

Introduction to Maven

Building, Testing, and Assembling Applications

Welcome to Introduction to Maven

In this class you will learn how to use Maven to build your projects.

Goals and Objectives

At the end of this class, you should know how to:

- Download and install Maven
- Run a simple build
- Configure and locate dependencies

More Goals and Objectives

- Configure unit testing
- Customize build output and packages
- Understand the basics of build customization
- Understand multi-module builds

Prerequisites

Here are a few prerequisites we've assumed...

You are Comfortable with the OS

You should know how to:

- Download a file to your workstation.
- Set an environment variable.
- Run shell and batch scripts.

You Understand Java Development

We assume that you:

- Know how to install Java.
- Understand the structure of simple programs.
- Have had experience compiling Java code.

Target Technologies

We have developed this class to address a set of technologies.

Operating Systems

This class targets:



Windows 7 or Higher OSX Mountain Lion

but you can use Linux too - at your own risk ;-)

Java Version

This class is based on Java 7.



Can you use Java 6 or Java 8? Use at your own risk.

Note: Examples may not work.

IDE

The examples in this class will feature Eclipse Luna.



But you can use IntelliJ or NetBeans, if that is what you are familiar with.
Again, use an alternative IDE at your own risk.

Instructor Intro

Who's teaching this class?

Planned Class Structure

In this class we are going to:

- Learn by doing
- Talk "Theory" only as-needed
- Emphasize demos over talk

Please Interrupt Me

..., if you have a question.

- Ask away.
- We've built in time for questions.
- Questions help tailor content to specific concerns.

Note: Instructors are available after class.

Building a Simple Project

Module Objectives

- Introduce an example project
- Examine the pom.xml
- Explain the directory structure
- Introduce coordinates
- Introduce dependencies

Do First, Learn Later

Our training philosophy is straightforward...

We learn from experience, not from watching slides.

Let's get to work.

First Example: Copy

Introducing a simple example:

```
$ cp -R labs/simple-project working
```

First Example: Purpose

This example is a single-module Maven project.

A command-line application which:

- Demonstrates using Maven.
- Builds a self-executable JAR.
- Shows a non-trivial POM example.

First Example: Student Lookup

A University has a Registrar to check student status.

- Registrar fires up a CLI application.
- Enters in a Student's ID Number.
- Student is Full Time if Credits > 12.

Building the Project

Let's build this project...

```
$ mvn clean package
```

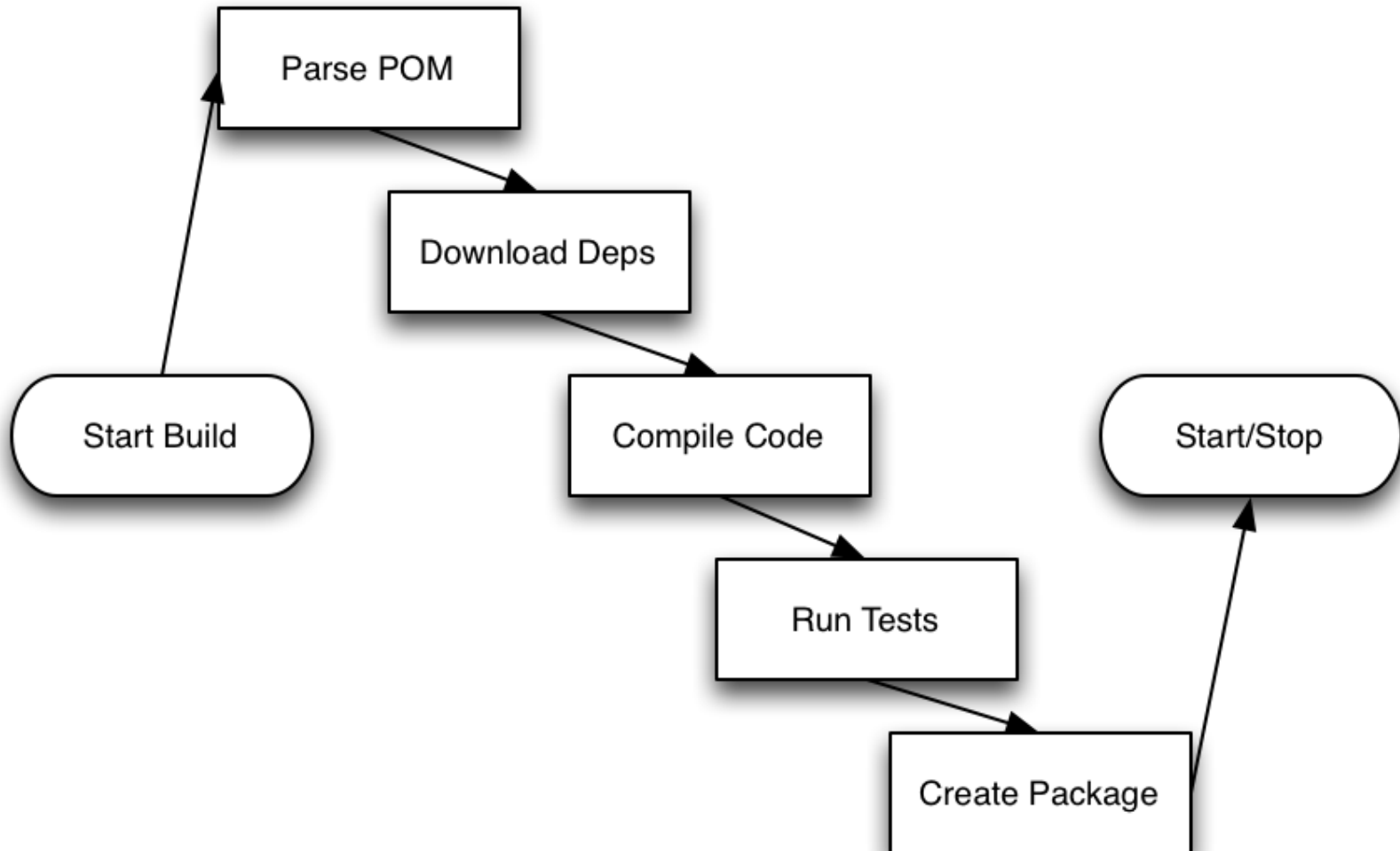
Note: This should take about a minute to complete.

What Just Happened?

The Maven build...

- ...**inspected** a file called the Project Object Model (POM)
- ...**downloaded** dependencies.
- ...**compiled** Java code into byte-code.
- ...**ran** a series of Unit Tests.
- ...**assembled** everything into a JAR file.

What Just Happened?



Running the Project

After the build, we have the following:

- `simple-project-1.0-SNAPSHOT.jar` in `target/`
- A number of JARs in `target/`

Run the example application like this:

```
$ cd target  
$ java -jar simple-project-1.0-SNAPSHOT.jar
```

Build Output

For later...

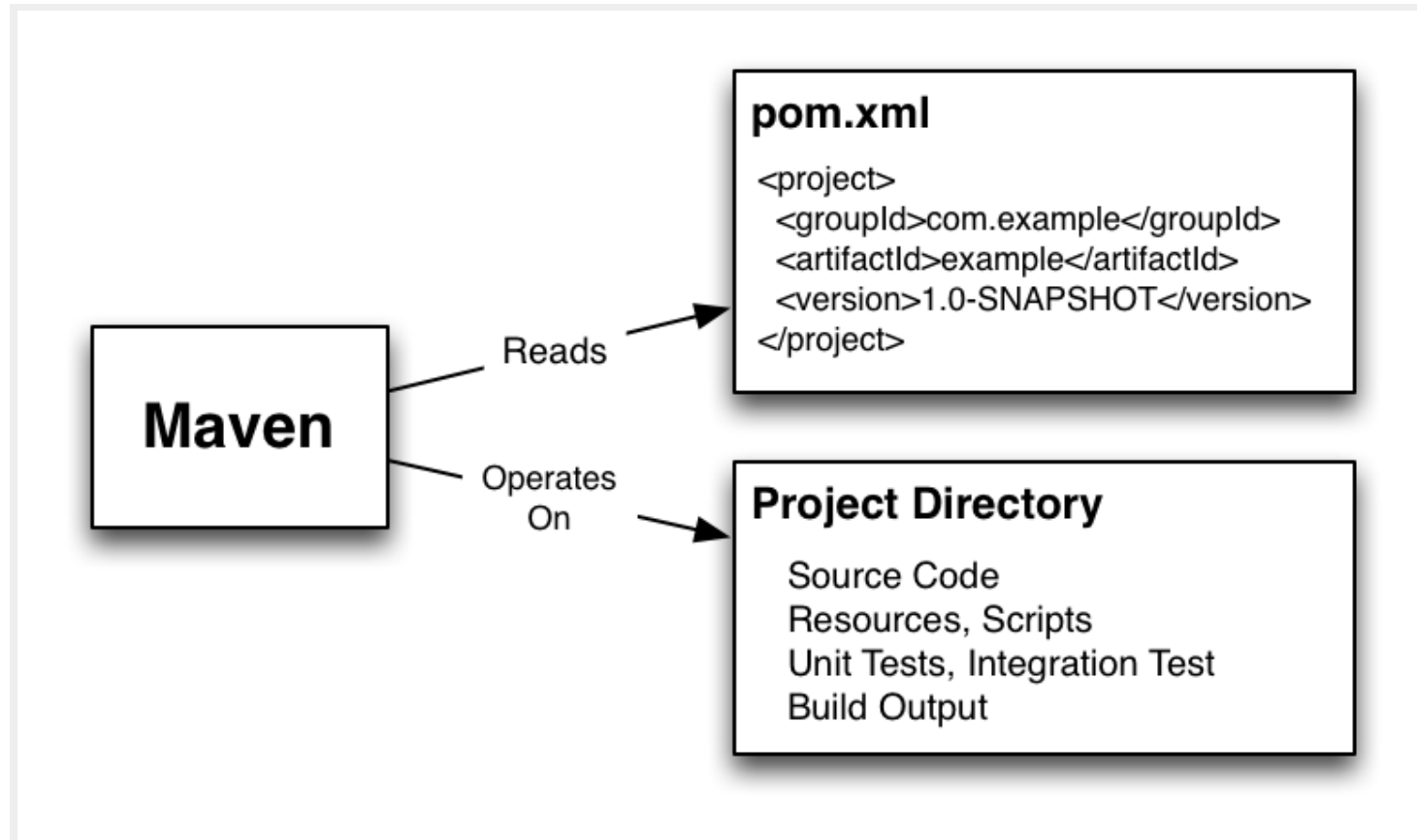
- `target/` contains build output
- Build configured the Jar Manifest
- Build copied dependencies to `target/`
- Build output is an executable (!) jar file

What is a Maven Project?

A Maven project is...

1. A collection of files and folders
2. Including a pom.xml file

What's in a POM?



Everything is in the POM - the Basics...

- Directory structure
- Coordinates
- Dependencies

... and Configuration

Plugins are configured in the POM for:

- Build configuration
- Unit testing configuration
- Package customizations
- ...

...and Project "Metadata"...

Projects often point to external resources...

- Continuous integration server configuration
- Project information e.g. name, description, developers, organization,...
- Source code management system
- Repository manager configuration

...and more

- Container configuration - servlet container or an application server.
- Custom user properties
- Anything related to your build!

Focus on First Steps...

Learning Maven is a process. The first steps?

- Directory structure
- Coordinates
- Dependencies

Directory Structure

The first thing to understand:

- there is a directory structure
- and it is based on conventions

Example Project Contents

Our example project has...

- Java source files
 - Java unit tests
- .. but could also have
- properties files,
 - a web application descriptor,
 - Javascript and HTML,
 - images, and more

A Common Structure

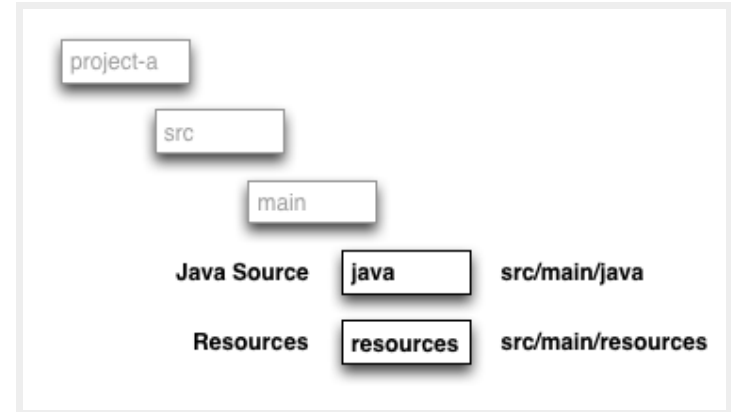
Maven provides a common structure

"Everything in its right place."

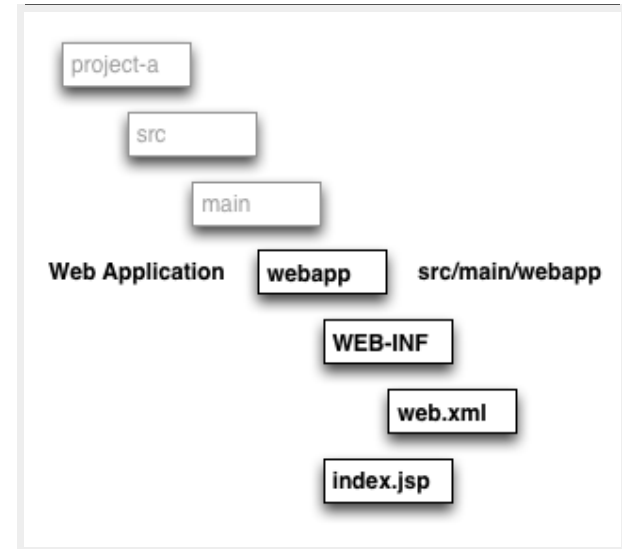
Source and Resources

- **Source code:** `src/main/java`
- **Resources:** `src/main/resources`

Both directories added to class-path and the resulting artifact.



Web Application

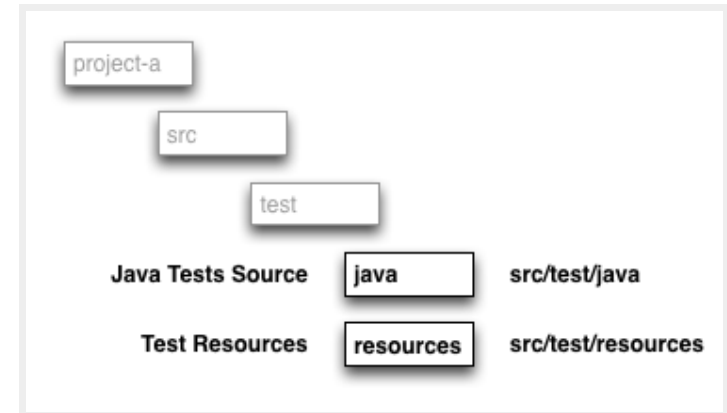


- **Web Application:** `src/main/webapp`
Stored in the resulting artifact.

