Welcome

What is Buildr?

Buildr is a build system for Java applications. We wanted something that's simple and intuitive to use, so we only need to tell it what to do, and it takes care of the rest. But also something we can easily extend for those one-off tasks, with a language that's a joy to use. And of course, we wanted it to be fast, reliable and have outstanding dependency management.

Here's what we got:

- A simple way to specify projects, and build large projects out of smaller sub-projects.
- Pre-canned tasks that require the least amount of configuration, keeping the build script DRY and simple.
- Compiling, copying and filtering resources, JUnit/TestNG test cases, APT source code generation, Javadoc and more.
- A dependency mechanism that only builds what has changed since the last release.
- A drop-in replacement for Maven 2.0, Buildr uses the same file layout, artifact specifications, local and remote repositories.
- All your Ant tasks belong to us! Anything you can do with Ant, you can do with Buildr.
- No overhead for building "plugins" or configuration. Just write new tasks or functions.
- Buildr is Ruby all the way down. No one-off task is too demanding when you write code using variables, functions and objects.
- Simple way to upgrade to new versions.

- Did we mention fast?

So let's get started. You can read the documentation online, or download the PDF.

News

Check out all that's new in Buildr 1.3.3.

- Buildr 1.3 now runs on JRuby 1.1 and Ruby 1.8.6.
- Support for building Scala and Groovy projects.
- Behavior-Driven Development frameworks (RSpec, JBehave, etc).
- Profiles and build.yml settings file.
- New API for accessing Java libraries.
- More documentation.
- Other features and bug fixes.

Notices

The Apache Software Foundation is a non-profit organization, consider sponsoring and check the thanks page.

Apache Buildr is an effort undergoing incubation at The Apache Software Foundation (ASF), sponsored by the Apache Incubator. Incubation is required of all newly accepted projects until a further review indicates that the infrastructure, communications, and decision making process have stabilized in a manner consistent with other successful ASF projects. While incubation status is not necessarily a reflection of the completeness or stability of the code, it does indicate that the project has yet to be fully endorsed by the ASF.

Getting Started

Installing Buildr

The installation instructions are slightly different for each operating system. Pick the one that best matches your operating system and target platform.

The `gem install` and `gem update` commands install Buildr from a binary distribution provided through RubyForge. This distribution is maintained by contributors to this project, but is **not** an official Apache distribution. You can obtain the official Apache distribution files from the download page.

The current release of Buildr for Ruby may not work well with Java 6, only Java 1.5 or earlier. If you need to use Java 6, consider Buildr for JRuby.

Linux

To get started you will need a recent version of Ruby, Ruby Gems and build tools for compiling native libraries (`make`, `gcc` and standard headers).

On **RedHat/Fedora** you can use yum to install Ruby and RubyGems, and then upgrade to the most recent version of RubyGems:

```
$ sudo yum install ruby rubygems ruby-devel gcc
$ sudo gem update --system
```

On **Ubuntu** you have to install several packages:

```
$ sudo apt-get install ruby-full ruby1.8-dev libopenssl-ruby
build-essential
```

The Debian package for `rubygems` will not allow you to install Buildr, so you need to install RubyGems from source:

```
$ wget http://rubyforge.org/frs/download.php/38646/rubygems-1.2.0.tgz
$ tar xzf rubygems-1.2.0.tgz
$ cd rubygems-1.2.0
$ sudo ruby setup.rb
$ sudo ln -s /usr/bin/gem1.8 /usr/bin/gem
```

Before installing Buildr, please set the `JAVA_HOME` environment variable to point to your JDK distribution. Next, use Ruby Gem to install Buildr:

```
$ sudo env JAVA_HOME=$JAVA_HOME gem install buildr
```

To upgrade to a new version or install a specific version:

```
$ sudo env JAVA_HOME=$JAVA_HOME gem update buildr
$ sudo env JAVA_HOME=$JAVA_HOME gem install buildr -v 1.3.3
```

You can also use this script to install Buildr on Linux. This script will install Buildr or if already installed, upgrade to a more recent version. It will also install Ruby 1.8.6 if not already installed (using `yum` or `apt-get`) and upgrage RubyGems to 1.0.1.

OS X

OS X 10.5 (Leopard) comes with a recent version of Ruby 1.8.6. OS X 10.4 (Tiger) includes an older version of Ruby, we recommend you first install Ruby 1.8.6 using MacPorts (`sudo port install ruby rb-rubygems`), Fink or the Ruby One-Click Installer for OS X.

We recommend you first upgrade to the latest version of Ruby gems:

```
$ sudo gem update --system
```

Before installing Buildr, please set the `JAVA_HOME` environment variable to point to your JDK distribution:

```
$ export JAVA_HOME=/Library/Java/Home
```

To install Buildr:

```
$ sudo env JAVA_HOME=$JAVA_HOME gem install buildr
```

To upgrade to a new version or install a specific version:

```
$ sudo env JAVA_HOME=$JAVA_HOME gem update buildr
$ sudo env JAVA_HOME=$JAVA_HOME gem install buildr -v 1.3.3
```

You can also use this script to install Buildr on OS X. This script will install Buildr or if already installed, upgrade to a more recent version. It will also install Ruby 1.8.6 if not already installed (using MacPorts) and upgrage RubyGems to 1.0.1.

Windows

If you don't already have Ruby installed, now is the time to do it. The easiest way to install Ruby is using the one-click installer.

We recommend you first upgrade to the latest version of Ruby gems:

```
> gem update --system
```

Before installing Buildr, please set the `JAVA_HOME` environment variable to point to your JDK distribution. Next, use Ruby Gem to install Buildr:

```
> gem install buildr
```

Buildr uses several libraries that include native extensions. During installation it will ask you to pick a platform for these libraries. By selecting `mswin32` it will download and install pre-compiled DLLs for these extensions.

To upgrade to a new version or install a specific version:

```
> gem update buildr
> gem install buildr -v 1.3.3
```

JRuby

If you don't already have JRuby 1.1 or later installed, you can download it from the JRuby site.

After uncompressing JRuby, update your `PATH` to include both `java` and `jruby` executables.

For Linux and OS X:

```
$ export PATH=$PATH:[path to JRuby]/bin:$JAVA_HOME/bin
$ jruby -S gem install buildr
```

For Windows:

```
> set PATH=%PATH%;[path to JRuby]/bin;%JAVA_HOME%/bin
> jruby -S gem install buildr
```

To upgrade to a new version or install a specific version:

```
$ jruby -S gem update buildr
$ jruby -S gem install buildr -v 1.3.3
```

You can also use this script to install Buildr on JRuby. This script will install Buildr or if already installed, upgrade to a more recent version. If necessary, it will also install JRuby 1.1 in `/opt/jruby` and update the `PATH` variable in `~/.bash_profile` or `~/.profile`.

Important: Running JRuby and Ruby side by side

Ruby and JRuby maintain separate Gem repositories, and in fact install slightly different versions of the Buildr Gem (same functionality, different dependencies). Installing Buildr for Ruby does not install it for JRuby and vice versa.

If you have JRuby installed but not Ruby, the `gem` and `buildr` commands will use JRuby. If you have both JRuby and Ruby installed, follow the instructions below. To find out if you have Ruby installed (some operating systems include it by default), run `ruby --version` from the command line.

To work exclusively with JRuby, make sure it shows first on the path, for example, by setting `PATH=/opt/jruby/bin:$PATH`.

You can use JRuby and Ruby side by side, by running scripts with the `-S` command line argument. For example:

```
$ # with Ruby
$ ruby -S gem install buildr
$ ruby -S buildr
$ # with JRuby
$ jruby -S gem install buildr
$ jruby -S buildr
```

Run `buildr --version` from the command line to find which version of Buildr you are using by default. If you see `(JRuby ...)`, Buildr is running on that version of JRuby.

Document Conventions

Lines that start with `$` are command lines, for example:

```
$ # Run Buildr
$ buildr
```

Lines that start with `=>` show output from the console or the result of a method, for example:

```
puts 'Hello world'
=> "Hello world"
```

And as you guessed, everything else is Buildfile Ruby or Java code. You can figure out which language is which.

Running Buildr

You need a **Buildfile**, a build script that tells Buildr all about the projects it's building, what they contain, what to produce, and so on. The Buildfile resides in the root directory of your project. We'll talk more about it in the next chapter. If you don't already have one, ask Buildr to create it:

```
$ buildr
```

You'll notice that Buildr creates a file called `buildfile`. It's case sensitive, but Buildr will look for either `buildfile` or `Buildfile`.

You use Buildr by running the `buildr` command:

```
$ buildr [options] [tasks] [name=value]
```

There are several options you can use, for a full list of options type:

```
$ buildr --help
```

Option	Usage
`-f/--buildfile [file]`	Specify the buildfile.
`-e/--environment [name]`	Environment name (e.g. development, test, production).
`-h/--help`	Display this help message.
`-n/--nosearch`	Do not search parent directories for the buildfile.
`-q/--quiet`	Do not log messages to standard output.
`-r/--require [file]`	Require MODULE before executing buildfile.
`-t/--trace`	Turn on invoke/execute tracing, enable full backtrace.
`-v/--version`	Display the program version.
`-P/--prereqs`	Display tasks and dependencies, then exit.

You can tell Buildr to run specific tasks and the order to run them. For example:

```
# Clean and rebuild
buildr clean build
# Package and install
buildr install
```

If you don't specify a task, Buildr will run the `build` task, compiling source code and running test cases. Running a task may run other tasks as well, for example, running the `install` task will also run `package`.

There are several environment variables that let you control how Buildr works, for example, to skip test cases during a build, or specify options for the JVM. Depending on the variable, you may want to set it once in your environment, or set a different value each time you run Buildr.

For example:

```
$ export JAVA_OPTS='-Xms1g -Xmx1g'
$ buildr TEST=no
```

Help Tasks

Buildr includes a number of informative tasks. Currently that number stands at two, but we'll be adding more tasks in future releases. These tasks report information from the Buildfile, so you need one to run them. For more general help (version number, command line arguments, etc) use `buildr --help`.

To start with, type:

```
$ buildr help
```

You can list the name and description of all your projects using the `help:projects` task. For example:

```
$ buildr help:projects
killer-app                # Code. Build. ??? Profit!
killer-app:teh-api        # Abstract classes and interfaces
killer-app:teh-impl       # All those implementation details
killer-app:la-web         # What our users see
```

You are, of course, describing your projects for the sake of those who will maintain your code, right? To describe a project, or a task, call the `desc` method before the project or task definition.

So next let's talk about projects.

Learning More

Ruby It pays to pick up Ruby as a second (or first) programming language. It's fun, powerful and slightly addictive. If you're interested in learning Ruby the language, a good place to start is Programming Ruby: The Pragmatic Programmer's Guide, fondly known as the *Pickaxe book*.

For a quicker read (and much more humor), Why's (Poignant) Guide to Ruby is available online. More resources are listed on the ruby-lang web site.

Rake Buildr is based on Rake, a Ruby build system that handles tasks and dependencies. Check out the Rake documentation for more information.

AntWrap Buildr uses AntWrap, for configuring and running Ant tasks. You can learn more from the Antwrap documentation.

YAML Buildr uses YAML for its profiles. You can learn more about YAML here, and use this handy YAML quick reference.

Projects

Starting Out

In Java, most projects are built the same way: compile source code, run test cases, package the code, release it. Rinse, repeat.

Feed it a project definition, and Buildr will set up all these tasks for you. The only thing you need to do is specify the parts that are specific to your project, like the classpath dependencies, whether you're packaging a JAR or a WAR, etc.

The remainder of this guide deals with what it takes to build a project. But first, let's pick up a sample project to work with. We'll call it *killer-app*:

```
require 'buildr'

VERSION_NUMBER = '1.0'

AXIS2 = 'org.apache.axis2:axis2:jar:1.2'
AXIOM  = group('axiom-api', 'axiom-impl', 'axiom-dom',
  :under=>'org.apache.ws.commons.axiom', :version=>'1.2.4')
AXIS_OF_WS = [AXIOM, AXIS2]
OPENJPA = ['org.apache.openjpa:openjpa-all:jar:0.9.7',
  'net.sourceforge.serp:serp:jar:1.12.0']

repositories.remote << 'http://www.ibiblio.org/maven2/'

desc 'Code. Build. ??? Profit!'
define 'killer-app' do

  project.version = VERSION_NUMBER
  project.group = 'acme'
  manifest['Copyright'] = 'Acme Inc (C) 2007'
  compile.options.target = '1.5'

  desc 'Abstract classes and interfaces'
  define 'teh-api' do
    package :jar
  end

  desc 'All those implementation details'
  define 'teh-impl' do
    compile.with AXIS_OF_WS, OPENJPA
    compile { open_jpa_enhance }
    package :jar
  end

  desc 'What our users see'
  define 'la-web' do
    test.using AXIS_OF_WS
    package(:war).with :libs=>projects('teh-api', 'teh-impl')
  end

  javadoc projects
  package :javadoc

end
```

A project definition requires four pieces of information: the project name, group identifier, version number and base directory. The project name … do we need to explain why its necessary? The group identifier and version number are used for packaging and deployment, we'll talk more about that in the Packaging section. The base directory lets you find files inside the project.

Everything else depends on what that particular project is building. And it all goes inside the project definition block, the piece of code that comes between `define <name> .. do` and `end`.

The Directory Structure

Buildr follows a convention we picked from years of working with Apache projects.

Java projects are laid out so the source files are in the `src/main/java` directory and compile into the `target/classes` directory. Resource files go in the `src/main/resources` directory, and copied over to `target/resources`. Likewise, tests come from `src/test/java` and `src/test/resources`, and end life in `target/test/classes` and `target/test/resources`, respectively.

WAR packages pick up additional files from the aptly named `src/main/webapp`. And most stuff, including generated source files are parked under the `target` directory. Test cases and such may generate reports in the, you guessed it, `reports` directory.

Other languages will use different directories, but following the same general conventions. For example, Scala code compiles from the `src/main/scala` directory, RSpec tests are found in the `src/test/rspec` directory, and Flash will compile to `target/flash`. Throughout this document we will show examples using mostly Java, but you can imagine how this pattern applies to other languages.

When projects grow big, you split them into smaller projects by nesting projects inside each other. Each sub-project has a sub-directory under the parent project and follows the same internal directory structure. You can, of course, change all of that to suite your needs, but if you follow these conventions, Buildr will figure all the paths for you.

Going back to the example above, the directory structure will look something like this:

Notice the `buildfile` at the top. That's your project build script, the one Buildr runs.

When you run the `buildr` command, it picks up the `buildfile` (which here we'll just call *Buildfile*) from the current directory, or if not there, from the closest parent directory. So you can run `buildr` from any directory inside your project, and it will always pick up the same Buildfile. That also happens to be the base directory for the top project. If you have any sub-projects, Buildr assumes they reflect sub-directories under their parent.

And yes, you can have two top projects in the same Buildfile. For example, you can use that to have one project that groups all the application modules (JARs, WARs, etc) and another project that groups all the distribution packages (binary, sources, javadocs).

When you start with a new project you won't see the `target` or `reports` directories. Buildr creates these when it needs to. Just know that they're there.

Naming And Finding Projects

Each project has a given name, the first argument you pass when calling `define`. The project name is just a string, but we advise to stay clear of colon (`:`) and slashes (`/` and `\`), which could conflict with other task and file names. Also, avoid using common Buildr task names, don't pick `compile` or `build` for your project name.

Since each project knows its parent project, child projects and siblings, you can reference them from within the project using just the given name. In other cases, you'll need to use the full name. The full name is just `parent:child`. So if you wanted to refer to *teh-impl*, you could do so with either `project('killer-app:teh-impl')` or `project('killer-app').project('teh-impl')`.

The `project` method is convenient when you have a dependency from one project to another, e.g. using the other project in the classpath, or accessing one of its source files. Call it with a project name and it will return that object or raise an error. You can also call it with no arguments and it will return the project itself. It's syntactic sugar that's useful when accessing project properties.

The `projects` method takes a list of names and returns a list of projects. If you call it with no arguments on a project, it returns all its sub-projects. If you call it with no argument in any other context, it returns all the projects defined so far.

