GRAILS

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, Luke Daley, Burt Beckwith
Version: 2.0.0

Table of Contents

1 Introduction

Java web development as it stands today is dramatically more complicated than it needs to be. Most mo
web frameworks in the Java space are over complicated and don't embrace the Don't Repeat Yourself (D
principles.

Dynamic frameworks like Rails, Django and TurboGears helped pave the way to a more modern wa
thinking about web applications. Grails builds on these concepts and dramatically reduces the complexit
building web applications on the Java platform. What makes it different, however, is that it does so
building on already established Java technologies like Spring and Hibernate.

Grails is a full stack framework and attempts to solve as many pieces of the web development puzzle thro
the core technology and its associated plugins. 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 Dor
Specific Languages (DSLs)

This documentation will take you through getting started with Grails and building web applications with
Grails framework.
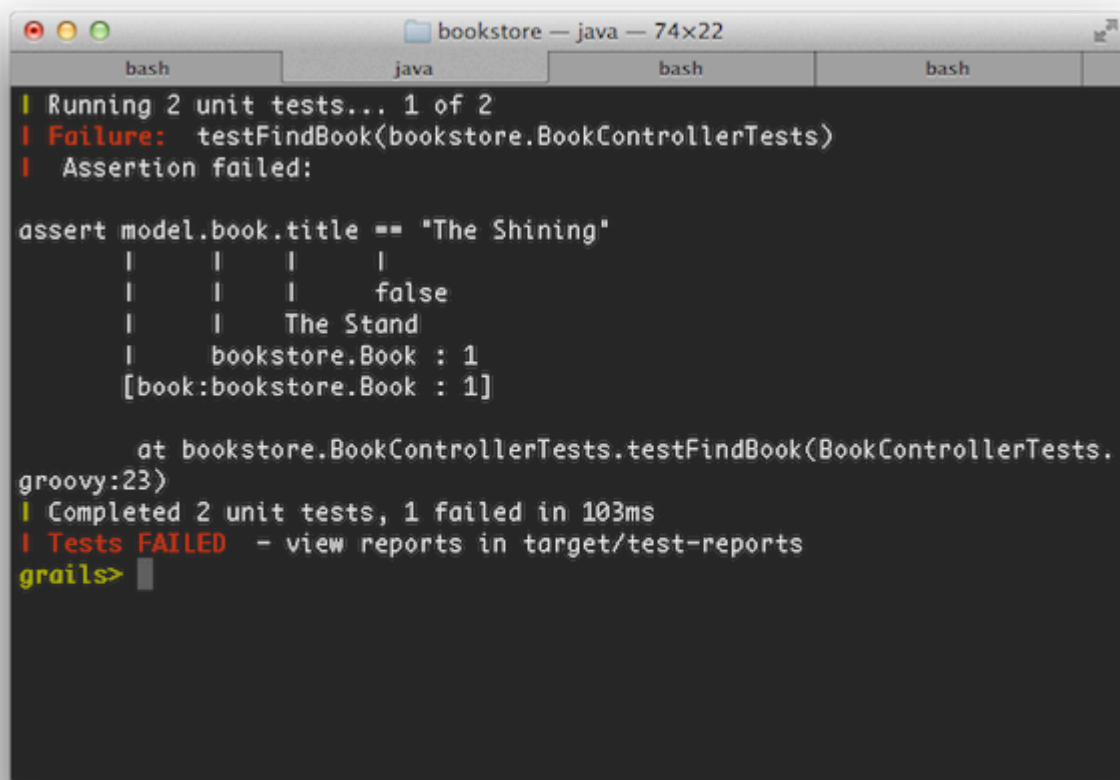
1.1 What's new in Grails 2.0?

This section covers the new features that are present in 2.0 and is broken down into sections covering the b
system, core APIs, the web tier, persistence enhancements and improvements in testing. Note there are n
more small enhancements and improvements, these sections just cover some of the highlights.

1.1.1 Development Environment Features

Interactive Mode and Console Enhancements

Grails 2.0 features brand new console output that is more concise and user friendly to consume. An exar
of the new output when running tests can be seen below:

In general Grails makes its best effort to display update information on a single line and only present information that is crucial. This means that while in previous versions of Grails the [war] command produces many lines of output, in Grails 2.0 only 1 line of output is produced:
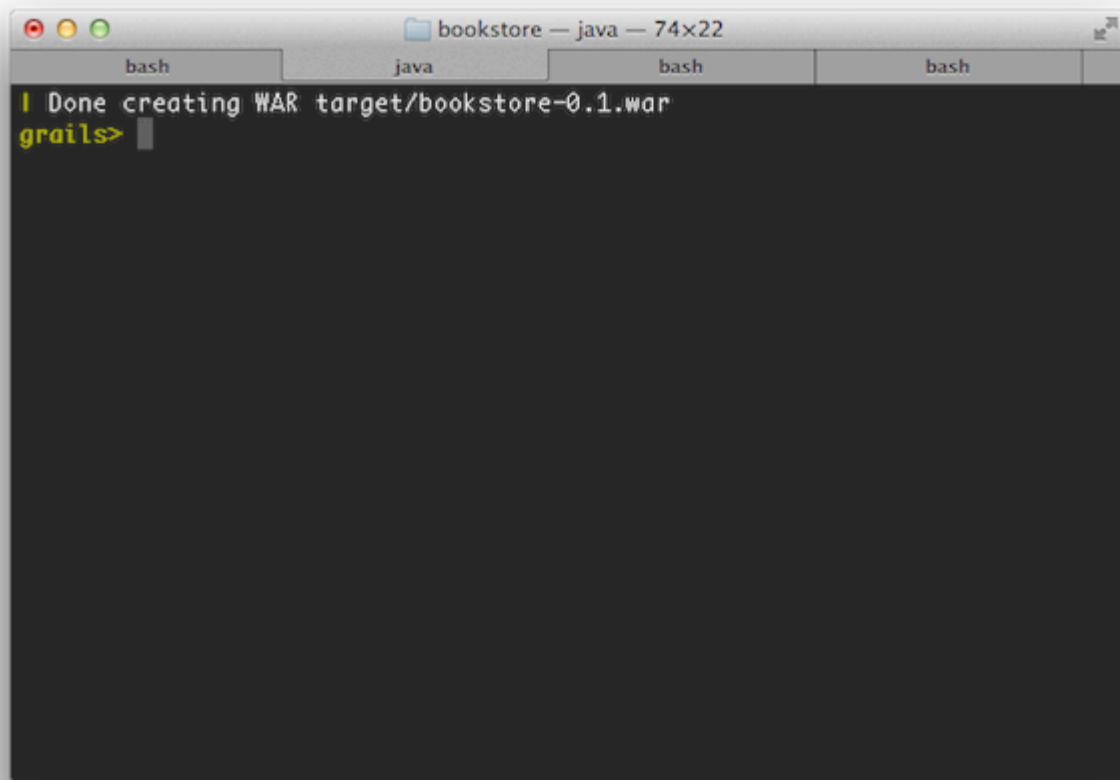
In addition simply typing 'grails' at the command line activates the new interactive mode which features ? completion, command history and keeps the JVM running to ensure commands execute much quicker otherwise

For more information on the new features of the console refer to the section of the user guide that covers console and interactive mode.

Reloading Agent

Grails 2.0 reloading mechanism no longer uses class loaders, but instead uses a JVM agent to reload cha to class files. This results in greatly improved reliability when reloading changes and also ensures that class files stored in disk remain consistent with the class files loaded in memory, which reduces the nee run the clean command.

New Test Report and Documentation Templates

There are new templates for displaying test results that are clearer and more user friendly than the prev reports:

In addition, the Grails documentation engine has received a facelift with a new template for presenting G
application and plugin documentation:

See the section on the **documentation engine** for more usage info.

Use a TOC for Project Docs

The old documentation engine relied on you putting section numbers into the gdoc filenames. Altho
convenient, this effectively made it difficult to restructure your user guide by inserting new chapters
sections. In addition, any such restructuring or renaming of section titles resulted in breaking changes to
URLs.

You can now use logical names for your gdoc files and define the structure and section titles in a YA
table-of-contents file, as described in the section on the **documentation engine**. The logical names appe
the URLs, so as long as you don't change those, your URLs will always remain the same no matter how n
restructuring or changing of titles you do.

Grails 2.0 even provides a **migrate-docs** command to aid you in migrating existing gdoc user guides.

Enhanced Error Reporting and Diagnosis

Error reporting and problem diagnosis has been greatly improved with a new errors view that analyses s
traces and recursively displays problem areas in your code:

GRAILS

Error 500: Internal Server Error

URI: /bookstore/book/find
Class: groovy.lang.MissingPropertyException
Message: No such property: titl for class: bookstore.BookService

Around line 6 of *grails-app/services/bookstore/BookService.groovy*

```
3: class BookService {
4:
5:     Book findByTitle(String title) {
6:         Book.findByTitle(titl)
7:     }
8: }
```

Around line 10 of *grails-app/controllers/bookstore/BookController.groovy*

```
7:      def bookService
8:      def find() {
9:
10:         def b = bookService.findByTitle(params.title)
11:
12:         [book:b]
13:     }
```

Trace

```
    Line | Method
->>    6 | findByTitle in BookService.groovy
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
|     10 | find          in BookController.groovy
|    886 | runTask . .  in java.util.concurrent.ThreadPoolExecutor$Worker
|    908 | run           in      ''
^    680 | run . . . . . in java.lang.Thread
```

In addition stack trace filtering has been further enhanced to display only relevant trace information:

```
Line | Method
->>   9 | getValue     in Book.groovy
- - - - - - - - - - - - - - - - - - - - - - - - - -
|      7 | getBookValue in BookService.groovy
|    886 | runTask . .  in ThreadPoolExecutor.java
|    908 | run           in      ''
^    662 | run . . . .  in Thread.java
```