Unit Tests

- **Unit Tests:** `src/test/java`
- **Test Resources:** `src/test/resources`

Not stored in the resulting artifact.



Demo: Inspect Example Artifact

Coordinates

Everything has a Coordinate

- Projects
- Dependencies
- Maven Plugins



Give Me the Coordinates?

1. Group ID
2. Artifact ID
3. Version

Group ID : Artifact ID : Version

general
identifier

specific
identifier

Coordinate Systems

Coordinate systems bring order to artifacts.

Group ID

- **Identifier:** `groupId`
- **Role:** Artifacts (JARs, WARs, EARs) are organized into groups.
- Groups correspond to organizations and projects.

Groups contain Many Artifacts

Groups are often organizational groups, but they can also be nested. The following are all Group IDs.

- **Google** - com.google
- **Google Android** - com.google.android
- **Apache** - org.apache
- **Takari** - io.takari

Think of a Group as a...



Artifact ID

- **Identifier:** `artifactId`
- **Role:** Projects produce artifacts.

"This project produces a WAR artifact."

A Maven project produces a single artifact.

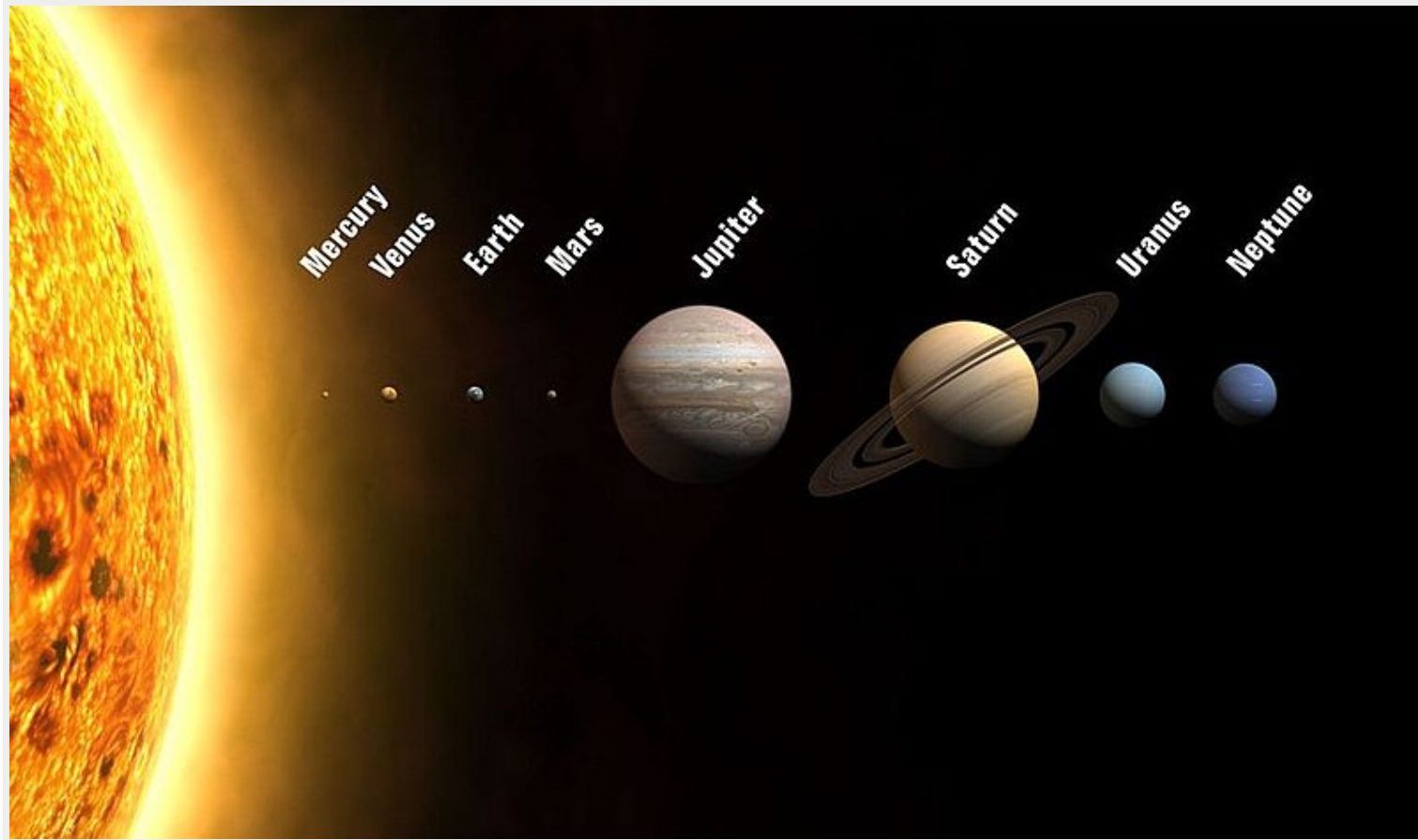
Artifacts contain Many Versions

Artifacts correspond to projects. For example Google Guice has an artifact.

The following are all Artifact IDs listed with corresponding Group IDs.

- `com.google:guice`
- `org.apache.maven:maven-core`
- `log4j:log4j`

Think of an Artifact as a...



Version

- **Identifier:** `version`
- **Role:** Software changes over time and releases need to be versioned.

Artifact Versions cause Unique Artifacts

There may be several versions of a single artifact.

- `com.google:guice:0.1`
- `com.google:guice:2.0`
- `com.google:guice:3.1`

A version is like a...



Example Coordinates

Open up our example project's pom.xml and see this...

```
<project>
  <groupId>io.takari.training.simple</groupId>
  <artifactId>simple-project</artifactId>
  <version>1.0-SNAPSHOT</version>
  ...
</project>
```

Example Coordinates

- `groupId: "io.takari.training.simple"`
- `artifactId: "simple-project"`
- `version: "1.0-SNAPSHOT"`

GAV Coordinate

Group ID, Artifact ID, Version

```
io.takari.training.simple:simple-project:1.0-SNAPSHOT
```

This coordinate set **uniquely** identifies an artifact.

Coordinates Affect Output

```
<project>  
  <groupId>com.example</groupId>  
  <artifactId>sample-project</artifactId>  
  <version>1.0-SNAPSHOT</version>  
</project>
```



sample-project-1.0-SNAPSHOT.jar

simple-project-1.0-SNAPSHOT.jar

Demo: Change Coordinates

If we change a project's coordinates, we change the project's output.

Dependencies

These days...

- No project is an island
- You use open source libraries
- Examples: Commons-*, Guice, Derby, Tomcat, Spring etc.

Example Project

We built an application that does the following:

- Parses command line inputs
- Creates and manages objects
- Uses DI to simplify codebase and separate concerns

Next steps could be to create a web interface for the application.

Example Project Dependencies

We assembled this project using open source libraries and could add more for further features:

- Dependency Injection: Guice
- Embedded Database: HSQLDB and jDBI
- REST API: Jersey
- Client-side Javascript: React

Example pom.xml Dependencies

```
<dependency>  
  <groupId>org.apache.commons</groupId>  
  <artifactId>commons-compress</artifactId>  
  <version>1.9</version>  
</dependency>
```

Dependencies have Coordinates

Take Commons-Compress as an example:

- **groupId:** org.apache.commons
- **artifactId:** commons-compress
- **version:** 1.9

Maven Downloads Dependencies

Maven turns these coordinates into a URL.

```
/org/apache/commons/  
    commons-compress/  
        1.9/  
            commons-compress-1.9.jar
```

Download dependencies

- from the Central Repository
- to your Local Repository.

Dependencies have Scope

This example uses two scopes:

- compile
- test

In Summary...

mvn clean package

Building a Maven Project is straightforward.

Project Object Model (POM)

All Maven projects have a POM.

An XML file -> pom.xml

Projects have Coordinates

When you create a project, you have to give it a GAV coordinate.

Projects have Dependencies

A big feature of Maven is the ability to declare project dependencies.

Dependencies have Coordinates

Coordinates enable Maven to go get artifacts from repositories.

Working with Dependencies

Where were we?

In the previous section you learned that:

- Dependencies have coordinates
- Maven downloads dependencies

Module Objectives

- Demo: Add a new dependency
- Coordinate details (5 minutes)
- Dependencies
 - Where do dependencies come from?
 - Where dependencies go?
 - Maven repository format
 - Release versions

Let's add a Feature

Our application could use some caching.

Let's add a caching library.

Demo: To Find a New Dependency...

... use the Central Repository at <http://search.maven.org>

We're looking for Guava.

Google Guava implements a cache that we can use in our application.

Wait? How did you find Guava?

Interesting question...

- There's no central site to find OSS
- You really just have to "know"
- Talk to your peers
- Follow the OSS projects
- Read a lot of DZone and InfoQ
- ...

Demo: Select the Appropriate Version

Locate an artifact on the Central Repository

Show list of available versions

Demo: Find the appropriate coordinates

Once you've located the artifact, grab the coordinates.

Demo:

Add the Coordinates to the POM

Once you have the appropriate coordinates, add them to the POM.

Demo: Run the build and track the download

Run the build, observe the artifact download.

Demo:

Modify project to use new dependency

Once the artifact is downloaded and available, we can use it.

Coordinate Details

- Rounding out Coordinates with
- `packaging and`
- `classifier`

GAV Coordinate Review

In the previous module we learned:

- groupId
- artifactId
- version

Dependencies have coordinates

Our dependencies:

- groupId:artifactId:version
- com.sample:sample-artifact:1.0

Two more coordinates...

In addition to the three GAV coordinates, there are two more:

- packaging
- classifier

Coordinate: Packaging

- Implicit value is "jar"
- Influences packaging type of a build
- Possible values? Anything...
 - zip
 - war
 - ear
 - sar
 - pom
 - apk

Coordinate: Classifier

- Rarely used
- Can be used for:
 - JDK compatibility
 - Extra Packages and Assemblies

Summary: There are 5 Coordinates

- groupId
- artifactId
- version
- packaging (*default 'jar'*)
- classifier (*optional*)

Next: How does Maven download a dependency?

How do Dependencies Work?

Once you tell Maven what you need,
how does it get the dependency files?

Maven sees a Coordinate

When you configure a dependency in a pom.xml,
Maven sees a coordinate.

```
com.sample:sample-artifact:1.0:jar
```

Maven creates a URL

`com.sample:sample-artifact:1.0:jar`

Turns into:

`http://repo1.maven.org/maven2/com/sample/sample-artifact/1.0/sample-artifact-1.0.jar`

Coordinates form a Pattern

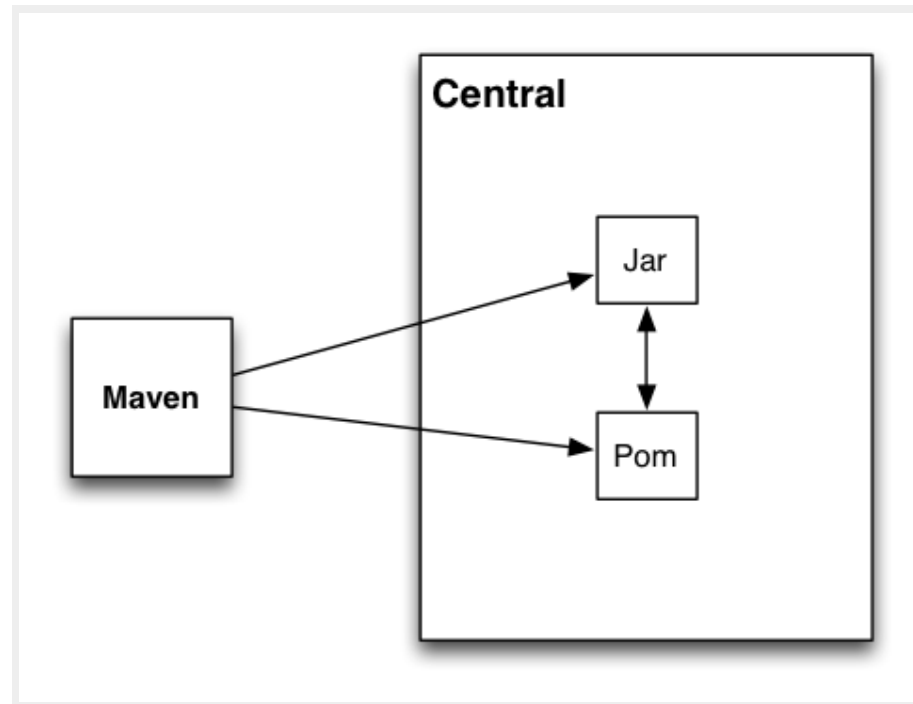
Breaking this URL down into component sections:

- Repository URL (default is `http://repo1.maven.org/maven2`)
- `/ {groupId}`
- `/ {artifactId}`
- `/ {version}`
- `/ {artifactId} - {version} . {packaging}`

Maven Downloads the Dependency

dependency

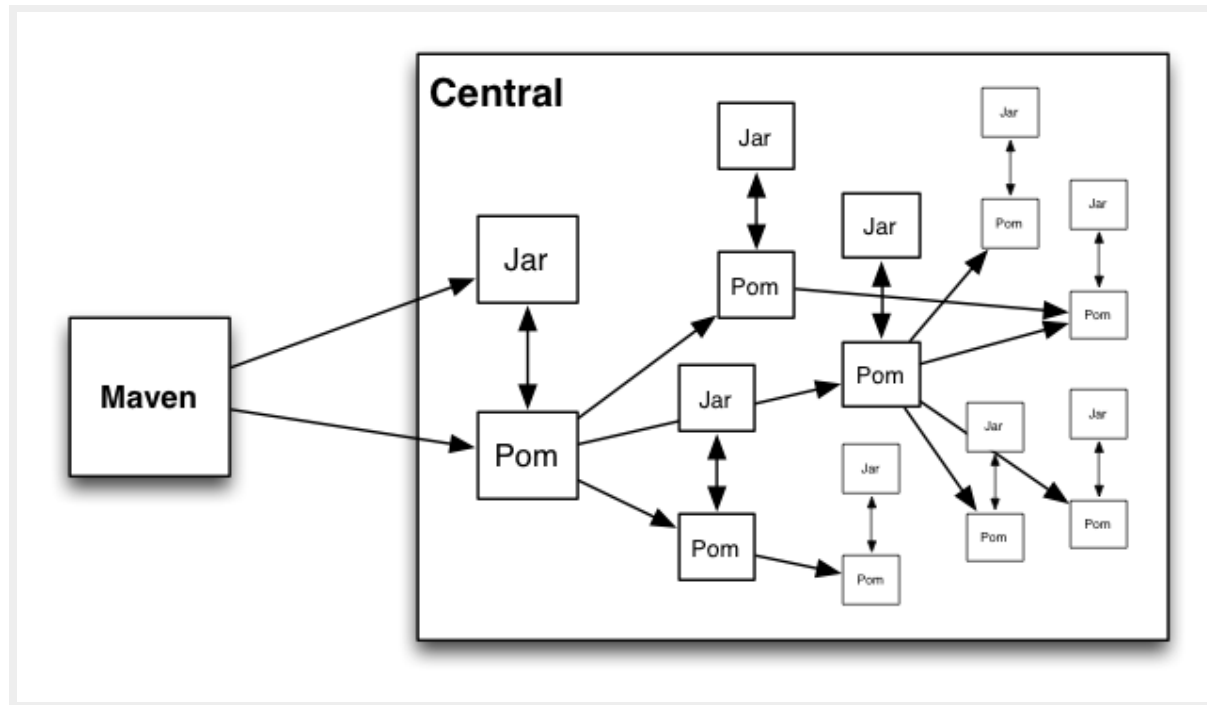
It also downloads a POM at the same coordinates...