Let's illustrate this with a few examples:

```
puts projects.inspect
=> [project("killer-app"), project("killer-app:teh-api") ... ]

puts project('killer-app').projects.inspect
=> [project("killer-app:teh-api"), project("killer-app:teh-impl") ...
]

puts project('teh-api')
=> No such project teh-api

puts project('killer-app:teh-api').inspect
=> project("killer-app:teh-api")

puts project('killer-app').project('teh-api').inspect
=> project("killer-app:teh-api")
```

To see a list of all projects defined in your Buildfile:

```
$ buildr help:projects
```

Running Project Tasks

Most times, you run tasks like `build` or `package` that operate on the current project and recursively on its sub-projects. The "current project" is the one that uses the current working directory. So if you're in the `la-web/src` directory looking at source files, *la-web* is the current project. For example:

```
# build killer-app and all its sub-projects
$ buildr build

# switch to and test only teh-impl
$ cd teh-impl
$ buildr test

# switch to and package only la-web
$ cd ../la-web
$ buildr package
```

You can use the project's full name to invoke one of its tasks directly, and it doesn't matter which directory you're in. For example:

```
# build killer-app and all its sub-projects
$ buildr killer-app:build

# test only teh-impl
$ buildr killer-app:teh-impl:test

# package only la-web
$ buildr killer-app:la-web:package
```

Buildr provides the following tasks that you can run on the current project, or on a specific project by prefixing them with the project's full name:

```
clean     # Clean files generated during a build
compile   # Compile all projects
build     # Build the project
upload    # Upload packages created by the project
install   # Install packages created by the project
javadoc   # Create the Javadocs for this project
package   # Create packages
test      # Run all test cases
uninstall # Remove previously installed packages
```

To see a list of all the tasks provided by Buildr:

```
$ buildr help:tasks
```

Setting Project Properties

We mentioned the group identifier, version number and base directory. These are project properties. There are a few more properties we'll cover later on.

There are two ways to set project properties. You can pass them as a hash when you call `define`, or use accessors to set them on the project directly. For example:

```
define 'silly', :version=>'1.0' do
  project.group = 'acme'
end

puts project('silly').version
=> 1.0
puts project('silly').group
=> acme
```

Project properties are inherited. You can specify them once in the parent project, and they'll have the same value in all its sub-projects. In the example, we only specify the version number once, for use in all sub-projects.

Resolving Paths

You can run `buildr` from any directory in your project. To keep tasks consistent and happy, it switches over to the Buildfile directory and executes all the tasks from there, before returning back to your working directory. Your tasks can all rely on relative paths that start from the same directory as the Buildfile.

But in practice, you'll want to use the `path_to` method. This method calculates a path relative to the project, a better way if you ever need to refactor your code, say turn a ad hoc task into a function you reuse.

The `path_to` method takes an array of strings and concatenates them into a path. Absolute paths are returned as they are, relative paths are expanded relative to the project's base directory. Slashes, if you don't already know, work very well on both Windows, Linux and OS X. And as a shortcut, you can use _.

For example:

```
# Relative to the current project
path_to('src', 'main', 'java')

# Exactly the same thing
_('src/main/java')

# Relative to the teh-impl project
project('teh-impl')._('src/main/java')
```

Defining The Project

The project definition itself gives you a lot of pre-canned tasks to start with, but that's not enough to build a project. You need to specify what gets built and how, which dependencies to use, the packages you want to create and so forth. You can configure existing tasks, extend them to do additional work, and create new tasks. All that magic happens inside the project definition block.

Since the project definition executes each time you run Buildr, you don't want to perform any work directly inside the project definition block. Rather, you want to use it to specify how different build task work when you invoke them. Here's an example to illustrate the point:

```
define 'silly' do
  puts 'Running buildr'

  build do
    puts 'Building silly'
  end
end
```

Each time you run Buildr, it will execute the project definition and print out "Running buildr". We also extend the `build` task, and whenever we run it, it will print "Building silly". Incidentally, `build` is the default task, so if you run Buildr with no arguments, it will print both messages while executing the build. If you run Buildr with a different task, say `clean`, it will only print the first message.

The `define` method gathers the project definition, but does not execute it immediately. It executes the project definition the first time you reference that project, directly or indirectly, for example, by calling `project` with that project's name, or calling `projects` to return a list of all projects. Executing a project definition will also execute all its sub-projects' definitions. And, of course, all project definitions are executed once the Buildfile loads, so Buildr can determine how to execute each of the build tasks.

If this sounds a bit complex, don't worry. In reality, it does the right thing. A simple rule to remember is that each project definition is executed before you need it, lazy evaluation of sort. The reason we do that? So you can write projects that depend on each other without worrying about their order.

In our example, the *la-web* project depends on packages created by the *teh-api* and *teh-impl* projects, the later requiring *teh-api* to compile. That example is simple enough that we ended up specifying the projects in order of dependency. But you don't always want to do that. For large projects, you may want to group sub-projects by logical units, or sort them alphabetically for easier editing.

One project can reference another ahead of its definition. If Buildr detects a cyclic dependency, it will let you know.

In this example we define one project in terms of another, using the same dependencies, so we only need to specify them once:

```
define 'bar' do
  compile.with project('foo').compile.dependencies
end

define 'foo' do
  compile.with ..lots of stuff..
end
```

One last thing to remember. Actually three, but they all go hand in hand.

Self is project Each of these project definition blocks executes in the context of that project, just as if it was a method defined on the project. So when you call the `compile` method, you're essentially calling that method on the current project: `compile`, `self.compile` and `project.compile` are all the same.

Blocks are closures The project definition is also a closure, which can reference variables from enclosing scopes. You can use that, for example, to define constants, variables and even functions in your Buildfile, and reference them from your project definition. As you'll see later on, in the Artifacts section, it will save you a lot of work.

Projects are namespaces While executing the project definition, Buildr switches the namespace to the project name. If you define the task "do-this" inside the *teh-impl* project, the actual task name is "killer-app:teh-impl:do-this". Likewise, the `compile` task is actually "killer-app:teh-impl:compile".

From outside the project you can reference a task by its full name, either `task('foo:do')` or `project('foo').task('do')`. If you need to reference a task defined outside the project from within the project, prefix it with "rake:", for example, `task('rake:globally-defined')`.

Writing Your Own Tasks

Of all the features Buildr provide, it doesn't have a task for making coffee. Yet. If you need to write your own tasks, you get all the power of Rake: you can use regular tasks, file tasks, task rules and even write your own custom task classes. Check out the Rake documentation for more information.

We mentioned projects as namespaces before. When you call `task` on a project, it finds or defines the task using the project namespace. So given a project object, `task('do-this')` will return it's "do-this" task. If you lookup the source code for the `compile` method, you'll find that it's little more than a shortcut for `task('compile')`.

Another shortcut is the `file` method. When you call `file` on a project, Buildr uses the `path_to` method to expand relative paths using the project's base directory. If you call `file('src')` on *teh-impl*, it will return you a file task that points at the `teh-impl/src` directory.

In the current implementation projects are also created as tasks, although you don't invoke these tasks directly. That's the reason for not using a project name that conflicts with an existing task name. If you do that, you'll find quick enough, as the task will execute each time you run Buildr.

So now that you know everything about projects and tasks, let's go and build some code.

Building

To remove any confusion, Buildr's build task is actually called `build`. It's also the default task that executes when you run `buildr` without any task name.

The `build` task runs two other tasks: `compile` and its associated tasks (that would be, `resources`) and `test` and its associated tasks (`test:compile`, `test:setup` and friends). We'll talk about `compile` more in this section, and `test` later on. We'll also show you how to run `build` without testing, not something we recommend, but a necessary feature.

Why `build` and not `compile`? Some projects do more than just compiling. Other projects don't compile at all, but perform other build tasks, for example, creating a database schema or command line scripts. So we want you to get in the practice of running the `build` task, and help you by making it the default task.

Compiling

Each project has its own `compile` task you can invoke directly, by running `buildr compile` or as part of another build task. (Yes, that `build`).

The `compile` task looks for source files in well known directories, determines which compiler to use, and sets the target directory accordingly. For example, if it finds any Java source files in the `src/main/java` directory, it selects the Javac compiler and generates bytecode in the `target/classes` directories. If it finds Scala source files in the `src/main/scala` directory it selects the Scalac compiler, and so forth.

A single project cannot use multiple compilers at the same time, hence you may prefer creating subprojects by programming language. Some compilers like Groovy's are joint-compilers, this means they can handle several languages. When the Groovy compiler is selected for a project, .groovy and .java files are compiled by groovyc.

Most often, that's just good enough and the only change you need to make is adding compile dependencies. You can use `compile.dependencies` to get the array of dependency file tasks. For Java, each of these tasks points to a JAR or a directory containing Java classes, and the entire set of dependencies is passed to Javac as the classpath.

Buildr uses file tasks to handle dependencies, but here we're talking about the Rake dependency mechanism. It's a double entendre. It invokes these tasks before running the compiler. Some of these tasks will download JARs from remote repositories, others will create them by compiling and packaging from a different project. Using file task ensures all the dependencies exist before the compiler can use them.

An easier way to specify dependencies is by calling the `compile.with` method. It takes a list of arguments and adds them to the dependency list. The `compile.with` method is easier to use, it accepts several type of dependencies. You can use file names, file tasks, projects, artifacts specifications and even pass arrays of dependencies.

Most dependencies fall into the last three categories. When you pass a project to `compile.with`, it picks up all the packages created by that project. In doing so, it establishes an order of dependency between the two projects (see Defining the Project). For example, if you make a change in project *teh-api* and build *teh-impl*, Buildr will detect that change, recompile and package *teh-api* before compiling *teh-impl*. You can also select a specific package using the `package` or `packages` methods (see Packaging).

When you pass an artifact specification to `compile.with`, it creates an `Artifact` task that will download that artifact from one of the remote repositories, install it in the local repository, and use it in your project. Rake's dependency mechanism is used here to make sure the artifact is downloaded once, when needed. Check the Artifacts section for more information about artifact specification and repositories.

For now let's just show a simple example:

```
compile.with 'org.apache.axis2:axis2:jar:1.2',
  'org.apache.derby:derby:jar:10.1.2.1', projects('teh-api',
'teh-impl')
```

Passing arrays to `compile.with` is just a convenient for handling multiple dependencies, we'll show more examples of that when we talk about Artifacts.

Likewise, the `compile` task has an array of file tasks that point at the source directories you want to compile from. You can access that array by calling `compile.sources`. You can use `compile.from` to add new source directories by passing a file name or a file task.

For example, let's run the APT tool on our annotated source code before compiling it:

```
compile.from apt
```

When you call `apt` on a project, it returns a file task that points to the `target/generated/apt` directory. This file task executes by running APT, using the same list of source directories, dependencies and compiler options. It then generates new source files in the target directory. Calling `compile.from` with that file task includes those additional source files in the list of compiled sources.

Here's another example:

```
jjtree = jjtree(_('src/main/jjtree'), :in_package=>'com.acme')
compile.from javacc(jjtree, :in_package=>'com.acme'), jjtree
```

This time, the variable `jjtree` is a file task that reads a JJTree source file from the `src/main/jjtree` directory, and generates additional source files in the `target/generated/jjtree` directory. The second line creates another file task that takes those source files, runs JavaCC on them, and generates yet more source files in `target/generated/javacc`. Finally, we include both sets of source files in addition to those already in `src/main/java`, and compile the lot.

The interesting thing about these two examples is how you're wiring file tasks together to create more complicated tasks, piping the output of one task into the inputs of another. Wiring tasks this way is the most common way to handle complex builds, and uses Rake's dependency mechanism to only run tasks when it detects a change to one of the source files.

You can also control the target directory. Use `compile.target` to get the target directory file task. If you need to change the target directory, call the `compile.into` method with the new path.

We use method pairs to give you finer control over the compiler, but also a way to easily configure it. Methods like `dependencies` and `sources` give you a live array you can manipulate, or iterate over. On the other hand, methods like `with` and `from` accept a wider set of arguments and clean them up for you. They also all return the same task you're calling, so you can chain methods together.

For example:

```
compile.from('srcs').with('org.apache.axis2:axis2:jar:1.2').
  into('classes').using(:target=>'1.4')
```

Buildr uses the method pair and method chaining idiom in many places to make your life easier without sacrificing flexibility.

Occasionally, you'll need to post-process the generated bytecode. Since you only want to do that after compiling, and let the compiler decide when to do that – only when changes require re-compiling – you'll want to extend the `compile` task. You can do that by calling `compile` with a block.

For example, to run the OpenJPA bytecode enhancer after compiling the source files:

```
compile { open_jpa_enhance }
```

You can change various compile options by calling, you guessed, `compile.options`. For example, to set the compiler to VM compatibility with Java 1.5 and turn on all Lint messages:

```
compile.options.target = '1.5'
compile.options.lint = 'all'
```

Or, if you want to chain methods together:

```
compile.using :target=>'1.5', :lint=>'all'
```

Sub-projects inherit compile options from their parent project, so you only need to change these settings once in the top project. You can do so, even if the top project itself doesn't compile anything.

The options available to you depend on which compiler you are using for this particular project, obviously the options are not the same for Java and Flash. Two options are designed to work consistently across compilers.

Buildr turns the `warning` option on by default, but turns it off when you run `buildr --silent`. It also sets the `debug` option on, but turns it off when making a release. You can also control the `debug` option from the command line, for example:

```
# When calling buildr
$ buildr compile debug=off

# Once until we change the variable
$ export DEBUG=off
$ buildr compile
```

The default source and target directories, compiler settings and other options you can use depend on the specific language. You can find more information in the Languages section.

Resources

The `compile` task comes bundled with a `resources` task. It copies files from the `src/main/resources` directory into `target/resources`. Best used for copying files that you want to included in the generated code, like configuration files, i18n messages, images, etc.

The `resources` task uses a filter that can change files as it copies them from source to destination. The most common use is by mapping values using a hash. For example, to substitute "${version}" for the project's version number and "${copyright}" for "Acme Inc © 2007" :

```
resources.filter.using 'version'=>version,
  'copyright'=>'Acme Inc (C) 2007'
```

You can also use profiles to supply a name/value map that all `resources` task should default to, by adding a `filter` element to each of the profiles. The following examples shows a `profiles.yaml` file that applies the same filter in development and test environments:

```
filter: &alpha1
  version: experimental
  copyright: Acme Inc (C) 2007

development:
  filter: *alpha1
test:
  filter: *alpha1
```

You can specify a different format by passing it as the first argument. Supported formats include:

Format	Usage
`:ant`	Map from `@key@` to value.
`:maven`	Map from `${key}` to value (default).
`:ruby`	Map from `#{key}` to value.

`Regexp`	Map using the matched value of the regular expression (e.g. `/=(.*?)=/`).

For example, using the `:ruby` format instead of the default `:maven` format:

```
resources.filter.using :ruby, 'version'=>version,
  'copyright'=>'Acme Inc (C) 2007'
```

For more complicated mapping you can also pass a method or a proc. The filter will call it once for each file with the file name and content.

If you need to copy resource files from other directories, add these source directories by calling the `from` method, for example:

```
resources.from _('src/etc')
```

You can select to copy only specific files using common file matching patterns. For example, to include only HTML files:

```
resources.include '*.html'
```

To include all files, except for files in the `scratch` directory:

```
resources.exclude 'scratch/*'
```

The filter always excludes the `CVS` and `.svn` directories, and all files ending with `.bak` or ~, so no need to worry about these.

A file pattern can match any file name or part of a file name using an asterisk (`*`). Double asterisk (`**`) matches directories recursively, for example, `'src/main/java/**/*.java'`. You can match any character using a question mark (`?`), or a set of characters using square brackets (`[]`), similar to regular expressions, for example, `'[Rr]eadme'`. You can also match from a set of names using curly braces (`{}`), for example, `'*.{html,css}'`.

You can use filters elsewhere. The `filter` method creates a filter, the `into` method sets the target directory, and `using` specifies the mapping. Last, you call `run` on the filter to activate it.

For example:

```
filter('src/specs').into('target/specs').
  using('version'=>version, 'created'=>Time.now).run
```

The `resources` task is, in fact, just a wrapper around such a filter that automatically adds the `src/main/resources` directory as one of the source directories.

More On Building

The `build` task runs the `compile` (and `resources`) tasks as prerequisites, followed by any actions you add to it, and completes by running the `test` task. The `build` task itself is a prerequisite to other tasks, for example, `package` and `upload`.

You can extend the `build` task in two ways. You can add more prerequisites that will execute before the task itself, or you can add actions that will execute as part of the task. Which one you choose is up to you, we'll show you how they differ in a second. If you call `build` with a list of tasks, it adds these tasks as prerequisites. Call `build` with a block, and it adds that block as an action. Again, a common idiom you'll see elsewhere in Buildr and Rake.

