GRAILS

spring source

SEE THE LIGHT - AGILE, INDUSTRIAL STRENGTH, RAPID WEB APPLICATION DEVELOPMENT MADE EASY

The Grails Framework - Reference Documentation

Authors: Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown

Version: 1.3.1

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

3

4

1. Introduction

Java web development as it stands today is dramatically more complicated than it needs to be. Most modern web frameworks in the Java space are over complicated and don't embrace the Don't Repeat Yourself (DRY) principles. Dynamic frameworks like Rails, Django and TurboGears helped pave the way to a more modern way of thinking about web applications. Grails builds on these concepts and dramatically reduces the complexity of building web applications on the Java platform. What makes it different, however, is that it does so by building on already established Java technology like Spring & Hibernate.

Grails is a full stack framework and attempts to solve as many pieces of the web development puzzle through the core technology and it's associated plug-ins. Included out the box are things like:

- An easy to use Object Relational Mapping (ORM) layer built on Hibernate
- An expressive view technology called Groovy Server Pages (GSP)
- A controller layer built on Spring MVC
- A command line scripting environment built on the Groovy-powered Gant
- An embedded Tomcat container which is configured for on the fly reloading
- Dependency injection with the inbuilt Spring container
- Support for internationalization (i18n) built on Spring's core MessageSource concept
- A transactional service layer built on Spring's transaction abstraction

All of these are made easy to use through the power of the Groovy language and the extensive use of Domain Specific Languages (DSLs)

This documentation will take you through getting started with Grails and building web applications with the Grails framework.

2. Getting Started

2.1 Downloading and Installing

The first step to getting up and running with Grails is to install the distribution. To do so follow these steps:

- Download a binary distribution of Grails and extract the resulting zip file to a location of your choice
- Set the GRAILS_HOME environment variable to the location where you extracted the zip
  - On Unix/Linux based systems this is typically a matter of adding something like the following `export GRAILS_HOME=/path/to/grails` to your profile
  - On Windows this is typically a matter of setting an environment variable under `My Computer/Advanced/Environment Variables`
- Now you need to add the `bin` directory to your `PATH` variable:
  - On Unix/Linux base system this can be done by doing a `export PATH="$PATH:$GRAILS_HOME/bin"`
  - On windows this is done by modifying the `Path` environment variable under `My Computer/Advanced/Environment Variables`

If Grails is working correctly you should now be able to type `grails` in the terminal window and see output similar to the below:

```
Welcome to Grails 1.0 - http://grails.org/
Licensed under Apache Standard License 2.0
Grails home is set to: /Developer/grails-1.0
No script name specified. Use 'grails help' for more info
```

2.2 Upgrading from previous versions of Grails

Although the Grails development team have tried to keep breakages to a minimum there are a number of items to consider when upgrading a Grails 1.0.x, 1.1.x, or 1.2.x applications to Grails 1.3. The major changes are described in detail below.

Upgrading from Grails 1.2.x

Plugin Repositories
As of Grails 1.3, Grails no longer natively supports resolving plugins against secured SVN repositories. Grails 1.2 and below's plugin resolution mechanism has been replaced by one built on Ivy the upside of which is that you can now resolve Grails plugins against Maven repositories as well as regular Grails repositories.
Ivy supports a much richer setter of repository resolvers for resolving plugins with, including support for Webdav, HTTP, SSH and FTP. See the section on resolvers in the Ivy docs for all the available options and the section of plugin repositories in the user guide which explains how to configure additional resolvers.
If you still need support for resolving plugins against secured SVN repositories then the IvySvn project provides a set of Ivy resolvers for resolving against SVN repositories.

Upgrading from Grails 1.1.x

Plugin paths
In Grails 1.1.x typically a `pluginContextPath` variable was used to establish paths to plugin resources. For example:

```
<g:resource dir="${pluginContextPath}/images" file="foo.jpg" />
```

In Grails 1.2 views have been made plugin aware and this is no longer necessary:

```
<g:resource dir="images" file="foo.jpg" />
```

Additionally the above example will no longer link to an application image from a plugin view. To do so you need to change the above to:

```
<g:resource contextPath="" dir="images" file="foo.jpg" />
```

The same rules apply to the [javascript](#) and [render](#)

Tag and Body return values

Tags no longer return `java.lang.String` instances but instead return a `StreamCharBuffer` instance. The `StreamCharBuffer` class implements all the same methods as `String`, however code like this may break:

```
def foo = body()
if(foo instanceof String) {
    // do something
}
```

In these cases you should use the `java.lang.CharSequence` interface, which both `String` and `StreamCharBuffer` implement:

```
def foo = body()
if(foo instanceof CharSequence) {
    // do something
}
```

New JSONBuilder

There is a new version of `JSONBuilder` which is semantically different to earlier versions of Grails. However, if your application depends on the older semantics you can still use the now deprecated implementation by settings the following property to `true` in Config.groovy:

```
grails.json.legacy.builder=true
```

Validation on Flush

Grails now executes validation routines when the underlying Hibernate session is flushed to ensure that no invalid objects are persisted. If one of your constraints (such as a custom validator) is executing a query then this can cause an addition flush resulting in a `StackOverflowError`. Example:

```
static constraints = {
    author validator: { a ->
        assert a != Book.findByTitle("My Book").author
    }
}
```

The above code can lead to a `StackOverflowError` in Grails 1.2. The solution is to run the query in a new Hibernate `session` (which is recommended in general as doing Hibernate work during flushing can cause other issues):

```
static constraints = {
    author validator: { a ->
        Book.withNewSession {
            assert a != Book.findByTitle("My Book").author
        }
    }
}
```

Upgrading from Grails 1.0.x

Groovy 1.6

Grails 1.1 and above ship with Groovy 1.6 and no longer supports code compiled against Groovy 1.5. If you have a library that is written in Groovy 1.5 you will need to recompile it against Groovy 1.6 before using it with Grails 1.1.

Java 5.0

Grails 1.1 now no longer supports JDK 1.4, if you wish to continue using Grails then it is recommended you stick to the Grails 1.0.x stream until you are able to upgrade your JDK.

Configuration Changes

1) The setting `grails.testing.reports.destDir` has been renamed to `grails.project.test.reports.dir` for consistency.
2) The following settings have been moved from `grails-app/conf/Config.groovy` to `grails-app/conf/BuildConfig.groovy`:

- `grails.config.base.webXml`
- `grails.war.destFile`
- `grails.war.dependencies`
- `grails.war.copyToWebApp`
- `grails.war.resources`

3) The `grails.war.java5.dependencies` option is no longer supported, since Java 5.0 is now the baseline (see above).
4) The use of jsessionid (now considered harmful) is disabled by default. If your application requires jsessionid you can re-enable its usage by adding the following to `grails-app/conf/Config.groovy`:

```
grails.views.enable.jsessionid=true
```

5) The syntax used to configure Log4j has changed. See the user guide section on Logging for more information.

Plugin Changes

Since 1.1, Grails no longer stores plugins inside your `PROJECT_HOME/plugins` directory by default. This may result in compilation errors in your application unless you either re-install all your plugins or set the following property in `grails-app/conf/BuildConfig.groovy`:

```
grails.project.plugins.dir="./plugins"
```

Script Changes

1) If you were previously using Grails 1.0.3 or below the following syntax is no longer support for importing scripts from GRAILS_HOME:

```
Ant.property(environment:"env")
grailsHome = Ant.antProject.properties."env.GRAILS_HOME"
includeTargets << new File ( "${grailsHome}/scripts/Bootstrap.groovy" )
```

Instead you should use the new `grailsScript` method to import a named script:

```
includeTargets << grailsScript( "Bootstrap.groovy" )
```

2) Due to an upgrade to Gant all references to the variable `Ant` should be changed to `ant`.
3) The root directory of the project is no long on the classpath, the result is that loading a resource like this will no longer work:

```
def stream = getClass().classLoader.getResourceAsStream("grails-app/conf/my-config.xml")
```

Instead you should use the Java File APIs with the `basedir` property:

```
new File("${basedir}/grails-app/conf/my-config.xml").withInputStream { stream ->
        // read the file
}
```

Command Line Changes

The `run-app-https` and `run-war-https` commands no longer exist and have been replaced by an argument to [run-app](#):

```
grails run-app -https
```

Data Mapping Changes

1) Enum types are now mapped using their String value rather than the ordinal value. You can revert to the old behavior by changing your mapping as follows:

```
static mapping = {
        someEnum enumType:"ordinal"
}
```

2) Bidirectional one-to-one associations are now mapped with a single column on the owning side and a foreign key reference. You shouldn't need to change anything, however you may want to drop the column on the inverse side as it contains duplicate data.

REST Support

Incoming XML requests are now no longer automatically parsed. To enable parsing of REST requests you can do so using the `parseRequest` argument inside a URL mapping:

```
"/book"(controller:"book",parseRequest:true)
```

Alternatively, you can use the new `resource` argument, which enables parsing by default:

```
"/book"(resource:"book")
```

2.3 Creating an Application

To create a Grails application you first need to familiarize yourself with the usage of the `grails` command which is used in the following manner:

```
grails [command name]
```

In this case the command you need to execute is [create-app](#):

```
grails create-app helloworld
```

This will create a new directory inside the current one that contains the project. You should now navigate to this directory in terminal:

```
cd helloworld
```

2.4 A Hello World Example

To implement the typical "hello world!" example run the [create-controller](#) command:

```
grails create-controller hello
```

This will create a new controller (Refer to the section on [Controllers](#) for more information) in the `grails-app/controllers` directory called `HelloController.groovy`.
Controllers are capable of dealing with web requests and to fulfil the "hello world!" use case our implementation needs to look like the following:

```groovy
class HelloController {
    def world = {
        render "Hello World!"
    }
}
```

Job done. Now start-up the container with another new command called [run-app](#):

```
grails run-app
```

This will start-up a server on port 8080 and you should now be able to access your application with the URL: `http://localhost:8080/helloworld`
The result will look something like the following screenshot:
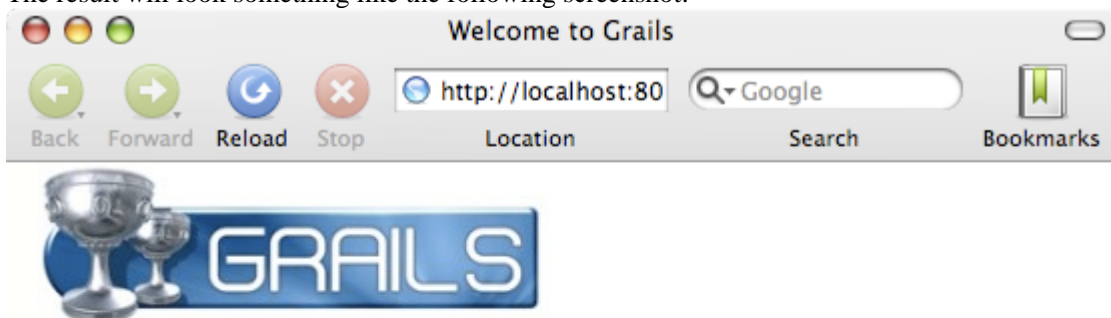
This is the Grails intro page which is rendered by the `web-app/index.gsp` file. You will note it has a detected the presence of your controller and clicking on the link to our controller we can see the text "Hello World!" printed to the browser window.

2.5 Getting Set-up in an IDE

11

IntelliJ IDEA

IntelliJ IDEA and the JetGroovy plug-in offer good support for Groovy & Grails developer. Refer to the section on Groovy and Grails support on the JetBrains website for a feature overview.
To integrate Grails 1.2 to with IntelliJ run the following command to generate appropriate project files:

```
grails integrate-with --intellij
```

NetBeans

A good Open Source alternative is Sun's NetBeans, which provides a Groovy/Grails plugin that automatically recognizes Grails projects and provides the ability to run Grails applications in the IDE, code completion and integration with Sun's Glassfish server. For an overview of features see the NetBeans Integration guide on the Grails website which was written by the NetBeans team.

Eclipse

We recommend that users of Eclipse looking to develop Grails application take a look at SpringSource Tool Suite, which offers built in support for Grails including automatic classpath management, a GSP editor and quick access to Grails commands. See the STS Integration page for an overview.

TextMate

Since Grails' focus is on simplicity it is often possible to utilize more simple editors and TextMate on the Mac has an excellent Groovy/Grails bundle available from the Texmate bundles SVN.
To integrate Grails 1.2 to with TextMate run the following command to generate appropriate project files:

```
grails integrate-with --textmate
```

Alternatively TextMate can easily open any project with its command line integration by issuing the following command from the root of your project:

```
mate .
```

2.6 Convention over Configuration

Grails uses "convention over configuration" to configure itself. This typically means that the name and location of files is used instead of explicit configuration, hence you need to familiarize yourself with the directory structure provided by Grails.
Here is a breakdown and links to the relevant sections:

- `grails-app` - top level directory for Groovy sources
    - `conf` - Configuration sources.
    - `controllers` - Web controllers - The C in MVC.
    - `domain` - The application domain.
    - `i18n` - Support for internationalization (i18n).
    - `services` - The service layer.
    - `taglib` - Tag libraries.
    - `views` - Groovy Server Pages.
- `scripts` - Gant scripts.
- `src` - Supporting sources
    - `groovy` - Other Groovy sources
    - `java` - Other Java sources
- `test` - Unit and integration tests.

2.7 Running an Application

Grails applications can be run with the built in Tomcat server using the run-app command which will load a server on port 8080 by default:

```
grails run-app
```

You can specify a different port by using the `server.port` argument:

```
grails -Dserver.port=8090 run-app
```

More information on the [run-app](#) command can be found in the reference guide.

2.8 Testing an Application

The `create-*` commands in Grails automatically create integration tests for you within the `test/integration` directory. It is of course up to you to populate these tests with valid test logic, information on which can be found in the section on [Testing](#). However, if you wish to execute tests you can run the [test-app](#) command as follows:

```
grails test-app
```

Grails also automatically generates an Ant `build.xml` which can also run the tests by delegating to Grails' [test-app](#) command:

```
ant test
```

This is useful when you need to build Grails applications as part of a continuous integration platform such as CruiseControl.

2.9 Deploying an Application

Grails applications are deployed as Web Application Archives (WAR files), and Grails includes the [war](#) command for performing this task:

```
grails war
```

This will produce a WAR file in the root of your project which can then be deployed as per your containers instructions.

> 🚫 NEVER deploy Grails using the [run-app](#) command as this command sets Grails up for auto-reloading at runtime which has a severe performance and scalability implication

When deploying Grails you should always run your containers JVM with the `-server` option and with sufficient memory allocation. A good set of VM flags would be:

```
-server -Xmx512M
```

2.10 Creating Artefacts

Grails ships with a few convenience targets such as [create-controller](#), [create-domain-class](#) and so on that will create [Controllers](#) and different artefact types for you.

> ⚠ These are merely for your convenience and you can just as easily use an IDE or your favourite text editor.

For example to create the basis of an application you typically need a [domain model](#):

```
grails create-domain-class book
```

This will result in the creation of a domain class at `grails-app/domain/Book.groovy` such as:

```
class Book {
}
```

There are many such `create-*` commands that can be explored in the command line reference guide.

2.10 Supported Java EE Containers

Grails runs on any Servlet 2.4 and above container and is known to work on the following specific container products:

- Tomcat 5.5
- Tomcat 6.0
- SpringSource tc Server
- SpringSource dm Server 1.0
- GlassFish v1 (Sun AS 9.0)
- GlassFish v2 (Sun AS 9.1)
- GlassFish v3 Prelude
- Sun App Server 8.2
- Websphere 6.1
- Websphere 5.1
- Resin 3.2
- Oracle AS
- JBoss 4.2
- Jetty 6.1
- Jetty 5
- Weblogic 7/8/9/10

Some containers have bugs however, which in most cases can be worked around. A [list of known deployment issues](#) can be found on the Grails wiki.

2.11 Generating an Application

To get started quickly with Grails it is often useful to use a feature called [Scaffolding](#) to generate the skeleton of an application. To do this use one of the `generate-*` commands such as [generate-all](#), which will generate a [controller](#) and the relevant [views](#):

```
grails generate-all Book
```

3. Configuration

It may seem odd that in a framework that embraces "convention-over-configuration" that we tackle this topic now, but since what configuration there is typically a one off, it is best to get it out the way.
With Grails' default settings you can actually develop and application without doing any configuration whatsoever. Grails ships with an embedded container and in-memory HSQLDB meaning there isn't even a database to set-up. However, typically you want to set-up a real database at some point and the way you do that is described in the following section.

3.1 Basic Configuration

For general configuration Grails provides a file called `grails-app/conf/Config.groovy`. This file uses Groovy's [ConfigSlurper](#) which is very similar to Java properties files except it is pure Groovy hence you can re-use variables and use proper Java types!
You can add your own configuration in here, for example:

```
foo.bar.hello = "world"
```

Then later in your application you can access these settings in one of two ways. The most common is via the [GrailsApplication](#) object, which is available as a variable in controllers and tag libraries:

```
assert "world" == grailsApplication.config.foo.bar.hello
```

The other way involves getting a reference to the [ConfigurationHolder](#) class that holds a reference to the configuration object:

```
import org.codehaus.groovy.grails.commons.*
…
def config = ConfigurationHolder.config
assert "world" == config.foo.bar.hello
```

3.1.1 Built in options

Grails also provides the following configuration options:

- `grails.config.locations` - The location of properties files or addition Grails Config files that should be merged with main configuration
- `grails.enable.native2ascii` - Set this to false if you do not require native2ascii conversion of Grails i18n properties files
- `grails.views.default.codec` - Sets the default encoding regime for GSPs - can be one of 'none', 'html', or 'base64' (default: 'none'). To reduce risk of XSS attacks, set this to 'html'.
- `grails.views.gsp.encoding` - The file encoding used for GSP source files (default is 'utf-8')
- `grails.mime.file.extensions` - Whether to use the file extension to dictate the mime type in [Content Negotiation](#)
- `grails.mime.types` - A map of supported mime types used for [Content Negotiation](#)
- `grails.serverURL` - A string specifying the server URL portion of absolute links, including server name e.g. grails.serverURL="http://my.yourportal.com". See [createLink](#).

War generation

- `grails.war.destFile` - Sets the location where the [war](#) command should place the generated WAR file
- `grails.war.dependencies` - A closure containing Ant builder syntax or a list of JAR filenames. Allows you to customise what libaries are included in the WAR file.
- `grails.war.java5.dependencies` - A list of the JARs that should be included in the WAR file for JDK 1.5 and above.
- `grails.war.copyToWebApp` - A closure containing Ant builder syntax that is legal inside an Ant copy, for example "fileset()". Allows you to control what gets included in the WAR file from the "web-app"

directory.
- grails.war.resources - A closure containing Ant builder syntax. Allows the application to do any other pre-warring stuff it needs to.

For more information on using these options, see the section on [deployment](#)

3.1.2 Logging

Logging Basics

Grails uses its common configuration mechanism to configure the underlying [Log4j](#) log system. To configure logging you must modify the file Config.groovy located in the grails-app/conf directory.

This single Config.groovy file allows you to specify separate logging configurations for development, test, and production [environments](#). Grails processes the Config.groovy and configures Log4j appropriately.

Since 1.1 Grails provides a Log4j DSL, that you can use to configure Log4j an example of which can be seen below:

```
log4j = {
    error  'org.codehaus.groovy.grails.web.servlet',  //  controllers
           'org.codehaus.groovy.grails.web.pages' //  GSP
    warn   'org.mortbay.log'
}
```

Essentially, each method translates into a log level and you can pass the names of the packages you want to log at that level as arguments to the method.

Some useful loggers include:

- org.codehaus.groovy.grails.commons - Core artefact information such as class loading etc.
- org.codehaus.groovy.grails.web - Grails web request processing
- org.codehaus.groovy.grails.web.mapping - URL mapping debugging
- org.codehaus.groovy.grails.plugins - Log plugin activity
- org.springframework - See what Spring is doing
- org.hibernate - See what Hibernate is doing

The Root Logger

The Root logger is the logger that all other loggers inherit from. You can configure the Root logger using the root method:

```
root {
    error()
    additivity = true
}
```

The above example configures the root logger to log messages at the error level and above to the default standard out appender. You can also configure the root logger to log to one or more named appenders:

```
appenders {
    file name:'file', file:'/var/logs/mylog.log'
}
root {
    debug 'stdout', 'file'
    additivity = true
}
```

Here the root logger will log to two appenders - the default 'stdout' appender and a 'file' appender.

You can also configure the root logger from the argument passed into the Log4J closure:

```
log4j = { root ->
    root.level = org.apache.log4j.Level.DEBUG
    …
}
```

The closure argument "root" is an instance of `org.apache.log4j.Logger` , so refer to the API documentation for Log4J to find out what properties and methods are available to you.

Custom Appenders
Using the Log4j you can define custom appenders. The following appenders are available by default:

- `jdbc` - An appender that logs to a JDBC connection
- `console` - An appender that logs to standard out
- `file` - An appender that logs to a file
- `rollingFile` - An appender that logs to rolling set of files

For example to configure a rolling file appender you can do:

```
log4j = {
    appenders {
        rollingFile name:"myAppender", maxFileSize:1024, file:"/tmp/logs/myApp.log"
    }
}
```

Each argument passed to the appender maps to a property of underlying [Appender](#) class. The example above sets the `name`, `maxFileSize` and `file` properties of the [RollingFileAppender](#) class.
If you prefer to simply create the appender programmatically yourself, or you have your own appender implementation then you can simply call the `appender` method and appender instance:

```
import org.apache.log4j.*
log4j = {
    appenders {
        appender new RollingFileAppender(name:"myAppender", maxFileSize:1024, file:"/tmp/log
    }
}
```

You can then log to a particular appender by passing the name as a key to one of the log level methods from the previous section:

```
error myAppender:"org.codehaus.groovy.grails.commons"
```

Custom Layouts
By default the Log4j DSL assumes that you want to use a [PatternLayout](#). However, there are other layouts available including:

- `xml` - Create an XML log file
- `html` - Creates an HTML log file
- `simple` - A simple textual log
- `pattern` - A Pattern layout

You can specify custom patterns to an appender using the `layout` setting:

```
log4j = {
    appenders {
        console name:'customAppender', layout:pattern(conversionPattern: '%c{2} %m%n')
    }
}
```

This also works for the built-in appender "stdout", which logs to the console:

```
log4j = {
    appenders {
        console name:'stdout', layout:pattern(conversionPattern: '%c{2} %m%n')
    }
}
```

Full stacktraces

When exceptions occur, there can be an awful lot of noise in the stacktrace from Java and Groovy internals. Grails filters these typically irrelevant details and restricts traces to non-core Grails/Groovy class packages.
When this happens, the full trace is always written to the `StackTrace` logger. This logs to a file called `stacktrace.log` - but you can change this in your `Config.groovy` to do anything you like. For example if you prefer full stack traces to go to standard out you can add this line:

```
error stdout:"StackTrace"
```

You can completely disable stacktrace filtering by setting the `grails.full.stacktrace` VM property to `true`:

```
grails -Dgrails.full.stacktrace=true run-app
```

Logging by Convention

All application artefacts have a dynamically added `log` property. This includes domain classes, controllers, tag libraries and so on. Below is an example of its usage:

```
def foo = "bar"
log.debug "The value of foo is $foo"
```

Logs are named using the convention `grails.app.<artefactType>.ClassName`. Below is an example of how to configure logs for different Grails artefacts:

```
log4j = {
    // Set level for all application artefacts
    info "grails.app"
    // Set for a specific controller
    debug "grails.app.controller.YourController"
    // Set for a specific domain class
    debug "grails.app.domain.Book"
    // Set for all taglibs
    info "grails.app.tagLib"
}
```

The artefacts names are dictated by convention, some of the common ones are listed below:

- `bootstrap` - For bootstrap classes
- `dataSource` - For data sources
- `tagLib` - For tag libraries
- `service` - For service classes
- `controller` - For controllers
- `domain` - For domain entities

3.1.3 GORM

Grails provides the following GORM configuration options:

18

- `grails.gorm.failOnError` - If set to `true`, causes the save() method on domain classes to throw a `grails.validation.ValidationException` if [validation](#) fails during a save. This option may also be assigned a list of Strings representing package names. If the value is a list of Strings then the failOnError behavior will only be applied to domain classes in those packages (including sub-packages). See the [save](#) method docs for more information.

Enable failOnError for all domain classes…

```
grails.gorm.failOnError=true
```

Enable failOnError for domain classes by package…

```
grails.gorm.failOnError = ['com.companyname.somepackage', 'com.companyname.someotherpackage'
```

- `grails.gorm.autoFlush` = If set to `true`, causes the [merge](#), [save](#) and [delete](#) methods to flush the session, replacing the need to do something like `save(flush: true)`.

3.2 Environments

Per Environment Configuration

Grails supports the concept of per environment configuration. Both the `Config.groovy` file and the `DataSource.groovy` file within the `grails-app/conf` directory can take advantage of per environment configuration using the syntax provided by [ConfigSlurper](#) As an example consider the following default `DataSource` definition provided by Grails:

```
dataSource {
    pooled = false
    driverClassName = "org.hsqldb.jdbcDriver"
    username = "sa"
    password = ""
}
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'createeate-drop','update'
            url = "jdbc:hsqldb:mem:devDB"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:hsqldb:file:prodDb;shutdown=true"
        }
    }
}
```

Notice how the common configuration is provided at the top level and then an `environments` block specifies per environment settings for the `dbCreate` and `url` properties of the `DataSource`. This syntax can also be used within `Config.groovy`.

Packaging and Running for Different Environments

Grails' [command line](#) has built in capabilities to execute any command within the context of a specific environment. The format is:

```
grails [environment] [command name]
```

In addition, there are 3 preset environments known to Grails: dev, prod, and test for development, production and test. For example to create a WAR for the test environment you could do:

```
grails test war
```

If you have other environments that you need to target you can pass a grails.env variable to any command:

```
grails -Dgrails.env=UAT run-app
```

Programmatic Environment Detection
Within your code, such as in a Gant script or a bootstrap class you can detect the environment using the Environment class:

```
import grails.util.Environment
...
switch(Environment.current) {
    case Environment.DEVELOPMENT:
        configureForDevelopment()
    break
    case Environment.PRODUCTION:
        configureForProduction()
    break
}
```

Per Environment Bootstrapping
Its often desirable to run code when your application starts up on a per-environment basis. To do so you can use the grails-app/conf/BootStrap.groovy file's support for per-environment execution:

```
def init = { ServletContext ctx ->
    environments {
        production {
            ctx.setAttribute("env", "prod")
        }
        development {
            ctx.setAttribute("env", "dev")
        }
    }
    ctx.setAttribute("foo", "bar")
}
```

Generic Per Environment Execution
The previous BootStrap example uses the grails.util.Environment class internally to execute. You can also use this class yourself to execute your own environment specific logic:

```
Environment.executeForCurrentEnvironment {
    production {
        // do something in production
    }
    development {
        // do something only in development
    }
}
```

3.3 The DataSource

Since Grails is built on Java technology to set-up a data source requires some knowledge of JDBC (the technology that doesn't stand for Java Database Connectivity).
Essentially, if you are using another database other than HSQLDB you need to have a JDBC driver. For example for

MySQL you would need [Connector/J](#)

Drivers typically come in the form of a JAR archive. Drop the JAR into your projects `lib` directory.

Once you have the JAR in place you need to get familiar Grails' DataSource descriptor file located at `grails-app/conf/DataSource.groovy`. This file contains the dataSource definition which includes the following settings:

- `driverClassName` - The class name of the JDBC driver
- `username` - The username used to establish a JDBC connection
- `password` - The password used to establish a JDBC connection
- `url` - The JDBC URL of the database
- `dbCreate` - Whether to auto-generate the database from the domain model or not
- `pooled` - Whether to use a pool of connections (defaults to true)
- `logSql` - Enable SQL logging
- `dialect` - A String or Class that represents the Hibernate dialect used to communicate with the database. See the [org.hibernate.dialect](#) package for available dialects.
- `properties` - Extra properties to set on the DataSource bean. See the [Commons DBCP BasicDataSource](#) documentation.

A typical configuration for MySQL may be something like:

```
dataSource {
    pooled = true
    dbCreate = "update"
    url = "jdbc:mysql://localhost/yourDB"
    driverClassName = "com.mysql.jdbc.Driver"
    username = "yourUser"
    password = "yourPassword"
}
```

> When configuring the DataSource do not include the type or the def keyword before any of the configuration settings as Groovy will treat these as local variable definitions and they will not be processed. For example the following is invalid:

```
dataSource {
    boolean pooled = true // type declaration results in local variable
    …
}
```

Example of advanced configuration using extra properties:

```
dataSource {
    pooled = true
    dbCreate = "update"
    url = "jdbc:mysql://localhost/yourDB"
    driverClassName = "com.mysql.jdbc.Driver"
    username = "yourUser"
    password = "yourPassword"
    properties {
        maxActive = 50
        maxIdle = 25
        minIdle = 5
        initialSize = 5
        minEvictableIdleTimeMillis = 60000
        timeBetweenEvictionRunsMillis = 60000
        maxWait = 10000
    }
}
```

3.3.1 DataSources and Environments

The previous example configuration assumes you want the same config for all environments: production, test, development etc.

Grails' DataSource definition is "environment aware", however, so you can do:

```
dataSource {
    // common settings here
}
environments {
  production {
    dataSource {
        url = "jdbc:mysql://liveip.com/liveDb"
    }
  }
}
```

3.3.2 JNDI DataSources

Referring to a JNDI DataSource
Since many Java EE containers typically supply `DataSource` instances via the Java Naming and Directory
Interface (JNDI). Sometimes you are required to look-up a `DataSource` via JNDI.
Grails supports the definition of JNDI data sources as follows:

```
dataSource {
    jndiName = "java:comp/env/myDataSource"
}
```

The format on the JNDI name may vary from container to container, but the way you define the `DataSource`
remains the same.

Configuring a Development time JNDI resource
The way in which you configure JNDI data sources at development time is plugin dependent. Using the Tomcat
plugin you can define JNDI resources using the `grails.naming.entries` setting in
`grails-app/conf/Config.groovy`:

```
grails.naming.entries = [
    "bean/MyBeanFactory": [
        auth: "Container",
        type: "com.mycompany.MyBean",
        factory: "org.apache.naming.factory.BeanFactory",
        bar: "23"
    ],
    "jdbc/EmployeeDB": [
        type: "javax.sql.DataSource", //required
        auth: "Container", // optional
        description: "Data source for Foo", //optional
        driverClassName: "org.hsql.jdbcDriver",
        url: "jdbc:HypersonicSQL:database",
        username: "dbusername",
        password: "dbpassword",
        maxActive: "8",
        maxIdle: "4"
    ],
    "mail/session": [
        type: "javax.mail.Session,
        auth: "Container",
        "mail.smtp.host": "localhost"
    ]
]
```

3.3.3 Automatic Database Migration

The `dbCreate` property of the `DataSource` definition is important as it dictates what Grails should do at runtime
with regards to automatically generating the database tables from GORM classes. The options are:

- `create-drop` - Drops and re-creates the database when Grails starts, and drops the schema at the end of a
  clean shutdown.
- `create` - Drops and re-creates the database when Grails starts, but doesn't drop the schema at the end of a

clean shutdown.
- ○ `update` - Creates the database if it doesn't exist, and modifies it if it does exist. The modifications are rather basic though, and generally only include adding missing columns and tables. Will not drop or modify anything.
- ○ `validate` - Makes no changes to your database. Compares the configuration with the existing database schema and reports warnings.
- ○ any other value - does nothing. Don't specify any value if you want to manage databases yourself or by using a 3rd-party tool.

> 🚫 Both `create-drop` and `create` will destroy all existing data hence use with caution!

In [development](#) mode `dbCreate` is by default set to "create-drop":

```
dataSource {
    dbCreate = "create-drop" // one of 'create', 'create-drop','update'
}
```

What this does is automatically drop and re-create the database tables on each restart of the application. Obviously this may not be what you want in production.

> ⚠️ Although Grails does not currently support Rails-style Migrations out of the box, there are currently three plugins that provide similar capabilities to Grails: Autobase (http://wiki.github.com/RobertFischer/autobase), The [LiquiBase](#) plugin and the [DbMigrate](#) plugin both of which are available via the `grails list-plugins` command

3.3.4 Transaction-aware DataSource Proxy

The actual `dataSource` bean is wrapped in a transaction-aware proxy so you will be given the connection that's being used by the current transaction or Hibernate `Session` if one is active.
If this were not the case, then retrieving a connection from the `dataSource` would be a new connection, and you wouldn't be able to see changes that haven't been committed yet (assuming you have a sensible transaction isolation setting, e.g. `READ_COMMITTED` or better).
The "real" unproxied `dataSource` is still available to you if you need access to it; its bean name is `dataSourceUnproxied`.
You can access this bean like any other Spring bean, i.e. using dependency injection:

```
class MyService {
    def dataSourceUnproxied
    …
}
```

or by pulling it from the `ApplicationContext`:

```
def dataSourceUnproxied = ctx.dataSourceUnproxied
```

3.4 Externalized Configuration

The default configuration file `Config.groovy` in `grails-app/conf` is fine in the majority of cases, but there may be circumstances where you want to maintain the configuration in a file *outside* the main application structure.
For example if you are deploying to a WAR some administrators prefer the configuration of the application to be externalized to avoid having to re-package the WAR due to a change of configuration.
In order to support deployment scenarios such as these the configuration can be externalized. To do so you need to

point Grails at the locations of the configuration files Grails should be using by adding a
`grails.config.locations` setting in `Config.groovy`:

```
grails.config.locations = [ "classpath:${appName}-config.properties",
                            "classpath:${appName}-config.groovy",
                            "file:${userHome}/.grails/${appName}-config.properties",
                            "file:${userHome}/.grails/${appName}-config.groovy"]
```

In the above example we're loading configuration files (both Java properties files and ConfigSlurper configurations) from different places on the classpath and files located in `USER_HOME`.
Ultimately all configuration files get merged into the `config` property of the GrailsApplication object and are hence obtainable from there.

> ⚠️ Grails also supports the concept of property place holders and property override configurers as defined in Spring For more information on these see the section on Grails and Spring

3.5 Versioning

Versioning Basics
Grails has built in support for application versioning. When you first create an application with the create-app command the version of the application is set to `0.1`. The version is stored in the application meta data file called `application.properties` in the root of the project.
To change the version of your application you can run the set-version command:

```
grails set-version 0.2
```

The version is used in various commands including the war command which will append the application version to the end of the created WAR file.

Detecting Versions at Runtime
You can detect the application version using Grails' support for application metadata using the GrailsApplication class. For example within controllers there is an implicit grailsApplication variable that can be used:

```
def version = grailsApplication.metadata['app.version']
```

If it is the version of Grails you need you can use:

```
def grailsVersion = grailsApplication.metadata['app.grails.version']
```

or the `GrailsUtil` class:

```
import grails.util.*
def grailsVersion = GrailsUtil.grailsVersion
```

3.6 Project Documentation

Since Grails 1.2, the documentation engine that powers the creation of this documentation is available to your Grails projects.
The documentation engine uses a variation on the Textile syntax to automatically create project documentation with smart linking, formatting etc.

Creating project documentation

To use the engine you need to follow a few conventions. Firstly you need to create a `src/docs/guide` directory and then have numbered text files using the `gdoc` format. For example:

```
+ src/docs/guide/1. Introduction.gdoc
+ src/docs/guide/2. Getting Started.gdoc
```

The title of each chapter is taken from the file name. The order is dictated by the numerical value at the beginning of the file name.

Creating reference items

Reference items appear in the left menu on the documentation and are useful for quick reference documentation. Each reference item belongs to a category and a category is a directory located in the `src/docs/ref` directory. For example say you defined a new method called `renderPDF`, that belongs to a category called `Controllers` this can be done by creating a gdoc text file at the following location:

```
+ src/ref/Controllers/renderPDF.gdoc
```

Configuring Output Properties

There are various properties you can set within your `grails-app/conf/Config.groovy` file that customize the output of the documentation such as:

- **grails.doc.authors** - The authors of the documentation
- **grails.doc.license** - The license of the software
- **grails.doc.copyright** - The copyright message to display
- **grails.doc.footer** - The footer to use

Other properties such as the name of the documentation and the version are pulled from your project itself.

Generating Documentation

Once you have created some documentation (refer to the syntax guide in the next chapter) you can generate an HTML version of the documentation using the command:

```
grails docs
```

This command will output an `docs/manual/index.html` which can be opened to view your documentation.

Documentation Syntax

As mentioned the syntax is largely similar to Textile or Confluence style wiki markup. The following sections walk you through the syntax basics.

Basic Formatting

Monospace: `monospace`

```
@monospace@
```

Italic: *italic*

```
_italic_
```

Bold: **bold**

```
*bold*
```

Image: **GRAILS**

```
!http://grails.org/images/new/grailslogo_topNav.png!
```

Linking

There are several ways to create links with the documentation generator. A basic external link can either be defined using confluence or textile style markup:

```
[SpringSource|http://www.springsource.com/] or "SpringSource":http://www.springsource.com/
```

For links to other sections inside the user guide you can use the `guide:` prefix:

```
[Intro|guide:1. Introduction]
```

The documentation engine will warn you if any links to sections in your guide break. Sometimes though it is preferable not to hard code the actual names of guide sections since you may move them around. To get around this you can create an alias inside `grails-app/conf/Config.groovy`:

```
grails.doc.alias.intro="1. Introduction"
```

And then the link becomes:

```
[Intro|guide:intro]
```

This is useful since if you linked the to "1. Introduction" chapter many times you would have to change all of those links.
To link to reference items you can use a special syntax:

```
[controllers|renderPDF]
```

In this case the category of the reference item is on the left hand side of the | and the name of the reference item on the right.
Finally, to link to external APIs you can use the `api:` prefix. For example:

```
[String|api:java.lang.String]
```

The documentation engine will automatically create the appropriate javadoc link in this case. If you want to add additional APIs to the engine you can configure them in `grails-app/conf/Config.groovy`. For example:

```
grails.doc.api.org.hibernate="http://docs.jboss.org/hibernate/stable/core/api"
```

The above example configures classes within the `org.hibernate` package to link to the Hibernate website's API docs.

Lists and Headings

Headings can be created by specifying the letter 'h' followed by a number and then a dot:

```
h3.<space>Heading3
h4.<space>Heading4
```

Unordered lists are defined with the use of the * character:

```
* item 1
** subitem 1
** subitem 2
* item 2
```

Numbered lists can be defined with the # character:

```
# item 1
```

Tables can be created using the `table` macro:

Name	Number
Albert	46
Wilma	1348
James	12

```
{table}
 *Name* |  *Number*
 Albert | 46
 Wilma  | 1348
 James  | 12
{table}
```

Code and Notes

You can define code blocks with the `code` macro:

```
class Book {
    String title
}
```

```
{code}
class Book {
    String title
}
{code}
```

The example above provides syntax highlighting for Java and Groovy code, but you can also highlight XML markup:

```
<hello>world</hello>
```

```
{code:xml}
<hello>world</hello>
{code}
```

There are also a couple of macros for displaying notes and warnings:
Note:

⚠   This is a note!

```
{note}
This is a note!
{note}
```

Warning:

⛔   This is a warning!

```
{warning}
This is a warning!
{warning}
```

3.7 Dependency Resolution

In order to control how JAR dependencies are resolved Grails features (since version 1.2) a dependency resolution DSL that allows you to control how dependencies for applications and plugins are resolved.
Inside the `grails-app/conf/BuildConfig.groovy` file you can specify a
`grails.project.dependency.resolution` property that configures how dependencies are resolved:

```
grails.project.dependency.resolution = {
    // config here
}
```

The default configuration looks like the following:

```
grails.project.dependency.resolution = {
    inherits "global" // inherit Grails' default dependencies
    log "warn" // log level of Ivy resolver, either 'error',
               // 'warn', 'info', 'debug' or 'verbose'
    repositories {
        grailsHome()
        // uncomment the below to enable remote dependency resolution
        // from public Maven repositories
        //mavenCentral()
        //mavenRepo "http://snapshots.repository.codehaus.org"
        //mavenRepo "http://repository.codehaus.org"
        //mavenRepo "http://download.java.net/maven/2/"
        //mavenRepo "http://repository.jboss.com/maven2/
    }
    dependencies {
        // specify dependencies here under either 'build', 'compile',
        // 'runtime', 'test' or 'provided' scopes, e.g.
        // runtime 'com.mysql:mysql-connector-java:5.1.5'
    }
}
```

The details of the above will be explained in the next few sections.

