support (cont.)

5. commit project

VCS -> git -> commit directory



IntelliJ IDEA Git support (cont.)

6. push to remote

VCS -> git -> push

- **first time: define remote repo**
  - name: repo
  - url: https://github.com/username/repo
    or git@gitlab.com:username/repo.git

  + pass credentials

change remote via VCS -> Git -> remotes

ref. https://www.jetbrains.com/help/idea/set-up-a-git-repository.html

Jenkins

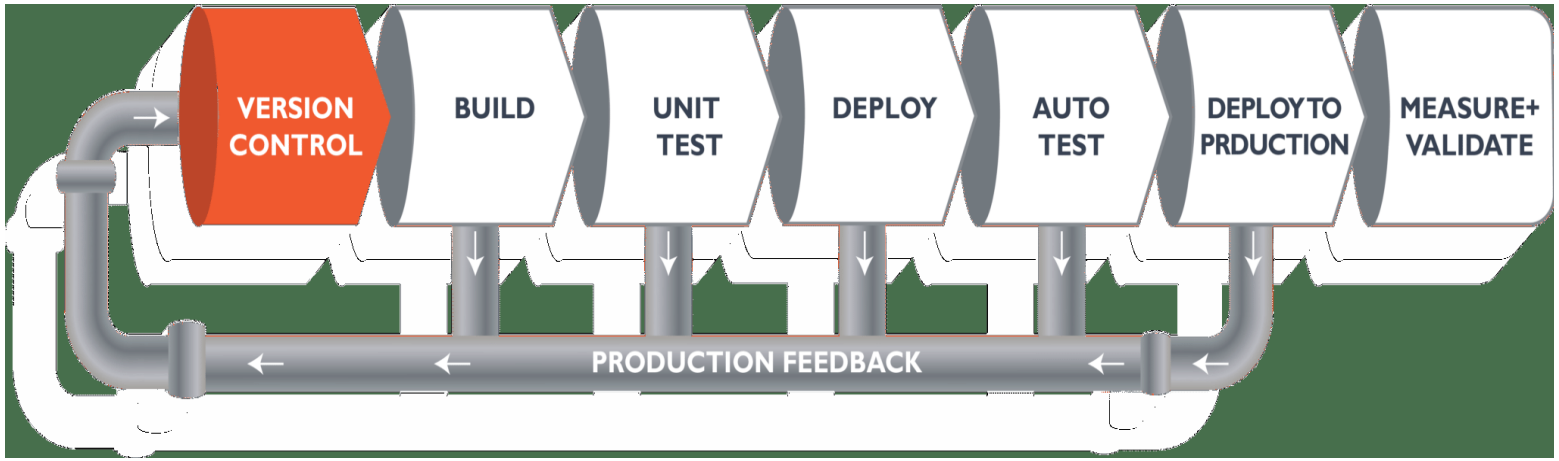
Objectives :

- **DevOps lifecycle**
- **Continuous Integration/ Continuous Delivery / Deployment**
- **Jenkins integration**

DevOps lifecycle

1

Development stages



plan - analyse - build - code - test

Operational stages

deploy - test - operate - monitor

Continuous Integration/ Continuous Delivery / Deployment

2

tools

- development
  - IDE - Eclipse, IntelliJ, Visual Studio, ...
  - build - Maven, Gradle, ...
  - VCS tool - SVN, Git, ...
- operations
  - SCM - Puppet, Chef, Ansible, Bamboo, ...
  - build - Ant, Maven, ...
  - deployment - Urban Code Deploy, AWS Code Deploy, Docker, ...

Build a pipe-line for integration - Jenkins

www.jenkins.io

Continuous Integration/ Continuous Delivery / Deployment philosophy

In Continuous Integration, after a code commit, the software is built and tested immediately. In a large project with many developers, commits are made many times during a day.

With each commit, code is built and tested.
If the test is passed, build is tested for deployment.
If the deployment is a success, the code is pushed to Production.

This commit, build, test, and deploy is a continuous process, and hence the name continuous integration/deployment.

Important:

immediate feedback to developer after (failed) build task

Commit early, commit often, but never commit broken code!

Jenkins integration

3

A Jenkins project is a repeatable build job which contains steps and post-build actions. These actions will be executed via plugins. You can configure build triggers and add security.

