



# buildr

- Quick Start ..... 5
- Installing and Running ..... 14
- Projects ..... 25
- Building ..... 38
- Artifacts ..... 49
- Packaging ..... 57
- Testing ..... 77
- Settings & Profiles ..... 86
- Languages ..... 96
- More Stuff ..... 117
- Extending Buildr ..... 145
- Contributing ..... 153



Copyright 2007-2009 Apache Buildr

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Daniel Spiewak:

*If you think about it, the question isn't "Why use Buildr?", it's really "Why use anything else?" The advantages afforded by Buildr are so substantial, I really can't see myself going with any other tool, at least not when I have a choice.*

Tristan Juricek:

*That's still the strongest sell: it builds everything I need, and as I've needed more, I just got things working without a lot of fuss.*

Matthieu Riou:

*We used to rely on Ant, with a fairly extensive set of scripts. It worked but was expensive to maintain. The biggest mistake afterward was to migrate to Maven2. I could write pages of rants explaining all the problems we ran into and we still ended up with thousands of lines of XML.*

Martin Grotzke:

*The positive side effect for me as a java user is that I learn a little ruby, and that's easy but lots of fun... :-)*



# Quick Start

---

Your First Project.....	5
Dependencies .....	8
Testing.....	10
Custom Tasks.....	11
Summary .....	13

This quick start guide is meant to be a *very* simple introduction to Buildr and its most basic concepts. However, despite its basic level, we will still cover most of the concepts you will ever need to be productive with Buildr. We will leave out some important things (like [sub-projects](#)), and we will over-simplify some other concepts (such as [artifacts](#)). Nevertheless, most Buildr projects never need to go beyond the techniques contained within these pages.

**No knowledge of Ruby is assumed.** Buildr is designed to be a very intuitive, very easy-to-use tool. You can create buildfiles which describe incredibly intricate projects, write custom tasks which do things far beyond Ant, and still never need to pick up more than a smattering of Ruby syntax. With that said, if you do know Ruby, Buildr’s DSL will seem very natural and welcoming. We do assume that you have already [downloaded and installed](#) Buildr and are ready to put the tool to good use.

## Your First Project

---

Much like Maven, Buildr is oriented around projects and tasks. You define your project in a concise, declarative fashion and most common tasks (such as compilation and testing) will be made available to you “at no extra charge”. Most of the project definition is contained within the *buildfile* — or *Buildfile*, if you’re really in love with the Make convention — a single file sitting at the root of your project. A project definition does not need to be any more complicated than the following:

```
define 'killer-app'
```

## Compiling

Of course, this isn't really giving Buildr much information. What it can't learn from the buildfile, Buildr will figure out by inspecting your directory structure. Java sources are expected to exist within the `src/main/java/` directory. If Buildr finds these sources, it will automatically configure the compilation to source that directory, depositing the results in the `target/classes/` directory (all under the project directory of course). We can run this compilation using the following command:

```
$ buildr compile
```

Information about the classpath and dependencies is described [later on](#).



By default, Buildr projects assume the Java language and the `src/main/java/` source directory. You can also have projects in the Scala or Groovy language (both languages support joint compilation with Java). To use Scala, place your `.scala` files in the `src/main/scala/` directory and include the following invocation at the head of your buildfile: `require 'buildr/scala'` Similarly, Groovy expects sources in the `src/main/groovy/` directory and necessitates `require 'buildr/groovy'` (see [languages](#) for more information).

The `compile` task will also detect *any* files which exist under the `src/main/resources/` directory. These resources are copied verbatim to the `target/resources/` directory as part of the compilation task. Buildr also performs some basic change detection to minimize compilation. If your source files haven't changed since the last compile, then they will not be recompiled.

## Packaging

At some point, we're going to want to wrap up our classes and produce a single JAR file for distribution. Buildr can certainly help us with this, but we are going to need to provide it with a little bit more information. Specifically, we need to say the type of package to produce (e.g. `:jar`, `:war`, etc) as well as the current version of our project. This information is placed within the buildfile:

```
define 'killer-app' do
  project.version = '0.1.0'
  package :jar
end
```

The `project.version` attribute can be any value you like. Even non-numeric versions are perfectly acceptable (e.g. `'ikj-0.3.1-E'`). This version — coupled with the packaging information — will be used to generate a JAR file: `killer-app-0.1.0.jar`. As would be expected, this file is placed within the `target/` directory when the following command is run:

```
$ buildr package
```

The `package` task depends upon the `compile` task, so if a rebuild is necessary prior to the creation of the JAR, Buildr will see to it.

We can also chain tasks together in a single invocation. For example, we may want to clean all of the old compilation results prior to recompiling and generating a packaged result:

```
$ buildr clean package
```

The `clean` task simply removes the `target/` directory, effectively wiping out any compilation results like class files or resources.

## Directory Structure

As you may have noticed, Buildr does have some default notions about what a project should look like and how it should be organized. We think that these defaults are quite nice and make for a more navigable project. However, we do understand that not all projects are exactly alike.

Buildr's [layouts](#) make it possible for any project to easily change these defaults. For example, this would allow you to easily migrate a project that had been based on a different directory structure, such as the `src/` and `bin/` convention often used by Ant.

## Dependencies

---

So far, we have seen how Buildr can automatically infer what amounts to dozens of lines of `build.xml` contents, all based on a buildfile and a directory structure. However, the best is yet to come. Buildr also provides Maven-style dependency management (but without the long loading times!). In other words, you specify each dependent library using a string descriptor and Buildr figures out how to download and configure your classpath (the library descriptors are just a Ruby array, therefore they are separated by commas (,)). You must specify at least one remote repository so that Buildr knows from where to download these libraries. For example, we can configure our project to reference the [Apache Commons CLI](#) library and download libraries from the Ibiblio repository:

```
repositories.remote << 'http://www.ibiblio.org/maven2'  
  
define 'killer-app' do  
  project.version = '0.1.0'  
  compile.with 'commons-cli:commons-cli:jar:1.2'  
  package :jar  
end
```

This sort of dependency declaration should look quite familiar if you are at all familiar with Maven. The general format for an artifact descriptor is `groupId:artifactId:packageType:version`. Any Maven artifacts included in this fashion will be retrieved from the [list of remote repositories](#) (in this case, Ibiblio) and installed in your local repository at `~/ .m2/repository/`.



You can search the global repository of artifacts at sites like [MvnBrowser](#). Simply enter the name of the library you are looking for, and the search should pull up the `groupId`, `artifactId` and a list of available versions.

Unfortunately, not all libraries are quite as simple as Commons CLI. Many libraries (such as Apache Wicket) have dependencies of their own. While we may be able to *compile* against Apache Wicket without these extra libraries on our classpath, we cannot actually *run* our application without its transitive dependencies. To avoid tracking down each of these dependencies and adding them manually, we can simply use the `transitive` directive (this is how Maven behaves by default):

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

define 'killer-app' do
  project.version = '0.1.0'
  compile.with transitive('org.apache.wicket:wicket:jar:1.4-rc6')
  package :jar
end
```

The `compile.with` property accepts a full array of comma-separated artifacts, making it possible to specify any number of dependencies as necessary. Of course, such a long list of verbose string descriptors could get very tiresome and messy. For this reason, it is conventional to assign each dependency to a constant (e.g. `WICKET`) which is declared just above the project in the buildfile and passed to `compile.with` in a clean, easy-to-read style:

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

WICKET = transitive('org.apache.wicket:wicket:jar:1.4-rc6')
SLF4J = 'org.slf4j:slf4j-jdk14:jar:1.5.8'

define 'killer-app' do
  project.version = '0.1.0'
  compile.with WICKET, SLF4J
  package :jar
end
```

Unfortunately as it may seem, not all libraries are available in Maven repositories. While most of the major libraries (e.g. Hibernate, Spring, etc) are kept updated by intrepid volunteers, some of the more obscure frameworks are left out in the cold. An example of one such framework is [DBPool](#), a very fast connection pool designed to integrate with JDBC. However, like most Java libraries, DBPool does provide a zip archive which contains the JAR file, as well as some documentation and perhaps a license or two.

Almost magically, we can instruct Buildr to get the DBPool artifact from this URL. Buildr will treat this download just like any other artifact, retrieving it when required by the `compile` task. However, unlike a normal Maven artifact, Buildr will do some extra processing once the download is complete. It will actually dig into the downloaded archive, detect and extract the JAR file, installing it into the local repository just like any other artifact:

```
DBPOOL = 'net.snaq:dbpool:jar:4.8.3'  
download artifact(DBPOOL) => 'http://www.snaq.net/java/DBPool/  
DBPool_v4.8.3.zip'  
  
define 'killer-app' do  
  project.version '0.1.0'  
  compile.with DBPool  
  package :jar  
end
```

This is one area where Buildr's dependency management vastly exceeds Maven's. With Maven, you would have to install the DBPool dependency manually. Buildr's auto-magical download and extraction keeps the dependency definitions centralized within the buildfile, available to your entire team and automatically resolved as needed by the compilation tasks.

## Testing

---

Buildr supports auto-magical integration with a number of mainstream testing frameworks. For Java, this includes the ubiquitous JUnit4, as well as TestNG and a number of others. Scala supports Specs and ScalaTest, while Groovy supports EasyB. Configuration is as simple as placing your test sources in the appropriate directory. In the case of JUnit or TestNG, this would be `src/test/java/`. Once these tests are in place, we can run them using the `test` task:

```
$ buildr test
```

When the `test` task runs, it will ensure that your main sources are compiled, as well as the tests themselves. In the case of JUnit4, test classes are auto-detected based on which base class they extend (`TestCase`). These tests will be invoked using the special test classpath. This classpath includes all of the dependencies passed to `compile.with` along with the dependencies required for testing. Thus, Buildr will actually go out and download JUnit 4.5 (if necessary) and place that JAR on the classpath in order to run your tests. It is also possible to add artifacts specifically required for testing. So, if your tests make use of the Commons Collections library, but your main sources do not, you can include that dependency only for the tests by using the `test.with` property. This functions identically to `compile.with`:

```
define 'killer-app' do
  project.version = '0.1.0'
  compile.with 'commons-cli:commons-cli:jar:1.2'
  test.with 'commons-collections:commons-collections:jar:3.2'
  package :jar
end
```

Of course, not everyone *likes* JUnit4. As mentioned previously, Buildr supports a number of test frameworks. It is possible to use TestNG instead of JUnit4 by setting the `test.using` property to `:testng`:

```
define 'killer-app' do
  project.version = '0.1.0'
  compile.with 'commons-cli:commons-cli:jar:1.2'
  test.with 'commons-collections:commons-collections:jar:3.2'
  test.using :testng
  package :jar
end
```

Note that only one test framework per-project may be used. This may seem like an obvious restriction given that both frameworks introduced so far have used the same directory, but other frameworks such as Specs and EasyB do not follow the same convention. In cases of ambiguity (for example, when tests are present in both `src/test/java/` and `src/spec/scala/`), only one test framework will be chosen, but this choice is not well-defined. When in doubt, explicitly specify the test framework with the `test.using` property. This overrides Buildr's auto-detection and ensures sane behavior.