3.7.1 Configurations and Dependencies

Grails features 5 dependency resolution configurations (or 'scopes') which you can take advantage of:

- `build`: Dependencies for the build system only
- `compile`: Dependencies for the compile step
- `runtime`: Dependencies needed at runtime but not for compilation (see above)
- `test`: Dependencies needed for testing but not at runtime (see above)
- `provided`: Dependencies needed at development time, but not during WAR deployment

Within the `dependencies` block you can specify a dependency that falls into one of these configurations by calling the equivalent method. For example if your application requires the MySQL driver to function at `runtime` you can specify as such:

```
runtime 'com.mysql:mysql-connector-java:5.1.5'
```

The above uses the string syntax which is `group:name:version`. You can also use a map-based syntax:

```
runtime group:'com.mysql', name:'mysql-connector-java', version:'5.1.5'
```

Multiple dependencies can be specified by passing multiple arguments:

```
runtime 'com.mysql:mysql-connector-java:5.1.5',
        'net.sf.ehcache:ehcache:1.6.1'
// Or
runtime(
    [group:'com.mysql', name:'mysql-connector-java', version:'5.1.5'],
    [group:'net.sf.ehcache', name:'ehcache', version:'1.6.1']
)
```

3.7.2 Dependency Repositories

Remote Repositories
Grails, when installed, does not use any remote public repositories. There is a default `grailsHome()` repository that will locate the JAR files Grails needs from your Grails installation. If you want to take advantage of a public repository you need to specify as such inside the `repositories` block:

```
repositories {
    mavenCentral()
}
```

In this case the default public Maven repository is specified. To use the SpringSource Enterprise Bundle Repository you can use the `ebr()` method:

```
repositories {
    ebr()
}
```

You can also specify a specific Maven repository to use by URL:

```
repositories {
    mavenRepo "http://repository.codehaus.org"
}
```

Local Resolvers

If you do not wish to use a public Maven repository you can specify a flat file repository:

```
repositories {
    flatDir name:'myRepo', dirs:'/path/to/repo'
}
```

Custom Resolvers

If all else fails since Grails builds on Apache Ivy you can specify an Ivy resolver:

```
repositories {
    resolver new URLResolver(...)
}
```

Authentication

If your repository requires some form of authentication you can specify as such using a `credentials` block:

```
credentials {
    realm = ".."
    host = "localhost"
    username = "myuser"
    password = "mypass"
}
```

The above can also be placed in your `USER_HOME/.grails/settings.groovy` file using the `grails.project.ivy.authentication` setting:

```
grails.project.ivy.authentication = {
    credentials {
        realm = ".."
        host = "localhost"
        username = "myuser"
        password = "mypass"
    }
}
```

3.7.3 Debugging Resolution

If you are having trouble getting a dependency to resolve you can enable more verbose debugging from the underlying engine using the `log` method:

```
// log level of Ivy resolver, either 'error', 'warn', 'info', 'debug' or 'verbose'
log "warn"
```

3.7.4 Inherited Dependencies

By default every Grails application inherits a bunch of framework dependencies. This is done through the line:

```
inherits "global"
```

Inside the `BuildConfig.groovy` file. If you wish exclude certain inherited dependencies then you can do so using the `excludes` method:

```
inherits("global") {
    excludes "oscache", "ehcache"
}
```

3.7.5 Dependency Reports

As mentioned in the previous section a Grails application consists of dependencies inherited from the framework, the plugins installed and the application dependencies itself.
To obtain a report of an application's dependencies you can run the dependency-report command:

```
grails dependency-report
```

This will output a report to the `target/dependency-report` directory by default. You can specify which configuration (scope) you want a report for by passing an argument containing the configuration name:

```
grails dependency-report runtime
```

3.7.6 Plugin JAR Dependencies

Specifying Plugin JAR dependencies
The way in which you specify dependencies for a plugin is identical to how you specify dependencies in an application. When a plugin is installed into an application the application automatically inherits the dependencies of the plugin.
If you want to define a dependency that is resolved for use with the plugin but not *exported* to the application then you can set the `exported` property of the dependency:

```
compile( 'org.hibernate:hibernate-core:3.3.1.GA') {
    exported = false
}
```

In this can the `hibernate-core` dependency will be available only to the plugin and not resolved as an application dependency.

Overriding Plugin JAR Dependencies in Your Application

If a plugin is using a JAR which conflicts with another plugin, or an application dependency then you can override how a plugin resolves its dependencies inside an application using exclusions. For example:

```
plugins {
    runtime( "org.grails.plugins:hibernate:1.3.0" ) {
        excludes "javassist"
    }
}
dependencies {
    runtime "javassist:javassist:3.4.GA"
}
```

In this case the application explicitly declares a dependency on the "hibernate" plugin and specifies an exclusion using the `excludes` method, effectively excluding the javassist library as a dependency.

3.7.7 Maven Integration

When using the Grails Maven plugin, Grails' dependency resolution mechanics are disabled as it is assumed that you will manage dependencies via Maven's `pom.xml` file.

However, if you would like to continue using Grails regular commands like run-app, test-app and so on then you can tell Grails' command line to load dependencies from the Maven `pom.xml` file instead.

To do so simply add the following line to your `BuildConfig.groovy`:

```
grails.project.dependency.resolution = {
    pom true
    ..
}
```

The line `pom true` tells Grails to parse Maven's `pom.xml` and load dependencies from there.

3.7.8 Deploying to a Maven Repository

You can deploy a Grails project or plugin to a Maven repository using the maven-publisher plugin.

The plugin provides the ability to publish Grails projects and plugins to local and remote Maven repositories. There are two key additional targets added by the plugin:

○ **maven-install** - Installs a Grails project or plugin into your local Maven cache
○ **maven-deploy** - Deploys a Grails project or plugin to a remote Maven repository

By default this plugin will automatically generate a valid `pom.xml` for you unless a `pom.xml` is already present in the root of the project, in which case this `pom.xml` file will be used.

maven-install

The `maven-install` command will install the Grails project or plugin artifact into your local Maven cache:

```
grails maven-install
```

In the case of plugins, the plugin zip file will be installed, whilst for application the application WAR file will be installed.

maven-deploy

The `maven-deploy` command will deploy a Grails project or plugin into a remote Maven repository:

```
grails maven-deploy
```

It is assumed that you have specified the necessary `<distributionManagement>` configuration within a `pom.xml` or that you specify the `id` of the remote repository to deploy to:

```
grails maven-deploy --repository=myRepo
```

The `repository` argument specifies the 'id' for the repository. You need to configure the details of the repository specified by this 'id' within your `grails-app/conf/BuildConfig.groovy` file or in your `USER_HOMER/.grails/settings.groovy` file:

```
grails.project.dependency.distribution = {
    localRepository = "/path/to/my/local"
    remoteRepository(id:"myRepo", url:"http://myserver/path/to/repo")
}
```

The syntax for configuring remote repositories matches the syntax from the [remoteRepository](#) element in the Ant Maven tasks. For example the following XML:

```
<remoteRepository id="myRepo" url="scp://localhost/www/repository">
    <authentication username="..." privateKey="${user.home}/.ssh/id_dsa"/>
</remoteRepository>
```

Can be expressed as:

```
remoteRepository(id:"myRepo", url:"scp://localhost/www/repository") {
    authentication username:"...", privateKey:"${userHome}/.ssh/id_dsa"
}
```

By default the plugin will try to detect the protocol to use from the URL of the repository (ie "http" from "http://.." etc.), however if you need to explicitly specify a different protocol you can do:

```
grails maven-deploy --repository=myRepo --protocol=webdav
```

The available protocols are:

- http
- scp
- scpexe
- ftp
- webdav

Groups, Artifacts and Versions

Maven defines the notion of a 'groupId', 'artifactId' and a 'version'. This plugin pulls this information from the Grails project conventions or plugin descriptor.

Projects

For applications this plugin will use the Grails application name and version provided by Grails when generating the `pom.xml` file. To change the version you can run the `set-version` command:

```
grails set-version 0.2
```

The Maven `groupId` will be the same as the project name, unless you specify a different one in Config.groovy:

```
grails.project.groupId="com.mycompany"
```

Plugins

With a Grails plugin the `groupId` and `version` are taken from the following properties in the
*GrailsPlugin.groovy descriptor:

```
String groupId = 'myOrg'
String version = '0.1'
```

The 'artifactId' is taken from the plugin name. For example if you have a plugin called `FeedsGrailsPlugin` the
`artifactId` will be "feeds". If your plugin does not specify a `groupId` then this defaults to "org.grails.plugins".

3.7.9 Plugin Dependencies

As of Grails 1.3 you can declaratively specify dependencies on plugins rather than using the [install-plugin](#) command:

```
plugins {
    runtime ':hibernate:1.2.1'
}
```

If you don't specify a group id the default plugin group id of `org.grails.plugins` is used. You can specify to
use the latest version of a particular plugin by using "latest.integration" as the version number:

```
plugins {
    runtime ':hibernate:latest.integration'
}
```

Integration vs. Release
The "latest.integration" version label will also include resolving snapshot versions. If you don't want to include
snapshot versions then you can use the "latest.release" label:

```
plugins {
    runtime ':hibernate:latest.release'
}
```

> ⚠️ The "latest.release" label only works with Maven compatible repositories. If you have a
> regular SVN-based Grails repository then you should use "latest.integration".

And of course if you are using a Maven repository with an alternative group id you can specify a group id:

```
plugins {
    runtime 'mycompany:hibernate:latest.integration'
}
```

Plugin Exclusions
You can control how plugins transitively resolves both plugin and JAR dependencies using exclusions. For example:

```
plugins {
    runtime( ':weceem:0.8' ) {
        excludes "searchable"
    }
}
```

Here we have defined a dependency on the "weceem" plugin which transitively depends on the "searchable" plugin.
By using the `excludes` method you can tell Grails *not* to transitively install the searchable plugin. You can
combine this technique to specify an alternative version of a plugin:

```
plugins {
    runtime( ':weceem:0.8' ) {
        excludes "searchable" // excludes most recent version
    }
    runtime ':searchable:0.5.4' // specifies a fixed searchable version
}
```

You can also completely disable transitive plugin installs, in which case no transitive dependencies will be resolved:

```
plugins {
    runtime( ':weceem:0.8' ) {
        transitive = false
    }
    runtime ':searchable:0.5.4' // specifies a fixed searchable version
}
```

4. The Command Line

Grails' command line system is built on [Gant](#) - a simple Groovy wrapper around [Apache Ant](#).
However, Grails takes it a bit further through the use of convention and the `grails` command. When you type:

```
grails [command name]
```

Grails does a search in the following directories for Gant scripts to execute:

○ `USER_HOME/.grails/scripts`
○ `PROJECT_HOME/scripts`
○ `PROJECT_HOME/plugins/*/scripts`
○ `GRAILS_HOME/scripts`

Grails will also convert command names that are in lower case form such as run-app into camel case. So typing

```
grails run-app
```

Results in a search for the following files:

○ `USER_HOME/.grails/scripts/RunApp.groovy`
○ `PROJECT_HOME/scripts/RunApp.groovy`
○ `PLUGINS_HOME/*/scripts/RunApp.groovy`
○ `GLOBAL_PLUGINS_HOME/*/scripts/RunApp.groovy`
○ `GRAILS_HOME/scripts/RunApp.groovy`

If multiple matches are found Grails will give you a choice of which one to execute. When Grails executes a Gant
script, it invokes the "default" target defined in that script. If there is no default, Grails will quit with an error.
To get a list and some help about the available commands type:

```
grails help
```

Which outputs usage instructions and the list of commands Grails is aware of:

```
Usage (optionals marked with *):
grails [environment]* [target] [arguments]*
Examples:
grails dev run-app
grails create-app books
Available Targets (type grails help 'target-name' for more info):
grails bootstrap
grails bug-report
grails clean
grails compile
...
```

> ⚠ Refer to the Command Line reference in left menu of the reference guide for more
> information about individual commands

4.1 Creating Gant Scripts

You can create your own Gant scripts by running the [create-script](#) command from the root of your project. For
example the following command:

```
grails create-script compile-sources
```

Will create a script called `scripts/CompileSources.groovy`. A Gant script itself is similar to a regular Groovy script except that it supports the concept of "targets" and dependencies between them:

```
target(default:"The default target is the one that gets executed by Grails") {
    depends(clean, compile)
}
target(clean:"Clean out things") {
    ant.delete(dir:"output")
}
target(compile:"Compile some sources") {
    ant.mkdir(dir:"mkdir")
    ant.javac(srcdir:"src/java", destdir:"output")
}
```

As demonstrated in the script above, there is an implicit `ant` variable that allows access to the [Apache Ant API](#).

> ⚠️ In previous versions of Grails (1.0.3 and below), the variable was `Ant`, i.e. with a capital first letter.

You can also "depend" on other targets using the `depends` method demonstrated in the `default` target above.

The default target

In the example above, we specified a target with the explicit name "default". This is one way of defining the default target for a script. An alternative approach is to use the `setDefaultTarget()` method:

```
target("clean-compile": "Performs a clean compilation on the app's source files.") {
    depends(clean, compile)
}
target(clean:"Clean out things") {
    ant.delete(dir:"output")
}
target(compile:"Compile some sources") {
    ant.mkdir(dir:"mkdir")
    ant.javac(srcdir:"src/java", destdir:"output")
}
setDefaultTarget("clean-compile")
```

This allows you to call the default target directly from other scripts if you wish. Also, although we have put the call to `setDefaultTarget()` at the end of the script in this example, it can go anywhere as long as it comes *after* the target it refers to ("clean-compile" in this case).

Which approach is better? To be honest, you can use whichever you prefer - there don't seem to be any major advantages in either case. One thing we would say is that if you want to allow other scripts to call your "default" target, you should move it into a shared script that doesn't have a default target at all. We'll talk some more about this in the next section.

4.2 Re-using Grails scripts

Grails ships with a lot of command line functionality out of the box that you may find useful in your own scripts (See the command line reference in the reference guide for info on all the commands). Of particular use are the [compile](#), [package](#) and [bootstrap](#) scripts.

The [bootstrap](#) script for example allows you to bootstrap a Spring [ApplicationContext](#) instance to get access to the data source and so on (the integration tests use this):

```
includeTargets << grailsScript("_GrailsBootstrap")
target ('default': "Load the Grails interactive shell") {
    depends( configureProxy, packageApp, classpath, loadApp, configureApp )
    Connection c
    try {
        // do something with connection
        c = appCtx.getBean('dataSource').getConnection()
    }
    finally {
        c?.close()
    }
}
```

Pulling in targets from other scripts

Gant allows you to pull in all targets (except "default") from another Gant script. You can then depend upon or invoke those targets as if they had been defined in the current script. The mechanism for doing this is the `includeTargets` property. Simply "append" a file or class to it using the left-shift operator:

```
includeTargets << new File("/path/to/my/script.groovy")
includeTargets << gant.tools.Ivy
```

Don't worry too much about the syntax using a class, it's quite specialised. If you're interested, look into the Gant documentation.

Core Grails targets

As you saw in the example at the beginning of this section, you use neither the File- nor the class-based syntax for `includeTargets` when including core Grails targets. Instead, you should use the special `grailsScript()` method that is provided by the Grails command launcher (note that this is not available in normal Gant scripts, just Grails ones).

The syntax for the `grailsScript()` method is pretty straightforward: simply pass it the name of the Grails script you want to include, without any path information. Here is a list of Grails scripts that you may want to re-use:

Script	Description
_GrailsSettings	You really should include this! Fortunately, it is included automatically by all other Grails scripts bar one (_GrailsProxy), so you usually don't have to include it explicitly.
_GrailsEvents	If you want to fire events, you need to include this. Adds an `event(String eventName, List args)` method. Again, included by almost all other Grails scripts.
_GrailsClasspath	Sets up compilation, test, and runtime classpaths. If you want to use or play with them, include this script. Again, included by almost all other Grails scripts.
_GrailsProxy	If you want to access the internet, include this script so that you don't run into problems with proxies.
_GrailsArgParsing	Provides a `parseArguments` target that does what it says on the tin: parses the arguments provided by the user when they run your script. Adds them to the `argsMap` property.
_GrailsTest	Contains all the shared test code. Useful if you want to add any extra tests.
_GrailsRun	Provides all you need to run the application in the configured servlet container, either normally (`runApp`/`runAppHttps`) or from a WAR file (`runWar`/`runWarHttps`).

There are many more scripts provided by Grails, so it is worth digging into the scripts themselves to find out what kind of targets are available. Anything that starts with an "_" is designed for re-use.

> ⚠️ In pre-1.1 versions of Grails, the "_Grails..." scripts were not available. Instead, you typically include the corresponding command script, for example "Init.groovy" or "Bootstrap.groovy". Also, in pre-1.0.4 versions of Grails you cannot use the `grailsScript()` method. Instead, you must use `includeTargets << new File(...)` and specify the script's location in full (i.e. $GRAILS_HOME/scripts).

Script architecture

You maybe wondering what those underscores are doing in the names of the Grails scripts. That is Grails' way of determining that a script is _internal_, or in other words that it has not corresponding "command". So you can't run "grails _grails-settings" for example. That is also why they don't have a default target.

Internal scripts are all about code sharing and re-use. In fact, we recommend you take a similar approach in your own scripts: put all your targets into an internal script that can be easily shared, and provide simple command scripts that parse any command line arguments and delegate to the targets in the internal script. Say you have a script that runs some functional tests - you can split it like this:

```
./scripts/FunctionalTests.groovy:
includeTargets << new File("${basedir}/scripts/_FunctionalTests.groovy")
target(default: "Runs the functional tests for this project.") {
    depends(runFunctionalTests)
}
./scripts/_FunctionalTests.groovy:
includeTargets << grailsScript("_GrailsTest")
target(runFunctionalTests: "Run functional tests.") {
    depends(...)
    …
}
```

Here are a few general guidelines on writing scripts:

- Split scripts into a "command" script and an internal one.
- Put the bulk of the implementation in the internal script.
- Put argument parsing into the "command" script.
- To pass arguments to a target, create some script variables and initialise them before calling the target.
- Avoid name clashes by using closures assigned to script variables instead of targets. You can then pass arguments direct to the closures.

4.3 Hooking into Events

Grails provides the ability to hook into scripting events. These are events triggered during execution of Grails target and plugin scripts.

The mechanism is deliberately simple and loosely specified. The list of possible events is not fixed in any way, so it is possible to hook into events triggered by plugin scripts, for which there is no equivalent event in the core target scripts.

Defining event handlers

Event handlers are defined in scripts called `_Events.groovy`. Grails searches for these scripts in the following locations:

- `USER_HOME/.grails/scripts` - user-specific event handlers
- `PROJECT_HOME/scripts` - applicaton-specific event handlers
- `PLUGINS_HOME/*/scripts` - plugin-specific event handlers
- `GLOBAL_PLUGINS_HOME/*/scripts` - event handlers provided by global plugins

Whenever an event is fired, *all* the registered handlers for that event are executed. Note that the registration of handlers is performed automatically by Grails, so you just need to declare them in the relevant `_Events.groovy` file.

> ⚠️ In versions of Grails prior to 1.0.4, the script was called `Events.groovy`, that is without the leading underscore.

Event handlers are blocks defined in `_Events.groovy`, with a name beginning with "event". The following example can be put in your /scripts directory to demonstrate the feature:

```
eventCreatedArtefact = { type, name ->
   println "Created $type $name"
}
eventStatusUpdate = { msg ->
   println msg
}
eventStatusFinal = { msg ->
   println msg
}
```

You can see here the three handlers `eventCreatedArtefact`, `eventStatusUpdate`, `eventStatusFinal`. Grails provides some standard events, which are documented in the command line reference guide. For example the [compile](#) command fires the following events:

- `CompileStart` - Called when compilation starts, passing the kind of compile - source or tests
- `CompileEnd` - Called when compilation is finished, passing the kind of compile - source or tests

Triggering events
To trigger an event simply include the Init.groovy script and call the event() closure:

```
includeTargets << grailsScript("_GrailsEvents")
event("StatusFinal", ["Super duper plugin action complete!"])
```

Common Events
Below is a table of some of the common events that can be leveraged:

Event	Parameters	Description
StatusUpdate	message	Passed a string indicating current script status/progress
StatusError	message	Passed a string indicating an error message from the current script
StatusFinal	message	Passed a string indicating the final script status message, i.e. when completing a target, even if the target does not exit the scripting environment
CreatedArtefact	artefactType,artefactName	Called when a create-xxxx script has completed and created an artefact
CreatedFile	fileName	Called whenever a project source filed is created, not including files constantly managed by Grails
Exiting	returnCode	Called when the scripting environment is about to exit cleanly
PluginInstalled	pluginName	Called after a plugin has been installed
CompileStart	kind	Called when compilation starts, passing the kind of compile - source or tests
CompileEnd	kind	Called when compilation is finished, passing the kind of compile - source or tests
DocStart	kind	Called when documentation generation is about to start - javadoc or groovydoc
DocEnd	kind	Called when documentation generation has ended - javadoc or groovydoc
SetClasspath	rootLoader	Called during classpath initialization so plugins can augment the classpath with rootLoader.addURL(...). Note that this augments the classpath **after** event scripts are loaded so you cannot use this to load a class that your event script needs to import, although you can do this if you load the class by name.
PackagingEnd	none	Called at the end of packaging (which is called prior to the Tomcat server being started and after web.xml is generated)

4.4 Customising the build

Grails is most definitely an opinionated framework and it prefers convention to configuration, but this doesn't mean you *can't* configure it. In this section, we look at how you can influence and modify the standard Grails build.

The defaults

In order to customise a build, you first need to know *what* you can customise. The core of the Grails build configuration is the `grails.util.BuildSettings` class, which contains quite a bit of useful information. It controls where classes are compiled to, what dependencies the application has, and other such settings.
Here is a selection of the configuration options and their default values:

Property	Config option	Default value
grailsWorkDir	grails.work.dir	$USER_HOME/.grails/<grailsVersion>
projectWorkDir	grails.project.work.dir	<grailsWorkDir>/projects/<baseDirName>
classesDir	grails.project.class.dir	<projectWorkDir>/classes
testClassesDir	grails.project.test.class.dir	<projectWorkDir>/test-classes
testReportsDir	grails.project.test.reports.dir	<projectWorkDir>/test/reports
resourcesDir	grails.project.resource.dir	<projectWorkDir>/resources
projectPluginsDir	grails.project.plugins.dir	<projectWorkDir>/plugins
globalPluginsDir	grails.global.plugins.dir	<grailsWorkDir>/global-plugins
verboseCompile	grails.project.compile.verbose	false

The `BuildSettings` class has some other properties too, but they should be treated as read-only:

Property	Description
baseDir	The location of the project.
userHome	The user's home directory.
grailsHome	The location of the Grails installation in use (may be null).
grailsVersion	The version of Grails being used by the project.
grailsEnv	The current Grails environment.
compileDependencies	A list of compile-time project dependencies as `File` instances.
testDependencies	A list of test-time project dependencies as `File` instances.
runtimeDependencies	A list of runtime-time project dependencies as `File` instances.

Of course, these properties aren't much good if you can't get hold of them. Fortunately that's easy to do: an instance of `BuildSettings` is available to your scripts via the `grailsSettings` script variable. You can also access it from your code by using the `grails.util.BuildSettingsHolder` class, but this isn't recommended.

Overriding the defaults

All of the properties in the first table can be overridden by a system property or a configuration option - simply use the "config option" name. For example, to change the project working directory, you could either run this command:

```
grails -Dgrails.project.work.dir=work compile
```

or add this option to your `grails-app/conf/BuildConfig.groovy` file:

```
grails.project.work.dir = "work"
```

Note that the default values take account of the property values they depend on, so setting the project working directory like this would also relocate the compiled classes, test classes, resources, and plugins.
What happens if you use both a system property and a configuration option? Then the system property wins because it takes precedence over the `BuildConfig.groovy` file, which in turn takes precedence over the default values. The `BuildConfig.groovy` file is a sibling of `grails-app/conf/Config.groovy` - the former contains options that only affect the build, whereas the latter contains those that affect the application at runtime. It's not limited to the options in the first table either: you will find build configuration options dotted around the documentation, such as ones for specifying the port that the embedded servlet container runs on or for determining what files get packaged in the WAR file.

Available build settings

Name	Description
grails.server.port.http	Port to run the embedded servlet container on ("run-app" and "run-war"). Integer.
grails.server.port.https	Port to run the embedded servlet container on for HTTPS ("run-app --https" and "run-war --https"). Integer.
grails.config.base.webXml	Path to a custom web.xml file to use for the application (alternative to using the web.xml template).
grails.compiler.dependencies	Legacy approach to adding extra dependencies to the compiler classpath. Set it to a closure containing "fileset()" entries.
grails.testing.patterns	A list of Ant path patterns that allow you to control which files are included in the tests. The patterns should not include the test case suffix, which is set by the next property.
grails.testing.nameSuffix	By default, tests are assumed to have a suffix of "Tests". You can change it to anything you like but setting this option. For example, another common suffix is "Test".
grails.war.destFile	A string containing the file path of the generated WAR file, along with its full name (include extension). For example, "target/my-app.war".
grails.war.dependencies	A closure containing "fileset()" entries that allows you complete control over what goes in the WAR's "WEB-INF/lib" directory.
grails.war.copyToWebApp	A closure containing "fileset()" entries that allows you complete control over what goes in the root of the WAR. It overrides the default behaviour of including everything under "web-app".
grails.war.resources	A closure that takes the location of the staging directory as its first argument. You can use any Ant tasks to do anything you like. It is typically used to remove files from the staging directory before that directory is jar'd up into a WAR.
grails.project.web.xml	The location to generate Grails' web.xml to

4.5 Ant and Maven

If all the other projects in your team or company are built using a standard build tool such as Ant or Maven, you become the black sheep of the family when you use the Grails command line to build your application. Fortunately, you can easily integrate the Grails build system into the main build tools in use today (well, the ones in use in Java projects at least).

Ant Integration
When you create a Grails application via the create-app command, Grails automatically creates an Apache Ant `build.xml` file for you containing the following targets:

- `clean` - Cleans the Grails application
- `compile` - Compiles your application's source code
- `test` - Runs the unit tests

- `run` - Equivalent to "grails run-app"
- `war` - Creates a WAR file
- `deploy` - Empty by default, but can be used to implement automatic deployment

Each of these can be run by Ant, for example:

```
ant war
```

The build file is all geared up to use [Apache Ivy](#) for dependency management, which means that it will automatically download all the requisite Grails JAR files and other dependencies on demand. You don't even have to install Grails locally to use it! That makes it particularly useful for continuous integration systems such as [CruiseControl](#) or [Hudson](#)

It uses the Grails [Ant task](#) to hook into the existing Grails build system. The task allows you to run any Grails script that's available, not just the ones used by the generated build file. To use the task, you must first declare it:

```
<taskdef name="grailsTask"
         classname="grails.ant.GrailsTask"
         classpathref="grails.classpath"/>
```

This raises the question: what should be in "grails.classpath"? The task itself is in the "grails-bootstrap" JAR artifact, so that needs to be on the classpath at least. You should also include the "groovy-all" JAR. With the task defined, you just need to use it! The following table shows you what attributes are available:

Attribute	Description	Required
home	The location of the Grails installation directory to use for the build.	Yes, unless classpath is specified.
classpathref	Classpath to load Grails from. Must include the "grails-bootstrap" artifact and should include "grails-scripts".	Yes, unless `home` is set or you use a `classpath` element.
script	The name of the Grails script to run, e.g. "TestApp".	Yes.
args	The arguments to pass to the script, e.g. "-unit -xml".	No. Defaults to "".
environment	The Grails environment to run the script in.	No. Defaults to the script default.
includeRuntimeClasspath	Advanced setting: adds the application's runtime classpath to the build classpath if true.	No. Defaults to true.

The task also supports the following nested elements, all of which are standard Ant path structures:

- `classpath` - The build classpath (used to load Gant and the Grails scripts).
- `compileClasspath` - Classpath used to compile the application's classes.
- `runtimeClasspath` - Classpath used to run the application and package the WAR. Typically includes everything in @compileClasspath.
- `testClasspath` - Classpath used to compile and run the tests. Typically includes everything in `runtimeClasspath`.

How you populate these paths is up to you. If you are using the `home` attribute and put your own dependencies in the `lib` directory, then you don't even need to use any of them. For an example of their use, take a look at the generated Ant build file for new apps.

Maven Integration

From 1.1 onwards, Grails provides integration with [Maven 2](#) via a Maven plugin. The current Maven plugin is based on, but effectively supercedes, the version created by [Octo](#), who did a great job.

Preparation

In order to use the new plugin, all you need is Maven 2 installed and set up. This is because **you no longer need to install Grails separately to use it with Maven!**

> ⚠ The Maven 2 integration for Grails has been designed and tested for Maven 2.0.9 and above. It will not work with earlier versions.

To make life easier for you, we do recommend that you add a plugin group for Grails to your Maven settings file (`$USER_HOME/.m2/settings.xml`):

```
<settings>
  …
  <pluginGroups>
    <pluginGroup>org.grails</pluginGroup>
  </pluginGroups>
</settings>
```

In addition, if you have the Octo Maven Tools for Grails set up then you'll need to remove the `com.octo.mtg` plugin group.

Creating a Grails Maven Project
To create a Mavenized Grails project simple run the following command:

```
mvn archetype:generate -DarchetypeGroupId=org.grails \
    -DarchetypeArtifactId=grails-maven-archetype \
    -DarchetypeVersion=1.0 \
    -DarchetypeRepository=http://snapshots.repository.codehaus.org \
    -DgroupId=example -DartifactId=my-app
```

Choose whichever group ID and artifact ID you want for your application, but everything else must be as written. This will create a new Maven project with a POM and a couple of other files. What you won't see is anything that looks like a Grails application. So, the next step is to create the project structure that you're used to:

```
cd my-app
mvn initialize
```

Now you have a Grails application all ready to go. The plugin integrates into the standard build cycle, so you can use the standard Maven phases to build and package your app: `mvn clean`, `mvn compile`, `mvn test`, `mvn package`.
You can also take advantage of some of the Grails commands that have been wrapped as Maven goals:

- `grails:create-controller` - Calls the [create-controller](#) command
- `grails:create-domain-class` - Calls the [create-domain-class](#) command
- `grails:create-integration-test` - Calls the [create-integration-test](#) command
- `grails:create-pom` - Creates a new Maven POM for an existing Grails project
- `grails:create-script` - Calls the [create-script](#) command
- `grails:create-service` - Calls the [create-service](#) command
- `grails:create-taglib` - Calls the [create-tag-lib](#) command
- `grails:create-unit-test` - Calls the [create-unit-test](#) command
- `grails:exec` - Executes an arbitrary Grails command line script
- `grails:generate-all` - Calls the [generate-all](#) command
- `grails:generate-controller` - Calls the [generate-controller](#) command
- `grails:generate-views` - Calls the [generate-views](#) command
- `grails:install-plugin` - Calls the [install-plugin](#) command
- `grails:install-templates` - Calls the [install-templates](#) command
- `grails:list-plugins` - Calls the [list-plugins](#) command
- `grails:package` - Calls the [package](#) command
- `grails:run-app` - Calls the [run-app](#) command
- `grails:uninstall-plugin` - Calls the [uninstall-plugin](#) command

Mavenizing an existing project

Creating a new project is great way to start, but what if you already have one? You don't want to create a new project and then copy the contents of the old one over. The solution is to create a POM for the existing project using this Maven command:

```
mvn grails:create-pom -DgroupId=com.mycompany
```

When this command has finished, you can immediately start using the standard phases, such as `mvn package`. Note that you have to specify a group ID when creating the POM.

Adding Grails commands to phases

The standard POM created for you by Grails already attaches the appropriate core Grails commands to their corresponding build phases, so "compile" goes in the "compile" phase and "war" goes in the "package" phase. That doesn't help though when you want to attach a plugin's command to a particular phase. The classic example is functional tests. How do you make sure that your functional tests (using which ever plugin you have decided on) are run during the "integration-test" phase?

Fear not: all things are possible. In this case, you can associate the command to a phase using an extra "execution" block:

```xml
<plugin>
    <groupId>org.grails</groupId>
    <artifactId>grails-maven-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <extensions>true</extensions>
    <executions>
        <execution>
            <goals>
                …
            </goals>
        </execution>
        <!-- Add the "functional-tests" command to the "integration-test" phase -->
        <execution>
            <id>functional-tests</id>
            <phase>integration-test</phase>
            <goals>
                <goal>exec</goal>
            </goals>
            <configuration>
                <command>functional-tests</command>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This also demonstrates the `grails:exec` goal, which can be used to run any Grails command. Simply pass the name of the command as the `command` system property, and optionally specify the arguments via the `args` property:

```
mvn grails:exec -Dcommand=create-webtest -Dargs=Book
```

45

5. Object Relational Mapping (GORM)

Domain classes are core to any business application. They hold state about business processes and hopefully also implement behavior. They are linked together through relationships, either one-to-one or one-to-many.
GORM is Grails' object relational mapping (ORM) implementation. Under the hood it uses Hibernate 3 (an extremely popular and flexible open source ORM solution) but because of the dynamic nature of Groovy, the fact that it supports both static and dynamic typing, and the convention of Grails there is less configuration involved in creating Grails domain classes.
You can also write Grails domain classes in Java. See the section on Hibernate Integration for how to write Grails domain classes in Java but still use dynamic persistent methods. Below is a preview of GORM in action:

```
def book = Book.findByTitle("Groovy in Action")
book
   .addToAuthors(name:"Dierk Koenig")
   .addToAuthors(name:"Guillaume LaForge")
   .save()
```

5.1 Quick Start Guide

A domain class can be created with the create-domain-class command:

```
grails create-domain-class Person
```

This will create a class at the location `grails-app/domain/Person.groovy` such as the one below:

```
class Person {
}
```

> ⚠ If you have the `dbCreate` property set to "update", "create" or "create-drop" on your DataSource, Grails will automatically generated/modify the database tables for you.

You can customize the class by adding properties:

```
class Person {
    String name
    Integer age
    Date lastVisit
}
```

Once you have a domain class try and manipulate it via the shell or console by typing:

```
grails console
```

This loads an interactive GUI where you can type Groovy commands.

5.1.1 Basic CRUD

Try performing some basic CRUD (Create/Read/Update/Delete) operations.

Create
To create a domain class use the Groovy new operator, set its properties and call save:

```
def p = new Person(name:"Fred", age:40, lastVisit:new Date())
p.save()
```

The save method will persist your class to the database using the underlying Hibernate ORM layer.

Read
Grails transparently adds an implicit id property to your domain class which you can use for retrieval:

```
def p = Person.get(1)
assert 1 == p.id
```

This uses the get method that expects a database identifier to read the Person object back from the db. You can also load an object in a read-only state by using the read method:

```
def p = Person.read(1)
```

In this case the underlying Hibernate engine will not do any dirty checking and the object will not be persisted. Note that if you explicitly call the save method then the object is placed back into a read-write state.
In addition, you can also load an proxy for an instance by using the load method:

```
def p = Person.load(1)
```

This incurs no database access until a method other than getId() is called. Hibernate then initializes the proxied instance, or throws an exception if no record is found for the specified id.

Update
To update an instance, set some properties and then simply call save again:

```
def p = Person.get(1)
p.name = "Bob"
p.save()
```

Delete
To delete an instance use the delete method:

```
def p = Person.get(1)
p.delete()
```

5.2 Domain Modelling in GORM

When building Grails applications you have to consider the problem domain you are trying to solve. For example if you were building an Amazon bookstore you would be thinking about books, authors, customers and publishers to name a few.
These are modeled in GORM as Groovy classes so a Book class may have a title, a release date, an ISBN number and so on. The next few sections show how to model the domain in GORM.
To create a domain class you can run the create-domain-class target as follows:

```
grails create-domain-class Book
```

47

The result will be a class at `grails-app/domain/Book.groovy`:

```
class Book {
}
```

> ⚠️ If you wish to use packages you can move the Book.groovy class into a sub directory under the domain directory and add the appropriate `package` declaration as per Groovy (and Java's) packaging rules.

The above class will map automatically to a table in the database called `book` (the same name as the class). This behaviour is customizable through the [ORM Domain Specific Language](#)
Now that you have a domain class you can define its properties as Java types. For example:

```
class Book {
    String title
    Date releaseDate
    String ISBN
}
```

Each property is mapped to a column in the database, where the convention for column names is all lower case separated by underscores. For example `releaseDate` maps onto a column `release_date`. The SQL types are auto-detected from the Java types, but can be customized via [Constraints](#) or the [ORM DSL](#).

5.2.1 Association in GORM

Relationships define how domain classes interact with each other. Unless specified explicitly at both ends, a relationship exists only in the direction it is defined.

5.2.1.1 One-to-one

A one-to-one relationship is the simplest kind, and is defined trivially using a property of the type of another domain class. Consider this example:

Example A

```
class Face {
    Nose nose
}
class Nose {
}
```

In this case we have unidirectional many-to-one relationship from `Face` to `Nose`. To make it a true one-to-one you should make `nose` unique:

```
class Face {
    Nose nose
    static constraints = {
        nose unique: true
    }
}
class Nose {
}
```

To make this relationship bidirectional define the other side as follows:

Example B

```
class Face {
    Nose nose
}
class Nose {
    static belongsTo = [face:Face]
}
```

In this case we use the `belongsTo` setting to say that `Nose` "belongs to" Face. The result of this is that we can create a Face and save it and the database updates/inserts will be *cascaded* down to `Nose`:

```
new Face(nose:new Nose()).save()
```

The example above will save both face and nose. Note that the inverse *is not* true and will result in an error due to a transient `Face`:

```
new Nose(face:new Face()).save() // will cause an error
```

Another important implication of `belongsTo` is that if you delete a `Face` instance the `Nose` will be deleted too:

```
def f = Face.get(1)
f.delete() // both Face and Nose deleted
```

In the previous example the foreign key associated the `Face` with the `Nose` is stored in the parent as column called `nose_id`. If you want the foreign key to be stored in the child you need a `hasOne` association:

Example C

```
class Face {
    static hasOne = [nose:Nose]
}
class Nose {
    Face face
}
```

In this example you get a bidirectional one-to-one where the foreign key column is stored in the `nose` table inside a column called `face_id`.

5.2.1.2 One-to-many

A one-to-many relationship is when one class, example `Author`, has many instances of a another class, example `Book`. With Grails you define such a relationship with the `hasMany` setting:

```
class Author {
    static hasMany = [ books : Book ]
    String name
}
class Book {
    String title
}
```

In this case we have a unidirectional one-to-many. Grails will, by default, map this kind of relationship with a join

table.

> ⚠ The [ORM DSL](#) allows mapping unidirectional relationships using a foreign key association instead

Grails will automatically inject a property of type `java.util.Set` into the domain class based on the `hasMany` setting. This can be used to iterate over the collection:

```
def a = Author.get(1)
a.books.each {
    println it.title
}
```

> ⚠ The default fetch strategy used by Grails is "lazy", which means that the collection will be lazily initialized. This can lead to the [n+1 problem](#) if you are not careful.
> If you need "eager" fetching you can use the [ORM DSL](#) or specify eager fetching as part of a [query](#)

The default cascading behaviour is to cascade saves and updates, but not deletes unless a `belongsTo` is also specified:

```
class Author {
    static hasMany = [ books : Book ]
    String name
}
class Book {
    static belongsTo = [author:Author]
    String title
}
```

If you have two properties of the same type on the many side of a one-to-many you have to use `mappedBy` to specify which the collection is mapped:

```
class Airport {
    static hasMany = [flights:Flight]
    static mappedBy = [flights:"departureAirport"]
}
class Flight {
    Airport departureAirport
    Airport destinationAirport
}
```

This is also true if you have multiple collections that map to different properties on the many side:

```
class Airport {
    static hasMany = [outboundFlights:Flight, inboundFlights:Flight]
    static mappedBy = [outboundFlights:"departureAirport", inboundFlights:"destinationAirpor
}
class Flight {
    Airport departureAirport
    Airport destinationAirport
}
```

5.2.1.3 Many-to-many

Grails supports many-to-many relationships by defining a `hasMany` on both sides of the relationship and having a `belongsTo` on the owned side of the relationship:

```
class Book {
    static belongsTo = Author
    static hasMany = [authors:Author]
    String title
}
class Author {
    static hasMany = [books:Book]
    String name
}
```

Grails maps a many-to-many using a join table at the database level. The owning side of the relationship, in this case `Author`, takes responsibility for persisting the relationship and is the only side that can cascade saves across. For example this will work and cascade saves:

```
new Author(name:"Stephen King")
        .addToBooks(new Book(title:"The Stand"))
        .addToBooks(new Book(title:"The Shining"))
        .save()
```

However the below will only save the `Book` and not the authors!