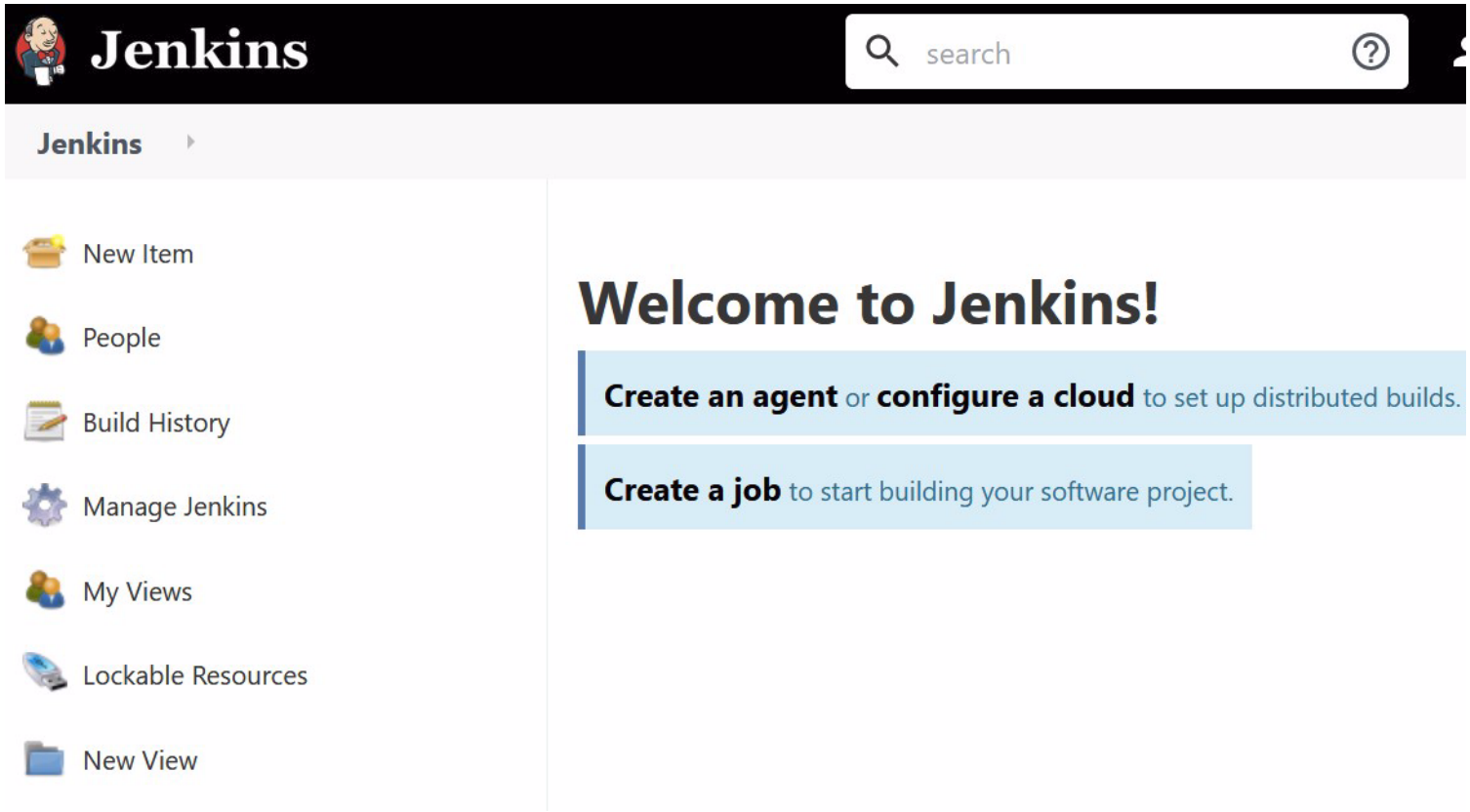
Jenkins provides various interfaces and tools to automate the entire process

- **start from code in Git repository**
- **define entire automation ‘job’ via tasks**
- **phases**
  - **commit**
  - **build - executed via Maven**
  - **test - JUnit + Maven**
  - **stage**
  - **deploy**
  - **monitoring**

Jenkins scenario

Dashboard (`jenkins-server:8080`)

3.1



Manage Jenkins: configure plugins, global tools, create users, ...

Create project

3.2

new item - Freestyle project, Maven project, pipeline, ...

