Development Infrastructure

II

Stepan Bolotnikov
Last time

- IDEs and IntelliJ IDEA
- VCS and Git
Today - Build and Collaboration Tools

- Dependency management and build automation
  - Don’t reinvent the wheel
  - Reduce, reuse, recycle
  - The less manual work the better
  - Gradle

- Wiki
  - Share and maintain documentation
  - Bitbucket

- Issue tracking
  - Plan and keep track of work
  - Bitbucket
But first! VCS in IDE

- Adding, committing changes
- Pulling, pushing
- Resolving conflicts
Automation: why

● Not reinventing the wheel every time is at the heart of software development
● Without task automation, development is more
  ○ Boring
  ○ Repetitive
  ○ Error-prone
  ○ Slow
● and less
  ○ Scalable
  ○ Efficient
  ○ Clear
  ○ Repeatable
Automation: what

- Generation of documentation, styleguides, etc
- Deployment
- Testing
- Building, packaging
- Dependency management
Automation: how

- Build scripts!
  - Special files that describe what and how to build
- Different formats and languages for different languages/environments/stacks
- You can always just use Make
Make (1976)

- Still widely used, especially in Unix
- By default described in Makefile
Makefile structure

main.o: main.c defs.h
    cc -c main.c

kbd.o: kbd.c defs.h command.h
    cc -c kbd.c

command.o: command.c defs.h command.h
    cc -c command.c

display.o: display.c defs.h
    cc -c display.c
Makefile structure

```
main.o: main.c defs.h
    cc -c main.c

kbd.o: kbd.c defs.h command.h
    cc -c kbd.c

command.o: command.c defs.h
command.h
    cc -c command.c

display.o: display.c defs.h
    cc -c display.c
```
Makefile structure

```
main.o: main.c defs.h
    cc -c main.c

kbd.o: kbd.c defs.h command.h
    cc -c kbd.c

command.o: command.c defs.h command.h
    cc -c command.c

display.o: display.c defs.h
    cc -c display.c
```

Rules:

1) **Target**: usually the name of the generated file, can also be an action, e.g. “clean”
### Makefile structure

main.o: main.c defs.h
	cc -c main.c

kbd.o: kbd.c defs.h command.h
	cc -c kbd.c

command.o: command.c defs.h command.h
	cc -c command.c

display.o: display.c defs.h
	cc -c display.c

---

**Rules:**

1) **Target:** usually the name of the generated file, can also be an action, e.g. “clean”

2) **Dependencies:** usually files that the command uses as input or other actions that need to run before
Makefile structure

main.o: main.c defs.h
       cc -c main.c

kbd.o: kbd.c defs.h command.h
     cc -c kbd.c

command.o: command.c defs.h command.h
          cc -c command.c

display.o: display.c defs.h
          cc -c display.c

Rules

1) **Target**: usually the name of the generated file, can also be an action, e.g. “clean”

2) **Dependencies**: usually files that the command uses as input or other actions that need to run before

3) **Recipe**: commands/CLIs that need to run as part of this rule
Dependencies

- You want to use (certain versions of) libraries and packages
- Those libraries and packages want to use (certain versions of) other libraries and packages
- Etc
- Impossible to handle manually
- Modern dependency management tools
  - Declare your dependencies
  - Dependencies declare their own dependencies
  - The software builds up a graph
  - Gets the relevant versions of relevant packages
  - Exposes relevant versions to relevant software
Back to Java

- Apache Ant
  - XML-based description of tasks for building the project
  - Dependency management through Apache Ivy
  - Verbose
  - Custom
<?xml version="1.0"?>
<project name="Hello" default="compile">
  <target name="clean" description="remove intermediate files">
    <delete dir="classes"/>
  </target>
  <target name="clobber" depends="clean" description="remove all artifact files">
    <delete file="hello.jar"/>
  </target>
  <target name="compile" depends="resolve" description="compile the Java source">
    <mkdir dir="classes"/>
    <javac srcdir="." destdir="classes"/>
  </target>
  <target name="jar" depends="compile" description="create a Jar file for the application">
    <jar destfile="hello.jar">
      <fileset dir="classes" includes="**/*.class"/>
      <manifest>
        <attribute name="Main-Class" value="HelloProgram"/>
      </manifest>
    </jar>
  </target>
  <target name="resolve" description="--> retrieve dependencies with Ivy">
    <ivy:retrieve/>
  </target>
</project>