```
new Book(name:"Groovy in Action")
        .addToAuthors(new Author(name:"Dierk Koenig"))
        .addToAuthors(new Author(name:"Guillaume Laforge"))
        .save()
```

This is the expected behaviour as, just like Hibernate, only one side of a many-to-many can take responsibility for managing the relationship.

> Grails' Scaffolding feature **does not** currently support many-to-many relationship and hence you must write the code to manage the relationship yourself

5.2.1.4 Basic Collection Types

As well as associations between different domain classes, GORM also supports mapping of basic collection types. For example, the following class creates a `nicknames` association that is a `Set` of `String` instances:

```
class Person {
    static hasMany = [nicknames:String]
}
```

GORM will map an association like the above using a join table. You can alter various aspects of how the join table is mapped using the `joinTable` argument:

```
class Person {
    static hasMany = [nicknames:String]
    static mapping = {
        hasMany joinTable: [name: 'bunch_o_nicknames',
                            key: 'person_id',
                            column: 'nickname',
                            type: "text"]
    }
}
```

The example above will map to a table that looks like the following:
bunch_o_nicknames Table

```
---------------------------------------------
| person_id          |      nickname         |
---------------------------------------------
|    1               |         Fred          |
---------------------------------------------
```

5.2.2 Composition in GORM

As well as association, Grails supports the notion of composition. In this case instead of mapping classes onto separate tables a class can be "embedded" within the current table. For example:

```
class Person {
    Address homeAddress
    Address workAddress
    static embedded = ['homeAddress', 'workAddress']
}
class Address {
    String number
    String code
}
```

The resulting mapping would looking like this:

Person Table

id	home_address_number	home_address_code	work_address_number	work_address_code
1	47	343432	67	43545

> ⚠️ If you define the `Address` class in a separate Groovy file in the `grails-app/domain` directory you will also get an `address` table. If you don't want this to happen use Groovy's ability to define multiple classes per file and include the `Address` class below the `Person` class in the `grails-app/domain/Person.groovy` file

5.2.3 Inheritance in GORM

GORM supports inheritance both from abstract base classes and concrete persistent GORM entities. For example:

```
class Content {
      String author
}
class BlogEntry extends Content {
    URL url
}
class Book extends Content {
    String ISBN
}
class PodCast extends Content {
    byte[] audioStream
}
```

In the above example we have a parent `Content` class and then various child classes with more specific behaviour.

Considerations
At the database level Grails by default uses table-per-hierarchy mapping with a discriminator column called `class` so the parent class (`Content`) and its sub classes (`BlogEntry`, `Book` etc.), share the **same** table.
Table-per-hierarchy mapping has a down side in that you **cannot** have non-nullable properties with inheritance mapping. An alternative is to use table-per-subclass which can be enabled via the ORM DSL
However, excessive use of inheritance and table-per-subclass can result in poor query performance due to the excessive use of join queries. In general our advice is if you're going to use inheritance, don't abuse it and don't make your inheritance hierarchy too deep.

Polymorphic Queries
The upshot of inheritance is that you get the ability to polymorphically query. For example using the list method on the `Content` super class will return all sub classes of `Content`:

```
def content = Content.list() // list all blog entries, books and pod casts
content = Content.findAllByAuthor('Joe Bloggs') // find all by author
def podCasts = PodCast.list() // list only pod casts
```

5.2.4 Sets, Lists and Maps

Sets of objects
By default when you define a relationship with GORM it is a `java.util.Set` which is an unordered collection that cannot contain duplicates. In other words when you have:

```
class Author {
    static hasMany = [books:Book]
}
```

The books property that GORM injects is a `java.util.Set`. The problem with this is there is no ordering when accessing the collection, which may not be what you want. To get custom ordering you can say that the set is a `SortedSet`:

```
class Author {
    SortedSet books
    static hasMany = [books:Book]
}
```

In this case a `java.util.SortedSet` implementation is used which means you have to implement `java.lang.Comparable` in your Book class:

53

```
class Book implements Comparable {
    String title
    Date releaseDate = new Date()
    int compareTo(obj) {
        releaseDate.compareTo(obj.releaseDate)
    }
}
```

The result of the above class is that the Book instances in the books collections of the Author class will be ordered by their release date.

Lists of objects
If you simply want to be able to keep objects in the order which they were added and to be able to reference them by index like an array you can define your collection type as a `List`:

```
class Author {
    List books
    static hasMany = [books:Book]
}
```

In this case when you add new elements to the books collection the order is retained in a sequential list indexed from 0 so you can do:

```
author.books[0] // get the first book
```

The way this works at the database level is Hibernate creates a `books_idx` column where it saves the index of the elements in the collection in order to retain this order at the db level.
When using a `List`, elements must be added to the collection before being saved, otherwise Hibernate will throw an exception (`org.hibernate.HibernateException`: null index column for collection):

```
// This won't work!
def book = new Book(title: 'The Shining')
book.save()
author.addToBooks(book)
// Do it this way instead.
def book = new Book(title: 'Misery')
author.addToBooks(book)
author.save()
```

Maps of Objects
If you want a simple map of string/value pairs GORM can map this with the following:

```
class Author {
    Map books // map of ISBN:book names
}
def a = new Author()
a.books = ["1590597583":"Grails Book"]
a.save()
```

In this case the key and value of the map MUST be strings.
If you want a Map of objects then you can do this:

```
class Book {
  Map authors
  static hasMany = [authors:Author]
}
def a = new Author(name:"Stephen King")
def book = new Book()
book.authors = [stephen:a]
book.save()
```

The static `hasMany` property defines the type of the elements within the Map. The keys for the map **must** be strings.

A Note on Collection Types and Performance
The Java `Set` type is a collection that doesn't allow duplicates. In order to ensure uniqueness when adding an entry to a `Set` association Hibernate has to load the entire associations from the database. If you have a large numbers of entries in the association this can be costly in terms of performance.
The same behavior is required for `List` types, since Hibernate needs to load the entire association in-order to maintain order. Therefore it is recommended that if you anticipate a large numbers of records in the association that you make the association bidirectional so that the link can be created on the inverse side. For example consider the following code:

```
def book = new Book(title:"New Grails Book")
def author = Author.get(1)
book.author = author
book.save()
```

In this example the association link is being created by the child (Book) and hence it is not necessary to manipulate the collection directly resulting in fewer queries and more efficient code. Given an `Author` with a large number of associated `Book` instances if you were to write code like the following you would see an impact on performance:

```
def book = new Book(title:"New Grails Book")
def author = Author.get(1)
author.addToBooks(book)
author.save()
```

5.3 Persistence Basics

A key thing to remember about Grails is that under the surface Grails is using [Hibernate](#) for persistence. If you are coming from a background of using [ActiveRecord](#) or [iBatis](#) Hibernate's "session" model may feel a little strange. Essentially, Grails automatically binds a Hibernate session to the currently executing request. This allows you to use the [save](#) and [delete](#) methods as well as other GORM methods transparently.

5.3.1 Saving and Updating

An example of using the [save](#) method can be seen below:

```
def p = Person.get(1)
p.save()
```

A major difference with Hibernate is when you call [save](#) it does not necessarily perform any SQL operations **at that point**. Hibernate typically batches up SQL statements and executes them at the end. This is typically done for you automatically by Grails, which manages your Hibernate session.
There are occasions, however, when you may want to control when those statements are executed or, in Hibernate terminology, when the session is "flushed". To do so you can use the flush argument to the save method:

```
def p = Person.get(1)
p.save(flush:true)
```

Note that in this case all pending SQL statements including previous saves will be synchronized with the db. This also allows you to catch any exceptions thrown, which is typically useful in highly concurrent scenarios involving optimistic locking:

```
def p = Person.get(1)
try {
    p.save(flush:true)
}
catch(Exception e) {
    // deal with exception
}
```

5.3.2 Deleting Objects

An example of the delete method can be seen below:

```
def p = Person.get(1)
p.delete()
```

By default Grails will use transactional write behind to perform the delete, if you want to perform the delete in place then you can use the `flush` argument:

```
def p = Person.get(1)
p.delete(flush:true)
```

Using the `flush` argument will also allow you to catch any errors that may potentially occur during a delete. A common error that may occur is if you violate a database constraint, although this is normally down to a programming or schema error. The following example shows how to catch a `DataIntegrityViolationException` that is thrown when you violate the database constraints:

```
def p = Person.get(1)
try {
    p.delete(flush:true)
}
catch(org.springframework.dao.DataIntegrityViolationException e) {
    flash.message = "Could not delete person ${p.name}"
    redirect(action:"show", id:p.id)
}
```

Note that Grails does not supply a `deleteAll` method as deleting data is discouraged and can often be avoided through boolean flags/logic.
If you really need to batch delete data you can use the executeUpdate method to do batch DML statements:

```
Customer.executeUpdate("delete Customer c where c.name = :oldName", [oldName:"Fred"])
```

5.3.3 Understanding Cascading Updates and Deletes

It is critical that you understand how cascading updates and deletes work when using GORM. The key part to remember is the `belongsTo` setting which controls which class "owns" a relationship.
Whether it is a one-to-one, one-to-many or many-to-many if you define `belongsTo` updates and deletes will

56

cascade from the owning class to its possessions (the other side of the relationship).
If you *do not* define `belongsTo` then no cascades will happen and you will have to manually save each object.
Here is an example:

```
class Airport {
    String name
    static hasMany = [flights:Flight]
}
class Flight {
    String number
    static belongsTo = [airport:Airport]
}
```

If I now create an `Airport` and add some `Flights` to it I can save the `Airport` and have the updates cascaded down to each flight, hence saving the whole object graph:

```
new Airport(name:"Gatwick")
    .addToFlights(new Flight(number:"BA3430"))
    .addToFlights(new Flight(number:"EZ0938"))
    .save()
```

Conversely if I later delete the `Airport` all `Flights` associated with it will also be deleted:

```
def airport = Airport.findByName("Gatwick")
airport.delete()
```

However, if I were to remove `belongsTo` then the above cascading deletion code **would not work**. To understand this better take a look at the summaries below that describe the default behaviour of GORM with regards to specific associations.

Bidirectional one-to-many with belongsTo

```
class A { static hasMany = [bees:B] }
class B { static belongsTo = [a:A] }
```

In the case of a bidirectional one-to-many where the many side defines a `belongsTo` then the cascade strategy is set to "ALL" for the one side and "NONE" for the many side.

Unidirectional one-to-many

```
class A { static hasMany = [bees:B] }
class B {   }
```

In the case of a unidirectional one-to-many where the many side defines no belongsTo then the cascade strategy is set to "SAVE-UPDATE".

Bidirectional one-to-many no belongsTo

```
class A { static hasMany = [bees:B] }
class B { A a }
```

In the case of a bidirectional one-to-many where the many side does not define a `belongsTo` then the cascade

strategy is set to "SAVE-UPDATE" for the one side and "NONE" for the many side.

Unidirectional One-to-one with belongsTo

```
class A {   }
class B { static belongsTo = [a:A] }
```

In the case of a unidirectional one-to-one association that defines a `belongsTo` then the cascade strategy is set to "ALL" for the owning side of the relationship (A->B) and "NONE" from the side that defines the `belongsTo` (B->A)
Note that if you need further control over cascading behaviour, you can use the ORM DSL.

5.3.4 Eager and Lazy Fetching

Associations in GORM are by default lazy. This is best explained by example:

```
class Airport {
    String name
    static hasMany = [flights:Flight]
}
class Flight {
    String number
    static belongsTo = [airport:Airport]
}
```

Given the above domain classes and the following code:

```
def airport = Airport.findByName("Gatwick")
airport.flights.each {
    println it.name
}
```

GORM will execute a single SQL query to fetch the `Airport` instance and then 1 extra query *for each* iteration over the `flights` association. In other words you get N+1 queries.
This can sometimes be optimal depending on the frequency of use of the association as you may have logic that dictates the associations is only accessed on certain occasions.

Configuring Eager Fetching
An alternative is to use eager fetching which can specified as follows:

```
class Airport {
    String name
    static hasMany = [flights:Flight]
    static mapping = {
        flight fetch:"join"
    }
}
```

In this case the association will be `Airport` instance and the `flights` association will be loaded all at once (depending on the mapping). This has the benefit of requiring fewer queries, however should be used carefully as you could load your entire database into memory with too many eager associations.

> ⚠️ Associations can also be declared non-lazy using the ORM DSL

Using Batch Fetching
Although eager fetching is appropriate for some cases, it is not always desirable. If you made everything eager you

could quite possibly load your entire database into memory resulting in performance and memory problems. An alternative to eager fetching is to use batch fetching. Essentially, you can configure Hibernate to lazily fetch results in "batches". For example:

```
class Airport {
    String name
    static hasMany = [flights:Flight]
    static mapping = {
        flight batchSize:10
    }
}
```

In this case, due to the `batchSize` argument, when you iterate over the `flights` association, Hibernate will fetch results in batches of 10. For example if you had an `Airport` that had 30 flights, if you didn't configure batch fetching you would get 1 query to fetch the `Airport` and then `30` queries to fetch each flight. With batch fetching you get 1 query to fetch the `Airport` and 3 queries to fetch each `Flight` in batches of 10. In other words, batch fetching is an optimization of the lazy fetching strategy. Batch fetching can also be configured at the class level as follows:

```
class Flight {
    …
    static mapping = {
        batchSize 10
    }
}
```

5.3.5 Pessimistic and Optimistic Locking

Optimistic Locking
By default GORM classes are configured for optimistic locking. Optimistic locking essentially is a feature of Hibernate which involves storing a version number in a special `version` column in the database.
The `version` column gets read into a `version` property that contains the current versioned state of persistent instance which you can access:

```
def airport = Airport.get(10)
println airport.version
```

When you perform updates Hibernate will automatically check the version property against the version column in the database and if they differ will throw a [StaleObjectException](#) and the transaction will be rolled back.
This is useful as it allows a certain level of atomicity without resorting to pessimistic locking that has an inherit performance penalty. The downside is that you have to deal with this exception if you have highly concurrent writes. This requires flushing the session:

```
def airport = Airport.get(10)
try {
    airport.name = "Heathrow"
    airport.save(flush:true)
}
catch(org.springframework.dao.OptimisticLockingFailureException e) {
    // deal with exception
}
```

The way you deal with the exception depends on the application. You could attempt a programmatic merge of the data or go back to the user and ask them to resolve the conflict.
Alternatively, if it becomes a problem you can resort to pessimistic locking.

Pessimistic Locking
Pessimistic locking is equivalent to doing a SQL "SELECT * FOR UPDATE" statement and locking a row in the database. This has the implication that other read operations will be blocking until the lock is released.

In Grails pessimistic locking is performed on an existing instance via the lock method:

```
def airport = Airport.get(10)
airport.lock() // lock for update
airport.name = "Heathrow"
airport.save()
```

Grails will automatically deal with releasing the lock for you once the transaction has been committed. However, in the above case what we are doing is "upgrading" from a regular SELECT to a SELECT..FOR UPDATE and another thread could still have updated the record in between the call to get() and the call to lock().
To get around this problem you can use the static lock method that takes an id just like get:

```
def airport = Airport.lock(10) // lock for update
airport.name = "Heathrow"
airport.save()
```

In this case only SELECT..FOR UPDATE is issued.

> Though Grails, through Hibernate, supports pessimistic locking, the embedded HSQLDB shipped with Grails which is used as the default in-memory database **does not**. If you need to test pessimistic locking you will need to do so against a database that does have support such as MySQL.

As well as the lock method you can also obtain a pessimistic locking using queries. For example using a dynamic finder:

```
def airport = Airport.findByName("Heathrow", [lock:true])
```

Or using criteria:

```
def airport = Airport.createCriteria().get {
    eq('name', 'Heathrow')
    lock true
}
```

5.3.6 Modification Checking

Once you have loaded and possibly modified a persistent domain class instance, it isn't straightforward to retrieve the original values. If you try to reload the instance using get Hibernate will return the current modified instance from its Session cache. Reloading using another query would trigger a flush which could cause problems if your data isn't ready to be flushed yet. So GORM provides some methods to retrieve the original values that Hibernate caches when it loads the instance (which it uses for dirty checking).

isDirty
You can use the isDirty method to check if any field has been modified:

```
def airport = Airport.get(10)
assert !airport.isDirty()
airport.properties = params
if (airport.isDirty()) {
   // do something based on changed state
}
```

You can also check if individual fields have been modified:

```
def airport = Airport.get(10)
assert !airport.isDirty()
airport.properties = params
if (airport.isDirty('name')) {
    // do something based on changed name
}
```

getDirtyPropertyNames

You can use the getDirtyPropertyNames method to retrieve the names of modified fields; this may be empty but will not be null:

```
def airport = Airport.get(10)
assert !airport.isDirty()
airport.properties = params
def modifiedFieldNames = airport.getDirtyPropertyNames()
for (fieldName in modifiedFieldNames) {
    // do something based on changed value
}
```

getPersistentValue

You can use the getPersistentValue method to retrieve the value of a modified field:

```
def airport = Airport.get(10)
assert !airport.isDirty()
airport.properties = params
def modifiedFieldNames = airport.getDirtyPropertyNames()
for (fieldName in modifiedFieldNames) {
    def currentValue = airport."$fieldName"
    def originalValue = airport.getPersistentValue(fieldName)
    if (currentValue != originalValue) {
        // do something based on changed value
    }
}
```

5.4 Querying with GORM

GORM supports a number of powerful ways to query from dynamic finders, to criteria to Hibernate's object oriented query language HQL.

Groovy's ability to manipulate collections via GPath and methods like sort, findAll and so on combined with GORM results in a powerful combination.

However, let's start with the basics.

Listing instances

If you simply need to obtain all the instances of a given class you can use the list method:

```
def books = Book.list()
```

The list method supports arguments to perform pagination:

```
def books = Book.list(offset:10, max:20)
```

as well as sorting:

```
def books = Book.list(sort:"title", order:"asc")
```

Here, the `sort` argument is the name of the domain class property that you wish to sort on, and the `order` argument is either `asc` for **asc**ending or `desc` for **desc**ending.

Retrieval by Database Identifier
The second basic form of retrieval is by database identifier using the get method:

```
def book = Book.get(23)
```

You can also obtain a list of instances for a set of identifiers using getAll:

```
def books = Book.getAll(23, 93, 81)
```

5.4.1 Dynamic Finders

GORM supports the concept of **dynamic finders**. A dynamic finder looks like a static method invocation, but the methods themselves don't actually exist in any form at the code level.
Instead, a method is auto-magically generated using code synthesis at runtime, based on the properties of a given class. Take for example the `Book` class:

```
class Book {
    String title
    Date releaseDate
    Author author
}
class Author {
    String name
}
```

The `Book` class has properties such as `title`, `releaseDate` and `author`. These can be used by the findBy and findAllBy methods in the form of "method expressions":

```
def book = Book.findByTitle("The Stand")
book = Book.findByTitleLike("Harry Pot%")
book = Book.findByReleaseDateBetween( firstDate, secondDate )
book = Book.findByReleaseDateGreaterThan( someDate )
book = Book.findByTitleLikeOrReleaseDateLessThan( "%Something%", someDate )
```

Method Expressions
A method expression in GORM is made up of the prefix such as findBy followed by an expression that combines one or more properties. The basic form is:

```
Book.findBy([Property][Comparator][Boolean Operator])?[Property][Comparator]
```

The tokens marked with a '?' are optional. Each comparator changes the nature of the query. For example:

```
def book = Book.findByTitle("The Stand")
book =  Book.findByTitleLike("Harry Pot%")
```

In the above example the first query is equivalent to equality whilst the latter, due to the `Like` comparator, is equivalent to a SQL `like` expression.

The possible comparators include:

- `InList` - In the list of given values
- `LessThan` - less than the given value
- `LessThanEquals` - less than or equal a give value
- `GreaterThan` - greater than a given value
- `GreaterThanEquals` - greater than or equal a given value
- `Like` - Equivalent to a SQL like expression
- `Ilike` - Similar to a `Like`, except case insensitive
- `NotEqual` - Negates equality
- `Between` - Between two values (requires two arguments)
- `IsNotNull` - Not a null value (doesn't require an argument)
- `IsNull` - Is a null value (doesn't require an argument)

Notice that the last 3 require different numbers of method arguments compared to the rest, as demonstrated in the following example:

```
def now = new Date()
def lastWeek = now - 7
def book = Book.findByReleaseDateBetween( lastWeek, now )
books = Book.findAllByReleaseDateIsNull()
books = Book.findAllByReleaseDateIsNotNull()
```

Boolean logic (AND/OR)

Method expressions can also use a boolean operator to combine two criteria:

```
def books =
    Book.findAllByTitleLikeAndReleaseDateGreaterThan("%Java%", new Date()-30)
```

In this case we're using `And` in the middle of the query to make sure both conditions are satisfied, but you could equally use `Or`:

```
def books =
    Book.findAllByTitleLikeOrReleaseDateGreaterThan("%Java%", new Date()-30)
```

At the moment, you can only use dynamic finders with a maximum of two criteria, i.e. the method name can only have one boolean operator. If you need to use more, you should consider using either Criteria or the HQL.

Querying Associations

Associations can also be used within queries:

```
def author = Author.findByName("Stephen King")
def books = author ? Book.findAllByAuthor(author) : []
```

In this case if the `Author` instance is not null we use it in a query to obtain all the `Book` instances for the given `Author`.

Pagination & Sorting

The same pagination and sorting parameters available on the list method can also be used with dynamic finders by supplying a map as the final parameter:

```
def books =
  Book.findAllByTitleLike("Harry Pot%", [max:3,
                                         offset:2,
                                         sort:"title",
                                         order:"desc"])
```

5.4.2 Criteria

Criteria is a type safe, advanced way to query that uses a Groovy builder to construct potentially complex queries. It is a much better alternative to using StringBuffer.
Criteria can be used either via the createCriteria or withCriteria methods. The builder uses Hibernate's Criteria API, the nodes on this builder map the static methods found in the Restrictions class of the Hibernate Criteria API.
Example Usage:

```
def c = Account.createCriteria()
def results = c {
    between("balance", 500, 1000)
    eq("branch", "London")
    or {
        like("holderFirstName", "Fred%")
        like("holderFirstName", "Barney%")
    }
    maxResults(10)
    order("holderLastName", "desc")
}
```

This criteria will select up to 10 `Account` objects matching the following criteria:

- `balance` is between 500 and 1000
- `branch` is 'London'
- `holderFirstName` starts with 'Fred' or 'Barney'

The results will be sorted in descending ordery by `holderLastName`.

Conjunctions and Disjunctions
As demonstrated in the previous example you can group criteria in a logical OR using a `or  {  }` block:

```
or {
    between("balance", 500, 1000)
    eq("branch", "London")
}
```

This also works with logical AND:

```
and {
    between("balance", 500, 1000)
    eq("branch", "London")
}
```

And you can also negate using logical NOT:

```
not {
    between("balance", 500, 1000)
    eq("branch", "London")
}
```

All top level conditions are implied to be AND'd together.

Querying Associations

Associations can be queried by having a node that matches the property name. For example say the `Account` class had many `Transaction` objects:

```
class Account {
    …
    static hasMany = [transactions:Transaction]
    …
}
```

We can query this association by using the property name `transaction` as a builder node:

```
def c = Account.createCriteria()
def now = new Date()
def results = c.list {
    transactions {
        between('date',now-10, now)
    }
}
```

The above code will find all the `Account` instances that have performed `transactions` within the last 10 days. You can also nest such association queries within logical blocks:

```
def c = Account.createCriteria()
def now = new Date()
def results = c.list {
    or {
        between('created',now-10,now)
        transactions {
            between('date',now-10, now)
        }
    }
}
```

Here we find all accounts that have either performed transactions in the last 10 days OR have been recently created in the last 10 days.

Querying with Projections

Projections may be used to customise the results. To use projections you need to define a "projections" node within the criteria builder tree. There are equivalent methods within the projections node to the methods found in the Hibernate [Projections](#) class:

```
def c = Account.createCriteria()
def numberOfBranches = c.get {
    projections {
        countDistinct('branch')
    }
}
```

Using SQL Restrictions

You can access Hibernate's SQL Restrictions capabilities.

```
def c = Person.createCriteria()
def peopleWithShortFirstNames = c.list {
    sqlRestriction "char_length( first_name ) <= 4"
}
```

65

> ⚠️ Note that the parameter there is SQL. The `first_name` attribute referenced in the example relates to the persistence model, not the object model. The `Person` class may have a property named `firstName` which is mapped to a column in the database named `first_name`.
> Also note that the SQL used here is not necessarily portable across databases.

Using Scrollable Results

You can use Hibernate's [ScrollableResults](#) feature by calling the scroll method:

```
def results = crit.scroll {
    maxResults(10)
}
def f = results.first()
def l = results.last()
def n = results.next()
def p = results.previous()
def future = results.scroll(10)
def accountNumber = results.getLong('number')
```

To quote the documentation of Hibernate ScrollableResults:

> *A result iterator that allows moving around within the results by arbitrary increments. The Query / ScrollableResults pattern is very similar to the JDBC PreparedStatement/ ResultSet pattern and the semantics of methods of this interface are similar to the similarly named methods on ResultSet.*

Contrary to JDBC, columns of results are numbered from zero.

Setting properties in the Criteria instance

If a node within the builder tree doesn't match a particular criterion it will attempt to set a property on the Criteria object itself. Thus allowing full access to all the properties in this class. The below example calls `setMaxResults` and `setFirstResult` on the [Criteria](#) instance:

```
import org.hibernate.FetchMode as FM
…
def results = c.list {
    maxResults(10)
    firstResult(50)
    fetchMode("aRelationship", FM.EAGER)
}
```

Querying with Eager Fetching

In the section on [Eager and Lazy Fetching](#) we discussed how to declaratively specify fetching to avoid the N+1 SELECT problem. However, this can also be achieved using a criteria query:

```
def criteria = Task.createCriteria()
def tasks = criteria.list{
    eq "assignee.id", task.assignee.id
    join 'assignee'
    join 'project'
    order 'priority', 'asc'
}
```

Notice the usage of the `join` method. This method indicates the criteria API that a `JOIN` query should be used to obtain the results.

Method Reference

If you invoke the builder with no method name such as:

```
c { … }
```

The build defaults to listing all the results and hence the above is equivalent to:

```
c.list { … }
```

Method	Description
list	This is the default method. It returns all matching rows.
get	Returns a unique result set, i.e. just one row. The criteria has to be formed that way, that it only queries one row. This method is not to be confused with a limit to just the first row.
scroll	Returns a scrollable result set.
listDistinct	If subqueries or associations are used, one may end up with the same row multiple times in the result set, this allows listing only distinct entities and is equivalent to DISTINCT_ROOT_ENTITY of the CriteriaSpecification class.
count	Returns the number of matching rows.

5.4.3 Hibernate Query Language (HQL)

GORM classes also support Hibernate's query language HQL, a very complete reference for which can be found Chapter 14. HQL: The Hibernate Query Language of the Hibernate documentation.
GORM provides a number of methods that work with HQL including find, findAll and executeQuery. An example of a query can be seen below:

```
def results =
      Book.findAll("from Book as b where b.title like 'Lord of the%'")
```

Positional and Named Parameters
In this case the value passed to the query is hard coded, however you can equally use positional parameters:

```
def results =
      Book.findAll("from Book as b where b.title like ?", ["The Shi%"])
```

```
def author = Author.findByName("Stephen King")
def books = Book.findAll("from Book as book where book.author = ?", [author])
```

Or even named parameters:

```
def results =
      Book.findAll("from Book as b where b.title like :search or b.author like :search",
                   [search:"The Shi%"])
```

```
def author = Author.findByName("Stephen King")
def books = Book.findAll("from Book as book where book.author = :author",
                         [author: author])
```

Multiline Queries

If you need to separate the query across multiple lines you can use a line continuation character:

```
def results = Book.findAll("\
from Book as b, \
     Author as a \
where b.author = a and a.surname = ?", ['Smith'])
```

> ⚠ Groovy multiline strings will NOT work with HQL queries

Pagination and Sorting

You can also perform pagination and sorting whilst using HQL queries. To do so simply specify the pagination options as a map at the end of the method call and include an "ORDER BY" clause in the HQL:

```
def results =
      Book.findAll("from Book as b where b.title like 'Lord of the%' order by b.title asc",
                   [max:10, offset:20])
```

5.5 Advanced GORM Features

The following sections cover more advanced usages of GORM including caching, custom mapping and events.

5.5.1 Events and Auto Timestamping

GORM supports the registration of events as methods that get fired when certain events occurs such as deletes, inserts and updates. The following is a list of supported events:

- `beforeInsert` - Executed before an object is initially persisted to the databse
- `beforeUpdate` - Executed before an object is updated
- `beforeDelete` - Executed before an object is deleted
- `afterInsert` - Executed after an object is persisted to the database
- `afterUpdate` - Executed after an object has been updated
- `afterDelete` - Executed after an object has been updated
- `onLoad` - Executed when an object is loaded from the database

To add an event simply register the relevant closure with your domain class.

> ⛔ Do not attempt to flush the session within an event (such as with obj.save(flush:true)). Since events are fired during flushing this will cause a StackOverflowError.

Event types

The beforeInsert event
Fired before an object is saved to the db

```
class Person {
   Date dateCreated
   def beforeInsert() {
       dateCreated = new Date()
   }
}
```

The beforeUpdate event
Fired before an existing object is updated

```
class Person {
   Date dateCreated
   Date lastUpdated
   def beforeInsert() {
       dateCreated = new Date()
   }
   def beforeUpdate() {
       lastUpdated = new Date()
   }
}
```

The beforeDelete event
Fired before an object is deleted.

```
class Person {
   String name
   Date dateCreated
   Date lastUpdated
   def beforeDelete() {
       ActivityTrace.withNewSession {
           new ActivityTrace(eventName:"Person Deleted",data:name).save()
       }
   }
}
```

Notice the usage of `withNewSession` method above. Since events are triggered whilst Hibernate is flushing using persistence methods like `save()` and `delete()` won't result in objects being saved unless you run your operations with a new `Session`.

Fortunately the `withNewSession` method allows you to share the same transactional JDBC connection even though you're using a different underlying `Session`.

The onLoad event
Fired when an object is loaded from the db:

```
class Person {
   String name
   Date dateCreated
   Date lastUpdated
   def onLoad() {
       name = "I'm loaded"
   }
}
```

Automatic timestamping
The examples above demonstrated using events to update a `lastUpdated` and `dateCreated` property to keep track of updates to objects. However, this is actually not necessary. By merely defining a `lastUpdated` and `dateCreated` property these will be automatically updated for you by GORM.
If this is not the behaviour you want you can disable this feature with:

```
class Person {
   Date dateCreated
   Date lastUpdated
   static mapping = {
       autoTimestamp false
   }
}
```

5.5.2 Custom ORM Mapping

Grails domain classes can be mapped onto many legacy schemas via an Object Relational Mapping Domain Specify Language. The following sections takes you through what is possible with the ORM DSL.

> ⚠ None if this is necessary if you are happy to stick to the conventions defined by GORM for table, column names and so on. You only needs this functionality if you need to in anyway tailor the way GORM maps onto legacy schemas or performs caching

Custom mappings are defined using a a static `mapping` block defined within your domain class:

```
class Person {
  ..
  static mapping = {
  }
}
```

5.5.2.1 Table and Column Names

Table names
The database table name which the class maps to can be customized using a call to `table`:

```
class Person {
  ..
  static mapping = {
      table 'people'
  }
}
```

In this case the class would be mapped to a table called `people` instead of the default name of `person`.

Column names
It is also possible to customize the mapping for individual columns onto the database. For example if its the name you want to change you can do:

```
class Person {
  String firstName
  static mapping = {
      table 'people'
      firstName column:'First_Name'
  }
}
```

In this case we define method calls that match each property name (in this case `firstName`). We then use the named parameter `column`, to specify the column name to map onto.

Column type
GORM supports configuration of Hibernate types via the DSL using the type attribute. This includes specifing user types that subclass the Hibernate org.hibernate.usertype.UserType class, which allows complete customization of how a type is persisted. As an example if you had a `PostCodeType` you could use it as follows:

```
class Address {
    String number
    String postCode
    static mapping = {
      postCode type:PostCodeType
    }
}
```

Alternatively if you just wanted to map it to one of Hibernate's basic types other than the default chosen by Grails

you could use:

```
class Address {
    String number
    String postCode
    static mapping = {
       postCode type:'text'
    }
}
```

This would make the `postCode` column map to the SQL TEXT or CLOB type depending on which database is being used.
See the Hibernate documentation regarding [Basic Types](#) for further information.

One-to-One Mapping
In the case of associations it is also possible to change the foreign keys used to map associations. In the case of a one-to-one association this is exactly the same as any regular column. For example consider the below:

```
class Person {
  String firstName
  Address address
  static mapping = {
      table 'people'
      firstName column:'First_Name'
      address column:'Person_Adress_Id'
  }
}
```

By default the `address` association would map to a foreign key column called `address_id`. By using the above mapping we have changed the name of the foreign key column to `Person_Adress_Id`.

One-to-Many Mapping
With a bidirectional one-to-many you can change the foreign key column used simple by changing the column name on the many side of the association as per the example in the previous section on one-to-one associations. However, with unidirectional association the foreign key needs to be specified on the association itself. For example given a unidirectional one-to-many relationship between `Person` and `Address` the following code will change the foreign key in the `address` table:

```
class Person {
  String firstName
  static hasMany = [addresses:Address]
  static mapping = {
      table 'people'
      firstName column:'First_Name'
      addresses column:'Person_Address_Id'
  }
}
```

If you don't want the column to be in the `address` table, but instead some intermediate join table you can use the `joinTable` parameter:

```
class Person {
  String firstName
  static hasMany = [addresses:Address]
  static mapping = {
      table 'people'
      firstName column:'First_Name'
      addresses joinTable:[name:'Person_Addresses', key:'Person_Id', column:'Address_Id']
  }
}
```

Many-to-Many Mapping

Grails, by default maps a many-to-many association using a join table. For example consider the below many-to-many association:

```
class Group {
    …
    static hasMany = [people:Person]
}
class Person {
    …
    static belongsTo = Group
    static hasMany = [groups:Group]
}
```

In this case Grails will create a join table called `group_person` containing foreign keys called `person_id` and `group_id` referencing the `person` and `group` tables. If you need to change the column names you can specify a column within the mappings for each class.

```
class Group {
    …
    static mapping = {
        people column:'Group_Person_Id'
    }
}
class Person {
    …
    static mapping = {
        groups column:'Group_Group_Id'
    }
}
```

You can also specify the name of the join table to use:

```
class Group {
    …
    static mapping = {
        people column:'Group_Person_Id',joinTable:'PERSON_GROUP_ASSOCIATIONS'
    }
}
class Person {
    …
    static mapping = {
        groups column:'Group_Group_Id',joinTable:'PERSON_GROUP_ASSOCIATIONS'
    }
}
```

5.5.2.2 Caching Strategy

Setting up caching

Hibernate features a second-level cache with a customizable cache provider. This needs to be configured in the `grails-app/conf/DataSource.groovy` file as follows:

```
hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
    cache.provider_class='org.hibernate.cache.EhCacheProvider'
}
```

You can of course customize these settings how you desire, for example if you want to use a distributed caching mechanism.

Caching instances

In your mapping block to enable caching with the default settings use a call to the `cache` method:

```
class Person {
  ..
  static mapping = {
      table 'people'
      cache true
  }
}
```

This will configure a 'read-write' cache that includes both lazy and non-lazy properties. If you need to customize this further you can do:

```
class Person {
  ..
  static mapping = {
      table 'people'
      cache usage:'read-only', include:'non-lazy'
  }
}
```

Caching associations

As well as the ability to use Hibernate's second level cache to cache instances you can also cache collections (associations) of objects. For example:

```
class Person {
  String firstName
  static hasMany = [addresses:Address]
  static mapping = {
      table 'people'
      version false
      addresses column:'Address', cache:true
  }
}
class Address {
    String number
    String postCode
}
```

This will enable a 'read-write' caching mechanism on the addresses collection. You can also use:

```
cache:'read-write' // or 'read-only' or 'transactional'
```

To further configure the cache usage.

Caching Queries

You can cache queries such as dynamic finders and criteria. To do so using a dynamic finder you can pass the `cache` argument:

```
def person = Person.findByFirstName("Fred", [cache:true])
```

73

> ⚠ Note that in order for the results of the query to be cached, you still need to enable caching in your mapping as discussed in the previous section.

You can also cache criteria queries:

```
def people = Person.withCriteria {
    like('firstName', 'Fr%')
    cache true
}
```

Cache usages

Below is a description of the different cache settings and their usages:

- ○ `read-only` - If your application needs to read but never modify instances of a persistent class, a read-only cache may be used.
- ○ `read-write` - If the application needs to update data, a read-write cache might be appropriate.
- ○ `nonstrict-read-write` - If the application only occasionally needs to update data (ie. if it is extremely unlikely that two transactions would try to update the same item simultaneously) and strict transaction isolation is not required, a `nonstrict-read-write` cache might be appropriate.
- ○ `transactional` - The `transactional` cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache may only be used in a JTA environment and you must specify `hibernate.transaction.manager_lookup_class` in the `grails-app/conf/DataSource.groovy` file's `hibernate` config.

5.5.2.3 Inheritance Strategies

By default GORM classes uses `table-per-hierarchy` inheritance mapping. This has the disadvantage that columns cannot have a `NOT-NULL` constraint applied to them at the db level. If you would prefer to use a `table-per-subclass` inheritance strategy you can do so as follows:

```
class Payment {
    Long id
    Long version
    Integer amount
    static mapping = {
        tablePerHierarchy false
    }
}
class CreditCardPayment extends Payment  {
    String cardNumber
}
```

The mapping of the root `Payment` class specifies that it will not be using `table-per-hierarchy` mapping for all child classes.

5.5.2.4 Custom Database Identity

You can customize how GORM generates identifiers for the database using the DSL. By default GORM relies on the native database mechanism for generating ids. This is by far the best approach, but there are still many schemas that have different approaches to identity.

To deal with this Hibernate defines the concept of an id generator. You can customize the id generator and the column it maps to as follows:

```
class Person {
    ..
    static mapping = {
        table 'people'
        version false
        id generator:'hilo', params:[table:'hi_value',column:'next_value',max_lo:100]
    }
}
```

In this case we're using one of Hibernate's built in 'hilo' generators that uses a separate table to generate ids.

⚠️ For more information on the different Hibernate generators refer to the [Hibernate reference documentation](#)

Note that if you want to merely customise the column that the id lives on you can do:

```
class Person {
    ..
    static mapping = {
        table 'people'
        version false
        id column:'person_id'
    }
}
```

5.5.2.5 Composite Primary Keys

GORM supports the concept of composite identifiers (identifiers composed from 2 or more properties). It is not an approach we recommend, but is available to you if you need it:

```
class Person {
    String firstName
    String lastName
    static mapping = {
        id composite:['firstName', 'lastName']
    }
}
```

The above will create a composite id of the `firstName` and `lastName` properties of the Person class. When you later need to retrieve an instance by id you have to use a prototype of the object itself:

```
def p = Person.get(new Person(firstName:"Fred", lastName:"Flintstone"))
println p.firstName
```

5.5.2.6 Database Indices

To get the best performance out of your queries it is often necessary to tailor the table index definitions. How you tailor them is domain specific and a matter of monitoring usage patterns of your queries. With GORM's DSL you can specify which columns need to live in which indexes:

```
class Person {
   String firstName
   String address
   static mapping = {
       table 'people'
       version false
       id column:'person_id'
       firstName column:'First_Name', index:'Name_Idx'
       address column:'Address', index:'Name_Idx,Address_Index'
   }
}
```

Note that you cannot have any spaces in the value of the `index` attribute; in this example `index:'Name_Idx, Address_Index'` will cause an error.

5.5.2.7 Optimistic Locking and Versioning

As discussed in the section on Optimistic and Pessimistic Locking, by default GORM uses optimistic locking and automatically injects a `version` property into every class which is in turn mapped to a `version` column at the database level.
If you're mapping to a legacy schema this can be problematic, so you can disable this feature by doing the following:

```
class Person {
   ..
   static mapping = {
       table 'people'
       version false
   }
}
```

> ⚠ If you disable optimistic locking you are essentially on your own with regards to concurrent updates and are open to the risk of users losing (due to data overriding) data unless you use pessimistic locking

5.5.2.8 Eager and Lazy Fetching

Lazy Collections

As discussed in the section on Eager and Lazy fetching, by default GORM collections use lazy fetching and is is configurable through the `fetchMode` setting. However, if you prefer to group all your mappings together inside the `mappings` block you can also use the ORM DSL to configure fetching:

```
class Person {
   String firstName
   static hasMany = [addresses:Address]
   static mapping = {
       addresses lazy:false
   }
}
class Address {
   String street
   String postCode
}
```

Lazy Single-Ended Associations
In GORM, one-to-one and many-to-one associations are by default lazy. Non-lazy single ended associations can be problematic in cases when you are loading many entities which have an association to another entity as a new SELECT statement is executed for each loaded entity.
You can make one-to-one and many-to-one associations non-lazy using the same technique as for lazy collections:

```
class Person {
    String firstName
    static belongsTo = [address:Address]
    static mapping = {
        address lazy:false
    }
}
class Address {
    String street
    String postCode
}
```

Here we set the `address` property of the `Person` class will be eagerly fetched.

Lazy Single-Ended Associations and Proxies
In order to facilitate single-ended lazy associations Hibernate uses runtime generated proxies. The way this works is that Hibernate dynamically subclasses the proxied entity to create the proxy.
In the previous example Hibernate would create a subclass of `Address` and return that as a proxy to the real entity.
When you call any of the getters or setters Hibernate will initialize the the entity from the database.
Unfortunately this technique can produce surprising results. Consider the following example classes:

```
class Animal {}
class Mammal extends Animal {}
class Dog extends Mammal {
    String name
}
class Owner {
    Animal pet
}
```

Given you have an `Owner` with a `pet` association that is a `Dog` consider the following code:

```
def owner = Owner.get(1)
def pet = Animal.get(owner.petId)
if(pet instanceof Dog) {
    // doesn't work!
}
```

Now you may think this code will work, but in fact it will not. The reason is Hibernate creates a dynamic proxy by subclassing `Animal` for the `owner.pet` association and caches it in the first level cache. So even if the actual proxied class is a `Dog` it won't be an instance of the `Dog` class due to the way proxies work.
The get around this problem GORM provides an `instanceOf` method that should always be used:

```
def owner = Owner.get(1)
def pet = Animal.get(owner.petId)
if(pet?.instanceof( Dog )) {
    // this works
}
```

However, there are cases where this particular Hibernate abstraction may still leak through. For example:

```
def owner = Owner.get(1)
Dog pet = Animal.get(owner.petId)
```

In this case you will get a `ClassCastException` because the proxied `Animal` is not a `Dog` even though the actual instance **is** a `Dog`.
Our best advice is to be aware of Hibernate proxies and how to deal with them when you do run into issues.

5.5.2.9 Custom Cascade Behaviour

As describes in the section on cascading updates, the primary mechanism to control the way updates and deletes are cascading from one association to another is the belongsTo static property.
However, the ORM DSL gives you complete access to Hibernate's transitive persistence capabilities via the `cascade` attribute.
Valid settings for the cascade attribute include:

- create - cascades creation of new records from one association to another
- merge - merges the state of a detached association
- save-update - cascades only saves and updates to an association
- delete - cascades only deletes to an association
- lock - useful if a pessimistic lock should be cascaded to its associations
- refresh - cascades refreshes to an association
- evict - cascades evictions (equivalent to discard() in GORM) to associations if set
- all - cascade ALL operations to associations
- delete-orphan - Applies only to one-to-many associations and indicates that when a child is removed from an association then it should be automatically deleted

> ⚠ It is advisable to read the section in the Hibernate documentation on transitive persistence to obtain a better understanding of the different cascade styles and recommendation for their usage

To specific the cascade attribute simply define one or many (comma-separated) of the aforementioned settings as its value:

```
class Person {
  String firstName
  static hasMany = [addresses:Address]
  static mapping = {
      addresses cascade:"all-delete-orphan"
  }
}
class Address {
  String street
  String postCode
}
```

5.5.2.10 Custom Hibernate Types

You saw in an earlier section that you can use composition (via the `embedded` property) to break a table into multiple objects. You can achieve a similar effect via Hibernate's custom user types. These are not domain classes themselves, but plain Java or Groovy classes with associated. Each of these types also has a corresponding "meta-type" class that implements org.hibernate.usertype.UserType.
The Hibernate reference manual has some information on custom types, but here we will focus on how to map them in Grails. Let's start by taking a look at a simple domain class that uses an old-fashioned (pre-Java 1.5) type-safe enum class:

```
class Book {
  String title
  String author
  Rating rating
  static mapping = {
      rating type: RatingUserType
  }
}
```

All we have done is declare the `rating` field the enum type and set the property's type in the custom mapping to the corresponding `UserType` implementation. That's all you have to do to start using your custom type. If you want, you can also use the other column settings such as "column" to change the column name and "index" to add it to an index.

Custom types aren't limited to just a single column - they can be mapped to as many columns as you want. In such cases you need to explicitly define in the mapping what columns to use, since Hibernate can only use the property name for a single column. Fortunately, Grails allows you to map multiple columns to a property using this syntax:

```
class Book {
  String title
  Name author
  Rating rating
  static mapping = {
      name type: NameUserType, {
          column name: "first_name"
          column name: "last_name"
      }
      rating type: RatingUserType
  }
}
```

The above example will create "first_name" and "last_name" columns for the `author` property. You'll be pleased to know that you can also use some of the normal column/property mapping attributes in the column definitions. For example:

```
column name: "first_name", index: "my_idx", unique: true
```

The column definitions do *not* support the following attributes: `type`, `cascade`, `lazy`, `cache`, and `joinTable`. One thing to bear in mind with custom types is that they define the *SQL types* for the corresponding database columns. That helps take the burden of configuring them yourself, but what happens if you have a legacy database that uses a different SQL type for one of the columns? In that case, you need to override column's SQL type using the `sqlType` attribute:

```
class Book {
  String title
  Name author
  Rating rating
  static mapping = {
      name type: NameUserType, {
          column name: "first_name", sqlType: "text"
          column name: "last_name", sqlType: "text"
      }
      rating type: RatingUserType, sqlType: "text"
  }
}
```

Mind you, the SQL type you specify needs to still work with the custom type. So overriding a default of "varchar" with "text" is fine, but overriding "text" with "yes_no" isn't going to work.

5.5.2.11 Derived Properties

A derived property is a property that takes its value from a SQL expression, often but not necessarily based on the value of some other persistent property. Consider a Product class like this:

```
class Product {
  Float price
  Float taxRate
  Float tax
}
```

If the `tax` property is derived based on the value of `price` and `taxRate` properties then there may be no need to persist the `tax` property in the database. The SQL used to derive the value of a derived property may be expressed in the ORM DSL like this:

```
class Product {
  Float price
  Float taxRate
  Float tax
  static mapping = {
      tax formula: 'PRICE * TAX_RATE'
  }
}
```

Note that the formula expressed in the ORM DSL is SQL so references to other properties should relate to the persistence model not the object model, which is why the example refers to PRICE and TAX_RATE instead of price and taxRate.
With that in place, when a Product is retrieved with something like Product.get(42), the SQL that is generated to support that will look something like this:

```
select
    product0_.id as id1_0_,
    product0_.version as version1_0_,
    product0_.price as price1_0_,
    product0_.tax_rate as tax4_1_0_,
    product0_.PRICE * product0_.TAX_RATE as formula1_0_
from
    product product0_
where
    product0_.id=?
```

Since the tax property is being derived at runtime and not stored in the database it might seem that the same effect could be achieved by adding a method like getTax() to the Product class that simply returns the product of the taxRate and price properties. With an approach like that you would give up the ability query the database based on the value of the tax property. Using a derived property allows exactly that. To retrieve all Product objects that have a tax value greater than 21.12 you could execute a query like this:

```
Product.findAllByTaxGreaterThan(21.12)
```

Derived properties may be referenced in the Criteria API:

```
Product.withCriteria {
  gt 'tax', 21.12f
}
```

The SQL that is generated to support either of those would look something like this:

```
select
    this_.id as id1_0_,
    this_.version as version1_0_,
    this_.price as price1_0_,
    this_.tax_rate as tax4_1_0_,
    this_.PRICE * this_.TAX_RATE as formula1_0_
from
    product this_
where
    this_.PRICE * this_.TAX_RATE>?
```

> ⚠️ Note that because the value of a derived property is generated in the database and depends on the execution of SQL code, derived properties may not have GORM constraints applied to them. If constraints are specified for a derived property, they will be ignored.

5.5.2.12 Custom Naming Strategy

By default Grails uses Hibernate's `ImprovedNamingStrategy` to convert domain class Class and field names to SQL table and column names by converting from camel-cased Strings to ones that use underscores as word separators. You can customize these on a per-instance basis in the `mapping` closure but if there's a consistent pattern you can specify a different `NamingStrategy` class to use.

Configure the class name to be used in `grails-app/conf/DataSource.groovy` in the `hibernate` section, e.g.