Let's look at a simple example. Say we want to generate a Derby database from an SQL file and include it in the ZIP package:

```
db = Derby.create(_('target/derby/db')=>_('src/main/sql/derby.sql'))
package(:zip).include db
```

There's nothing fundamentally wrong with this code, if that's what you intend to do. But in practice, you don't always run the `package` task during development, so you won't notice if something is wrong with this task when you build. For example, if it fails to generate the SQL file. In addition, the `package` task runs after `build`, so you can't use the database in your test cases.

So let's refactor it. We're going to use the variable `db` to reference the file task that creates the database, and make it a prerequisite of the `build` task. And use that same variable again to include the database in the ZIP package:

```
db = Derby.create(_('target/derby/db')=>_('src/main/sql/derby.sql'))
build db
package(:zip).include db
```

Much better. We're using the same task twice, but since we're using Rake here, it will only execute once. In fact, it will only execute if we don't already have a Derby database, or if it detects a change to the SQL file and needs to recreate the database.

`Derby.create` is not part of Buildr, you can get derby.rake here.

Here's another example. We want to copy some files over as part of the build, and apply a filter to them. This time, we're going to extend the `build` task:

```
build do
  filter('src/specs').into('target/specs').
    using('version'=>version, 'created'=>Time.now).run
end
```

The `build` task is recursive, so running `buildr build` picks the current project and runs its `build` task, which in turn runs the `build` task on each of its sub-projects. One `build` task to rule them all.

Cleaning

The `build` task has an evil twin, the `clean` task. It's the task you use to remove all the files created during the build, especially when you mess things up and want to start all over.

It basically erases the target directories, the one called `target`, and if you get creative and change the target directory for tasks like `compile`, it will also erase those. If you decide to generate files outside the target directory and want to cleanup after yourself, just extend the `clean` task.

For example:

```
clean { rm_rf _('staged') }
```

The `rm_rf` method deletes the directory and all files in it. It's named after UNIX's infamous `rm -rf`. Use it wisely. This is also a good time to introduce you to `FileUtils`, a standard Ruby library that contains convenient methods for creating and deleting directories, copying and moving files, even comparing two files. They're all free of charge when you use Buildr.

Now let's talk about the artifacts we mentioned before.

Artifacts

In Buildr, almost everything is a file or a file task. You compile source files that come from the file system using dependencies found on the file system, generating even more files. But how do you get these dependencies to start with, and how do you share them with others?

Artifacts. We designed Buildr to work as a drop-in replacement for Maven 2.0, and share artifacts through the same local and remote repositories. Artifact tasks know how to download a file from one of the remote repositories, and install it in the local repository, where Buildr can find it. Packages know how to create files and upload them to remote repositories.

We'll get into all of that in a second, but first, let's introduce the artifact specification. It's a simple string that takes one of two forms:

```
group:id:type:version
group:id:type:classifier:version
```

For example, `'org.apache.axis2:axis2:jar:1.2'` refers to an artifact with group identifier org.apache.axis2, artifact identifier axis2, a JAR file with version 1.2. Classifiers are typically used to distinguish between similar file types, for example, a source distribution and a binary distribution that otherwise have the same identifier and are both ZIP files.

Specifying Artifacts

If your Buildfile spells out `'org.apache.axis2:axis2:jar:1.2'` more than once, you're doing something wrong. Repeating the same string over and over will make your code harder to maintain. You'll know that when you upgrade to a new version in one place, forget to do it in another, and end up with a mismatch.

You can use Ruby's syntax to do simple string substitution, for example:

```
AXIS_VERSION = '1.2'

compile.with "org.apache.axis2:axis2:jar:#{AXIS_VERSION}"
```

Better yet, you can define all your artifacts at the top of the Buildfile and use constants to reference them in your project definition. For example:

```
AXIS2 = 'org.apache.axis2:axis2:jar:1.2'

compile.with AXIS2
```

Note that we're not using a separate constant for the version number. In our experience, it's unnecessary. The version number intentionally appears at the end of the string, where it stands out easily.

If you have a set of artifacts that belong to the same group and version, and that's quite common, you can use the `group` shortcut:

```
AXIOM  = group('axiom-api', 'axiom-impl', 'axiom-dom',
  :under=>'org.apache.ws.commons.axiom', :version=>'1.2.4')
```

Buildr projects also define a `group` attribute which can lead to some confusion. If you want to define an artifact group within a project definition, you should use the explicit qualifier `Buildr::group`.

If you have several artifacts you always use together, consider placing them in an array. Methods that accept lists of artifacts also accept arrays. For example:

```
OPENJPA = ['org.apache.openjpa:openjpa-all:jar:0.9.7',
  'net.sourceforge.serp:serp:jar:1.12.0']
AXIS_OF_WS = [AXIS2, AXIOM]

compile.with OPENJPA, AXIS_OF_WS
```

Another way to group related artifacts together and access them individually is using the `struct` shortcut. For example:

```
JAVAX = struct(
  :activation  =>'javax.activation:activation:jar:1.1',
  :persistence =>'javax.persistence:persistence-api:jar:1.0',
  :stream      =>'stax:stax-api:jar:1.0.1',
)

compile.with JAVAX.persistence, OPENJPA
```

In our experience, using constants in this manner makes your Buildfile much easier to write and maintain.

And, of course, you can always place your artifact specifications in a separate file and require it into your Buildfile. For example, if you're working on several different projects that all share the same artifacts:

```
require '../shared/artifacts'
```

When you use `require`, Ruby always looks for a filename with the `.rb` extension, so in this case it expects to find `artifacts.rb` in the `shared` directory.

One last thing. You can also treat artifact specifications as hashes. For example:

```
AXIS = { :group=>'org.apache.axis2', :id=>'axis2', :version=>'1.2' }
compile.with AXIS
puts compile.dependencies.first.to_hash
=> { :group=>'org.apache.axis2', :id=>'axis2',
     :version=>'1.2', :type=>:jar }
```

Specifying Repositories

Buildr can download artifacts for you, but only if you tell it where to find them. You need to specify at least one remote repository, from which to download these artifacts.

When you call `repositories.remote`, you get an array of URLs for the various remote repositories. Initially, it's an empty array, to which you can add new repositories. For example:

```
repositories.remote << 'http://www.ibiblio.org/maven2/'
```

If you need to use a proxy server to access remote repositories, you can set the environment variable `HTTP_PROXY` to the proxy server URL. You can also work without a proxy for certain hosts by specifying the `NO_PROXY` environment variable. For example:

```
$ export HTTP_PROXY = 'http://myproxy:8080'
$ export NO_PROXY = '*.mycompany.com,localhost,special:800'
```

Alternatively you can use the Buildr options `proxy.http` and `proxy.exclude`:

```
options.proxy.http = 'http://myproxy:8080'
options.proxy.exclude << '*.mycompany.com'
options.proxy.exclude << 'localhost'
```

All the artifacts download into the local repository. Since all your projects share the same local repository, you only need to download each artifact once. Buildr was designed to be used alongside Maven 2.0, for example, when migrating projects from Maven 2.0 over to Buildr. By default it will share the same local repository, expecting the repository to be the `.m2/repository` directory inside your home directory.

You can choose to relocate the local repository by giving it a different path, for example:

```
repositories.local = '/usr/local/maven/repository'
```

That's one change you don't want to commit into the Buildfile, so the best place to do it is in your home directory's `buildr.rb` file.

Buildr downloads artifacts when it needs to use them, for example, to compile a project. You don't need to download artifacts directly. Except when you do, for example, if you want to download all the latest artifacts and then go off-line. It's as simple as:

```
$ buildr artifacts
```

Downloading Artifacts

Within your buildfile you can download artifacts directly by invoking them, for example:

```
artifact('org.apache.openjpa:openjpa-all:jar:0.9.7').invoke
artifacts(OPENJPA).each(&:invoke)
```

When you let Buildr download artifacts for you, or by invoking the artifact task yourself, it scans through the remote repositories assuming each repository follows the Maven 2 structure. Starting from the root repository URL, it will look for each artifact using the path `group/id/version/id-version.type` (or `…/id-version-classifier.type`). The group identifier becomes a path by turning periods (`.`) into slashes (`/`). So to find `org.apache.axis2:axis2:jar:1.2`, we're going to look for `org/apache/axis2/axis2/1.2/axis2-1.2.jar`.

You'll find a lot of open source Java libraries in public repositories that support this structure (for example, the Ibiblio Maven repository). And, of course, every remote repository you setup for your projects.

But there are exceptions to the rule. Say we want to download the Dojo widget library and use it in our project. It's available from the Dojo Web site, but that site doesn't follow the Maven repository conventions, so our feeble attempt to use existing remote repositories will fail.

We can still treat Dojo as an artifact, by telling Buildr where to download it from:

```
DOJO = '0.2.2'

url = "http://download.dojotoolkit.org/
release-#{DOJO}/dojo-#{DOJO}-widget.zip"
download(artifact("dojo:dojo:zip:widget:#{DOJO}")=>url)
```

Explaining how it works is tricky, skip if you don't care for the details. On the other hand, it will give you a better understanding of Buildr/Rake, so if not now, come back and read it later.

We use the `artifact` method to create an `Artifact` task that references the Dojo widget in our local repository. The `Artifact` task is a file task with some additional behavior added by Buildr. When you call `compile.with`, that's exactly what it does internally, turning each of your artifact specifications into an `Artifact` task.

But the `Artifact` task doesn't know how to download the Dojo widget, only how to handle conventional repositories. So we're going to create a download task as well. We use the `download` method to create a file task that downloads the file from a remote URL. (Of course, it will only download the file if it doesn't already exist.)

But which task gets used when? We could have defined these tasks separately and used some glue code to make one use the other. Instead, we call `download` with the results of `artifact`. Essentially, we're telling `download` to use the same file path as `artifact`. So now we have two file tasks that point to the very same file. We wired them together.

You can't have more than one task pointing to the same file. Rake's rule of the road. What Rake does is merge the tasks together, creating a single file task for `artifact`, and then enhancing it with another action from `download`. One task, two actions. Statistically, we've doubled the odds that at least one of these actions will manage to download the Dojo widget and install it in the local repository.

Since we ordered the calls to `artifact` first and `download` second, we know the actions will execute in that order. But `artifact` is slightly devilish: when its action runs, it adds another action to the end of the list. So the `artifact` action runs first, adds an action at the end, the `download` action runs second, and downloads the Dojo widget for us. The second `artifact` action runs last, but checks that the file already exist and doesn't try to download it again.

Magic.

Install and Upload

Generally you use artifacts that download from remote repositories into the local repository, or artifacts packaged by the project itself (see Packaging), which are then installed into the local repository and uploaded to the release server.

Some artifacts do not fall into either category. In this example we're going to download a ZIP file, extract a JAR file from it, and use that JAR file as an artifact. We would then expect to install this JAR in the local repository and upload it to the release server, where it can be shared with other projects.

So let's start by creating a task that downloads the ZIP, and another one to extract it and create the JAR file:

```
app_zip = download('target/app.zip'=>url)
bean_jar = file('target/app/bean.jar'=>unzip('target/app'=>app_zip))
```

When you call `artifact`, it returns an `Artifact` task that points to the artifact file in the local repository, downloading the file if it doesn't already exist. You can override this behavior by enhancing the task and creating the file yourself (you may also want to create a POM file). Or much simpler, call the `from` method on the artifact and tell it where to find the source file.

So the next step is to specify the artifact and tell it to use the extracted JAR file:

```
bean = artifact('example.com:beans:jar:1.0').from(bean_jar)
```

The artifact still points to the local repository, but when we invoke the task it copies the source file over to the local repository, instead of attempting a download.

Use the `install` method if you want the artifact and its POM installed in the local repository when you run the `install` task. Likewise, use the `upload` method if you want the artifact uploaded to the release server when you run the `upload` task. You do not need to do this on artifacts downloaded from a remote server, or created with the `package` method, the later are automatically added to the list of installed/uploaded artifacts.

Our example ends by including the artifact in the `install` and `upload` tasks:

```
install bean
upload bean
```

Calling the `install` (and likewise `upload`) method on an artifact run `buildr install`. If you need to download and install an artifact, invoke the task directly with `install(<artifact>).invoke`.

Next we're going to package some artifacts.

Packaging

For our next trick, we're going to try and create an artifact ourselves. We're going to start with:

```
package :jar
```

We just told the project to create a JAR file in the `target` directory, including all the classes (and resources) that we previously compiled into `target/classes`. Or we can create a WAR file:

```
package :war
```

The easy case is always easy, but sometimes we have more complicated use cases which we'll address through the rest of this section.

Now let's run the build, test cases and create these packages:

```
$ buildr package
```

The `package` task runs the `build` task (remember: `compile` and `test`) and then runs each of the packaging tasks, creating packages in the projects' target directories.

The package task and package methods are related, but that relation is different from other task/method pairs. The package method creates a file task that points to the package in the target directory and knows how to create it. It then adds itself as a prerequisite to the package task. Translation: you can create multiple packages from the same project.

Specifying And Referencing Packages

Buildr supports several packaging types, and so when dealing with packages, you have to indicate the desired package type. The packaging type can be the first argument, or the value of the :type argument. The following two are equivalent:

```
package :jar
package :type=>:jar
```

If you do not specify a package type, Buildr will attempt to infer one.

In the documentation you will find a number of tasks dealing with specific packaging types (ZipTask, JarTask, etc). The package method is a convenience mechanism that sets up the package for you associates it with various project life cycle tasks.

To package a particular file, use the :file argument, for example:

```
package :zip, :file=>_('target/interesting.zip')
```

This returns a file task that will run as part of the project's package task (generating all packages). It will invoke the build task to generate any necessary prerequisites, before creating the specified file.

The package type does not have to be the same as the file name extension, but if you don't specify the package type, it will be inferred from the extension.

Most often you will want to use the second form to generate packages that are also artifacts. These packages have an artifact specification, which you can use to reference them from other projects (and buildfiles). They are also easier to share across projects: artifacts install themselves in the local repository when running the `install` task, and upload to the remote repository when running the `upload` task (see Installing and Uploading).

The artifact specification is based on the project name (using dashes instead of colons), group identifier and version number, all three obtained from the project definition. You can specify different values using the `:id`, `:group`, `:version` and `:classifier` arguments. For example:

```
define 'killer-app', :version=>'1.0' do
  # Generates silly-1.0.jar
  package :jar, :id=>'silly'

  # Generates killer-app-la-web-1.x.war
  project 'la-web' do
    package :war, :version=>'1.x'
  end

  # Generates killer-app-the-api-1.0-sources.zip
  project 'teh-api' do
    package :zip, :classifier=>'sources'
  end
end
```

The file name is determined from the identifier, version number, classifier and extension associated with that packaging type.

If you do not specify the packaging type, Buildr attempt to infer it from the project definition. In the general case it will use the default packaging type, ZIP. A project that compiles Java classes will default to JAR packaging; for other languages, consult the specific documentation.

A single project can create multiple packages. For example, a Java project may generate a JAR package for the runtime library and another JAR containing just the API; a ZIP file for the source code and another ZIP for the documentation. Make sure to always call `package` with enough information to identify the specific package you are referencing. Even if the project only defines a single package, calling the `package` method with no arguments does not necessarily refer to that one.

You can use the `packages` method to obtain a list of all packages defined in the project, for example:

```
project('killer-app:teh-impl').packages.first
project('killer-app:teh-impl').packages.select { |pkg| pkg.type ==
:zip }
```

Packaging ZIPs

ZIP is the most common form of packaging, used by default when no other packaging type applies. It also forms the basis for many other packaging types (e.g. JAR and WAR). Most of what you'll find here applies to other packaging types.

Let's start by including additional files in the ZIP package. We're going to include the `target/docs` directory and `README` file:

```
package(:zip).include _('target/docs'), 'README'
```

The `include` method accepts files, directories and file tasks. You can also use file pattern to match multiple files and directories. File patterns include asterisk (`*`) to match any file name or part of a file name, double asterisk (`**`) to match directories recursively, question mark (`?`) to match any character, square braces (`[]`) to match a set of characters, and curly braces (`{}`) to match one of several names.