The POM for a dependency points to other dependencies.

Maven Builds a Graph of Dependencies

Maven grabs your dependencies...



...then it grabs your dependencies' dependencies
...and so on.

Transitive Dependencies

Your dependencies can have other dependencies. For example:

- Google Sitebricks has dependencies on e.g. Google Guava, Jackson and Google Guice.
- To use Google Sitebricks you'll need to download these libraries.
- There are a lot more in fact...

Demo: Show Complex Dependency Graph

Take a look at a large project's dependency graph

Maven then Creates a Classpath

There are a few different class-paths:

- for compilation
- for test execution
- for packaging
- for runtime usage

Dependency Scopes

Maven has several scopes, here are two:

- **compile** - for compilation, packaging, and runtime
- **test** - for test compilation and execution

Demo: Dependency Resolution Result for Compile Scope

Demo:

Add a Dependency to the Test Scope

Summary: Coordinates and Dependencies

It's easy to manage dependencies

Coordinates form Basis of Dependency Management

Maven turn Coordinates into a URL

Maven downloads artifacts + POMs

Maven builds a Graph

Maven turns Graph into Class-paths

Scopes Influence Class-paths

Working with Unit Tests

Where were we?

You just learned about coordinates and dependencies.
Now, we're going to dive into Maven's execution model
by learning how Maven executes unit tests.

Module Objectives

- Introduce Surefire plugin
- Demonstrate Surefire plugin
- Running a single test
- Including and excluding tests
- Discuss test resources
- Demonstrate test resources

Demo: Execute Unit Tests

Open simple-project and execute unit tests.

Demo: Execute a Single Unit Test

Pinpoint one unit test and run it.

What happened here?

- Instead of packaging a project...
- We executed unit tests
- Unit testing is the domain of the...

Maven Surefire Plugin

What is a Plugin? Quickly.

If you use Maven you'll need to understand:

- That most of what happens in Maven, happens in plugins.

Everything is a Goal

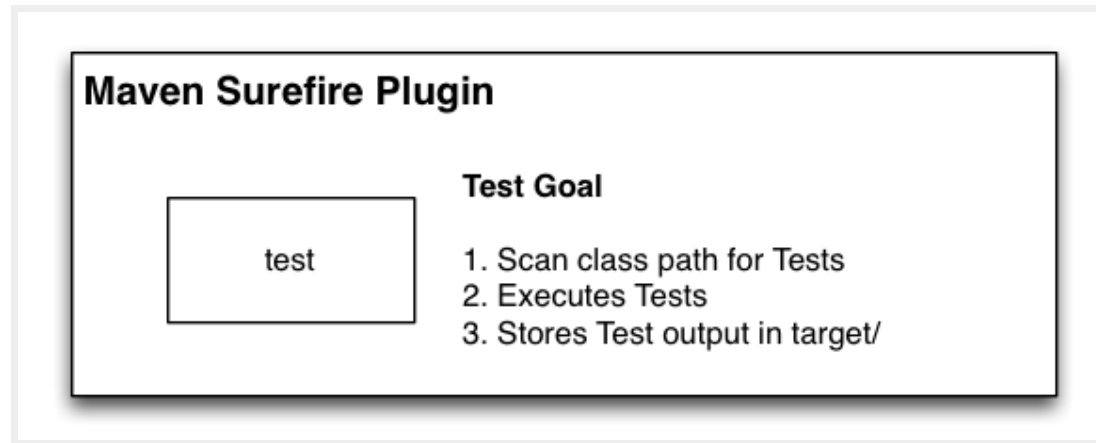
In Maven, every action is a goal.

From compilation to running unit tests to packaging.

Every Goal is in a Plugin

Every goal is contained in a plugin.

In Maven, the Surefire plugin is responsible for unit testing.



Maven's Surefire Plugin

Provides one important goal:

- test - Runs a projects Unit Tests

[Maven Surefire Plugin Page](#)

Surefire's Test Goal is a Default

When we run "mvn test" we invoke a phase.
The Surefire test goal is bound to this phase .
It's a default. (We'll learn more about this later.)

Running Unit Tests in Maven

Unit tests are an integral part of software development.

This SOUNDS good, but it's really true. So true that tests are "baked-in" to the default Maven lifecycle.

Test in Project Layout

- Unit tests are found in `src/test/java`
- Test resources are in `src/test/resources`

Test Output

- Bytecode goes into target/test-classes
- Reports go into target/surefire-reports

Demo: Demonstrate Tests Failing a Build

By default, a failing test will fail the entire Maven build.

Here's how...

Demo: Using a Command-line Option to Skip Tests

In a particularly large project it often makes sense to skip tests:

- There could be tests that take hours to complete.
- Or, there could be failing tests.

Let's try

```
mvn install -DskipTests=true  
mvn install -Dmaven.test.skip=true
```

Modifying Plugin Behaviour - from the Command-line

What just happened?

- The Surefire Plugins's...
- `test` goal ...
- changed behaviour based on a command-line argument.


This is one of several ways to influence build behaviour.

Test Reporting

Maven offers some simple reports for Unit Tests.

Generating Reporting Data

Maven generates data that can be easily understood by other tools.

Overview
[Introduction](#)
[Usage](#)
[Goals](#)
[FAQ](#)
Examples
Project Documentation
[Project Information](#)
[Project Reports](#)
[Checkstyle](#)
[Cobertura Test](#)
[Coverage](#)
[CPD Report](#)
[JavaDocs](#)
Maven Surefire Report
[Plugin documentation](#)
[PMD Report](#)
[Source Xref](#)
[Tag List](#)
[Test Source Xref](#)


Summary

[\[Summary\]](#)[\[Package List\]](#)[\[Test Cases\]](#)

Tests	Errors	Failures	Success Rate	Time
24	0	0	100%	1.281

Note: failures are anticipated and checked for with assertions while errors are unanticipated.





Package List

[\[Summary\]](#)[\[Package List\]](#)[\[Test Cases\]](#)

Package	Tests	Errors	Failures	Success Rate	Time
org.apache.maven.plugins.surefire.report	24	0	0	100%	1.281

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.



[org.apache.maven.plugins.surefire.report](#)

	Class	Tests	Errors	Failures	Success Rate	Time
	ReportTestCaseTest	4	0	0	100%	0
	ReportTestSuiteTest	7	0	0	100%	0.125
	SurefireReportMojoTest	4	0	0	100%	1.093
	SurefireReportParserTest	9	0	0	100%	0.063

Test Cases

[\[Summary\]](#)[\[Package List\]](#)[\[Test Cases\]](#)

[ReportTestCaseTest](#)

	testSetName	0.094
	testSetTestCases	0

Demo: Show simple HTML Report

View the HTML generated by the Site plugin.

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-report-plugin</artifactId>
      <version>2.17</version>
    </plugin>
  </plugins>
</reporting>
```

Surefire Reports

Surefire is an example of a Maven plugin, which generates data, that can be reported on.

This enables integration with 3rd-party tools like Eclipse, Jenkins, and SonarQube.

Summary

You've just encountered quite a bit, that needs to be explained....

Tests Run by Default

Nothing needs to be configured to run Java tests in JUnit.

Maven Surefire Plugin Runs Tests

Goals are the lowest 'unit-of-work' in Maven.

Similar goals are assembled into Plugins.

Goal Behaviour can be Customized

We changed behaviour from the command-line.

By passing in the -D option, we can change the way Maven executes goals.

Tests Generate Reports

Maven is about more than just building software.

Maven can generate reports that give you visibility into test success or failure.

Standards Make Integration Easier

If you are developing a tool like m2eclipse or Jenkins, it helps to have conventions that everyone agrees upon.

- Standard directory layout
- How tests are executed
- When tests are executed

Building Your Software

Where were we?

Last module was a brief introduction to the Surefire Plugin.

- Tests are integral to a build
- Surefire plugin has a Test goal
- This goal can be configured
- Tests generate reports

Module Objectives

- What is a Goal?
- What is the Lifecycle?
- Convention over configuration

What is a Goal?

Test Goal

In the previous module we ran the "test" goal.

- A single action.
- Configurable from the command-line
- Take a look at this goals configurable properties...

Demo: Goal Documentation for test

The best way to understand the configuration options for a plugin is to...
look at the [goal documentation](#).

Other goals in our build

Our default build does many things...

- It compiles Java code.
- It runs unit tests.
- It creates a package.

Each of these actions is encapsulated in a goal.

Demo: Goal Documentation for compile

The project compiles Java code, right? Well...

Look at the [compile goal documentation](#).

Demo: Customize the Compile Goal

Target different Java versions with "source" and "target".
From the command-line.

Configuring Goals in a POM

The compile Goal of the compiler plugin can be configured in a POM.

```
<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>7</source>
          <target>7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Demo: Configure Compile Goal in POM

Let's use that previous example in a POM and see what happens.

Configure Compiler Behaviour in the POM

Configuring a Goal Execution

In the last demonstration we configured a Goal execution.

- Goals are executed at a specific stage in the lifecycle
- The Lifecycle is a sequence of stages
- called "phases"
- ...more on this later

What is a Plugin?

Zooming Out a Level

- Reiterating what a plugin is...
- ...a plugin contains related goals.
- Let's explore some of the plugins we're using...

Demo: Maven Surefire Plugin Documentation

- `surefire:help`
- `surefire:test`

Demo: Maven Compiler Plugin Documentation

- `compiler:compile`
- `compiler:help`
- `compiler:testCompile`

Demo: Maven War Plugin Documentation

- `war:exploded`
- `war:help`
- `war:inplace`
- `war:manifest`
- `war:war`

Plugins have Coordinates

Now, let's tie plugins back to dependencies. Plugins are a lot like dependencies - they have coordinates:

- groupId
- artifactId
- version

Plugins are Artifacts

Maven downloads plugins as needed.

Just like dependencies.

Also... Maven can have more than one version of a plugin.

Demo: Show Maven Downloading Plugins

Let's modify a POM and change a plugin version.

Watch Maven download this plugin. Where does it go?

Demo: Configuring Goals at Plugin Level

Plugins can be configured at different "levels".

Let's move plugin configuration from an execution to the plugin level.

Ways to Configure a Goal

- From the command-line
- In a goal execution
- In the plugin configuration

Why do these goals run?

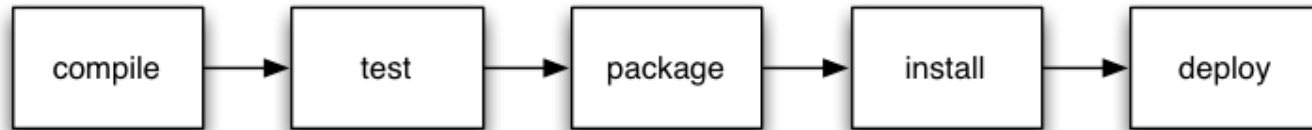
We've hinted at the Lifecycle...

...but we haven't explained it...

The Lifecycle

Zooming Out Another Level

A *VERY* Simplified View of the Lifecycle



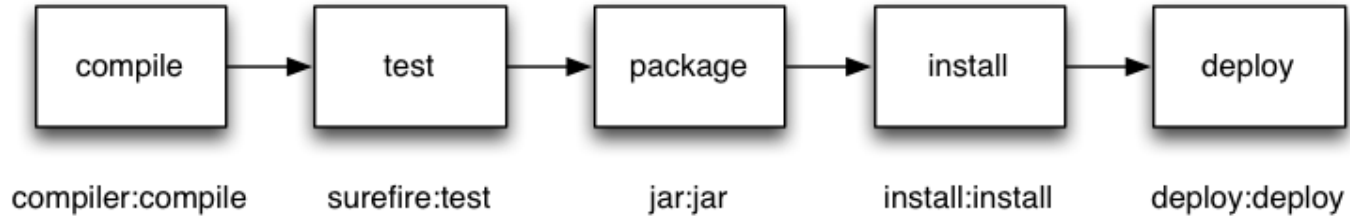
Everything Hinges on the Lifecycle

The Lifecycle is...

- A single, linear series of steps (or phases)
- Goals are bound to these phases.

Goals are Bound to Lifecycle

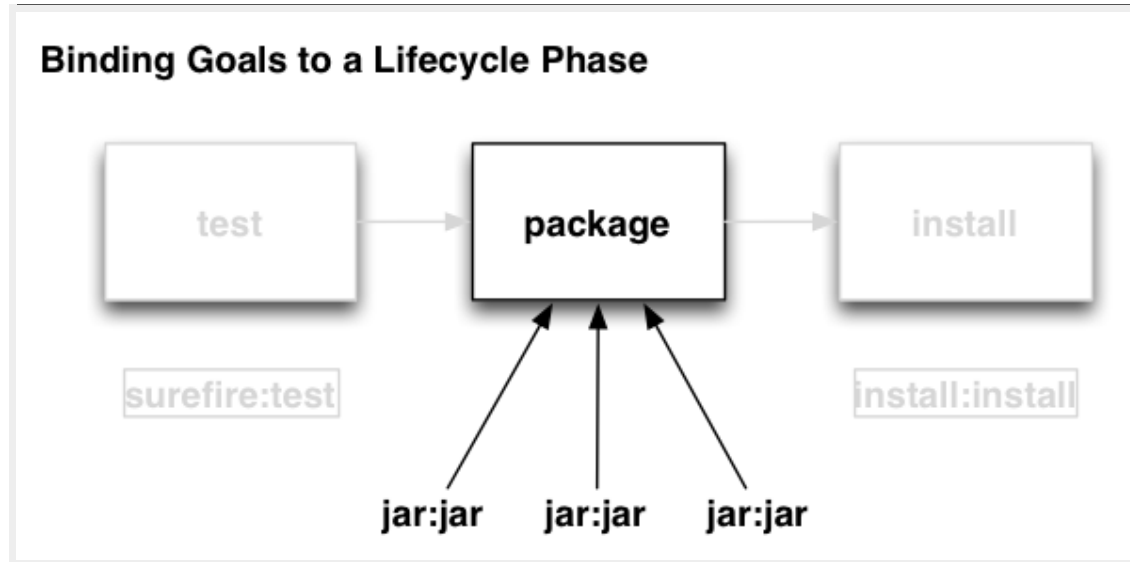
A ***VERY*** Simplified View of the Lifecycle



The Default Lifecycle

```
validate, initialize
generate-sources, process-sources
generate-resources, process-resources
compile, process-classes
generate-test-sources, process-test-sources,
generate-test-resources, process-test-resources
test-compile, process-test-classes,
test
prepare-package, package
pre-integration-test, integration-test, post-integration-test, verify
install, deploy
```

Goals Bind to Lifecycle Phases



Demo Running Phases::

Test

```
mvn test
```

Demo Running Phases:

Install

```
mvn install
```

Default Lifecycle Bindings

process-resources

org.apache.maven.plugins:maven-resources-plugin:2.6:resources

compile

org.apache.maven.plugins:maven-compiler-plugin:2.5.1:compile

process-test-resources

org.apache.maven.plugins:maven-resources-plugin:2.6:testResources

test-compile

org.apache.maven.plugins:maven-compiler-plugin:2.5.1:testCompile

test

org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test

package

org.apache.maven.plugins:maven-jar-plugin:2.4:jar

install

org.apache.maven.plugins:maven-install-plugin:2.4:install

deploy

org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy

Customizing the Lifecycle

How do you add new goals to the Lifecycle?