```
dataSource {
    pooled = true
    dbCreate = "create-drop"
     …
}
hibernate {
    cache.use_second_level_cache = true
     …
    naming_strategy = com.myco.myproj.CustomNamingStrategy
}
```

You can use an existing class or write your own, for example one that prefixes table names and column names:

```
package com.myco.myproj
import org.hibernate.cfg.ImprovedNamingStrategy
import org.hibernate.util.StringHelper
class CustomNamingStrategy extends ImprovedNamingStrategy {
    String classToTableName(String className) {
        "table_" + StringHelper.unqualify(className)
    }
    String propertyToColumnName(String propertyName) {
        "col_" + StringHelper.unqualify(propertyName)
    }
}
```

5.5.3 Default Sort Order

You can sort objects using queries arguments such as those found in the list method:

```
def airports = Airport.list(sort:'name')
```

However, you can also declare the sort order declaratively:

```
class Airport {
    …
    static mapping = {
        sort "name"
    }
}
```

You can also configure the sort order if necessary:

```
class Airport {
    …
    static mapping = {
        sort name:"desc"
    }
}
```

Alternatively, you can configure sort order at the association level:

```
class Airport {
    …
    static hasMany = [flights:Flight]
    static mapping = {
        flights sort:'number'
    }
}
```

5.6 Programmatic Transactions

Grails is built on Spring and hence uses Spring's Transaction abstraction for dealing with programmatic transactions. However, GORM classes have been enhanced to make this more trivial through the withTransaction method which accepts a block the first argument to which is the Spring TransactionStatus object.
A typical usage scenario is as follows:

```
def transferFunds = {
    Account.withTransaction { status ->
        def source = Account.get(params.from)
        def dest = Account.get(params.to)
        def amount = params.amount.toInteger()
        if(source.active) {
            source.balance -= amount
            if(dest.active) {
                dest.amount += amount
            }
            else {
                status.setRollbackOnly()
            }
        }

    }
}
```

In this example we rollback the transactions if the destination account is not active and if any exception are thrown during the process the transaction will automatically be rolled back.
You can also use "save points" to rollback a transaction to a particular point in time if you don't want to rollback the entire transaction. This can be achieved through the use of Spring's SavePointManager interface.
The withTransaction method deals with the begin/commit/rollback logic for you within the scope of the block.

5.7 GORM and Constraints

Although constraints are covered in the Validation section, it is important to mention them here as some of the constraints can affect the way in which the database schema is generated.
Where feasible, Grails uses a domain class's constraints to influence the database columns generated for the corresponding domain class properties.
Consider the following example. Suppose we have a domain model with the following property.

```
String name
String description
```

By default, in MySQL, Grails would define these columns as...

```
column name | data type
description | varchar(255)
```

But perhaps the business rules for this domain class state that a description can be up to 1000 characters in length. If that were the case, we'd likely define the column as follows *if* we were creating the table via an SQL script.

```
column name │ data type
description  │ TEXT
```

Chances are we'd also want to have some application-based validation to make sure we don't exceed that 1000 character limit *before* we persist any records. In Grails, we achieve this validation via constraints. We'd add the following constraint declaration to the domain class.

```
static constraints = {
    description(maxSize:1000)
}
```

This constraint would provide both the application-based validation we want and it would also cause the schema to be generated as shown above. Below is a description of the other constraints that influence schema generation.

Constraints Affecting String Properties

- inList
- maxSize
- size

If either the `maxSize` or the `size` constraint is defined, Grails sets the maximum column length based on the constraint value.

In general, it's not advisable to use both constraints on the same domain class property. However, if both the `maxSize` constraint and the `size` constraint are defined, then Grails sets the column length to the minimum of the `maxSize` constraint and the upper bound of the size constraint. (Grails uses the minimum of the two, because any length that exceeds that minimum will result in a validation error.)

If the inList constraint is defined (and the `maxSize` and the `size` constraints are not defined), then Grails sets the maximum column length based on the length of the longest string in the list of valid values. For example, given a list including values "Java", "Groovy", and "C++", Grails would set the column length to 6 (i.e., the number of characters in the string "Groovy").

Constraints Affecting Numeric Properties

- min
- max
- range

If the `max` constraint, the `min` constraint, or the `range` constraint is defined, Grails attempts to set the column precision based on the constraint value. (The success of this attempted influence is largely dependent on how Hibernate interacts with the underlying DBMS.)

In general, it's not advisable to combine the pair min/max and range constraints together on the same domain class property. However, if both of these constraints is defined, then Grails uses the minimum precision value from the constraints. (Grails uses the minimum of the two, because any length that exceeds that minimum precision will result in a validation error.)

- scale

If the scale constraint is defined, then Grails attempts to set the column scale based on the constraint value. This rule only applies to floating point numbers (i.e., java.lang.Float, java.Lang.Double, java.lang.BigDecimal, or subclasses of java.lang.BigDecimal). (The success of this attempted influence is largely dependent on how Hibernate interacts with the underlying DBMS.)

The constraints define the minimum/maximum numeric values, and Grails derives the maximum number of digits for use in the precision. Keep in mind that specifying only one of min/max constraints will not affect schema generation (since there could be large negative value of property with max:100, for example), unless specified constraint value requires more digits than default Hibernate column precision is (19 at the moment). For example...

```
someFloatValue(max:1000000, scale:3)
```

would yield:

```
someFloatValue DECIMAL(19, 3) // precision is default
```

but

```
someFloatValue(max:12345678901234567890, scale:5)
```

would yield:

```
someFloatValue DECIMAL(25, 5) // precision = digits in max + scale
```

and

```
someFloatValue(max:100, min:-100000)
```

would yield:

```
someFloatValue DECIMAL(8, 2) // precision = digits in min + default scale
```

6. The Web Layer

6.1 Controllers

A controller handles requests and creates or prepares the response and is request-scoped. In other words a new instance is created for each request. A controller can generate the response or delegate to a view. To create a controller simply create a class whose name ends with `Controller` and place it within the `grails-app/controllers` directory.

The default URL Mapping setup ensures that the first part of your controller name is mapped to a URI and each action defined within your controller maps to URI within the controller name URI.

6.1.1 Understanding Controllers and Actions

Creating a controller
Controllers can be created with the create-controller target. For example try running the following command from the root of a Grails project:

```
grails create-controller book
```

The command will result in the creation of a controller at the location `grails-app/controllers/BookController.groovy`:

```
class BookController { … }
```

`BookController` by default maps to the /book URI (relative to your application root).

> ⚠ The `create-controller` command is merely for convenience and you can just as easily create controllers using your favorite text editor or IDE

Creating Actions
A controller can have multiple properties that are each assigned a block of code. Each of these properties maps to a URI:

```
class BookController {
    def list = {
        // do controller logic
        // create model
        return model
    }
}
```

This example maps to the `/book/list` URI by default thanks to the property being named `list`.

The Default Action
A controller has the concept of a default URI that maps to the root URI of the controller. By default the default URI in this case is /book. The default URI is dictated by the following rules:

- If only one action is present the default URI for a controller maps to that action.
- If you define an `index` action which is the action that handles requests when no action is specified in the URI /book
- Alternatively you can set it explicitly with the `defaultAction` property:

```
static defaultAction = "list"
```

6.1.2 Controllers and Scopes

Available Scopes

Scopes are essentially hash like objects that allow you to store variables. The following scopes are available to controllers:

- servletContext - Also known as application scope, this scope allows you to share state across the entire web application. The servletContext is an instance of javax.servlet.ServletContext
- session - The session allows associating state with a given user and typically uses cookies to associate a session with a client. The session object is an instance of HttpSession
- request - The request object allows the storage of objects for the current request only. The request object is an instance of HttpServletRequest
- params - Mutable map of incoming request (CGI) parameters
- flash - See below.

Accessing Scopes

Scopes can be accessed using the variable names above in combination with Groovy's array index operator even on classes provided by the Servlet API such as the HttpServletRequest:

```
class BookController {
    def find = {
        def findBy = params["findBy"]
        def appContext = request["foo"]
        def loggedUser = session["logged_user"]
    }
}
```

You can even access values within scopes using the de-reference operator making the syntax even clearer:

```
class BookController {
    def find = {
        def findBy = params.findBy
        def appContext = request.foo
        def loggedUser = session.logged_user
    }
}
```

This is one of the ways that Grails unifies access to the different scopes.

Using Flash Scope

Grails supports the concept of flash scope is a temporary store for attributes which need to be available for this request and the next request only. Afterwards the attributes are cleared. This is useful for setting a message directly before redirection, for example:

```
def delete = {
    def b = Book.get( params.id )
    if(!b) {
        flash.message = "User not found for id ${params.id}"
        redirect(action:list)
    }
    … // remaining code
}
```

6.1.3 Models and Views

Returning the Model

A model is essentially a map that the view uses when rendering. The keys within that map translate to variable names accessible by the view. There are a couple of ways to return a model, the first way is to explicitly return a map instance:

```
def show = {
    [ book : Book.get( params.id ) ]
}
```

If no explicit model is returned the controller's properties will be used as the model thus allowing you to write code like this:

```
class BookController {
    List books
    List authors
    def list = {
        books = Book.list()
        authors = Author.list()
    }
}
```

> ⚠ This is possible due to the fact that controllers are prototype scoped. In other words a new controller is created for each request. Otherwise code such as the above would not be thread safe.

In the above example the books and authors properties will be available in the view.
A more advanced approach is to return an instance of the Spring ModelAndView class:

```
import org.springframework.web.servlet.ModelAndView
def index = {
    def favoriteBooks = … // get some books just for the index page, perhaps your favorites
    // forward to the list view to show them
    return new ModelAndView("/book/list", [ bookList : favoriteBooks ])
}
```

Selecting the View
In both of the previous two examples there was no code that specified which view to render. So how does Grails know which view to pick? The answer lies in the conventions. For the action:

```
class BookController {
    def show = {
        [ book : Book.get( params.id ) ]
    }
}
```

Grails will automatically look for a view at the location grails-app/views/book/show.gsp (actually Grails will try to look for a JSP first, as Grails can equally be used with JSP).
If you wish to render another view, then the render method there to help:

```
def show = {
    def map = [ book : Book.get( params.id ) ]
    render(view:"display", model:map)
}
```

In this case Grails will attempt to render a view at the location grails-app/views/book/display.gsp.
Notice that Grails automatically qualifies the view location with the book folder of the grails-app/views

directory. This is convenient, but if you have some shared views you need to access instead use:

```
def show = {
    def map = [ book : Book.get( params.id ) ]
    render(view:"/shared/display", model:map)
}
```

In this case Grails will attempt to render a view at the location `grails-app/views/shared/display.gsp`.

Rendering a Response

Sometimes its easier (typically with Ajax applications) to render snippets of text or code to the response directly from the controller. For this, the highly flexible `render` method can be used:

```
render "Hello World!"
```

The above code writes the text "Hello World!" to the response, other examples include:

```
// write some markup
render {
    for(b in books) {
        div(id:b.id, b.title)
    }
}
// render a specific view
render(view:'show')
// render a template for each item in a collection
render(template:'book_template', collection:Book.list())
// render some text with encoding and content type
render(text:"<xml>some xml</xml>",contentType:"text/xml",encoding:"UTF-8")
```

If you plan on using Groovy's MarkupBuilder to generate html for use with the render method becareful of naming clashes between html elements and Grails tags. e.g.

```
def login = {
    StringWriter w = new StringWriter()
    def builder = new groovy.xml.MarkupBuilder(w)
    builder.html{
        head{
            title 'Log in'
        }
        body{
            h1 'Hello'
            form{
            }
        }
    }
    def html = w.toString()
    render html
}
```

Will actually call the form tag (which will return some text that will be ignored by the MarkupBuilder). To correctly output a <form> elemement, use the following:

```
def login = {
    // …
    body{
        h1 'Hello'
        builder.form{
        }
    }
    // …
}
```

6.1.4 Redirects and Chaining

Redirects
Actions can be redirected using the redirect method present in all controllers:

```
class OverviewController {
    def login = {}
    def find = {
        if(!session.user)
            redirect(action:login)
        …
    }
}
```

Internally the redirect method uses the HttpServletResonse object's sendRedirect method.
The redirect method expects either:

○ Another closure within the same controller class:

```
// Call the login action within the same class
redirect(action:login)
```

○ The name of a controller and action:

```
// Also redirects to the index action in the home controller
redirect(controller:'home',action:'index')
```

○ A URI for a resource relative the application context path:

```
// Redirect to an explicit URI
redirect(uri:"/login.html")
```

○ Or a full URL:

```
// Redirect to a URL
redirect(url:"http://grails.org")
```

Parameters can be optionally passed from one action to the next using the params argument of the method:

```
redirect(action:myaction, params:[myparam:"myvalue"])
```

These parameters are made available through the [params](#) dynamic property that also accesses request parameters. If a parameter is specified with the same name as a request parameter the request parameter is overridden and the controller parameter used.

Since the `params` object is also a map, you can use it to pass the current request parameters from one action to the next:

```
redirect(action:"next", params:params)
```

Finally, you can also include a fragment in the target URI:

```
redirect(controller: "test", action: "show", fragment: "profile")
```

will (depending on the URL mappings) redirect to something like "/myapp/test/show#profile".

Chaining

Actions can also be chained. Chaining allows the model to be retained from one action to the next. For example calling the `first` action in the below action:

```
class ExampleChainController {
    def first = {
        chain(action:second,model:[one:1])
    }
    def second  = {
        chain(action:third,model:[two:2])
    }
    def third = {
        [three:3])
    }
}
```

Results in the model:

```
[one:1, two:2, three:3]
```

The model can be accessed in subsequent controller actions in the chain via the `chainModel` map. This dynamic property only exists in actions following the call to the `chain` method:

```
class ChainController {
    def nextInChain = {
        def model = chainModel.myModel
        …
    }
}
```

Like the `redirect` method you can also pass parameters to the `chain` method:

```
chain(action:"action1", model:[one:1], params:[myparam:"param1"])
```

6.1.5 Controller Interceptors

Often it is useful to intercept processing based on either request, session or application state. This can be achieved via action interceptors. There are currently 2 types of interceptors: before and after.

> ⚠ If your interceptor is likely to apply to more than one controller, you are almost certainly better off writing a [Filter](). Filters can be applied to multiple controllers or URIs, without the need to change the logic of each controller

Before Interception

The `beforeInterceptor` intercepts processing before the action is executed. If it returns `false` then the intercepted action will not be executed. The interceptor can be defined for all actions in a controller as follows:

```
def beforeInterceptor = {
    println "Tracing action ${actionUri}"
}
```

The above is declared inside the body of the controller definition. It will be executed before all actions and does not interfere with processing. A common use case is however for authentication:

```
def beforeInterceptor = [action:this.&auth,except:'login']
// defined as a regular method so its private
def auth() {
    if(!session.user) {
        redirect(action:'login')
        return false
    }
}
def login = {
    // display login page
}
```

The above code defines a method called `auth`. A method is used so that it is not exposed as an action to the outside world (i.e. it is private). The `beforeInterceptor` then defines an interceptor that is used on all actions 'except' the login action and is told to execute the 'auth' method. The 'auth' method is referenced using Groovy's method pointer syntax, within the method itself it detects whether there is a user in the session otherwise it redirects to the login action and returns false, instruction the intercepted action not to be processed.

After Interception

To define an interceptor that is executed after an action use the `afterInterceptor` property:

```
def afterInterceptor = { model ->
    println "Tracing action ${actionUri}"
}
```

The after interceptor takes the resulting model as an argument and can hence perform post manipulation of the model or response.

An after interceptor may also modify the Spring MVC [ModelAndView]() object prior to rendering. In this case, the above example becomes:

```
def afterInterceptor = { model, modelAndView ->
    println "Current view is ${modelAndView.viewName}"
    if(model.someVar) modelAndView.viewName = "/mycontroller/someotherview"
    println "View is now ${modelAndView.viewName}"
}
```

This allows the view to be changed based on the model returned by the current action. Note that the `modelAndView` may be `null` if the action being intercepted called redirect or render.

Interception Conditions

Rails users will be familiar with the authentication example and how the 'except' condition was used when executing the interceptor (interceptors are called 'filters' in Rails, this terminology conflicts with the servlet filter terminology in Java land):

```
def beforeInterceptor = [action:this.&auth,except:'login']
```

This executes the interceptor for all actions except the specified action. A list of actions can also be defined as follows:

```
def beforeInterceptor = [action:this.&auth,except:['login','register']]
```

The other supported condition is 'only', this executes the interceptor for only the specified actions:

```
def beforeInterceptor = [action:this.&auth,only:['secure']]
```

6.1.6 Data Binding

Data binding is the act of "binding" incoming request parameters onto the properties of an object or an entire graph of objects. Data binding should deal with all necessary type conversion since request parameters, which are typically delivered via a form submission, are always strings whilst the properties of a Groovy or Java object may well not be. Grails uses Spring's underlying data binding capability to perform data binding.

Binding Request Data to the Model
There are two ways to bind request parameters onto the properties of a domain class. The first involves using a domain classes' implicit constructor:

```
def save = {
  def b = new Book(params)
  b.save()
}
```

The data binding happens within the code new Book(params). By passing the params object to the domain class constructor Grails automatically recognizes that you are trying to bind from request parameters. So if we had an incoming request like:

```
/book/save?title=The%20Stand&author=Stephen%20King
```

Then the title and author request parameters would automatically get set on the domain class. If you need to perform data binding onto an existing instance then you can use the properties property:

```
def save = {
  def b = Book.get(params.id)
  b.properties = params
  b.save()
}
```

This has exactly the same effect as using the implicit constructor.

Data binding and Single-ended Associations
If you have a one-to-one or many-to-one association you can use Grails' data binding capability to update these relationships too. For example if you have an incoming request such as:

```
/book/save?author.id=20
```

Grails will automatically detect the `.id` suffix on the request parameter and look-up the `Author` instance for the given id when doing data binding such as:

```
def b = new Book(params)
```

An association property can be set to `null` by passing the literal `String` "null". For example:

```
/book/save?author.id=null
```

Data Binding and Many-ended Associations

If you have a one-to-many or many-to-many association there are different techniques for data binding depending of the association type.

If you have a `Set` based association (default for a `hasMany`) then the simplest way to populate an association is to simply send a list of identifiers. For example consider the usage of `<g:select>` below:

```
<g:select name="books"
          from="${Book.list()}"
          size="5" multiple="yes" optionKey="id"
          value="${author?.books}" />
```

This produces a select box that allows you to select multiple values. In this case if you submit the form Grails will automatically use the identifiers from the select box to populate the `books` association.

However, if you have a scenario where you want to update the properties of the associated objects the this technique won't work. Instead you have to use the subscript operator:

```
<g:textField name="books[0].title" value="the Stand" />
<g:textField name="books[1].title" value="the Shining" />
```

However, with `Set` based association it is critical that you render the mark-up in the same order that you plan to do the update in. This is because a `Set` has no concept of order, so although we're referring to `books0` and `books1` it is not guaranteed that the order of the association will be correct on the server side unless you apply some explicit sorting yourself.

This is not a problem if you use `List` based associations, since a `List` has a defined order and an index you can refer to. This is also true of `Map` based associations.

Note also that if the association you are binding to has a size of 2 and you refer to an element that is outside the size of association:

```
<g:textField name="books[0].title" value="the Stand" />
<g:textField name="books[1].title" value="the Shining" />
<g:textField name="books[2].title" value="Red Madder" />
```

Then Grails will automatically create a new instance for you at the defined position. If you "skipped" a few elements in the middle:

```
<g:textField name="books[0].title" value="the Stand" />
<g:textField name="books[1].title" value="the Shining" />
<g:textField name="books[5].title" value="Red Madder" />
```

Then Grails will automatically create instances in between. For example in the above case Grails will create 4 additional instances if the association being bound had a size of 2.

You can bind existing instances of the associated type to a `List` using the same `.id` syntax as you would use with a single-ended association. For example:

```
<g:select name="books[0].id" from="${Book.list()}" value="${author?.books[0]?.id}" />
<g:select name="books[1].id" from="${Book.list()}" value="${author?.books[1]?.id}" />
<g:select name="books[2].id" from="${Book.list()}" value="${author?.books[2]?.id}" />
```

Would allow individual entries in the `books List` to be selected separately.

Entries at particular indexes can be removed in the same way too. For example:

```
<g:select name="books[0].id" from="${Book.list()}" value="${author?.books[0]?.id}" noSelecti
```

Will render a select box that will remove the association at `books0` if the empty option is chosen.

Binding to a `Map` property works in exactly the same way except that the list index in the parameter name is replaced by the map key:

```
<g:select name="images[cover].id" from="${Image.list()}" value="${book?.images[cover]?.id}"
```

This would bind the selected image into the `Map` property `images` under a key of `"cover"`.

Data binding with Multiple domain classes
It is possible to bind data to multiple domain objects from the params object.
For example so you have an incoming request to:

```
/book/save?book.title=The%20Stand&author.name=Stephen%20King
```

You'll notice the difference with the above request is that each parameter has a prefix such as `author.` or `book.` which is used to isolate which parameters belong to which type. Grails' `params` object is like a multi-dimensional hash and you can index into to isolate only a subset of the parameters to bind.

```
def b = new Book(params['book'])
```

Notice how we use the prefix before the first dot of the `book.title` parameter to isolate only parameters below this level to bind. We could do the same with an `Author` domain class:

```
def a = new Author(params['author'])
```

Data binding and type conversion errors
Sometimes when performing data binding it is not possible to convert a particular String into a particular target type. What you get is a type conversion error. Grails will retain type conversion errors inside the errors property of a Grails domain class. Take this example:

```
class Book {
    …
    URL publisherURL
}
```

Here we have a domain class `Book` that uses the Java concrete type `java.net.URL` to represent URLs. Now say we had an incoming request such as:

```
/book/save?publisherURL=a-bad-url
```

In this case it is not possible to bind the string `a-bad-url` to the `publisherURL` property os a type mismatch error occurs. You can check for these like this:

```
def b = new Book(params)
if(b.hasErrors()) {
    println "The value ${b.errors.getFieldError('publisherURL').rejectedValue} is not a valid
}
```

Although we have not yet covered error codes (for more information see the section on [Validation](#)), for type conversion errors you would want a message to use for the error inside the grails-app/i18n/messages.properties file. You can use a generic error message handler such as:

```
typeMismatch.java.net.URL=The field {0} is not a valid URL
```

Or a more specific one:

```
typeMismatch.Book.publisherURL=The publisher URL you specified is not a valid URL
```

Data Binding and Security concerns
When batch updating properties from request parameters you need to be careful not to allow clients to bind malicious data to domain classes that end up being persisted to the database. You can limit what properties are bound to a given domain class using the subscript operator:

```
def p = Person.get(1)
p.properties['firstName','lastName'] = params
```

In this case only the `firstName` and `lastName` properties will be bound.
Another way to do this is instead of using domain classes as the target of data binding you could use [Command Objects](#). Alternatively there is also the flexible [bindData](#) method.
The `bindData` method allows the same data binding capability, but to arbitrary objects:

```
def p = new Person()
bindData(p, params)
```

However, the `bindData` method also allows you to exclude certain parameters that you don't want updated:

```
def p = new Person()
bindData(p, params, [exclude:'dateOfBirth'])
```

Or include only certain properties:

```
def p = new Person()
bindData(p, params, [include:['firstName','lastName]])
```

6.1.7 XML and JSON Responses

Using the render method to output XML
Grails' supports a few different ways to produce XML and JSON responses. The first one covered is via the render method.
The render method can be passed a block of code to do mark-up building in XML:

```
def list = {
    def results = Book.list()
    render(contentType:"text/xml") {
        books {
            for(b in results) {
                book(title:b.title)
            }
        }
    }
}
```

The result of this code would be something like:

```
<books>
      <book title="The Stand" />
      <book title="The Shining" />
</books>
```

Note that you need to be careful to avoid naming conflicts when using mark-up building. For example this code would produce an error:

```
def list = {
    def books = Book.list()  // naming conflict here
    render(contentType:"text/xml") {
        books {
            for(b in results) {
                book(title:b.title)
            }
        }
    }
}
```

The reason is that there is local variable books which Groovy attempts to invoke as a method.

Using the render method to output JSON
The render method can also be used to output JSON:

```
def list = {
    def results = Book.list()
    render(contentType:"text/json") {
        books = array {
            for(b in results) {
                book title:b.title
            }
        }
    }
}
```

In this case the result would be something along the lines of:

```
[
    {title:"The Stand"},
    {title:"The Shining"}
]
```

Again the same dangers with naming conflicts apply to JSON building.

Automatic XML Marshalling
Grails also supports automatic marshaling of domain classes to XML via special converters.
To start off with import the `grails.converters` package into your controller:

```
import grails.converters.*
```

Now you can use the following highly readable syntax to automatically convert domain classes to XML:

```
render Book.list() as XML
```

The resulting output would look something like the following::

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<list>
  <book id="1">
    <author>Stephen King</author>
    <title>The Stand</title>
  </book>
  <book id="2">
    <author>Stephen King</author>
    <title>The Shining</title>
  </book>
</list>
```

An alternative to using the converters is to use the codecs feature of Grails. The codecs feature provides encodeAsXML and encodeAsJSON methods:

```
def xml = Book.list().encodeAsXML()
render xml
```

For more information on XML marshaling see the section on REST

Automatic JSON Marshalling
Grails also supports automatic marshaling to JSON via the same mechanism. Simply substitute XML with JSON:

```
render Book.list() as JSON
```

The resulting output would look something like the following:

```
[
    {"id":1,
     "class":"Book",
     "author":"Stephen King",
     "title":"The Stand"},
    {"id":2,
     "class":"Book",
     "author":"Stephen King",
     "releaseDate":new Date(1194127343161),
     "title":"The Shining"}
  ]
```

Again as an alternative you can use the `encodeAsJSON` to achieve the same effect.

6.1.8 More on JSONBuilder

The previous section on on XML and JSON responses covered simplistic examples of rendering XML and JSON responses. Whilst the XML builder used by Grails is the standard [XmlSlurper](#) found in Groovy, the JSON builder is a custom implementation specific to Grails.

JSONBuilder and Grails versions
JSONBuilder behaves different depending on the version of Grails you use. For version below 1.2 there deprecated [grails.util.JSonBuilder](#) class is used. This section covers the usage of the Grails 1.2 JSONBuilder
For backwards compatibility the old `JSonBuilder` class is used with the `render` method for older applications, if you want to use the newer/better `JSONBuilder` class then you can do so by setting the following in `Config.groovy`:

```
grails.json.legacy.builder=false
```

Rendering Simple Objects
To render a simple JSON object just set properties within the context of the closure:

```
render(contentType:"text/json") {
    hello = "world"
}
```

The above will produce the JSON:

```
{"hello":"world"}
```

Rendering JSON Arrays
To render a list of objects simple assign a list:

```
render(contentType:"text/json") {
    categories = ['a', 'b', 'c']
}
```

This will produce:

```
{"categories":["a","b","c"]}
```

You can also render lists of complex objects, for example:

```
render(contentType:"text/json") {
    categories = [ { a = "A" }, { b = "B" } ]
}
```

This will produce:

```
{"categories":[ {"a":"A"} , {"b":"B"}] }
```

If you want to return a list as the root then you have to use the special element method:

```
render(contentType:"text/json") {
    element 1
    element 2
    element 3
}
```

The above code produces:

```
[1,2,3]
```

Rendering Complex Objects

Rendering complex objects can be done with closures. For example:

```
render(contentType:"text/json") {
    categories = ['a', 'b', 'c']
    title ="Hello JSON"
    information = {
        pages = 10
    }
}
```

The above will produce the JSON:

```
{"categories":["a","b","c"],"title":"Hello JSON","information":{"pages":10}}
```

Arrays of Complex Objects

As mentioned previously you can nest complex objects within arrays using closures:

```
render(contentType:"text/json") {
    categories = [ { a = "A" }, { b = "B" } ]
}
```

However, if you need to build them up dynamically then you may want to use the array method:

```
def results = Book.list()
render(contentType:"text/json") {
    books = array {
        for(b in results) {
            book title:b.title
        }
    }
}
```

Direct JSONBuilder API Access

If you don't have access to the render method, but still want to produce JSON you can use the API directly:

```
def builder = new JSONBuilder()
def result = builder.build {
    categories = ['a', 'b', 'c']
    title ="Hello JSON"
    information = {
        pages = 10
    }
}
// prints the JSON text
println result.toString()
def sw = new StringWriter()
result.render sw
```

6.1.9 Uploading Files

Programmatic File Uploads

Grails supports file uploads via Spring's [MultipartHttpServletRequest](#) interface. To upload a file the first step is to create a multipart form like the one below:

```
Upload Form: <br />
    <g:form action="upload" method="post" enctype="multipart/form-data">
        <input type="file" name="myFile" />
        <input type="submit" />
    </g:form>
```

There are then a number of ways to handle the file upload. The first way is to work with the Spring [MultipartFile](#) instance directly:

```
def upload = {
    def f = request.getFile('myFile')
    if(!f.empty) {
      f.transferTo( new File('/some/local/dir/myfile.txt') )
      response.sendError(200,'Done');
    }
    else {
       flash.message = 'file cannot be empty'
       render(view:'uploadForm')
    }
}
```

This is clearly handy for doing transfers to other destinations and manipulating the file directly as you can obtain an InputStream and so on via the [MultipartFile](#) interface.

File Uploads through Data Binding

File uploads can also be performed via data binding. For example say you have an Image domain class as per the below example:

```
class Image {
    byte[] myFile
}
```

Now if you create an image and pass in the `params` object such as the below example, Grails will automatically bind the file's contents as a byte to the `myFile` property:

```
def img = new Image(params)
```

It is also possible to set the contents of the file as a string by changing the type of the `myFile` property on the image to a String type:

```
class Image {
    String myFile
}
```

6.1.10 Command Objects

Grails controllers support the concept of command objects. A command object is similar to a form bean in something like Struts and they are useful in circumstances when you want to populate a subset of the properties needed to update a domain class. Or where there is no domain class required for the interaction, but you need features such as data binding and validation.

Declaring Command Objects
Command objects are typically declared in the same source file as a controller directly below the controller class definition. For example:

```
class UserController {
    …
}
class LoginCommand {
    String username
    String password
    static constraints = {
        username(blank:false, minSize:6)
        password(blank:false, minSize:6)
    }
}
```

As the previous example demonstrates you can supply constraints to command objects just as you can with domain classes.

Using Command Objects
To use command objects, controller actions may optionally specify any number of command object parameters. The parameter types must be supplied so that Grails knows what objects to create, populate and validate.
Before the controller action is executed Grails will automatically create an instance of the command object class, populate the properties of the command object with request parameters having corresponding names and the command object will be validated. For Example:

```
class LoginController {
    def login = { LoginCommand cmd ->
        if(cmd.hasErrors()) {
            redirect(action:'loginForm')
        }
        else {
            // do something else
        }
    }
}
```

Command Objects and Dependency Injection

Command objects can participate in dependency injection. This is useful if your command object has some custom validation logic which may need to interact with Grails services:

```
class LoginCommand {
    def loginService
    String username
    String password
    static constraints = {
        username validator: { val, obj ->
            obj.loginService.canLogin(obj.username, obj.password)
        }
    }
}
```

In this example the command object interacts with a bean injected by name from the Spring ApplicationContext.

6.1.11 Handling Duplicate Form Submissions

Grails has built in support for handling duplicate form submissions using the "Synchronizer Token Pattern". To get started you need to define a token on the form tag:

```
<g:form useToken="true" ...>
```

Then in your controller code you can use the withForm method to handle valid and invalid requests:

```
withForm {
    // good request
}.invalidToken {
    // bad request
}
```

If you only provide the withForm method and not the chained invalidToken method then by default Grails will store the invalid token in a flash.invalidToken variable and redirect the request back to the original page. This can then be checked in the view:

```
<g:if test="${flash.invalidToken}">
  Don't click the button twice!
</g:if>
```

The withForm tag makes use of the session and hence requires session affinity if used in a cluster.

6.1.12 Simple Type Converters

Type Conversion Methods
If you prefer to avoid the overhead of Data Binding and simply want to convert incoming parameters (typically Strings) into another more appropriate type the params object has a number of convenience methods for each type:

```
def total = params.int('total')
```

The above example uses the `int` method, there are also methods for `boolean`, `long`, `char`, `short` and so on. Each of these methods are null safe and safe from any parsing errors so you don't have to perform any addition checks on the parameters.
These same type conversion methods are also available on the `attrs` parameter of GSP tags.

Handling Multi Parameters
A common use case is dealing with multiple request parameters of the same name. For example you could get a query string such as `?name=Bob&name=Judy`.
In this case dealing with 1 parameter and dealing with many has different semantics since Groovy's iteration mechanics for `String` iterate over each character. To avoid this problem the params object provides a `list` method that always returns a list:

```
for(name in params.list('name')) {
    println name
}
```

6.2 Groovy Server Pages

Groovy Servers Pages (or GSP for short) is Grails' view technology. It is designed to be familiar for users of technologies such as ASP and JSP, but to be far more flexible and intuitive.
In Grails GSPs live in the `grails-app/views` directory and are typically rendered automatically (by convention) or via the render method such as:

```
render(view:"index")
```

A GSP is typically a mix of mark-up and GSP tags which aid in view rendering.

> ⚠ Although it is possible to have Groovy logic embedded in your GSP and doing this will be covered in this document the practice is strongly discouraged. Mixing mark-up and code is a **bad** thing and most GSP pages contain no code and needn't do so.

A GSP typically has a "model" which is a set of variables that are used for view rendering. The model is passed to the GSP view from a controller. For example consider the following controller action:

```
def show = {
    [book: Book.get(params.id)]
}
```

This action will look-up a `Book` instance and create a model that contains a key called `book`. This key can then be reference within the GSP view using the name `book`:

```
<%=book.title%>
```

6.2.1 GSP Basics

In the next view sections we'll go through the basics of GSP and what is available to you. First off let's cover some basic syntax that users of JSP and ASP should be familiar with.
GSP supports the usage of `<% %>` blocks to embed Groovy code (again this is discouraged):

```
<html>
    <body>
        <% out << "Hello GSP!" %>
    </body>
</html>
```

As well as this syntax you can also use the `<%= %>` syntax to output values:

```
<html>
    <body>
        <%="Hello GSP!" %>
    </body>
</html>
```

GSP also supports JSP-style server-side comments as the following example demonstrates:

```
<html>
    <body>
        <%-- This is my comment --%>
        <%="Hello GSP!" %>
    </body>
</html>
```

6.2.1.1 Variables and Scopes

Within the `<% %>` brackets you can of course declare variables:

```
<% now = new Date() %>
```

And then re-use those variables further down the page:

```
<%=now%>
```

However, within the scope of a GSP there are a number of pre-defined variables including:

- `application` - The javax.servlet.ServletContext instance
- `applicationContext` The Spring ApplicationContext instance
- `flash` - The flash object
- `grailsApplication` - The GrailsApplication instance
- `out` - The response writer for writing to the output stream
- `params` - The params object for retrieving request parameters
- `request` - The HttpServletRequest instance
- `response` - The HttpServletResponse instance
- `session` - The HttpSession instance
- `webRequest` - The GrailsWebRequest instance

6.2.1.2 Logic and Iteration

Using the `<% %>` syntax you can of course embed loops and so on using this syntax:

```
<html>
    <body>
        <% [1,2,3,4].each { num -> %>
            <p><%="Hello ${num}!" %></p>
        <%}%>
    </body>
</html>
```

As well as logical branching:

```
<html>
    <body>
        <% if(params.hello == 'true' )%>
        <%="Hello!"%>
        <% else %>
        <%="Goodbye!"%>
    </body>
</html>
```

6.2.1.3 Page Directives

GSP also supports a few JSP-style page directives.
The import directive allows you to import classes into the page. However, it is rarely needed due to Groovy's default imports and [GSP Tags](#):

```
<%@ page import="java.awt.*" %>
```

GSP also supports the contentType directive:

```
<%@ page contentType="text/json" %>
```

The contentType directive allows using GSP to render other formats.

6.2.1.4 Expressions

In GSP the `<%= %>` syntax introduced earlier is rarely used due to the support for GSP expressions. It is present mainly to allow ASP and JSP developers to feel at home using GSP. A GSP expression is similar to a JSP EL expression or a Groovy GString and takes the form `${expr}`:

```
<html>
   <body>
     Hello ${params.name}
   </body>
</html>
```

However, unlike JSP EL you can have any Groovy expression within the `${..}` parenthesis. Variables within the `${..}` are **not** escaped by default, so any HTML in the variable's string is output directly to the page. To reduce the risk of Cross-site-scripting (XSS) attacks, you can enable automatic HTML escaping via the `grails.views.default.codec` setting in `grails-app/conf/Config.groovy`:

```
grails.views.default.codec='html'
```

Other possible values are 'none' (for no default encoding) and 'base64'.

6.2.2 GSP Tags

Now that the less attractive JSP heritage has been set aside, the following sections cover GSP's built-in tags, which are the favored way to define GSP pages.

> ⚠️ The section on Tag Libraries covers how to add your own custom tag libraries.