And the same way you `include`, you can also `exclude` specific files you don't want showing up in the ZIP. For example, to exclude `.draft` and `.raw` files:

```
package(:zip).include('target/docs').
  exclude('target/docs/**/*.{draft,raw}')
```

So far we've included files under the root of the ZIP. Let's include some files under a given path using the `:path` option:

```
package(:zip).include 'target/docs', :path=>"#{id}-#{version}"
```

If you need to use the `:path` option repeatedly, consider using the `tap` method instead. For example:

```
package(:zip).path("#{id}-#{version}").tap do |path|
  path.include 'target/docs'
  path.include 'README'
end
```

The `tap` method is not part of the core library, but a very useful extension. It takes an object, yields to the block with that object, and then returns that object.

To allow you to spread files across different paths, the include/exclude patterns are specific to a path. So in the above example, if you want to exclude some files from the "target/docs" directory, make sure to call `exclude` on the path, not on the ZIP task itself.

If you need to include a file or directory under a different name, use the `:as` option. For example:

```
package(:zip).include('corporate-logo-350x240.png', :as=>'logo.png')
```

You can also use `:as=>'.'` to include all files from the given directory. For example:

```
package(:zip).include 'target/docs/*'
package(:zip).include 'target/docs', :as=>'.'
```

These two are almost identical. They both include all the files from the `target/docs` directory, but not the directory itself. But they operate differently. The first line expands to include all the files in `target/docs`. If you don't already have files in `target/docs`, well, then it won't do anything interesting. Your ZIP will come up empty. The second file includes the directory itself, but strips the path during inclusion. You can define it now, create these files later, and then ZIP them all up.

For example, when `package :jar` decides to include all the files from `target/classes`, it's still working on the project definition, and has yet to compile anything. Since `target/classes` may be empty, may not even exist, it uses `:as=>'.'`.

If you need to get rid of all the included files, call the `clean` method. Some packaging types default to adding various files and directories, for example, JAR packaging will include all the compiled classes and resources.

You can also merge two ZIP files together, expanding the content of one ZIP into the other. For example:

```
package(:zip).merge 'part1.zip', 'part2.zip'
```

If you need to be more selective, you can apply the include/exclude pattern to the expanded ZIP. For example:

```
# Everything but the libs
package(:zip).merge('bigbad.war').exclude('libs/**/*')
```

Packaging JARs

JAR packages extend ZIP packages with support for Manifest files and the META-INF directory. They also default to include the class files found in the `target/classes` directory.

You can tell the JAR package to include a particular Manifest file:

```
package(:jar).with :manifest=>_('src/main/MANIFEST.MF')
```

Or generate a manifest from a hash:

```
package(:jar).with :manifest=>{ 'Copyright'=>'Acme Inc (C) 2007' }
```

You can also generate a JAR with no manifest with the value `false`, create a manifest with several sections using an array of hashes, or create it from a proc.

In large projects, where all the packages use the same manifest, it's easier to set it once on the top project using the `manifest` project property. Sub-projects inherit the property from their parents, and the `package` method uses that property if you don't override it, as we do above.

For example, we can get the same result by specifying this at the top project:

```
manifest['Copyright'] = 'Acme Inc (C) 2007'
```

If you need to mix-in the project's manifest with values that only one package uses, you can do so easily:

```
package(:jar).with
:manifest=>manifest.merge('Main-Class'=>'com.acme.Main')
```

If you need to include more files in the `META-INF` directory, you can use the `:meta_inf` option. You can give it a file, or array of files. And yes, there is a `meta_inf` project property you can set once to include the same set of file in all the JARs. It works like this:

```
meta_inf << file('DISCLAIMER') << file('NOTICE')
```

If you have a `LICENSE` file, it's already included in the `meta_inf` list of files.

Other than that, `package :jar` includes the contents of the compiler's target directory and resources, which most often is exactly what you intend it to do. If you want to include other files in the JAR, instead or in addition, you can do so using the `include` and `exclude` methods. If you do not want the target directory included in your JAR, simply call the `clean` method on it:

```
package(:jar).clean.include( only_these_files )
```

Packaging WARs

Pretty much everything you know about JARs works the same way for WARs, so let's just look at the differences.

Without much prompting, `package :war` picks the contents of the `src/main/webapp` directory and places it at the root of the WAR, copies the compiler target directory into the `WEB-INF/classes` path, and copies any compiled dependencies into the `WEB-INF/libs` paths.

Again, you can use the `include` and `exclude` methods to change the contents of the WAR. There are two convenience options you can use to make the more common changes. If you need to include a classes directory other than the default:

```
package(:war).with :classes=>_('target/additional')
```

If you want to include a different set of libraries other than the default:

```
package(:war).with :libs=>MYSQL_JDBC
```

Both options accept a single value or an array. The `:classes` option accepts the name of a directory containing class files, initially set to `compile.target` and `resources.target`. The `:libs` option accepts artifact specifications, file names and tasks, initially set to include everything in `compile.dependencies`.

As you can guess, the package task has two attributes called `classes` and `libs`; the `with` method merely sets their value. If you need more precise control over these arrays, you can always work with them directly, for example:

```
# Add an artifact to the existing set:
package(:war).libs += artifacts(MYSQL_JDBC)
# Remove an artifact from the existing set:
package(:war).libs -= artifacts(LOG4J)
# List all the artifacts:
puts 'Artifacts included in WAR package:'
puts package(:war).libs.map(&:to_spec)
```

Packaging AARs

Axis2 service archives are similar to JAR's (compiled classes go into the root of the archive) but they can embed additional libraries under /lib and include `services.xml` and WSDL files.

```
package(:aar).with(:libs=>'log4j:log4j:jar:1.1')
package(:aar).with(:services_xml=>_('target/services.xml'),
:wsdls=>_('target/*.wsdl'))
```

The `libs` attribute is a list of .jar artifacts to be included in the archive under /lib. The default is no artifacts; compile dependencies are not included by default.

The `services_xml` attribute points to an Axis2 services configuration file called `services.xml` that will be placed in the `META-INF` directory inside the archive. The default behavior is to point to the `services.xml` file in the project's `src/main/axis2` directory. In the second example above we set it explicitly.

The `wsdls` attribute is a collection of file names or glob patterns for WSDL files that get included in the `META-INF` directory. In the second example we include WSDL files from the `target` directory, presumably created by an earlier build task. In addition, AAR packaging will include all files ending with `.wsdl` from the `src/main/axis2` directory.

If you already have WSDL files in the `src/main/axis2` directory but would like to perform some filtering, for example, to set the HTTP port number, consider ignoring the originals and including only the filtered files, for example:

```
# Host name depends on environment.
host = ENV['ENV'] == 'test' ? 'test.host' : 'ws.example.com'
filter.from('src/main/axis2').into('target').
  include('services.xml', '*.wsdl').using('http_port'=>'8080',
'http_host'=>host)
package(:aar).wsdls.clear
package(:aar).with(:services_xml=>_('target/services.xml'),
:wsdls=>_('target/*.wsdl'))
```

Packaging EARs

EAR packaging is slightly different from JAR/WAR packaging. It's main purpose is to package components together, and so it includes special methods for handling component inclusion that take care to update application.xml and the component's classpath.

EAR packages support four component types:

Argument	Component
`:war`	J2EE Web Application (WAR).
`:ejb`	Enterprise Java Bean (JAR).
`:jar`	J2EE Application Client (JAR).
`:lib`	Shared library (JAR).

This example shows two ways for adding components built by other projects:

```
package(:ear) << project('coolWebService').package(:war)
package(:ear).add project('commonLib') # By default, the JAR package
```

Adding a WAR package assumes it's a WAR component and treats it as such, but JAR packages can be any of three component types, so by default they are all treated as shared libraries. If you want to add an EJB or Application Client component, you need to say so explicitly, either passing `:type=>package`, or by passing the component type in the `:type` option.

Here are three examples:

```
# Assumed to be a shared library.
package(:ear).add 'org.springframework:spring:jar:2.6'
# Component type mapped to package.
package(:ear).add :ejb=>project('beanery')
# Adding component with specific package type.
package(:ear).add project('client'), :type=>:jar
```

By default, WAR components are all added under the `/war` path, and likewise, EJB components are added under the `/ejb` path, shared libraries under `/lib` and Application Client components under `/jar`.

If you want to place components in different locations you can do so using the `:path` option, or by specifying a different mapping between component types and their destination directory. The following two examples are equivalent:

```
# Specify once per component.
package(:ear).add project('coolWebService').package(:war),
:path=>'coolServices'
# Configure once and apply to all added components.
package(:ear).dirs[:war] = 'coolServices'
package(:ear) << project('coolWebService').package(:war)
```

EAR packages include an `application.xml` file in the `META-INF` directory that describes the application and its components. This file is created for you during packaging, by referencing all the components added to the EAR. There are a couple of things you will typically want to change.

- **display-name** — The application's display name defaults to the project's identifier. You can change that by setting the `display_name` attribute.
- **context-root** — WAR components specify a context root, based on the package identifier, for example, "cool-web-1.0.war" will have the context root "cool-web". To specify a different context root, add the WAR package with the `context_root` option.

Again, by example:

```
package(:ear).display_name = 'MyCoolWebService'
package(:ear).add project('coolWebService').package(:war),
:context-root=>'coolness'
```

If you need to disable the context root (e.g. for Portlets), set `context_root` to `false`.

Packaging Tars and GZipped Tars

Everything you know about working with ZIP files translates to Tar files, the two tasks are identical in more respect, so here we'll just go over the differences.

```
package(:tar).include _('target/docs'), 'README'
package(:tgz).include _('target/docs'), 'README'
```

The first line creates a Tar archive with the extension `.tar`, the second creates a GZipped Tar archive with the extension `.tgz`.

In addition to packaging that includes the archive in the list of installed/released files, you can use the method `tar` to create a `TarTask`. This task is similar to `ZipTask`, and introduces the `gzip` attribute, which you can use to tell it whether to create a regular file, or GZip it. By default the attribute it set to true (GZip) if the file name ends with either `.gz` or `.tgz`.

Installing and Uploading

You can bring in the artifacts you need from remote repositories and install them in the local repositories. Other projects have the same expectation, that your packages be their artifacts.

So let's create these packages and install them in the local repository where other projects can access them:

```
$ buildr install
```

If you changes your mind you can always:

```
$ buildr uninstall
```

That works between projects you build on the same machine. Now let's share these artifacts with other developers through a remote repository:

```
$ buildr upload
```

Of course, you'll need to tell Buildr about the release server:

```
repositories.release_to = 'sftp://john:secret@release/usr/share/repo'
```

This example uses the SFTP protocol. In addition, you can use the HTTP protocol — Buildr supports HTTP and HTTPS, Basic Authentication and uploads using PUT — or point to a directory on your file system.

The URL in this example contains the release server ("release"), path to repository ("user/share/repo") and username/password for access. The way SFTP works, you specify the path on the release server, and give the user permissions to create directories and files inside the repository. The file system path is different from the path you use to download these artifacts through an HTTP server, and starts at the root, not the user's home directory.

Of course, you'll want to specify the release server URL in the Buildfile, but leave the username/password settings private in your local `buildr.rb` file. Let's break up the release server settings:

```
# build.rb, loaded first
repositories.release_to[:username] = 'john'
repositories.release_to[:password] = 'secret'

# Buildfile, loaded next
repositories.release_to[:url] = 'sftp://release/usr/share/repo'
```

The `upload` task takes care of uploading all the packages created by your project, along with their associated POM files and MD5/SHA1 signatures (Buildr creates these for you).

If you need to upload other files, you can always extend the `upload` task and use `repositories.release_to` in combination with `URI.upload`. You can also extend it to upload to different servers, for example, to publish the documentation and test coverage reports to your site:

```
# We'll let some other task decide how to create 'docs'
task 'deploy'=>'docs' do
  uri = URI("sftp://#{username}:#{password}@var/www/docs")
  uri.upload file('docs')
end
```

Packaging Sources and JavaDocs

IDEs can take advantage of source packages to help you debug and trace through compiled code. We'll start with a simple example:

```
package :sources
```

This one creates a ZIP package with the classifier "sources" that will contain all the source directories in that project, typically `src/main/java`, but also other sources generated from Apt, JavaCC, XMLBeans and friends.

You can also generate a ZIP package with the classifier "javadoc" that contains the JavaDoc documentation for the project. It uses the same set of documentation files generated by the project's `javadoc` task, so you can use it in combination with the `javadoc` method. For example:

```
package :javadoc
javadoc :windowtitle=>'Buggy but Works'
```

By default Buildr picks the project's description for the window title.

You can also tell Buildr to automatically create sources and JavaDoc packages in all the sub-projects that have any source files to package or document. Just add either or both of these methods in the top-level project:

```
package_with_sources
package_with_javadoc
```

You can also tell it to be selective using the `:only` and `:except` options.
For example:

```
package_with_javadoc :except=>'la-web'
```

We packaged the code, but will it actually work? Let's see what the tests say.

Testing

Untested code is broken code, so we take testing seriously. Off the bat you get to use either JUnit or TestNG for writing unit tests and integration tests. And you can also add your own framework, or even script tests using Ruby. But= first, let's start with the basics.

Writing Tests

Each project has a `TestTask` that you can access using the `test` method. `TestTask` reflects on the fact that each project has one task responsible for getting the tests to run and acting on the results. But in fact there are several tasks that do all the work, and a `test` task coordinates all of that.

The first two tasks to execute are `test.compile` and `test.resources`. They work similar to `compile` and `resources`, but uses a different set of directories. For example, Java tests compile from the `src/test/java` directory into the `target/test/classes` directory, while resources are copied from `src/test/resources` into `target/test/resources`.

The `test.compile` task will run the `compile` task first, then use the same dependencies to compile the test classes. That much you already assumed. It also adds the test framework (e.g. JUnit, TestNG) and JMock to the dependency list. Less work for you.

If you need more dependencies, the best way to add them is by calling `test.with`. This method adds dependencies to both `compile.dependencies` (for compiling) and `test.dependencies` (for running). You can manage these two dependency lists separately, but using `test.with` is good enough in more cases.

Once compiled, the `test` task runs all the tests.

Different languages use different test frameworks. You can find out more about available test frameworks in the Languages section.

Excluding Tests and Ignoring Failures

If you have a lot of tests that are failing or just hanging there collecting dusts, you can tell Buildr to ignore them. You can either tell Buildr to only run specific tests, for example:

```
test.include 'com.acme.tests.passing.*'
```

Or tell it to exclude specific tests, for example:

```
test.exclude '*FailingTest', '*FailingWorseTest'
```

Note that we're always using the package qualified class name, and you can use star (`*`) to substitute for any set of characters.

When tests fail, Buildr fails the `test` task. This is usually a good thing, but you can also tell Buildr to ignore failures by resetting the `:fail_on_failure` option:

```
test.using :fail_on_failure=>false
```

Besides giving you a free pass to ignore failures, you can use it for other causes, for example, to be somewhat forgiving:

```
test do
  fail 'More than 3 tests failed!' if test.failed_tests.size > 3
end
```

The `failed_tests` collection holds the names of all classes with failed tests. And there's `classes`, which holds the names of all test classes. Ruby arithmetic allows you to get the name of all passed test classes with a simple `test.classes - test.failed_tests`. We'll let you imagine creative use for these two.

Running Tests

It's a good idea to run tests every time you change the source code, so we wired the `build` task to run the `test` task at the end of the build. And conveniently enough, the `build` task is the default task, so another way to build changes in your code and run your tests:

```
$ buildr
```

That only works with the local `build` task and any local task that depends on it, like `package`, `install` and `upload`. Each project also has its own `build` task that does not invoke the `test` task, so `buildr build` will run the tests cases, but `buildr foo:build` will not.

While it's a good idea to always run your tests, it's not always possible. There are two ways you can get `build` to not run the `test` task. You can set the environment variable `test` to `no` (but `skip` and `off` will also work). You can do that when running Buildr:

```
$ buildr test=no
```

Or set it once in your environment:

```
$ export TEST=no
$ buildr
```

If you're feeling really adventurous, you can also disable tests from your Buildfile or `buildr.rb` file, by setting `options.test = false`. We didn't say it's a good idea, we're just giving you the option.