This is explored later in the class.

Summary

Goals -> Plugins -> Lifecycle

Plugins are Artifacts

- Plugins are artifacts
- They have coordinates
- Maven downloads them on-demand

Plugins Aggregate Goals

- Plugins collect similar goals together

Every Action is a Goal

- Everything that happens in Maven is a Goal

Goals Bind to Phases

- When you execute goals, you bind them to phases in the Maven Lifecycle.

Lifecycle: Sequence of Phases

- The Lifecycle is everything.
- Understand the Lifecycle and Maven will make sense.

Default Lifecycle

- The only reason Maven does anything by default.
- There is a default lifecycle...
- ...with default lifecycle bindings
- ...defined within Maven itself.

Packaging Your Software

Where were we?

In the previous sections:

- You've learned about plugins
- and the lifecycle
- and how they relate to each other.

Module Objectives

This module does two things.

- It focuses on packaging
- and it uses a simple Assembly:

Remember the lifecycle from the previous section,
well the packaging changes it...

And here is what we are going to do

- Demonstrate packaging
- Demonstrate (but don't fully delve into) the Assembly plugin
- Build a program that creates a jar with dependencies
- Introduce SNAPSHOT versions

Creating Packages

While builds compile and unit test, there's an important last step - packaging.

- **Java applications** are packaged and distributed: JARs, WARs, EARs.
- **Alternative package formats** ZIP, Tarball, other language formats such as aar, gems and npm packages.

JARs, WARs, EARs

- Remember that packaging is a coordinate
- groupId, artifactId, version, **packaging**, classifier.

Packaging influences the package lifecycle phase.

JAR Lifecycle

When packaging is "jar":

- compile - compiler:compile
- test - surefire:test
- package - jar:jar
- install - install:install

This is a simplified lifecycle.

WAR Lifecycle

When packaging is "war":

- compile - compiler:compile
- test - surefire:test
- package - war:war
- install - install:install

WAR packaging has one difference: war:war instead of jar:jar

EAR Lifecycle

- generate-resources - ear:generate-application-xml
- package - ear:ear
- install - install:install

Same lifecycle, different bindings. EAR is very different.

Complex, Custom Packages

Need more advanced packages?

Use the Maven Assembly Plugin

Or a plugin that natively supports a specific packaging

Built-in Assemblies

The Maven Assembly plugin has a number of built-in descriptors:

- bin
- jar-with-dependencies
- src
- project

Custom Assemblies

You can also define your own assembly descriptors.

An assembly descriptor can:

- Copy files from your project to a custom directory structure.
- Include dependencies in your project's output.
- ...and much more.

Running Applications

- In this next series of demonstrations...
- We're going to create a self-executing JAR
- With a webapp we could deploy a WAR to a container.

Demo: JAR with Dependencies

- Let's configure an assembly that includes dependencies...
- ...and configure a MANIFEST.

Demo: Configuring the JAR Plugin

- Configuring the JAR plugin to include a Manifest.

Demo: Creating a Self-executing JAR

- Configuring the Assembly plugin to generate a Self-executing JAR
- Note that this example is advanced.
- Uses the Shade plugin + Nifty Executable.

Creating a Multi-module Project

Module Objectives

What if things are not that simple...

- How does Maven scale projects?
- Why would you do this? Why group projects together?
- Define "inheritance" and discuss what is inherited
- Define "aggregation" and discuss what is aggregated
- Demonstrate a multi-module Build
- Discuss the "reactor" and how it orders builds

Demo: Build a Large Multi-module Project

How Projects Scale in Maven

- Maven projects scale in a particular way.
- Through the aggregation of smaller projects.

This is a core assumption of Maven.

One common pattern: Monolithic Apps

A popular build pattern for large projects is...

- **The Monolith**
- Many modules crammed into one
- A very large, unmanageable build
- Produces multiple outputs

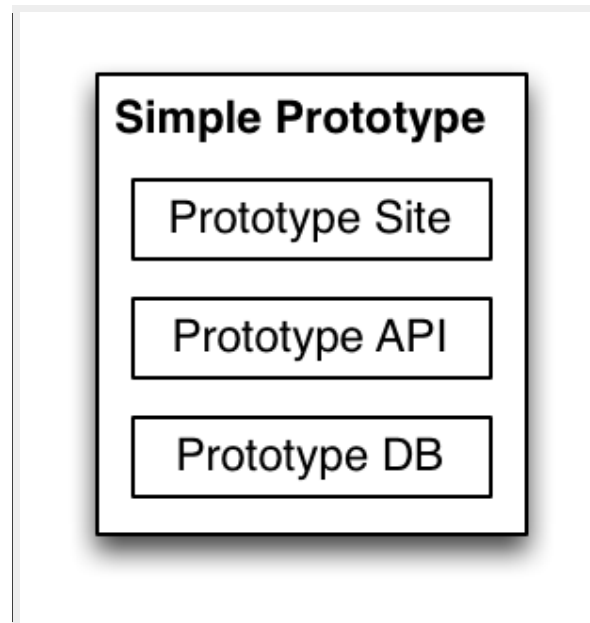
How All Projects Start: Prototypes

Almost every good idea starts simple.

- Facebook was once just a bunch of PHP code.
- Google was someone's Stanford thesis.
- Twitter was a side-project from Odeo.

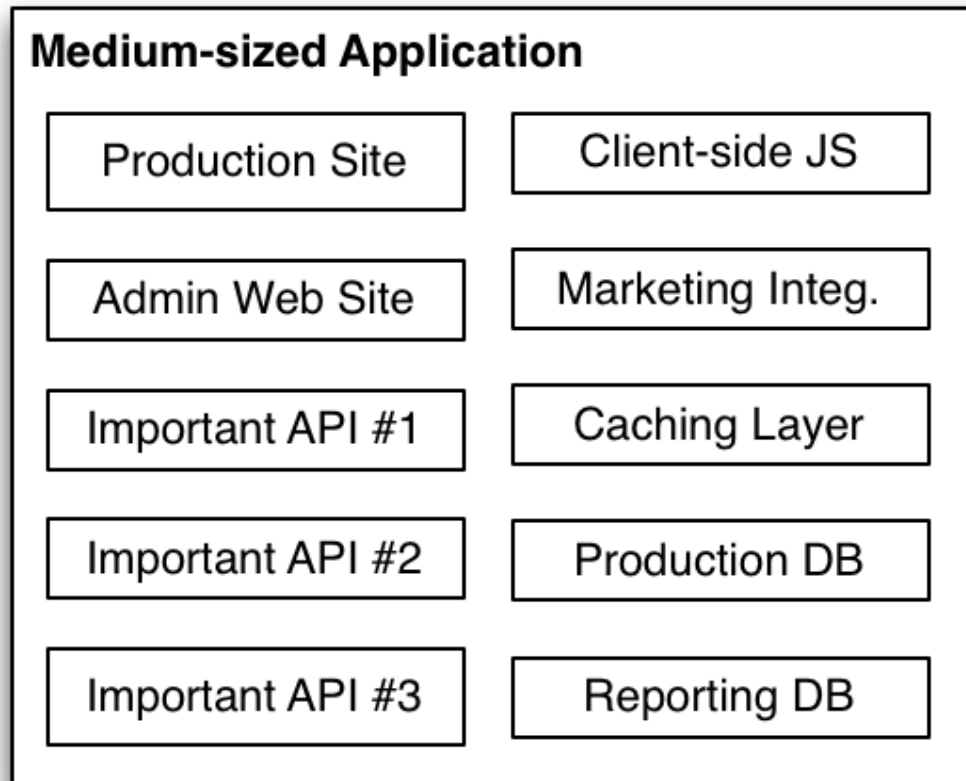
In the Beginning...

...the build is simple:



The plot thickens...

Your proof of concept worked and your project got staff and resources.



One day, you wake up to this...

Enterprise-sized Disaster Project

Production Site

CDN Library

Client-side JS #1

Custserv Site

Site Optimizer

Client-side JS #2

Inventory Site

Digital Media Stuff

Marketing Integ.

Admin Web Site

Cloud Integration

Caching Layer

Important API #1

Session Services

Session Services

Important API #2

Accounting Rpts

Production DB

Important API #3

Analytics Rpts

Inventory DB

Important API #4

Compliance Lib

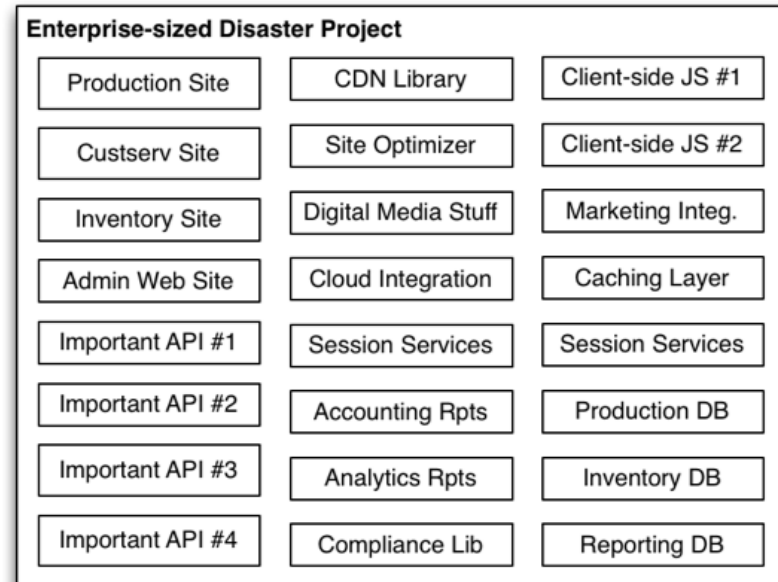
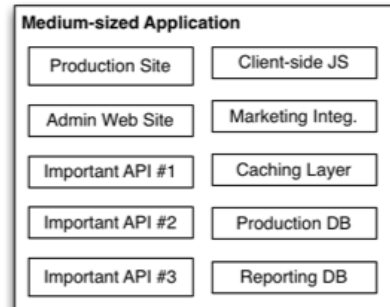
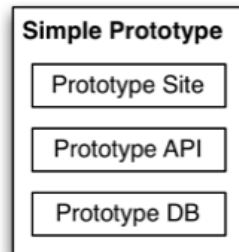
Reporting DB

Monoliths Don't Scale

Software lives for years (often decades)

- As systems and teams scale, single builds do not.
- As modularity develops, monolithic builds constrain.
- At scale, monolithic builds inhibit progress.

The End-game of a Monolith



Time →

3 developers
1 department
1 team

Build Time:
5 minutes

20 developers
1 department
3 teams

Build Time:
40 minutes

200 developers
4 departments
20 teams

Build Time:
6 hours

The End-game of a Monolith

- Developer productivity is affected as devs can't build local.
- Releases will take **forever** and the business will suffer.
- New projects are difficult to integrate
- And the build becomes too brittle to change.

Maven's Approach: Modularity

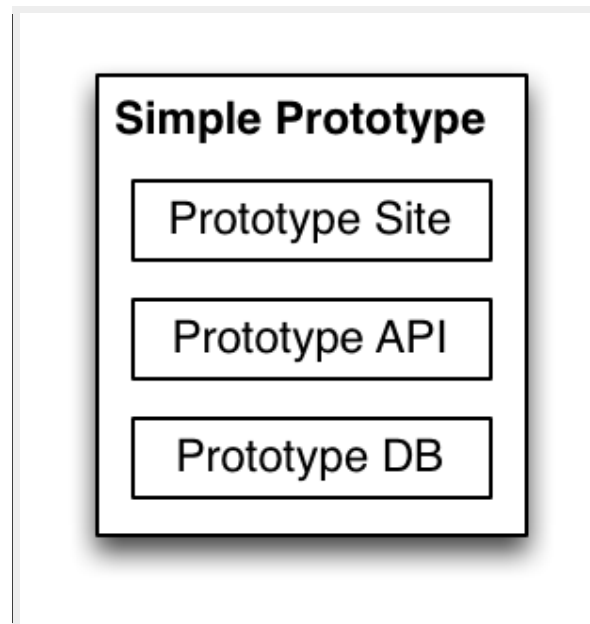
Maven favours smaller components.

Components that produce one artifact.

- Easier to model dependencies
- Easier to separate from larger builds
- Supported by dependency management

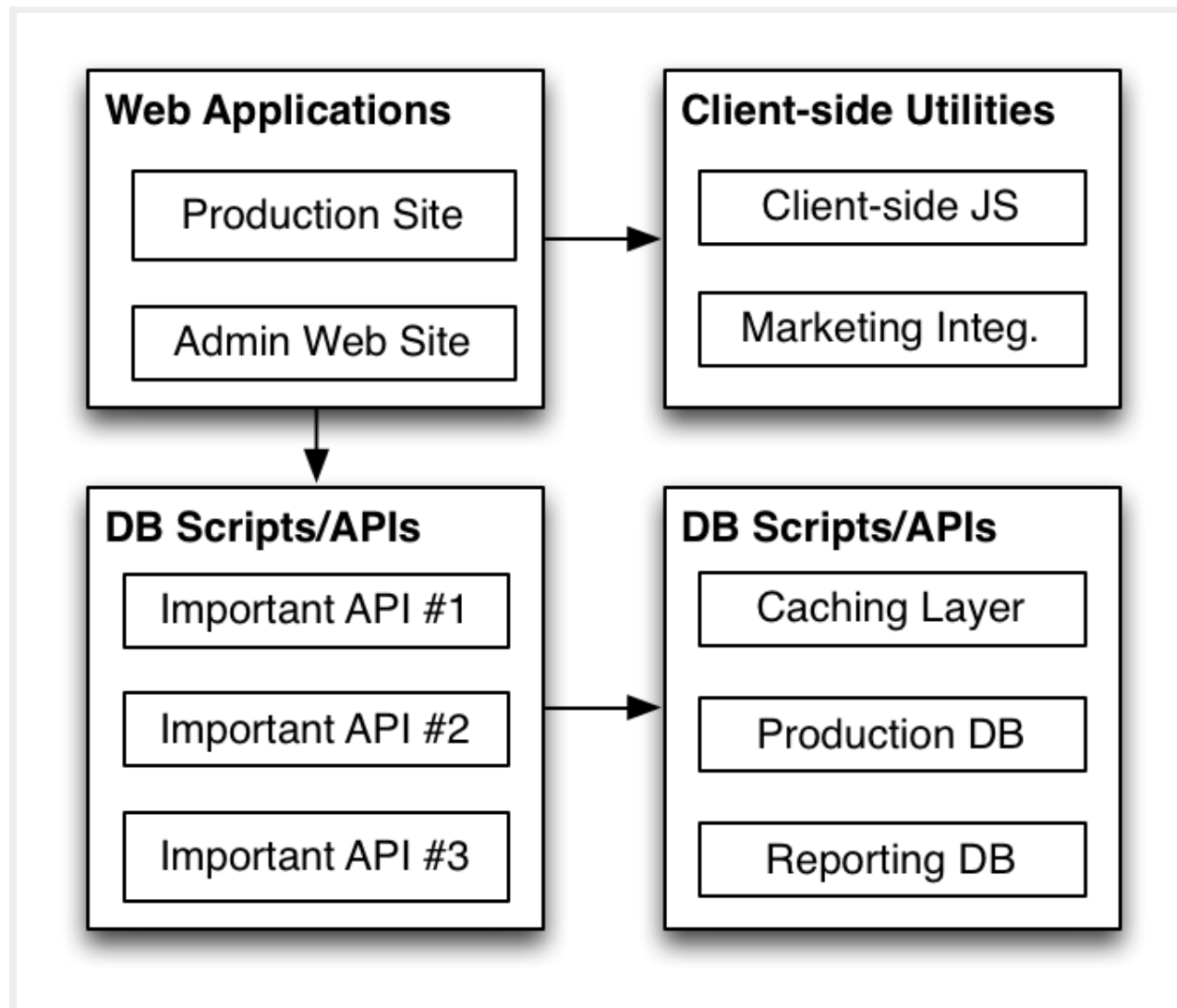
In the Beginning...