H2 Database and Console

Grails 2.0 now uses the H2 database instead of HSQLDB, and enables the H2 database consol
development mode (at the URI /dbconsole) so that the in-memory database can be easily queried from
browser:

N | ⚡ | ☑ Auto commit ↩ / | Max rows: 1000 ⇅ | ▶ ■ | ⬆ | Auto complete Normal ⇅ ?

jdbc:h2:mem:devDb

☐ BOOK
  ⊞ ID
  ⊞ VERSION
  ⊞ TITLE
  ⊞ Indexes
⊞ INFORMATION_SCHEMA
⊞ Sequences
⊞ Users
ⓘ H2 1.2.147 (2010-11-21)

Run (Ctrl+Enter) Clear SQL statement:

select * from book;
| ID | VERSION | TITLE |
(no rows, 3 ms)

Edit
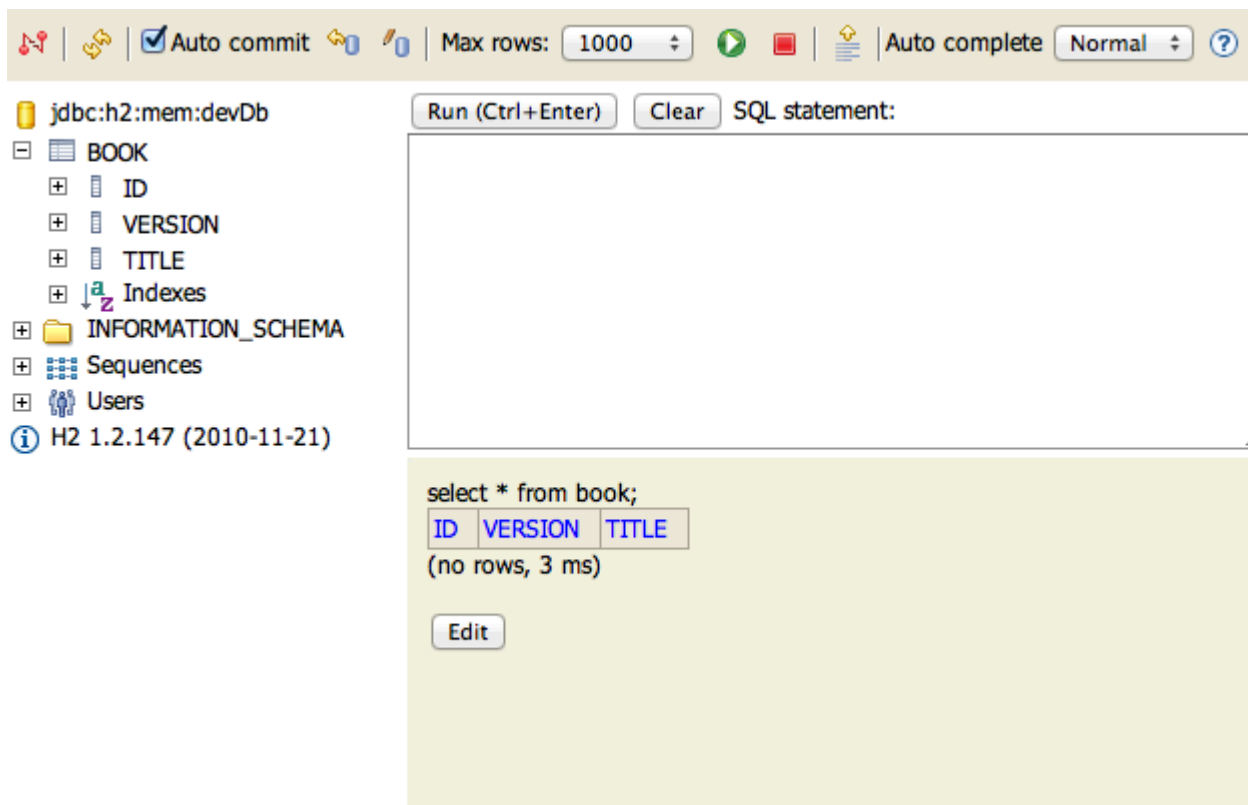
Plugin Usage Tracking

To enhance community awareness of the most popular plugins an opt-in plugin usage tracking system been included where users can participate in providing feedback to the plugin community on which plu are most popular.

This will help drive the roadmap and increase support of key plugins while reducing the need to support or less popular plugins thus helping plugin development teams focus their efforts.

Dependency Resolution Improvements

There are numerous improvements to dependency resolution handling via Ivy including:

- Grails now makes a best effort to cache the previous resolve and avoid resolving again unless you ch `BuildConfig.groovy`.

- Plugins dependencies now appear in the dependency report generated by `gra dependency-report`

- Plugins published with the release plugin now publish their transitive plugin dependencies in generated POM which are later resolved.