Other test frameworks are documented [here](#) and [here](#).

## Custom Tasks

---

If there is one area in which Buildr excels, it is defining custom tasks. This is something which is notoriously difficult in both Ant and Maven, often requiring separate Java plugins and mountains of code simply to perform basic tasks. For example, let's imagine that we wanted to define a `run` task which would compile and run our "killer-app" project. This is a simple matter of invoking the `java` command against our main class:

```
define 'killer-app' do
  project.version = '0.1.0'
  package :jar

  task :run => :compile do
    system 'java -cp target/classes org.apache.killer.Main'
  end
end
```

This code defines a new task, `run`, which depends upon the `compile` task. This task only performs a single operation: it invokes the `system` method, passing the relevant command as a string. Note that the `system` method documentation may be found [here](#). Tasks use real Ruby (actually, the entire buildfile is real Ruby too), so if you are familiar with that language, then you should be right at home writing custom tasks in Buildr. We can invoke this task in the following way:

```
$ buildr killer-app:run
```

This works, but it's clumsy. The reason we had to give the “`killer-app:`” prefix is because we defined the `run` task *within* our project, rather than outside of the `define` block. However, if we define `run` outside of the project, then we don't really have access to the `compile` task (which is project-specific). The solution here is a bit of magic known as `local_task`. This is how tasks like `compile` and `test`, which are technically project-specific (think: instance methods) can be invoked without the fully-qualified project name:

```
Project.local_task :run

define 'killer-app' do
  project.version '0.1.0'

  package :jar

  task :run => :compile do
    system 'java -cp target/classes org.apache.killer.Main'
  end
end
```

Now, we can invoke run exactly the way we want, with a minimum of wasted characters:

```
$ buildr run
```

## Summary

---

As long as this guide was, we have barely even scratched the surface of Buildr's true power. This was meant only to get you up and running as quickly as possible, exploiting some of Buildr's unique features to ease your build process. For more comprehensive documentation, start reading about [projects in Buildr](#) and work your way from there.

# Installing and Running

---

Installing on Linux.....	16
Installing on OS X .....	18
Installing on Windows .....	19
Installing for JRuby.....	19
Using multiple versions of Buildr	
Running Buildr .....	21
Help Tasks .....	23
Learning More .....	24

# All In One Bundle

---

Contributors to this project maintain a separate distribution over on [RubyForge](#). Using this distribution, it is possible to install Buildr just like you would install Apache Ant or Apache Maven. It comes bundled with JRuby and distributed as a cross-platform zip or tar.gz file.

The RubyForge distribution is **not** an official Apache distribution. The all-in-one bundles are hosted on the [Files page](#).

**In details:** Navigate to [Rubyforge's buildr's Files page](#) and download the all-in-one bundle for the latest version available.

Unzip the bundle in a convenient location.

You can execute the buildr executable under the bin directory directly.

You should consider adding the bin directory of the result to the PATH global environment variable. See [Installing Buildr for JRuby](#) for instructions to update it.

## Installing the gem

---

**The easy way:** We recommend you pick the platform you want to run Buildr on and then follow the *easy way* instructions for that platform. It could save you an hour or two struggling to install all the right dependencies.

[Installing Buildr for JRuby](#) is the same on all operating systems. Choose JRuby if you're working with Java 6 on OS X, developing with multiple JDKs, or just like JRuby better.

If you are running behind a proxy server, make sure the environment variable `HTTP_PROXY` is set, as many of these steps require HTTP access.

**In details:** 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](#).

Older versions of RubyGems are all kind of fail. You want to avoid these unless you have the patience to install each Buildr dependency manually. Get RubyGems 1.3.1 or later, and when using Debian packages (e.g. Ubuntu), make sure to get the unmolested RubyGems straight from the source.

The Ruby interpreter and JVM must use compatible architectures. For example, OS X comes with 32-bit version of Ruby, Java 1.5 in both 32-bit and 64-bit flavors, and 64-bit Java 6. As a result you can run Ruby with Java 1.5 (32-bit), but to use Java 6 you either need to build Ruby from source for 64-bit, or use [Buildr for JRuby](#).

## Installing on Linux

---

**The easy way:** Use this bash script to [install Buildr on Linux](#). This script will install the most recent version of Buildr, or if already installed, upgrade to the most recent version. It will also install Ruby 1.8.6 if not already installed (requires `apt-get`, `yum` or `urpmi`) and upgrade to RubyGems 1.3.1 or later.

**In details:** 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
```

If using Ubuntu 9.10 or earlier, 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/45905/rubygems-1.3.1.tgz
$ tar xzf rubygems-1.3.1.tgz
$ cd rubygems-1.3.1
$ 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 1.4.3
```

## Installing on OS X

---

**The easy way:** Use this script to [install Buildr on OS X](#). This script will install the most recent version of Buildr, or if already installed, upgrade to the most recent version. It will also install Ruby 1.8.6 if not already installed (using MacPorts/Fink) and upgrade RubyGems to 1.3.1 or later.

You need to have the Apple Development Tools installed. They are available on the Mac OSX installation CD.



Java Update 3 for Snow Leopard removes header files necessary to compile the native Ruby-Java Bridge (RJB) gem, so installing rjb gem may fail on OS X. The solution is to install Java for Mac OS X 10.6 Update 3 Developer Package from <http://connect.apple.com> before `gem install`.

**Using RVM?** If you're not using the built-in ruby on OS X (e.g., if you're using RVM), you'll need to force-install the platform-independent RJB:

```
$ gem install rjb -v 1.3.3 --platform ruby
```

The darwin pre-built binary seems to only work with the built-in ruby.

**In details:** OS X 10.5 (Leopard) comes with a recent version of Ruby 1.8.6. You do not need to install a different version of Ruby when running OS X 10.5.

OS X 10.4 (Tiger) includes an older version of Ruby that is not compatible with Buildr. You can 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 1.3.4
```

## Installing on Windows

---

**The easy way:** The easiest way to install Ruby is using the [one-click installer](#). Be sure to install Ruby 1.8.6; support for Ruby 1.9.x is still a work in progress. Once installed, set the `JAVA_HOME` environment variable and run `gem install buildr --platform mswin32`.

**In details:** 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 --platform mswin32
```

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

```
> gem update buildr
> gem install buildr -v 1.3.4 --platform mswin32
```

## Installing for JRuby

---

**The easy way:** Use this bash script to [install Buildr on JRuby](#). This script will install the most recent version of Buildr, or if already installed, upgrade to the most recent version. If necessary, it will also install JRuby 1.6.1 in `/opt/jruby` and update the `PATH` variable in `~/.bash_profile` or `~/.profile`.

**In details:** If you don't already have JRuby 1.5.1 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 1.3.4
```

### **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.

## Using multiple versions of Buildr

---

Rubygems makes it possible to install several versions of Buildr side-by-side on the same system. If you want to run a specific version, you can do so by adding the version number between underscores (‘\_’) as the first command-line parameter. For example,

```
$ buildr _1.3.4_ clean # runs Buildr v1.3.4

$ buildr _1.4.4_ clean # runs Buildr v1.4.4
```



There are two ‘buildr’ executables installed by Rubygems. One script serves to select the specified (or default) version of Buildr and is typically found under ‘/usr/bin/buildr’ or ‘/var/lib/gems/1.8/bin/buildr’. The exact location will vary depending on your system. The other script is the Buildr bootstrap per se and can be found under the specific version of Buildr, e.g. ‘/var/lib/gems/1.8/gems/buildr-1.4.0/bin/buildr’. The first script should be on your ‘PATH’. The second script should not be called directly and should not be on your ‘PATH’.

## 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 by running `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
<code>-f/--buildfile [file]</code>	Specify the buildfile.
<code>-e/--environment [name]</code>	Environment name (e.g. development, test, production).
<code>-h/--help</code>	Display this help message.
<code>-n/--nosearch</code>	Do not search parent directories for the buildfile.
<code>-q/--quiet</code>	Do not log messages to standard output.
<code>-r/--require [file]</code>	Require MODULE before executing buildfile.
<code>-t/--trace</code>	Turn on invoke/execute tracing, enable full backtrace.
<code>-v/--verbose</code>	Log message to standard output
<code>-V/--version</code>	Display the program version.
<code>-P/--prereqs</code>	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.....	25
The Directory Structure .....	27
Naming And Finding Projects .....	30
Running Project Tasks .....	31
Setting Project Properties.....	32
Resolving Paths.....	33
Defining The Project.....	34
Writing Your Own Tasks.....	36

## 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/openjpa"

include Buildr::OpenJPA

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:jar:1.2.0',
  '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.with 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`, `build` or any existing task name 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 run `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 run `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')

# the same using symbols
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

---

Compiling .....	38
Resources .....	42
More On Building .....	44
Cleaning .....	46
Continuous Compilation .....	46

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 project's `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:

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

This time, the variable `jtree` is a file task that reads a JTree source file from the `src/main/jtree` directory, and generates additional source files in the `target/generated/jtree` 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 include 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.	
:erb	Map from <%=key%> to value.	Map using the matched value of the regular expression (e.g. /=(.*?)/). Regexp

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`.

## Continuous Compilation

---

And if all that weren't enough, `Buildr` also offers a time-saving feature called continuous compilation. This feature, implemented by the `cc` task, instructs `Buildr` to loop eternally, polling your project's source directories for changes. Whenever a change is detected, `Buildr` immediately triggers the appropriate compilation step and goes right back to polling. This allows you to reap many of the benefits of an incrementally compiling IDE like Eclipse without sacrificing your favorite build tool.

To get started, simply invoke the `cc` task at the command prompt:

```
$ buildr cc
```

This task will immediately invoke the `compile` and `test:compile` tasks on your project if necessary. This ensures that your build is completely up to the minute before polling is initiated. After this initial build (if required), Buildr will print a notification indicating which directories are being monitored. By default, these directories will include any source folders (e.g. `src/main/java/`), any test directories (e.g. `src/spec/scala/`) as well as any resources (e.g. `@src/main/resources/`). The Buildr process will remain running during this time, meaning that in order to test this functionality, we will need to open a new shell:

```
$ touch src/main/java/Test.java
```

The moment we run this command, Buildr will detect the change and invoke the `compile` task. It will **not** invoke the `test:compile` task, since none of the test files were actually changed. This ensures that potentially time-consuming tasks are avoided if possible. Note that, unlike the `build` task, the continuous compilation also does not actually run any of your tests. Continuous compilation is designed to be a simple daemon which runs forever, quickly recompiling your project as soon as you save or delete a file. We can terminate the continuous compilation task by pressing Ctrl-C. Left to its own devices, the `cc` task really will loop forever.

There are several advantages to continuous compilation. Number one is convenience. Once you invoke the `cc` task, you can focus exclusively on the code, editing and saving your files in an unbroken workflow. There is no need to break your concentration to invoke Buildr manually unless you need to run the test suite, deploy the application or anything beyond compilation. The second advantage is speed. By using the continuous compilation process, you avoid repeatedly incurring Buildr's startup overhead. While this startup time is kept to a minimum, it is still perceptible, particularly when running on JRuby. Since the `cc` task runs within a Buildr instance which has already been started, there is no need for repeated, inefficient load times. Again, this allows you to focus more completely on what's really important: the code.

By default, the `cc` task will poll your sources once every 200 milliseconds. We have found that this frequency strikes a nice balance between CPU load (which is insignificant) and nearly-instant detection. However, you may wish to tune this value based on your own needs. To do so, simply use the `cc.frequency` property in your project definition:

```
project 'foo' do
  cc.frequency 1.5      # poll every one-and-a-half seconds
end
```

If you find that the 200 ms default imposes too much overhead, try changing `cc.frequency` to a higher value. On the flip side, if you find that you're waiting too long for changes to be caught by the poll, tune the frequency lower.

Now let's [talk about the artifacts](#) we mentioned before.

# Artifacts

---

Specifying Artifacts .....	49
Specifying Repositories .....	51
Downloading Artifacts .....	53
Install and Upload .....	55

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:jar:1.2.1',
  '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 your repository requires HTTP authentication, you can write,

```
repositories.remote <<
URI.parse("http://user:password@repository.example.com")
```

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 (use `HTTPS_PROXY` for proxying HTTPS connections). 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 the `buildr.rb` file in the `.buildr` directory under your home directory.

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:jar:1.2.1').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`.

We'll talk more about installing and uploading in the next chapter, but right now we're going to [package some artifacts](#).

# Packaging

---

Specifying And Referencing Packages .....	58
Packaging ZIPs .....	60
Packaging JARs .....	62
Packaging WARs .....	63
Packaging AARs .....	64
Packaging EARs .....	65
Packaging OSGi Bundles.....	67
Packaging Tars and GZipped Tars.....	73
Installing and Uploading.....	74
Packaging Sources and JavaDocs .....	75

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('*.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 perform identically. They both include all the files from the `target/docs` directory, but not the directory itself, and they are both lazy, meaning that the files can be created later and they will still get packaged into the zip package.

For example, when you use `package :jar`, under the hood it specifies to include all the files from `target/classes` with `:as=> '.'`. Even though this happens during project definition and nothing has been compiled yet (and in fact `target/classes` may not even exist yet), the `.class` files generated during compilation are still packaged in the `.jar` file, as expected.

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
<code>:war</code>	J2EE Web Application (WAR).
<code>:ejb</code>	Enterprise Java Bean (JAR).
<code>:jar</code>	J2EE Application Client (JAR).
<code>:lib</code>	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.

- **description** — The application’s description defaults to the project’s comment. You can change that by setting the `description` 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).description = 'MyCoolWebService: Making coolness kool again'
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`.

It is also possible to add `security-role` tags to the `application.xml` file by appending a hash with `:id`, `:description` and `:name` to the `security_role` array, like so:

```
package(:ear).security_roles << {:id=>'SecurityRole_123',
:description=>'Read only user', :name=>'coolUser'}
package(:ear).security_roles << {:id=>'SecurityRole_456',
:description=>'Super user', :name=>'superCoolUser'}
```

## Packaging OSGi Bundles

---

OSGi bundles are jar files with additional metadata stored in the manifest. Buildr uses an external tool [Bnd](#) to create the package. Directives and properties can be explicitly passed to the build tool and buildr will provide reasonable defaults for properties that can be derived from the project model. Please see the [bnd tool](#) for documentation on the available properties.

The bundle packaging format is included as an addon so the build file must explicitly require the addon using `require "buildr/bnd"` and must add a remote repository from which the `bnd` can be downloaded. A typical project that uses the bundle packaging addon may look something like;

```
require "buildr/bnd"

repositories.remote << Buildr::Bnd.remote_repository

define 'myProject' do
  ...
  package(:bundle).tap do |bnd|
    bnd['Import-Package'] = "*/resolution:=optional"
    bnd['Export-Package'] = "*/version=#{version}"
  end
  ...
end
```

The [] method on the bundle package is used to provide directives to the bnd tool that are not inherited by sub-projects while the standard 'manifest' setting is used to define properties that inherited by sub-projects.

## Defaults

The addon sets the following bnd parameters;

- "Bundle-Version" defaults to the project version.
- "Bundle-SymbolicName" defaults to the concatenation of the project group and project id, replacing ':' characters with '.'.
- "Bundle-Name" defaults to the project description if present else the project name
- "Bundle-Description" defaults to the project description.
- "-classpath" is set to the compile target directory and any compile time dependencies.
- "Include-Resource" defaults to the dir project.resources.target if it exists.

## Parameters

### classpath\_element

The user can also specify additional elements that are added to the classpath using the ‘classpath\_element’ method. If the parameter to this element is a task, artifact, artifact namespace etc. then it will be resolved prior to invoking bnd.

```
...
define 'foo' do
  ...
  package(:bundle).tap do |bnd|
    # This dependency will be added to classpath
    bnd.classpath_element 'someOtherExistingFile.zip'
    # All of these dependencies will be invoked and added to classpath
    bnd.classpath_element artifact('com.sun.messaging.mq:mq:jar:4.4')
    bnd.classpath_element project('bar') # Adds all the packages
    bnd.classpath_element 'org.apache.ant:ant:jar:1.8.0'
    bnd.classpath_element file('myLocalFile.jar')
  ...
end

project 'bar' do
  ...
end
end
```

### classpath

The user can specify the complete classpath using the ‘classpath’ method. The classpath should be an array of elements. If the element is a task, artifact, artifact namespace etc. then it will be resolved prior to invoking bnd.

```

...
define 'foo' do
  ...
  package(:bundle).tap do |bnd|
    bnd.classpath [ project.compile.target,
                    'someOtherExistingFile.zip',
                    artifact('com.sun.messaging.mq:imq:jar:4.4'),
                    project('bar'),
                    'org.apache.ant:ant:jar:1.8.0',
                    file('myLocalFile.jar') ]
    ...
  end

  project 'bar' do
    ...
  end
end

```

## Examples

### Including non-class resources in a bundle

Bnd can be used to include non-class resources in a bundle. The following example includes all resources in 'src/etc' into the bundle.

```

define 'myproject' do
  ...
  package(:bundle).tap do |bnd|
    bnd['Include-Resource'] = project._('src/etc') + '/'
    ...
  end
end

```

## Using bnd to wrap an existing jar

Bnd can be used to wrap an existing jar as an OSGi bundle. The following example wraps the OpenMQ JMS provider as an OSGi bundle.

```
...
# Add repository for OpenMQ
repositories.remote << 'http://download.java.net/maven/2'

desc 'OSGi bundle for OpenMQ JMS provider client library'
define 'com.sun.messaging.mq.imq' do
  project.version = '4.4'
  project.group = 'iris'
  package(:bundle).tap do |bnd|
    bnd['Import-Package'] = ".*;resolution:=optional"
    bnd['Export-Package'] = "com.sun.messaging.*;version=#{version}"
    bnd.classpath_element 'com.sun.messaging.mq:imq:jar:4.4'
  end
end
```

## Create an OSGi bundle with an Activator

The following example presents a basic buildfile for building an OSGi bundle with an activator.

```
...
# repository for OSGi core bundle
repositories.remote << 'https://repository.apache.org/content/repositories/
releases'

desc 'Hello World bundle'
define 'helloworld' do
  project.version = '1.0'
  project.group = 'org.example'
  compile.with 'org.apache.felix:org.osgi.core:jar:1.4.0'
  package(:bundle).tap do |bnd|
    bnd['Export-Package'] = "org.example.helloworld.api.*;version=#{version}"
    bnd['Bundle-Activator'] = "org.example.helloworld.Activator"
  end
end
```

### Inheriting parameters for bnd tool

The following example shows how you can use ‘manifest’ to define a bnd parameter that is inherited by all child sub-projects. The “Bundle-License” defined in the top level project is passed to the bnd tool when generating both the ‘fe’ and ‘fi’ sub-projects but the ‘fo’ sub-project overrides this parameter with a local value.