...the build is simple:

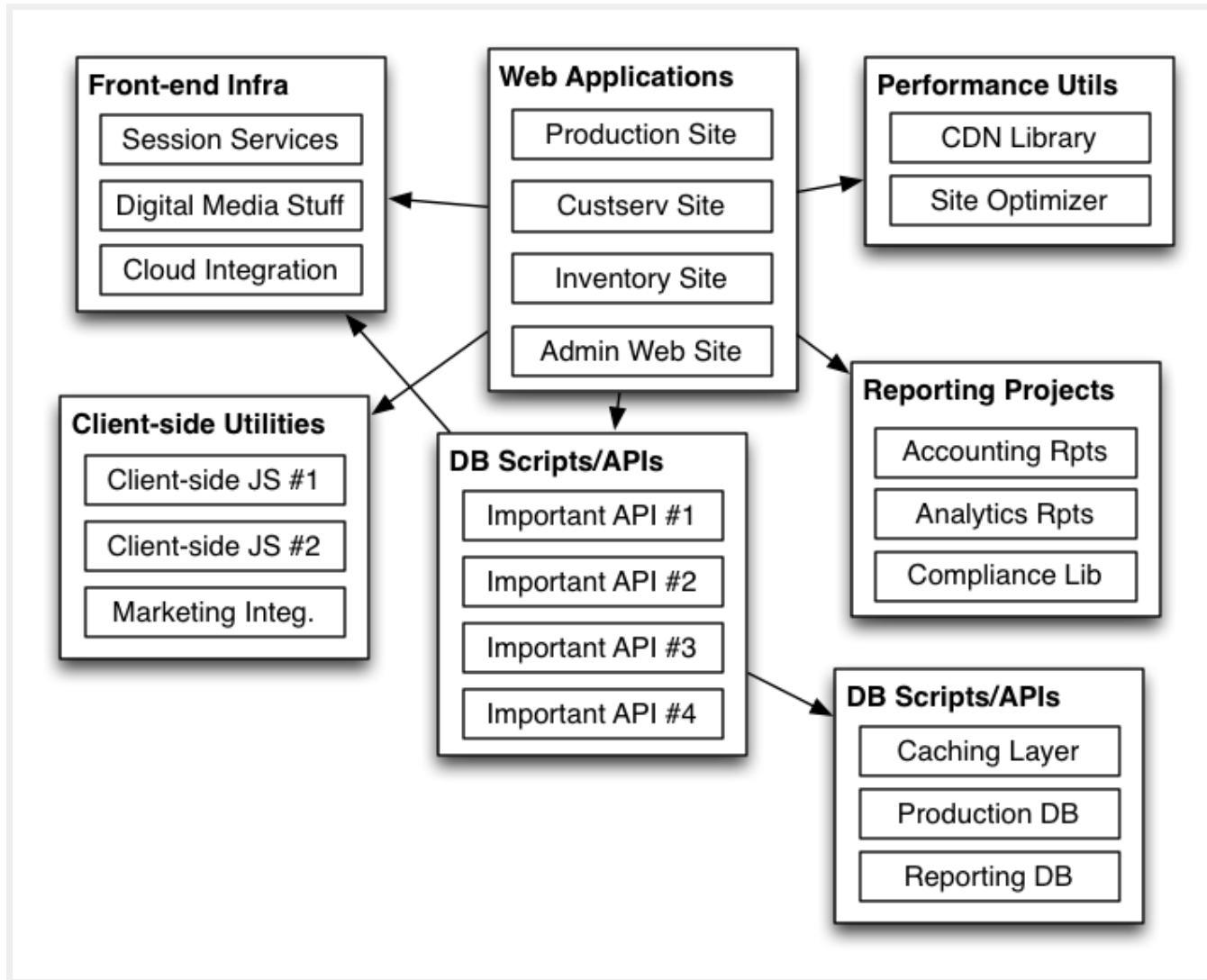


Creating Modules as Needed

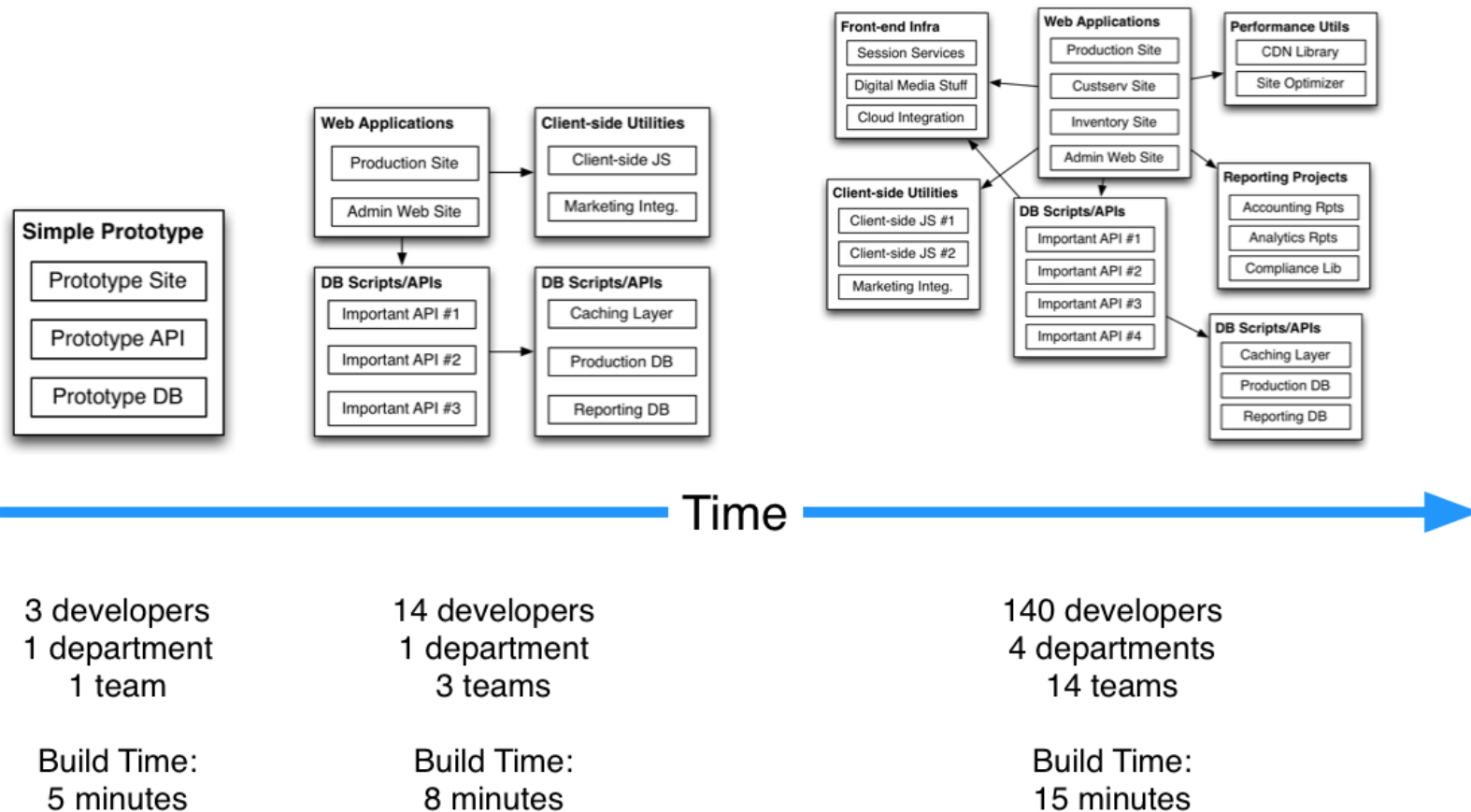
Your proof of concept got traction
and your project got staff and resources.