<ivy-module version="2.0">
  <info organisation="org.apache" module="hello-ivy"/>
  <dependencies>
    <dependency org="commons-lang" name="commons-lang" rev="2.0"/>
    <dependency org="commons-cli" name="commons-cli" rev="1.0"/>
  </dependencies>
</ivy-module>
Apache Maven

- XML-based
- Handles both building and dependencies
- Standardized java project structure (directories etc)
- Project described as Project Object Model - pom.xml file
Example pom.xml

```xml
<project>
  <!-- model version is always 4.0.0 for Maven 2.x POMs -->
  <modelVersion>4.0.0</modelVersion>
  <!-- project coordinates, i.e. a group of values which uniquely identify this project -->
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>
  <!-- library dependencies -->
  <dependencies>
    <!-- coordinates of the required library -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <!-- this dependency is only used for running and compiling tests -->
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```
Convention over Configuration

- As long as you “follow the Maven way”, you need to write very little
- Only exceptions, not rules, need to be specified
- Previous example is enough to build the project and run tests
  - Assuming that source and test files are in the correct locations
- → pom.xml is much less verbose
- → Once you know how to navigate one Maven project, you know how to
  navigate them all
Gradle

- Open-source
- Build on the ideas of Ant and Maven
- No longer XML!
  - Groovy - Java-syntax-like programming language
  - Can write full scripts
  - Can include plugins and extend build files
Gradle plugins

apply plugin: 'java'

- Already defines most of the tasks you’d need for a standard Java project
- Assumes Maven directory structure

- Plugins are Groovy scripts
- Easy to maintain and share
- Many available online - Google before writing your own tasks
Demo time

- Gradle build files
  - Plugins
  - Dependencies
  - Tasks
- Running tasks from command line
- IDE integration
In conclusion

- Build automation software helps you make processes in your application’s life cycle less error prone, scalable, etc
- Dependency management software helps you use third-party libraries
- Gradle lets you write Groovy scripts to automate both
- Gradle scripts can be extended with plugins
There’s a lot of information surrounding a project

- Documentation
- Guides
- Development plans
- etc

- Wiki!
  - Interlinkable
  - In a consistent location
  - Can be edited by contributors
  - Version controlled
Wiki

- Rich textual content
  - Links, tables, images,…
- Doesn’t require programming knowledge
  - Often uses the Markdown markup language
- Kept in a separate Git repository
  - Can see history
  - Can roll back changes
Demo time

- Enabling wiki & creating pages
- Markdown essentials
- Editing pages, seeing history
● How to know what needs to be done
● How to know what is broken
● How to break needed work into smaller bits
● How to plan development sprints

● Issue tracking!
Issue tracking software

- Create issues, tasks, bug reports
- Track the progress
- Allow non-developers to request features/fixes
- Often part of repository hosting
  - Github
  - Gitlab
  - Bitbucket
- Sometimes separate software
  - Jira
Planning development and sprints

- Many issue tracking solutions allow to organise tasks into “Boards”
- Kanban, Agile, etc
- Get a quick overview of state of project
- Group issues into categories
- Plan development sprints
How issue tracking hooks into VCS

- Separate branches for issues
  - Bitbucket branching model
- Smart commit messages
  - Close commits from VCS
Demo time

- Adding issues
- Issue life cycle
- Creating branches for issues
- Smart commits
- Boards
Collaborating on code in 2018: pull requests

- Do work on a separate branch
  - In a separate (forked) repository in case of open-source project
- When ready, open a pull request
  - Sometimes called merge request
- Signals the maintainer that your code is ready to be included
Don’t just accept any changes from anyone

- Before merging a PR, thoroughly review changes
- PRs can be commented on
- You can (and should) perform Code Review
  - Sanity check
  - Code style
  - Errors
Code review

- Everyone can review changes and leave their comments
- Certain people can be assigned as reviewers
  - People familiar with the code
- You can set up permissions so code review is mandatory
Demo time

- Opening pull requests
- Code review
In conclusion

- Use wiki to document your project
- Use issue tracker to plan development and report bugs
- Use branching to your advantage
That’s not all, folks!

● Many tools helping development
  ○ Continuous Integration / Continuous Delivery
  ○ Automated testing

● Many alternatives to the tools we’ve covered

● Find ones you like, learn to use them well

● Keep up with the news!
Questions? Comments?