Jenkins search 2

Jenkins › All ›

Enter an item name

DemoABIS

» *Required field*

Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Maven project

Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

OK

Create project (cont.)

specify characteristics (via tab pages)

Jenkins › DemoABIS1 ›

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actio

Description: freestyle demo ABIS

[Plain text] Preview

☐ Discard old builds

☑ GitHub project

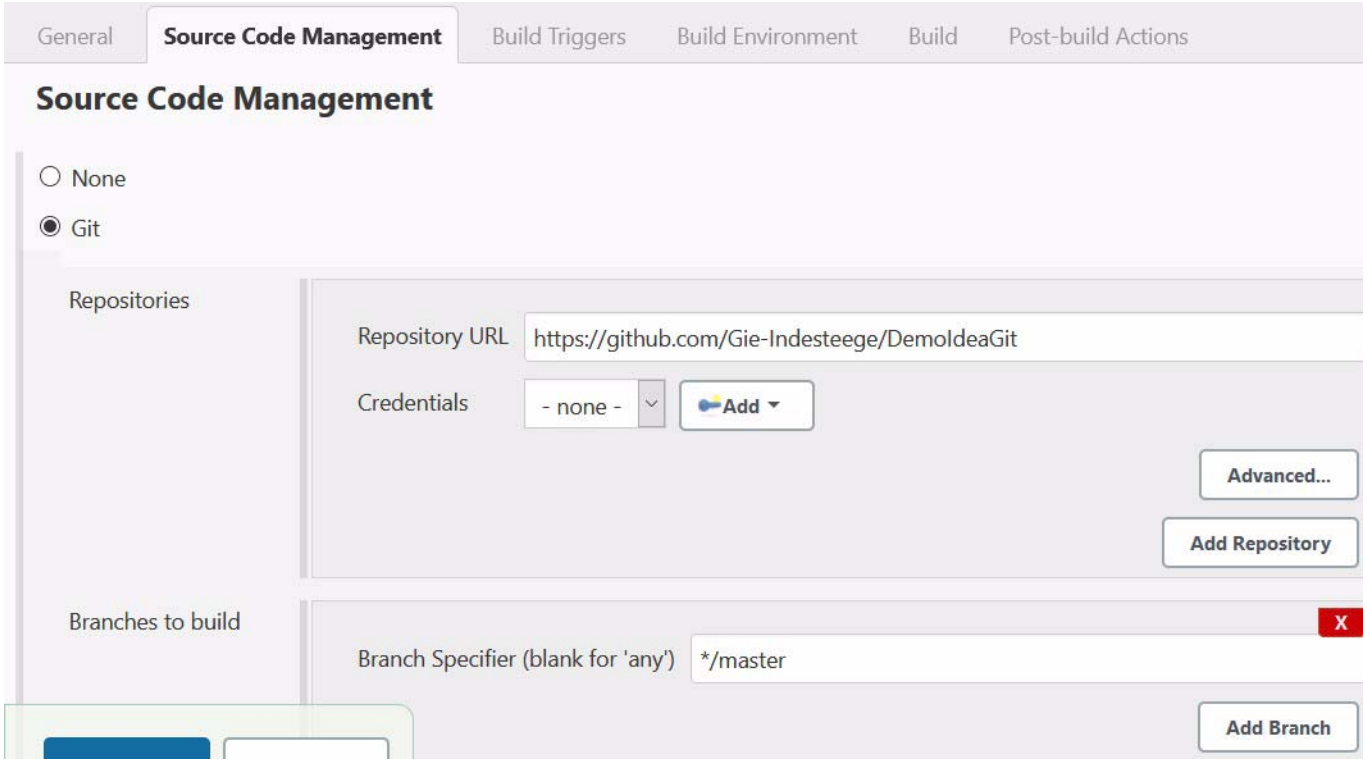
Project url: https://github.com/Gie-Indesteege/DemoIdeaGit