```
...
define 'myproject' do
  manifest['Bundle-License'] = "http://www.apache.org/licenses/LICENSE-2.0"
  ...
  define 'fe' do
    ...
    package(:bundle).tap do |bnd|
      ...
    end
  end
end

define 'fi' do
  ...
  package(:bundle).tap do |bnd|
    ...
  end
end

define 'fo' do
  ...
  package(:bundle).tap do |bnd|
    bnd['Bundle-License'] = "http://www.apache.org/licenses/LICENSE-1.1"
  end
end
end
```

## 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 is 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 change 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 'upload'=>'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 doc task, so you can use it in combination with the doc method. For example:

```
package :javadoc
doc :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

---

Writing Tests.....	77
Excluding Tests and Ignoring Failures .....	78
Running Tests .....	79
Integration Tests.....	81
Using Setup and Teardown.....	82
Testing Your Build .....	82
Behaviour-Driven Development.....	85

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, as a gentle reminder:

```
test do
  warn "Did you forget something?" if test.tests.nil? || test.tests.empty?
end
```

The `tests` collection holds the names of all classes with tests, if any. And there's `classes`, which holds the names of all test classes. 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
```

Buildr forcefully runs all tests that match the pattern. If you want to re-run all tests even if your sources have not changed, you can execute:

```
$ buildr test:*
```

You can exclude tests by preceding them with a minus sign ('-'):

```
$ buildr test:-Bar
```

The above would run all tests except those with a name containing Bar. Exclusions can be combined with inclusions:

```
$ buildr test:Foo,-Bar
```

Buildr would then run tests with names containing Foo but not Bar.

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.

During development, if you want to re-run only tests that have failed during the last test execution, you can execute:

```
$ buildr test:failed
```

One last note on running tests. By default when you run tests, Buildr will automatically run all transitive test dependencies. This means if you run “`buildr test`” inside project `bar` and `bar` depends on project `foo`, Buildr will first run tests in project `foo` if there have been any changes affecting `foo` that haven’t been taken into account yet. This behavior often surprises people, especially when they are trying to get things done and only care about tests in `bar` at that moment. For those times when you’d like to focus your testing on specific projects, Buildr has the only option that will only run tests for projects specified on the command line,

```
$ buildr test=only bar:test
```

## 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 complicated 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 an 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 ...
<code>exist</code>	Given a file task checks that the file (or directory) exists.
<code>empty</code>	Given a file task checks that the file (or directory) is empty.
<code>contain</code>	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 <code>*</code> and <code>**</code> are allowed.

All these matchers operate against a file task. If you run them against a `ZipTask` (including JAR, WAR, etc) or a `TarTask`, they can also check the contents of the archive. And as you can see in the examples above, you can also run them against a path in an archive, checking its contents as if it was a directory, or against an entry in an archive, 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 .....	86
Personal Settings .....	88
Build Settings .....	90
Non constant settings .....	91
Environments .....	92
Profiles .....	93

## 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, the environment variables TEST and DEBUG are case insensitive, you can use either test=no or TEST=no. Any other environment 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 <code>-e</code> command line option.
DEBUG	Set to <code>no/off</code> if you want Buildr to compile without debugging information (default when running the <code>release</code> task, see <a href="#">Compiling</a> ).
HOME	Your home directory.
HTTP_PROXY	URL for HTTP proxy server (see <a href="#">Specifying Repositories</a> ).
HTTPS_PROXY	URL for HTTPS proxy server (see <a href="#">Specifying Repositories</a> ).
IGNORE_BUILDFILE	Set to “true” or “yes” to ignore changes in Buildfile or its dependencies when running tests.
JAVA_HOME	Points to your JDK, required when using Java and Ant.
JAVA_OPTS	Command line options to pass to the JDK (e.g. <code>'-Xms1g'</code> ).
M2_REPO	Location of the Maven2 local repository. Defaults to the <code>.m2</code> directory in your home directory ( <code>ENV['HOME']</code> ).
NO_PROXY	Comma separated list of hosts and domain that should not be proxied (see <a href="#">Specifying Repositories</a> ).
TEST	Set to <code>no/off</code> to tell Buildr to skip tests, or <code>all</code> to tell Buildr to run all tests and ignore failures (see <a href="#">Running Tests</a> ).
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 `m512mb`, where it should be `Xm512mb`. 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.release_to[:username] = ENV['USERNAME']  
repositories.release_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 preferred 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

```
release_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 add-on.

Artifacts defined on `build.yml` 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 retrieved 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 in the `.buildr` directory under 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 `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.release_to[:username] = 'assaf'
repositories.release_to[:password] = 'supersecret'
# Search here first, it's faster
repositories.remote << 'http://inside-the-firewall'
```



Buildr 1.3 and earlier used the file `buildr.rb` directly in your home directory. Starting with version 1.4, Buildr loads `buildr.rb` from the `.buildr` directory under your home directory in preference. If you use Buildr 1.3 and earlier and don't want to duplicate your settings, you can move your existing `buildr.rb` under the `.buildr` directory and create a new `buildr.rb` in your home directory containing:

```
# Backward compatibility: Buildr 1.4+ uses $HOME/.buildr/buildr.rb
load File.expand_path('buildr.rb', Buildr.application.home_dir)
```

## 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 = (Buildr.environment == 'production' ? 'oracle' : 'mysql')
  resources.from(_("src/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(_("src/main/#{Buildr.settings.profile['db']}"))
  # Set the JDBC URL in copied resource files (config.xml needs this).
  resources.filter.using :jdbc=>Buildr.settings.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 .....	96
Scala.....	101
Groovy .....	110
Ruby.....	113

## 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
<code>:debug</code>	Generates bytecode with debugging information. You can also override this by setting the environment variable <code>debug</code> to <code>off</code> .
<code>:deprecation</code>	If true, shows deprecation messages. False by default.
<code>:lint</code>	Defaults to false. Set this option to true to use all lint options, or specify a specific lint option (e.g. <code>:lint=&gt;'cast'</code> ).
<code>:other</code>	Array of options passed to the compiler (e.g. <code>:other=&gt;'-implicit:none'</code> ).

<code>:source</code>	Source code compatibility (e.g. '1.5').
<code>:target</code>	Bytecode compatibility (e.g. '1.4').
<code>:warnings</code>	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
<code>:fork</code>	VM forking, defaults to true.
<code>:clonevm</code>	If true clone the VM each time it is forked.
<code>:properties</code>	Hash of system properties available to the test case.
<code>:environment</code>	Hash of environment variables available to the test case.
<code>:java_args</code>	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

<code>:once</code>	Create one VM to run all test classes in the project, separate VMs for each project.
<code>:each</code>	Create one VM for each test case class. Slow but provides the best isolation between test classes.
<code>false</code>	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
<code>:properties</code>	Hash of system properties available to the test case.
<code>:java_args</code>	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
--------	-------

<code>:properties</code>	Hash of system properties available to the test case.
<code>:java_args</code>	Arguments passed as is to the JVM.

You can use the `build.yml` 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
```

## Documentation

Buildr offers support for using JavaDoc to generate documentation from any Java sources in a project. This is done using the `doc` task:

```
$ buildr doc
```

This will use the same `.java` sources used by the `compile` task to produce JavaDoc results in the `target/doc/` directory. By default, these sources are chosen only from the current project. However, it is possible to override this and generate documentation from the sources in a sub-project (potentially more than one):

```
define 'foo' do
  # ...

  doc.from projects('foo:bar', 'foo')

  define 'bar' do
    # ...
  end
end
```

With this configuration, the `doc` task will use sources from both `foo:bar` and `foo`.

The `doc` task supports any option that the `javadoc` command does (e.g. `-windowtitle`). To pass an option to the JavaDoc generator, simply specify it using the `doc` method:

```
define 'foo' do
  # ...

  doc :windowtitle => 'Abandon All Hope, Ye Who Enter Here', :private => true
end
```

## Scala

---

Before using Scala, you must first require the Scala compiler:

```
require 'buildr/scala'
```

By default, Buildr will attempt to use the latest stable release of Scala, which is currently Scala 2.9.0 as of May 2011. Of course you can configure a specific version of Scala for your project by adding the following entry in `build.yaml`:

```
scala.version: 2.8.0.Beta1 # Pick your version
```

Or, you can do the same programmatically:

```
# Must be placed before require 'buildr/scala'
Buildr.settings.build['scala.version'] = "2.8.0.Beta1"
```

You may also determine the version in use by querying the `Scala.version` attribute:

```
Scala.version # => '2.8.0'
```

Regardless of how the Scala version is determined, if you have the same Scala version installed on your system and the `SCALA_HOME` environment variable points to it, then your local installation will be used. Otherwise, Buildr will download it from the [Scala Tools repository](#) which is automatically enlisted when you require Scala. The only drawback if you don't have a local installation is the FSC compiler won't be available.



For Mac users, if you have installed Scala via [MacPorts](#) Buildr will look in the `/opt/local/share/scala/` directory if you have not set `SCALA_HOME`.

## 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.

Any Java source files found in the `src/main/java` directory will be compiled using the Scala/Java joint compiler into the `target/classes` directory. Both the Java and the Scala sources are compiled with an inclusive classpath, meaning that you may have a Java class which depends upon a Scala class which depends upon a Java class, all within the same project. The Java sources will be compiled with the same dependencies as the Scala sources with the addition of the `scala-library.jar` file as required for Scala interop.

Note that you cannot use the Groovy **and** the Scala joint compilers in the same project. If both are required, the Groovy joint compiler will take precedence.

+

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`. The joint compilation of Java sources may only be pointed at an alternative directory using the feature to redefine the `_( :src, :main, :java )` path.

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
<code>:debug</code>	If true, generates bytecode with debugging information. Scala 2.9 also accepts: <code>none,source,line,vars,notc</code> .
<code>:deprecation</code>	If true, shows deprecation messages. False by default.
<code>:make</code>	Make strategy to be used by the compiler (e.g. <code>:make=&gt;'transitive'</code> ). <b>Scala 2.8 only</b>
<code>:optimise</code>	Generates faster bytecode by applying optimisations to the program.
<code>:other</code>	Array of options passed to the compiler (e.g. <code>:other=&gt;'-Xprint-types'</code> ).

