Project Automation: Build Systems and Continuous Integration

CSCE 247 - Lecture 10 - 02/20/2019

Project Automation

- Last time, we discussed automating test execution using unit testing frameworks.
  - Tests can be re-executed on command.
  - Much faster than human-in-the-loop testing.
  - Reduced human effort and risk of human error.
- Testing is not all that can be automated.
  - Project compilation, installation, deployment, etc.
- Today:
  - **Project build automation:** Automating the entire compilation, testing, and deployment process.
  - **Continuous integration:** Executing and managing the build process each time code is checked in.

Build Systems

Build Systems

- Building software, running test cases, and packaging and distributing the executable are very common, effort-intensive tasks.
- Building and deploying the project should be as easy as possible.
- Build systems ease this process by automating as much of it as possible.
  - Repetitive tasks can be automated and run at-will.

Build Systems

- Build systems allow control over code compilation, test execution, executable packaging, and deployment to production.
- Script defines actions that can be automatically invoked at any time.
- Many frameworks for build scripting.
  - Most popular for Java include Ant, Maven, Gradle.
  - Gradle is very common for Android projects.

Build Lifecycle

- **Validate** the project is correct and all necessary information is available
- **Compile** the source code of the project.
- **Test** the compiled source code using a suitable unit testing framework.
  - Run **unit tests** against classes and **subsystem integration tests** against groups of classes.
- Take the compiled code and **package** it in its distributable format, such as a JAR.

Build Lifecycle

- **Verify** - run system tests to ensure quality criteria are met.
  - System tests require a packaged executable.
  - This is also when tests of non-functional criteria like performance are executed.
- **Install** the package for use as a dependency in other projects locally.
- **Deploy** the package to the installation environment.

Apache Ant

- Ant (Another Neat Tool) is a build system for Java projects.
- Build scripts define a set of **targets** that can be executed on command.
  - Targets can correspond to lifecycle phases or other desired automated tasks.
  - Targets can trigger other targets.
  - Build scripts written in XML.
    - Platform neutral.
      - But can invoke platform-specific commands.
    - Human and machine readable.
    - Created automatically by many IDEs (Eclipse).

A Basic Build Script

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <target name = "info">
      <echo>Hello World - Welcome to Apache Ant!</echo>
   </target>
</project>
```

- File typically named **build.xml**, and placed in the base directory of the project.
- All build scripts require a **project** element and at least one **target**.
  - Project defines a **name** and a default **target**.
  - This target prints project information.
    - **Echo** prints information to the terminal.

Targets

```
<target name = "deploy" depends = "package"> .... </target>
<target name = "package" depends = "clean,compile"> .... </target>
<target name = "clean" > .... </target>
<target name = "compile" > .... </target>
```

- A target is a collection of tasks you want to run in a single unit.
  - Targets can depend on other targets.
  - If you issue the **deploy** command, it will complete the **package** target first, which will complete **clean** and **compile** first.
  - Dependencies are denoted using the **depends** attribute.

Targets

```
<target name = "deploy" depends = "package"> .... </target>
<target name = "package" depends = "clean,compile"> .... </target>
<target name = "clean" > .... </target>
<target name = "compile" > .... </target>
```

- Target attributes:
  - **name** defines the name of the target (required)
  - **depends** lists dependencies of the target.
  - **description** is used to add a short textual description of the target.
  - **if** and **unless** allow execution of the target to depend on a conditional attribute.
    - Execute the target **if** the attribute is true, or execute is **unless** the attribute is true.

Executing targets

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <target name = "info">
      <echo>Hello World - Welcome to Apache Ant!</echo>
   </target>
</project>
```

```
>> ant
Buildfile: build.xml
info: [echo] Hello World - Welcome to Apache
Ant!
BUILD SUCCESSFUL
Total time: 0 seconds
```

- In the command line, invoke:
  - **ant <target name>**
- If no target name is supplied, the default will be executed.
  - In this case, **ant** and **ant info** will give the same result because info is the default target.