Associate to SCM

3.3

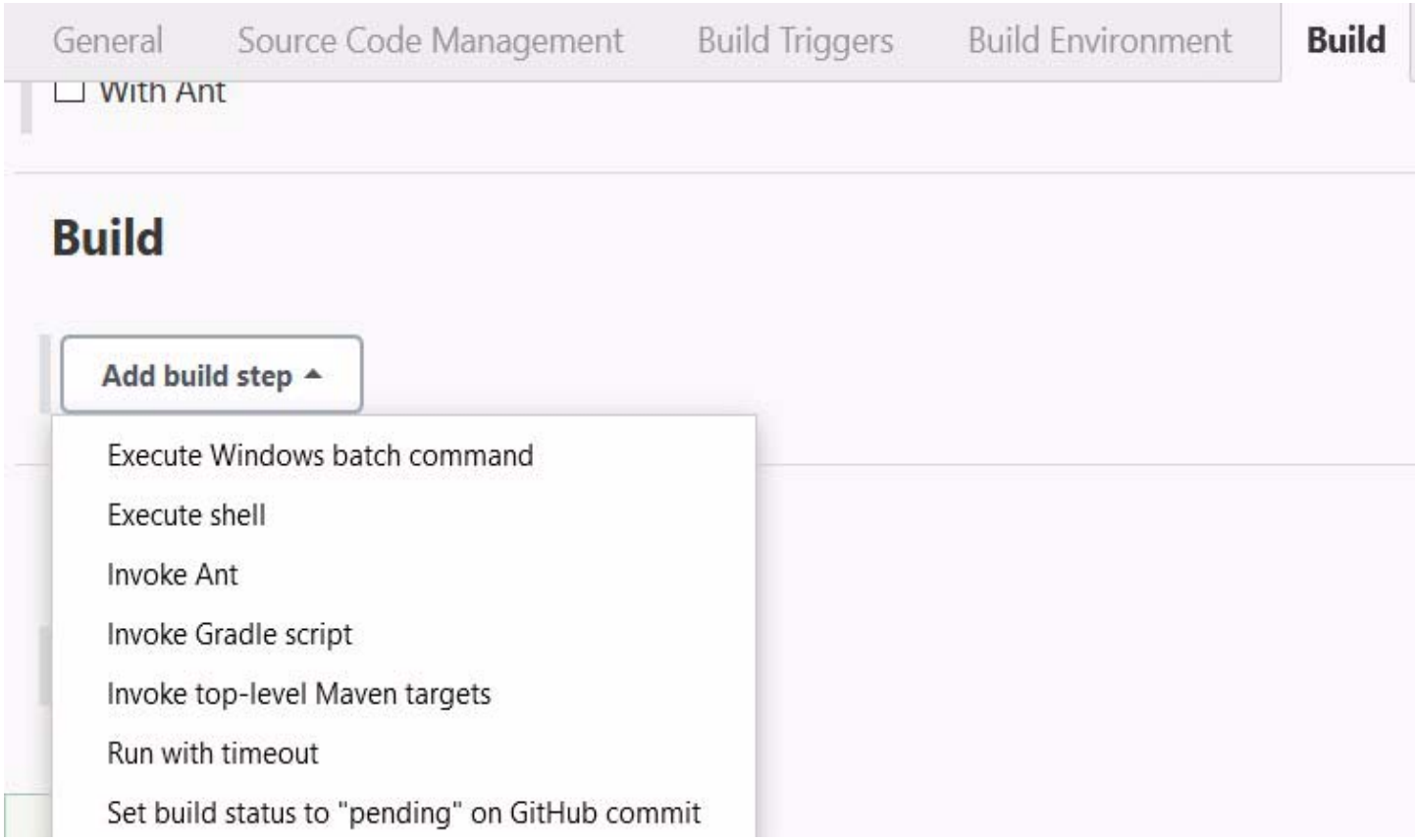
define Git repository

- **remote: https or ssh**
- **local: file**



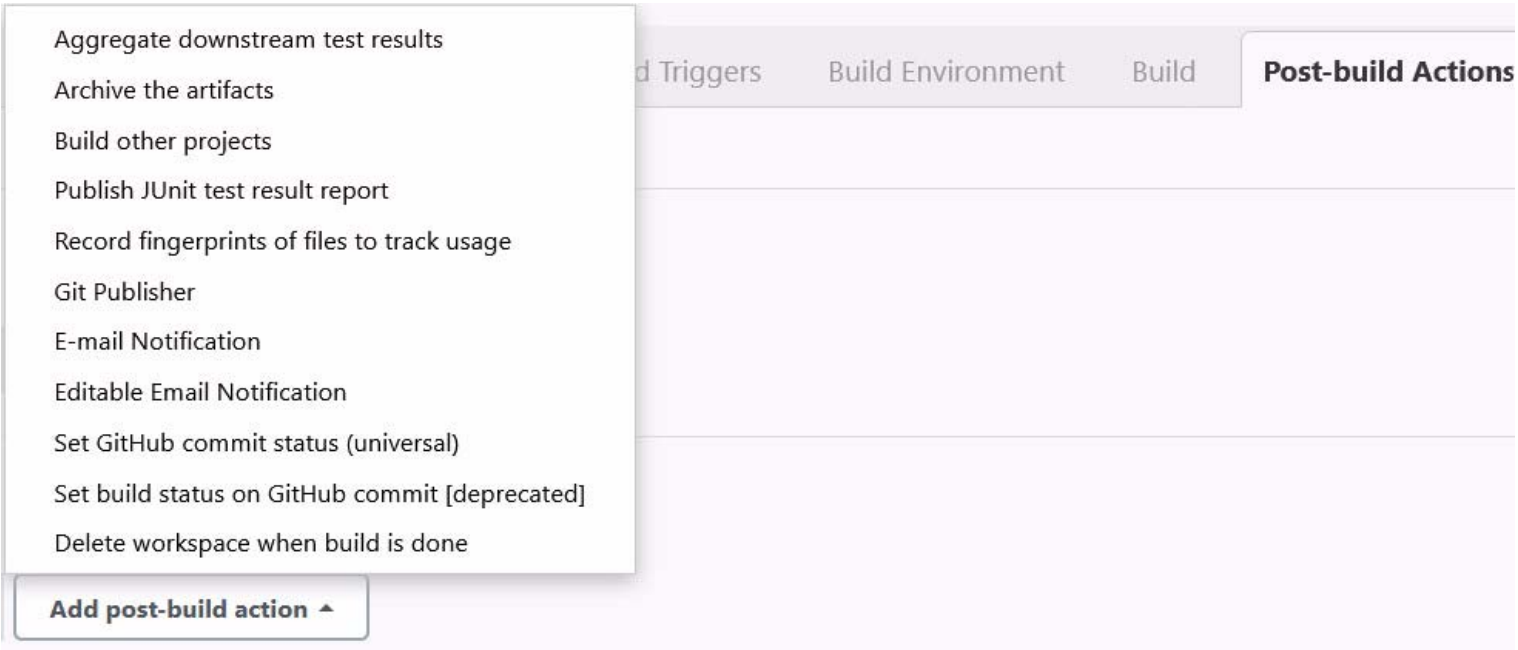
Build information

build environment, build triggers, build tasks



Post build actions

3.5



Dashboard

3.6

check project information, start build, check project status ...

Jenkins › DemoABIS1 ›

- Back to Dashboard
- Status
- Changes
- Workspace
- Build Now
- Delete Project
- Configure
- GitHub
- Rename