All built-in GSP tags start with the prefix `g:`. Unlike JSP, you don't need to specify any tag library imports. If a tag starts with `g:` it is automatically assumed to be a GSP tag. An example GSP tag would look like:

```
<g:example />
```

GSP tags can also have a body such as:

```
<g:example>
    Hello world
</g:example>
```

Expressions can be passed into GSP tag attributes, if an expression is not used it will be assumed to be a String value:

```
<g:example attr="${new Date()}">
    Hello world
</g:example>
```

Maps can also be passed into GSP tag attributes, which are often used for a named parameter style syntax:

```
<g:example attr="${new Date()}" attr2="[one:1, two:2, three:3]">
    Hello world
</g:example>
```

Note that within the values of attributes you must use single quotes for Strings:

```
<g:example attr="${new Date()}" attr2="[one:'one', two:'two']">
    Hello world
</g:example>
```

With the basic syntax out the way, the next sections look at the tags that are built into Grails by default.

6.2.2.1 Variables and Scopes

Variables can be defined within a GSP using the set tag:

```
<g:set var="now" value="${new Date()}" />
```

Here we assign a variable called `now` to the result of a GSP expression (which simply constructs a new `java.util.Date` instance). You can also use the body of the `<g:set>` tag to define a variable:

106

```
<g:set var="myHTML">
   Some re-usable code on: ${new Date()}
</g:set>
```

Variables can also be placed in one of the following scopes:

- ○ `page` - Scoped to the current page (default)
- ○ `request` - Scoped to the current request
- ○ `flash` - Placed within flash scope and hence available for the next request
- ○ `session` - Scoped for the user session
- ○ `application` - Application-wide scope.

To select which scope a variable is placed into use the `scope` attribute:

```
<g:set var="now" value="${new Date()}" scope="request" />
```

6.2.2.2 Logic and Iteration

GSP also supports logical and iterative tags out of the box. For logic there are if, else and elseif which support your typical branching scenarios:

```
<g:if test="${session.role == 'admin'}">
   <%-- show administrative functions --%>
</g:if>
<g:else>
   <%-- show basic functions --%>
</g:else>
```

For iteration GSP has the each and while tags:

```
<g:each in="${[1,2,3]}" var="num">
   <p>Number ${num}</p>
</g:each>
<g:set var="num" value="${1}" />
<g:while test="${num < 5 }">
    <p>Number ${num++}</p>
</g:while>
```

6.2.2.3 Search and Filtering

If you have collections of objects you often need to sort and filter them in some way. GSP supports the findAll and grep for this task:

```
Stephen King's Books:
<g:findAll in="${books}" expr="it.author == 'Stephen King'">
    <p>Title: ${it.title}</p>
</g:findAll>
```

The `expr` attribute contains a Groovy expression that can be used as a filter. Speaking of filters the grep tag does a similar job such as filter by class:

```
<g:grep in="${books}" filter="NonFictionBooks.class">
    <p>Title: ${it.title}</p>
</g:grep>
```

107

Or using a regular expression:

```
<g:grep in="${books.title}" filter="~/.*?Groovy.*?/">
    <p>Title: ${it}</p>
</g:grep>
```

The above example is also interesting due to its usage of GPath. GPath is Groovy's equivalent to an XPath like language. Essentially the `books` collection is a collection of `Book` instances. However assuming each `Book` has a `title`, you can obtain a list of Book titles using the expression `books.title`. Groovy will auto-magically go through the list of Book instances, obtain each title, and return a new list!

6.2.2.4 Links and Resources

GSP also features tags to help you manage linking to controllers and actions. The link tag allows you to specify controller and action name pairing and it will automatically work out the link based on the URL Mappings, even if you change them! Some examples of the link can be seen below:

```
<g:link action="show" id="1">Book 1</g:link>
<g:link action="show" id="${currentBook.id}">${currentBook.name}</g:link>
<g:link controller="book">Book Home</g:link>
<g:link controller="book" action="list">Book List</g:link>
<g:link url="[action:'list',controller:'book']">Book List</g:link>
<g:link action="list" params="[sort:'title',order:'asc',author:currentBook.author]">
    Book List
</g:link>
```

6.2.2.5 Forms and Fields

Form Basics
GSP supports many different tags for aiding in dealing with HTML forms and fields, the most basic of which is the form tag. The `form` tag is a controller/action aware version of the regular HTML form tag. The `url` attribute allows you to specify which controller and action to map to:

```
<g:form name="myForm" url="[controller:'book',action:'list']">...</g:form>
```

In this case we create a form called `myForm` that submits to the `BookController`'s `list` action. Beyond that all of the usual HTML attributes apply.

Form Fields
As well as easy construction of forms GSP supports custom tags for dealing with different types of fields including:

- textField - For input fields of type 'text'
- checkBox - For input fields of type 'checkbox'
- radio - For input fields of type 'radio'
- hiddenField - For input fields of type 'hidden'
- select - For dealing with HTML select boxes

Each of these allow GSP expressions as the value:

```
<g:textField name="myField" value="${myValue}" />
```

GSP also contains extended helper versions of the above tags such as radioGroup (for creating groups of radio tags), localeSelect, currencySelect and timeZoneSelect (for selecting locale's, currencies and time zone's respectively).

Multiple Submit Buttons
The age old problem of dealing with multiple submit buttons is also handled elegantly with Grails via the actionSubmit tag. It is just like a regular submit, but allows you to specify an alternative action to submit to:

```
<g:actionSubmit value="Some update label" action="update" />
```

6.2.2.6 Tags as Method Calls

One major different between GSP tags and other tagging technologies is that GSP tags can be called as either regular tags or as method calls from either [controllers](#), [tag libraries](#) or GSP views.

Tags as method calls from GSPs
When called as methods tags return their results as a String instead of writing directly to the response. So for example the [createLinkTo](#) tag can equally be called as a method:

```
Static Resource: ${createLinkTo(dir:"images", file:"logo.jpg")}
```

This is particularly useful when you need to use a tag within an attribute:

```
<img src="${createLinkTo(dir:'images', file:'logo.jpg')}" />
```

In view technologies that don't support this feature you have to nest tags within tags, which becomes messy quickly and often has an adverse effect of WYSWIG tools such as Dreamweaver that attempt to render the mark-up as it is not well-formed:

```
<img src="<g:createLinkTo dir="images" file="logo.jpg" />" />
```

Tags as method calls from Controllers and Tag Libraries
You can also invoke tags from controllers and tag libraries. Tags within the default g: [namespace](#) can be invoked without the prefix and a String result is returned:

```
def imageLocation = createLinkTo(dir:"images", file:"logo.jpg")
```

However, you can also prefix the namespace to avoid naming conflicts:

```
def imageLocation = g.createLinkTo(dir:"images", file:"logo.jpg")
```

If you have a [custom namespace](#) you can use that prefix instead (Example using the [FCK Editor plugin](#)):

```
def editor = fck.editor()
```

6.2.3 Views and Templates

As well as views, Grails has the concept of templates. Templates are useful for separating out your views into maintainable chunks and combined with [Layouts](#) provide a highly re-usable mechanism for structure views.

Template Basics
Grails uses the convention of placing an underscore before the name of a view to identify it as a template. For example a you may have a template that deals with rendering Books located at
`grails-app/views/book/_bookTemplate.gsp`:

```
<div class="book" id="${book?.id}">
   <div>Title: ${book?.title}</div>
   <div>Author: ${book?.author?.name}</div>
</div>
```

To render this template from one of the views in `grails-app/views/book` you can use the [render](#) tag:

```
<g:render template="bookTemplate" model="[book:myBook]" />
```

Notice how we pass into a model to use using the `model` attribute of the render tag. If you have multiple `Book` instances you can also render the template for each `Book` using the render tag:

```
<g:render template="bookTemplate" var="book" collection="${bookList}" />
```

Shared Templates
In the previous example we had a template that was specific to the `BookController` and its views at `grails-app/views/book`. However, you may want to share templates across your application.
In this case you can place them in the root views directory at grails-app/views or any subdirectory below that location and then with the template attribute use a / before the template name to indicate the relative template path. For example if you had a template called `grails-app/views/shared/_mySharedTemplate.gsp`, you could reference it as follows:

```
<g:render template="/shared/mySharedTemplate" />
```

You can also use this technique to reference templates in any directory from any view or controller:

```
<g:render template="/book/bookTemplate" model="[book:myBook]" />
```

The Template Namespace
Since templates are used so frequently there is template namespace, called `tmpl`, available that makes using templates easier. Consider for example the following usage pattern:

```
<g:render template="bookTemplate" model="[book:myBook]" />
```

This can be expressed with the `tmpl` namespace as follows:

```
<tmpl:bookTemplate book="${myBook}" />
```

Templates in Controllers and Tag Libraries
You can also render templates from controllers using the [render](#) method found within controllers, which is useful for [Ajax](#) applications:

```
def show = {
    def b = Book.get(params.id)
    render(template:"bookTemplate", model:[book:b])
}
```

The render method within controllers writes directly to the response, which is the most common behaviour. If you need to instead obtain the result of template as a String you can use the render tag:

```
def show = {
    def b = Book.get(params.id)
    String content = g.render(template:"bookTemplate", model:[book:b])
    render content
}
```

Notice the usage of the g. namespace which tells Grails we want to use the tag as method call instead of the render method.

6.2.4 Layouts with Sitemesh

Creating Layouts

Grails leverages Sitemesh, a decorator engine, to support view layouts. Layouts are located in the grails-app/views/layouts directory. A typical layout can be seen below:

```
<html>
    <head>
        <title><g:layoutTitle default="An example decorator" /></title>
        <g:layoutHead />
    </head>
    <body onload="${pageProperty(name:'body.onload')}">
        <div class="menu"><!--my common menu goes here--></menu>
            <div class="body">
                <g:layoutBody />
            </div>
        </div>
    </body>
</html>
```

The key elements are the layoutHead, layoutTitle and layoutBody tag usages, here is what they do:

- ○ layoutTitle - outputs the target page's title
- ○ layoutHead - outputs the target pages head tag contents
- ○ layoutBody - outputs the target pages body tag contents

The previous example also demonstrates the pageProperty tag which can be used to inspect and return aspects of the target page.

Triggering Layouts

There are a few ways to trigger a layout. The simplest is to add a meta tag to the view:

```
<html>
    <head>
        <title>An Example Page</title>
        <meta name="layout" content="main"></meta>
    </head>
    <body>This is my content!</body>
</html>
```

In this case a layout called grails-app/views/layouts/main.gsp will be used to layout the page. If we were to use the layout from the previous section the output would resemble the below:

111

```
<html>
    <head>
        <title>An Example Page</title>
    </head>
    <body onload="">
        <div class="menu"><!--my common menu goes here--></div>
        <div class="body">
            This is my content!
        </div>
    </body>
</html>
```

Specifying A Layout In A Controller

Another way to specify a layout is to specify the name of the layout by assigning a value to the "layout" property in a controller. For example, if you have a controller such as:

```
class BookController {
    static layout = 'customer'
    def list = {  … }
}
```

You can create a layout called `grails-app/views/layouts/customer.gsp` which will be applied to all views that the `BookController` delegates to. The value of the "layout" property may contain a directory structure relative to the `grails-app/views/layouts/` directory. For example:

```
class BookController {
    static layout = 'custom/customer'
    def list = {  … }
}
```

Views rendered from that controller would be decorated with the `grails-app/views/layouts/custom/customer.gsp` template.

Layout by Convention

Another way to associate layouts is to use "layout by convention". For example, if you have a controller such as:

```
class BookController {
    def list = {  … }
}
```

You can create a layout called `grails-app/views/layouts/book.gsp`, by convention, which will be applied to all views that the `BookController` delegates to.
Alternatively, you can create a layout called `grails-app/views/layouts/book/list.gsp` which will only be applied to the `list` action within the `BookController`.
If you have both the above mentioned layouts in place the layout specific to the action will take precedence when the list action is executed.
If a layout may not be located using any of those conventions, the convention of last resort is to look for the application default layout which is `grails-app/views/layouts/application.gsp`. The name of the application default layout may be changed by defining a property in `grails-app/conf/Config.groovy` as follows:

```
// grails-app/conf/Config.groovy
grails.sitemesh.default.layout='myLayoutName'
```

With that property in place, the application default layout will be `grails-app/views/layouts/myLayoutName.gsp`.

Inline Layouts

Grails' also supports Sitemesh's concept of inline layouts with the applyLayout tag. The `applyLayout` tag can be used to apply a layout to a template, URL or arbitrary section of content. Essentially, this allows to even further modularize your view structure by "decorating" your template includes.

Some examples of usage can be seen below:

```
<g:applyLayout name="myLayout" template="bookTemplate" collection="${books}" />
<g:applyLayout name="myLayout" url="http://www.google.com" />
<g:applyLayout name="myLayout">
The content to apply a layout to
</g:applyLayout>
```

Server-Side Includes

While the applyLayout tag is useful for applying layouts to external content, if you simply want to include external content in the current page you can do so with the include:

```
<g:include controller="book" action="list"></g:include>
```

You can even combine the include tag and the applyLayout tag for added flexibility:

```
<g:applyLayout name="myLayout">
   <g:include controller="book" action="list"></g:include>
</g:applyLayout>
```

Finally, you can also call the include tag from a controller or tag library as a method:

```
def content = include(controller:"book", action:"list")
```

The resulting content will be provided via the return value of the include tag.

6.2.5 Sitemesh Content Blocks

Although it is useful to decorate an entire page sometimes you may find the need to decorate independent sections of your site. To do this you can use content blocks. To get started you need to divide the page to be decorate using the `<content>` tag:

```
<content tag="navbar">
… draw the navbar here…
</content>
<content tag="header">
… draw the header here…
</content>
<content tag="footer">
… draw the footer here…
</content>
<content tag="body">
… draw the body here…
</content>
```

Then within the layout you can reference these components and apply individual layouts to each:

```
<html>
    <body>
        <div id="header">
            <g:applyLayout name="headerLayout">
                <g:pageProperty name="page.header">
            </g:applyLayout>
        </div>
        <div id="nav">
            <g:applyLayout name="navLayout">
                <g:pageProperty name="page.navbar">
            </g:applyLayout>
        </div>
        <div id="body">
            <g:applyLayout name="bodyLayout">
                <g:pageProperty name="page.body">
            </g:applyLayout>
        </div>
        <div id="footer">
            <g:applyLayout name="footerLayout">
                <g:pageProperty name="page.footer">
            </g:applyLayout>
        </div>
    </body>
</html>
```

6.3 Tag Libraries

Like [Java Server Pages](#) (JSP), GSP supports the concept of custom tag libraries. Unlike JSP, Grails tag library mechanism is simply, elegant and completely reload-able at runtime.
Quite simply, to create a tag library create a Groovy class that ends with the convention `TagLib` and place it within the `grails-app/taglib` directory:

```
class SimpleTagLib {
}
```

Now to create a tag simply create property that is assigned a block of code that takes two arguments: The tag attributes and the body content:

```
class SimpleTagLib {
    def simple = { attrs, body ->
    }
}
```

The `attrs` argument is a simple map of the attributes of the tag, whilst the `body` argument is another invokable block of code that returns the body content:

```
class SimpleTagLib {
    def emoticon = { attrs, body ->
        out << body() << (attrs.happy == 'true' ? " :-)" : " :-(")
    }
}
```

As demonstrated above there is an implicit `out` variable that refers to the output `Writer` which you can use to append content to the response. Then you can simply reference the tag inside your GSP, no imports necessary:

```
<g:emoticon happy="true">Hi John</g:emoticon>
```

6.3.1 Variables and Scopes

114

Within the scope of a tag library there are a number of pre-defined variables including:

- `actionName` - The currently executing action name
- `controllerName` - The currently executing controller name
- `flash` - The [flash](#) object
- `grailsApplication` - The [GrailsApplication](#) instance
- `out` - The response writer for writing to the output stream
- `pageScope` - A reference to the [pageScope](#) object used for GSP rendering (i.e. the binding)
- `params` - The [params](#) object for retrieving request parameters
- `pluginContextPath` - The context path to the plugin that contains the tag library
- `request` - The [HttpServletRequest](#) instance
- `response` - The [HttpServletResponse](#) instance
- `servletContext` - The [javax.servlet.ServletContext](#) instance
- `session` - The [HttpSession](#) instance

6.3.2 Simple Tags

As demonstrated it the previous example it is trivial to write simple tags that have no body and merely output content. Another example is a `dateFormat` style tag:

```
def dateFormat = { attrs, body ->
    out << new java.text.SimpleDateFormat(attrs.format).format(attrs.date)
}
```

The above uses Java's `SimpleDateFormat` class to format a date and then write it to the response. The tag can then be used within a GSP as follows:

```
<g:dateFormat format="dd-MM-yyyy" date="${new Date()}" />
```

With simple tags sometimes you need to write HTML mark-up to the response. One approach would be to embed the content directly:

```
def formatBook = { attrs, body ->
    out << "<div id="${attrs.book.id}">"
    out << "Title : ${attrs.book.title}"
    out << "</div>"
}
```

Although this approach may be tempting it is not very clean. A better approach would be to re-use the [render](#) tag:

```
def formatBook = { attrs, body ->
    out << render(template:"bookTemplate", model:[book:attrs.book])
}
```

And then have a separate GSP template that does the actual rendering.

6.3.3 Logical Tags

You can also create logical tags where the body of the tag is only output once a set of conditions have been met. An example of this may be a set of security tags:

115

```
def isAdmin = { attrs, body ->
    def user = attrs['user']
    if(user != null && checkUserPrivs(user)) {
        out << body()
    }
}
```

The tag above checks if the user is an administrator and only outputs the body content if he/she has the correct set of access privileges:

```
<g:isAdmin user="${myUser}">
    // some restricted content
</g:isAdmin>
```

6.3.4 Iterative Tags

Iterative tags are trivial too, since you can invoke the body multiple times:

```
def repeat = { attrs, body ->
    attrs.times?.toInteger().times { num ->
        out << body(num)
    }
}
```

In this example we check for a `times` attribute and if it exists convert it to a number then use Groovy's `times` method to iterate by the number of times specified by the number:

```
<g:repeat times="3">
<p>Repeat this 3 times! Current repeat = ${it}</p>
</g:repeat>
```

Notice how in this example we use the implicit `it` variable to refer to the current number. This works because when we invoked the body we passed in the current value inside the iteration:

```
out << body(num)
```

That value is then passed as the default variable `it` to the tag. However, if you have nested tags this can lead to conflicts, hence you should should instead name the variables that the body uses:

```
def repeat = { attrs, body ->
    def var = attrs.var ? attrs.var : "num"
    attrs.times?.toInteger().times { num ->
        out << body((var):num)
    }
}
```

Here we check if there is a `var` attribute and if there is use that as the name to pass into the body invocation on this line:

```
out << body((var):num)
```

> ⚠️ Note the usage of the parenthesis around the variable name. If you omit these Groovy assumes you are using a String key and not referring to the variable itself.

Now we can change the usage of the tag as follows:

```
<g:repeat times="3" var="j">
<p>Repeat this 3 times! Current repeat = ${j}</p>
</g:repeat>
```

Notice how we use the `var` attribute to define the name of the variable `j` and then we are able to reference that variable within the body of the tag.

6.3.5 Tag Namespaces

By default, tags are added to the default Grails namespace and are used with the `g:` prefix in GSP pages. However, you can specify a different namespace by adding a static property to your `TagLib` class:

```
class SimpleTagLib {
    static namespace = "my"
    def example = { attrs ->
        …
    }
}
```

Here we have specified a `namespace` of `my` and hence the tags in this tag lib must then be referenced from GSP pages like this:

```
<my:example name="..." />
```

Where the prefix is the same as the value of the static `namespace` property. Namespaces are particularly useful for plugins.
Tags within namespaces can be invoked as methods using the namespace as a prefix to the method call:

```
out << my.example(name:"foo")
```

This works from GSP, controllers or tag libraries

6.3.6 Using JSP Tag Libraries

In addition to the simplified tag library mechanism provided by GSP, you can also use JSP tags from GSP. To do so simply declare the JSP you want to use via the `taglib` directive:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

Then you can use it like any other tag:

```
<fmt:formatNumber value="${10}" pattern=".00"/>
```

117

With the added bonus that you can invoke JSP tags like methods:

```
${fmt.formatNumber(value:10, pattern:".00")}
```

6.3.7 Tag return value

Since Grails 1.2, a tag library call returns an instance of
`org.codehaus.groovy.grails.web.util.StreamCharBuffer` class by default. This change improves
performance by reducing object creation and optimizing buffering during request processing. In earlier Grails
versions, a `java.lang.String` instance was returned.
Tag libraries can also return direct object values to the caller since Grails 1.2.. Object returning tag names are listed
in a static `returnObjectForTags` property in the tag library class.
Example:

```
class ObjectReturningTagLib {
    static namespace = "cms"
    static returnObjectForTags = ['content']
    def content = { attrs, body ->
        CmsContent.findByCode(attrs.code)?.content
    }
}
```

6.4 URL Mappings

Throughout the documentation so far the convention used for URLs has been the default of
`/controller/action/id`. However, this convention is not hard wired into Grails and is in fact controlled by a
URL Mappings class located at `grails-app/conf/UrlMappings.groovy`.
The `UrlMappings` class contains a single property called `mappings` that has been assigned a block of code:

```
class UrlMappings {
    static mappings = {
    }
}
```

6.4.1 Mapping to Controllers and Actions

To create a simple mapping simply use a relative URL as the method name and specify named parameters for the
controller and action to map to:

```
"/product"(controller:"product", action:"list")
```

In this case we've mapped the URL `/product` to the `list` action of the `ProductController`. You could of
course omit the action definition to map to the default action of the controller:

```
"/product"(controller:"product")
```

An alternative syntax is to assign the controller and action to use within a block passed to the method:

```
"/product" {
    controller = "product"
    action = "list"
}
```

Which syntax you use is largely dependent on personal preference. If you simply want to rewrite on URI onto another explicit URI (rather than a controller/action pair) this can be achieved with the following example:

```
"/hello"(uri:"/hello.dispatch")
```

Rewriting specific URIs is often useful when integrating with other frameworks.

6.4.2 Embedded Variables

Simple Variables
The previous section demonstrated how to map trivial URLs with concrete "tokens". In URL mapping speak tokens are the sequence of characters between each slash / character. A concrete token is one which is well defined such as as /product. However, in many circumstances you don't know what the value of a particular token will be until runtime. In this case you can use variable placeholders within the URL for example:

```
static mappings = {
  "/product/$id"(controller:"product")
}
```

In this case by embedding a $id variable as the second token Grails will automatically map the second token into a parameter (available via the params object) called id. For example given the URL /product/MacBook, the following code will render "MacBook" to the response:

```
class ProductController {
    def index = { render params.id }
}
```

You can of course construct more complex examples of mappings. For example the traditional blog URL format could be mapped as follows:

```
static mappings = {
    "/$blog/$year/$month/$day/$id"(controller:"blog", action:"show")
}
```

The above mapping would allow you to do things like:

```
/graemerocher/2007/01/10/my_funky_blog_entry
```

The individual tokens in the URL would again be mapped into the params object with values available for year, month, day, id and so on.

Dynamic Controller and Action Names
Variables can also be used to dynamically construct the controller and action name. In fact the default Grails URL mappings use this technique:

```
static mappings = {
    "/$controller/$action?/$id?"()
}
```

Here the name of the controller, action and id are implicitly obtained from the variables controller, action and id embedded within the URL.

You can also resolve the controller name and action name to execute dynamically using a closure:

```
static mappings = {
    "/$controller" {
        action = { params.goHere }
    }
}
```

Optional Variables

Another characteristic of the default mapping is the ability to append a ? at the end of a variable to make it an optional token. In a further example this technique could be applied to the blog URL mapping to have more flexible linking:

```
static mappings = {
    "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

With this mapping all of the below URLs would match with only the relevant parameters being populated in the params object:

```
/graemerocher/2007/01/10/my_funky_blog_entry
/graemerocher/2007/01/10
/graemerocher/2007/01
/graemerocher/2007
/graemerocher
```

Arbitrary Variables

You can also pass arbitrary parameters from the URL mapping into the controller by merely setting them in the block passed to the mapping:

```
"/holiday/win" {
    id = "Marrakech"
    year = 2007
}
```

This variables will be available within the params object passed to the controller.

Dynamically Resolved Variables

The hard coded arbitrary variables are useful, but sometimes you need to calculate the name of the variable based on runtime factors. This is also possible by assigning a block to the variable name:

```
"/holiday/win" {
    id = { params.id }
    isEligible = { session.user != null } // must be logged in
}
```

In the above case the code within the blocks is resolved when the URL is actually matched and hence can be used in combination with all sorts of logic.

6.4.3 Mapping to Views

If you want to resolve a URL to a view, without a controller or action involved, you can do so too. For example if you wanted to map the root URL / to a GSP at the location grails-app/views/index.gsp you could use:

```
static mappings = {
      "/"(view:"/index")  // map the root URL
}
```

Alternatively if you need a view that is specific to a given controller you could use:

```
static mappings = {
    "/help"(controller:"site",view:"help") // to a view for a controller
}
```

6.4.4 Mapping to Response Codes

Grails also allows you to map HTTP response codes to controllers, actions or views. All you have to do is use a method name that matches the response code you are interested in:

```
static mappings = {
    "403"(controller: "errors", action: "forbidden")
    "404"(controller: "errors", action: "notFound")
    "500"(controller: "errors", action: "serverError")
}
```

Or alternatively if you merely want to provide custom error pages:

```
static mappings = {
    "403"(view: "/errors/forbidden")
    "404"(view: "/errors/notFound")
    "500"(view: "/errors/serverError")
}
```

Declarative Error Handling
In addition you can configure handlers for individual exceptions:

```
static mappings = {
    "403"(view: "/errors/forbidden")
    "404"(view: "/errors/notFound")
    "500"(controller: "errors", action: "illegalArgument", exception: IllegalArgumentExceptio
    "500"(controller: "errors", action: "nullPointer", exception: NullPointerException)
    "500"(controller: "errors", action: "customException", exception: MyException)
    "500"(view: "/errors/serverError")
}
```

With this configuration, an `IllegalArgumentException` will be handled by the `illegalArgument` action in `ErrorsController`, a `NullPointerException` will be handled by the `nullPointer` action, and a `MyException` will be handled by the `customException` action. Other exceptions will be handled by the catch-all rule and use the `/errors/serverError` view.

6.4.5 Mapping to HTTP methods

URL mappings can also be configured to map based on the HTTP method (GET, POST, PUT or DELETE). This is extremely useful for RESTful APIs and for restricting mappings based on HTTP method.
As an example the following mappings provide a RESTful API URL mappings for the `ProductController`:

```
static mappings = {
    "/product/$id"(controller:"product"){
        action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
    }
}
```

6.4.6 Mapping Wildcards

Grails' URL mappings mechanism also supports wildcard mappings. For example consider the following mapping:

```
static mappings = {
    "/images/*.jpg"(controller:"image")
}
```

This mapping will match all paths to images such as /image/logo.jpg. Of course you can achieve the same effect with a variable:

```
static mappings = {
    "/images/$name.jpg"(controller:"image")
}
```

However, you can also use double wildcards to match more than one level below:

```
static mappings = {
    "/images/**.jpg"(controller:"image")
}
```

In this cases the mapping will match /image/logo.jpg as well as /image/other/logo.jpg. Even better you can use a double wildcard variable:

```
static mappings = {
    // will match /image/logo.jpg and /image/other/logo.jpg
    "/images/$name**.jpg"(controller:"image")
}
```

In this case it will store the path matched by the wildcard inside a `name` parameter obtainable from the params object:

```
def name = params.name
println name // prints "logo" or "other/logo"
```

If you are using wildcard URL mappings then you may want to exclude certain URIs from Grails' URL mapping process. To do this you can provide an `excludes` setting inside the `UrlMappings.groovy` class:

```
class UrlMappings = {
    static excludes = ["/images/**", "/css/**"]
    static mappings = {
        …
    }
}
```

In this case Grails won't attempt to match any URIs that start with `/images` or `/css`.

6.4.7 Automatic Link Re-Writing

Another great feature of URL mappings is that they automatically customize the behaviour of the <u>link</u> tag so that changing the mappings don't require you to go and change all of your links.
This is done through a URL re-writing technique that reverse engineers the links from the URL mappings. So given a mapping such as the blog one from an earlier section:

```
static mappings = {
    "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

If you use the link tag as follows:

```
<g:link controller="blog" action="show" params="[blog:'fred', year:2007]">My Blog</g:link>
<g:link controller="blog" action="show" params="[blog:'fred', year:2007, month:10]">My Blog
```

Grails will automatically re-write the URL in the correct format:

```
<a href="/fred/2007">My Blog</a>
<a href="/fred/2007/10">My Blog - October 2007 Posts</a>
```

6.4.8 Applying Constraints

URL Mappings also support Grails' unified <u>validation constraints</u> mechanism, which allows you to further "constrain" how a URL is matched. For example, if we revisit the blog sample code from earlier, the mapping currently looks like this:

```
static mappings = {
    "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

This allows URLs such as:

```
/graemerocher/2007/01/10/my_funky_blog_entry
```

However, it would also allow:

```
/graemerocher/not_a_year/not_a_month/not_a_day/my_funky_blog_entry
```

This is problematic as it forces you to do some clever parsing in the controller code. Luckily, URL Mappings can be constrained to further validate the URL tokens:

```
"/$blog/$year?/$month?/$day?/$id?" {
    controller = "blog"
    action = "show"
    constraints {
        year(matches:/\d{4}/)
        month(matches:/\d{2}/)
        day(matches:/\d{2}/)
    }
}
```

In this case the constraints ensure that the `year`, `month` and `day` parameters match a particular valid pattern thus relieving you of that burden later on.

6.4.9 Named URL Mappings

URL Mappings also support named mappings. Simply put, named mappings are mappings which have a name associated with them. The name may be used to refer to a specific mapping when links are being generated. The syntax for defining a named mapping is as follows:

```
static mappings = {
    name <mapping name>: <url pattern> {
        // …
    }
}
```

An example:

```
static mappings = {
    name personList: "/showPeople" {
        controller = 'person'
        action = 'list'
    }
    name accountDetails: "/details/$acctNumber" {
        controller = 'product'
        action = 'accountDetails'
    }
}
```

The mapping may be referenced in a link tag in a GSP.

```
<g:link mapping="personList">List People</g:link>
```

That would result in:

```
<a href="/showPeople">List People</a>
```

Parameters may be specified using the params attribute.

```
<g:link mapping="accountDetails" params="[acctNumber:'8675309']">Show Account</g:link>
```

That would result in:

```
<a href="/details/8675309">Show Account</a>
```

Alternatively you may reference a named mapping using the link namespace.

```
<link:personList>List People</link:personList>
```

That would result in:

```
<a href="/showPeople">List People</a>
```

The link namespace approach allows parameters to be specified as attributes.

```
<link:accountDetails acctNumber="8675309">Show Account</link:accountDetails>
```

That would result in:

```
<a href="/details/8675309">Show Account</a>
```

6.5 Web Flow

Overview
Grails supports the creation of web flows built on the [Spring Web Flow](#) project. A web flow is a conversation that spans multiple requests and retains state for the scope of the flow. A web flow also has a defined start and end state. Web flows don't require an HTTP session, but instead store their state in a serialized form, which is then restored using a flow execution key that Grails passes around as a request parameter. This makes flows far more scalable than other forms of stateful application that use the HttpSession and its inherit memory and clustering concerns.
Web flow is essentially an advanced state machine that manages the "flow" of execution from one state to the next. Since the state is managed for you, you don't have to be concerned with ensuring that users enter an action in the middle of some multi step flow, as web flow manages that for you. This makes web flow perfect for use cases such as shopping carts, hotel booking and any application that has multi page work flows.

Creating a Flow
To create a flow create a regular Grails controller and then add an action that ends with the convention `Flow`. For example:

```
class BookController {
    def index = {
        redirect(action:"shoppingCart")
    }
    def shoppingCartFlow = {
        …
    }
}
```

Notice when redirecting or referring to the flow as an action we omit the `Flow` suffix. In other words the name of the action of the above flow is `shoppingCart`.

6.5.1 Start and End States

As mentioned before a flow has a defined start and end state. A start state is the state which is entered when a user first initiates a conversation (or flow). The start state of A Grails flow is the first method call that takes a block. For

example:

```
class BookController {
    …
    def shoppingCartFlow = {
        showCart {
            on("checkout").to "enterPersonalDetails"
            on("continueShopping").to "displayCatalogue"
        }
        …
        displayCatalogue {
            redirect(controller:"catalogue", action:"show")
        }
        displayInvoice()
    }
}
```

Here the `showCart` node is the start state of the flow. Since the showCart state doesn't define an action or redirect it is assumed be a [view state](#) that, by convention, refers to the view
`grails-app/views/book/shoppingCart/showCart.gsp`.
Notice that unlike regular controller actions, the views are stored within a directory that matches the name of the flow: `grails-app/views/book/shoppingCart`.
The `shoppingCart` flow also has two possible end states. The first is `displayCatalogue` which performs an external redirect to another controller and action, thus exiting the flow. The second is `displayInvoice` which is an end state as it has no events at all and will simply render a view called
`grails-app/views/book/shoppingCart/displayInvoice.gsp` whilst ending the flow at the same time.
Once a flow has ended it can only be resumed from the start state, in this case `showCart`, and not from any other state.

6.5.2 Action States and View States

View states
A view state is a one that doesn't define an `action` or a `redirect`. So for example the below is a view state:

```
enterPersonalDetails {
    on("submit").to "enterShipping"
    on("return").to "showCart"
}
```

It will look for a view called `grails-app/views/book/shoppingCart/enterPersonalDetails.gsp` by default. Note that the `enterPersonalDetails` state defines two events: `submit` and `return`. The view is responsible for [triggering](#) these events. If you want to change the view to be rendered you can do so with the render method:

```
enterPersonalDetails {
    render(view:"enterDetailsView")
    on("submit").to "enterShipping"
    on("return").to "showCart"
}
```

Now it will look for `grails-app/views/book/shoppingCart/enterDetailsView.gsp`. If you want to use a shared view, start with a / in view argument:

```
enterPersonalDetails {
    render(view:"/shared/enterDetailsView")
    on("submit").to "enterShipping"
    on("return").to "showCart"
}
```

Now it will look for `grails-app/views/shared/enterDetailsView.gsp`

Action States

An action state is a state that executes code but does not render any view. The result of the action is used to dictate flow transition. To create an action state you need to define an action to to be executed. This is done by calling the `action` method and passing it a block of code to be executed:

```
listBooks {
    action {
        [ bookList:Book.list() ]
    }
    on("success").to "showCatalogue"
    on(Exception).to "handleError"
}
```

As you can see an action looks very similar to a controller action and in fact you can re-use controller actions if you want. If the action successfully returns with no errors the `success` event will be triggered. In this case since we return a map, this is regarded as the "model" and is automatically placed in flow scope.
In addition, in the above example we also use an exception handler to deal with errors on the line:

```
on(Exception).to "handleError"
```

What this does is make the flow transition to a state called `handleError` in the case of an exception.
You can write more complex actions that interact with the flow request context:

```
processPurchaseOrder  {
    action {
        def a =  flow.address
        def p = flow.person
        def pd = flow.paymentDetails
        def cartItems = flow.cartItems
        flow.clear()
        def o = new Order(person:p, shippingAddress:a, paymentDetails:pd)
        o.invoiceNumber = new Random().nextInt(9999999)
        cartItems.each { o.addToItems(it) }
        o.save()
        [order:o]
    }
    on("error").to "confirmPurchase"
    on(Exception).to "confirmPurchase"
    on("success").to "displayInvoice"
}
```

Here is a more complex action that gathers all the information accumulated from the flow scope and creates an `Order` object. It then returns the order as the model. The important thing to note here is the interaction with the request context and "flow scope".

Transition Actions

Another form of action is what is known as a *transition* action. A transition action is executed directly prior to state transition once an event has been triggered. A trivial example of a transition action can be seen below:

```
enterPersonalDetails {
    on("submit") {
        log.trace "Going to enter shipping"
    }.to "enterShipping"
    on("return").to "showCart"
}
```

Notice how we pass a block of the code to `submit` event that simply logs the transition. Transition states are extremely useful for data binding and validation, which is covered in a later section.

6.5.3 Flow Execution Events

In order to *transition* execution of a flow from one state to the next you need some way of trigger an *event* that indicates what the flow should do next. Events can be triggered from either view states or action states.

Triggering Events from a View State
As discussed previously the start state of the flow in a previous code listing deals with two possible events. A `checkout` event and a `continueShopping` event:

```
def shoppingCartFlow = {
    showCart {
        on("checkout").to "enterPersonalDetails"
        on("continueShopping").to "displayCatalogue"
    }
    …
}
```

Since the `showCart` event is a view state it will render the view `grails-app/book/shoppingCart/showCart.gsp`. Within this view you need to have components that trigger flow execution. On a form this can be done use the submitButton tag:

```
<g:form action="shoppingCart">
    <g:submitButton name="continueShopping" value="Continue Shopping"></g:submitButton>
    <g:submitButton name="checkout" value="Checkout"></g:submitButton>
</g:form>
```

The form must submit back to the `shoppingCart` flow. The name attribute of each submitButton tag signals which event will be triggered. If you don't have a form you can also trigger an event with the link tag as follows:

```
<g:link action="shoppingCart" event="checkout" />
```

Triggering Events from an Action
To trigger an event from an `action` you need to invoke a method. For example there is the built in `error()` and `success()` methods. The example below triggers the `error()` event on validation failure in a transition action:

```
enterPersonalDetails {
    on("submit") {
            def p = new Person(params)
            flow.person = p
            if(!p.validate())return error()
    }.to "enterShipping"
    on("return").to "showCart"
}
```

In this case because of the error the transition action will make the flow go back to the `enterPersonalDetails` state.
With an action state you can also trigger events to redirect flow:

```
shippingNeeded {
    action {
        if(params.shippingRequired) yes()
        else no()
    }
    on("yes").to "enterShipping"
    on("no").to "enterPayment"
}
```

6.5.4 Flow Scopes

Scope Basics

You'll notice from previous examples that we used a special object called `flow` to store objects within "flow scope". Grails flows have 5 different scopes you can utilize:

- `request` - Stores an object for the scope of the current request
- `flash` - Stores the object for the current and next request only
- `flow` - Stores objects for the scope of the flow, removing them when the flow reaches an end state
- `conversation` - Stores objects for the scope of the conversation including the root flow and nested subflows
- `session` - Stores objects inside the users session

> ⚠️ Grails service classes can be automatically scoped to a web flow scope. See the documentation on <u>Services</u> for more information.

Also returning a model map from an action will automatically result in the model being placed in flow scope. For example, using a transition action, you can place objects within `flow` scope as follows:

```
enterPersonalDetails {
    on("submit") {
        [person:new Person(params)]
    }.to "enterShipping"
    on("return").to "showCart"
}
```

Be aware that a new request is always created for each state, so an object placed in request scope in an action state (for example) will not be available in a subsequent view state. Use one of the other scopes to pass objects from one state to another. Also note that Web Flow:

1. Moves objects from flash scope to request scope upon transition between states;
2. Merges objects from the flow and conversation scopes into the view model before rendering (so you shouldn't include a scope prefix when referencing these objects within a view, e.g. GSP pages).

Flow Scopes and Serialization

When placing objects in `flash`, `flow` or `conversation` scope they must implement `java.io.Serializable` otherwise you will get an error. This has an impact on <u>domain classes</u> in that domain classes are typically placed within a scope so that they can be rendered in a view. For example consider the following domain class:

```
class Book {
    String title
}
```

In order to place an instance of the `Book` class in a flow scope you will need to modify it as follows:

```
class Book implements Serializable {
    String title
}
```

This also impacts associations and closures you declare within a domain class. For example consider this:

```
class Book implements Serializable {
    String title
    Author author
}
```

Here if the `Author` association is not `Serializable` you will also get an error. This also impacts closures used in [GORM events](#) such as `onLoad`, `onSave` and so on. The following domain class will cause an error if an instance is placed in a flow scope:

```
class Book implements Serializable {
    String title
    def onLoad = {
        println "I'm loading"
    }
}
```

The reason is that the assigned block on the `onLoad` event cannot be serialized. To get around this you should declare all events as `transient`:

```
class Book implements Serializable {
    String title
    transient onLoad = {
        println "I'm loading"
    }
}
```

6.5.5 Data Binding and Validation

In the section on [start and end states](#), the start state in the first example triggered a transition to the `enterPersonalDetails` state. This state renders a view and waits for the user to enter the required information:

```
enterPersonalDetails {
    on("submit").to "enterShipping"
    on("return").to "showCart"
}
```

The view contains a form with two submit buttons that either trigger the submit event or the return event:

```
<g:form action="shoppingCart">
    <!-- Other fields -->
    <g:submitButton name="submit" value="Continue"></g:submitButton>
    <g:submitButton name="return" value="Back"></g:submitButton>
</g:form>
```

However, what about the capturing the information submitted by the form? To to capture the form info we can use a flow transition action:

```
enterPersonalDetails {
    on("submit") {
        flow.person = new Person(params)
        !flow.person.validate() ? error() : success()
    }.to "enterShipping"
    on("return").to "showCart"
}
```

Notice how we perform data binding from request parameters and place the `Person` instance within `flow` scope. Also interesting is that we perform validation and invoke the `error()` method if validation fails. This signals to the flow that the transition should halt and return to the `enterPersonalDetails` view so valid entries can be entered by the user, otherwise the transition should continue and go to the `enterShipping` state.

Like regular actions, flow actions also support the notion of Command Objects by defining the first argument of the closure:

```
enterPersonalDetails {
    on("submit") { PersonDetailsCommand cmd ->
          flow.personDetails = cmd
          !flow.personDetails.validate() ? error() : success()
    }.to "enterShipping"
    on("return").to "showCart"
}
```

6.5.6 Subflows and Conversations

Grails' Web Flow integration also supports subflows. A subflow is like a flow within a flow. For example take this search flow:

```
def searchFlow = {
    displaySearchForm {
        on("submit").to "executeSearch"
    }
    executeSearch {
        action {
            [results:searchService.executeSearch(params.q)]
        }
        on("success").to "displayResults"
        on("error").to "displaySearchForm"
    }
    displayResults {
        on("searchDeeper").to "extendedSearch"
        on("searchAgain").to "displaySearchForm"
    }
    extendedSearch {
        subflow(extendedSearchFlow)   // <--- extended search subflow
        on("moreResults").to "displayMoreResults"
        on("noResults").to "displayNoMoreResults"
    }
    displayMoreResults()
    displayNoMoreResults()
}
```

It references a subflow in the `extendedSearch` state. The subflow is another flow entirely:

```
def extendedSearchFlow = {
    startExtendedSearch {
        on("findMore").to "searchMore"
        on("searchAgain").to "noResults"
    }
    searchMore {
        action {
            def results = searchService.deepSearch(ctx.conversation.query)
            if(!results)return error()
            conversation.extendedResults = results
        }
        on("success").to "moreResults"
        on("error").to "noResults"
    }
    moreResults()
    noResults()
}
```

Notice how it places the `extendedResults` in conversation scope. This scope differs to flow scope as it allows you to share state that spans the whole conversation not just the flow. Also notice that the end state (either `moreResults` or `noResults` of the subflow triggers the events in the main flow:

```
extendedSearch {
    subflow(extendedSearchFlow)    // <--- extended search subflow
    on("moreResults").to "displayMoreResults"
    on("noResults").to "displayNoMoreResults"
}
```

6.6 Filters

Although Grails controllers support fine grained interceptors, these are only really useful when applied to a few controllers and become difficult to manage with larger applications. Filters on the other hand can be applied across a whole group of controllers, a URI space or a to a specific action. Filters are far easier to plug-in and maintain completely separately to your main controller logic and are useful for all sorts of cross cutting concerns such as security, logging, and so on.

6.6.1 Applying Filters

To create a filter create a class that ends with the convention `Filters` in the `grails-app/conf` directory. Within this class define a code block called `filters` that contains the filter definitions:

```
class ExampleFilters {
    def filters = {
        // your filters here
    }
}
```

Each filter you define within the `filters` block has a name and a scope. The name is the method name and the scope is defined using named arguments. For example if you need to define a filter that applies to all controllers and all actions you can use wildcards:

```
sampleFilter(controller:'*', action:'*') {
  // interceptor definitions
}
```

The scope of the filter can be one of the following things:

- A controller and/or action name pairing with optional wildcards
- A URI, with Ant path matching syntax

Filter rule attributes:

- `controller` - controller matching pattern, by default * is replaced with .* and a regex is compiled
- `action` - action matching pattern, by default * is replaced with .* and a regex is compiled
- `regex` (true/false) - use regex syntax (don't replace '*' with '.*')
- `uri` - a uri to match, expressed with as Ant style path (e.g. /book/**)
- `find` (true/false) - rule matches with partial match (see java.util.regex.Matcher.find())
- `invert` (true/false) - invert the rule (NOT rule)

Some examples of filters include:

- All controllers and actions

```
all(controller:'*', action:'*') {
}
```

- Only for the `BookController`

132

```
justBook(controller:'book', action:'*') {
}
```

- All controllers except the `BookController`

```
notBook(controller:'book', invert:true) {
}
```

- All actions containing 'save' in the action name

```
saveInActionName(action:'save', find:true) {
}
```

- Applied to a URI space

```
someURIs(uri:'/book/**') {
}
```

- Applied to all URIs

```
allURIs(uri:'/**') {
}
```

In addition, the order in which you define the filters within the `filters` code block dictates the order in which they are executed. To control the order of execution between `Filters` classes, you can use the `dependsOn` property discussed in [filter dependencies](#) section.

6.6.2 Filter Types

Within the body of the filter you can then define one or several of the following interceptor types for the filter:

- `before` - Executed before the action. Can return false to indicate all future filters and the action should not execute
- `after` - Executed after an action. Takes a first argument as the view model
- `afterView` - Executed after view rendering

For example to fulfill the common authentication use case you could define a filter as follows:

```
class SecurityFilters {
    def filters = {
        loginCheck(controller:'*', action:'*') {
            before = {
                if(!session.user && !actionName.equals('login')) {
                    redirect(action:'login')
                    return false
                }
            }
        }
    }
}
```

Here the `loginCheck` filter uses a `before` interceptor to execute a block of code that checks if a user is in the session and if not redirects to the login action. Note how returning false ensure that the action itself is not executed.

6.6.3 Variables and Scopes

Filters support all the common properties available to <u>controllers</u> and <u>tag libraries,</u> plus the application context:

- <u>request</u> - The HttpServletRequest object
- <u>response</u> - The HttpServletResponse object
- <u>session</u> - The HttpSession object
- <u>servletContext</u> - The ServletContext object
- <u>flash</u> - The flash object
- <u>params</u> - The request parameters object
- <u>actionName</u> - The action name that is being dispatched to
- <u>controllerName</u> - The controller name that is being dispatched to
- <u>grailsApplication</u> - The Grails application currently running
- <u>applicationContext</u> - The ApplicationContext object

However, filters only support a subset of the methods available to controllers and tag libraries. These include:

- <u>redirect</u> - For redirects to other controllers and actions
- <u>render</u> - For rendering custom responses

6.6.4 Filter Dependencies

In a `Filters` class, you can specify any other `Filters` classes that should first be executed using the `dependsOn` property. The `dependsOn` property is used when a `Filters` class depends on the behavior of another `Filters` class (e.g. setting up the environment, modifying the request/session, etc.) and is defined as an array of `Filters` classes.
Take the following example `Filters` classes:

```
class MyFilters {
    def dependsOn = [MyOtherFilters]
    def filters = {
        checkAwesome(uri:"/*") {
            before = {
                if (request.isAwesome) { // do something awesome }
            }
        }
        checkAwesome2(uri:"/*") {
            before = {
                if (request.isAwesome) { // do something else awesome }
            }
        }
    }
}
class MyOtherFilters {
    def filters = {
        makeAwesome(uri:"/*") {
            before = {
                request.isAwesome = true;
            }
        }
        doNothing(uri:"/*") {
            before = {
                // do nothing
            }
        }
    }
}
```

MyFilters specifically `dependsOn` MyOtherFilters. This will cause all the filters in MyOtherFilters to be executed before those in MyFilters, given their scope matches the current request. For a request of "/test", which will match the scope of every filter in the example, the execution order would be as follows:

- MyOtherFilters - makeAwesome
- MyOtherFilters - doNothing
- MyFilters - checkAwesome
- MyFilters - checkAwesome2

The filters within the MyOtherFilters class are processed in order first, followed by the filters in the MyFilters class. Execution order between `Filters` classes are enabled and the execution order of filters within each `Filters` class are preserved.
If any cyclical dependencies are detected, the filters with cyclical dependencies will be added to the end of the filter

chain and processing will continue. Information about any cyclical dependencies that are detected will be written to the logs. Ensure that your root logging level is set to at least WARN or configure an appender for the Grails Filters Plugin (org.codehaus.groovy.grails.plugins.web.filters.FiltersGrailsPlugin) when debugging filter dependency issues.

6.7 Ajax

Ajax stands for Asynchronous Javascript and XML and is the driving force behind the shift to richer web applications. These types of applications in general are better suited to agile, dynamic frameworks written in languages like Ruby and Groovy Grails provides support for building Ajax applications through its Ajax tag library for a full list of these see the Tag Library Reference.

6.7.1 Ajax using Prototype

By default Grails ships with the Prototype library, but through the Plug-in system provides support for other frameworks such as Dojo Yahoo UI and the Google Web Toolkit
This section covers Grails' support for Prototype. To get started you need to add this line to the `<head>` tag of your page:

```
<g:javascript library="prototype" />
```

This uses the javascript tag to automatically place the correct references in place for Prototype. If you require Scriptaculous too you can do the following instead:

```
<g:javascript library="scriptaculous" />
```

This works because of Grails' support for adaptive tag libraries. Thanks to Grails' plugin system there is support for a number of different Ajax libraries including (but not limited to):

- prototype
- dojo
- yui
- mootools

6.7.1.1 Remoting Linking

Remote content can be loaded in a number of ways, the most commons way is through the remoteLink tag. This tag allows the creation of HTML anchor tags that perform an asynchronous request and optionally set the response in an element. The simplest way to create a remote link is as follows:

```
<g:remoteLink action="delete" id="1">Delete Book</g:remoteLink>
```

The above link sends an asynchronous request to the `delete` action of the current controller with an id of `1`.

6.7.1.2 Updating Content

This is great, but usually you would want to provide some kind of feedback to the user as to what has happened:

```
def delete = {
     def b = Book.get( params.id )
     b.delete()
     render "Book ${b.id} was deleted"
}
```

GSP code:

```
<div id="message"></div>
<g:remoteLink action="delete" id="1" update="message">Delete Book</g:remoteLink>
```

The above example will call the action and set the contents of the `message div` to the response in this case `"Book 1 was deleted"`. This is done by the `update` attribute on the tag, which can also take a map to indicate what should be updated on failure:

```
<div id="message"></div>
<div id="error"></div>
<g:remoteLink action="delete" id="1"
              update="[success:'message',failure:'error']">Delete Book</g:remoteLink>
```

Here the `error` div will be updated if the request failed.

6.7.1.3 Remote Form Submission

An HTML form can also be submitted asynchronously in one of two ways. Firstly using the [formRemote](#) tag which expects similar attributes to those for the [remoteLink](#) tag:

```
<g:formRemote url="[controller:'book',action:'delete']"
              update="[success:'message',failure:'error']">
    <input type="hidden" name="id" value="1" />
    <input type="submit" value="Delete Book!" />
</g:formRemote >
```

Or alternatively you can use the [submitToRemote](#) tag to create a submit button. This allows some buttons to submit remotely and some not depending on the action:

```
<form action="delete">
    <input type="hidden" name="id" value="1" />
    <g:submitToRemote action="delete" update="[success:'message',failure:'error']" />
</form>
```

6.7.1.4 Ajax Events

Specific javascript can be called if certain events occur, all the events start with the "on" prefix and allow you to give feedback to the user where appropriate, or take other action:

```
<g:remoteLink action="show"
              id="1"
              update="success"
              onLoading="showProgress()"
              onComplete="hideProgress()">Show Book 1</g:remoteLink>
```

The above code will execute the "showProgress()" function which may show a progress bar or whatever is appropriate. Other events include:

- `onSuccess` - The javascript function to call if successful
- `onFailure` - The javascript function to call if the call failed
- `on_ERROR_CODE` - The javascript function to call to handle specified error codes (eg on404="alert('not found!')")
- `onUninitialized` - The javascript function to call the a ajax engine failed to initialise
- `onLoading` - The javascript function to call when the remote function is loading the response
- `onLoaded` - The javascript function to call when the remote function is completed loading the response
- `onComplete` - The javascript function to call when the remote function is complete, including any updates

If you need a reference to the `XmlHttpRequest` object you can use the implicit event parameter `e` to obtain it:

```
<g:javascript>
    function fireMe(e) {
        alert("XmlHttpRequest = " + e)
    }
}
</g:javascript>
<g:remoteLink action="example"
              update="success"
              onSuccess="fireMe(e)">Ajax Link</g:remoteLink>
```

6.7.2 Ajax with Dojo

Grails features an external plug-in to add [Dojo](#) support to Grails. To install the plug-in type the following command from the root of your project in a terminal window:

```
grails install-plugin dojo
```

This will download the current supported version of Dojo and install it into your Grails project. With that done you can add the following reference to the top of your page:

```
<g:javascript library="dojo" />
```

Now all of Grails tags such as [remoteLink](#), [formRemote](#) and [submitToRemote](#) work with Dojo remoting.

6.7.3 Ajax with GWT

Grails also features support for the [Google Web Toolkit](#) through a plug-in comprehensive [documentation](#) for can be found on the Grails wiki.

6.7.4 Ajax on the Server

Although Ajax features the X for XML there are a number of different ways to implement Ajax which are typically broken down into:

- Content Centric Ajax - Where you merely use the HTML result of a remote call to update the page
- Data Centric Ajax - Where you actually send an XML or JSON response from the server and programmatically update the page
- Script Centric Ajax - Where the server sends down a stream of Javascript to be evaluated on the fly

Most of the examples in the [Ajax](#) section cover Content Centric Ajax where you are updating the page, but you may also want to use Data Centric or Script Centric. This guide covers the different styles of Ajax.

Content Centric Ajax
Just to re-cap, content centric Ajax involves sending some HTML back from the server and is typically done by rendering a template with the [render](#) method:

```
def showBook = {
    def b = Book.get(params.id)
    render(template:"bookTemplate", model:[book:b])
}
```

Calling this on the client involves using the [remoteLink](#) tag:

```
<g:remoteLink action="showBook" id="${book.id}"
              update="book${book.id}">Update Book</g:remoteLink>
<div id="book${book.id}">
    <!--existing book mark-up -->
</div>
```

Data Centric Ajax with JSON
Data Centric Ajax typically involves evaluating the response on the client and updating programmatically. For a
JSON response with Grails you would typically use Grails' JSON marshaling capability:

```
import grails.converters.*
def showBook = {
    def b = Book.get(params.id)
    render b as JSON
}
```

And then on the client parse the incoming JSON request using an Ajax event handler:

```
<g:javascript>
function updateBook(e) {
    var book = eval("("+e.responseText+")") // evaluate the JSON
    $("book" + book.id + "_title").innerHTML = book.title
}
<g:javascript>
<g:remoteLink action="test" update="foo" onSuccess="updateBook(e)">Update Book</g:remoteLink
<g:set var="bookId">book${book.id}</g:set>
<div id="${bookId}">
    <div id="${bookId}_title">The Stand</div>
</div>
```

Data Centric Ajax with XML
On the server side using XML is equally trivial:

```
import grails.converters.*
def showBook = {
    def b = Book.get(params.id)
    render b as XML
}
```

However, since DOM is involved the client gets more complicated:

```
<g:javascript>
function updateBook(e) {
    var xml = e.responseXML
    var id = xml.getElementsByTagName("book").getAttribute("id")
    $("book" + id + "_title") = xml.getElementsByTagName("title")[0].textContent
}
<g:javascript>
<g:remoteLink action="test" update="foo" onSuccess="updateBook(e)">Update Book</g:remoteLink
<g:set var="bookId">book${book.id}</g:set>
<div id="${bookId}">
    <div id="${bookId}_title">The Stand</div>
</div>
```

Script Centric Ajax with JavaScript
Script centric Ajax involves actually sending Javascript back that gets evaluated on the client. An example of this can
be seen below:

138

```
def showBook = {
    def b = Book.get(params.id)
    response.contentType = "text/javascript"
    String title = b.title.encodeAsJavascript()
    render "$('book${b.id}_title')='${title}'"
}
```

The important thing to remember is to set the `contentType` to `text/javascript`. If you are using Prototype on the client the returned Javascript will automatically be evaluated due to this `contentType` setting. Obviously in this case it is critical that you have an agreed client-side API as you don't want changes on the client breaking the server. This is one of the reasons Rails has something like RJS. Although Grails does not currently have a feature such as RJS there is a [Dynamic JavaScript Plug-in](#) that offers similar capabilities.

6.8 Content Negotiation

Grails has built in support for [Content negotiation](#) using either the HTTP `Accept` header, an explicit format request parameter or the extension of a mapped URI.

Configuring Mime Types

Before you can start dealing with content negotiation you need to tell Grails what content types you wish to support. By default Grails comes configured with a number of different content types within `grails-app/conf/Config.groovy` using the `grails.mime.types` setting:

```
grails.mime.types = [ xml: ['text/xml', 'application/xml'],
                      text: 'text-plain',
                      js: 'text/javascript',
                      rss: 'application/rss+xml',
                      atom: 'application/atom+xml',
                      css: 'text/css',
                      cvs: 'text/csv',
                      all: '*/*',
                      json: 'text/json',
                      html: ['text/html','application/xhtml+xml']
                    ]
```

The above bit of configuration allows Grails to detect to format of a request containing either the 'text/xml' or 'application/xml' media types as simply 'xml'. You can add your own types by simply adding new entries into the map.

Content Negotiation using the Accept header

Every incoming HTTP request has a special [Accept](#) header that defines what media types (or mime types) a client can "accept". In older browsers this is typically:

```
*/*
```

Which simply means anything. However, on newer browser something all together more useful is sent such as (an example of a Firefox `Accept` header):

```
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/
```

Grails parses this incoming format and adds a `property` to the [request](#) object that outlines the preferred request format. For the above example the following assertion would pass:

```
assert 'html' == request.format
```

Why? The `text/html` media type has the highest "quality" rating of 0.9, therefore is the highest priority. If you have an older browser as mentioned previously the result is slightly different:

```
assert 'all' == request.format
```

In this case 'all' possible formats are accepted by the client. To deal with different kinds of requests from [Controllers](#) you can use the [withFormat](#) method that acts as kind of a switch statement:

```
import grails.converters.*
class BookController {
    def books
    def list = {
        this.books = Book.list()
        withFormat {
            html bookList:books
            js { render "alert('hello')" }
            xml { render books as XML }
        }
    }
}
```

What happens here is that if the preferred format is `html` then Grails will execute the `html()` call only. What this is does is make Grails look for a view called either `grails-app/views/books/list.html.gsp` or `grails-app/views/books/list.gsp`. If the format is `xml` then the closure will be invoked and an XML response rendered.
How do we handle the "all" format? Simply order the content-types within your `withFormat` block so that whichever one you want executed comes first. So in the above example, "all" will trigger the `html` handler.

> ⚠️ When using [withFormat](#) make sure it is the last call in your controller action as the return value of the `withFormat` method is used by the action to dictate what happens next.

Content Negotiation with the format Request Parameter
If fiddling with request headers if not your favorite activity you can override the format used by specifying a `format` request parameter:

```
/book/list?format=xml
```

You can also define this parameter in the [URL Mappings](#) definition:

```
"/book/list"(controller:"book", action:"list") {
    format = "xml"
}
```

Content Negotiation with URI Extensions
Grails also supports content negotiation via URI extensions. For example given the following URI:

```
/book/list.xml
```

Grails will shave off the extension and map it to `/book/list` instead whilst simultaneously setting the content format to `xml` based on this extension. This behaviour is enabled by default, so if you wish to turn it off, you must set the `grails.mime.file.extensions` property in `grails-app/conf/Config.groovy` to `false`:

```
grails.mime.file.extensions = false
```

Testing Content Negotiation
To test content negotiation in an integration test (see the section on Testing) you can either manipulate the incoming request headers:

```
void testJavascriptOutput() {
    def controller = new TestController()
    controller.request.addHeader "Accept",
                "text/javascript, text/html, application/xml, text/xml, */*"
    controller.testAction()
    assertEquals "alert('hello')", controller.response.contentAsString
}
```

Or you can set the format parameter to achieve a similar effect:

```
void testJavascriptOutput() {
    def controller = new TestController()
    controller.params.format = 'js'
    controller.testAction()
    assertEquals "alert('hello')", controller.response.contentAsString
}
```

7. Validation

Grails validation capability is built on [Spring's Validator API](#) and data binding capabilities. However Grails takes this further and provides a unified way to define validation "constraints" with its constraints mechanism. Constraints in Grails are a way to declaratively specify validation rules. Most commonly they are applied to [domain classes](#), however [URL Mappings](#) and [Command Objects](#) also support constraints.

7.1 Declaring Constraints

Within a domain class a [constraints](#) are defined with the constraints property that is assigned a code block:

```
class User {
    String login
    String password
    String email
    Integer age
    static constraints = {
        …
    }
}
```

You then use method calls that match the property name for which the constraint applies in combination with named parameters to specify constraints:

```
class User {
    ...
    static constraints = {
        login(size:5..15, blank:false, unique:true)
        password(size:5..15, blank:false)
        email(email:true, blank:false)
        age(min:18, nullable:false)
    }
}
```

In this example we've declared that the `login` property must be between 5 and 15 characters long, it cannot be blank and must be unique. We've all applied other constraints to the `password`, `email` and `age` properties.

> ⚠️ A complete reference for the available constraints can be found on the reference guide

7.2 Validating Constraints

Validation Basics
To validate a domain class you can call the [validate](#) method on any instance:

```
def user =  new User(params)
if(user.validate()) {
    // do something with user
}
else {
    user.errors.allErrors.each {
        println it
    }
}
```

The `errors` property on domain classes is an instance of the Spring [Errors](#) interface. The `Errors` interface provides methods to navigate the validation errors and also retrieve the original values.

Validation Phases

Within Grails there are essentially 2 phases of validation, the first phase is [data binding](#) which occurs when you bind request parameters onto an instance such as:

```
def user = new User(params)
```

At this point you may already have errors in the `errors` property due to type conversion (such as converting Strings to Dates). You can check these and obtain the original input value using the `Errors` API:

```
if(user.hasErrors()) {
    if(user.errors.hasFieldErrors("login")) {
        println user.errors.getFieldError("login").rejectedValue
    }
}
```

The second phase of validation happens when you call [validate](#) or [save](#). This is when Grails will validate the bound values agaist the [constraints](#) you defined. For example, by default the persistent [save](#) method calls `validate` before executing hence allowing you to write code like:

```
if(user.save()) {
    return user
}
else {
    user.errors.allErrors.each {
        println it
    }
}
```

7.3 Validation on the Client

Displaying Errors
Typically if you get a validation error you want to redirect back to the view for rendering. Once there you need some way of rendering errors. Grails supports a rich set of tags for dealing with errors. If you simply want to render the errors as a list you can use [renderErrors](#):

```
<g:renderErrors bean="${user}" />
```

If you need more control you can use [hasErrors](#) and [eachError](#):

```
<g:hasErrors bean="${user}">
  <ul>
   <g:eachError var="err" bean="${user}">
       <li>${err}</li>
   </g:eachError>
  </ul>
</g:hasErrors>
```

Highlighting Errors
It is often useful to highlight using a red box or some indicator when a field has been incorrectly input. This can also be done with the [hasErrors](#) by invoking it as a method. For example:

```
<div class='value ${hasErrors(bean:user,field:'login','errors')}'>
   <input type="text" name="login" value="${fieldValue(bean:user,field:'login')}"/>
</div>
```

What this code does is check if the `login` field of the `user` bean has any errors and if it does adds an `errors` CSS class to the `div` thus allowing you to use CSS rules to highlight the `div`.

Retrieving Input Values
Each error is actually an instance of the [FieldError](#) class in Spring, which retains the original input value within it. This is useful as you can use the error object to restore the value input by the user using the [fieldValue](#) tag:

```
<input type="text" name="login" value="${fieldValue(bean:user,field:'login')}"/>
```

This code will look if there is an existing `FieldError` in the `User` bean and if there is obtain the originally input value for the `login` field.

7.4 Validation and Internationalization

Another important thing to note about errors in Grails is that the messages that the errors display are not hard coded anywhere. The [FieldError](#) class in Spring essentially resolves messages from message bundles using Grails' [i18n](#) support.

Constraints and Message Codes
The codes themselves are dictated by a convention. For example consider the constraints we looked at earlier:

```
package com.mycompany.myapp
class User {
    ...
    static constraints = {
        login(size:5..15, blank:false, unique:true)
        password(size:5..15, blank:false)
        email(email:true, blank:false)
        age(min:18, nullable:false)
    }
}
```

If the `blank` constraint was violated Grails will, by convention, look for a message code in the form:

```
[Class Name].[Property Name].[Constraint Code]
```

In the case of the `blank` constraint this would be `user.login.blank` so you would need a message such as the following in your `grails-app/i18n/messages.properties` file:

```
user.login.blank=Your login name must be specified!
```

The class name is looked for both with and without a package, with the packaged version taking precedence. So for example, com.mycompany.myapp.User.login.blank will be used before user.login.blank. This allows for cases where you domain class encounters message code clashes with plugins.
For a reference on what codes are for which constraints refer to the reference guide for each constraint.

Displaying Messages
The [renderErrors](#) tag will automatically deal with looking up messages for you using the [message](#) tag. However, if you need more control of rendering you will need to do this yourself:

```
<g:hasErrors bean="${user}">
  <ul>
   <g:eachError var="err" bean="${user}">
       <li><g:message error="${err}" /></li>
   </g:eachError>
  </ul>
</g:hasErrors>
```

In this example within the body of the eachError tag we use the message tag in combination with its error
argument to read the message for the given error.

7.5 Validation Non Domain and Command Object Classes

Domain classes and command objects support validation by default. Other classes may be made validateable by
defining the static constraints property in the class (as described above) and then telling the framework about them. It
is important that the application register the validateable classes with the framework. Simply defining the constraints
property is not sufficient.

The Validateable Annotation
Classes which define the static constraints property and are marked with the @Validateable annotation may be made
validateable by the framework. Consider this example:

```
// src/groovy/com/mycompany/myapp/User.groovy
package com.mycompany.myapp
import org.codehaus.groovy.grails.validation.Validateable
@Validateable
class User {
    ...
    static constraints = {
        login(size:5..15, blank:false, unique:true)
        password(size:5..15, blank:false)
        email(email:true, blank:false)
        age(min:18, nullable:false)
    }
}
```

You need to tell the framework which packages to search for @Validateable classes by assigning a list of Strings to
the grails.validateable.packages property in Config.groovy.

```
// grails-app/conf/Config.groovy
...
grails.validateable.packages = ['com.mycompany.dto', 'com.mycompany.util']
...
```

The framework will only search those packages (and child packages of those) for classes marked with
@Validateable.

Registering Validateable Classes
If a class is not marked with @Validateable, it may still be made validateable by the framework. The steps required
to do this are to define the static constraints property in the class (as described above) and then telling the framework
about the class by assigning a value to the grails.validateable.classes property in Config.groovy.

```
// grails-app/conf/Config.groovy
...
grails.validateable.classes = [com.mycompany.myapp.User, com.mycompany.dto.Account]
...
```

8. The Service Layer

As well as the [Web layer](#), Grails defines the notion of a service layer. The Grails team discourages the embedding of core application logic inside controllers, as it does not promote re-use and a clean separation of concerns. Services in Grails are seen as the place to put the majority of the logic in your application, leaving controllers responsible for handling request flow via redirects and so on.

Creating a Service
You can create a Grails service by running the [create-service](#) command from the root of your project in a terminal window:

```
grails create-service simple
```

The above example will create a service at the location `grails-app/services/SimpleService.groovy`. A service's name ends with the convention `Service`, other than that a service is a plain Groovy class:

```
class SimpleService {
}
```

8.1 Declarative Transactions

Default Declarative Transactions
Services are typically involved with co-ordinating logic between [domain classes](#), and hence often involved with persistence that spans large operations. Given the nature of services they frequently require transactional behaviour. You can of course use programmatic transactions with the [withTransaction](#) method, however this is repetitive and doesn't fully leverage the power of Spring's underlying transaction abstraction.
Services allow the enablement of transaction demarcation, which is essentially a declarative way of saying all methods within this service are to be made transactional. All services have transaction demarcation enabled by default - to disable it, simply set the `transactional` property to `false`:

```
class CountryService {
    static transactional = false
}
```

You may also set this property to `true` in case the default changes in the future, or simply to make it clear that the service is intentionally transactional.

> ⛔ Warning: [dependency injection](#) is the **only** way that declarative transactions work. You will not get a transactional service if you use the `new` operator such as `new BookService()`

The result is all methods are wrapped in a transaction and automatic rollback occurs if an exception is thrown in the body of one of the methods. The propagation level of the transaction is by default set to [PROPAGATION_REQUIRED](#).

Custom Transaction Configuration
Grails also fully supports Spring's `Transactional` annotation for cases where you need more fine-grained control over transactions at a per-method level or need specify an alternative propagation level:

```
import org.springframework.transaction.annotation.*
class BookService {
    @Transactional(readOnly = true)
    def listBooks() { Book.list() }
    @Transactional def updateBook() {
        // …
    }
}
```

For more information refer to the section of the Spring user guide on Using @Transactional.

> ⚠ Unlike Spring you do not need any prior configuration to use Transactional, just specify the annotation as needed and Grails will pick them up automatically.

8.2 Scoped Services

By default, access to service methods is not synchronised, so nothing prevents concurrent execution of those functions. In fact, because the service is a singleton and may be used concurrently, you should be very careful about storing state in a service. Or take the easy (and better) road and never store state in a service.
You can change this behaviour by placing a service in a particular scope. The supported scopes are:

- prototype - A new service is created every time it is injected into another class
- request - A new service will be created per request
- flash - A new service will be created for the current and next request only
- flow - In web flows the service will exist for the scope of the flow
- conversation - In web flows the service will exist for the scope of the conversation. ie a root flow and its sub flows
- session - A service is created for the scope of a user session
- singleton (default) - Only one instance of the service ever exists

> ⚠ If your service is flash, flow or conversation scoped it will need to implement java.io.Serializable and can only be used in the context of a Web Flow

To enable one of the scopes, add a static scope property to your class whose value is one of the above:

```
static scope = "flow"
```

8.3 Dependency Injection and Services

Dependency Injection Basics
A key aspect of Grails services is the ability to take advantage of the Spring Framework's dependency injection capability. Grails supports "dependency injection by convention". In other words, you can use the property name representation of the class name of a service, to automatically inject them into controllers, tag libraries, and so on.
As an example, given a service called BookService, if you place a property called bookService within a controller as follows:

```
class BookController {
   def bookService
   …
}
```

In this case, the Spring container will automatically inject an instance of that service based on its configured scope.
All dependency injection is done by name. You can also specify the type as follows:

```
class AuthorService {
    BookService bookService
}
```

Dependency Injection and Services

You can inject services in other services with the same technique. Say you had an `AuthorService` that needed to use the `BookService`, declaring the `AuthorService` as follows would allow that:

```
class AuthorService {
    def bookService
}
```

Dependency Injection and Domain Classes

You can even inject services into domain classes, which can aid in the development of rich domain models:

```
class Book {
    …
    def bookService
    def buyBook() {
        bookService.buyBook(this)
    }
}
```

8.4 Using Services from Java

One of the powerful things about services is that since they encapsulate re-usable logic, you can use them from other classes, including Java classes. There are a couple of ways you can re-use a service from Java. The simplest way is to move your service into a package within the `grails-app/services` directory. The reason this is a critical step is that it is not possible to import classes into Java from the default package (the package used when no package declaration is present). So for example the `BookService` below cannot be used from Java as it stands:

```
class BookService {
    void buyBook(Book book) {
        // logic
    }
}
```

However, this can be rectified by placing this class in a package, by moving the class into a sub directory such as `grails-app/services/bookstore` and then modifying the package declaration:

```
package bookstore
class BookService {
    void buyBook(Book book) {
        // logic
    }
}
```

An alternative to packages is to instead have an interface within a package that the service implements:

```
package bookstore;
interface BookStore {
    void buyBook(Book book);
}
```

And then the service:

```
class BookService implements bookstore.BookStore {
    void buyBook(Book b) {
        // logic
    }
}
```

This latter technique is arguably cleaner, as the Java side only has a reference to the interface and not to the implementation class. Either way, the goal of this exercise to enable Java to statically resolve the class (or interface) to use, at compile time. Now that this is done you can create a Java class within the src/java package, and provide a setter that uses the type and the name of the bean in Spring:

```
package bookstore;
// note: this is Java class
public class BookConsumer {
    private BookStore store;
    public void setBookStore(BookStore storeInstance) {
        this.store = storeInstance;
    }
    …
}
```

Once this is done you can configure the Java class as a Spring bean in grails-app/conf/spring/resources.xml (For more information one this see the section on Grails and Spring):

```
<bean id="bookConsumer" class="bookstore.BookConsumer">
    <property name="bookStore" ref="bookService" />
</bean>
```

9. Testing

Automated testing is seen as a key part of Grails, implemented using [Groovy Tests](). Hence, Grails provides many ways to making testing easier from low level unit testing to high level functional tests. This section details the different capabilities that Grails offers in terms of testing.

The first thing to be aware of is that all of the `create-*` commands actually end up creating `integration` tests automatically for you. For example say you run the [create-controller]() command as follows:

```
grails create-controller simple
```

Not only will Grails create a controller at `grails-app/controllers/SimpleController.groovy`, but also an integration test at `test/integration/SimpleControllerTests.groovy`. What Grails won't do however is populate the logic inside the test! That is left up to you.

> ⚠️ As of Grails 1.2.2,the suffix of `Test` is also supported for test cases.

Running Tests

Test are run with the [test-app]() command:

```
grails test-app
```

The above command will produce output such as:

```
-------------------------------------------------------
Running Unit Tests…
Running test FooTests...FAILURE
Unit Tests Completed in 464ms …
-------------------------------------------------------
Tests failed: 0 errors, 1 failures
```

Whilst reports will have been written out the `target/test-reports` directory.

> ⚠️ You can force a clean before running tests by passing `-clean` to the `test-app` command.

Targeting Tests

You can selectively target the test(s) to be run in different ways. To run all tests for a controller named `SimpleController` you would run:

```
grails test-app SimpleController
```

This will run any tests for the class named `SimpleController`. Wildcards can be used...

```
grails test-app *Controller
```

This will test all classes ending in `Controller`. Package names can optionally be specified...

```
grails test-app some.org.*Controller
```

or to run all tests in a package...

```
grails test-app some.org.*
```

or to run all tests in a package including subpackages...

```
grails test-app some.org.**
```

You can also target particular test methods...

```
grails test-app SimpleController.testLogin
```

This will run the `testLogin` test in the `SimpleController` tests. You can specify as many patterns in combination as you like...

```
grails test-app some.org.* SimpleController.testLogin BookController
```

Targeting Test Types and/or Phases

In addition to targeting certain tests, you can also target test *types* and/or *phases* by using the `phase:type` syntax.

> ⚠️ Grails organises tests by phase and by type. A test phase relates to the state of the Grails application during the tests, and the type relates to the testing mechanism.
> Grails comes with support for 4 test phases (`unit`, `integration`, `functional` and `other`) and JUnit test types for the `unit` and `integration` phases. These test types have the same name as the phase.
> Testing plugins may provide new test phases or new test types for existing phases. Refer to the plugin documentation.

To execute the JUnit `integration` tests you can run:

```
grails test-app integration:integration
```

Both `phase` and `type` are optional. Their absence acts as a wildcard. The following command will run all test types in the `unit` phase:

```
grails test-app unit:
```

The Grails [Spock Plugin](#) is one plugin that adds new test types to Grails. It adds a `spock` test type to the `unit`, `integration` and `functional` phases. To run all spock tests in all phases you would run the following:

151

```
grails test-app :spock
```

To run the all of the spock tests in the `functional` phase you would run...

```
grails test-app functional:spock
```

More than one pattern can be specified...

```
grails test-app unit:spock integration:spock
```

Targeting Tests in Types and/or Phases

Test and type/phase targetting can be applied at the same time:

```
grails test-app integration: unit: some.org.**
```

This would run all tests in the `integration` and `unit` phases that are in the page `some.org` or a subpackage of.

9.1 Unit Testing

Unit testing are tests at the "unit" level. In other words you are testing individual methods or blocks of code without considering for surrounding infrastructure. In Grails you need to be particularity aware of the difference between unit and integration tests because in unit tests Grails **does not** inject any of the dynamic methods present during integration tests and at runtime.
This makes sense if you consider that the methods injected by Grails typically communicate with the database (with GORM) or the underlying Servlet engine (with Controllers). For example say you have service like the following in `BookController`:

```
class MyService {
    def otherService
    String createSomething() {
        def stringId = otherService.newIdentifier()
        def item = new Item(code: stringId, name: "Bangle")
        item.save()
        return stringId
    }
    int countItems(String name) {
        def items = Item.findAllByName(name)
        return items.size()
    }
}
```

As you can see the service takes advantage of GORM methods. So how do you go about testing the above code in a unit test? The answer can be found in Grails' testing support classes.

The Testing Framework
The core of the testing plugin is the `grails.test.GrailsUnitTestCase` class. This is a sub-class of `GroovyTestCase` geared towards Grails applications and their artifacts. It provides several methods for mocking particular types as well as support for general mocking a la Groovy's MockFor and StubFor classes.
Normally you might look at the `MyService` example shown previously and the dependency on another service and the use of dynamic domain class methods with a bit of a groan. You can use meta-class programming and the "map as object" idiom, but these can quickly get ugly. How might we write the test with GrailsUnitTestCase ?

```
import grails.test.GrailsUnitTestCase
class MyServiceTests extends GrailsUnitTestCase {
    void testCreateSomething() {
        // Mock the domain class.
        mockDomain(Item)
        // Mock the "other" service.
        String testId = "NH-12347686"
        def otherControl = mockFor(OtherService)
        otherControl.demand.newIdentifier(1..1) {-> return testId }
        //      Initialise the service and test the target method.
        def testService = new MyService()
        testService.otherService = otherControl.createMock()
        def retval = testService.createSomething()
        // Check that the method returns the identifier returned by the
        // mock "other" service and also that a new Item instance has
        // been saved.
        def testInstances = Item.list()
        assertEquals testId, retval
        assertEquals 1, testInstances.size()
        assertTrue testInstances[0] instanceof Item
    }
    void testCountItems() {
        // Mock the domain class, this time providing a list of test
        // Item instances that can be searched.
        def testInstances = [ new Item(code: "NH-4273997", name: "Laptop"),
                              new Item(code: "EC-4395734", name: "Lamp"),
                              new Item(code: "TF-4927324", name: "Laptop") ]
        mockDomain(Item, testInstances)
        // Initialise the service and test the target method. def testService = new MyServic
        assertEquals 2, testService.countItems("Laptop")
        assertEquals 1, testService.countItems("Lamp")
        assertEquals 0, testService.countItems("Chair")
    }
}
```

OK, so a fair bit of new stuff there, but once we break it down you should quickly see how easy it is to use the methods available to you. Take a look at the "testCreateSomething()" test method. The first thing you will probably notice is the mockDomain() method, which is one of several provided by GrailsUnitTestCase:

```
def testInstances = []
mockDomain(Item, testInstances)
```

It adds all the common domain methods (both instance and static) to the given class so that any code using it sees it as a full-blown domain class. So for example, once the Item class has been mocked, we can safely call the save() method on instances of it. Invoking the save() method doesn't really save the instance to any database but it will cache the object in the testing framework so the instance will be visible to certain queries. The following code snippet demonstrates the effect of calling the save() method.

```
void testSomething() {
    def testInstances=[]
    mockDomain(Song, testInstances)
    assertEquals(0, Song.count())
    new Song(name:"Supper's Ready").save()
    assertEquals(1, Song.count())
}
```

The next bit we want to look at is centered on the mockFor method:

```
def otherControl = mockFor(OtherService)
otherControl.demand.newIdentifier(1..1) {-> return testId }
```

This is analagous to the MockFor and StubFor classes that come with Groovy and it can be used to mock any class you want. In fact, the "demand" syntax is identical to that used by Mock/StubFor, so you should feel right at home. Of course you often need to inject a mock instance as a dependency, but that is pretty straight forward with the

`createMock()` method, which you simply call on the mock control as shown. For those familiar with EasyMock, the name `otherControl` highlights the role of the object returned by `mockFor()` - it is a control object rather than the mock itself.

The rest of the `testCreateSomething()` method should be pretty familiar, particularly as you now know that the mock `save()` method adds instances to `testInstances` list. However, there is an important technique missing from the test method. We can determine that the mock `newIdentifier()` method is called because its return value has a direct impact on the result of the `createSomething()` method. But what if that weren't the case? How would we know whether it had been called or not? With Mock/StubFor the check would be performed at the end of the `use()` closure, but that's not available here. Instead, you can call `verify()` on the control object - in this case `otherControl`. This will perform the check and throw an assertion error if it hasn't been called when it should have been.

Lastly, `testCountItems()` in the example demonstrates another facet of the `mockDomain()` method:

```
def testInstances = [ new Item(code: "NH-4273997", name: "Laptop"),
                      new Item(code: "EC-4395734", name: "Lamp"),
                      new Item(code: "TF-4927324", name: "Laptop") ]
mockDomain(Item, testInstances)
```

It is normally quite fiddly to mock the dynamic finders manually, and you often have to set up different data sets for each invocation. On top of that, if you decide a different finder should be used then you have to update the tests to check for the new method! Thankfully the `mockDomain()` method provides a lightweight implementation of the dynamic finders backed by a list of domain instances. Simply provide the test data as the second argument of the method and the mock finders will just work.

GrailsUnitTestCase - the mock methods
You have already seen a couple of examples in the introduction of the `mock..()` methods provided by the `GrailsUnitTestCase` class. Here we will look at all the available methods in some detail, starting with the all-purpose `mockFor()`. But before we do, there is a very important point to make: using these methods ensures that any changes you make to the given classes do not leak into other tests! This is a common and serious problem when you try to perform the mocking yourself via meta-class programming, but that headache just disappears as long as you use at least one of `mock..()` methods on each class you want to mock.