Properties

- XML does not natively allow variable declaration.
  - Instead, Ant allows the creation of **property** elements, which can be referred to by name.

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <property name = "sitename" value = "http://cse.sc.edu"/>
   <target name = "info">
      <echo>Apache Ant version is ${ant.version} - You are at ${sitename} </echo>
   </target>
</project>
```

- Properties have a name and a value.
  - Property value is referred to as **${property name}**.
  - Ant pre-defines **ant.version**, **ant.file** (location of the build file), **ant.project.name**, **ant.project.default-target**, and other properties.

Property Files

- A separate file can be used to define a set of static properties.
  - Allows reuse of a build file in different execution environments (development, testing, production).
  - Allows easy lookup of property values.
- Typically called **build.properties** and stored in the same directory as the build script.
  - Lists one property per line: `<name> = <value>`
  - Comments can be added using `# <comment>`

Property Files

- build.xml

```
<?xml version = "1.0"?>
<project name = "Hello World Project" default = "info">
   <property file = "build.properties"/>
   <target name = "info">
      <echo>You are at ${sitename}, version ${buildversion}.</echo>
   </target>
</project>
```

- build.properties

```
# The Site Name
sitename = http://cse.sc.edu
buildversion = 3.3.2
```

Conditions

```
<target name = "myTarget" depends = "myTarget.check" if = "myTarget.run"> .... </target>
<target name = "myTarget.check">
    <condition property = "myTarget.run">
        <and>
            <available file = "foo.txt"/>
            <available file = "bar.txt"/>
        </and>
    </condition>
</target>
```

- Conditions are properties whose value is determined by **and** and **or** expressions.
  - **And** requires each listed property to be true.
    - In this case, both foo.txt and bar.txt must exist.
      - (**available** is an Ant command that checks for file existence)
  - **Or** requires only one listed property to be true.
  - Calling target **myTarget.check** creates a property (**myTarget.run**) that is true if both files are present.
  - When **myTarget** is called, it will run only if myTarget.run is true.

Ant Utilities

- **Fileset** generates a list of files matching set criteria for inclusion or exclusion.
  - ** means that the file can be in any subdirectory.
  - * allows partial file name matches.

```
<fileset dir = "${src}" casesensitive = "yes">
   <include name = "**/*.java"/>
   <exclude name = "**/*Stub*"/>
</fileset>
```

- **Path** is used to represent a classpath.
  - **pathelement** is used to add items or other paths to the path.

```
<path id = "build.classpath.jar">
   <pathelement path = "${env.J2EE_HOME}/j2ee.jar"/>
   <fileset dir = "lib"> <include name = "**/*.jar"/> </fileset>
</path>
```

Building a Project

```
<project name = "Hello-World" basedir = "." default = "build">
   <property name = "src.dir" value = "src"/>
   <property name = "build.dir" value = "target"/>
   <path id = "master-classpath">
      <fileset dir = "${src.dir}/lib"> <include name = "*.jar"/> </fileset>
      <pathelement path = "${build.dir}"/>
   </path>

   ...
</project>
```

- Properties **src.dir** and **build.dir** define where the source files are stored and where the built classes are deployed.
- Path **master-classpath** includes all JAR files in the lib folder and all files in the build.dir folder.

Building a Project

```
<project name = "Hello-World" basedir = "." default = "build">

   ...

   <target name = "clean" description = "Clean output directories">
      <delete>
         <fileset dir = "${build.dir}">
            <include name = "**/*.class"/>
         </fileset>
      </delete>
   </target>
</project>
```

- The clean target is used to prepare for the build process by cleaning up any remnants of previous builds.
  - In this case, it deletes all compiled files (.class)
  - May also remove JAR files or other temporary artifacts that will be regenerated by the build.

Building a Project

```
<project name = "Hello-World" basedir = "." default = "build">

   ...