<code>:target</code>	Bytecode compatibility (e.g. '1.4').
<code>:warnings</code>	Issue warnings when compiling. True when running in verbose mode.
<code>:javac</code>	A hash of options passed to the javac compiler verbatim.

## 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 2.7 compiler task assumes that each `.scala` source file generates a corresponding `.class` file under `target/classes` (or `target/test/classes` 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 is 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.

Fortunately, Scala 2.8 provides a substantially better interface for implementing change detection. Whenever you use Scala 2.8 (see below), Buildr will auto-detect the version and enable this feature dynamically. After the `compile` task runs, the relevant target directory will contain a `.scala-deps` file, generated by the Scala compiler. The manner in which this file is used can be configured using the `:make compiler` option. The following values are available:

- `:all` – Disables compiler-level change detection
- `:changed` – Only build changed files without considering file dependencies
- `:immediate` – **unknown**
- `:transitive` – Build changed files as well as their transitive file dependencies
- `:transitivenocp` – Build changed files as well as their transitive file dependencies  
(**default**)

Please note that there are limits to compiler-level change detection. Most notably, dependencies cannot be tracked across separate compilation targets. This would cause problems in the case where an API has been changed in a main source file. The test suite for the project will **not** be detected as requiring recompilation, potentially resulting in unexpected runtime exceptions. When in doubt, run `clean` to remove all dependency information. In extreme cases, it is possible to completely disable compiler-level change detection by adding the following statement to your project definition:

```
compile.using :make => :all
```

Effectively, this is telling the Scala compiler to ignore the information it has built up regarding source file dependencies. When in this mode, only Buildr's change detection semantics remain in play (as described above).

To avoid unusual behavior, compiler-level change detection is disabled whenever the joint Scala-Java compiler is used. Thus, any `.java` files in a project handled by the Scala compiler will cause the `:make` option to be ignored and revert to the exclusive use of Buildr's change detection mechanism (as described above).

## Scala 2.8

As of version 1.4, Buildr has **non-default** support for Scala 2.8. If your `SCALA_HOME` environment variable is pointing to an installation of Scala 2.8, then Buildr will use that compiler and enable 2.8-specific features.

As Scala 2.8 is currently pre-release, it is often desirable to maintain an installation of Scala 2.7 concurrently with Scala 2.8, defaulting to the former with the option to select the latter on a project-by-project basis. This is most easily accomplished by setting `SCALA_HOME` to point to the Scala 2.7 installation and `SCALA28_HOME` to point to the Scala 2.8 installation. With this configuration in place, Scala 2.8 can be selected for a specific project by dynamically reassigning `SCALA_HOME` at the top of the buildfile (**before** `require 'buildr/scala'`):

```
ENV['SCALA_HOME'] = ENV['SCALA28_HOME']
```

```
require 'buildr/scala'
```

```
...
```

## Testing with Scala

Buildr supports two main Scala testing frameworks: [ScalaTest](#) and [Specs](#). [ScalaCheck](#) is also supported within the confines of either of these two frameworks. Thus, your Specs may use ScalaCheck properties, as may your ScalaTest suites.

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

### ScalaTest

ScalaTest support is activated automatically when there are any `.scala` source files contained in the `src/test/scala` directory. If you are not using this directory convention, you may force the test framework by using the `test.using :scalatest` directive.

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

Specs is automatically selected whenever there are `.scala` source files under the `src/spec/scala` directory. It is also possible to force selection of the test framework by using the `test.using :specs` directive. This can sometimes be useful when Scala sources may be found in **both** `src/test/scala` and `src/spec/scala`. Normally in such cases, `ScalaTest` will have selection precedence, meaning that in case of a conflict between it and `Specs`, `ScalaTest` will be chosen.

Any objects which extend the `org.specs.Specification` superclass will be automatically detected and run. Note that any **classes** which extend `Specification` will also be invoked. As such classes will not have a `main` method, such an invocation will raise an error.

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 mustBe 0
    }
  }
}
```

ScalaCheck is automatically added to the classpath when Specs is used. However, JMock, Mockito, CGlib and similar are *not*. This is to avoid downloading extraneous artifacts which are only used by a small percentage of specifications. To use Specs with Mockito (or any other library) in a Buildr project, simply add the appropriate dependencies to `test.with`:

```
MOCKITO = 'org.mockito:mockito-all:jar:1.7'
CGLIB = 'cglib:cglib:jar:2.1_3'
ASM = 'asm:asm:jar:1.5.3'
OBJENESIS = 'org.objenesis:objenesis:jar:1.1'

define 'killer-app' do
  ...

  test.with MOCKITO, CGLIB, ASM, OBJENESIS
end
```

The dependencies for Specs's optional features are defined [here](#).

## 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
  )
}
```

## Documentation

Buildr offers support for using ScalaDoc or VScalaDoc to generate documentation from any Scala sources in a project. This is done using the doc task:

```
$ buildr doc
```

This will use the same `.scala` sources used by the `compile` task to produce ScalaDoc results in the `target/doc/` directory. By default, these sources are chosen only from the current project. However, it is possible to override this and generate documentation from the sources in a sub-project (potentially more than one):

```
define 'foo' do
  # ...

  doc.from projects('foo:bar', 'foo')

define 'bar' do
  # ...
end
end
```

With this configuration, the `doc` task will use sources from both `foo:bar` and `foo`.

The `doc` task supports any option that the `scaladoc` command does (e.g. `-windowtitle`). To pass an option to the ScalaDoc (or VScalaDoc) generator, simply specify it using the `doc` method:

```
define 'foo' do
  # ...

  doc :windowtitle => 'Abandon All Hope, Ye Who Enter Here', :private => true
end
```

By default, the `doc` task will use the ScalaDoc generator on Scala projects. To select the VScalaDoc generator, you must use the `doc.using` invocation:

```
define 'foo' do
  doc.using :vscaladoc
end
```

The `doc` task is **not** joint-compilation aware. Thus, it will only generate ScalaDoc for mixed-source projects, it will not attempt to generate both JavaDoc and ScalaDoc.

## 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
<code>encoding</code>	Encoding of source files.
<code>verbose</code>	Asks the compiler for verbose output, true when running in verbose mode.
<code>fork</code>	Whether to execute groovyc using a spawned instance of the JVM. Defaults to no.
<code>memoryInitialSize</code>	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)
<code>memoryMaximumSize</code>	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)

<code>listfiles</code>	Indicates whether the source files to be compiled will be listed. Defaults to no.
<code>stacktrace</code>	If true each compile error message will contain a stacktrace.
<code>warnings</code>	Issue warnings when compiling. True when running in verbose mode.
<code>debug</code>	Generates bytecode with debugging information. Set from the debug environment variable/global option.
<code>deprecation</code>	If true, shows deprecation messages. False by default.
<code>optimise</code>	Generates faster bytecode by applying optimisations to the program.
<code>source</code>	Source code compatibility.
<code>target</code>	Bytecode compatibility.
<code>javac</code>	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
<code>:properties</code>	Hash of system properties available to the test case.

<code>:java_args</code>	Arguments passed as is to the JVM.
<code>:format</code>	Report format, either <code>:txt</code> or <code>:xml</code>

## Documentation

Buildr offers support for using GroovyDoc to generate documentation from any Groovy sources in a project. This is done using the `doc` task:

```
$ buildr doc
```

This will use the same `.groovy` sources used by the `compile` task to produce GroovyDoc results in the `target/doc/` directory. By default, these sources are chosen only from the current project. However, it is possible to override this and generate documentation from the sources in a sub-project (potentially more than one):

```
define 'foo' do
  # ...

  doc.from projects('foo:bar', 'foo')

  define 'bar' do
    # ...
  end
end
```

With this configuration, the `doc` task will use sources from both `foo:bar` and `foo`.

The `doc` task supports any option that the `groovydoc` command does (e.g. `-windowtitle`). To pass an option to the GroovyDoc generator, simply specify it using the `doc` method:

```
define 'foo' do
  # ...

  doc :windowtitle => 'Abandon All Hope, Ye Who Enter Here', :private => true
end
```

The doc task is **not** joint-compilation aware. Thus, it will only generate GroovyDoc for mixed-source projects, it will not attempt to generate both JavaDoc and GroovyDoc.

## 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](#). That 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.yml` settings file to specify a particular version of JRuby (defaults to 1.4.0 as of Buildr 1.3.5). For example:

```
jruby: 1.3.1
```

### 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
<code>:gems</code>	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 <code>ci_reporter</code> gem to generate xml reports
<code>:requires</code>	Array of ruby files to require before running the specs
<code>:format</code>	Array of valid RSpec <code>--format</code> option values. Defaults to html report on the <code>reports</code> directory and text progress
<code>:output</code>	File path to output dump. <code>false</code> to suppress output
<code>:fork</code>	Run the tests on a new java vm. (enabled unless running on JRuby)
<code>:properties</code>	Hash of system properties available to the test case.
<code>:java_args</code>	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
<a href="#">RSpec</a>	Files in <code>src/spec/ruby</code> ending with <code>*_spec.rb</code> or <code>*_story.rb</code>

<a href="#">TestUnit</a>	Files in <code>src/spec/ruby</code> ending with <code>*_test.rb</code> , <code>*Test.rb</code>
<a href="#">Expectations</a>	Files in <code>src/spec/ruby</code> ending with <code>*_expect.rb</code>
<a href="#">JUnit</a>	Classes from <code>src/test/java</code> that either subclass <code>junit.framework.TestCase</code> , include methods annotated with <code>org.junit.Test</code> , or test suites annotated with <code>org.org.junit.runner.RunWith</code> .
<a href="#">TestNG</a>	Classes from <code>src/test/java</code> annotated with <code>org.testng.annotations.Test</code>

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
<code>:config</code>	The JtestR config file to be loaded after being configured by Buildr. Defaults to <code>src/spec/ruby/jtestr_config.rb</code> .
<code>:gems</code>	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 <code>ci_reporter</code> gem to generate xml reports
<code>:requires</code>	Array of ruby files to require before running the specs
<code>:format</code>	Array of valid RSpec <code>--format</code> option values. Defaults to <code>html report</code> on the <code>reports</code> directory and <code>text progress</code>
<code>:output</code>	File path to output dump. <code>false</code> to suppress output
<code>:fork</code>	Run the tests on a new java vm. (enabled unless running on JRuby)