- It is now possible to customize the ivy cache directory via `BuildConfig.groovy`

```
grails.project.dependency.resolution = {
    cacheDir "target/ivy-cache"
}
```

- You can change the ivy cache directory for all projects via `settings.groovy`

```
grails.dependency.cache.dir = "${userHome}/.ivy2/cache"
```

- It is now possible to completely disable resolution from inherited repositories (repositories defined other plugins):

```
grails.project.dependency.resolution = {
repositories {
        inherits false // Whether to inherit repository definitions from plugins
        …
    }
    …
}
```

- It is now possible to easily disable checksum validation errors:

```
grails.project.dependency.resolution = {
    checksums false // whether to verify checksums or not
}
```

1.1.2 Core Features

Binary Plugins

Grails plugins can now be packaged as JAR files and published to standard maven repositories. This works for GSP and static resources (with resources plugin 1.0.1). See the section on Binary plugins for information.

Groovy 1.8

Grails 2.0 comes with Groovy 1.8 which includes many new features and enhancements

Spring 3.1 Profile Support

Grails' existing environment support has been bridged into the Spring 3.1 profile support. For example running with a custom Grails environment called "production", a Spring profile of "production" is activate that you can use Spring's bean configuration APIs to configure beans for a specific profile.

1.1.3 Web Features

Controller Actions as Methods

It is now possible to define controller actions as methods instead of using closures as in previous version Grails. In fact this is now the preferred way of expressing an action. For example:

16

```
// action as a method
def index() {

}
// action as a closure
def index = {

}
```

Binding Primitive Method Action Arguments

It is now possible to bind form parameters to action arguments where the name of the form element mat
the argument name. For example given the following form:

```
<g:form name="myForm" action="save">
    <input name="name" />
    <input name="age" />
</g:form>
```

You can define an action that declares arguments for each input and automatically converts the paramete
the appropriate type:

```
def save(String name, int age) {
    // remaining
}
```

Static Resource Abstraction

A new static resource abstraction is included that allows declarative handling of JavaScript, CSS and in
resources including automatic ordering, compression, caching and gzip handling.

Servlet 3.0 Async Features

Grails now supports Servlet 3.0 including the Asynchronous programming model defined by the specificat

```
def index() {
    def ctx = startAsync()
    ctx.start {
        new Book(title:"The Stand").save()
        render template:"books", model:[books:Book.list()]
        ctx.complete()
    }
}
```

Link Generation API

A general purpose `LinkGenerator` class is now available that is usable anywhere within a G
application and not just within the context of a controller. For example if you need to generate links
service or an asynchronous background job outside the scope of a request:

```
LinkGenerator grailsLinkGenerator

def generateLink() {
    grailsLinkGenerator.link(controller:"book", action:"list")
}
```

Page Rendering API

Like the `LinkGenerator` the new `PageRenderer` can be used to render GSP pages outside the scope
a web request, such as in a scheduled job or web service. The `PageRenderer` class features a very sir
API to the `render` method found within controllers:

```
grails.gsp.PageRenderer groovyPageRenderer

void welcomeUser(User user) {
    def contents = groovyPageRenderer.render(view:"/emails/welcomeLetter",
model:[user: user])
    sendEmail {
        to user.email
        body contents
    }
}
```

The `PageRenderer` service also allows you to pre-process GSPs into HTML templates:

```
new File("/path/to/welcome.html").withWriter { w ->
    groovyPageRenderer.renderTo(view:"/page/content", w)
}
```