    <target name = "build" description = "Compile source tree java files">
      <mkdir dir = "${build.dir}"/>
      <javac destdir = "${build.dir}" source = "1.8" target = "1.8">
         <src path = "${src.dir}"/>
         <classpath refid = "master-classpath"/>
      </javac>
   </target>

</project>
```

- The build target will create the build directory, compile the source code (using javac), and place the class files in the build directory.
  - Can specify which java version to target (1.8).
  - Must reference the classpath to use during compilation.

Creating a JAR File

- The **jar** command is used to create a JAR (executable) from your compiled classes.

```
<target name = "package">

    <jar destfile = "lib/util.jar" basedir = "${build.dir}/classes"
        includes = "app/util/**" excludes = "**/Test.class">

    <manifest><attribute name = "Main-Class" value = "com.util.Util"/></manifest>

    </jar>

</target>
```

  - **destfile** is the location to place the JAR file.
  - **basedir** is the base directory of included files.
  - **includes** defines the files to include in the JAR.
  - **excludes** prevents certain files from being added.
  - The **manifest** declares metadata about the JAR.
    - Attribute Main-Class makes the JAR executable.

Running Unit Tests

- JUnit tests are run using the **junit** command.

```
<target name = "test">
   <junit haltonfailure = "true" haltonerror = "false"
          printsummary = "true" timeout = "5000">
      <test name = "com.utils.UtilsTest"/>
   </junit>
</target>
```

  - **test** entries list the test classes to execute.
  - **haltonfailure** will stop test execution if any tests fail, **haltonerror** if errors occur.
  - **printsummary** displays test statistics (number of tests run, number of failures/errors, time elapsed).
  - **timeout** will stop a test and issue an error if the specified time limit is exceeded.

Best Practices

- Automate everything you can!
  - Ant can integrate with version control, run scripts, send files, zip files, etc.
  - Use it as a comprehensive project management tool.
- Require all team members to use Ant.
  - Even if different team members use different IDEs or workflow, make them use Ant to build the project.
  - Require an Ant build before checking changes into version control.
- Provide a “clean” target.
  - All build files need the ability to clean up before a fresh build. Clean should only retain the files in VCS.

Best Practices: Follow Consistent Naming Conventions

- Call the build file **build.xml**, properties should be stored in **build.properties**.
  - And these should be in the root of the project.
- Prefix internal targets with a hyphen.
  - “build” might be available for external use, but a subtarget “-build.part1” might not be intended for use in isolation.
  - By prefixing a hyphen, you give readers context.
  - Hyphenated targets also cannot be run from the command line.
- Format and document the XML file.
  - Try to make the file readable to the human eye.

Best Practices: Design for Maintenance

- Will your build file be readable in the future?
- Will the file execute on a clean machine?
  - Document the build process.
    - Write a text file describing the build and deployment process.
    - List programs and libraries needed for the build.
  - Avoid dependencies on programs/JAR files that are not stored with the project.
    - If licensing allows, store external libraries with the project for easier builds.
  - Do not distribute usernames/passwords in the build files. These change + this is bad security.

Continuous Integration

Continuous Integration

- Development practice that requires code be frequently checked into a shared repository.
- Each check-in is then verified by an automated build.
  - The system is compiled and subjected to an automated test suite, then packaged into a new executable.
  - Uses the build script you wrote.
- By integrating regularly, developers can detect errors quickly, and locate them more easily.

CI Practices

- Maintain a code repository.
- Automate the build.
- Make the build self-testing.
- Every commit should be built.
- Keep the build fast.
- Test in a clone of the production environment.
- Make it easy to get the latest executable.
- Everyone can see build results.
- Automate deployment.

How Integration is Performed

- Developers check out code to their machine.
- Changes are committed to the repository.
- The CI server:
  - Monitors the repository and checks out changes when they occur.
  - Builds the system and runs unit/integration tests.
  - Releases deployable artefacts for testing.
  - Assigns a build label to the version of the code.
  - Informs the team of the successful build.

How Integration is Performed

- If the build or tests fail, the CI server alerts the team.
  - The team fixes the issue at the earliest opportunity.
  - Developers are expected not to check in code they know is broken.
  - Developers are expected to write and run tests on all code before checking it in.
  - No one is allowed to check in while a build is broken.
- Continue to continually integrate and test throughout the project.

TravisCI

- CI service that is free for open-source developers, hooked into GitHub.
- Connects to a GitHub repository and performs the CI process at specified times.
  - When code is pushed to a repository.
  - When a pull request is created.
- Adds a “badge” to the GitHub project page displaying the current build status.

TravisCI Process

- When code is checked into a repository, TravisCI starts a **job**.
  - An automated process that clones the repository into a virtual environment.
    - An isolated environment with a clean OS install.
  - A job is split into a series of **phases**.
    - Sequential steps of a job.
    - Three core phases in TravisCI:
      - **Install**: Installs required dependencies in the virtual environment.
      - **Script:** Performs build tasks (compile, test, package, etc.)
      - **Deploy:** Deploy code to a production environment (Amazon, Heroku, etc.)

The TravisCI Configuration File

- Travis uses a config file, **.travis.yml**, to determine how to build the project.