---

<code>:properties</code>	Hash of system properties available to the test case. (only when fork is enabled)
<code>:java_args</code>	Arguments passed as is to the JVM. (only when fork is enabled)

---

# More Stuff

---

Interactive Shells (REPLs).....	117
Running Your Application.....	122
Using Gems.....	124
Using Java Libraries .....	126
BuildrServer.....	127
Notifications: Growl, Libnotify, Qube.....	128
Eclipse.....	129
Cobertura, Emma, JDepend.....	140
Xjc Compiler	
Anything Ruby Can Do .....	143

## Interactive Shells (REPLs)

---

Many languages (including Scala and Groovy) provide an interactive shell tool which allows developers to run arbitrary expressions and see the results immediately:

```
$ scala
Welcome to Scala version 2.7.4.final (Java HotSpot(TM) 64-Bit Server VM, Java
1.6.0_03-p3).
Type in expressions to have them evaluated.
Type :help for more information.

scala> 6 * 7
res0: Int = 42

scala>
```

These tools are alternatively known as “REPLs” (Read, Eval, Print Loop), a term which originally comes from Lisp. This sort of interactive shell can be an invaluable asset when developing frameworks or other non-UI-oriented modules. A common use-case is running a quick, manual test to make sure that the framework is functioning properly. It is faster and easier to do this in a shell, and also averts the need for extra test cases to be developed just to check simple things during development.

Unfortunately, for such a tool to be useful in Java, Scala or Groovy development, it must have access to the CLASSPATH, not only the compiled project binaries, but also its full list of dependencies. While it is usually possible with such tools to specify the classpath upon startup (e.g. the `-cp` switch for the Scala shell), this can get extremely tedious for project’s with more dependencies, especially when some of these dependencies are defined using transitive artifacts.

Buildr is capable of easing this workflow by providing full support for the configuration and launch of interactive shells, the relevant shell may be launched simply by invoking the `shell` task:

```
$ buildr shell
```

The `shell` task depends upon `compile`, meaning that any changed source files will be recompiled prior to the shell’s launch. Tests will not be run, nor will test files be recompiled. From within the shell itself, you should have access to your project’s compilation classpath (anything specified using `compile.with`) as well as the compiled sources for your project.

The project classpath is determined by checking the current working directory of your system shell (the path from which the `buildr shell` command was invoked) and recursively finding the relevant project for that directory. Thus, if you have a parent project `foo` which has sub-projects `bar` and `baz`, you may invoke an interactive shell for the `baz` project simply by `cd`-ing into its project directory (or any of its sub-directories) and invoking the `shell` task. You may also invoke a shell against a specific project by giving its fully-qualified descriptor as a prefix to the `shell` task:

```
$ buildr foo:bar:shell
```

## Supported Shells

By default, Buildr will open the interactive shell which corresponds to your project's language. Thus, if your project is using Groovy, it will invoke the `groovysh` command, configured with the appropriate classpath. If your project is using Scala, then the `shell` task will invoke the `scala` command. Unfortunately, the Java language does not define an interactive shell of any kind, however for those projects using exclusively the Java language, Buildr will use the [BeanShell](#) console.

However, you may still wish to exploit the advantages of an interactive shell from some other JVM language even while working in Java. Alternatively, you may want to use some shell other than the default even when working in a language which has its own. There are two ways to achieve this aim. The quickest way is to explicitly specify the relevant shell at the call-site:

```
$ buildr foo:shell:jirb
```

This will open the JIRB shell (the JRuby REPL) for the `foo` project. Whenever you are specifying a shell explicitly in this fashion, you **must** fully-qualify the project name:

```
$ buildr shell:jirb      # wrong!!
```

The above will fail because of the way that Rake tasks are dispatched.

Obviously, this explicit specification is a little bit clunky. It would be much easier if we could simply say that we **always** want to use the JIRB shell for this particular project. This can be done by using the `shell.using` directive within your project definition:

```
define 'foo' do
  shell.using :jirb
end
```

With this project definition, we can now invoke the `shell` task and JIRB will be launched, regardless of the default for our project's language:

```
$ buildr shell
```

Buildr supports several different shell providers, and the framework is flexible enough that additional support can be added almost trivially. The following table gives the complete list of supported shells, their corresponding `shell.using` descriptor and the language for which they are the default (if applicable):

Shell Name	Descriptor	Language
BeanShell Console	:bsh	Java
Clojure REPL	:clj	N/A
GroovySH	:groovysh	Groovy
JRuby IRB	:jirb	N/A
Scala Interpreter	:scala	Scala

Note that some of these shells impose certain requirements to enable use. The Groovy shell requires the `GROOVY_HOME` environment variable to point to the Groovy install path. The Clojure REPL makes a similar requirement of `CLOJURE_HOME`. The JRuby and Scala shells will use `JRUBY_HOME` and `SCALA_HOME` respectively if they are defined. However, if these environment variables are not defined, the relevant JAR files will be automatically downloaded from the appropriate Maven2 repository.

## Verbosity and Tracing

By default, Buildr is moderately verbose, meaning that it attempts to give you enough context into what's happening during the build.

It's possible to silence Buildr if you're inconvenienced by its default verbosity by issuing,

```
$ buildr --silent
```

On the other hand, if you want Buildr to give you more context in order to trace what's happening, you can use the `-t` options:

```
$ buildr -t
```

Using `-t` will also display backtraces if and when they occur.

Many components can be individually configured to display more output if you're debugging a specific area of your build. For instance, you could use `--trace=javac,groovyc` to enable tracing of the Java and Groovy compilers:

```
$ buildr --trace=javac,groovyc
```

If you don't know which tracing category you need to enable or if you want a full firehose worth of traces, you can enable all traces:

```
$ buildr --trace=all
```

## JavaRebel Integration

[JavaRebel](#) is a live bytecode reloading solution by Zero Turnaround. It's a lot like the hot code reload feature found in many Java IDE debuggers (like Eclipse and IntelliJ), but capable of handling things like member addition or removal and new class definition. The tool itself is commercial and works with any JVM language, but they do offer a free license for use with Scala classes only.

If available, Buildr will use JavaRebel when configuring the launch for each interactive shell. Shells launched with JavaRebel integration will automatically receive recompiled changes on the fly without any need to restart the shell. There are some limitations to this which are specific to the shell in question, but for the most part, things Just Work.

JavaRebel auto-magical integration is done by searching for a valid JavaRebel install path in the following list of environment variables (in order):

- REBEL\_HOME
- JAVA\_REBEL
- JAVAREBEL
- JAVAREBEL\_HOME

These environment variables may point to either the JavaRebel install directory (e.g. `~/javarebel-2.0.1`), or the JAR file itself (e.g. `~/javarebel-2.0.1/javarebel.jar`). If the path is valid, Buildr will automatically append the appropriate JVM switches to the launch configurations of each shell:

```

$ buildr shell
(in ~/foo, development)
Compiling foo into ~/foo/target/classes
Running java scala.tools.nsc.MainGenericRunner

#####

ZeroTurnaround JavaRebel 2.0.1 (200905251513)
(c) Copyright Webmedia, Ltd, 2007-2009. All rights reserved.

This product is licensed to Daniel Spiewak
for personal use only.

#####

Welcome to Scala version 2.7.4.final (Java HotSpot(TM) 64-Bit Server VM, Java
1.6.0_03-p3).
Type in expressions to have them evaluated.
Type :help for more information.

scala>

```

Note that Buildr does **not** check to make sure that you have a valid JavaRebel license, so you may end up launching with JavaRebel configured but without the ability to use it (in which case, JavaRebel will print a notification).

## Running Your Application

---

The run task lets you easily run programs from your buildfile, such as launching your own application.

In its simplest form, you simply define the main class of your Java application,

```
define 'my-project' do
  compile.with COMMONS_IO, HTTPCLIENT
  run.using :main => "org.example.Main"
end
```

And then run,

```
~/my-project$ buildr run
```

which would launch your application using the project's compile classpath.

It's also possible to pass arguments to the JVM using the `:java_args` option:

```
run.using :main => "org.example.Main",
          :java_args => ["-server"]
```

If your application requires arguments, you can pass in an array of values for the `:main` option, or provide a set of system properties using `:properties`.

```
run.using :main => ["org.example.Main", "-t", "input.txt"],
          :properties => { :debug => "true" }
```

The `run` task is a local task, which means that Buildr will automatically pick the `run` task matching the project in the current directory. Executing the following command:

```
~/my-project/subproject$ buildr run
```

will run the `my-project:subproject:run` task, assuming `my-project` is your top-level project.

Here is a summary of `run.using` options,

Option	Description ...
<code>:main</code>	The java main class, e.g. "com.example.Main". Can also be an array if the main class requires arguments.
<code>:properties</code>	A hash of system properties to be passed to java.
<code>:java_args</code>	An array of additional parameters to pass to java

`:classpath` An array of additional classpath elements (i.e. artifacts, files, etc.). By default, the run task automatically uses the `compile.dependencies`, `test.dependencies` and `test.compile.target` of your project.

---



The run task also detects and uses JavaRebel if it's available. See the [JavaRebel](#) section for details.

## 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.

## BuildrServer

---

Buildr provides an addon that allows you start a [dRuby](#) server hosting a buildfile, so that you can later invoke tasks on it without having to load the complete buildr runtime again.

Usage:

```
buildr -r buildr/drb drb:start
```

To stop the BuildrServer simply use Ctrl+C or kill the process.

Once the server has been started you can invoke tasks using a simple script:

```
#!/usr/bin/env ruby
require 'rubygems'
require 'buildr/drb'
Buildr::DRbApplication.run
```

Save this script as `dbuildr`, make it executable and use it to invoke tasks.

```
$ dbuildr clean compile
```

The `dbuildr` command will start the `BuildrServer` if there isn't one already running. Subsequent calls to `dbuildr` will act as the client and invoke the tasks you provide to the server. If the buildfile has been modified it will be reloaded on the `BuildrServer`.