Project DemoABIS1

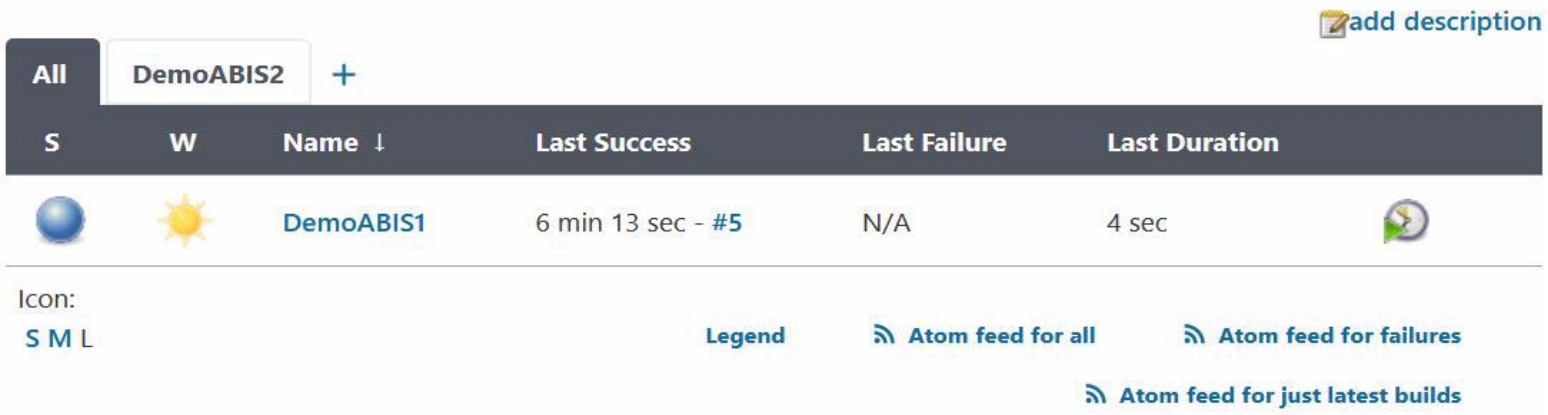
freestyle demo ABIS

- Workspace
- Recent Changes

Permalinks

Build status

3.7

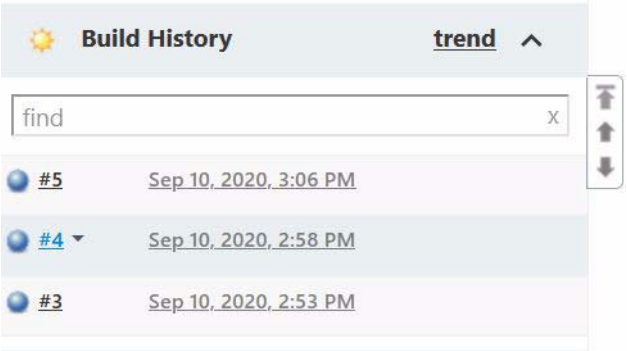


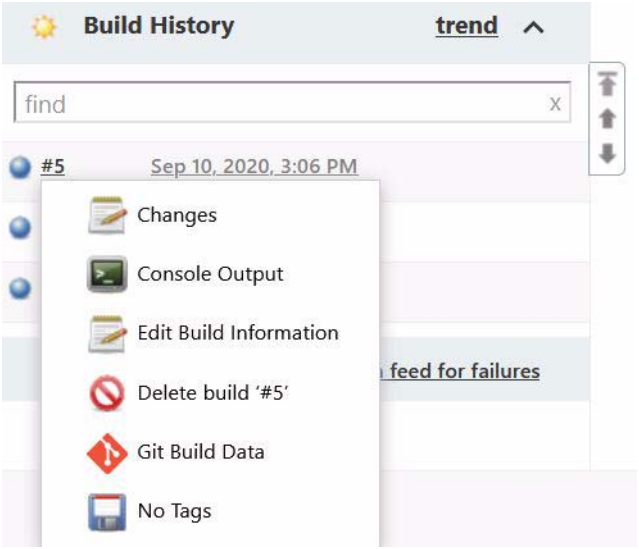
Jenkins

1. DevOps lifecycle
2. Continuous Integration/ Continuous Delivery / Deployment
3. Jenkins integration
4. Jenkins pipeline

Build status (cont.)

Details on build(s) history, and git





Verify timing of steps!

Console output: should end with

Finished: SUCCESS

Jenkins pipeline

4

- a combination of plugins that support the integration and implementation of continuous delivery
- a group of states (build, deploy, test and release). Every state has its events, which work in a sequence

Jenkins pipeline DSL

4.1

pipeline DSL (Domain-specific Language)

- pipeline has an extensible automation server for creating simple or complex delivery pipelines "as code"
- text file -> Jenkinsfile

  stored in git repository

The machine on which Jenkins runs is called a **node**.

A **stage block** contains a series of steps in a pipeline.

A **step** is nothing but a single task that executes a specific process at a defined time. A pipeline involves a series of steps.

Pipeline example

4.2

Add new pipeline

View name: Demo ABIS pipeline

- (●) **Build Pipeline View**

  Shows the jobs in a build pipeline view. The complete pipeline of jobs that a version propagates through are shown as a row in the view.

- ( ) **List View**

  Shows items in a simple list format. You can choose which jobs are to be displayed in which view.

- ( ) **My View**

  This view automatically displays all the jobs that the current user has an access to.

OK

Pipeline example (cont.)

Specify description and details

Pipeline Flow

Layout

Based on upstream/downstream relationship

This layout mode derives the pipeline structure based on the upstream/downstream trigger relationship between jobs. This is the only out-of-the-box supported layout mode, but is open for extension.

Upstream / downstream config

Select Initial Job

DemoABIS1

Example (cont.)