```
mockFor(class, loose = false)
```

General purpose mocking that allows you to set up either strict or loose demands on a class.
This method is surprisingly intuitive to use. By default it will create a strict mock control object (one for which the order in which methods are called is important) that you can use to specify demands:

```
def strictControl = mockFor(MyService)
strictControl.demand.someMethod(0..2) { String arg1, int arg2 -> ‚Ä¶ }
strictControl.demand.static.aStaticMethod {-> ‚Ä¶ }
```

Notice that you can mock static methods as well as instance ones simply by using the "static" property after "demand". You then specify the name of the method that you want to mock with an optional range as its argument. This range determines how many times you expect the method to be called, so if the number of invocations falls outside of that range (either too few or too many) then an assertion error will be thrown. If no range is specified, a default of "1..1" is assumed, i.e. that the method must be called exactly once.
The last part of a demand is a closure representing the implementation of the mock method. The closure arguments should match the number and types of the mocked method, but otherwise you are free to add whatever you want in the body.
As we mentioned before, if you want an actual mock instance of the class that you are mocking, then you need to call `mockControl.createMock()`. In fact, you can call this as many times as you like to create as many mock instances as you need. And once you have executed the test method, you can call `mockControl.verify()` to check whether the expected methods were actually called or not.
Lastly, the call:

```
def looseControl = mockFor(MyService, true)
```

will create a mock control object that has only loose expectations, i.e. the order that methods are invoked does not matter.

mockDomain(class, testInstances =)
Takes a class and makes mock implementations of all the domain class methods (both instance- and static-level) accessible on it.

Mocking domain classes is one of the big wins from using the testing plugin. Manually doing it is fiddly at best, so it's great that mockDomain() takes that burden off your shoulders.

In effect, mockDomain() provides a lightweight version of domain classes in which the "database" is simply a list of domain instances held in memory. All the mocked methods (save(), get(), findBy*(), etc.) work against that list, generally behaving as you would expect them to. In addition to that, both the mocked save() and validate() methods will perform real validation (support for the unique constraint included!) and populate an errors object on the corresponding domain instance.

There isn't much else to say other than that the plugin does not support the mocking of criteria or HQL queries. If you use either of those, simply mock the corresponding methods manually (for example with mockFor()) or use an integration test with real data.

mockForConstraintsTests(class, testInstances =)
Highly specialised mocking for domain classes and command objects that allows you to check whether the constraints are behaving as you expect them to.

Do you test your domain constraints? If not, why not? If your answer is that they don't need testing, think again. Your constraints contain logic and that logic is highly susceptible to bugs - the kind of bugs that can be tricky to track down (particularly as save() doesn't throw an exception when it fails). If your answer is that it's too hard or fiddly, that is no longer an excuse. Enter the mockForConstraintsTests() method.

This is like a much reduced version of the mockDomain() method that simply adds a validate() method to a given domain class. All you have to do is mock the class, create an instance with field values, and then call validate(). You can then access the errors property on your domain instance to find out whether the validation failed or not. So if all we are doing is mocking the validate() method, why the optional list of test instances? That is so that we can test unique constraints as you will soon see.

So, suppose we have a simple domain class like so:

```
class Book {
    String title
    String author
    static constraints = {
        title(blank: false, unique: true)
        author(blank: false, minSize: 5)
    }
}
```

Don't worry about whether the constraints are sensible or not (they're not!), they are for demonstration only. To test these constraints we can do the following:

```
class BookTests extends GrailsUnitTestCase {
    void testConstraints() {
        def existingBook = new Book(title: "Misery", author: "Stephen King")
        mockForConstraintsTests(Book, [ existingBook ])
        // Validation should fail if both properties are null.
        def book = new Book()
        assertFalse book.validate()
        assertEquals "nullable", book.errors["title"]
        assertEquals "nullable", book.errors["author"]
        // So let's demonstrate the unique and minSize constraints.
        book = new Book(title: "Misery", author: "JK")
        assertFalse book.validate()
        assertEquals "unique",    book.errors["title"]
        assertEquals "minSize", book.errors["author"]
        // Validation should pass!
        book = new Book(title: "The Shining", author: "Stephen King")
        assertTrue book.validate()
    }
}
```

You can probably look at that code and work out what's happening without any further explanation. The one thing we will explain is the way the errors property is used. First, it does return a real Spring Errors instance, so you can access all the properties and methods you would normally expect. Second, this particular Errors object also has map/property access as shown. Simply specify the name of the field you are interested in and the map/property access will return the name of the constraint that was violated. Note that it is the constraint name , not the message code (as you might expect).

That's it for testing constraints. One final thing we would like to say is that testing the constraints in this way catches a common error: typos in the "constraints" property! It is currently one of the hardest bugs to track down normally, and yet a unit test for your constraints will highlight the problem straight away.

mockLogging(class, enableDebug = false)
Adds a mock "log" property to a class. Any messages passed to the mock logger are echoed to the console.

mockController(class)
Adds mock versions of the dynamic controller properties and methods to the given class. This is typically used in conjunction with the ControllerUnitTestCase class.

mockTagLib(class)
Adds mock versions of the dynamic taglib properties and methods to the given class. This is typically used in conjunction with the TagLibUnitTestCase class.

9.2 Integration Testing

Integration tests differ from unit tests in that you have full access to the Grails environment within the test. Grails will use an in-memory HSQLDB database for integration tests and clear out all the data from the database in between each test.

Testing Controllers
To test controllers you first have to understand the Spring Mock Library.
Essentially Grails automatically configures each test with a MockHttpServletRequest, MockHttpServletResponse, and MockHttpSession which you can then use to perform your tests. For example consider the following controller:

```
class FooController {
    def text = {
        render "bar"
    }
    def someRedirect = {
        redirect(action:"bar")
    }
}
```

The tests for this would be:

156

```
class FooControllerTests extends GroovyTestCase {
    void testText() {
        def fc = new FooController()
        fc.text()
        assertEquals "bar", fc.response.contentAsString
    }
    void testSomeRedirect() {
        def fc = new FooController()
        fc.someRedirect()
        assertEquals "/foo/bar", fc.response.redirectedUrl
    }
}
```

In the above case the response is an instance of `MockHttpServletResponse` which we can use to obtain the `contentAsString` (when writing to the response) or the URL redirected to for example. These mocked versions of the Servlet API are, unlike the real versions, all completely mutable and hence you can set properties on the request such as the `contextPath` and so on.

Grails **does not** invoke interceptors or servlet filters automatically when calling actions during integration testing. You should test interceptors and filters in isolation, and via functional testing if necessary.

Testing Controllers with Services

If your controller references a service (or other Spring beans), you have to explicitly initialise the service from your test.

Given a controller using a service:

```
class FilmStarsController {
    def popularityService
    def update = {
        // do something with popularityService
    }
}
```

The test for this would be:

```
class FilmStarsTests extends GroovyTestCase {
    def popularityService
    void testInjectedServiceInController () {
        def fsc = new FilmStarsController()
        fsc.popularityService = popularityService
        fsc.update()
    }
}
```

Testing Controller Command Objects

With command objects you just supply parameters to the request and it will automatically do the command object work for you when you call your action with no parameters:

Given a controller using a command object:

```
class AuthenticationController {
    def signup = { SignupForm form ->
        …
    }
}
```

You can then test it like this:

```
def controller = new AuthenticationController()
controller.params.login = "marcpalmer"
controller.params.password = "secret"
controller.params.passwordConfirm = "secret"
controller.signup()
```

Grails auto-magically sees your call to `signup()` as a call to the action and populates the command object from the mocked request parameters. During controller testing, the `params` are mutable with a mocked request supplied by Grails.

Testing Controllers and the render Method

The [render](#) method allows you to render a custom view at any point within the body of an action. For instance, consider the example below:

```
def save = {
    def book = Book(params)
    if(book.save()) {
        // handle
    }
    else {
        render(view:"create", model:[book:book])
    }
}
```

In the above example the result of the model of the action is not available as the return value, but instead is stored within the `modelAndView` property of the controller. The `modelAndView` property is an instance of Spring MVC's [ModelAndView](#) class and you can use it to the test the result of an action:

```
def bookController = new BookController()
bookController.save()
def model = bookController.modelAndView.model.book
```

Simulating Request Data

If you're testing an action that requires request data such as a REST web service you can use the Spring [MockHttpServletRequest](#) object to do so. For example consider this action which performs data binding from an incoming request:

```
def create = {
    [book: new Book(params['book']) ]
}
```

If you wish the simulate the 'book' parameter as an XML request you could do something like the following:

```
void testCreateWithXML() {
    def controller = new BookController()
    controller.request.contentType = 'text/xml'
    controller.request.content = '''<?xml version="1.0" encoding="ISO-8859-1"?>
    <book>
        <title>The Stand</title>
        …
    </book>
    '''.getBytes() // note we need the bytes
    def model = controller.create()
    assert model.book
    assertEquals "The Stand", model.book.title
}
```

The same can be achieved with a JSON request:

```
void testCreateWithJSON() {
    def controller = new BookController()
    controller.request.contentType = "text/json"
    controller.request.content = '{"id":1,"class":"Book","title":"The Stand"}'.getBytes()
    def model = controller.create()
    assert model.book
    assertEquals "The Stand", model.book.title
}
```

> ⚠ With JSON don't forget the `class` property to specify the name the target type to bind too.
> In the XML this is implicit within the name of the `<book>` node, but with JSON you need
> this property as part of the JSON packet.

For more information on the subject of REST web services see the section on REST.

Testing Web Flows

Testing Web Flows requires a special test harness called `grails.test.WebFlowTestCase` which sub classes
Spring Web Flow's AbstractFlowExecutionTests class.

> ⚠ Subclasses of `WebFlowTestCase` **must** be integration tests

For example given this trivial flow:

```
class ExampleController {
    def exampleFlow = {
        start {
            on("go") {
                flow.hello = "world"
            }.to "next"
        }
        next {
            on("back").to "start"
            on("go").to "end"
        }
        end()
    }
}
```

You need to tell the test harness what to use for the "flow definition". This is done via overriding the abstract
`getFlow` method:

```
class ExampleFlowTests extends grails.test.WebFlowTestCase {
    def getFlow() { new ExampleController().exampleFlow }
    …
}
```

If you need to specify the flow id you can do so by overriding the getFlowId method otherwise the default is `test`:

```
class ExampleFlowTests extends grails.test.WebFlowTestCase {
    String getFlowId() { "example" }
    …
}
```

Once this is done in your test you need to kick off the flow with the `startFlow` method which returns a
`ViewSelection` object:

159

```
void testExampleFlow() {
    def viewSelection = startFlow()
    assertEquals "start", viewSelection.viewName
    …
}
```

As demonstrated above you can check you're on the right state using the `viewName` property of the `ViewSelection` object. To trigger and event you need to use the `signalEvent` method:

```
void testExampleFlow() {
    …
    viewSelection = signalEvent("go")
    assertEquals "next", viewSelection.viewName
    assertEquals "world", viewSelection.model.hello
}
```

Here we have signaled to the flow to execute the event "go" this causes a transition to the "next" state. In the example a transition action placed a `hello` variable into the flow scope. We can test the value of this variable by inspecting the `model` property of the `ViewSelection` as above.

Testing Tag Libraries

Testing tag libraries is actually pretty trivial because when a tag is invoked as a method it returns its result as a string. So for example if you have a tag library like this:

```
class FooTagLib {
    def bar = { attrs, body ->
            out << "<p>Hello World!</p>"
    }
    def bodyTag = { attrs, body ->
        out << "<${attrs.name}>"
            out << body()
        out << "</${attrs.name}>"
    }
}
```

The tests would look like:

```
class FooTagLibTests extends GroovyTestCase {
    void testBarTag() {
        assertEquals "<p>Hello World!</p>", new FooTagLib().bar(null,null)
    }
    void testBodyTag() {
        assertEquals "<p>Hello World!</p>", new FooTagLib().bodyTag(name:"p") {
            "Hello World!"
        }
    }
}
```

Notice that for the second example, `testBodyTag`, we pass a block that returns the body of the tag. This is handy for representing the body as a String.

Testing Tag Libraries with GroovyPagesTestCase

In addition to doing simply testing of tag libraries like the above you can also use the `grails.test.GroovyPagesTestCase` class to test tag libraries.
The `GroovyPagesTestCase` class is a sub class of the regular `GroovyTestCase` class and provides utility methods for testing the output of a GSP rendering.

> ⚠   `GroovyPagesTestCase` can only be used in an integration test.

As an example given a date formatting tag library such as the one below:

```
class FormatTagLib {
    def dateFormat = { attrs, body ->
        out << new java.text.SimpleDateFormat(attrs.format) << attrs.date
    }
}
```

This can be easily tested as follows:

```
class FormatTagLibTests extends GroovyPagesTestCase {
    void testDateFormat() {
        def template = '<g:dateFormat format="dd-MM-yyyy" date="${myDate}" />'
        def testDate = … // create the date
        assertOutputEquals( '01-01-2008', template, [myDate:testDate] )
    }
}
```

You can also obtain the result of a GSP using the `applyTemplate` method of the `GroovyPagesTestCase` class:

```
class FormatTagLibTests extends GroovyPagesTestCase {
    void testDateFormat() {
        def template = '<g:dateFormat format="dd-MM-yyyy" date="${myDate}" />'
        def testDate = … // create the date
        def result = applyTemplate( template, [myDate:testDate] )
        assertEquals '01-01-2008', result
    }
}
```

Testing Domain Classes

Testing domain classes is typically a simple matter of using the [GORM API](#), however there are some things to be aware of. Firstly, if you are testing queries you will often need to "flush" in order to ensure the correct state has been persisted to the database. For example take the following example:

```
void testQuery() {
    def books = [ new Book(title:"The Stand"), new Book(title:"The Shining")]
    books*.save()
    assertEquals 2, Book.list().size()
}
```

This test will actually fail, because calling [save](#) does not actually persist the `Book` instances when called. Calling `save` merely indicates to Hibernate that at some point in the future these instances should be persisted. If you wish to commit changes immediately you need to "flush" them:

```
void testQuery() {
    def books = [ new Book(title:"The Stand"), new Book(title:"The Shining")]
    books*.save(flush:true)
    assertEquals 2, Book.list().size()
}
```

In this case since we're passing the argument `flush` with a value of `true` the updates will be persisted immediately and hence will be available to the query later on.

9.3 Functional Testing

Functional tests involve testing the actual running application and are often harder to automate. Grails does not ship

with any functional testing support out of the box, but has support for [Canoo WebTest](#) via a plug-in.
To get started install Web Test with the following commands:

```
grails install-plugin webtest
```

Then refer to the [reference on the wiki](#) which explains how to go about using Web Test and Grails.

10. Internationalization

Grails supports Internationalization (i18n) out of the box through the underlying Spring MVC support for internationalization. With Grails you are able to customize the text that appears in any view based on the users Locale. To quote the javadoc for the Locale class in Java:

> *A Locale object represents a specific geographical, political, or cultural region. An operation that requires a Locale to perform its task is called locale-sensitive and uses the Locale to tailor information for the user. For example, displaying a number is a locale-sensitive operation--the number should be formatted according to the customs/conventions of the user's native country, region, or culture.*

A Locale is made up of a language code and a country code. For example "en_US" is the code for US english, whilst "en_GB" is the for British English.

10.1 Understanding Message Bundles

Now that you have an idea of locales, to take advantage of them in Grails you have to create message bundles that contain the different languages that you wish to render. Message bundles in Grails are located inside the `grails-app/i18n` directory and are simple Java properties files.
Each bundle starts with the name `messages` by convention and ends with the locale. Grails ships with a bunch of built in message bundles for a whole range of languages within the `grails-app/i18n` directory. For example:

```
messages.properties
messages_de.properties
messages_es.properties
etc.
```

By default Grails will look in `messages.properties` for messages, unless the user has specified a custom locale. You can create your own message bundle by simply creating a new properties file that ends with the locale you are interested. For example `messages_en_GB.properties` for British English.

10.2 Changing Locales

By default the user locale is detected from the incoming `Accept-Language` header. However, you can provide users the capability to switch locales by simply passing a parameter called `lang` to Grails as a request parameter:

```
/book/list?lang=es
```

Grails will automatically switch the user locale and store it in a cookie so subsequent requests will have the new header.

10.3 Reading Messages

Reading Messages in the View
The most common place that you need messages is inside the view. To read messages from the view just use the message tag:

```
<g:message code="my.localized.content" />
```

As long as you have a key in your `messages.properties` (with appropriate locale suffix) such as the one below then Grails will look-up the message:

```
my.localized.content=Hola, Me llamo John. Hoy es domingo.
```

Note that sometimes you may need to pass arguments to the message. This is also possible with the `message` tag:

163

```
<g:message code="my.localized.content" args="${ ['Juan', 'lunes'] }" />
```

And then use positional parameters in the message:

```
my.localized.content=Hola, Me llamo {0}. Hoy es {1}.
```

Reading Messages in Controllers and Tag Libraries
Since you can invoke tags as methods from controllers it is also trivial to read messages within in a controller:

```
def show = {
    def msg = message(code:"my.localized.content", args:['Juan', 'lunes'])
}
```

The same technique can be used on tag libraries, but note if your tag library has a different namespace then you will need to g. prefix:

```
def myTag = { attrs, body ->
    def msg = g.message(code:"my.localized.content", args:['Juan', 'lunes'])
}
```

10.4 Scaffolding and i18n

Grails does not ship with i18n aware scaffolding templates to generate the controller and views. However, i18n aware templates are available via the i18n templates plugin. The templates are identical to the default scaffolding templates, except that they are i18n aware using the message tag for labels, buttons etc.
To get started install the i18n templates with the following command:

```
grails install-plugin i18n-templates
```

Then refer to the reference on the wiki which explains how to use the i18n templates.

11. Security

Grails is no more or less secure than Java Servlets. However, Java servlets (and hence Grails) are extremely secure and largely immune to common buffer overrun and malformed URL exploits due to the nature of the Java Virtual Machine underpinning the code.

Web security problems typically occur due to developer naivety or mistakes, and there is a little Grails can do to avoid common mistakes and make writing secure applications easier to write.

What Grails Automatically Does
Grails has a few built in safety mechanisms by default.

1. All standard database access via GORM domain objects is automatically SQL escaped to prevent SQL injection attacks
2. The default scaffolding templates HTML escape all data fields when displayed
3. Grails link creating tags (link, form, createLink, createLinkTo and others) all use appropriate escaping mechanisms to prevent code injection
4. Grails provides codecs to allow you to trivially escape data when rendered as HTML, JavaScript and URLs to prevent injection attacks here.

11.1 Securing Against Attacks

SQL injection
Hibernate, which is the technology underlying GORM domain classes, automatically escapes data when committing to database so this is not an issue. However it is still possible to write bad dynamic HQL code that uses unchecked request parameters. For example doing the following is vulnerable to HQL injection attacks:

```
def vulnerable = {
    def books = Book.find("from Book as b where b.title ='" + params.title + "'")
}
```

Do **not** do this. If you need to pass in parameters use named or positional parameters instead:

```
def safe = {
    def books = Book.find("from Book as b where b.title =?", [params.title])
}
```

Phishing
This really a public relations issue in terms of avoiding hijacking of your branding and a declared communication policy with your customers. Customers need to know how to identify bonafide emails received.

XSS - cross-site scripting injection
It is important that your application verifies as much as possible that incoming requests were originated from your application and not from another site. Ticketing and page flow systems can help this and Grails' support for Spring Web Flow includes security like this by default.

It is also important to ensure that all data values rendered into views are escaped correctly. For example when rendering to HTML or XHTML you must call encodeAsHTML on every object to ensure that people cannot maliciously inject JavaScript or other HTML into data or tags viewed by others. Grails supplies several Dynamic Encoding Methods for this purpose and if your output escaping format is not supported you can easily write your own codec.

You must also avoid the use of request parameters or data fields for determining the next URL to redirect the user to. If you use a `successURL` parameter for example to determine where to redirect a user to after a successful login, attackers can imitate your login procedure using your own site, and then redirect the user back to their own site once logged in, potentially allowing JS code to then exploit the logged-in account on the site.

HTML/URL injection
This is where bad data is supplied such that when it is later used to create a link in a page, clicking it will not cause the expected behaviour, and may redirect to another site or alter request parameters.

HTML/URL injection is easily handled with the codecs supplied by Grails, and the tag libraries supplied by Grails all use encodeAsURL where appropriate. If you create your own tags that generate URLs you will need to be mindful of

doing this too.

Denial of service

Load balancers and other appliances are more likely to be useful here, but there are also issues relating to excessive queries for example where a link is created by an attacker to set the maximum value of a result set so that a query could exceed the memory limits of the server or slow the system down. The solution here is to always sanitize request parameters before passing them to dynamic finders or other GORM query methods:

```
def safeMax = Math.max(params.max?.toInteger(), 100) // limit to 100 results
return Book.list(max:safeMax)
```

Guessable IDs

Many applications use the last part of the URL as an "id" of some object to retrieve from GORM or elsewhere. Especially in the case of GORM these are easily guessable as they are typically sequential integers.

Therefore you must assert that the requesting user is allowed to view the object with the requested id before returning the response to the user.

Not doing this is "security through obscurity" which is inevitably breached, just like having a default password of "letmein" and so on.

You must assume that every unprotected URL is publicly accessible one way or another.

11.2 Encoding and Decoding Objects

Grails supports the concept of dynamic encode/decode methods. A set of standard codecs are bundled with Grails. Grails also supports a simple mechanism for developers to contribute their own codecs that will be recognized at runtime.

Codec Classes

A Grails codec class is a class that may contain an encode closure, a decode closure or both. When a Grails application starts up the Grails framework will dynamically load codecs from the `grails-app/utils/` directory. The framework will look under `grails-app/utils/` for class names that end with the convention `Codec`. For example one of the standard codecs that ship with Grails is `HTMLCodec`.

If a codec contains an `encode` property assigned a block of code Grails will create a dynamic `encode` method and add that method to the Object class with a name representing the codec that defined the encode closure. For example, the `HTMLCodec` class defines an `encode` block so Grails will attach that closure to the `Object` class with the name `encodeAsHTML`.

The `HTMLCodec` and `URLCodec` classes also define a `decode` block so Grails will attach those with the names `decodeHTML` and `decodeURL`. Dynamic codec methods may be invoked from anywhere in a Grails application. For example, consider a case where a report contains a property called 'description' and that description may contain special characters that need to be escaped to be presented in an HTML document. One way to deal with that in a GSP is to encode the description property using the dynamic encode method as shown below:

```
${report.description.encodeAsHTML()}
```

Decoding is performed using `value.decodeHTML()` syntax.

Standard Codecs
HTMLCodec

This codec perfoms HTML escaping and unescaping, so that values you provide can be rendered safely in an HTML page without creating any HTML tags or damaging the page layout. For example, given a value "Don't you know that 2 > 1?" you wouldn't be able to show this safely within an HTML page because the > will look like it closes a tag, which is especially bad if you render this data within an attribute, such as the value attribute of an input field. Example of usage:

```
<input name="comment.message" value="${comment.message.encodeAsHTML()}"/>
```

> ⚠ Note that the HTML encoding does not re-encode apostrophe/single quote so you must use double quotes on attribute values to avoid text with apostrophes messing up your page.

URLCodec

URL encoding is required when creating URLs in links or form actions, or any time data may be used to create a URL. It prevents illegal characters getting into the URL to change its meaning, for example a "Apple & Blackberry" is not going to work well as a parameter in a GET request as the ampersand will break the parsing of parameters. Example of usage:

```
<a href="/mycontroller/find?searchKey=${lastSearch.encodeAsURL()}">Repeat last search</a>
```

Base64Codec

Performs Base64 encode/decode functions. Example of usage:

```
Your registration code is: ${user.registrationCode.encodeAsBase64()}
```

JavaScriptCodec

Will escape Strings so they can be used as valid JavaSctipt strings. Example of usage:

```
Element.update('${elementId}', '${render(template: "/common/message").encodeAsJavaScript()}'
```

HexCodec

Will encode byte arrays or lists of integers to lowercase hexadecimal strings, and can decode hexadecimal strings into byte arrays. Example of usage:

```
Selected colour: #${[255,127,255].encodeAsHex()}
```

MD5Codec

Will use the MD5 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default system encoding), as a lowercase hexadecimal string. Example of usage:

```
Your API Key: ${user.uniqueID.encodeAsMD5()}
```

MD5BytesCodec

Will use the MD5 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default system encoding), as a byte array. Example of usage:

```
byte[] passwordHash = params.password.encodeAsMD5Bytes()
```

SHA1Codec

Will use the SHA1 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default system encoding), as a lowercase hexadecimal string. Example of usage:

```
Your API Key: ${user.uniqueID.encodeAsSHA1()}
```

SHA1BytesCodec
Will use the SHA1 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default system encoding), as a byte array. Example of usage:

```
byte[] passwordHash = params.password.encodeAsSHA1Bytes()
```

SHA256Codec
Will use the SHA256 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default system encoding), as a lowercase hexadecimal string. Example of usage:

```
Your API Key: ${user.uniqueID.encodeAsSHA256()}
```

SHA256BytesCodec
Will use the SHA256 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default system encoding), as a byte array. Example of usage:

```
byte[] passwordHash = params.password.encodeAsSHA256Bytes()
```

Custom Codecs
Applications may define their own codecs and Grails will load them along with the standard codecs. A custom codec class must be defined in the grails-app/utils/ directory and the class name must end with Codec. The codec may contain a static encode block, a static decode block or both. The block should expect a single argument which will be the object that the dynamic method was invoked on. For Example:

```
class PigLatinCodec {
  static encode = { str ->
    // convert the string to piglatin and return the result
  }
}
```

With the above codec in place an application could do something like this:

```
${lastName.encodeAsPigLatin()}
```

11.3 Authentication

Although there is no current default mechanism for authentication as it is possible to implement authentication in literally thousands of different ways. It is however, trivial to implement a simple authentication mechanism using either interceptors or filters.
Filters allow you to apply authentication across all controllers or across a URI space. For example you can create a new set of filters in a class called grails-app/conf/SecurityFilters.groovy:

```
class SecurityFilters {
    def filters = {
        loginCheck(controller:'*', action:'*') {
            before = {
                if(!session.user && actionName != "login") {
                    redirect(controller:"user",action:"login")
                    return false
                }
            }
        }
    }
}
```

Here the `loginCheck` filter will intercept execution *before* an action executed and if their is no user in the session and the action being executed is not the `login` action then redirect to the `login` action.
The `login` action itself is trivial too:

```
def login = {
    if(request.get) render(view:"login")
    else {
        def u = User.findByLogin(params.login)
        if(u) {
            if(u.password == params.password) {
                session.user = u
                redirect(action:"home")
            }
            else {
                render(view:"login", model:[message:"Password incorrect"])
            }
        }
        else {
            render(view:"login", model:[message:"User not found"])
        }
    }
}
```

11.4 Security Plug-ins

If you need more advanced functionality beyond simple authentication such as authorization, roles etc. then you may want to consider using one of the available security plug-ins.

11.4.1 Acegi

The Acegi Plugin is built on the [Spring Security](#) project which provides a flexible, extensible framework for building all sorts of authentication and authorization schemes.
The Acegi plugin requires you to specify a mapping between URIs and roles and provides a default domain model to model people, authorities and request maps. See the [documentation on the wiki](#) for more information.

11.4.2 JSecurity

[JSecurity](#) is another Java POJO oriented security framework that again provides a default domain model that models realms, users, roles and permissions. With JSecurity you have to extends a controller base called called `JsecAuthBase` in each controller you want secured and then provide an `accessControl` block to setup the roles. An example below:

```
class ExampleController extends JsecAuthBase {
    static accessControl = {
        // All actions require the 'Observer' role.
        role(name: 'Observer')
        // The 'edit' action requires the 'Administrator' role.
        role(name: 'Administrator', action: 'edit')
        // Alternatively, several actions can be specified.
        role(name: 'Administrator', only: [ 'create', 'edit', 'save', 'update' ])
    }
    …
}
```

For more information on the JSecurity plug-in refer to the [JSecurity Quick Start](#).

12. Plug-ins

Grails provides a number of extension points that allow you to extend anything from the command line interface to the runtime configuration engine. The following sections detail how to go about it.

12.1 Creating and Installing Plug-ins

Creating Plugins

Creating a Grails plugin is a simple matter of running the command:

```
grails create-plugin [PLUGIN NAME]
```

This will create a plugin project for the name you specify. Say for example you run `grails create-plugin example`. This would create a new plugin project called `example`.
The structure of a Grails plugin is exactly the same as a regular Grails project's directory structure, except that in the root of the plugin directory you will find a plugin Groovy file called the "plugin descriptor".
Being a regular Grails project has a number of benefits in that you can immediately get going testing your plugin by running:

```
grails run-app
```

The plugin descriptor itself ends with the convention `GrailsPlugin` and is found in the root of the plugin project. For example:

```
class ExampleGrailsPlugin {
   def version = 0.1
   …
}
```

All plugins must have this class in the root of their directory structure to be valid. The plugin class defines the version of the plugin and optionally various hooks into plugin extension points (covered shortly).
You can also provide additional information about your plugin using several special properties:

- `title` - short one sentence description of your plugin
- `version` - The version of your problem. Valid versions are for example "0.1", "0.2-SNAPSHOT", "0.1.4" etc.
- `grailsVersion` - The version of version range of Grails that the plugin supports. eg. "1.1 > *"
- `author` - plug-in author's name
- `authorEmail` - plug-in author's contact e-mail
- `description` - full multi-line description of plug-in's features
- `documentation` - URL where plug-in's documentation can be found

Here is an example from [Quartz Grails plugin](#)

```
class QuartzGrailsPlugin {
    def version = "0.1"
    def grailsVersion = "1.1 > *"
    def author = "Sergey Nebolsin"
    def authorEmail = "nebolsin@gmail.com"
    def title = "This plugin adds Quartz job scheduling features to Grails application."
    def description = '''
Quartz plugin allows your Grails application to schedule jobs to be
executed using a specified interval or cron expression. The underlying
system uses the Quartz Enterprise Job Scheduler configured via Spring,
but is made simpler by the coding by convention paradigm.
'''
    def documentation = "http://grails.org/Quartz+plugin"
  …
}
```

Installing & Distributing Plugins

To distribute a plugin you need to navigate to its root directory in a terminal window and then type:

```
grails package-plugin
```

This will create a zip file of the plugin starting with `grails-` then the plugin name and version. For example with the example plug-in created earlier this would be `grails-example-0.1.zip`. The `package-plugin` command will also generate `plugin.xml` file which contains machine-readable information about plugin's name, version, author, and so on.

Once you have a plugin distribution file you can navigate to a Grails project and type:

```
grails install-plugin /path/to/plugin/grails-example-0.1.zip
```

If the plugin is hosted on a remote HTTP server you can also do:

```
grails install-plugin http://myserver.com/plugins/grails-example-0.1.zip
```

Notes on excluded Artefacts

Although the create-plugin command creates certain files for you so that the plug-in can be run as a Grails application, not all of these files are included when packaging a plug-in. The following is a list of artefacts created, but not included by package-plugin:

- `grails-app/conf/DataSource.groovy`
- `grails-app/conf/UrlMappings.groovy`
- `build.xml`
- Everything within `/web-app/WEB-INF`

If you need artefacts within `WEB-INF` it is recommended you use the `_Install.groovy` script (covered later), which is executed when a plug-in is installed, to provide such artefacts. In addition, although `UrlMappings.groovy` is excluded you are allowed to include a `UrlMappings` definition with a different name, such as `FooUrlMappings.groovy`.

Specifying Plugin Locations

An application can load plugins from anywhere on the file system, even if they have not been installed. Simply add the location of the (unpacked) plugin to the application's `grails-app/conf/BuildConfig.groovy` file:

```
// Useful to test plugins you are developing.
grails.plugin.location.jsecurity = "/home/dilbert/dev/plugins/grails-jsecurity"
// Useful for modular applications where all plugins and
// applications are in the same directory.
grails.plugin.location.'grails-ui' = "../grails-grails-ui"
```

This is particularly useful in two cases:

- You are developing a plugin and want to test it in a real application without packaging and installing it first.
- You have split an application into a set of plugins and an application, all in the same "super-project" directory.

12.2 Plugin Repositories

Distributing Plugins in the Grails Central Plugins Repository

The preferred way of plugin distribution is to publish your under Grails Plugins Repository. This will make your plugin visible to the list-plugins command:

```
grails list-plugins
```

Which lists all plugins in the Grails Plugin repository and also the plugin-info command:

```
grails plugin-info [plugin-name]
```

Which outputs more information based on the meta info entered into the plug-in descriptor.

> ⚠️ If you have created a Grails plugin and want it to be hosted in the central repository take a look at the wiki page , which details how to go about releasing your plugin in the repository.

When you have access to the Grails Plugin repository to release your plugin you simply have to execute the release-plugin command:

```
grails release-plugin
```

This will automatically commit changes to SVN, do some tagging and make your changes available via the list-plugins command.

Configuring Additional Repositories
The way in which you configure repositories in Grails differs between Grails versions. For version of Grails 1.2 and earlier please refer to the Grails 1.2 documentation on the subject. The following sections cover Grails 1.3 and above. Grails 1.3 and above use Ivy under the hood to resolve plugin dependencies. The mechanism for defining additional plugin repositories is largely the same as defining repositories for JAR dependencies. For example you can define a remote Maven repository that contains Grails plugins using the following syntax in grails-app/conf/BuildConfig.groovy:

```
repositories {
    mavenRepo "http://repository.codehaus.org"
}
```

You can also define a SVN-based Grails repository (such as the one hosted at http://plugins.grails.org/) using the grailsRepo method:

```
repositories {
    grailsRepo "http://myserver/mygrailsrepo"
}
```

There is a shortcut to setup the Grails central repository:

```
repositories {
    grailsCentral()
}
```

The order in which plugins are resolved is based on the ordering of the repositories. So for example in this case the Grails central repository will be searched last:

173

```
repositories {
    grailsRepo "http://myserver/mygrailsrepo"
    grailsCentral()
}
```

All of the above examples use HTTP, however you can specify any [Ivy resolver](#) to resolve plugins with. Below is an example that uses an SSH resolver:

```
def sshResolver = new SshResolver(user:"myuser", host:"myhost.com")
sshResolver.addArtifactPattern("/path/to/repo/grails-[artifact]/tags/LATEST_RELEASE/grails-[
sshResolver.latestStrategy = new org.apache.ivy.plugins.latest.LatestTimeStrategy()
sshResolver.changingPattern = ".*SNAPSHOT"
sshResolver.setCheckmodified(true)
```

The above example defines an artifact pattern which tells Ivy how to resolve a plugin zip file. For a more detailed explanation on Ivy patterns see the [relevant section](#) in the Ivy user guide.

Publishing to Maven Compatible Repositories
In general it is recommended for Grails 1.3 and above to use standard Maven-style repositories to self host plugins. The benefits of doing so include the ability for existing tooling and repository managers to interpret the structure of a Maven repository. In addition Maven compatible repositories are not tied to SVN as Grails repositories are.
In order to publish a plugin to a Maven repository you need to use the Maven publisher plugin. Please refer to the section of the [Maven deployment](#) user guide on the subject.

Publishing to Grails Compatible Repositories
To publish a Grails plugin to a Grails compatible repository you specify the
`grails.plugin.repos.distribution.myRepository` setting within the
grails-app/conf/BuildConfig.groovy file:

```
grails.plugin.repos.distribution.myRepository="https://svn.codehaus.org/grails/trunk/grails-
```

You can also provide this settings in the USER_HOME/.grails/settings.groovy file if you prefer to share the same settings across multiple projects.
Once this is done you need to use the `repository` argument of the `release-plugin` command to specify the repository you want to release the plugin into:

```
grails release-plugin -repository=myRepository
```

12.3 Understanding a Plug-ins Structure

As as mentioned previously, a plugin is merely a regular Grails application with a contained plug-in descriptors. However when installed, the structure of a plugin differs slightly. For example, take a look at this plugin directory structure:

```
+ grails-app
    + controllers
    + domain
    + taglib
    etc.
 + lib
 + src
    + java
    + groovy
 + web-app
    + js
    + css
```

Essentially when a plugin is installed into a project, the contents of the grails-app directory will go into a directory such as plugins/example-1.0/grails-app. They **will not** be copied into the main source tree. A plugin never interferes with a project's primary source tree.

Dealing with static resources is slightly different. When developing a plugin, just like an application, all static resources can go in the web-app directory. You can then link to static resources just like in an application (example below links to a javascript source):

```
<g:resource dir="js" file="mycode.js" />
```

When you run the plugin in development mode the link to the resource will resolve to something like /js/mycode.js. However, when the plugin is installed into an application the path will automatically change to something like /plugin/example-0.1/js/mycode.js and Grails will deal with making sure the resources are in the right place.

There is a special pluginContextPath variable that can be used whilst both developing the plugin and when in the plugin is installed into the application to find out what the correct path to the plugin is.

At runtime the pluginContextPath variable will either evaluate to an empty string or /plugins/example depending on whether the plugin is running standalone or has been installed in an application

Java & Groovy code that the plugin provides within the lib and src/java and src/groovy directories will be compiled into the main project's web-app/WEB-INF/classes directory so that they are made available at runtime.

12.4 Providing Basic Artefacts

Adding a new Script

A plugin can add a new script simply by providing the relevant Gant script within the scripts directory of the plugin:

```
+ MyPlugin.groovy
   + scripts      <-- additional scripts here
   + grails-app
       + controllers
       + services
       + etc.
    + lib
```

Adding a new Controller, Tag Library or Service

A plugin can add a new controller, tag libraries, service or whatever by simply creating the relevant file within the grails-app tree. Note that when the plugin is installed it will be loaded from where it is installed and not copied into the main application tree.

```
+ ExamplePlugin.groovy
   + scripts
   + grails-app
       + controllers  <-- additional controllers here
       + services <-- additional services here
       + etc.   <-- additional XXX here
    + lib
```

Providing Views, Templates and View resolution

When a plug-in provides a controller it may also provide default views to be rendered. This is an excellent way to modularize your application through plugins. The way it works is that Grails' view resolution mechanism will first look the view in the application it is installed into and if that fails will attempt to look for the view within the plug-in. For example given a AmazonGrailsPlugin plug-n provided controller called BookController if the action being executed is list, Grails will first look for a view called grails-app/views/book/list.gsp then if that fails will look for the same view relative to the plug-in.

Note however that if the view uses templates that are also provided by the plugin then the following syntax may be necessary:

```
<g:render template="fooTemplate" plugin="amazon"/>
```

Note the usage of the `plugin` attribute, which contains the name of the plugin where the template resides. If this is not specified then Grails will look for the template relative to the application.

Excluded Artefacts

Note that by default, when packaging a plug-in, Grails will excludes the following files from the packaged plug-in:

- ○ grails-app/conf/DataSource.groovy
- ○ grails-app/conf/UrlMappings.groovy
- ○ Everything under web-app/WEB-INF

If your plug-in does require files under the `web-app/WEB-INF` directory it is recommended that you modify the plug-in's `scripts/_Install.groovy` Gant script to install these artefacts into the target project's directory tree. In addition, the default `UrlMappings.groovy` file is excluded to avoid naming conflicts, however you are free to add a UrlMappings definition under a different name which **will** be included. For example a file called `grails-app/conf/BlogUrlMappings.groovy` is fine.

Additionally the list of includes is extensible via the `pluginExcludes` property:

```
// resources that are excluded from plugin packaging
def pluginExcludes = [
        "grails-app/views/error.gsp"
]
```

This is useful, for example, if you want to include demo or test resources in the plugin repository, but not include them in the final distribution.

12.5 Evaluating Conventions

Before moving onto looking at providing runtime configuration based on conventions you first need to understand how to evaluated those conventions from a plug-in. Essentially every plugin has an implicit `application` variable which is an instance of the [GrailsApplication](#) interface.

The `GrailsApplication` interface provides methods to evaluate the conventions within the project and internally stores references to all classes within a GrailsApplication using the [GrailsClass](#) interface.

A `GrailsClass` represents a physical Grails resources such as a controller or a tag library. For example to get all `GrailsClass` instances you can do:

```
application.allClasses.each { println it.name }
```

There are a few "magic" properties that the `GrailsApplication` instance possesses that allow you to narrow the type of artefact you are interested in. For example if you only want to controllers you can do:

```
application.controllerClasses.each { println it.name }
```

The dynamic method conventions are as follows:

- ○ `*Classes` - Retrieves all the classes for a particular artefact name. Example `application.controllerClasses`.
- ○ `get*Class` - Retrieves a named class for a particular artefact. Example `application.getControllerClass("ExampleController")`
- ○ `is*Class` - Returns true if the given class is of the given artefact type. Example `application.isControllerClass(ExampleController.class)`

The `GrailsClass` interface itself provides a number of useful methods that allow you to further evaluate and work with the conventions. These include:

- ○ `getPropertyValue` - Gets the initial value of the given property on the class
- ○ `hasProperty` - Returns true if the class has the specified property
- ○ `newInstance` - Creates a new instance of this class.
- ○ `getName` - Returns the logical name of the class in the application without the trailing convention part if applicable
- ○ `getShortName` - Returns the short name of the class without package prefix
- ○ `getFullName` - Returns the full name of the class in the application with the trailing convention part and with the package name
- ○ `getPropertyName` - Returns the name of the class as a property name
- ○ `getLogicalPropertyName` - Returns the logical property name of the class in the application without the trailing convention part if applicable
- ○ `getNaturalName` - Returns the name of the property in natural terms (eg. 'lastName' becomes 'Last Name')
- ○ `getPackageName` - Returns the package name

For a full reference refer to the javadoc API.

12.6 Hooking into Build Events

Post-Install Configuration and Participating in Upgrades

Grails plug-ins can do post-install configuration and participate in application upgrade process (the upgrade command). This is achieved via two specially named scripts under `scripts` directory of the plugin - `_Install.groovy` and `_Upgrade.groovy`.

`_Install.groovy` is executed after the plugin has been installed and `_Upgrade.groovy` is executed each time the user upgrades his application with upgrade command.

These scripts are normal Gant scripts so you can use the full power of Gant. An addition to the standard Gant variables is the `pluginBasedir` variable which points at the plugin installation basedir.

As an example the below `_Install.groovy` script will create a new directory type under the `grails-app` directory and install a configuration template:

```
Ant.mkdir(dir:"${basedir}/grails-app/jobs")
Ant.copy(file:"${pluginBasedir}/src/samples/SamplePluginConfiguration.groovy",
        todir:"${basedir}/grails-app/conf")
// To access Grails home you can use following code:
// Ant.property(environment:"env")
// grailsHome = Ant.antProject.properties."env.GRAILS_HOME"
```

Scripting events

It is also possible to hook into command line scripting events through plug-ins. These are events triggered during execution of Grails target and plugin scripts.

For example, you can hook into status update output (i.e. "Tests passed", "Server running") and the creation of files or artefacts.

A plug-in merely has to provide an `_Events.groovy` script to listen to the required events. Refer the documentation on Hooking into Events for further information.

12.7 Hooking into Runtime Configuration

Grails provides a number of hooks to leverage the different parts of the system and perform runtime configuration by convention.

Hooking into the Grails Spring configuration

First, you can hook in Grails runtime configuration by providing a property called `doWithSpring` which is assigned a block of code. For example the following snippet is from one of the core Grails plugins that provides i18n support:

```
import org.springframework.web.servlet.i18n.CookieLocaleResolver;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.context.support.ReloadableResourceBundleMessageSource;
class I18nGrailsPlugin {
    def version = 0.1
    def doWithSpring = {
        messageSource(ReloadableResourceBundleMessageSource) {
            basename = "WEB-INF/grails-app/i18n/messages"
        }
        localeChangeInterceptor(LocaleChangeInterceptor) {
            paramName = "lang"
        }
        localeResolver(CookieLocaleResolver)
    }
}
```

This plugin sets up the Grails `messageSource` bean and a couple of other beans to manage Locale resolution and switching. It using the [Spring Bean Builder](#) syntax to do so.

Participating in web.xml Generation

Grails generates the `WEB-INF/web.xml` file at load time, and although plugins cannot change this file directly, they can participate in the generation of the file. Essentially a plugin can provide a `doWithWebDescriptor` property that is assigned a block of code that gets passed the `web.xml` as a `XmlSlurper GPathResult`. Consider the below example from the `ControllersPlugin`:

```
def doWithWebDescriptor = { webXml ->
    def mappingElement = webXml.'servlet-mapping'
    def lastMapping = mappingElement[mappingElement.size()-1]
    lastMapping + {
        'servlet-mapping' {
            'servlet-name'("grails")
            'url-pattern'("*.dispatch")
        }
    }
}
```

Here the plugin goes through gets a reference to the last `<servlet-mapping>` element and appends Grails' servlet to the end of it using XmlSlurper's ability to programmatically modify XML using closures and blocks.

Doing Post Initialisation Configuration

Sometimes it is useful to be able do some runtime configuration after the Spring [ApplicationContext](#) has been built. In this case you can define a `doWithApplicationContext` closure property.