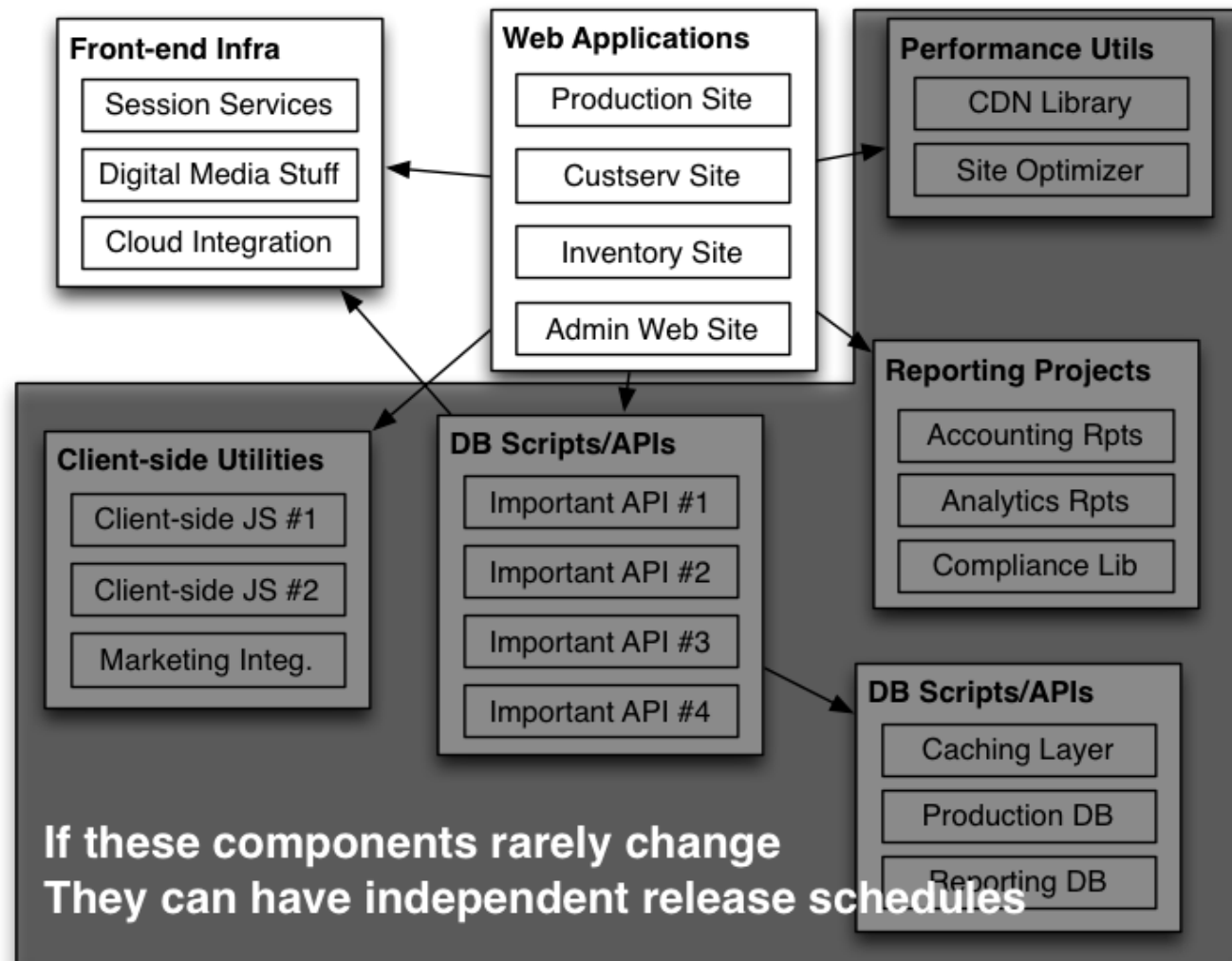
Application Development @ Scale



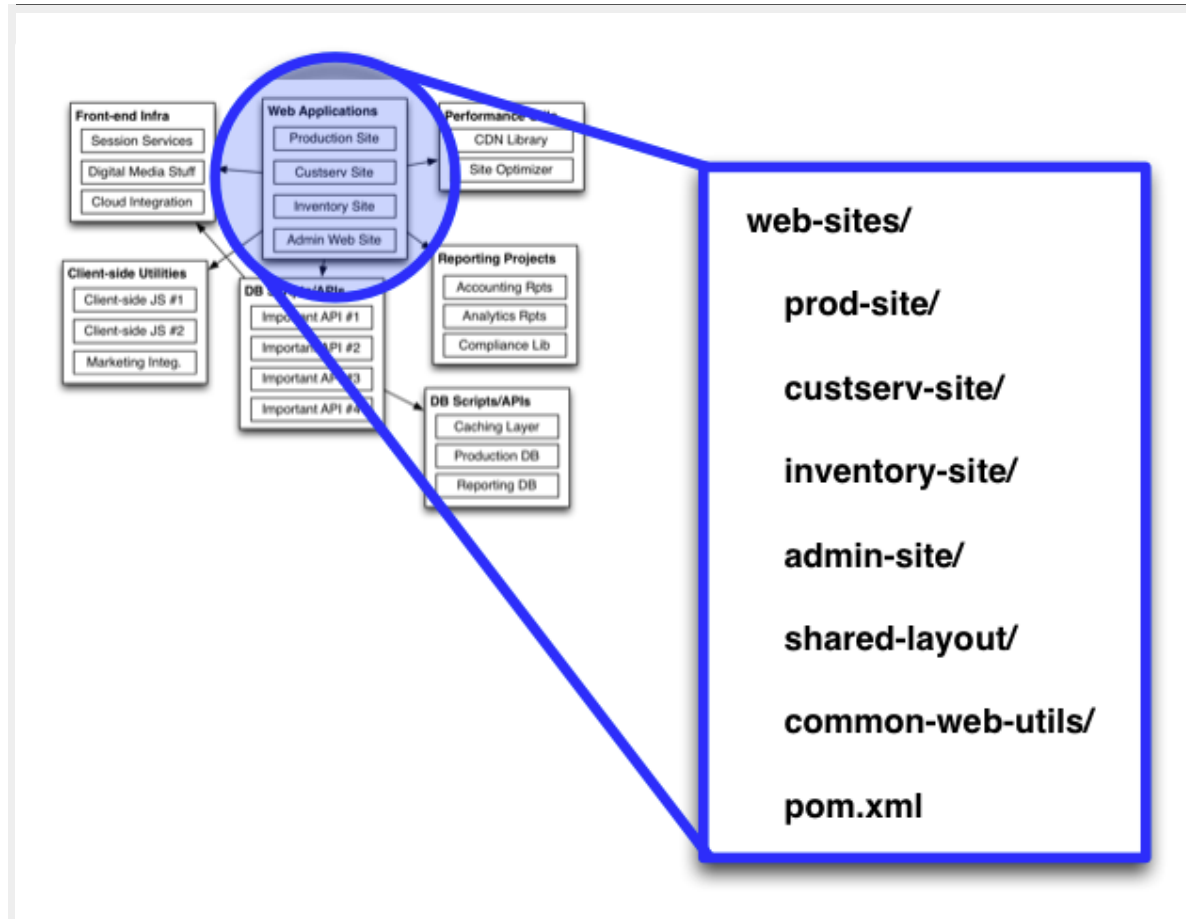
Application Development @ Scale



Faster, More Focused Releases



Each Group has Structure



Demo: Structure of a Multi-module Project

Two Concepts: Aggregation and Inheritance

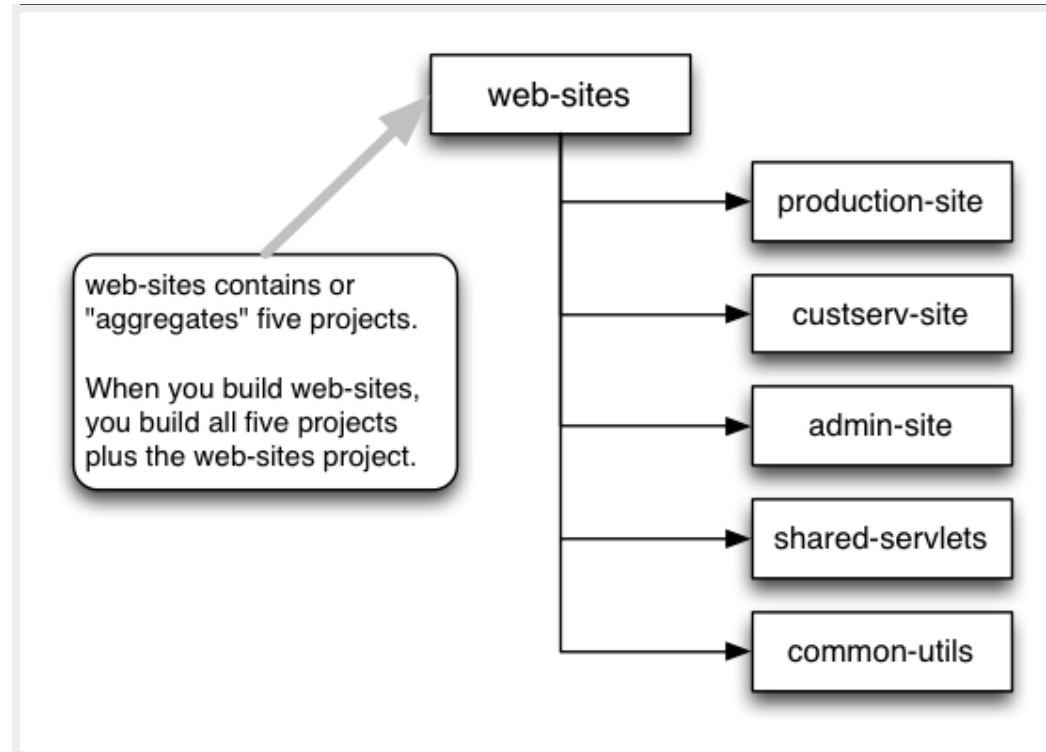
- **Aggregation:** A project contains modules
- **Inheritance:** A project Declares a "parent" and inherits POM configuration

Aggregation

When you build a multi-module project you are building an **aggregation** of several independent projects.

Here's a simple diagram...

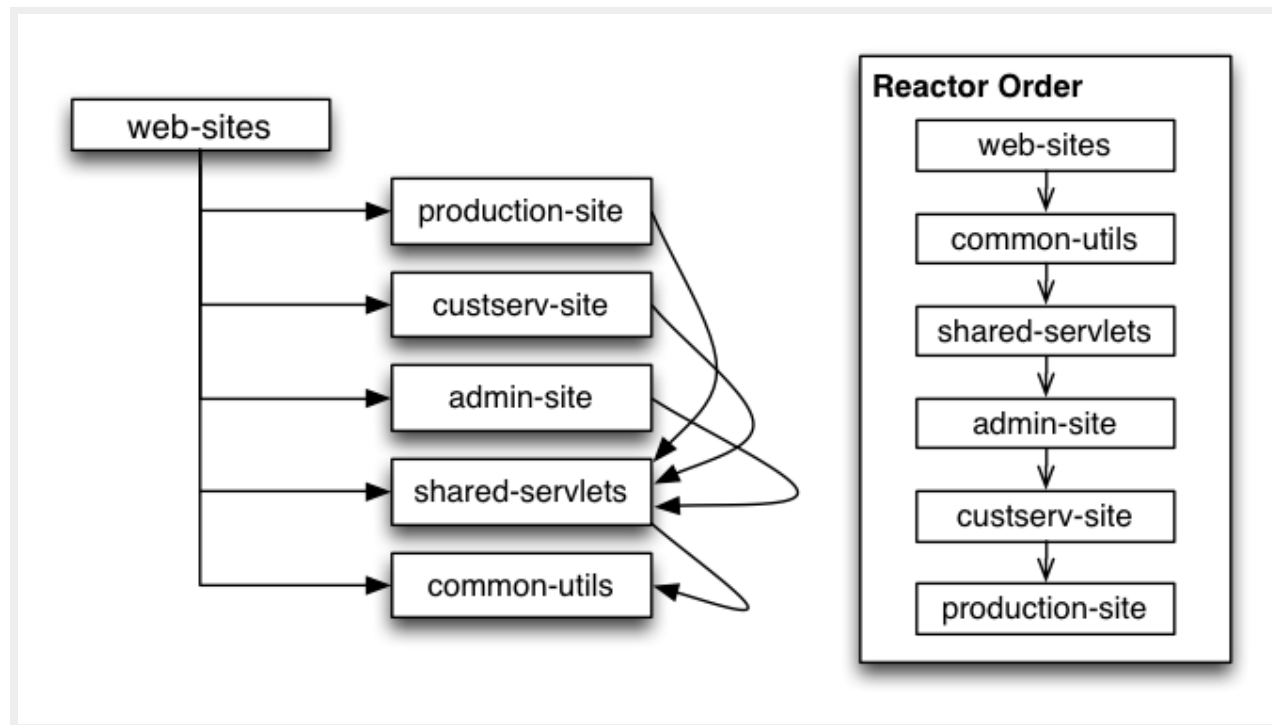
Aggregation



Demo: Building a Multi-module Project

Interdependent Projects

Internal dependencies between modules dictate build order.

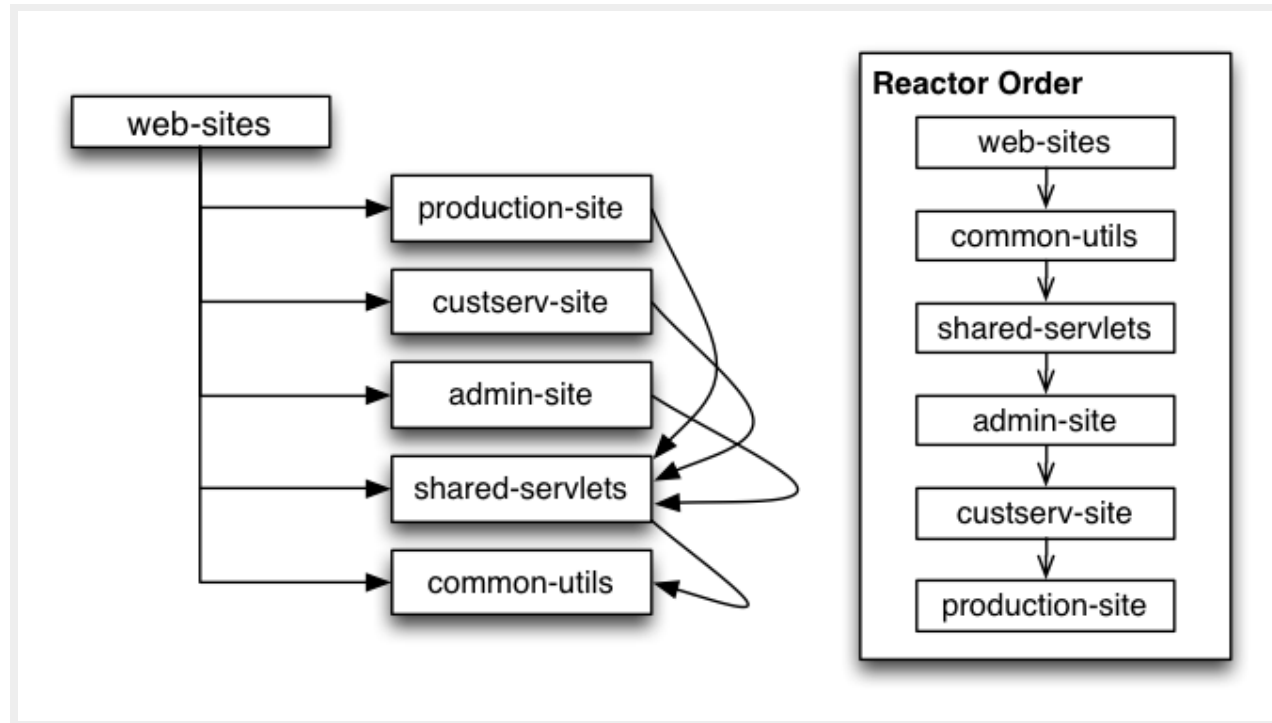


Maven has a Reactor?

The reactor orders builds in a multi-module build.

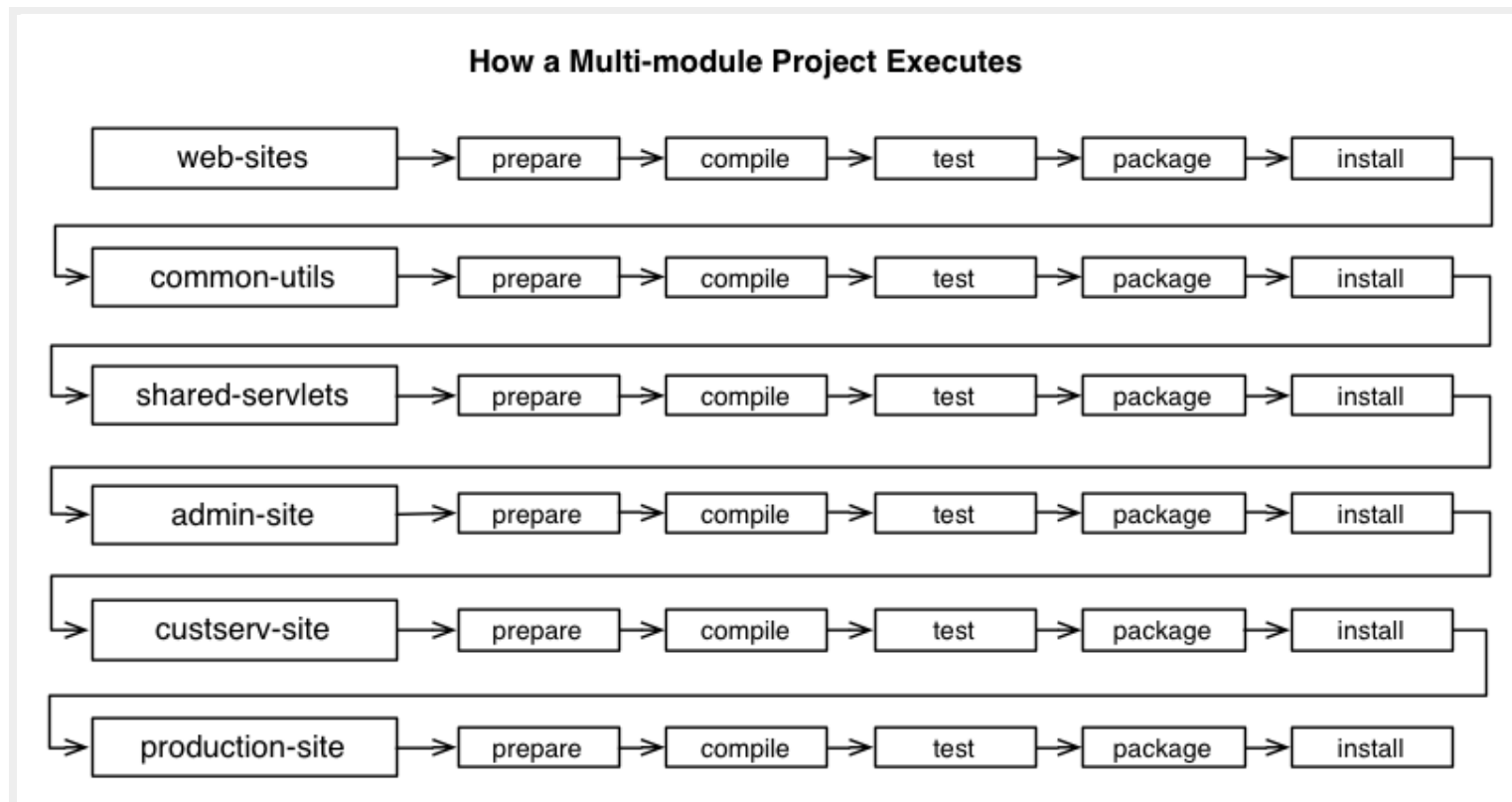
- **Parses** all POMs in a multi-module build
- **Constructs** a model of all dependencies
- **Builds** all project in the correct order

The Reactor



How a Multi-Module Build Works

The lifecycle is executed for each project.