Prepare chain via project configuration -> build triggers

General | Source Code Management | **Build Triggers** | Build Environment | Build

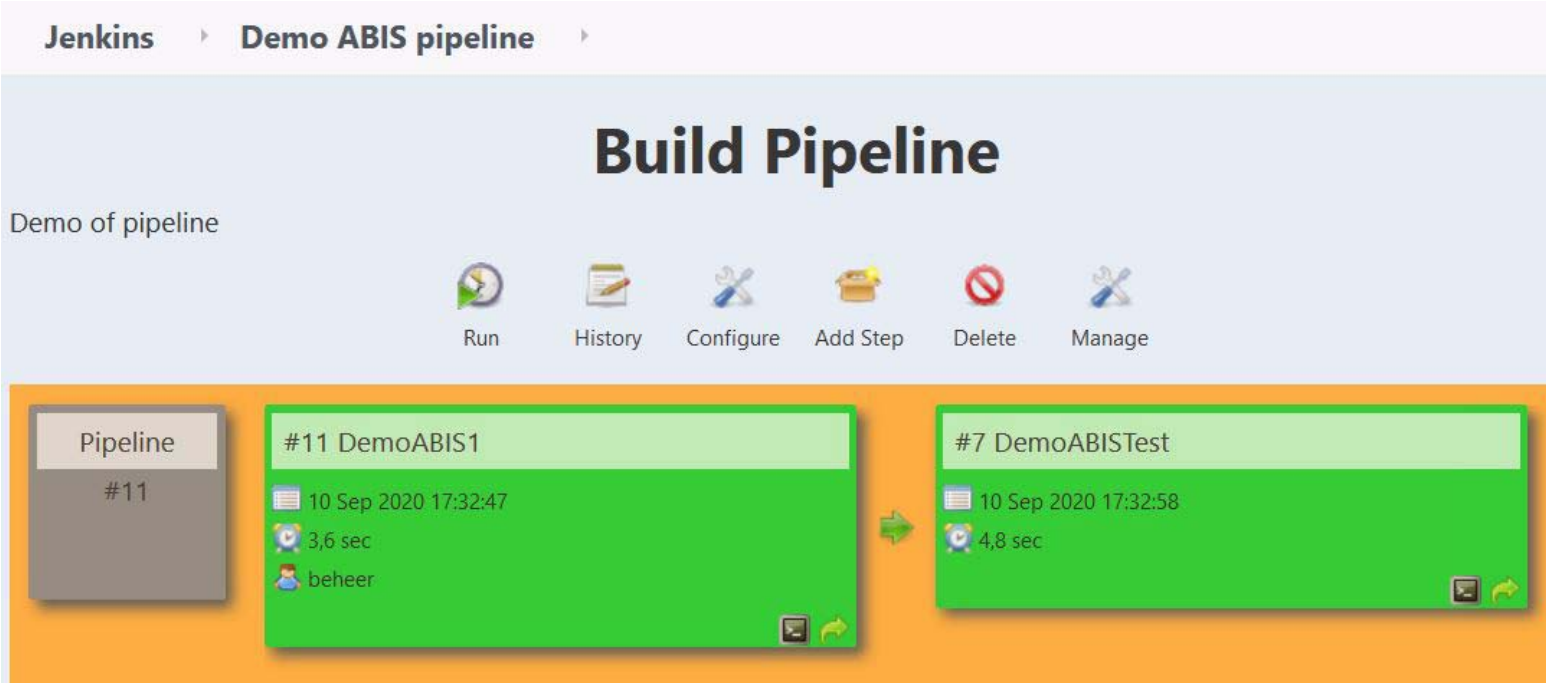
Build Triggers

- [ ] Trigger builds remotely (e.g., from scripts)
- [x] Build after other projects are built

Projects to watch: DemoABIS1

- (●) Trigger only if build is stable
- ( ) Trigger even if the build is unstable
- ( ) Trigger even if the build fails

Example (cont.)



Pipeline with Jenkinsfile - example

in Jenkins multibranch pipeline

```
pipeline {
   agent any
   stages {
      stage('Build') {
         steps {
            echo 'Building..'
         }
      }
      stage('Test') {
         steps {
            echo 'Testing..'
         }
      }
      stage('Deploy') {
         steps {
             echo 'Deploying'
         }
      }
   }
}
```

Pipeline with Jenkinsfile (cont.)

```
post {
  always {
     echo 'This will always run'
  }
  success {
     echo 'This will run only if successful'
  }
  failure {
     echo 'This will run only if failed'
  }
.......
```

APPENDIX A. EXERCISES

Create a Java project for handling Course information.