## Notifications: Growl, Libnotify, Qube

---

Buildr support sending notifications when the build completes or fails, such as displaying the outcome message in an overlaid window on top of other applications.

For OS X users, Buildr supports [Growl](#) out of the box by using the Ruby Cocoa bindings.

For Debian-based Linux users, Buildr supports notifications via the [notify-send](#) command which is part of the “libnotify-bin”:[“http://packages.debian.org/search?keywords=libnotify-bin”](http://packages.debian.org/search?keywords=libnotify-bin) package. Just make sure `notify-send` is installed and on your path is on your `PATH`.

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 the `.buildr` directory under your home directory.

## Eclipse

---

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. To have your libraries' source code available in Eclipse, run the `artifacts:sources` task.

You may explicitly specify the nature of your project, for example if you are developing an Eclipse plugin:

```
define 'my-plugin' do
  eclipse.natures :plugin
end
```

The currently supported natures are :java, :scala and :plugin. Buildr will attempt to auto-detect your project type and apply the most relevant settings by default. If it doesn't or you need something special, you may also explicitly set the nature, container and builders of your project by doing:

```
define 'custom-plugin' do
  eclipse.natures 'org.eclipse.pde.PluginNature'
  eclipse.classpath_containers 'org.eclipse.pde.core.requiredPlugins'
  eclipse.builders ['org.eclipse.pde.ManifestBuilder',
  'org.eclipse.pde.SchemaBuilder']
end
```

One more thing; these settings are inherited hierarchically so you may set them on a parent project if you want to share them across different projects.

## IntelliJ IDEA

---

If you use IntelliJ IDEA, you can generate project files by issuing:

```
$ buildr idea:generate
```

This task will generate a .iml file for every project (or subproject) and a .ipr that you can directly open for the root project.

The generated project files can be removed by issuing:

```
$ buildr idea:clean
```

The idea task generates the project files based on the settings of each project and idea extension specific settings. The main and test source trees are added to the .iml file for each project as are the respective resource directories. The target and report directories are excluded from the project. If the project files exist on the file system the extension will replace specific component sections in the xml with the generated component configurations.

Dependencies come in two forms. Dependencies on other projects and dependencies on external jars. Dependencies on other projects are added as module dependencies in the `.iml` while jars are added as regular file dependencies. Dependencies are exported from the `.iml` file if they are compile dependencies. If a artifact that matches dependency but has a classifier of 'sources' is present then it is configured as the source for the dependency. Note: Use "buildr artifacts:sources" to download the source for dependencies.

## Idea Specific Directives

The extension specific settings of sub-projects inherit the parent projects settings unless overwritten.

## Project file naming

The extension will use the last element of the projects name when generating the `.ipr` and `.iml` files. i.e. A project named "foo" will generate "foo.iml" and "foo.ipr" while a project named "foo:bar" will generate "bar/bar.iml" and no ipr. (The `.ipr` project files are only generated for the base project). The name can be modified by setting the "ipr.suffix" or "iml.suffix" settings which specifies the suffix appended to the file names. The user can also override the name completely by setting "ipr.id" or "iml.id".

### Example: Setting id

```
define "foo" do
  ipr.id = "beep"
  define "bar" do
    iml.id = "baz"
  end
end
```

Will generate:

```
beep.ipr
foo.iml
bar/baz.iml
```

**Example: Setting suffix**

```
define "foo" do
  ipr.suffix = "-suffix1"
  iml.suffix = "-suffix2"
  define "bar"
end
```

Will generate:

```
foo-suffix1.ipr
foo-suffix2.iml
bar/bar-suffix2.iml
```

**Disabling project file generation**

The extension will not generate an iml file for a project if the “project.no\_iml” method is invoked. Generation of ipr files can be disabled by invoking the method “project.no\_ipr”.

**Example**

```
define "foo" do
  project.no_ipr
  define "bar" do
    project.no_iml
  end
end
```

Will generate:

```
foo.iml
```

## Disabling generation of content section in .iml file

The extension will not generate a content section in an iml file if the “iml.skip\_content!” method is invoked. This can be useful if a project is just exporting dependencies and has no associated source code. This may also be of use in scenarios where the build is repackaging an existing jar with more meta-data or the project is just a container for other projects.

### Example

```
define "foo" do
  iml.skip_content!
end
```

## VCS Integration

The extension will attempt to guess the VCS type of the project by looking for a .svn or .git directory in the base projects directory. If either of these are set it will configure the component as appropriate. Otherwise the user will need to manually specify the project to one of either ‘Git’ or ‘svn’ using the ipr.vcs setting.

### Example

```
define "foo" do
  ipr.vcs = 'Git'
end
```

## Adding main, test or exclude paths to the .iml file

The extension allows you to add source paths, test source paths or add paths to the excluded set by modifying the “iml.main\_source\_directories”, “iml.test\_source\_directories” or “iml.excluded\_directories” settings respectively. This is only needed when the defaults inherited from project.compile or project.test are not sufficient.

**Example**

```
define "foo" do
  # Add path for generated resources to .iml file
  iml.main_source_directories << _("generated/main/resources")

  # Add path for generated test resources to .iml file
  iml.test_source_directories << _("generated/test/resources")

  # Exclude the temp directory created during testing
  iml.excluded_directories << _("tmp")

  ...
end
```

**Adding main or test dependencies to the .iml file**

The extension allows you to add main or test dependencies by modifying the “`iml.main_dependencies`” or “`iml.test_dependencies`” settings respectively. This is only needed when the defaults inherited from `project.compile` or `project.test` are not sufficient. Note: These dependencies are not included on compile path when running buildr.

**Example**

```
define "foo" do
  # Add idea specific jar dependency to .iml file
  iml.main_dependencies << 'group:id:jar:1.0'

  # Add idea specific test jar dependency to .iml file
  iml.test_dependencies << 'group:id:jar:1.0'

  ...
end
```

## Dependency generation

A file dependency that exists in the local maven 2 repository is stored in the IML file relative to the `$MAVEN_REPOSITORY$` environment variable (that defaults to `~/.m2/repository`). The user can override the environment variable by setting the “`iml.local_repository_env_override`” setting. If the dependency does not exist in to maven repository or the “`iml.local_repository_env_override`” setting is set to nil, then the path stored in the IML is relative to the IML file.

### Example: Setting `local_repository_env_override`

```
define "foo" do
  iml.local_repository_env_override = nil
  compile.with 'group:id:jar:1.0'
end
```

Will generate a dependency with a path like:

```
jar:///home/peter/.m2/repository/group/id/1.0/id-1.0.jar!/  
rather than the default
```

```
jar://$MAVEN_REPOSITORY$/group/id/1.0/id-1.0.jar!/  
rather than the default
```

### Example: A dependency outside the maven repository

```
define "foo" do
  compile.with _("foos-dep.jar")
end
```

Will generate a dependency with a path like:

```
jar://$MODULE_DIR$/foo-dep.jar!/  
rather than the default
```

## Module Facets

Facets are IDEAs mechanism for adding support for languages, tools and frameworks other than core java. A facet can be added to a project so that it can be deployed as a web application or a hibernate application. A facet can also be used to provide support for other languages such as ruby and scala. The extension makes it possible to generate .iml with the appropriate facets via the “iml.add\_facet” method. It should be noted that facets are NOT inherited by sub-projects.

### Example

This example adds the web facet to a project.

```
define "foo" do
  iml.add_facet("Web","web") do |facet|
    facet.configuration do |conf|
      conf.descriptors do |desc|
        desc.deploymentDescriptor :name => 'web.xml',
          :url => "file://$MODULE_DIR$/src/main/webapp/WEB-INF/web.xml",
          :optional => "false", :version => "2.4"
      end
      conf.webroots do |webroots|
        webroots.root :url => "file://$MODULE_DIR$/src/main/webapp",
          :relative => "/"
      end
    end
  end
end
```

## Custom Component Sections

If the extension does not provide capability to generate configuration for a particular IDEA plugin the user can provide their own configuration data via the “ipr.add\_component” or “iml.add\_component” methods.

**Example: Adding .ipr specific component**

This example changes the compiler configuration for project.

```
define "foo" do
  ipr.add_component("CompilerConfiguration") do |component|
    component.option :name => 'DEFAULT_COMPILER', :value => 'Javac'
    component.option :name => 'DEPLOY_AFTER_MAKE', :value => '0'
    component.resourceExtensions do |xml|
      xml.entry :name => '.+\.nonexistent'
    end
    component.wildcardResourceExtensions do |xml|
      xml.entry :name => '?*.nonexistent'
    end
  end
end
```

**Example: Adding .iml specific component**

This example adds the web facet to a project. Note: This overrides the facets defined by the “iml.add\_facet” method.

```

define "foo" do
  iml.add_component("FacetManager") do |component|
    component.facet :type => 'web', :name => 'Web' do |facet|
      facet.configuration do |conf|
        conf.descriptors do |desc|
          desc.deploymentDescriptor :name => 'web.xml',
            :url => "file://$MODULE_DIR$/src/main/webapp/WEB-INF/web.xml",
            :optional => "false", :version => "2.4"
        end
      end
      conf.webroots do |webroots|
        webroots.root :url => "file://$MODULE_DIR$/src/main/webapp",
:relative => "/"
      end
    end
  end
end
end
end
end

```

## Templates

The underlying project files are xml the contain elements for a number of “components”. The extension will load any existing project files and replace or add any component elements that are generated by the extension. The extension also allows the user to specify a template with either “ipr.template” or “iml.template” settings. If a template is specified it will be loaded and any component elements in these documents will be merged into the base document prior to merging in generated sections. Templates are useful if you want to enforce certain configuration options (i.e. project specific code style).

### Example

```

define "foo" do
  ipr.template = 'project.ipr.template'
  iml.template = 'module.iml.template'
end

```

## Groups

IDEA provides the facility to organise modules into groups. By default the extension does not do this but it can be enabled by “iml.group” setting. If that setting is set to true then the .iml file will be placed in a group based on the parent projects name. If the setting is a string then that is used as the name of the group.