The `test` task is just smart enough to run all the tests it finds, but will accept include/exclude patterns. Often enough you're only working on one broken test and you only want to run that one test. Better than changing your Buildfile, you can run the `test` task with a pattern. For example:

```
$ buildr test:KillerAppTest
```

Buildr will then run only tests that match the pattern `KillerAppTest`. It uses pattern matching, so `test:Foo` will run `com.acme.FooTest` and `com.acme.FooBarTest`. With Java, you can use this to pick a class name, or a package name to run all tests in that package, or any such combination. In fact, you can specify several patterns separated with commas. For example:

```
$ buildr test:FooTest,BarTest
```

As you probably noticed, Buildr will stop your build at the first test that fails. We think it's a good idea, except when it's not. If you're using a continuous build system, you'll want a report of all the failed tests without stopping at the first failure. To make that happen, set the environment variable `test` to "all", or the Buildr `options.test` option to `:all`. For example:

```
$ buildr package test=all
```

We're using `package` and not `build` above. When using a continuous build system, you want to make sure that packages are created, contain the right files, and also run the integration tests.

Integration Tests

So far we talked about unit tests. Unit tests are run in isolation on the specific project they test, in an isolated environment, generally with minimal setup and teardown. You get a sense of that when we told you tests run after the `build` task, and include JMock in the dependency list.

In contrast, integration tests are run with a number of components, in an environment that resembles production, often with more complicates setup and teardown procedures. In this section we'll talk about the differences between running unit and integration tests.

You write integration tests much the same way as you write unit tests, using `test.compile` and `test.resources`. However, you need to tell Buildr that your tests will execute during integration test. To do so, add the following line in your project definition:

```
test.using :integration
```

Typically you'll use unit tests in projects that create internal modules, such as JARs, and integration tests in projects that create components, such as WARs and EARs. You only need to use the `:integration` option with the later.

To run integration tests on the current project:

```
$ buildr integration
```

You can also run specific tests cases, for example:

```
$ buildr integration:ClientTest
```

If you run the `package` task (or any task that depends on it, like `install` and `upload`), Buildr will first run the `build` task and all its unit tests, and then create the packages and run the integration tests. That gives you full coverage for your tests and ready to release packages. As with unit tests, you can set the environment variable `test` to "no" to skip integration tests, or "all" to ignore failures.

Using Setup and Teardown

Some tests need you to setup an environment before they run, and tear it down afterwards. The test frameworks (JUnit, TestNG) allow you to do that for each test. Buildr provides two additional mechanisms for dealing with more complicated setup and teardown procedures.

Integration tests run a setup task before the tests, and a teardown task afterwards. You can use this task to setup a Web server for testing your Web components, or a database server for testing persistence. You can access either task by calling `integration.setup` and `integration.teardown`. For example:

```
integration.setup { server.start ; server.deploy }
integration.teardown { server.stop }
```

Depending on your build, you may want to enhance the setup/teardown tasks from within a project, for example, to populate the database with data used by that project's test, or from outside the project definition, for example, to start and stop the Web server.

Likewise, each project has its own setup and teardown tasks that are run before and after tests for that specific project. You can access these tasks using `test.setup` and `test.teardown`.

Testing Your Build

So you got the build running and all the tests pass, binaries are shipping when you find out some glaring omissions. The license file is empty, the localized messages for Japanese are missing, the CSS files are not where you expect them to be. The fact is, some errors slip by unit and integration tests. So how do we make sure the same mistake doesn't happen again?

Each project has a `check` task that runs just after packaging. You can use this task to verify that your build created the files you wanted it to create. And to make it extremely convenient, we introduced the notion of expectations.

You use the `check` method to express and expectation. Buildr will then run all these expectations against your project, and fail at the first expectation that doesn't match. An expectation says three things. Let's look at a few examples:

```
check package(:war), 'should exist' do
  it.should exist
end
check package(:war), 'should contain a manifest' do
  it.should contain('META-INF/MANIFEST.MF')
end
check package(:war).path('WEB-INF'), 'should contain files' do
  it.should_not be_empty
end
check package(:war).path('WEB-INF/classes'), 'should contain classes'
do
  it.should contain('**/*.class')
end
check package(:war).entry('META-INF/MANIFEST'), 'should have license'
do
  it.should contain(/Copyright (C) 2007/)
end
check file('target/classes'), 'should contain class files' do
  it.should contain('**/*.class')
end
check file('target/classes/killerapp/Code.class'), 'should exist' do
  it.should exist
end
```

The first argument is the subject, or the project if you skip the first argument. The second argument is the description, optional, but we recommend using it. The method `it` returns the subject.

You can also write the first expectation like this:

```
check do
  package(:jar).should exist
end
```

We recommend using the subject and description, they make your build easier to read and maintain, and produce better error messages.

There are two methods you can call on just about any object, called `should` and `should_not`. Each method takes an argument, a matcher, and executes that matcher. If the matcher returns false, `should` fails. You can figure out what `should_not` does in the same case.

Buildr provides the following matchers:

Method	Checks that …
`exist`	Given a file task checks that the file (or directory) exists.
`empty`	Given a file task checks that the file (or directory) is empty.
`contain`	Given a file task referencing a file, checks its contents, using string or regular expression. For a file task referencing a directory, checks that it contains the specified files; global patterns using * and ** are allowed.

All these matchers operate against a file task. If you run them against a ZipTask (including JAR, WAR, etc) they can also check the contents of the ZIP file. And as you can see in the examples above, you can also run them against a path in a ZIP file, checking its contents as if it was a directory, or against an entry in a ZIP file, checking the content of that file.

The `package` method returns a package task based on packaging type, identifier, group, version and classifier. The last four are inferred, but if you create a package with different specifications (for example, you specify a classifier) your checks must call `package` with the same qualifying arguments to return the very same package task.

Buildr expectations are based on RSpec. RSpec is the behavior-driven development framework we use to test Buildr itself. Check the RSpec documentation if want to see all the supported matchers, or want to write your own.

Behaviour-Driven Development

Buildr supports several Behaviour-Driven Development(BDD) frameworks for testing your projects. Buildr follows each framework naming conventions, searching for files under the `src/spec/{lang}` directory.

You can learn more about each BDD framework in the Languages section.

Next, let's talk about customizing your environment and using profiles

Settings/Profiles

Environment Variables

Buildr uses several environment variables that help you control how it works. Some environment variables you will only set once or change infrequently. You can set these in your profile, OS settings or any tool you use to launch Buildr (e.g. continuous integration).

For example:

```
$ export HTTP_PROXY=http://myproxy:8080
```

There are other environment variables you will want to set when running Buildr, for example, to do a full build without running the tests:

```
$ buildr test=no
```

For convenience, when you set environment variables on the command line, the variable name is case insensitive, you can use either `test=no` or `TEST=no`. Any other way (`export`, `ENV`, etc) the variable names are case sensitive.

You can also set environment variables from within your Buildfile. For example, if you discover that building your project requires gobs of JVM heap space, and you want all other team members to run with the same settings:

```
# This project builds a lot of code.
ENV['JAVA_OPTS'] ||= '-Xms1g -Xmx1g'
```

Make sure to set any environment variables at the very top of the Buildfile, above any Ruby statement (even `require`).

Using `||=` sets the environment variable, if not already set, so it's still possible for other developers to override this environment variable without modifying the Buildfile.

Buildr supports the following environment variables:

Variable	Description
`BUILDR_ENV`	Environment name (development, production, test, etc). Another way to set this is using the `-e` command line option.
`DEBUG`	Set to `no/off` if you want Buildr to compile without debugging information (default when running the `release` task, see Compiling).
`HOME`	Your home directory.
`HTTP_PROXY`	URL for HTTP proxy server (see Specifying Repositories).
`JAVA_HOME`	Points to your JDK, required when using Java and Ant.
`JAVA_OPTS`	Command line options to pass to the JDK (e.g. `'-Xms1g'`).
`M2_REPO`	Location of the Maven2 local repository. Defaults to the `.m2` directory in your home directory (`ENV['HOME']`).
`NO_PROXY`	Comma separated list of hosts and domain that should not be proxied (see Specifying Repositories).
`TEST`	Set to `no/off` to tell Buildr to skip tests, or `all` to tell Buildr to run all tests and ignore failures (see Running Tests).
`USER`	Tasks that need your user name, for example to log to remote servers, will use this environment variable.

Buildr does not check any of the arguments in `JAVA_OPTS`. A common mistake is to pass an option like `mx512mb`, where it should be `Xmx512mb`. Make sure to double check `JAVA_OPTS`.

Some extensions may use additional environment variables, and of course, you can always add your own. This example uses two environment variables for specifying the username and password:

```
repositories.upload_to[:username] = ENV['USERNAME']
repositories.upload_to[:password] = ENV['PASSWORD']
```

Personal Settings

Some things clearly do not belong in the Buildfile. For example, the username and password you use to upload releases. If you're working in a team or on an open source project, you'd want to keep these in a separate place.

You may want to use personal settings for picking up a different location for the local repository, or use a different set of preferred remote repositories, and so forth.

The prefered way to store personal settings is to create a `.buildr/settings.yaml` file under your home directory. Settings stored there will be applied the same across all builds.

Here's an example `settings.yaml`:

```
# The repositories hash is read automatically by buildr.
repositories:

  # customize user local maven2 repository location
  local: some/path/to/my_repo

  # prefer the local or nearest mirrors
  remote:
   - https://intra.net/maven2
   - http://example.com

  relase_to:
    url: http://intra.net/maven2
    username: john
    password: secret

# You can place settings of your own, and reference them
# on buildfiles.
im:
  server: jabber.company.com
  usr: notifier@company-jabber.com
  pwd: secret
```

Later your buildfile or addons can reference user preferences using the hash returned by the `Buildr.settings.user` accessor.

```
task 'relase-notification' do
 usr, pwd, server = settings.user['im'].values_at('usr', 'pwd',
'server')
 jabber = JabberAPI.new(server, usr, pwd)
 jabber.msg("We are pleased to announce the last stable version
#{VERSION}")
end
```

Build Settings

Build settings are local to the project being built, and are placed in the `build.yaml` file located in the same directory that the `buildfile`. Normally this file would be managed by the project revision control system, so settings here are shared between developers.

They help keep the buildfile and build.yaml file simple and readable, working to the advantages of each one. Example for build settings are gems, repositories and artifacts used by that build.

```
# This project requires the following ruby gems, buildr addons
gems:
  # Suppose we want to notify developers when testcases fail.
  - buildr-twitter-notifier-addon >=1
  # we test with ruby mock objects
  - mocha
  - ci_reporter

# The artifact declarations will be automatically loaded by buildr,
so that
# you can reference artifacts by name (a ruby-symbol) on your
buildfile.
artifacts:
  spring: org.springframework:spring:jar:2.0
  log4j: log4j:log4j:jar:1.0
  j2ee: geronimo-spec:geronimo-spec-j2ee:jar:1.4-rc4

# Of course project settings can be defined here
twitter:
  notify:
    test_failure: unless-modified
    compile_failure: never
  developers:
    - joe
    - jane

jira:
  uri: https://jira.corp.org
```

When buildr is loaded, required ruby gems will be installed if needed, thus adding features like the imaginary twitter notifier addon.

Artifacts defined on `build.yaml` can be referenced on your buildfile by supplying the ruby symbol to the `Buildr.artifact` and `Buildr.artifacts` methods. The `compile.with`, `test.with` methods can also be given these names.

```
define 'my_project' do
  compile.with artifacts(:log4j, :j2ee)
  test.with :spring, :j2ee
end
```

Build settings can be retreived using the `Buildr.settings.build` accessor.

```
 task 'create_patch' do
   patch = Git.create_patch :interactive => true
   if patch && agree("Would you like to request inclusion of
#{patch}")
     jira = Jira.new( Buildr.settings.build['jira']['uri'] )  #
submit a patch
     jira.create(:improvement, patch.summary, :attachment =>
patch.blob)
   end
 end
```

Non constant settings

Before loading the Buildfile, Buildr will attempt to load two other files: the `buildr.rb` file it finds in your home directory, followed by the `buildr.rb` file it finds in the build directory.

The loading order allows you to place global settings that affect all your builds in your home directory's `buildr.rb`, but also over-ride those with settings for a given project.

Here's an example `buildr.rb`:

```
# Only I should know that
repositories.upload_to[:username] = 'assaf'
repositories.upload_to[:password] = 'supersecret'
# Search here first, it's faster
repositories.remote << 'http://inside-the-firewall'
```

Environments

One common use case is adapting the build to different environments. For example, to compile code with debugging information during development and testing, but strip it for production. Another example is using different databases for development, testing and production, or running services at different URLs.

So let's start by talking about the build environment. Buildr has a global attributes that indicates which environment it's running in, accessible from the `environment` method. You can set the current build environment in one of two ways. Using the `-e/--environment` command line option:

```
$ buildr -e test
(in /home/john/project, test)
```

Or by setting the environment variable `BUILDR_ENV`:

```
$ export BUILDR_ENV=production
$ buildr
(in /home/john/project, production)
```

Unless you tell it otherwise, Buildr assumes you're developing and sets the environment to `development`.

Here's a simple example for handling different environments within the Buildfile:

```
project 'db-module' do
  db = (environment == 'production' ? 'oracle' : 'hsql')
  resources.from(_(:source, :main, db))
end
```

We recommend picking a convention for your different environments and following it across all your projects. For example:

Environment	Use when …
development	Developing on your machine.
test	Running in test environment, continuous integration.
production	Building for release/production.

Profiles

Different environments may require different configurations, some you will want to control with code, others you will want to specify in the profiles file.

The profiles file is a YAML file called `profiles.yaml` that you place in the same directory as the Buildfile. We selected YAML because it's easier to read and edit than XML.

For example, to support three different database configurations, we could write:

```
# HSQL, don't bother storing to disk.
development:
  db: hsql
  jdbc: hsqldb:mem:devdb

# Make sure we're not messing with bigstrong.
test:
  db: oracle
  jdbc: oracle:thin:@localhost:1521:test

# The real deal.
production:
  db: oracle
  jdbc: oracle:thin:@bigstrong:1521:mighty
```

Here's a simple example for a buildfile that uses the profile information:

```
project 'db-module' do
  # Copy SQL files specific for the database we're using,
  # for example, everything under src/main/hsql.
  resources.from(_(:source, :main, profile['db']))
  # Set the JDBC URL in copied resource files (config.xml needs this).
  resources.filter :jdbc=>profile['jdbc']
end
```

The `profile` method returns the current profile, selected based on the current environment. You can get a list of all profiles by calling `profiles`.

When you run the above example in `development`, the current profile will return the hash `{ 'db'=>'hsql', 'jdbc'=>'hsqldb:mem:devdb' }`.

We recommend following conventions and using the same environments in all your projects, but sometimes the profiles end up being the same, so here's a trick you can use to keep your profiles DRY.

YAML allows you to use anchors (`&`), similar to ID attributes in XML, reference the anchored element (`*`) elsewhere, and merge one element into another (`<<`). For example:

```
# We'll reference this one as common.
development: &common
  db: hsql
  jdbc: hsqldb:mem:devdb
  resources:
    copyright: Me (C) 2008
# Merge the values from common, override JDBC URL.
test:
  <<: *common
  jdbc: hsqldb:file:testdb
```

You can learn more about YAML here, and use this handy YAML quick reference.

Languages

Java

Compiling Java

The Java compiler looks for source files in the project's `src/main/java` directory, and defaults to compiling them into the `target/classes` directory. It looks for test cases in the project's `src/test/java` and defaults to compile them into the `target/test/classes` directory.

If you point the `compile` task at any other source directory, it will use the Java compiler if any of these directories contains files with the extension `.java`.

When using the Java compiler, if you don't specify the packaging type, it defaults to JAR. If you don't specify the test framework, it defaults to JUnit.

The Java compiler supports the following options:

Option	Usage
`:debug`	Generates bytecode with debugging information. You can also override this by setting the environment variable `debug` to `off`.
`:deprecation`	If true, shows deprecation messages. False by default.
`:lint`	Defaults to false. Set this option to true to use all lint options, or specify a specific lint option (e.g. `:lint=>'cast'`).
`:other`	Array of options passed to the compiler (e.g. `:other=>'-implicit:none'`).
`:source`	Source code compatibility (e.g. '1.5').
`:target`	Bytecode compatibility (e.g. '1.4').
`:warnings`	Issue warnings when compiling. True when running in verbose mode.

Testing with Java

JUnit

The default test framework for Java projects is JUnit 4.

When you use JUnit, the dependencies includes JUnit and JMock, and Buildr picks up all test classes from the project by looking for classes that either subclass `junit.framework.TestCase`, include methods annotated with `org.junit.Test`, or test suites annotated with `org.org.junit.runner.RunWith`.

The JUnit test framework supports the following options:

Option	Value
`:fork`	VM forking, defaults to true.
`:clonevm`	If true clone the VM each time it is forked.
`:properties`	Hash of system properties available to the test case.