```
language: java
jdk: oraclejdk8
install: ...
script: ...
```

  - **Language** informs TravisCI which language you are developing in.
    - There is a default build process for all supported languages.
  - For Java, the **jdk** field lists the compiler you want to use to build.

The TravisCI Configuration File

```
os: linux
```

- Used to determine the OS you want to build on. Supports Linux and MacOS.

```
addons:
 apt:
   packages:
     - maven
```

- **Addons** are additional programs you need to perform a build.
  - **Apt** is a package manager used in Linux.
  - This example says to install the Maven package before performing the build.

The TravisCI Configuration File

```
env:
- MY_VAR=EverythingIsAwesome
- NODE_ENV=TEST
```

- Env is used to set up environmental variables needed to perform a build.

```
before_install: (after_install, before_script, after_script, etc)
  - ...
```

- Used to perform commands before or after one of the major phases (install, script, deploy).

Install, Script, Deploy

- Major phases specified by listing a set of commands to run.
- If you have a build file, you do not need to explicitly specify commands.
  - TravisCI can detect Ant, Maven, and Gradle build files and has default targets it will run.
  - By default, the script phase will execute "**ant test**".
    - By convention, this will compile and test the project.
    - If you want to execute different targets instead, you can specify this in the configuration file.

Best Practices

- Minimize build time.
  - Time spent waiting for results is wasted time.
  - Do not make developers wait more than 10 min.
    - If they need to switch tasks, that adds time.
  - TravisCI can execute jobs in parallel. Split the test suite into multiple jobs and execute them concurrently in their own virtual environments.
- Pull complex logic into shell scripts.
  - The configuration file will run any commands you list.
  - If your build task is complex, split commands into their own file and call that file.
  - Scripts can be run outside of TravisCI too.

Best Practices

- Test multiple language versions for libraries.
  - Libraries need to operate in multiple version of a language. Make sure you can build in each of them.
  - You can specify multiple versions in the configuration file (i.e., openjdk8, openjdk9).
    - Each will be tried when you build.
- Skip unnecessary builds
  - If you just change documentation or comments, there is no reason to re-test.
  - Skip commits by adding “[ci skip]” to the commit message.
  - Can also cancel builds on the TravisCI website.

Ant and TravisCI Demo

Ant:
https://github.com/apache/commons-lang/blob/687b2e62b7c6e81cd9d5c872b7fa9cc8fd3f1509/build.xml
TravisCI: https://github.com/Greg4cr/defects4j/blob/master/.travis.yml

We Have Learned

- Testing is not all that can be automated.
  - Project compilation, installation, deployment, etc.
- **Project build automation:**
  - Automating the entire compilation, testing, and deployment process.
  - Ant is an XML-based language for automating the build process.
- **Continuous integration:**
  - Executing and managing the build process each time code is checked in.
  - TravisCI is a common, free CI system.

Next Time

- Unit Testing Laboratory
  - Bring a laptop (at least one per group), with an IDE installed that supports JUnit (Eclipse, IntelliJ).
  - Download code for MeetingPlanner from the course website and import it into the IDE.

- Assignment 2
  - Due March 3rd