Filter Exclusions

Filters may now express controller, action and uri exclusions to offer more options for expressing to w
requests a particular filter should be applied.
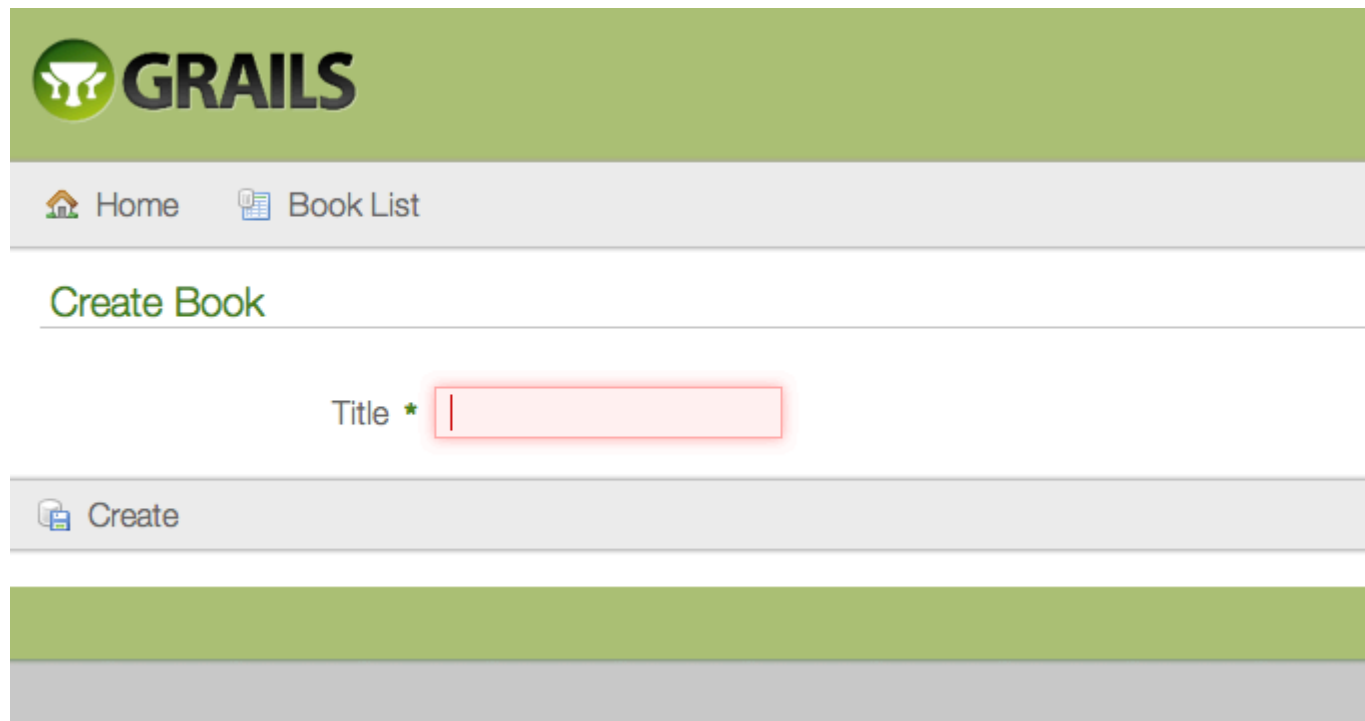
```
filter1(actionExclude: 'log*') {
    before = {
        // …
    }
}
filter2(controllerExclude: 'auth') {
    before = {
        // …
    }
}
filter3(uriExclude: '/secure*') {
    before = {
        // …
    }
}
```

Performance Improvements

Performance of GSP page rendering has once again been improved by optimizing the GSP compiler to in method calls where possible.

HTML5 Scaffolding

There is a new HTML5-based scaffolding UI:

jQuery by Default

The jQuery plugin is now the default JavaScript library installed into a Grails application. For backw compatibility a [Prototype plugin](#) is available. Refer to the [documentation](#) on the Prototype plugin installation instructions.

Easy Date Parsing

A new `date` method has been added to the `params` object to allow easy, null-safe parsing of dates:

```
def val = params.date('myDate', 'dd-MM-yyyy')

// or a list for formats
def val = params.date('myDate', ['yyyy-MM-dd', 'yyyyMMdd', 'yyMMdd'])

// or the format read from messages.properties via the key 'date.myDate.format'
def val = params.date('myDate')
```

Customizable URL Formats

The default URL Mapping mechanism supports camel case names in the URLs. The default URL accessing an action named `addNumbers` in a controller named `MathHelperController` would something like `/mathHelper/addNumbers`. Grails allows for the customization of this pattern provides an implementation which replaces the camel case convention with a hyphenated convention would support URLs like `/math-helper/add-numbers`. To enable hyphenated URLs assign a valu "hyphenated" to the `grails.web.url.converter` property in `grails-app/conf/Config.groovy`.

```
// grails-app/conf/Config.groovy
grails.web.url.converter = 'hyphenated'
```

Arbitrary strategies may be plugged in by providing a class which implements the [UrlConverter](#) interface adding an instance of that class to the Spring application context with the bean name `grails.web.UrlConverter.BEAN_NAME`. If Grails finds a bean in the context with that name, it be used as the default converter and there is no need to assign a value to th `grails.web.url.converter` config property.

```
// src/groovy/com/myapplication/MyUrlConverterImpl.groovy
package com.myapplication
class MyUrlConverterImpl implements grails.web.UrlConverter {
String toUrlElement(String propertyOrClassName) {
        // return some representation of a property or class name that should be
used in URLs…
    }
}
```

```
// grails-app/conf/spring/resources.groovy
beans = {
    "${grails.web.UrlConverter.BEAN_NAME}"(com.myapplication.MyUrlConverterImpl)
}
```

1.1.4 Persistence Features

The GORM API

The GORM API has been formalized into a set of classes (`GormStaticApi`, `GormInstanceApi` `GormValidationApi`) that get statically wired into every domain class at the byte code level. The resu better code completion for IDEs, better integration with Java and the potential for more GO implementations for other types of data stores.

Detached Criteria and Where Queries

Grails 2.0 features support for [DetachedCriteria](#) which are criteria queries that are not associated with session or connection and thus can be more easily reused and composed:

```
def criteria = new DetachedCriteria(Person).build {
    eq 'lastName', 'Simpson'
}
def results = criteria.list(max:4, sort:"firstName")
```

To support the addition of [DetachedCriteria](#) queries and encourage their use a new `where` method and |
has been introduced to greatly reduce the complexity of criteria queries:

```
def query = Person.where {
    (lastName != "Simpson" && firstName != "Fred") || (firstName == "Bart" && age
> 9)
}
def results = query.list(sort:"firstName")
```

See the documentation on [DetachedCriteria](#) and [Where Queries](#) for more information.