`:environment`	Hash of environment variables available to the test case.
`:java_args`	Arguments passed as is to the JVM.

For example, to pass properties to the test case:

```
test.using :properties=>{ :currency=>'USD' }
```

There are benefits to running test cases in separate VMs. The default forking mode is `:once`, and you can change it by setting the `:fork` option.

:fork=>	Behavior
`:once`	Create one VM to run all test classes in the project, separate VMs for each project.
`:each`	Create one VM for each test case class. Slow but provides the best isolation between test classes.
`false`	Without forking, Buildr runs all test cases in a single VM. This option runs fastest, but at the risk of running out of memory and causing test cases to interfere with each other.

You can see your tests running in the console, and if any tests fail, Buildr will show a list of the failed test classes. In addition, JUnit produces text and XML report files in the project's `reports/junit` directory. You can use that to get around too-much-stuff-in-my-console, or when using an automated test system.

In addition, you can get a consolidated XML or HTML report by running the `junit:report` task. For example:

```
$ buildr test junit:report test=all
$ firefox report/junit/html/index.html
```

The `junit:report` task generates a report from all tests run so far. If you run tests in a couple of projects, it will generate a report only for these two projects. The example above runs tests in all the projects before generating the reports.

You can use the `build.yaml` settings file to specify a particular version of JUnit or JMock. For example, to force your build to use JUnit version 4.4 and JMock 2.0:

```
junit: 4.4
jmock: 2.0
```

TestNG

You can use TestNG instead of JUnit. To select TestNG as the test framework, add this to your project:

```
test.using :testng
```

Like all other options you can set with `test.using`, it affects the projects and all its sub-projects, so you only need to do this once at the top-most project to use TestNG throughout. You can also mix TestNG and JUnit by setting different projects to use different frameworks, but you can't mix both frameworks in the same project. (And yes, `test.using :junit` will switch a project back to using JUnit)

TestNG works much like JUnit, it gets included in the dependency list along with JMock, Buildr picks test classes that contain methods annotated with `org.testng.annotations.Test`, and generates test reports in the `reports/testng` directory. At the moment we don't have consolidated HTML reports for TestNG.

The TestNG test framework supports the following options:

Option	Value
`:properties`	Hash of system properties available to the test case.
`:java_args`	Arguments passed as is to the JVM.

You can use the `build.yaml` settings file to specify a particular version of TestNG, for example, to force your build to use TestNG 5.7:

```
testng: 5.7
```

JBehave

JBehave is a pure Java BDD framework, stories and behaviour specifications are written in the Java language.

To use JBehave in your project you can select it with `test.using :jbehave`.

This framework will search for the following patterns under your project:

```
src/spec/java/**/*Behaviour.java
```

Supports the following options:

Option	Value
`:properties`	Hash of system properties available to the test case.
`:java_args`	Arguments passed as is to the JVM.

You can use the `build.yaml` settings file to specify a particular version of JBehave, for example, to force your build to use JBehave 1.0.1:

```
jbehave: 1.0.1
```

Scala

Before using Scala features, you must first set the `SCALA_HOME` environment variable to point to the root of your Scala distribution.

On Windows:

```
> set SCALA_HOME=C:\Path\To\Scala-2.7.1
```

On Linux and other Unix variants,

```
> export SCALA_HOME=/path/to/scala-2.7.1
```

The `SCALA_HOME` base directory should be such that Scala core libraries are located directly under the “lib” subdirectory, and Scala scripts are under the “bin” directory.

Compiling Scala

The Scala compiler looks for source files in the project's `src/main/scala` directory, and defaults to compiling them into the `target/classes` directory. It looks for test cases in the project's `src/test/scala` and defaults to compile them into the `target/test/classes` directory.

If you point the `compile` task at any other source directory, it will use the Scala compiler if any of these directories contains files with the extension `.scala`.

When using the Scala compiler, if you don't specify the packaging type, it defaults to JAR.

The Scala compiler supports the following options:

Option	Usage
`:debug`	Generates bytecode with debugging information. You can also override this by setting the environment variable `debug` to `off`.
`:deprecation`	If true, shows deprecation messages. False by default.
`:optimise`	Generates faster bytecode by applying optimisations to the program.
`:other`	Array of options passed to the compiler (e.g. `:other=>'-Xprint-types'`).
`:target`	Bytecode compatibility (e.g. '1.4').
`:warnings`	Issue warnings when compiling. True when running in verbose mode.

Fast Scala Compiler

You may use `fsc`, the Fast Scala Compiler, which submits compilation jobs to a compilation daemon, by setting the environment variable `USE_FSC` to `yes`. Note that `fsc` *may* cache class libraries — don't forget to run `fsc -reset` if you upgrade a library.

Rebuild detection

The Scala compiler task assumes that each `.scala` source file generates a corresponding `.class` file under `target/classes` (or `target/test/classses` for tests). The source may generate more `.class` files if it contains more than one class, object, trait or for anonymous functions and closures.

For example, `src/main/scala/com/example/MyClass.scala` should generate at least `target/classes/com/example/MyClass.class`. If that it not the case, Buildr will always recompile your sources because it will assume this is a new source file that has never been compiled before.

Testing with Scala

Buildr supports three Scala testing frameworks: ScalaTest, ScalaCheck and Specs.

Scala testing is automatically enabled if you have any `.scala` source files under `src/test/scala`. If you are not using this convention, you can explicit set the test framework by doing,

```
test.using(:scalatest)
```

The `:scalatest` test framework handles ScalaTest, Specs and ScalaCheck therefore all 3 frameworks may be used within the same project.

ScalaTest

Buildr automatically detects and runs tests that extend the `org.scalatest.Suite` interface.

A very simplistic test class might look like,

```
class MySuite extends org.scalatest.FunSuite {
  test("addition") {
    val sum = 1 + 1
    assert(sum === 2)
  }
}
```

You can also pass properties to your tests by doing `test.using :properties => { 'name'=>'value' }`, and by overriding the `Suite.runTests` method in a manner similar to:

```
import org.scalatest._

class PropertyTestSuite extends FunSuite {
  var properties = Map[String, Any]()

  test("testProperty") {
    assert(properties("name") === "value")
  }

  protected override def runTests(testName: Option[String],
    reporter: Reporter, stopper: Stopper, includes: Set[String],
    excludes: Set[String], properties: Map[String, Any])
  {
    this.properties = properties;
    super.runTests(testName, reporter, stopper,
                   includes, excludes, properties)
  }
}
```

Specs

The `:scalatest` framework currently recognizes specifications with class names ending with "Specs", e.g., org.example.StringSpecs.

A simple specification might look like this:

```
import org.specs._
import org.specs.runner._

object StringSpecs extends Specification {
  "empty string" should {
    "have a zero length" in {
      ("".length) mustEqual(0)
    }
  }
}
```

ScalaCheck

You may use ScalaCheck inside ScalaTest- and Specs-inherited classes. Here is an example illustrating checks inside a ScalaTest suite,

```
import org.scalatest.prop.PropSuite
import org.scalacheck.Arbitrary._
import org.scalacheck.Prop._

class MySuite extends PropSuite {

  test("list concatenation") {
    val x = List(1, 2, 3)
    val y = List(4, 5, 6)
    assert(x ::: y === List(1, 2, 3, 4, 5, 6))
    check((a: List[Int], b: List[Int]) => a.size + b.size == (a :::
b).size)
  }

  test(
    "list concatenation using a test method",
    (a: List[Int], b: List[Int]) => a.size + b.size == (a ::: b).size
  )
}
```

Groovy

Compiling Groovy

Before using the Groovy compiler, you must first require it on your buildfile:

```
require 'buildr/java/groovyc'
```

Once loaded, the groovyc compiler will be automatically selected if any .groovy source files are found under `src/main/groovy` directory, compiling them by default into the `target/classes` directory.

If the project has java sources in `src/main/java` they will get compiled using the groovyc joint compiler.

Sources found in `src/test/groovy` are compiled into the `target/test/classes`.

If you don't specify the packaging type, it defaults to JAR.

The Groovy compiler supports the following options:

Option	Usage
`encoding`	Encoding of source files.
`verbose`	Asks the compiler for verbose output, true when running in verbose mode.
`fork`	Whether to execute groovyc using a spawned instance of the JVM. Defaults to no.
`memoryInitialSize`	The initial size of the memory for the underlying VM, if using fork mode, ignored otherwise. Defaults to the standard VM memory setting. (Examples: `83886080`, `81920k`, or `80m`)
`memoryMaximumSize`	The maximum size of the memory for the underlying VM, if using fork mode, ignored otherwise. Defaults to the standard VM memory setting. (Examples: `83886080`, `81920k`, or `80m`)
`listfiles`	Indicates whether the source files to be compiled will be listed. Defaults to no.
`stacktrace`	If true each compile error message will contain a stacktrace.
`warnings`	Issue warnings when compiling. True when running in verbose mode.
`debug`	Generates bytecode with debugging information. Set from the debug environment variable/global option.
`deprecation`	If true, shows deprecation messages. False by default.
`optimise`	Generates faster bytecode by applying optimisations to the program.
`source`	Source code compatibility.

`target`	Bytecode compatibility.
`javac`	Hash of options passed to the ant javac task.

Testing with Groovy

EasyB

EasyB is a BDD framework using Groovy.

Specifications are written in the Groovy language, of course you get seamless Java integration as with all things groovy.

To use this framework in your project you can select it with `test.using :easyb`.

This framework will search for the following patterns under your project:

```
src/spec/groovy/**/*Behavior.groovy
src/spec/groovy/**/*Story.groovy
```

Supports the following options:

Option	Value
`:properties`	Hash of system properties available to the test case.
`:java_args`	Arguments passed as is to the JVM.
`:format`	Report format, either `:txt` or `:xml`

Ruby

Testing with Ruby

Buildr provides integration with some ruby testing frameworks, allowing you to test your Java code with state of the art tools.

Testing code is written in Ruby language, and is run by using JRuby means you have access to all your Java classes and any Java or Ruby tool out there.

Because of the use of JRuby, you will notice that running ruby tests is faster when running Buildr on JRuby, as in this case there's no need to run another JVM.

When not running on JRuby, Buildr will use the `JRUBY_HOME` environment variable to find the JRuby installation directory. If no `JRUBY_HOME` is set or it points to an empty directory, Buildr will prompt you to either install JRuby manually or let it extract it for you.

You can use the `build.yaml` settings file to specify a particular version of JRuby (defaults to `1.1.4`). For example:

```
jruby: 1.1.3
```

RSpec

RSpec is the de-facto BDD framework for ruby. It's the framework used to test Buildr itself.

To use this framework in your project you can select it with `test.using :rspec`.

This framework will search for the following patterns under your project:

```
src/spec/ruby/**/*_spec.rb
```

Supports the following options:

Option	Value
`:gems`	Hash of gems needed before running the tests. Keys are gem names, values are the required gem version. An example use of this option would be to require the ci_reporter gem to generate xml reports
`:requires`	Array of ruby files to require before running the specs
`:format`	Array of valid RSpec `--format` option values. Defaults to html report on the `reports` directory and text progress

`:output`	File path to output dump. `false` to supress output
`:fork`	Run the tests on a new java vm. (enabled unless running on JRuby)
`:properties`	Hash of system properties available to the test case.
`:java_args`	Arguments passed as is to the JVM. (only when fork is enabled)

JtestR

JtestR is a tool that makes it easier to test Java code with state of the art Ruby tools. Using JtestR you can describe your application behaviour using many testing frameworks at the same time.

To use this framework in your project you can select it with `test.using :jtestr`.

You can use the `build.yaml` settings file to specify a particular version of JtestR (defaults to `0.3.1`). For example:

```
jtestr: 0.3.1
```

To customize TestNG/JUnit versions refer to their respective section.

When selected, Buildr will configure JtestR to use your project/testing classpath and will search for the following test patterns for each framework supported by JtestR:

Framework	Patterns
RSpec	Files in `src/spec/ruby` ending with `*_spec.rb` or `*_story.rb`
TestUnit	Files in `src/spec/ruby` ending with `*_test.rb`, `*Test.rb`
Expectations	Files in `src/spec/ruby` ending with `*_expect.rb`
JUnit	Classes from `src/test/java` that either subclass `junit.framework.TestCase`, include methods annotated with `org.junit.Test`, or test suites annotated with `org.org.junit.runner.RunWith`.

TestNG	Classes from `src/test/java` annotated with `org.testng.annotations.Test`

If you create a `src/spec/ruby/jtestr_config.rb` file, it will be loaded by JtestR, just after being configured by Buildr, this way you can configure as described on JtestR guide.

If you have a `jtestr_config.rb` file, don't set `JtestR::result_handler`. Buildr uses its (`RSpecResultHandler`) so that it can know which tests succeeded/failed, this handler is capable of using RSpec formatter classes, so that you can obtain an html report or use a custom rspec formatter with `JtestR`. See the `format` option.

Supports the following options:

Option	Value
`:config`	The JtestR config file to be loaded after being configured by Buildr. Defaults to `src/spec/ruby/jtestr_config.rb`.
`:gems`	Hash of gems needed before running the tests. Keys are gem names, values are the required gem version. An example use of this option would be to require the ci_reporter gem to generate xml reports
`:requires`	Array of ruby files to require before running the specs
`:format`	Array of valid RSpec `--format` option values. Defaults to html report on the `reports` directory and text progress
`:output`	File path to output dump. `false` to supress output
`:fork`	Run the tests on a new java vm. (enabled unless running on JRuby)
`:properties`	Hash of system properties available to the test case. (only when fork is enabled)

`:java_args`	Arguments passed as is to the JVM. (only when fork is enabled)

More Stuff

Using Gems

The purpose of the buildfile is to define your projects, and the various tasks and functions used for building them. Some of these are specific to your projects, others are more general in nature, and you may want to share them across projects.

There are several mechanisms for developing extensions and build features across projects which we cover in more details in the section Extending Buildr. Here we will talk about using extensions that are distributed in the form of RubyGems.

RubyGems provides the `gem` command line tool that you can use to search, install, upgrade, package and distribute gems. It installs all gems into a local repository that is shared across your builds and all other Ruby applications you may have running. You can install a gem from a local file, or download and install it from any number of remote repositories.

RubyGems is preconfigured to use the RubyForge repository. You'll find a large number of open source Ruby libraries there, including Buildr itself and all its dependencies. RubyForge provides a free account that you can use to host your projects and distribute your gems (you can use RubyForge strictly for distribution, as we do with Buildr).

You can also set up your own private repository and use it instead or in addition to RubyForge. Use the `gem sources` command to add repositories, and the `gem server` command to run a remote repository. You can see all available options by running `gem help`.

If your build depends on other gems, you will want to specify these dependencies as part of your build and check that configuration into source control. That way you can have a specific environment that will guarantee repeatable builds, whether you're building a particular version, moving between branches, or joining an existing project. Buildr will take care of installing all the necessary dependencies, which you can then manage with the `gem` command.

Use the `build.yaml` file to specify these dependencies (see Build Settings for more information), for example:

```
# This project requires the following gems
gems:
  # Suppose we want to notify developers when testcases fail.
  - buildr-twitter-notifier-addon >=1
  # we test with ruby mock objects
  - mocha
  - ci_reporter
```

Gems contain executable code, and for that reason Buildr will not install gems without your permission. When you run a build that includes any dependencies that are not already installed on your machine, Buildr will ask for permission before installing them. On Unix-based operating systems, you will also need sudo privileges and will be asked for your password before proceeding.

Since this step requires your input, it will only happen when running Buildr interactively from the command line. In all other cases, Buildr will fail and report the missing dependencies. If you have an automated build environment, make sure to run the build once manually to install all the necessary dependencies.

When installing a gem for the first time, Buildr will automatically look for the latest available version. You can specify a particular version number, or a set of version numbers known to work with that build. You can use equality operations to specify a range of versions, for example, `1.2.3` to install only version 1.2.3, and `=> 1.2.3` to install version 1.2.3 or later.

You can also specify a range up to one version bump, for example, `~> 1.2.3` is the same as `>= 1.2.3 < 1.3.0`, and `~> 1.2` is the same as `>= 1.2.0 < 2.0.0`. If necessary, you can exclude a particular version number, for example, `~> 1.2.3 != 1.2.7`.

Buildr will install the latest version that matches the version requirement. To keep up with newer versions, execute the `gem update` command periodically. You can also use `gem outdated` to determine which new versions are available.

