Build process & build management

PTS3 / FMFI UK / 24.4.2024

Jana Kostičová

Build process

Building SW = processing source code and its dependencies into a form that can be executed or used by a computer system.

It depends on various factors what activities are involved (programming language, size and type of application, target environment, ...)

- Writing / generating sources
- Static checking
- ….

Broader build process

- Preprocessing (C / C++, some Fortran dialects, ..)
- Compilation (compiled languages)
- Linking (C / C++, assembly language, ..)
- Dependency resolution
- Packaging / bundling

Core build process

- Automated tests
- Deployment
- ….

Broader build process

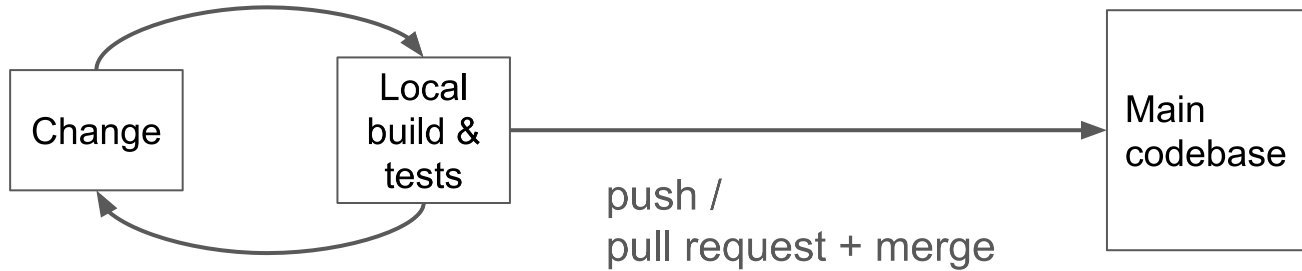
Build tools

- Manual building - may suffice for extremely small projects with minimal dependencies
- Build tools - highly recommended if we have anything more complex

Key functionalities

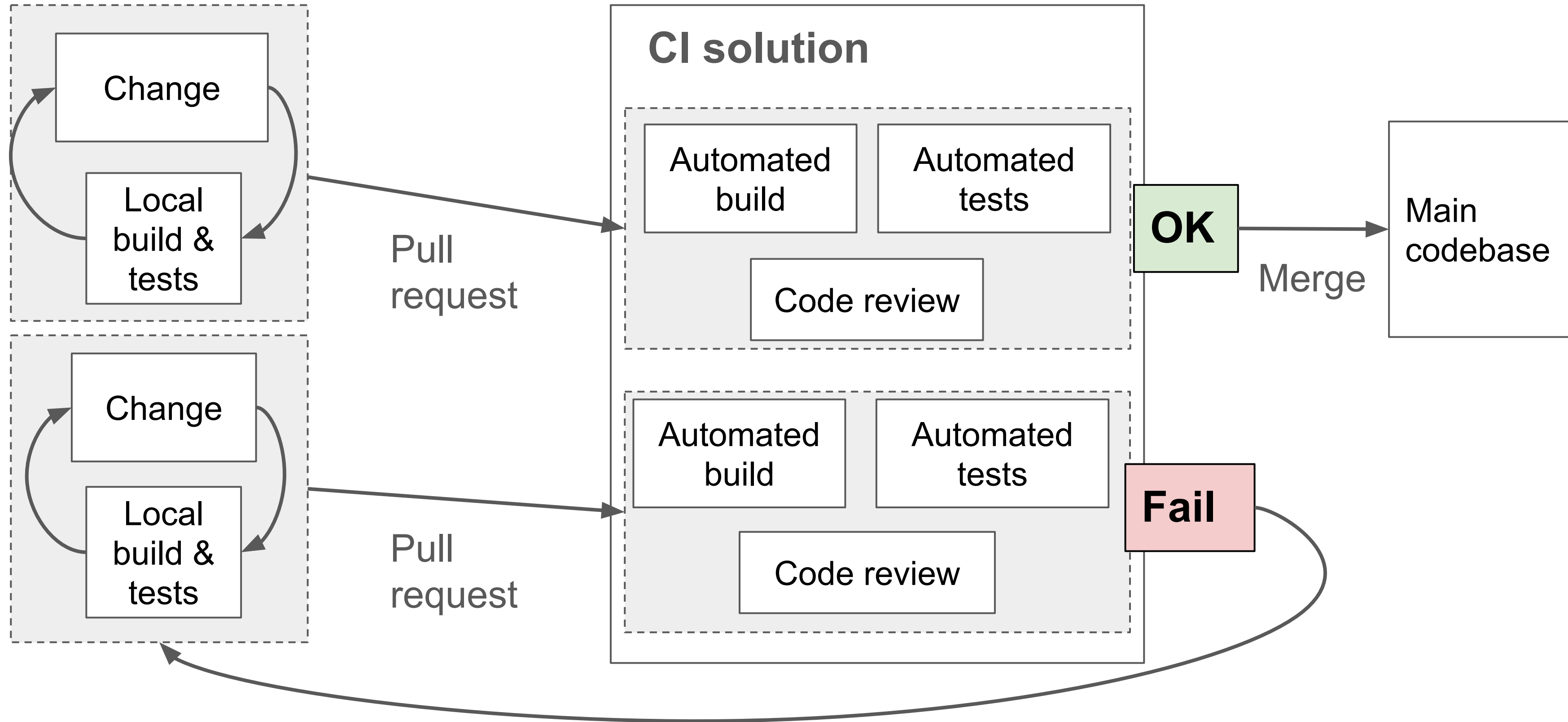
- Build automation
  - Scripting tasks for efficient and repeatable builds
- Build configuration
  - Defining different build configurations (e.g., development, testing, production).
- Dependency management
  - Automatically managing external libraries needed by the software
- Running automated tests
- Facilitating Continuous Integration (CI)