```
class SimplePlugin {
    def name="simple"
    def version = 1.1
    def doWithApplicationContext = { appCtx ->
        SessionFactory sf = appCtx.getBean("sessionFactory")
        // do something here with session factory
    }
}
```

12.8 Adding Dynamic Methods at Runtime

The Basics

Grails plugins allow you to register dynamic methods with any Grails managed or other class at runtime. New methods can only be added within a `doWithDynamicMethods` closure of a plugin.

For Grails managed classes like controllers, tag libraries and so forth you can add methods, constructors etc. using the [ExpandoMetaClass](#) mechanism by accessing each controller's [MetaClass](#):

```
class ExamplePlugin {
  def doWithDynamicMethods = { applicationContext ->
      application.controllerClasses.each { controllerClass ->
          controllerClass.metaClass.myNewMethod = {-> println "hello world" }
      }
  }
}
```

In this case we use the implicit application object to get a reference to all of the controller classes' MetaClass instances and then add a new method called `myNewMethod` to each controller. Alternatively, if you know before hand the class you wish the add a method to you can simple reference that classes `metaClass` property:

```
class ExamplePlugin {
  def doWithDynamicMethods = { applicationContext ->
      String.metaClass.swapCase = {->
          def sb = new StringBuffer()
          delegate.each {
              sb << (Character.isUpperCase(it as char) ?
                      Character.toLowerCase(it as char) :
                      Character.toUpperCase(it as char))
          }
          sb.toString()
      }
      assert "UpAndDown" == "uPaNDdOWN".swapCase()
  }
}
```

In this example we add a new method `swapCase` to `java.lang.String` directly by accessing its `metaClass`.

Interacting with the ApplicationContext
The `doWithDynamicMethods` closure gets passed the Spring `ApplicationContext` instance. This is useful as it allows you to interact with objects within it. For example if you where implementing a method to interact with Hibernate you could use the `SessionFactory` instance in combination with a `HibernateTemplate`:

```
import org.springframework.orm.hibernate3.HibernateTemplate
class ExampleHibernatePlugin {
  def doWithDynamicMethods = { applicationContext ->
      application.domainClasses.each { domainClass ->
          domainClass.metaClass.static.load = { Long id->
              def sf = applicationContext.sessionFactory
              def template = new HibernateTemplate(sf)
              template.load(delegate, id)
          }
      }
  }
}
```

Also because of the autowiring and dependency injection capability of the Spring container you can implement more powerful dynamic constructors that use the application context to wire dependencies into your object at runtime:

```
class MyConstructorPlugin {
    def doWithDynamicMethods = { applicationContext ->
        application.domainClasses.each { domainClass ->
            domainClass.metaClass.constructor = {->
                return applicationContext.getBean(domainClass.name)
            }
        }
    }
}
```

Here we actually replace the default constructor with one that looks up prototyped Spring beans instead!

12.9 Participating in Auto Reload Events

Monitoring Resources for Changes

Often it is valuable to monitor resources for changes and then reload those changes when they occur. This is how Grails implements advanced reloading of application state at runtime. For example, consider the below simplified snippet from the `ServicesPlugin` that Grails comes with:

```
class ServicesGrailsPlugin {
    …
    def watchedResources = "file:./grails-app/services/*Service.groovy"
    …
    def onChange = { event ->
        if(event.source) {
            def serviceClass = application.addServiceClass(event.source)
            def serviceName = "${serviceClass.propertyName}"
            def beans = beans {
                "$serviceName"(serviceClass.getClazz()) { bean ->
                    bean.autowire =   true
                }
            }
            if(event.ctx) {
                event.ctx.registerBeanDefinition(serviceName,
                                            beans.getBeanDefinition(serviceName))
            }
        }
    }
}
```

Firstly it defines a set of `watchedResources` as either a String or a List of strings that contain either the references or patterns of the resources to watch. If the watched resources is a Groovy file, when it is changed it will automatically be reloaded and passed into the `onChange` closure inside the `event` object.

The `event` object defines a number of useful properties:

- ○ `event.source` - The source of the event which is either the reloaded class or a Spring Resource
- ○ `event.ctx` - The Spring `ApplicationContext` instance
- ○ `event.plugin` - The plugin object that manages the resource (Usually this)
- ○ `event.application` - The `GrailsApplication` instance

From these objects you can evaluate the conventions and then apply the appropriate changes to the `ApplicationContext` and so forth based on the conventions, etc. In the "Services" example above, a new services bean is re-registered with the `ApplicationContext` when one of the service classes changes.

Influencing Other Plugins

As well as being able to react to changes that occur when a plugin changes, sometimes one plugin needs to "influence" another plugin.

Take for example the Services & Controllers plugins. When a service is reloaded, unless you reload the controllers too, problems will occur when you try to auto-wire the reloaded service into an older controller Class.

To get round this, you can specify which plugins another plugin "influences". What this means is that when one plugin detects a change, it will reload itself and then reload all influenced plugins. See this snippet from the `ServicesGrailsPlugin`:

```
def influences = ['controllers']
```

Observing other plugins

If there is a particular plugin that you would like to observe for changes but not necessary watch the resources that it monitors you can use the "observe" property:

```
def observe = ["controllers"]
```

In this case when a controller is changed you will also receive the event chained from the controllers plugin. It is also possible for a plugin to observe all loaded plugins by using a wildcard:

```
def observe = ["*"]
```

The Logging plugin does exactly this so that it can add the `log` property back to *any* artefact that changes while the application is running.

12.10 Understanding Plug-in Load Order

Controlling Plug-in Dependencies
Plug-ins often depend on the presence of other plugins and can also adapt depending on the presence of others. To cover this, a plugin can define two properties. The first is called `dependsOn`. For example, take a look at this snippet from the Grails Hibernate plugin:

```
class HibernateGrailsPlugin {
    def version = 1.0
    def dependsOn = [dataSource:1.0,
                     domainClass:1.0,
                     i18n:1.0,
                     core: 1.0]
}
```

As the above example demonstrates the Hibernate plugin is dependent on the presence of 4 plugins: The `dataSource` plugin, The `domainClass` plugin, the `i18n` plugin and the `core` plugin.
Essentially the dependencies will be loaded first and then the Hibernate plugin. If all dependencies do not load, then the plugin will not load.
The `dependsOn` property also supports a mini expression language for specifying version ranges. A few examples of the syntax can be seen below:

```
def dependsOn = [foo:"* > 1.0"]
def dependsOn = [foo:"1.0 > 1.1"]
def dependsOn = [foo:"1.0 > *"]
```

When the wildcard * character is used it denotes "any" version. The expression syntax also excludes any suffixes such as -BETA, -ALPHA etc. so for example the expression "1.0 > 1.1" would match any of the following versions:

- 1.1
- 1.0
- 1.0.1
- 1.0.3-SNAPSHOT
- 1.1-BETA2

Controlling Load Order
Using `dependsOn` establishes a "hard" dependency in that if the dependency is not resolved, the plugin will give up and won't load. It is possible though to have a "weaker" dependency using the `loadAfter` property:

```
def loadAfter = ['controllers']
```

Here the plugin will be loaded after the `controllers` plugin if it exists, otherwise it will just be loaded. The plugin can then adapt to the presence of the other plugin, for example the Hibernate plugin has this code in the `doWithSpring` closure:

```
if(manager?.hasGrailsPlugin("controllers")) {
    openSessionInViewInterceptor(OpenSessionInViewInterceptor) {
            flushMode = HibernateAccessor.FLUSH_MANUAL
            sessionFactory = sessionFactory
    }
        grailsUrlHandlerMapping.interceptors << openSessionInViewInterceptor
}
```

Here the Hibernate plugin will only register an `OpenSessionInViewInterceptor` if the `controllers` plugin has been loaded. The manager variable is an instance of the [GrailsPluginManager](#) interface and it provides methods to interact with other plugins and the `GrailsPluginManager` itself from any plugin.

13. Web Services

Web services are all about providing a web API onto your web application and are typically implemented in either SOAP or REST.

13.1 REST

REST is not really a technology in itself, but more an architectural pattern. REST is extremely simple and just involves using plain XML or JSON as a communication medium, combined with URL patterns that are "representational" of the underlying system and HTTP methods such as GET, PUT, POST and DELETE. Each HTTP method maps to an action. For example GET for retrieving data, PUT for creating data, POST for updating and so on. In this sense REST fits quite well with CRUD.

URL patterns

The first step to implementing REST with Grails is to provide RESTful URL mappings:

```
static mappings = {
    "/product/$id?"(resource:"product")
}
```

What this does is map the URI /product onto a ProductController. Each HTTP method such as GET, PUT, POST and DELETE map to unique actions within the controller as outlined by the table below:

Method	Action
GET	show
PUT	update
POST	save
DELETE	delete

You can alter how HTTP methods by using the capability of URL Mappings to map to HTTP methods:

```
"/product/$id"(controller:"product"){
    action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
}
```

However, unlike the resource argument used previously, in this case Grails will not provide automatic XML or JSON marshaling for you unless you specify the parseRequest argument in the URL mapping:

```
"/product/$id"(controller:"product", parseRequest:true){
    action = [GET:"show", PUT:"update", DELETE:"delete", POST:"save"]
}
```

HTTP Methods

In the previous section you saw how you can easily define URL mappings that map specific HTTP methods onto specific controller actions. Writing a REST client that then sends a specific HTTP method is then trivial (example in Groovy's HTTPBuilder module):

```
import groovyx.net.http.*
import static groovyx.net.http.ContentType.JSON
def http = new HTTPBuilder("http://localhost:8080/amazon")
 http.request(Method.GET, JSON) {
     url.path = '/book/list'
     response.success = {resp, json ->
         json.books.each { book ->
             println book.title
         }
     }
 }
```

However, issuing a request with a method other than GET or POST from a regular browser is not possible without some help from Grails. When defining a form you can specify an alternative method such as DELETE:

```
<g:form controller="book" method="DELETE">
    ..
</g:form>
```

Grails will send a hidden parameter called _method, which will be used as the request's HTTP method. Another alternative for changing the method for non-browser clients is to use the X-HTTP-Method-Override to specify the alternative method name.

XML Marshaling - Reading
The controller implementation itself can use Grails' XML marshaling support to implement the GET method:

```
import grails.converters.*
class ProductController {
    def show = {
        if(params.id && Product.exists(params.id)) {
            def p = Product.findByName(params.id)
            render p as XML
        }
        else {
            def all = Product.list()
            render all as XML
        }
    }
    ..
}
```

Here what we do is if there is an id we search for the Product by name and return it otherwise we return all Products. This way if we go to /products we get all products, otherwise if we go to /product/MacBook we only get a MacBook.

XML Marshalling - Updating
To support updates such as PUT and POST you can use the params object which Grails enhances with the ability to read an incoming XML packet. Given an incoming XML packet of:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<product>
    <name>MacBook</name>
    <vendor id="12">
        <name>Apple</name>
    </vender>
</product>
```

You can read this XML packet using the same techniques described in the Data Binding section via the params object:

```
def save = {
    def p = new Product(params['product'])
    if(p.save()) {
        render p as XML
    }
    else {
        render p.errors
    }
}
```

In this example by indexing into the `params` object using the key `'product'` we can automatically create and bind the XML using the constructor of the `Product` class. An interesting aspect of the line:

```
def p = new Product(params['product'])
```

Is that it requires no code changes to deal with a form submission that submits form data than it does to deal with an XML request. The exact same technique can be used with a JSON request too.

> ⚠️ If you require different responses to different clients (REST, HTML etc.) you can use content negotation

The `Product` object is then saved and rendered as XML, otherwise an error message is produced using Grails' validation capabilities in the form:

```
<error>
    <message>The property 'title' of class 'Person' must be specified</message>
</error>
```

13.2 SOAP

There are several plugins that add SOAP support to Grails depending on your preferred approach. For Contract First SOAP services there is a Spring WS plugin, whilst if you want to generate a SOAP API from Grails services there are several plugins that do this including:

- XFire plugin which uses the XFire SOAP stack
- CXF plugin which uses the CXF SOAP stack
- Axis2 plugin which uses Axis2
- Metro plugin which uses the Metro framework (and can also be used for Contract First)

Most of the SOAP integrations integrate with Grails services via the `exposes` static property. The below example is taken from the XFire plugin:

```
class BookService {
  static expose=['xfire']
  Book[] getBooks(){
    Book.list() as Book[]
  }
}
```

The WSDL can then be accessed at the location:
`http://127.0.0.1:8080/your_grails_app/services/book?wsdl`
For more information on the XFire plug-in refer the documentation on the wiki.

13.3 RSS and Atom

No direct support is provided for RSS or Atom within Grails. You could construct RSS or ATOM feeds with the render method's XML capability. There is however a Feeds plug-in available for Grails that provides a RSS and

Atom builder using the popular [ROME](#) library. An example of its usage can be seen below:

```
def feed = {
    render(feedType:"rss", feedVersion:"2.0") {
        title = "My test feed"
        link = "http://your.test.server/yourController/feed"
        Article.list().each() {
            entry(it.title) {
                link = "http://your.test.server/article/${it.id}"
                it.content // return the content
            }
        }
    }
}
```

14. Grails and Spring

This section is for advanced users and those who are interested in how Grails integrates with and builds on the [Spring Framework](#) This section is also useful for [plug-in developers](#) considering doing runtime configuration Grails.

14.1 The Underpinnings of Grails

Grails is actually a [Spring MVC](#) application in disguise. Spring MVC is the Spring framework's built-in MVC web application framework. Although Spring MVC suffers from the same difficulties as frameworks like Struts in terms of its ease of use, it is superbly designed and architected and was, for Grails, the perfect framework to build another framework on top of.
Grails leverages Spring MVC in the following areas:

- Basic controller logic - Grails subclasses Spring's [DispatcherServlet](#) and uses it to delegate onto Grails [controllers](#)
- Data Binding and Validation - Grails' [validation](#) and [data binding](#) capabilities are built on those provided by Spring
- Runtime configuration - Grails' entire runtime convention based system is wired together by a Spring [ApplicationContext](#)
- Transactions - Grails uses Spring's transaction management in [GORM](#)

In other words Grails has Spring embedded running all the way through it.

The Grails ApplicationContext
Spring developers are often keen to understand how the Grails `ApplicationContext` instance is constructed. The basics of it are as follows.

- Grails constructs a parent `ApplicationContext` from the `web-app/WEB-INF/applicationContext.xml`. This `ApplicationContext` sets up the [GrailsApplication](#) instance and the [GrailsPluginManager](#).
- Using this `ApplicationContext` as a parent Grails' analyses the conventions with the `GrailsApplication` instance and constructs a child `ApplicationContext` that is used as the root `ApplicationContext` of the web application

Configured Spring Beans
Most of Grails' configuration happens at runtime. Each [plug-in](#) may configure Spring beans that are registered with the `ApplicationContext`. For a reference as to which beans are configured refer to the reference guide which describes each of the Grails plug-ins and which beans they configure.

14.2 Configuring Additional Beans

Using XML
Beans can be configured using the `grails-app/conf/spring/resources.xml` file of your Grails application. This file is typical Spring XML file and the Spring documentation has an [excellent reference](#) on how to go about configuration Spring beans. As a trivial example you can configure a bean with the following syntax:

```
<bean id="myBean" class="my.company.MyBeanImpl"></bean>
```

Once configured the bean, in this case named `myBean`, can be auto-wired into most Grails types including controllers, tag libraries, services and so on:

```
class ExampleController {
    def myBean
}
```

Referencing Existing Beans
Beans declared in `resources.xml` can also reference Grails classes by convention. For example if you need a reference to a service such as `BookService` in your bean you use the property name representation of the class name. In the case of `BookService` this would be `bookService`. For example:

```
<bean id="myBean" class="my.company.MyBeanImpl">
    <property name="bookService" ref="bookService" />
</bean>
```

The bean itself would of course need a public setter, which in Groovy is defined like this:

```
package my.company
class MyBeanImpl {
    BookService bookService
}
```

or in Java like this:

```
package my.company;
class MyBeanImpl {
    private BookService bookService;
    public void setBookService(BookService theBookService) {
        this.bookService = theBookService;
    }
}
```

Since much of Grails configuration is done at runtime by convention many of the beans are not declared anywhere, but can still be referenced inside your Spring configuration. For example if you need a reference to the Grails DataSource you could do:

```
<bean id="myBean" class="my.company.MyBeanImpl">
    <property name="bookService" ref="bookService" />
    <property name="dataSource" ref="dataSource" />
</bean>
```

Or if you need the Hibernate SessionFactory this will work:

```
<bean id="myBean" class="my.company.MyBeanImpl">
    <property name="bookService" ref="bookService" />
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

For a full reference of the available beans see the Plug-in reference in the reference guide.

Using the Spring DSL
If you want to use the [Spring DSL] that Grails provides then you need to create a grails-app/conf/spring/resources.groovy file and define a property called beans that is assigned a block:

```
beans = {
    // beans here
}
```

The same configuration for the XML example could be represented as:

```
beans = {
    myBean(my.company.MyBeanImpl) {
        bookService = ref("bookService")
    }
}
```

The main advantage of this way is that you can now mix logic in within your bean definitions, for example based on the environment:

```
import grails.util.*
beans = {
    switch(GrailsUtil.environment) {
        case "production":
            myBean(my.company.MyBeanImpl) {
                bookService = ref("bookService")
            }
        break
        case "development":
            myBean(my.company.mock.MockImpl) {
                bookService = ref("bookService")
            }
        break
    }
}
```

14.3 Runtime Spring with the Beans DSL

This Bean builder in Grails aims to provide a simplified way of wiring together dependencies that uses Spring at its core.
In addition, Spring's regular way of configuration (via XML) is essentially static and very difficult to modify and configure at runtime other than programmatic XML creation which is both error prone and verbose. Grails' BeanBuilder changes all that by making it possible to programmatically wire together components at runtime thus allowing you to adapt the logic based on system properties or environment variables.
This enables the code to adapt to its environment and avoids unnecessary duplication of code (having different Spring configs for test, development and production environments)

The BeanBuilder class
Grails provides a grails.spring.BeanBuilder class that uses dynamic Groovy to construct bean definitions. The basics are as follows:

```
import org.apache.commons.dbcp.BasicDataSource
import org.codehaus.groovy.grails.orm.hibernate.ConfigurableLocalSessionFactoryBean;
import org.springframework.context.ApplicationContext;
def bb = new grails.spring.BeanBuilder()
bb.beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
    }
    sessionFactory(ConfigurableLocalSessionFactoryBean) {
        dataSource = dataSource
        hibernateProperties = [ "hibernate.hbm2ddl.auto":"create-drop",
                                "hibernate.show_sql":true  ]
    }
}
ApplicationContext appContext = bb.createApplicationContext()
```

> ⚠️ Within plug-ins and the grails-app/conf/spring/resources.groovy file you don't need to create a new instance of `BeanBuilder`. Instead the DSL is implicitly available inside the `doWithSpring` and `beans` blocks respectively.

The above example shows how you would configure Hibernate with an appropriate data source with the `BeanBuilder` class.

Essentially, each method call (in this case `dataSource` and `sessionFactory` calls) map to the name of the bean in Spring. The first argument to the method is the bean's class, whilst the last argument is a block. Within the body of the block you can set properties on the bean using standard Groovy syntax

Bean references are resolved automatically be using the name of the bean. This can be seen in the example above with the way the `sessionFactory` bean resolves the `dataSource` reference.

Certain special properties related to bean management can also be set by the builder, as seen in the following code:

```
sessionFactory(ConfigurableLocalSessionFactoryBean) { bean ->
    bean.autowire = 'byName'      // Autowiring behaviour. The other option is 'byType'. [a
    bean.initMethod = 'init'      // Sets the initialisation method to 'init'. [init-method
    bean.destroyMethod = 'destroy' // Sets the destruction method to 'destroy'. [destroy-met
    bean.scope = 'request'        // Sets the scope of the bean. [scope]
    dataSource = dataSource
    hibernateProperties = [ "hibernate.hbm2ddl.auto":"create-drop",
                            "hibernate.show_sql":true  ]
}
```

The strings in square brackets are the names of the equivalent bean attributes in Spring's XML definition.

Using BeanBuilder with Spring MVC

If you want to take advantage of BeanBuilder in a regular Spring MVC application you need to make sure the `grails-spring-<version>.jar` file is in your classpath. Once that is done you can need to set the following `<context-param>` values in your `/WEB-INF/web.xml` file:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.groovy</param-value>
</context-param>
<context-param>
    <param-name>contextClass</param-name>
    <param-value>org.codehaus.groovy.grails.commons.spring.GrailsWebApplicationContext</para
</context-param>
```

With that done you can then create a /WEB-INF/applicationContext.groovy file that does the rest:

```
beans {
    dataSource(org.apache.commons.dbcp.BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
    }
}
```

Loading Bean Definitions from the File System

You can use the `BeanBuilder` class to load external Groovy scripts that define beans using the same path matching syntax defined here. Example:

```
def bb = new BeanBuilder()
bb.loadBeans("classpath:*SpringBeans.groovy")
def applicationContext = bb.createApplicationContext()
```

Here the `BeanBuilder` will load all Groovy files on the classpath ending with `SpringBeans.groovy` and parse them into bean definitions. An example script can be seen below:

```
beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
    }
    sessionFactory(ConfigurableLocalSessionFactoryBean) {
        dataSource = dataSource
        hibernateProperties = [ "hibernate.hbm2ddl.auto":"create-drop",
                                "hibernate.show_sql":true  ]
    }
}
```

Adding Variables to the Binding (Context)

If you're loading beans from a script you can set the binding to use by creating a Groovy Binding object:

```
def binding = new Binding()
binding.foo = "bar"
def bb = new BeanBuilder()
bb.binding = binding
bb.loadBeans("classpath:*SpringBeans.groovy")
def ctx = bb.createApplicationContext()
```

14.4 The BeanBuilder DSL Explained

Using Constructor Arguments

Constructor arguments can be defined using parameters to each method that reside between the class of the bean and the last closure:

```
bb.beans {
    exampleBean(MyExampleBean, "firstArgument", 2) {
        someProperty = [1,2,3]
    }
}
```

Configuring the BeanDefinition (Using factory methods)

The first argument to the closure is a reference to the bean configuration instance, which you can use to configure factory methods and invoke any method on the AbstractBeanDefinition class:

```
bb.beans {
    exampleBean(MyExampleBean) { bean ->
        bean.factoryMethod = "getInstance"
        bean.singleton = false
        someProperty = [1,2,3]
    }
}
```

As an alternative you can also use the return value of the bean defining method to configure the bean:

```
bb.beans {
    def example = exampleBean(MyExampleBean) {
        someProperty = [1,2,3]
    }
    example.factoryMethod = "getInstance"
}
```

Using Factory beans

Spring defines the concept of factory beans and often a bean is created not from a class, but from one of these factories. In this case the bean has no class and instead you must pass the name of the factory bean to the bean:

191

```
bb.beans {
    myFactory(ExampleFactoryBean) {
        someProperty = [1,2,3]
    }
    myBean(myFactory) {
        name = "blah"
    }
}
```

Note in the example above instead of a class we pass a reference to the `myFactory` bean into the bean defining method. Another common task is provide the name of the factory method to call on the factory bean. This can be done using Groovy's named parameter syntax:

```
bb.beans {
    myFactory(ExampleFactoryBean) {
        someProperty = [1,2,3]
    }
    myBean(myFactory:"getInstance") {
        name = "blah"
    }
}
```

Here the `getInstance` method on the `ExampleFactoryBean` bean will be called in order to create the `myBean` bean.

Creating Bean References at Runtime
Sometimes you don't know the name of the bean to be created until runtime. In this case you can use a string interpolation to invoke a bean defining method dynamically:

```
def beanName = "example"
bb.beans {
    "${beanName}Bean"(MyExampleBean) {
        someProperty = [1,2,3]
    }
}
```

In this case the `beanName` variable defined earlier is used when invoking a bean defining method. Furthermore, because sometimes bean names are not known until runtime you may need to reference them by name when wiring together other beans. In this case using the `ref` method:

```
def beanName = "example"
bb.beans {
    "${beanName}Bean"(MyExampleBean) {
        someProperty = [1,2,3]
    }
    anotherBean(AnotherBean) {
        example = ref("${beanName}Bean")
    }
}
```

Here the example property of `AnotherBean` is set using a runtime reference to the `exampleBean`. The `ref` method can also be used to refer to beans from a parent `ApplicationContext` that is provided in the constructor of the `BeanBuilder`:

```
ApplicationContext parent = ...//
der bb = new BeanBuilder(parent)
bb.beans {
    anotherBean(AnotherBean) {
        example = ref("${beanName}Bean", true)
    }
}
```

Here the second parameter `true` specifies that the reference will look for the bean in the parent context.

Using Anonymous (Inner) Beans
You can use anonymous inner beans by setting a property of the bean to a block that takes an argument that is the bean type:

```
bb.beans {
    marge(Person.class) {
        name = "marge"
        husband =   { Person p ->
            name = "homer"
            age = 45
            props = [overweight:true, height:"1.8m"]
        }
        children = [bart, lisa]
    }
    bart(Person) {
        name = "Bart"
        age = 11
    }
    lisa(Person) {
        name = "Lisa"
        age = 9
    }
}
```

In the above example we set the `marge` bean's husband property to a block that creates an inner bean reference. Alternatively if you have a factory bean you can ommit the type and just use passed bean definition instead to setup the factory:

```
bb.beans {
    personFactory(PersonFactory.class)
    marge(Person.class) {
        name = "marge"
        husband =   { bean ->
            bean.factoryBean = "personFactory"
            bean.factoryMethod = "newInstance"
            name = "homer"
            age = 45
            props = [overweight:true, height:"1.8m"]
        }
        children = [bart, lisa]
    }
}
```

Abstract Beans and Parent Bean Definitions
To create an abstract bean definition define a bean that takes no class:

```
class HolyGrailQuest {
    def start() { println "lets begin" }
}
class KnightOfTheRoundTable {
    String name
    String leader
    KnightOfTheRoundTable(String n) {
        this.name = n
    }
    HolyGrailQuest quest
    def embarkOnQuest() {
        quest.start()
    }
}
def bb = new grails.spring.BeanBuilder()
bb.beans {
    abstractBean {
        leader = "Lancelot"
    }
    …
}
```

Here we define an abstract bean that sets that has a `leader` property with the value of `"Lancelot"`. Now to use the abstract bean set it as the parent of the child bean:

```
bb.beans {
    …
    quest(HolyGrailQuest)
    knights(KnightOfTheRoundTable, "Camelot") { bean ->
        bean.parent = abstractBean
        quest = quest
    }
}
```

> ⚠ When using a parent bean you must set the parent property of the bean before setting any other properties on the bean!

If you want an abstract bean that has a class you can do it this way:

```
def bb = new grails.spring.BeanBuilder()
bb.beans {
    abstractBean(KnightOfTheRoundTable) { bean ->
        bean.'abstract' = true
        leader = "Lancelot"
    }
    quest(HolyGrailQuest)
    knights("Camelot") { bean ->
        bean.parent = abstractBean
        quest = quest
    }
}
```

In the above example we create an abstract bean of type `KnightOfTheRoundTable` and use the bean argument to set it to abstract. Later we define a knights bean that has no class, but inherits the class from the parent bean.

Using Spring Namespaces
Since Spring 2.0, users of Spring have been granted easier access to key features via XML namespaces. With BeanBuilder you can use any Spring namespace by first declaring it:

```
xmlns context:"http://www.springframework.org/schema/context"
```

And then invoking a method that matches the names of the Spring namespace tag and its associated attributes:

```
context.'component-scan'( 'base-package' :"my.company.domain" )
```

You can do some useful things with Spring namespaces, such as looking up a JNDI resource:

```
xmlns jee:"http://www.springframework.org/schema/jee"
jee.'jndi-lookup'(id:"dataSource", 'jndi-name':"java:comp/env/myDataSource")
```

The example above will create a Spring bean with the identifier of `dataSource` by performing a JNDI lookup on the given JNDI name. With Spring namespaces you also get full access to all of the powerful AOP support in Spring from BeanBuilder. For example given the following two classes:

```
class Person {
 int age;
 String name;
 void birthday() {
      ++age;
 }
}
class BirthdayCardSender {
   List peopleSentCards = []
   void onBirthday(Person person) {
      peopleSentCards << person
   }
}
```

You can define an AOP aspect that uses a pointcut to detect whenever the `birthday()` method is called:

```
xmlns aop:"http://www.springframework.org/schema/aop"
fred(Person) {
    name = "Fred"
    age = 45
}
birthdayCardSenderAspect(BirthdayCardSender)
aop {
   config("proxy-target-class":true) {
       aspect( id:"sendBirthdayCard",ref:"birthdayCardSenderAspect" ) {
          after method:"onBirthday", pointcut: "execution(void ..Person.birthday()) and this
       }
   }
}
```

14.5 Property Placeholder Configuration

Grails supports the notion of property placeholder configuration through an extended version of Spring's PropertyPlaceholderConfigurer, which is typically useful when used in combination with externalized configuration. Settings defined in either ConfigSlurper scripts of Java properties files can be used as placeholder values for Spring configuration in `grails-app/conf/spring/resources.xml`. For example given the following entries in `grails-app/conf/Config.groovy` (or an externalized config):

```
database.driver="com.mysql.jdbc.Driver"
database.dbname="mysql:mydb"
```

You can then specify placeholders in `resources.xml` as follows using the familiar ${..} syntax:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"><value>${database.driver}</value></property>
    <property name="url"><value>jdbc:${database.dbname}</value></property>
 </bean>
```

14.6 Property Override Configuration

Grails supports setting of bean properties via [configuration](). This is often useful when used in combination with [externalized configuration]().

Essentially you define a `beans` block with the names of beans and their values:

```
beans {
    bookService {
        webServiceURL = "http://www.amazon.com"
    }
}
```

The general format is:

```
[bean name].[property name] = [value]
```

The same configuration in a Java properties file would be:

```
beans.bookService.webServiceURL=http://www.amazon.com
```

15. Grails and Hibernate

If [GORM (Grails Object Relational Mapping)](#) is not flexible enough for your liking you can alternatively map your domain class using Hibernate. To do this create a `hibernate.cfg.xml` file in the `grails-app/conf/hibernate` directory of your project and the corresponding HBM mapping xml files for your domain classes.

> ⚠ For more info on how to do this read the [documentation on mapping](#) on the Hibernate Website

This will allow you to map Grails domain classes onto a wider range of legacy systems and be more flexible in the creation of your database schema.
Grails also allows you to write your domain model in Java or re-use an existing domain model that has been mapped using Hibernate. All you have to do is place the necessary `hibernate.cfg.xml` file and corresponding mappings files in the `grails-app/conf/hibernate` directory.
Additionally, the good news is you will still be able to call all of the dynamic persistent and query methods allowed in [GORM](#)!

15.1 Mapping with Hibernate Annotations

Grails also supports creating domain classes mapped with Hibernate's Java 5.0 Annotations support.
Now to create an annotated class we simply create a new Java class in `src/java` and use the annotations defined as part of the EJB 3.0 spec (for more info on this see the [Hibernate Annotations Docs](#)):

```java
package com.books;
@Entity
public class Book {
    private Long id;
    private String title;
    private String description;
    private Date date;
    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}
```

Once that is done you need to register the class with the Hibernate `sessionFactory`, to do you need to add entries to the `grails-app/conf/hibernate/hibernate.cfg.xml` file as follows:

```xml
<!DOCTYPE hibernate-configuration SYSTEM
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <mapping package="com.books" />
        <mapping class="com.books.Book" />
    </session-factory>
</hibernate-configuration>
```

By default the `hibernate.cfg.xml` file is located in the `grails-app/conf/hibernate` directory. If you wish to change this you can do so by specifying an alternative location in `grails-app/conf/DataSource.groovy`:

```
hibernate {
    config.location = "file:/path/to/my/hibernate.cfg.xml"
}
```

Or even a list of locations:

```
hibernate {
    config.location = ["file:/path/to/one/hibernate.cfg.xml",
                       "file:/path/to/two/hibernate.cfg.xml"]
}
```

When Grails loads it will register the necessary dynamic methods with the class. To see what else you can do with a Hibernate domain class see the section on [Scaffolding](#).

15.2 Further Reading

Grails committer, Jason Rudolph, took the time to write many useful articles about using Grails with custom Hibernate mappings including:

- [Hoisting Grails to Your Legacy DB](#) - An excellent article about using Grails with Hibernate XML
- [Grails + EJB3 Domain Models](#) - Another great article about using Grails with EJB3-style annotated domain models

16. Scaffolding

Scaffolding allows you to auto-generate a whole application for a given domain class including:

- The necessary views
- Controller actions for create/read/update/delete (CRUD) operations

Enabling Scaffolding
The simplest way to get started with scaffolding is to enable scaffolding via the `scaffold` property. For the `Book` domain class, you need to set the `scaffold` property on a controller to true:

```
class BookController {
    def scaffold = true
}
```

The above works because the `BookController` follows the same naming convention as the `Book` domain class, if we wanted to scaffold a specific domain class you can reference the class directly in the scaffold property:

```
def scaffold = Author
```

With that done if you run this grails application the necessary actions and views will be auto-generated at runtime. The following actions are dynamically implemented by default by the runtime scaffolding mechanism:

- list
- show
- edit
- delete
- create
- save
- update

As well as this a CRUD interface will be generated. To access the interface in the above example simply go to `http://localhost:8080/app/book`
If you prefer to keep your domain model in Java and mapped with Hibernate you can still use scaffolding, simply import the necessary class and set the scaffold property to it.

Dynamic Scaffolding
Note that when using the scaffold property Grails does not use code templates, or code generation to achieve this so you can add your own actions to the scaffolded controller that interact with the scaffolded actions. For example, in the below example, `changeAuthor` redirects to the `show` action which doesn't actually exist physically:

```
class BookController {
    def scaffold = Book
    def changeAuthor = {
        def b = Book.get( params["id"] )
        b.author = Author.get( params["author.id"] )
        b.save()
        // redirect to a scaffolded action
        redirect(action:show)
    }
}
```

You can also override the scaffolded actions with your own actions if necessary:

```
class BookController {
    def scaffold = Book
    // overrides scaffolded action to return both authors and books
    def list = {
            [ "books" : Book.list(), "authors": Author.list() ]
    }
}
```

All of this is what is known as "dynamic scaffolding" where the CRUD interface is generated dynamically at runtime. Grails also supports "static" scaffolding which will be discussed in the following sections.

Customizing the Generated Views
The views that Grails generates have some form of intelligence in that they adapt to the Validation constraints. For example you can change the order that fields appear in the views simply by re-ordering the constraints in the builder:

```
def constraints = {
    title()
    releaseDate()
}
```

You can also get the generator to generate lists instead of text inputs if you use the `inList` constraint:

```
def constraints = {
    title()
    category(inList:["Fiction", "Non-fiction", "Biography"])
    releaseDate()
}
```

Or if you use the `range` constraint on a number:

```
def constraints = {
    age(range:18..65)
}
```

Restricting the size via a constraint also effects how many characters can be entered in the generated view:

```
def constraints = {
    name(size:0..30)
}
```

Generating Controllers & Views
The above scaffolding features are useful but in real world situations its likely that you will want to customize the logic and views. Grails allows you to generate a controller and the views used to create the above interface via the command line. To generate a controller type:

```
grails generate-controller Book
```

Or to generate the views type:

```
grails generate-views Book
```

Or to generate everything type:

```
grails generate-all Book
```

If you have a domain class in a package or are generating from a [Hibernate mapped class](#) remember to include the fully qualified package name:

```
grails generate-all com.bookstore.Book
```

Customizing the Scaffolding templates
The templates used by Grails to generate the controller and views can be customized by installing the templates with the [install-templates](#) command.

17. Deployment

Grails applications can be deployed in a number of ways, each of which has its pros and cons.

"grails run-app"
You should be very familiar with this approach by now, since it is the most common method of running an application during the development phase. An embedded Tomcat server is launched that loads the web application from the development sources, thus allowing it to pick up an changes to application files.
This approach is not recommended at all for production deployment because the performance is poor. Checking for and loading changes places a sizable overhead on the server. Having said that, `grails prod run-app` removes the per-request overhead and allows you to fine tune how frequently the regular check takes place.
Setting the system property "disable.auto.recompile" to `true` disables this regular check completely, while the property "recompile.frequency" controls the frequency. This latter property should be set to the number of seconds you want between each check. The default is currently 3.

"grails run-war"
This is very similar to the previous option, but Tomcat runs against the packaged WAR file rather than the development sources. Hot-reloading is disabled, so you get good performance without the hassle of having to deploy the WAR file elsewhere.

WAR file
When it comes down to it, current java infrastructures almost mandate that web applications are deployed as WAR files, so this is by far the most common approach to Grails application deployment in production. Creating a WAR file is as simple as executing the war command:

```
grails war
```

There are also many ways in which you can customise the WAR file that is created. For example, you can specify a path (either absolute or relative) to the command that instructs it where to place the file and what name to give it:

```
grails war /opt/java/tomcat-5.5.24/foobar.war
```

Alternatively, you can add a line to `grails-app/conf/BuildConfig.groovy` that changes the default location and filename:

```
grails.war.destFile = "foobar-prod.war"
```

Of course, any command line argument that you provide overrides this setting.
It is also possible to control what libraries are included in the WAR file, in case you need to avoid conflicts with libraries in a shared folder for example. The default behavior is to include in the WAR file all libraries required by Grails, plus any libraries contained in plugin "lib" directories, plus any libraries contained in the application's "lib" directory. As an alternative to the default behavior you can explicitly specify the complete list of libraries to include in the WAR file by setting the properties `grails.war.dependencies` and `grails.war.java5.dependencies` in Config.groovy to either lists of Ant include patterns or closures containing AntBuilder syntax. Closures are invoked from within an Ant "copy" step, so only elements like "fileset" can be included, whereas each item in a pattern list is included. Any closure or pattern assigned to the latter property will be included in addition to `grails.war.dependencies` only if you are running JDK 1.5 or above.
Be careful with these properties: if any of the libraries Grails depends on are missing, the application will almost certainly fail. Here is an example that includes a small subset of the standard Grails dependencies:

```
def deps = [
    "hibernate3.jar",
    "groovy-all-*.jar",
    "standard-${servletVersion}.jar",
    "jstl-${servletVersion}.jar",
    "oscache-*.jar",
    "commons-logging-*.jar",
    "sitemesh-*.jar",
    "spring-*.jar",
    "log4j-*.jar",
    "ognl-*.jar",
    "commons-*.jar",
    "xstream-1.2.1.jar",
    "xpp3_min-1.1.3.4.O.jar" ]
grails.war.dependencies = {
    fileset(dir: "libs") {
        deps.each { pattern ->
            include(name: pattern)
        }
    }
}
```

This example only exists to demonstrate the syntax for the properties. If you attempt to use it as is in your own application, the application will probably not work. You can find a list of dependencies required by Grails in the "dependencies.txt" file that resides in the root directory of the unpacked distribution. You can also find a list of the default dependencies included in WAR generation in the "War.groovy" script - see the "DEFAULT_DEPS" and "DEFAULT_J5_DEPS" variables.

The remaining two configuration options available to you are `grails.war.copyToWebApp` and `grails.war.resources`. The first of these allows you to customise what files are included in the WAR file from the "web-app" directory. The second allows you to do any extra processing you want before the WAR file is finally created.

```
// This closure is passed the command line arguments used to start the
// war process.
grails.war.copyToWebApp = { args ->
    fileset(dir:"web-app") {
        include(name: "js/**")
        include(name: "css/**")
        include(name: "WEB-INF/**")
    }
}
// This closure is passed the location of the staging directory that
// is zipped up to make the WAR file, and the command line arguments.
// Here we override the standard web.xml with our own.
grails.war.resources = { stagingDir, args ->
    copy(file: "grails-app/conf/custom-web.xml", tofile: "${stagingDir}/WEB-INF/web.xml")
}
```

Application servers

Ideally you should be able to simply drop a WAR file created by Grails into any application server and it should work straight away. However, things are rarely ever this simple. The Grails website contains an up-to-date list of application servers that Grails has been tested with, along with any additional steps required to get a Grails WAR file working.