Most gems include documentation that you can access in several forms. You can use the `ri` command line tool to find out more about a class, module or specific method. For example:

```
$ ri Buildr::Jetty
$ ri Buildr::Jetty.start
```

You can also access documentation from a Web browser by running `gem server` and pointing your browser to http://localhost:8808. Note that after installing a new gem, you will need to restart the gem server to see its documentation.

Using Java Libraries

Buildr runs along side a JVM, using either RJB or JRuby. The Java module allows you to access Java classes and create Java objects.

Java classes are accessed as static methods on the `Java` module, for example:

```
str = Java.java.lang.String.new('hai!')
str.toUpperCase
=> 'HAI!'
Java.java.lang.String.isInstance(str)
=> true
Java.com.sun.tools.javac.Main.compile(args)
```

The `classpath` attribute allows Buildr to add JARs and directories to the classpath, for example, we use it to load Ant and various Ant tasks, code generators, test frameworks, and so forth.

When using an artifact specification, Buildr will automatically download and install the artifact before adding it to the classpath.

For example, Ant is loaded as follows:

```
Java.classpath << 'org.apache.ant:ant:jar:1.7.0'
```

Artifacts can only be downloaded after the Buildfile has loaded, giving it a chance to specify which remote repositories to use, so adding to classpath does not by itself load any libraries. You must call `Java.load` before accessing any Java classes to give Buildr a chance to load the libraries specified in the classpath.

When building an extension, make sure to follow these rules:

1. Add to the `classpath` when the extension is loaded (i.e. in module or class definition), so the first call to `Java.load` anywhere in the code will include the libraries you specify.
2. Call `Java.load` once before accessing any Java classes, allowing Buildr to set up the classpath.
3. Only call `Java.load` when invoked, otherwise you may end up loading the JVM with a partial classpath, or before all remote repositories are listed.
4. Check on a clean build with empty local repository.

Nailgun

Nailgun is a client, protocol, and server for running Java programs from the command line without incurring the JVM startup overhead. Nailgun integration is available only when running Buildr within JRuby.

Buildr provides a custom nailgun server, allowing you to start a single JVM and let buildr create a queue of runtimes. These JRuby runtimes can be cached (indexed by buildfile path) and are automatically reloaded when the buildfile has been modified. Runtime caching allows you to execute tasks without spending time creating the buildr environment.

Start the BuildrServer by executing

```
$ jruby -S buildr -rbuildr/nailgun nailgun:start
```

Server output will display a message when it becomes ready, you will also see messages when the JRuby runtimes are being created, or when a new buildr environment is being loaded on them. After the runtime queues have been populated, you can start calling buildr as you normally do, by invoking the $NAILGUN_HOME/ng binary:

```
# on another terminal, change directory to a project.
# if this project is the same nailgun:start was invoked on, it's
# runtime has been cached, so no loading is performed unless
# the buildfile has been modified. otherwise the buildfile
# will be loaded on a previously loaded fresh-buildr runtime
# and it will be cached.
cd /some/buildr/project
ng nailgun:help                      # display nailgun help
ng nailgun:tasks                     # display overview of ng tasks
ng clean compile               # just invoke those two tasks
```

Some nailgun tasks have been provided to manage the cached runtimes, to get an overview of them execute the `nailgun:tasks` task.

Be sure to read the nailgun help by executing the `nailgun:help` task.

Growl, Qube

For OS X users, Buildr supports Growl out of the box to send "completed and “failed” notifications to the user.

For other platforms or if you want to notify the user differently, Buildr offers two extension points:

- `Buildr.application.on_completion`
- `Buildr.application.on_failure`

Here is an example using these extension points to send notifications using Qube:

```
# Send notifications using Qube
notify = lambda do |type, title, message|
  param = case type
    when 'completed'; '-i'
    when 'failed'; '-e'
    else '-i'
  end
  system "qube #{param} #{title.inspect} #{message.inspect}"
end

Buildr.application.on_completion do |title, message|
  notify['completed', title, message]
end
Buildr.application.on_failure do |title, message, ex|
  notify['failed', title, message]
end
```

You can place this code inside `buildr.rb` in your home directory.

Eclipse, IDEA

If you’re using Eclipse, you can generate `.classpath` and `.project` from your Buildfile and use them to create a project in your workspace:

```
$ buildr eclipse
```

The `eclipse` task will generate a `.classpath` and `.project` file for each of projects (and sub-project) that compiles source code. It will not generate files for other projects, for examples, projects you use strictly for packaging a distribution, or creating command line scripts, etc.

If you add a new project, change the dependencies, or make any other change to your Buildfile, just run the `eclipse` task again to re-generate the Eclipse project files.

If you prefer IntelliJ IDEA, you can always:

```
$ buildr idea
```

It will generate a `.iml` file for every project (or subproject) and a `.ipr` that you can directly open for the root project. To allow IntelliJ Idea to resolve external dependencies properly, you will need to add a `M2_REPO` variable pointing to your Maven2 repository directory (`Settings / Path Variables`).

If you're using IDEA 7 or later, use the `buildr idea7x` task instead. This task creates the proper `.ipr` and `.iml` files for IDEA version 7. It includes the `-7x` suffix in the generated files, so you can use the `idea` and `idea7x` tasks side by side on the same project.

Also, check out the Buildr plugin for IDEA (IDEA 7 and later). Once installed, open your project with IDEA. If IDEA finds that you have Buildr installed and finds a buildfile in the project's directory, it will show all the tasks available for that project. To run a task, double-click it. When the task completes, IDEA will show the results in the Buildr Output window.

Cobertura, Emma, JDepend

You can use Cobertura or Emma to instrument your code, run the tests and create a test coverage report in either HTML or XML format.

There are two tasks for each tool, both of which generate a test coverage report in the `reports/cobertura` (respectively `reports/emma`) directory. For example:

```
$ buildr test cobertura:html
```

As you can guess, the other tasks are `cobertura:xml`, `emma:html` and `emma:xml`.

If you want to generate a test coverage report only for a specific project, you can do so by using the project name as prefix to the tasks.

```
$ buildr subModule:cobertura:html
```

Each project can specify which classes to include or exclude from cobertura instrumentation by giving a class-name regexp to the `cobertura.include` or `cobertura.exclude` methods:

```
define 'someModule' do
  cobertura.include 'some.package.*'
  cobertura.include /some.(foo|bar).*/
  cobertura.exclude 'some.foo.util.SimpleUtil'
  cobertura.exclude /*.Const(ants)?/i
end
```

Emma has `include` and `exclude` methods too, but they take glob patterns instead of regexps.

You can use JDepend on to generate design quality metrics. There are three tasks this time, the eye candy one:

```
$ buildr jdepend:swing
```

The other two tasks are `jdepend:text` and `jdepend:xml`.

We want Buildr to load fast, and not everyone cares for these tasks, so we don't include them by default. If you want to use one of them, you need to require it explicitly. The proper way to do it in Ruby:

```
require 'buildr/cobertura'
require 'buildr/emma'
require 'buildr/jdepend'
```

You may want to add those to the Buildfile. Alternatively, you can use these tasks for all your projects without modifying the Buildfile. One convenient method is to add these lines to the `buildr.rb` file in your home directory.

Another option is to require it from the command line (`--require` or `-r`), for example:

```
$ buildr --require buildr/jdepend jdepend:swing
$ buildr -rbuildr/cobertura cobertura:html
```

Anything Ruby Can Do

Buildr is Ruby code. That's an implementation detail for some, but a useful features for others. You can use Ruby to keep your build scripts simple and DRY, tackle ad hoc tasks and write reusable features without the complexity of "plugins".

We already showed you one example where Ruby could help. You can use Ruby to manage dependency by setting constants and reusing them, grouping related dependencies into arrays and structures.

You can use Ruby to perform ad hoc tasks. For example, Buildr doesn't have any pre-canned task for setting file permissions. But Ruby has a method for that, so it's just a matter of writing a task:

```
bins = file('target/bin'=>FileList[_('src/main/dist/bin/*')]) do
|task|
  mkpath task.name
  cp task.prerequisites, task.name
  chmod 0755, FileList[task.name + '/*.sh'], :verbose=>false
end
```

You can use functions to keep your code simple. For example, in the ODE project we create two binary distributions, both of which contain a common set of files, and one additional file unique to each distribution. We use a method to define the common distribution:

```
def distro(project, id)
  project.package(:zip, :id=>id).path("#{id}-#{version}").tap do |zip|
    zip.include meta_inf + ['RELEASE_NOTES', 'README'].map { |f|
path_to(f) }
    zip.path('examples').include project.path_to(:source, :examples),
:as=>'.'
    zip.merge project('ode:tools-bin').package(:zip)
    zip.path('lib').include artifacts(COMMONS.logging, COMMONS.codec,
      COMMONS.httpclient, COMMONS.pool, COMMONS.collections, JAXEN,
SAXON,
      LOG4J, WSDL4J, XALAN, XERCES)
    project('ode').projects('utils', 'tools', 'bpel-compiler',
'bpel-api',
      'bpel-obj', 'bpel-schemas').map(&:packages).flatten.each do
|pkg|
        zip.include(pkg.to_s, :as=>"#{pkg.id}.#{pkg.type}",
:path=>'lib')
      end
    yield zip
  end
end
```

And then use it in the project definition:

```
define 'distro-axis2' do
  parent.distro(self, "#{parent.id}-war") { |zip|
    zip.include project('ode:axis2-war').package(:war),
:as=>'ode.war' }
end
```

Ruby's functional style and blocks make some task extremely easy. For example, let's say we wanted to count how many source files we have, and total number of lines:

```
sources = projects.map { |prj| prj.compile.sources.
  map { |src| FileList["#{src}/**/*.java"] } }.flatten
puts "There are #{source.size} source files"
lines = sources.inject(0) { |lines, src| lines +=
File.readlines(src).size }
puts "That contain #{lines} lines"
```

Extending Buildr

Organizing Tasks

A couple of things we learned while working on Buildr. Being able to write your own Rake tasks is a very powerful feature. But if you find yourself doing the same thing over and over, you might also want to consider functions. They give you a lot more power and easy abstractions.

For example, we use OpenJPA in several projects. It's a very short task, but each time I have to go back to the OpenJPA documentation to figure out how to set the Ant MappingTool task, tell Ant how to define it. After the second time, you're recognizing a pattern and it's just easier to write a function that does all that for you.

Compare this:

```
file('derby.sql') do
  REQUIRES = [
    'org.apache.openjpa:openjpa-all:jar:0.9.7-incubating',
    'commons-collections:commons-collections:jar:3.1',
    . . .
    'net.sourceforge.serp:serp:jar:1.11.0' ]
  ant('openjpa') do |ant|
    ant.taskdef :name=>'mapping',
      :classname=>'org.apache.openjpa.jdbc.ant.MappingToolTask',
      :classpath=>REQUIRES.join(File::PATH_SEPARATOR)
    ant.mapping :schemaAction=>'build', :sqlFile=>task.name,
      :ignoreErrors=>true do
        ant.config :propertiesFile=>_(:source, :main, :sql,
'derby.xml')
        ant.classpath :path=>projects('store', 'utils' ).
          flatten.map(&:to_s).join(File::PATH_SEPARATOR)
    end
  end
end
```

To this:

```
file('derby.sql') do
  mapping_tool :action=>'build', :sql=>task.name,
    :properties=>_(:source, :main, :sql, 'derby.xml'),
    :classpath=>projects('store', 'utils')
end
```

I prefer the second. It's easier to look at the Buildfile and understand what it does. It's easier to maintain when you only have to look at the important information.

But just using functions is not always enough. You end up with a Buildfile containing a lot of code that clearly doesn't belong there. For starters, I recommend putting it in the `tasks` directory. Write it into a file with a `.rake` extension and place that in the `tasks` directory next to the Buildfile. Buildr will automatically pick it up and load it for you.

If you want to share these pre-canned definitions between projects, you have a few more options. You can share the `tasks` directory using SVN externals. Another mechanism with better version control is to package all these tasks, functions and modules into a Gem and require it from your Buildfile. You can run your own internal Gem server for that.

For individual task files, you can also use Sake for system-wide Rake tasks deployment.

Creating Extensions

The basic mechanism for extending projects in Buildr are Ruby modules. In fact, base features like compiling and testing are all developed in the form of modules, and then added to the core Project class.

A module defines instance methods that are then mixed into the project and become instance methods of the project. There are two general ways for extending projects. You can extend all projects by including the module in Project:

```
class Project
  include MyExtension
end
```

You can also extend a given project instance and only that instance by extending it with the module:

```
define 'foo' do
  extend MyExtension
end
```

Some extensions require tighter integration with the project, specifically for setting up tasks and properties, or for configuring tasks based on the project definition. You can do that by adding callbacks to the process.

The easiest way to add callbacks is by incorporating the Extension module in your own extension, and using the various class methods to define callback behavior.

Method	Usage
`first_time`	This block will be called once for any particular extension. You can use this to setup top-level and local tasks.

`before_define`	This block is called once for the project with the project instance, right before running the project definition. You can use this to add tasks and set properties that will be used in the project definition.
`after_define`	This block is called once for the project with the project instance, right after running the project definition. You can use this to do any post-processing that depends on the project definition.

This example illustrates how to write a simple extension:

```
module LinesOfCode
  include Extension

  first_time do
    # Define task not specific to any projet.
    desc 'Count lines of code in current project'
    Project.local_task('loc')
  end

  before_define do |project|
    # Define the loc task for this particular project.
    define_task 'loc' do |task|
      lines = task.prerequisites.map { |path| Dir["#{path}/**/*"]
}.flatten.uniq.
        inject(0) { |total, file| total + File.readlines(file).count }
      puts "Project #{project.name} has #{lines} lines of code"
    end
  end

  after_define do |project|
    # Now that we know all the source directories, add them.
    task('loc'=>compile.sources + compile.test.sources)
  end

  # To use this method in your project:
  #   loc path_1, path_2
  def loc(*paths)
    task('loc'=>paths)
  end

end

class Buildr::Project
  include LinesOfCode
end
```

Using Alternative Layouts

Buildr follows a common convention for project layouts: Java source files appear in `src/main/java` and compile to `target/classes`, resources are copied over from `src/main/resources` and so forth. Not all projects follow this convention, so it's now possible to specify an alternative project layout.

The default layout is available in `Layout.default`, and all projects inherit it. You can set `Layout.default` to your own layout, or define a project with a given layout (recommended) by setting the `:layout` property. Projects inherit the layout from their parent projects. For example:

```
define 'foo', :layout=>my_layout do
  ...
end
```

A layout is an object that implements the `expand` method. The easiest way to define a custom layout is to create a new `Layout` object and specify mapping between names used by Buildr and actual paths within the project. For example:

```
my_layout = Layout.new
my_layout[:source, :main, :java] = 'java'
my_layout[:source, :main, :resources] = 'resources'
```

Partial expansion also works, so you can specify the above layout using:

```
my_layout = Layout.new
my_layout[:source, :main] = ''
```

If you need anything more complex, you can always subclass `Layout` and add special handling in the `expand` method, you'll find one such example in the API documentation.

The built-in tasks expand lists of symbols into relative paths, using the following convention:

Path	Expands to
`:source, :main, <lang/usage>`	Directory containing source files for a given language or usage, for example, `:java`, `:resources`, `:webapp`.
`:source, :test, <lang/usage>`	Directory containing test files for a given language or usage, for example, `:java`, `:resources`.
`:target, :generated`	Target directory for generated code (typically source code).

`:target, :main, <lang/usage>`	Target directory for compiled code, for example, `:classes`, `:resources`.
`:target, :test, <lang/usage>`	Target directory for compile test cases, for example, `:classes`, `:resources`.
`:reports, <framework/usage>`	Target directory for generated reports, for example, `:junit`, `:coverage`.

All tasks are encouraged to use the same convention, and whenever possible, we recommend using the project's `path_to` method to expand a list of symbols into a path, or use the appropriate path when available. For example:

```
define 'bad' do
  # This may not be the real target.
  puts 'Compiling to ' + path_to('target/classes')
  # This will break with different layouts.
  package(:jar).include 'src/main/etc/*'
end

define 'good' do
  # This is always the compiler's target.
  puts 'Compiling to ' + compile.target.to_s
  # This will work with different layouts.
  package(:jar).include path_to(:source, :main, :etc, '*')
end
```

Recipes

Commond recipes for Buildr, collected from the mailing list.