Simple projects



- Build tools might not be needed
- Code quality is only checked locally

Continuous Integration (CI) workflow



Local builds vs CI builds

- Local builds
  - Provide a fast feedback loop for developers during their coding process
  - Help to catch errors early on before committing code
- CI builds
  - Offer a “safety net” by ensuring code integrates and functions correctly before merging into the main codebase
  - Provides feedback to the developer on the success or failure of the build and test
  - Requires **automatization of build process**

- Together, they help maintain **code quality** and catch issues early in the development cycle
- Both of them should use the same build tool and similar configuration
  - Some differences may appear (e.g., omitting some tests in local environment)

Build tools - examples

- Make
- Apache Ant
- Apache Maven
- Gradle
- Npm (Node.js / JavaScript)
-

GNU make

- Emerged in the late 1970s and still used today
- Popular for C/C++, can be used also for other languages
- Configuration file: Makefile

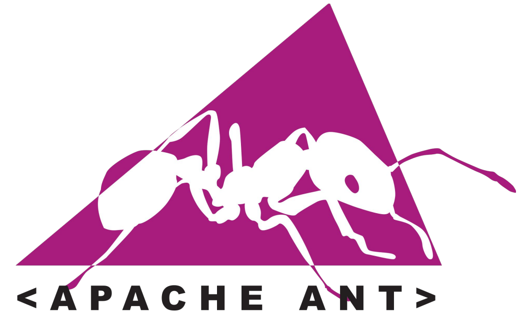
Main features

- Tracks dependencies between target files and their prerequisites efficiently
- Rebuilds only what's necessary based on changes
- Extensible through scripting and external tools

Limitations

- Relies on shell scripting
- Works well with simple to moderately complex projects, but very large projects can become challenging to maintain
- Does not provide support for managing external dependencies (core requirement for Java projects)

Apache Ant



- Released in 2000
- Configuration file: build.xml

Main features
- Uses XML
- Specifies build steps and tracks compilation dependencies similarly to “make” tool (imperative approach)
- Provides limited dependency management
- Cross-platform
- Extensible
- Adapts to existing project layout

Limitations
- Limited dependency management - Ant has no means of downloading external dependencies from a central repository or resolve version conflicts
- No direct support for running automated tests

Apache Maven

- Released in 2004
- Emerged from within the Java ecosystem to address its specific needs
- Configuration file: pom.xml

Main features

- It uses XML
- Predefined build phases - build process is defined using plugins and goals, build phases are performed based on this configuration (declarative approach)
- Less verbose than Ant
- Robust dependency management - libraries are downloaded from maven repository, versions are tracked explicitly allowing version conflict detection

Limitations

- Steeper learning curve
- It works best with standard Maven project structure
- Customization might be complex - it is required to be familiar with Maven plugin development

Apache Maven - default build phases

- validate
- compile
- test
- package (e.g. to JAR / WAR / EAR file)
- integrate-test
- verify (additional checks on the packaged artifact)
- install (installing packaged artifact into local Maven repository)
- deploy (deploying packaged artifact into remote repository)

Gradle

- Released in 2009
- Configuration file: build.gradle

Main features

- Concise syntax - uses Groovy and Kotlin-based DSL
- Flexible approach to determine the execution order of the tasks (DAG)
- Robust dependency management

Limitations

- High flexibility may lead to complex configuration files with hard-to-understand semantics

References

- GNU make: https://www.gnu.org/software/make/
- Apache Ant: https://ant.apache.org/
- Apache Maven: https://maven.apache.org/
- Gradle: https://gradle.org/