New findOrCreate and findOrSave Methods

Domain classes have support for the findOrCreateWhere, findOrSaveWhere, findOrCreateBy
findOrSaveBy query methods which behave just like findWhere and findBy methods except that they sh
never return null. If a matching instance cannot be found in the database then a new instance is crea
populated with values represented in the query parameters and returned. In the case of findOrSaveWhere
findOrSaveBy, the instance is saved before being returned.

```
def book = Book.findOrCreateWhere(author: 'Douglas Adams', title: "The Hitchiker'
Guide To The Galaxy")
def book = Book.findOrSaveWhere(author: 'Daniel Suarez', title: 'Daemon')
def book = Book.findOrCreateByAuthorAndTitle('Daniel Suarez', 'Daemon')
def book = Book.findOrSaveByAuthorAndTitle('Daniel Suarez', 'Daemon')
```

Abstract Inheritance

GORM now supports abstract inheritance trees which means you can define queries and associations lin
to abstract classes:

```
abstract class Media {
    String title
    …
}
class Book extends Media {
}
class Album extends Media {

}
class Account {
    static hasMany = [purchasedMedia:Media]
}

..

def allMedia = Media.list()
```

Multiple Data Sources Support

It is now possible to define multiple datasources in `DataSource.groovy` and declare one or r
datasources a particular domain uses by default:

```
class ZipCode {

String code

static mapping = {
    datasource 'ZIP_CODES'
  }
}
```

If multiple datasources are specified for a domain then you can use the name of a particular datasource
namespace in front of any regular GORM method:

```
def zipCode = ZipCode.auditing.get(42)
```

For more information see the section on <u>Multiple Data Sources</u> in the user guide.

Database Migrations

A new <u>database migration plugin</u> has been designed and built for Grails 2.0 allowing you to apply migrat
to your database, rollback changes and diff your domain model with the current state of the database.

Database Reverse Engineering

A new <u>database reverse engineering</u> plugin has been designed and built for Grails 2.0 that allows yo
generate a domain model from an existing database schema.

Hibernate 3.6

Grails 2.0 is now built on Hibernate 3.6

Bag Collections
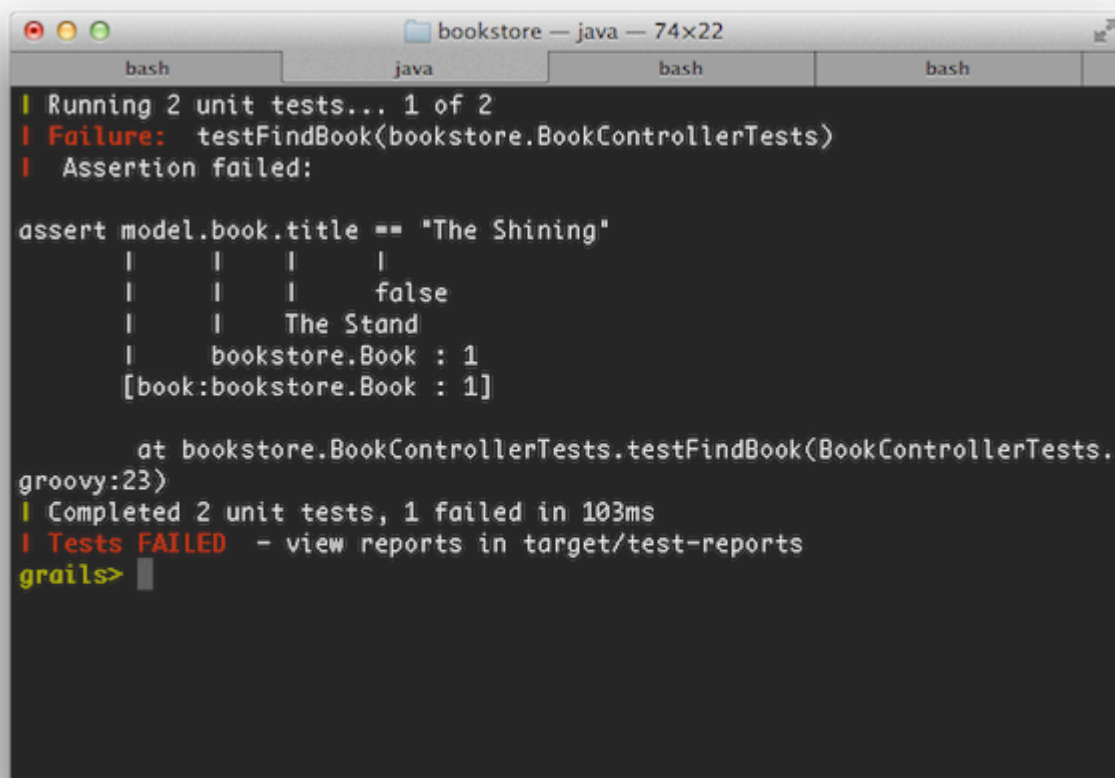
You can now use Hibernate <u>Bag</u>s for mapped collections to avoid the memory and performance issue
loading large collections to enforce `Set` uniqueness or `List` order.