Creating a classpath

For Java, the classpath argument is simply a list of paths joined with an OS-specific path separator:

```
cp = paths.join(File::PATH_SEPARATOR)
```

This assumes `paths` points to files and/or directories, but what if you have a list of artifact specifications? You can turn those into file names in two steps. First, use `artifacts` to return a list of file tasks that point to the local repository:

```
tasks = Buildr.artifacts(specs)
```

Next, map that list of tasks into list of file names (essentially calling `name` on each task):

```
paths = tasks.map(&:name)
```

This works as long as the artifacts are already in your local repository, otherwise they can't be found, but you can ask Buildr to download them by calling `invoke` on each of these tasks:

```
tasks = Buildr.artifacts(specs).each(&:invoke)
```

So let's roll this all into a single line:

```
cp =
Buildr.artifacts(specs).each(&:invoke).map(&:name).join(File::PATH_SEPARATOR)
```

Keeping your Profiles.yaml file DRY

YAML allows you to use anchors (&), similar to ID attributes in XML, and reference them later on (*). For example, if you have two profiles that are identical, you can tell YAML that one is an alias for the other:

```
development: &common
  db: oracle
  port: 8080
test: *common
production: *common
```

If you have two elements that are almost identical, you can merge the values of one element into another (<<), for example:

```
development: &common
  db: hsql
  jdbc: hsqldb:mem:devdb
test:
  <<: *common
  jdbc: hsqldb:file:testdb
```

Speeding JRuby

When using JRuby you will notice that Buildr takes a few seconds to start up. To speed it up, we recommend switching to Java 1.6 and running the JVM in client mode:

```
$ export JAVA_OPTS=-client
```

Continuous Integration with Atlassian Bamboo

This recipe outlines how to configure a new Bamboo project to use Buildr. The following steps assume that you have logged-on to Bamboo as an Administrator.

1. Configure a Builder

- Select the Administration tab from the Bamboo toolbar.
- Select the Builders area (first option) from the Administration menu.
- Using the Add Builder dialog, configure a custom builder for Buildr with the following options:
    - Label: `buildr`
    - Type: `Custom Command`
    - Path: `/path/to/buildr` (typically "/usr/bin/buildr")

2. Create a Plan

- Select the Create Plan tab from the Bamboo toolbar to enter the Create Plan wizard.
- In "1. Plan Details", define your build plan including project name, project key, build plan name and build plan key.
- In "2. Source Repository", specify your source code repository settings (CVS or SVN).
- In "3. Builder Configuration", specify the "buildr" builder that you defined above, along with the following:
    - Argument: `"test=all"` (ensures that all tests are run through even if failures are encountered)
    - Test Results Directory: `"**/reports/junit/*.xml"` (or your path to test results, if different).
- The remaining wizard sections may be either skipped or configured with your preferred settings.

3. Trigger a Build

A build should occur automatically at the point of project creation. It can also be manually triggered at any time

- Navigate to the project summary page (located at: `http://YOUR_BAMBOO_URL/browse/PROJECTKEY-YOURPLAN`).
- Click on the small arrow to the left of the label "Build Actions"
- Select "Checkout and Build" from the pop-up menu to force a build.

The project page will contain full status information for previous builds and the results tabs will integrate the results from your JUnit tests.

Troubleshooting

Common troubleshooting tips collected from the mailing list.

Running out of heap space

You can give the JVM more heap space by setting the `JAVA_OPTS` environment variables. This environment variable provides arguments for staring the JVM. For example, to set the heap space to 1GB:

```
$ export "JAVA_OPTS=-Xms1g -Xmx1g"
$ buildr compile
```

If you’re sharing the build with other developers, you’ll want to specify these options in the Buildfile itself. You can set the environment variable within the Buildfile, but make sure to do so at the very top of the Buildfile.

For example:

```
ENV['JAVA_OPTS'] = '-Xms1g -Xmx1g'
```

RJB fails to compile

On Linux, BSD and Cygwin, RJB locates the JDK headers files — which it uses to compile a native C extension — based on the `JAVA_HOME` environment variable. Make sure `JAVA_HOME` points to the JDK, not JRE.

If you are using `sudo gem install`, note that some environments do not pass the `JAVA_HOME` environment variable over to `sudo`. To get around this, run `gem` with the `env JAVA_HOME=$JAVA_HOME` option:

```
$ sudo env JAVA_HOME=$JAVA_HOME gem install buildr
```

Segmentation Fault when running Java code

This is most likely a JVM inconsistency, for example, when part of the RJB library uses JDK 1.6, the other part uses JDK 1.5.

During installation RJB builds a native C extension using header files supplied by the JVM, and compiles a Java bridge class using the Javac. It is possible for RJB to use two different versions of the JVM, for example, if `JAVA_HOME` points to JDK 1.5, but `/usr/bin/javac` points to JDK 1.6.

Make sure `JAVA_HOME` and `/usr/bin/javac` both point to the same JDK:

```
echo $JAVA_HOME
ls -l /usr/bin/javac
```

Note: It seems that RJB works with Java 6, except when it doesn't, and for a few people it doesn't. In that case, either switch to Java 1.5, or simply run Buildr on JRuby using Java 6.

Bugs resulting from a dangling comma or period

Ruby statements don't need a delimiter and can span multiple lines, which can lead to bugs resulting from dangling commas and periods left at the end of the line. For example:

```
compile.with 'org.apache.axis2:axis2:jar:1.2',
test.with 'log4j:log4j:jar:1.1'
```

This is actually a single method call with two arguments, separated by a comma. The second argument is the result of calling `test.with`, and makes the test task a pre-requisite of the compile task, leading to a circular dependency.

As you can imagine this happens usually after editing, specifically for commas and periods which are small enough that you won't notice them from a cursory glance at the code, so if all else fails, search for lines that end with one of these characters.

Missing POM breaks transitive dependencies

Occasionally, artifacts are deployed to remote repositories with missing or broken POMs. Buildr will fail when attempting to resolve transitive dependencies with broken or missing POMs.

In this particular case, failing is doing the right thing. There's no way for Buildr to tell whether the POM is nowhere to be found, or just a temporary problem accessing the remote server.

If you can determine that the POM file is missing you can work around the problem in three ways. If you published the artifact, make the release again, getting it to upload the missing files.

If the source repository is not under your control, but you are also using your own repository for the project, you can always create a dummy POM in your own repository. Buildr will attempt to download the file from either repository, using the first file it finds.

Alternatively, you can make Buildr create a dummy POM file in the local repository, instead of downloading it from a remote repository. This example creates a dummy POM for Axis JAX-RPC:

```
artifact 'axis:axis-jaxrpc:pom:1.3' do |task|
  write task.name, <<-POM
    <project>
      <modelVersion>4.0.0</modelVersion>
      <groupId>axis</groupId>
      <artifactId>axis-jaxrpc</artifactId>
      <version>1.4</version>
    </project>
  POM
end
```

Buildr fails to run after install with a "stack level too deep (SystemStackError)" error

A particular quirk of an existing Ruby setup can cause problems when running Buildr. If a system already has several Ruby directories that are in the `PATH`, it is often nice (appropriate?) to have them in `RUBYLIB` as well (to be able to require them). If there are several of them a user may decide that `RUBYLIB=$PATH` is a good way to handle this (or some less automated method that has the same effect).

The culprit is having the Gem's binary directory show up in `RUBYLIB`. For example, Buildr's `bin/buildr` includes this line:

```
require 'buildr'
```

Under normal circumstances, this tells RubyGems to load `buildr.rb` from the Gem's library directory. When `RUBYLIB` points to the Gem's `bin` directory, it ends up loading itself repeatedly.

To solve this, remove Buildr's `bin` directory from `RUBYLIB`. Removing all directories that you don't actually need is better (other Gems may have the same problem).

Contributing

Buildr is a community effort, and we welcome all contributors. Here's your chance to get involved and help your fellow developers.

Getting involved

All our discussions are done in the open, over email, and that would be the first place to look for answers, raise ideas, etc. For bug reports, issues and patches, see below.

Mailing Lists

We run two mailing lists, the buildr-user mailing list for developers working with Buildr, that would be you if you're using Buildr or interested in using it. There's the buildr-dev mailing list for talking about development of Buildr itself, and commits mailing list for following SVN commits and JIRA issues.

Check the mailing lists page for more information on subscribing, searching and posting to the mailing list.

Bugs (aka Issues)

We really do try to keep bugs to a minimum, and anticipate everything you'll ever want to do with Buildr. We're also, not perfect. So you may have found a bug, or have an enhancement in mind, or better yet, a patch to contribute. Here's what you can do.

If it's a bug, enhancement or patch, add it to JIRA. For trivial stuff, that's good enough.

If it needs more attention, start a discussion over on the mailing list. We will still use JIRA to log the progress, but the mailing list is a better place for talking things through.

When reporting a bug, please tell us which version of Ruby, Buildr and Java you are using, and also which operating system you are on:

```
$ ruby --version
$ buildr --version
$ java --version
```

Community Wiki

Our community Wiki is available at http://cwiki.apache.org/confluence/display/BUILDR/Index.

Contributing Code

Yes, please.

If you have a patch to submit, do it through JIRA. We want to make sure Apache gets the right to use your contribution, and the JIRA upload form includes a simple contribution agreement. Lawyer not included.

The Perfect Patch

If you want to get your patch accepted quickly:

1. Provide a good summary of the bug/fix. We use that to decide which issue we can do quickly, and also copy and paste it into the changelog.
2. Provide short explanation of what failed, under what conditions, why, and what else could be affected by the change (when relevant). The helps us understand the problem and move on to the next step.
3. Provide a patch with relevant specs, or a fix to incomplete/broken specs. First thing we have to do is replicate the problem, before applying the change, and then make sure the change fixes that problem. And we need to have those specs in there, they make sure we don't accidentally break it again in the future.
4. Provide a patch with the fix/change itself. Keep it separate from the specs, so it's easy to apply them individually.

If you don't know how to fix it, but can at least write a spec for the correct behavior (which, obviously would fail), do just that. A spec is preferred to a fix.

Working on a new feature?

If you want to work on a cool new feature, but not quite ready to submit a patch, there's still a way you can get the Buildr community involved. We're experimenting with using Git for that. You can use Git to maintain a fork of Buildr that can keep up with changes in the main branch (tip: use `git rebase`), while developing your own changes/features on it.

That way you can get other people involved, checking out the code, and eventually merge it back with the main branch. Check out the Git section below and the post Git forking for fun and profit.

Living on the edge

Did we mention Buildr is an open source project? In fact, when you install Buildr you get all the source code, documentation, test case and everything you need to use it, extend it and patch it. Have a look in your Gem directory.

SVN

But if you want to work with the latest and greatest, you'll want to check out Buildr from SVN:

```
$ svn co http://svn.apache.org/repos/asf/incubator/buildr/trunk buildr
```

You can also browse the Buildr repository.

Git

Not a fan SVN? We understand. You can also grab a copy of Buildr from GitHub:

```
$ git clone git://github.com/vic/buildr.git
```

If you want to learn more about Git, you can start by watching Scott Chacon's Git presentation (PDF), or any of the Git screencasts. For more, there's also the Git Internals book.

And keep this Git cheat sheet close at hand. Very useful.

Note: The GitHub repository is maintained by contributors to this project, but is **not** an official Apache repository. To obtain Buildr from the official Apache repository, consider using `git-svn` instead.

Working with Source Code

To install Buildr from the source directory:

```
$ cd buildr
$ rake setup install
```

When using Buildr for JRuby:

```
$ cd buildr
$ jruby -S rake setup install
```

The *setup* task takes care of installing all the necessary dependencies used for building, testing and running Buildr. Once in a while we upgrade or add new dependencies, if you're experiencing a missing dependency, simply run `rake setup` again.

The *install* task creates a Gem in your working directory (*pkg/*) and install it in your local repository. Since Ruby Gems uses version numbers to detect new releases, if you installed Buildr this way and want to upgrade to the latest official release, you need to use `gem install buildr` rather than `gem upgrade`.

Both *setup* and *install* tasks use the `sudo` command on platforms that require it (i.e. not Windows), so there's no need to run `sudo rake` when working with the Buildr source code.

Using development build

Occasionally we'll make development builds from the current code in trunk/head. We appreciate if you can take the time to test those out and report any bugs. To install development builds, use the Gem repository at `people.apache.org/~assaf/buildr/snapshot`:

```
gem source --add http://people.apache.org/~assaf/buildr/snapshot/
```

Since Ruby Gems uses version numbers to detect new releases, if you installed Buildr from a snapshot and want to upgrade to a newer snapshot or the latest official release, you need to use `gem install buildr` rather than `gem upgrade`.

If you want to go back to using the RubyForge releases:

```
gem source --remove http://people.apache.org/~assaf/buildr/snapshot/
gem install buildr
```

Tested and Documented

Two things we definitely encourage!

Testing/Specs

Obviously we won't turn down patches, but we'll love you even more if you include a test case. One that will fail without the patch, and run successfully with it. If not for our love, then think of the benefit to you: once we add that test case, we won't accidentally break that feature in the next release.

We test using RSpec, a Behavior-Driven Development test framework. The main difference between RSpec and xUnit is that RSpec helps you formulate test cases in terms of specifications: you describe how the code should behave, and run RSpec to make sure it matches that specification.

You can run an individual specifications using the `spec` command, for example:

```
$ spec spec/compiler_spec.rb
$ spec spec/compiler_spec.rb -l 409
```

The first command will run all the specifications in `compiler_spec`, the second command will run only the specification identified by line 409 of that file. You can use line numbers to point at a particular specification (lines starting with `it`), or set of specifications (lines starting with `describe`). You can also use the `-e` command line option to name a particular specification.

To make sure your change did not break anything else, you can run all the specifications (be patient, we have a lot of these):

```
$ rake spec
```

If you get any failures, you can use `rake failed` to run only the failed specs, and repeat until there are no more failed specs to run. The list of failed specs is stored in the file *failed*.

We always `rake spec` before making a release.

For full test coverage:

```
$ rake coverage
```

Specification and coverage reports are HTML files you can view with a Web browser, look for them in the *reports* directory. You can also check out the RSpec report and test coverage we publish with each release.

Documentation

Yes, we do make typos, spelling errors and sometimes we write things that don't make sense, so if you find a documentation bug, or want to help make the documentation even better, here's the way to do it.

For simple typos and quick fixes, just send a message to the mailing list or log an issue in JIRA.

If you end up rewriting a significant piece of text, or add new documentation (you rock!), send a patch. Making documentation patches is fairly easy. All the documentation is generated from text files in the `doc/pages` directory, so all you need to do is check it out from SVN, edit, and `svn diff` to create a patch.

We use Textile as the markup language, it takes all of a few minutes to learn, it's intuitive to use, and produces clean HTML. You can learn it all in a few minutes from the Textile Reference Manual. Also check out the Textile Quick Reference.

You can always check the documentation to see which conventions we use, and also a couple of extensions we have for styling source code (with syntax highlighting!) and handling footnotes. The table of contents is auto-generated form H1/H2 headers.

The tool we use for this is called Docter, which we developed specifically for Buildr, and use to create the Web site and printable PDF. If you want to try it out you'll need to first `gem install docter`. To generate a copy of the Web site, simple run `rake html` .

If you're thinking of editing the docs, and using `rake html` to see what the HTML looks like, you may want to try something simpler. Start by running the Docter Web server with `rake docter` and then point your browser at `http://localhost:3000`. To see your edits, simply refresh the page.

Generating the PDF is a bit more tricky, we use the HTML in combination with print media CSS stylesheets and run them through the wonderful PrinceXML, so you'll need to install PrinceXML
first before you can `rake pdf`.

Contributors

Here is the list of people who are actively working and committing on Buildr:

Assaf Arkin (assaf at apache.org)

Alex Boisvert

Came to Buildr as a refuge from the Maven Uncertainty Principle. Alex has been working mostly on the Scala integration and believes Ruby scripting is a great complement to statically typed languages.

Matthieu Riou

Victor Hugo Borja (vborja at apache.org)

Currently a Java Developer at http://jwmsolutions.com, Victor has been enjoying and using Apache's software since 1999 when he started with Java, now he prefers programming Ruby and is happy to help on Apache's first ruby project.

Lacton (lacton at apache.org)

A test-infected developer since 2001, Lacton yearns for a development infrastructure that would shorten feedback loops so much that testing, building, refactoring and committing would feel as easy and natural as breathing air.