How a Multi-Module Build Doesn't Work

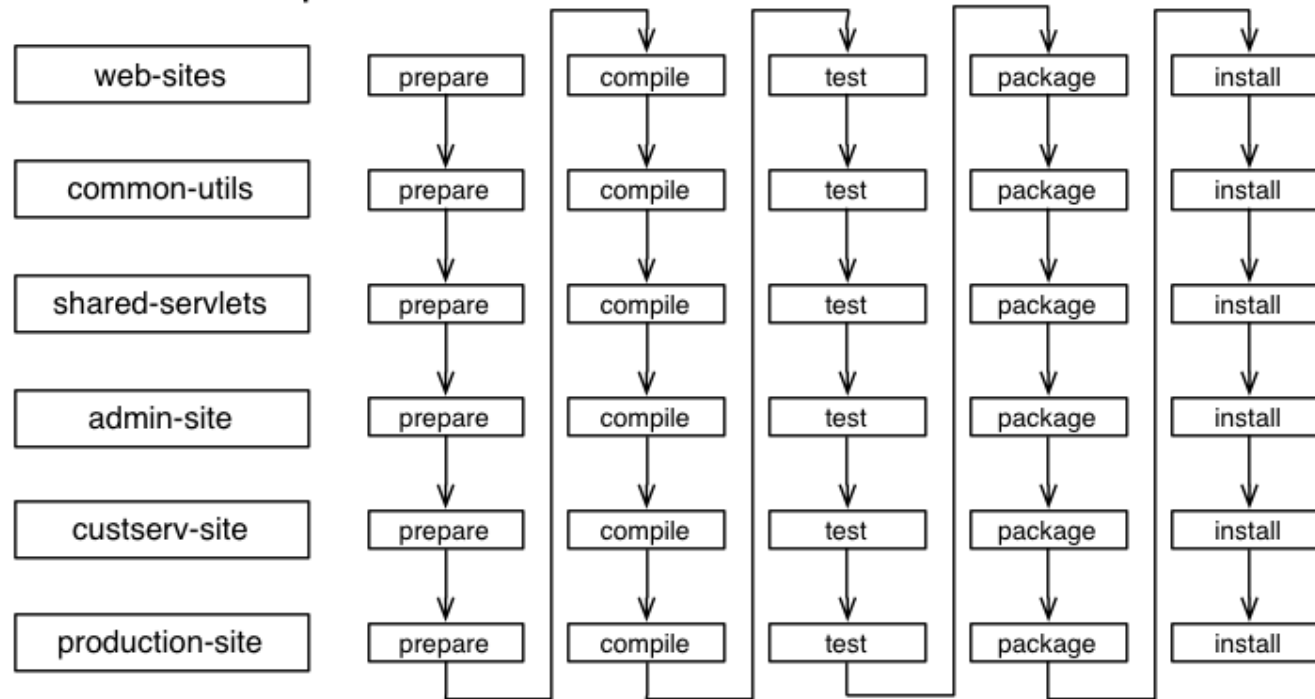
If you come to Maven from Ant...

- You may expect the build to run in "stages"
- All compilation completes across all projects
- ..then all testing completes across all projects.

This is a common misconception.

How a Multi-Module Build Doesn't Work

A Common Misconception



No. It doesn't work like this.

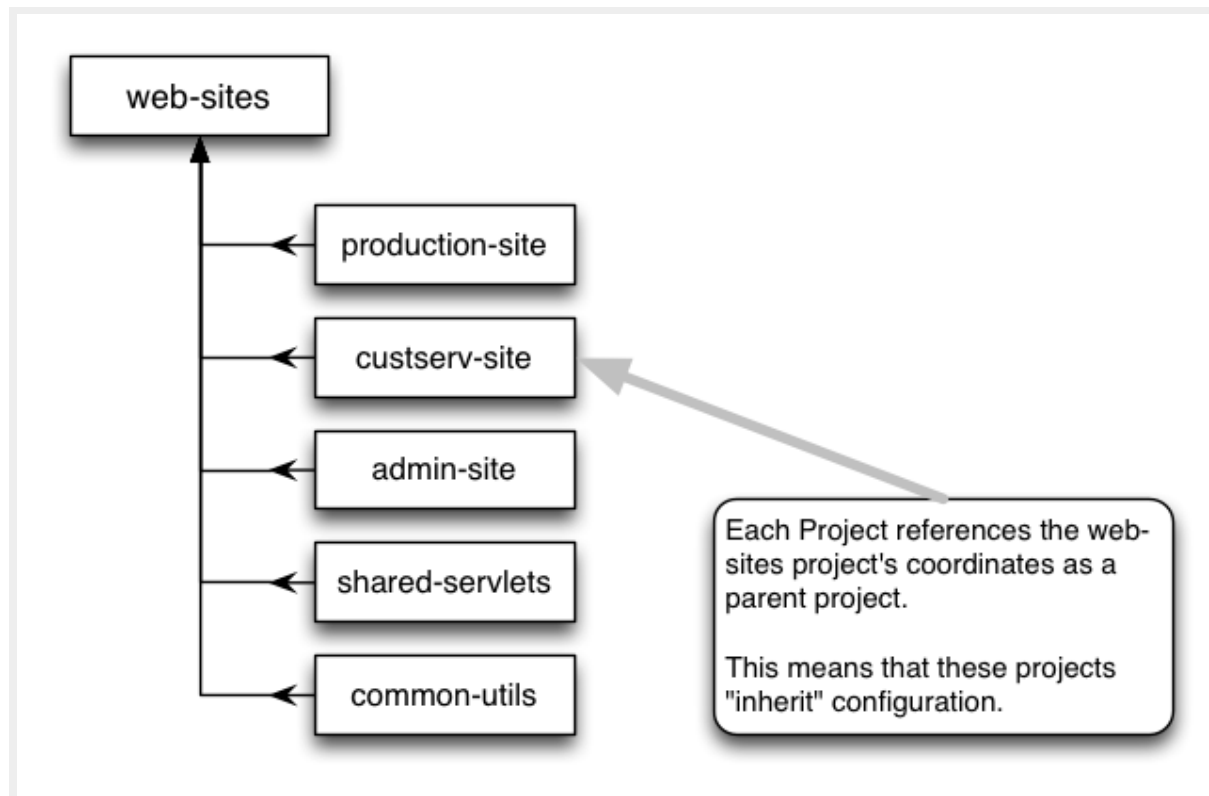
Demo: Reactor Options

In particularly large builds...

- ...sometimes you want to skip through projects.
- ...sometimes you want to focus on a single component.

Inheritance

Projects can declare a parent.



Demo: Inherited Dependencies

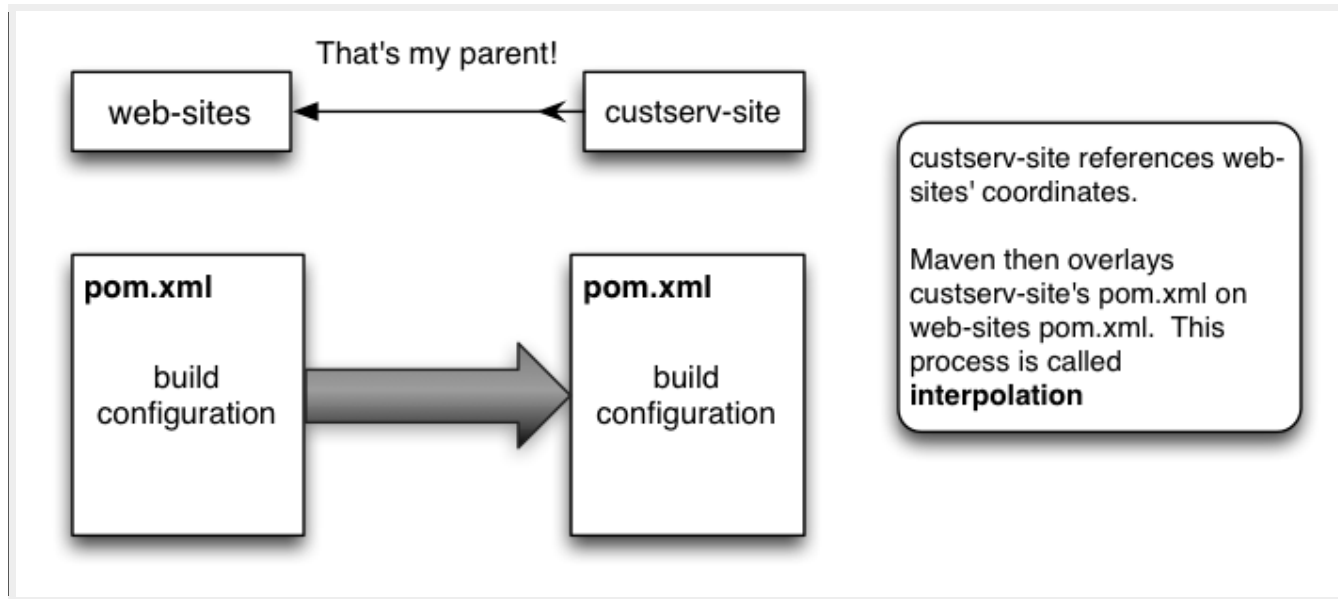
...for our next trick.

How is this possible?

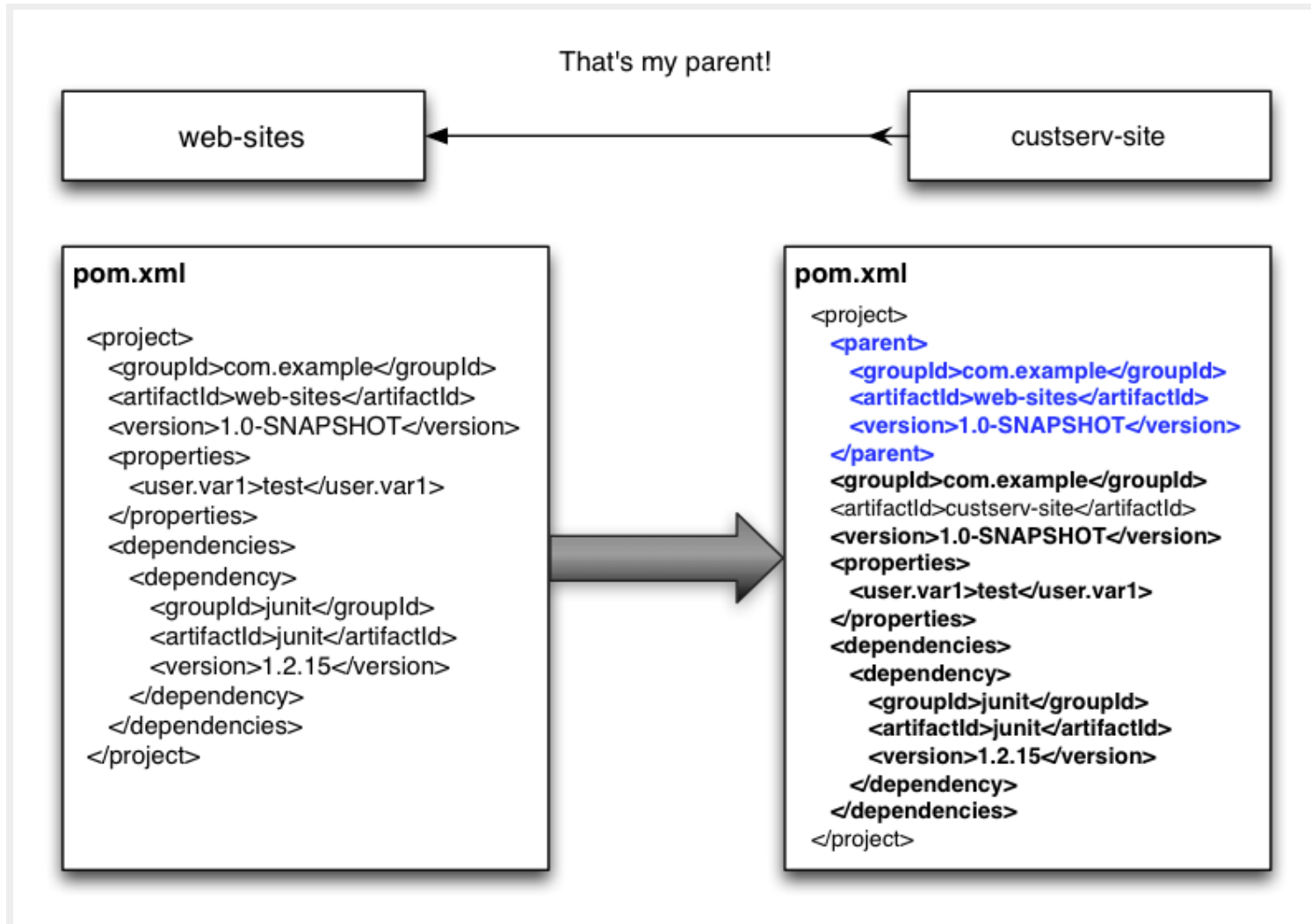
The parent element.

- Tells Maven to "Interpolate" POMs
- POMs are merged together.
- Overlaid to create an effective POM.

Declaring a Parent



Inheriting Build Configuration

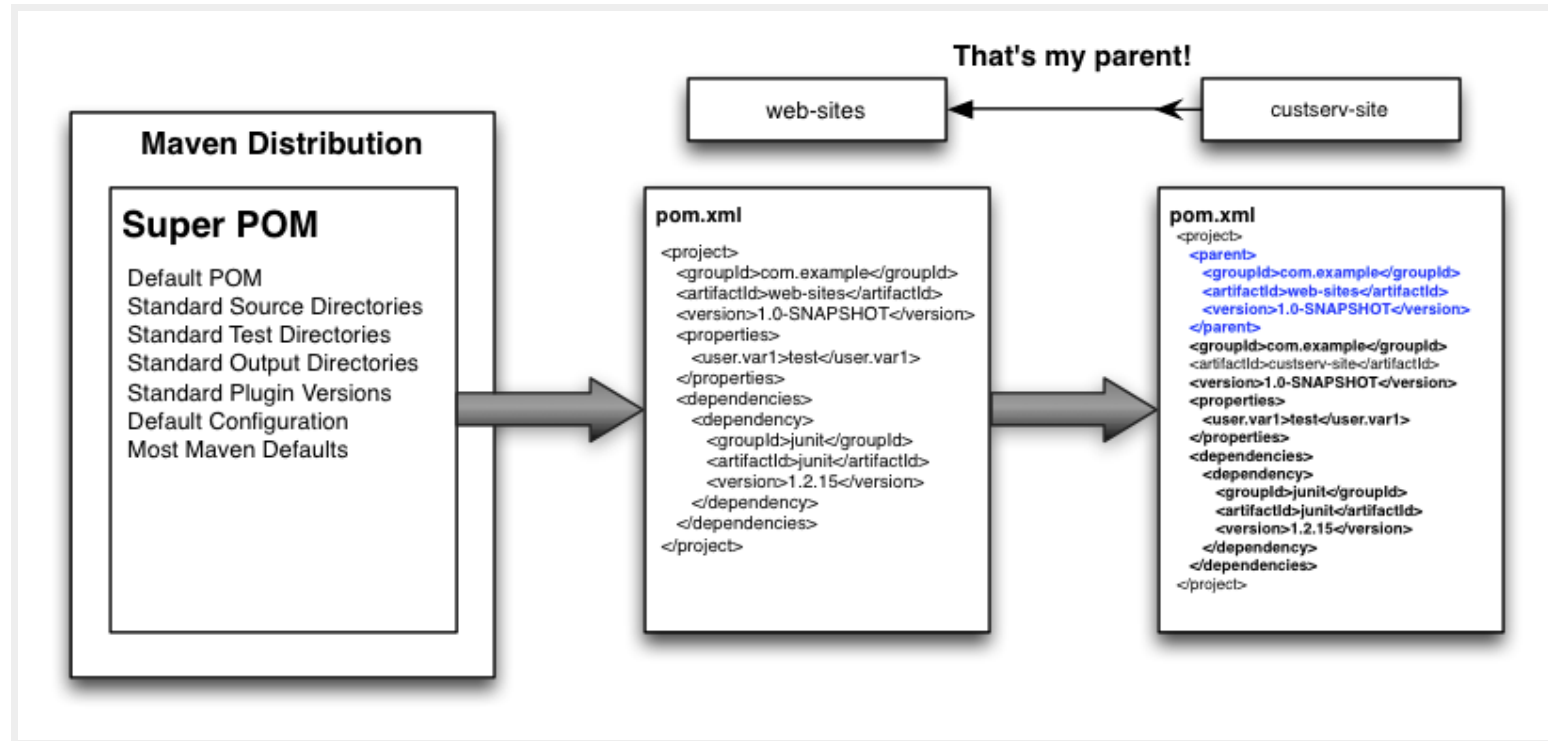


Inheritance at Every Level

With Inheritance...