For more information see the section on <u>Sets, Lists and Maps</u> in the user guide.

1.1.5 Testing Features

New Unit Testing Console Output

Test output from the test-app command has been improved:

New Unit Testing API

There is a new unit testing API based on mixins that supports JUnit 3, 4 and Spock style tests (with Spock and above). Example:

```
import grails.test.mixin.TestFor

@TestFor(SimpleController)
class SimpleControllerTests  {
    void testIndex() {
        controller.home()

assert view == "/simple/homePage"
        assert model.title == "Hello World"
    }
}
```

The documentation on testing has also been re-written around this new framework.

Unit Testing GORM

A new in-memory GORM implementation is present that supports many more features of the GORM making unit testing of criteria queries, named queries and other previously unsupported methods possible.

Faster Unit Testing with Interactive Mode

The new interactive mode (activated by typing 'grails') greatly improves the execution time of running and integration tests.

Unit Test Scaffolding

A unit test is now generated for scaffolded controllers.

2 Getting Started

2.1 Installation Requirements

Before installing Grails you will as a minimum need a Java Development Kit (JDK) installed version 1. above and environment variable called `JAVA_HOME` pointing to the location of this installation. On s platforms (for example OS X) the Java installation is automatically detected. However in many cases you want to manually configure the location of Java. For example:

```
export JAVA_HOME=/Library/Java/Home
export PATH="$PATH:$JAVA_HOME/bin"
```

Note that although JDK 1.6 is required to use Grails at development time it is possible to deploy Grails to 1.5 VMs by setting the `grails.project.source.level` and `grails.project.target.le` settings to "1.5" in `grails-app/conf/BuildConfig.groovy`:

```
grails.project.source.level = 1.5
grails.project.target.level = 1.5
```

In addition, Grails supports Servlet versions 2.5 and above. If you wish to use newer features of the Se API (such as 3.0) you should configure the `grails.servlet.version` in `BuildConfig.gro` appropriately:

```
grails.servlet.version = "3.0"
```

2.2 Downloading and Installing

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

- [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 follow `export GRAILS_HOME=/path/to/grails` to your profile
  - On Windows this is typically a matter of setting an environment variable under `Computer/Advanced/Environment Variables`
- Then add the `bin` directory to your `PATH` variable:
  - On Unix/Linux based systems this can be done by adding `exp PATH="$PATH:$GRAILS_HOME/bin"` to your profile
  - On Windows this is done by modifying the `Path` environment variable under `Computer/Advanced/Environment Variables`

If Grails is working correctly you should now be able to type `grails -version` in the terminal win and see output similar to this:

```
Grails version: 2.0.0
```

2.3 Upgrading from previous versions of Grails

Although the Grails development team have tried to keep breakages to a minimum there are a number of it to consider when upgrading a Grails 1.0.x, 1.1.x, 1.2.x, or 1.3.x applications to Grails 2.0. The major cha are described in more detail below, but here's a brief summary of what you might encounter when upgra from Grails 1.3.x:

- Logging by convention packages have changed, so you may not see the logging output you exp Update your logging configuration as described below.

- HSQLDB has been replaced with H2 as default in-memory database. If you use the former, either cha your data source to H2 or add HSQLDB as a runtime dependency.

- The `release-plugin` command has been removed. You must now install the [Release plugin](#) and its [`publish-plugin` command](#) instead.

- The `redirect()` method no longer commits the response, so `isCommitted()` will return `fals` you use that method, then call `request.isRedirected()` instead.

- The `redirect()` method now uses the grails.serverURL config setting to generate the redirect U You may need to remove the setting, particularly from the development and test environments.

- `withFormat()` no longer takes account of the request content type. If you want to do something b on the request content type, use `request.withFormat()`.

- Adaptive AJAX tags using Prototype will break. In this situation you must install the new Proto plugin.

- If you install Resources (or it is installed automatically), tags like `<g:javascript>` won't v anything to the page until you add the `<r:layoutResources/>` tags to your layout.

- Resources adds a '/static' URL, so you may have to update your access control rules accordingly.

- Some plugins may fail to install because one or more of their dependencies can not be found. If happens, the plugin probably has a custom repository URL that you need to add to your proje `BuildConfig.groovy`.

- The behaviour of abstract domain classes has changed, so if you use them you will either have to n the abstract classes to 'src/groovy' or migrate your database schema and data.

- Criteria queries default to INNER_JOIN for associations rather than OUTER_JOIN. This may a some of your result data.

- Constraints declared for non-existent properties will now throw an exception.

- Public methods in controllers will now be treated as actions. If you don't want this, make them prote or private.