### Example

```
define "foo" do
  iml.group = true
  define 'bar' do
    define 'baz'
  end
  define 'rab' do
    iml.group = "MyGroup"
  end
end
```

Will place the generated .imls in the following groups:

```
foo.iml           => ''
bar/bar.iml       => 'foo'
bar/baz/baz.iml   => 'foo/bar'
rab/rab.iml       => 'MyGroup'
```

### Add Extra .iml files to .ipr

The ‘ipr.extra\_modules’ setting makes it possible to add extra modules to the generated iml file. The setting is an array of file names relative to the base project directory.

**Example**

```
define "foo" do
  ipr.extra_modules << 'other.iml'
  ipr.extra_modules << 'other_other.iml'
end
```

Will add the 'other.iml' and 'other\_other.iml' files to the .ipr project files.

**Buildr plugin for IDEA**

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 main 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.

Cobertura also provides a `cobertura:check` task. This task is intended to be used as a dependency for other tasks (such as `deploy`) which might wish to fail if coverage is unacceptable. The respective thresholds for task failure may be defined using the `cobertura.check` configuration namespace. For example:

```
define 'someModule' do
  cobertura.check.branch_rate = 75
  cobertura.check.line_rate = 100
  cobertura.check.total_line_rate = 98

  task(:deploy).enhance 'cobertura:check'
end
```

The `cobertura:check` task supports all of the configuration parameters allowed by the `cobertura-check` Ant task (as [documented here](#)). Configuration parameters are “Ruby-ized” (as demonstrated in the example above).

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/java/cobertura'
require 'buildr/java/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 the `.buildr` directory under 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/java/cobertura cobertura:html
```

## JAXB Xjc Compiler

---

Buildr includes an extension that provides the ability to invoke `jaxb xjc` binding compiler. A typical project that uses the extension may look something like;

```
require 'buildr/jaxb_xjc'

define "foo" do
  project.version = "1.0.0"
  compile.from compile_jaxb(['src/schemas/wildfire-1.3.xsd'],
                           "-quiet",
                           :package => "org.foo.api")

  package :jar
end
```

The method `compile_jaxb` accepts either an array of files or a single file as the first parameter. It then accepts 0 or more arguments that are passed to the underlying XJC compiler. The arguments are documented on the [jaxb site](#). If the last argument is an options hash then the extension handles the options hash specially. The supported options include:

- `:directory`: The directory to which source is generated. Defaults to `_( :target, :generated, :jaxb)`
- `:keep_content`: By default the generated directory will be deleted. If `true` is specified for this parameter the directory will not be deleted.

- `:package`: The package in which the source is generated.

## Anything Ruby Can Do

---

Buildr is Ruby code. That's an implementation detail for some, but a useful feature 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('src/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
  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 .....	145
Creating Extensions .....	147
Using Alternative Layouts .....	150

## 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=>_('src/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=>_('src/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, Git modules, or whichever cross-repository feature your source control system supports. 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.

To summarize, there are several common ways to distribute extensions:

- Put them in the same place (e.g. `~/ .buildr`) and require them from your `buildfile`
- Put them directly in the project, typically under the `tasks` directory.
- Put them in a shared code repository, and link to them from your project's `tasks` directory
- As Ruby gems and specify which gems are used in the settings file

You can also get creative and devise your own way to distribute extensions.

[Sake](#) is a good example of such initiative that lets you deploy Rake tasks on a system-wide basis.

## 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
<code>first_time</code>	This block will be called once for any particular extension. You can use this to setup top-level and local tasks.
<code>before_define</code>	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.
<code>after_define</code>	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.
    project.recursive_task 'loc' do |task|
      lines = task.prerequisites.map { |path|
        Dir["#{path}/**/*"]
      }.flatten.uniq.inject(0) { |total, file|
        total = 0 if total.nil?
        if File.file? file then
          total + File.readlines(file).count
        end
      }
      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' => project.compile.sources + project.test.sources)
  end

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

```

end

class Buildr::Project
  include LinesOfCode
end

```

You may find interesting that this Extension API is used pervasively inside Buildr itself. Many of the standard tasks such as `compile`, `test`, `package` are extensions to a very small core.

Starting with Buildr 1.4, it's possible to define ordering between `before_define` and `after_define` code blocks in a way similar to Rake's dependencies. For example, if you wanted to override `project.test.compile.from` in `after_define`, you could do so by in

```

after_define(:functional_tests) do |project|
  # Change project.test.compile.from if it's not already pointing
  # to a location with Java sources
  if Dir["#{project.test.compile.from}/**/*.java"].size == 0 &&
    Dir["#{project._(:src, 'test-functional', :java)}/**/*.java"].size > 0
    project.test.compile.from project._(:src, 'test-functional', :java)
  end
end

# make sure project.test.compile.from is updated before the
# compile extension picks up its value
after_define(:compile => :functional_test)

```

Core extensions provide the following named callbacks: `compile`, `test`, `build`, `package` and `check`.

## 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
<code>:source, :main, &lt;lang/usage&gt;</code>	Directory containing source files for a given language or usage, for example, <code>:java, :resources, :webapp</code> .
<code>:source, :test, &lt;lang/usage&gt;</code>	Directory containing test files for a given language or usage, for example, <code>:java, :resources</code> .
<code>:target, :generated</code>	Target directory for generated code (typically source code).
<code>:target, :main, &lt;lang/usage&gt;</code>	Target directory for compiled code, for example, <code>:classes, :resources</code> .

<code>:target, :test,</code> <code>&lt;lang/usage&gt;</code>	Target directory for compile test cases, for example, <code>:classes,</code> <code>:resources.</code>
<code>:reports,</code> <code>&lt;framework/usage&gt;</code>	Target directory for generated reports, for example, <code>:junit,</code> <code>:coverage.</code>

---

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
```

# Contributing

---

Getting involved.....	153
Mailing Lists.....	153
Bugs (aka Issues).....	154
Community Wiki.....	154
Contributing Code.....	154
Living on the edge.....	156
Tested and Documented.....	158
Documentation.....	159
Contributors.....	161

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 [users](#) mailing list for developers working with Buildr, that would be you if you're using Buildr or interested in using it. There's the [dev](#) mailing list for talking about development of Buildr itself. There's also [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.

## IRC

---

We are live on IRC under the buildr channel on [irc.freenode.net](https://irc.freenode.net), with a broad coverage of the US timezone. We tend to idle there, so feel free to ping the channel owner (toulmean) to make noise.

Our “conversations”: are logged by the [echelog](#) and [irclogger](#) bots. If you’re really curious, we also have [activity statistics](#)

Sincere thanks to Matzon and Christopher Schneider for setting these up!

## 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](#).

## 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). This 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/buildr/trunk buildr
```

You can also browse the [Buildr repository](#).

### Git

Not a fan of SVN? We understand. You can also use the [official Apache Git clone](#). This clone is maintained by the ASF and kept in sync with the SVN repository (though, in practice there may be some delay in cloning recent commits). Apache's Git hosting supports both `git://` and `http://` protocols (you should use `git://` if at all possible as it is faster than `http://`):

```
$ git clone git://git.apache.org/buildr.git
# or...
$ git clone http://git.apache.org/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.

### GitHub

You are also welcome to fork or clone the [Buildr repository on GitHub](#). This repository is just an exact mirror of the official Apache Git clone referenced above (updated every 30 minutes). Some of the core committers also maintain their own forks of Buildr on GitHub, often containing experimental and in-progress development slated for eventual inclusion into the SVN. For reference, they are listed below:

- [Assaf Arkin](#)
- [Victor Hugo Borja](#)
- [Daniel Spiewak](#)

## 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/](http://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 Git/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](#).

Syntax highlighting handled by [Pygments](#). Use the special `highlight` tag to separate code sample from the rest of the text and to tell Pygments which language to use. For example:

```
{% highlight ruby %}  
define 'project' do  
  # Just a sample  
end  
{% endhighlight %}
```

Have a look at existing documentation to see writing conventions, specifically:

- Separate paragraphs with two newlines.
- Use one newline only if you need a `<br>` tag, otherwise, no newlines inside the paragraph.
- When creating a new page, don't forget the YAML preamble at the top (Jekyll needs the page title and layout).
- The layout uses H1 to render the page title; only use H2 through H4 for the page content.
- Use H2 headers for the major page sections. Give each H2 header a unique ID so the table of contents can link to it.
- Separating sentences with two spaces, just a convenience when editing in a text editor using monospaced fonts.
- If in doubt, ask.

To go from Textile to HTML we use [Jekyll](#). You can use the `jeekyll` rake task to transform the files under `doc` and create a copy of the Web site in the directory `_site`. For example:

```
$ rake jeekyll  
$ open _site/index.html
```

There is no live editing, but you can run `rake jeekyll auto=true`, and when you update and save a Textile page it will regenerate the corresponding HTML page.

To go from HTML to PDF we use [PrinceXML](#). The target file `buildr.pdf` is generated by first running Jekyll and then merging the generated HTML pages into a single PDF document. For example:

```
$ rake buildr.pdf  
$ open buildr.pdf
```

## Contributors

---

Here is the list of people who are actively working and committing on Buildr:

**Assaf Arkin** (assaf at apache.org)

Started working on Buildr because Maven was too much pain and Rake wasn't enough. Assaf has been hanging around Apache since 1999, as founding contributor to XML Apache, Ode and Buildr. Assaf is also co-author of [Ruby In Practice](#).

**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.

**Daniel Spiewak** (djspiewak at apache.org)

Daniel originally came to Buildr in search of a Scala build tool which was better than Ant. He got more than he bargained for. Now, he works to advance Buildr as the absolute best tool for supporting Scala development.

**Antoine Toulme** (toulmean at apache.org)

Antoine used Buildr first as an excuse to evade in Ruby land, creating plugins for Debian packaging, GWT compilation, or the NSIS installer. His main area of interest is the resolving of dependencies in the OSGi world. He works on making Buildr a standalone rock solid tool.

- Peter Donald

Peter already used rake to automate jobs in his ruby and java projects. When it came time to upgrade that home grown ant/java/rake build system Buildr seemed the perfect match.