- ...you define build-wide configuration.
- ...you can group similar projects together.
- ...you gain Maven's defaults.

Every POM Inherits the SuperPOM



Demo: Common Compiler Configuration

A very common use case is to define compiler plugin configuration in a top-level POM.

Demo: Version and Group Inheritance

Version numbers are often the same across a large multi-module project.

Group Identifiers are also often the same in a multi-module project.

Publishing Packages

- Think of the Central Repository: How did those JARs get there?
- Answer: Maven. Maven packages can be published.
- Note: Outside the scope of this class.

Public and Private Repos

- You can proxy the Central Repository with a Repository Manager.
- You can also publish your own artifacts to a private repository.
- Bringing the ease and accessibility of the Central Repository in-house.
- And you can publish your open source project to the Central Repository

Releasing Software

Maven has a standard for a transition from development to release.

- When software is released it is a point-in-time release
- A software release is an immutable artifact
- Releases are published to repositories

Releases vs. Snapshots

A critical distinction: two types of versions.

- **Snapshots** - Snapshot versions are for development. They are constantly changing, and Maven builds a date and timestamp into the versions of SNAPSHOTs deployed to a repository.
- **Releases** - Releases are point-in-time builds of a particular artifact. Once published they should never change.

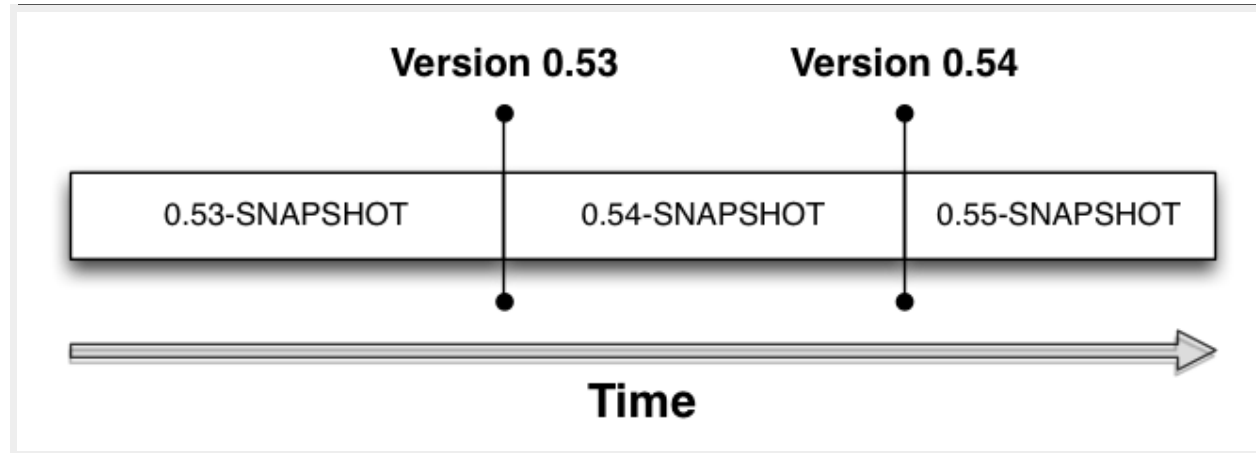
What is a SNAPSHOT?

- Snapshots are timestamped builds for artifacts still in development
- You develop with snapshots
- Snapshots are constantly changing artifacts

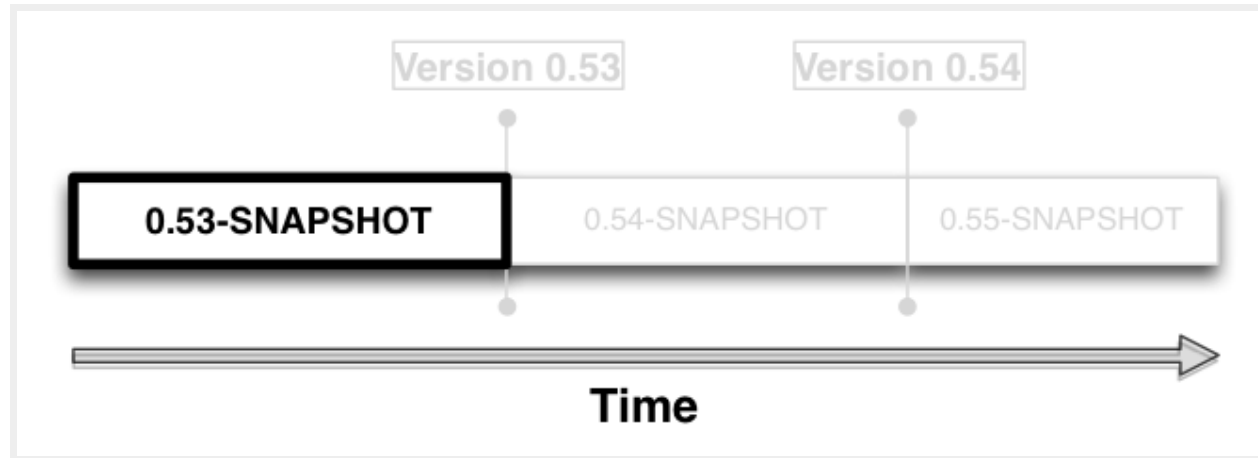
What is a Release?

- Releases are point-in-time versions
- They never change. For example commons-lang 1.1 will never change.
- You download releases from the Central Repository
- .. or other release repositories.

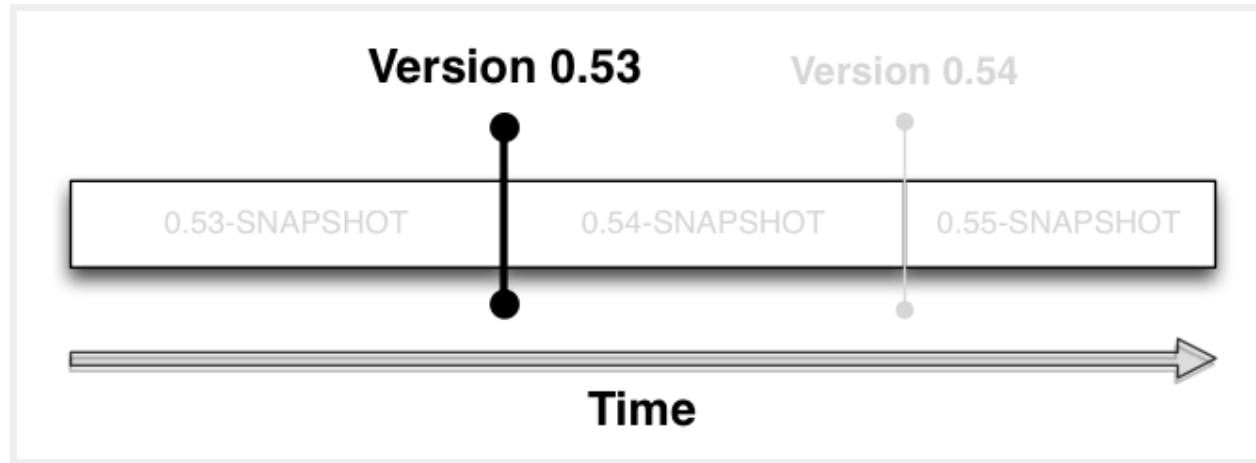
Illustrating Snapshots and Releases



Snapshot are for Development



Releases are Point in Time Events

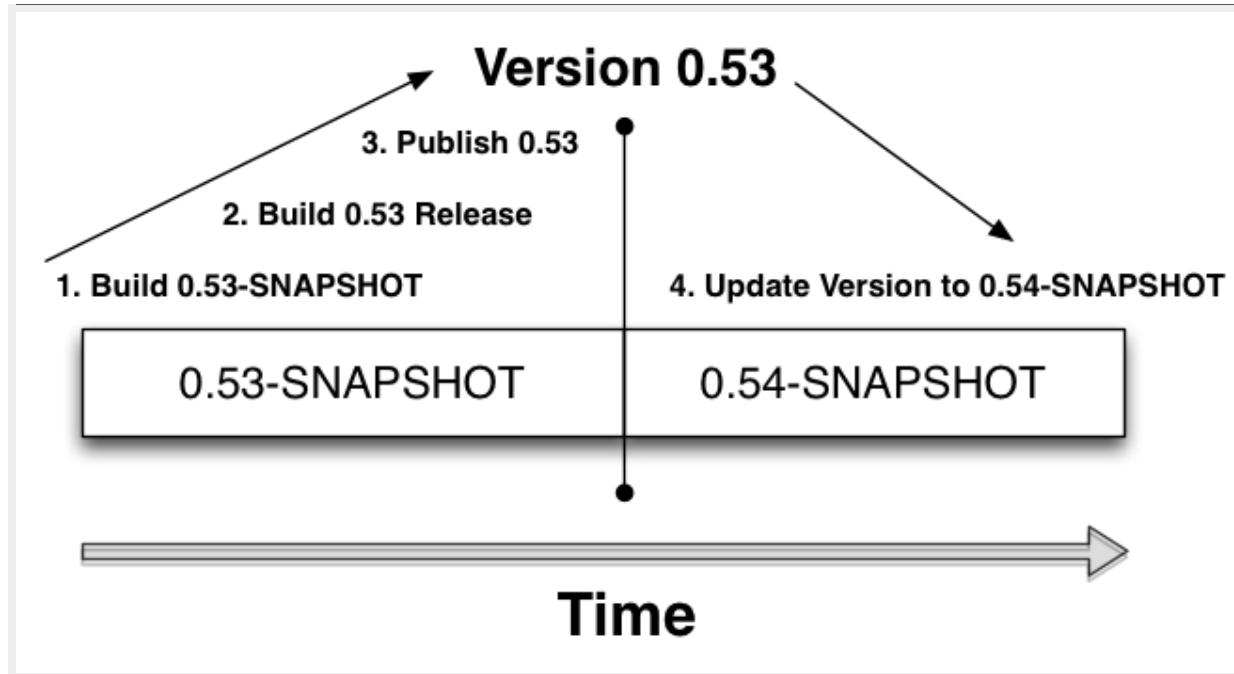


Preparing for Release

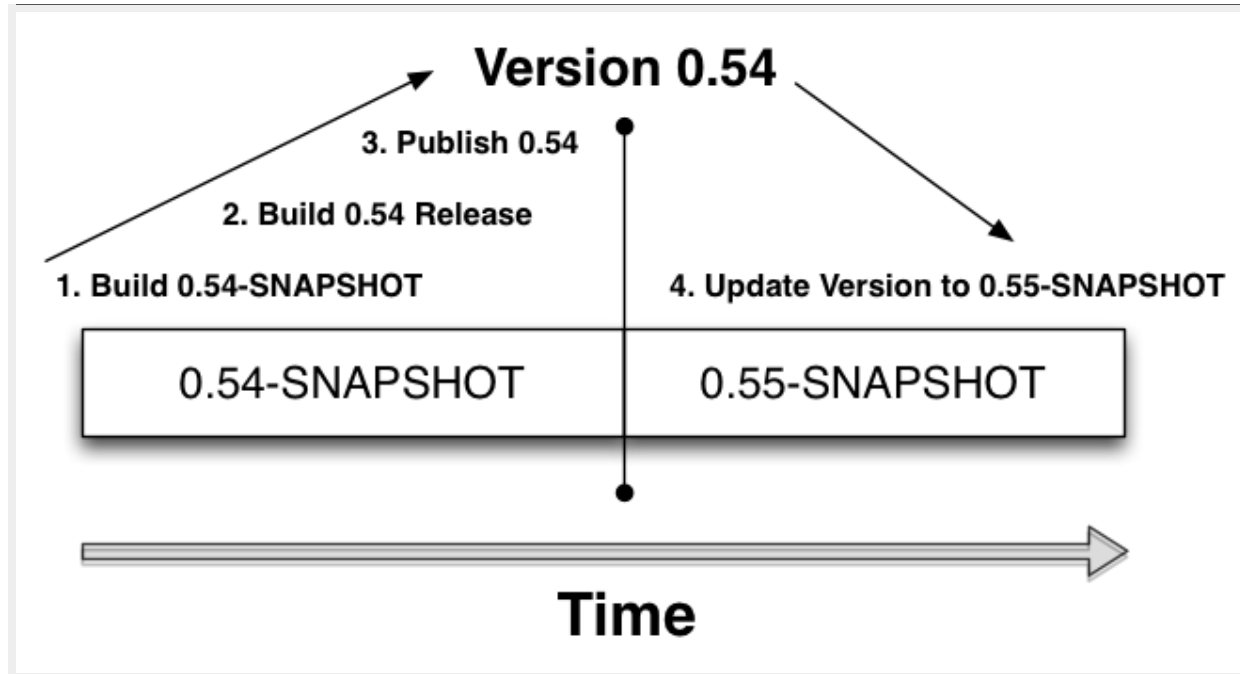
When you need to release a project you perform the following tasks:

- Update project version numbers to a release version
- Run a build and generate output artifacts
- Publish artifacts to a binary repository manager
- Increment version numbers to next snapshot

Example Release Pattern



The Next Release



Demo: Perform a Release

- Let's see exactly what happens between SNAPSHOTS and Releases.

Demo: Perform a Release

- Add distributionMgt for a localhost Nexus to multi module project.
- Demonstrate a manual release.
- Discuss the general process for Releases.
- In Advanced Maven you'll learn about the Release plugin.

Course Summary