- an ABIS course is characterised by the following attributes
  - course Id - integer
  - short title - String
  - long title - String
  - course duration - integer
  - course price - double
  - list of sessions - list
- an ABIS session is associated to a course, and is characterised by the following attributes
  - session number - integer
  - session date - localDate
  - session kind - String
  - cancel indicator - boolean
  - course reference - Course

Create an application with a main method to construct a Course with a few sessions related to that course. Print a little report with:

- course Id, long title, session dates and session kinds for that course.

1 Maven

Define a new Java SE project and the corresponding POM.

1. Manually, i.e. create the project structure, directories and POM.XML, using command line commands (MKDIR, CD) and a simple editor; e.g. Notepad++

   Create a simple “Hello world” main app.

   Run the Maven build, and run the main program, using the generated JAR file.
2. Start a new Java project with the Maven archetype **maven-archetype-quickstart**

   Run the Maven build, and run the main program, using the generated JAR file.
3. Working with dependencies

   hint: use the central Maven repository: https://mvnrepository.com
   - add Apache commons logging 1.2

     ```
     <dependency>
       <groupId>commons-logging</groupId>
       <artifactId>commons-logging</artifactId>
       <version>1.2</version>
     </dependency>
     ```
   - check the dependency tree
   - generate the project site with the dependency report

2 JUnit

1. Create JUnit tests for the classes Course and Session.

   Define these test classes in a separate package
2. Integrate the JUnit test cases now into your Maven project definition, in order to run the tests automatically.

3 Git

Setup environment

- Create a GitHub account at www.github.com, and sign in to that account.
  (or, create a Gitlab account at www.gitlab.com)

  define a public remote repository `https://github.com/username/JavaTool` `(or git@gitlab.com:username/JavaTool.git)`

  add a readme.md -> description for the project

  add a firstfile.txt -> welcome message
- Prepare for a local repository on your workstation (using GitBash).

  Create a working directory inside `c:\temp\JavaToolProjects`
- Configure user information used across all local repositories.
  - set a name that is identifiable for credit when reviewing history
    ```
    git config --global user.name "[firstname lastname]"
    ```
  - set an email address that will be associated with history marker, use the same email, you used on github/gitlab
    ```
    git config --global user.email "[valid-email]"
    ```
- Clone remote repository now to your local repository
  ```
  git clone https://github.com/username/JavaTool
  git clone git@gitlab.com:username/JavaTool.git
  ```
- Create a new branch "**feature1**" and switch to that branch (try to do this in 1 command).

Initialise project

- Create a file hello.txt (with some welcome text in it) and adapt the readme.md for the project, and stage both to the branch. Check the git status
- Alter the welcome text in the hello.txt.
- Try to commit both files now. Pay attention to the commit message, and check again the git status.
- Push the result to your remote repository, and check in GitHub/Gitlab whether everything is OK there.

Altering the project

- Add 2 more files: support.txt (containing your name and email address) and company.txt (containing information about your company).
- Stage both files.
- Unstage company.txt again.
- Commit now, and check the git status and git log.
- Change the commit message
- Revert the last commit, in order to delete support.txt from your directory
- Try to undo your revert
- Now, stage company.txt and commit.
- Undo the commit, but company.txt should be kept inside your directory.
- Push to the remote repository, and verify that company.txt is NOT present there.

Additional/optional exercises

- Create a new branch "**feature2**", and make sure it contains all files from branch feature1.
- Delete now the branch feature1 (both in your local and remote repository)
- Switch (again) to the master branch, and add a file hello.txt to it (note that you created a file hello.txt in the branch feature1) with some other welcome text.
- Stage and commit these changes.
- Try now to merge the master branch with branch feature2. What happens? Try to solve the conflicts.
- Create a file secret.txt in your master branch. How can you prevent that the file gets committed to the branch?
- Push the results to your remote repository, and check in GitHub/Gitlab whether everything is OK there, and the file secret.txt is NOT present.
- Create a subdirectory "future" in your local repo, and create a file "niceFeatures.txt" , with some contents, in it. Try to add, commit and push it to the remote repo.

Final exercise

Put the (maven) project of the previous exercise under Git control.

- Set up a sample freestyle project, where you use the application setup of the previous topics.

  Try to build the project.
- create now a pipeline, starting from the previous project.

  Add a test step, and build the project again