- The new unit testing framework won't work with the old `GrailsUnitTestCase` class hierar Your old tests will continue to work, but if you wish to use the new annotations, do not extend any of `*UnitTestCase` classes.

- Output from Ant tasks is now hidden by default. If your scripts are using `ant.echo(`, `ant.input()`, etc. you might want to use alternative mechanisms for output.

- Domain properties of type java.net.URL may no longer work with your existing data. The serialisa mechanism for them appears to have changed. Consider migrating your data and domain mode String.

- The Ivy cache location has changed. If you want to use the old location, configure the appropriate gl setting (see below) but be aware that you may run into problems running Grails 1.3.x and 2.x pro side by side.

- With new versions of various dependencies, some APIs (such as the Servlet API) may have change you have code that implements any of those APIs, you will need to update it. Problems will typi manifest as compilation errors.

- The following deprecated classes have been removed: `grails.web.JsonBuilder` a `grails.web.OpenRicoBuilder`.

Upgrading from Grails 1.3.x

Changes to web.xml template

If you have customized the web.xml provided by `grails install-templates` then you will nee update this customized template with the latest version provided by Grails. Failing to do so will lead `ClassNotFoundException` for the `org.codehaus.groovy.grails.web.util.Log4jConfigListener` class.

Groovy 1.8 Changes

Groovy 1.8 is a little stricter in terms of compilation so you may be required to fix compilation errors in application that didn't occur under Grails 1.3.x.

Groovy 1.8 also requires that you update many of the libraries that you may be using in your applica Libraries known to require an upgrade include:

- Spock

- Geb

- GMock (upgrade unavailable as of this writing)

HSQLDB Has Been Replaced With H2

HSQLDB is still bundled with Grails but is not configured as a default runtime dependency. Upgrade opt include replacing HSQLDB references in DataSource.groovy with H2 references or adding HSQLDB runtime dependency for the application.

If you want to run an application with different versions of Grails, it's simplest to add HSQLDB as a run dependency, which you can do in BuildConfig.groovy:

```
grails.project.dependency.resolution = {
    inherits("global") {
    }
    repositories {
        grailsPlugins()
        grailsHome()
        grailsCentral()
    }

dependencies {
        // Add HSQLDB as a runtime dependency
        runtime 'hsqldb:hsqldb:1.8.0.10'
    }
}
```

A default DataSource.groovy which is compatible with H2 looks like this:

```
dataSource {
    driverClassName = "org.h2.Driver"
    username = "sa"
    password = ""
}
// environment specific settings
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'create-drop','update'
            url = "jdbc:h2:mem:devDb"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:prodDb"
        }
    }
}
```

Another significant difference between H2 and HSQLDB is in the handling of `byte[]` domain c
properties. HSQLDB's default BLOB size is large and so you typically don't need to specify a maximum
But H2 defaults to a maximum size of 255 bytes! If you store images in the database, the saves are likel
fail because of this. The easy fix is to add a `maxSize` constraint to the `byte[]` property:

```
class MyDomain {
    byte[] data

static constraints = {
        data maxSize: 1024 * 1024 * 2 // 2MB
    }
}
```

This constraint influences schema generation, so in the above example H2 will have the `data` column se
`BINARY(2097152)` by Hibernate.

Abstract Inheritance Changes

In previous versions of Grails abstract classes in `grails-app/domain` were not treated as persistent.
is no longer the case and has a significant impact on upgrading your application. For example consider
following domain model in a Grails 1.3.x application:

```
abstract class Sellable {

}
class Book extends Sellable {

}
```

In Grails 1.3.x you would get a BOOK table and the properties from the `Sellable` class would be st
within the `BOOK` table. However, in Grails 2.x you will get a `SELLABLE` table and the def
table-per-hierarchy inheritance rules apply with all properties of the `Book` stored in the `SELLABLE` table.

You have two options when upgrading in this scenario:

1. Move the abstract `Sellable` class into the src/groovy package. If the `Sellable` class is in
   `src/groovy` directory it will no longer be regarded as persistent.

2. Use the [database migration](#) plugin to apply the appropriate changes to the database (typically renam
   the table to the root abstract class of the inheritance tree).

Criteria Queries Default to INNER JOIN

The previous default of LEFT JOIN for criteria queries across associations is now INNER JOIN.

Invalid Constraints Now Thrown an Exception

Previously if you defined a constraint on a property that doesn't exist no error would be thrown:

```
class Person {
    String name
    static constraints = {
        bad nullable:false // invalid property, no error thrown
    }
}
```

Now the above code will result in an exception

Logging By Convention Changes

The packages that you should use for Grails artifacts have mostly changed. In particular:

- `service` -> `services`
- `controller` -> `controllers`
- `tagLib` -> `taglib` (case change)
- `bootstrap` -> `conf`
- `dataSource` -> `conf`

You can find out more about logging by convention in the [main part](#) of the user guide, under "Configu loggers". This change is a side-effect of injecting the `log` property into artefacts at compile time.

jQuery Replaces Prototype

The Protoype Javascript library has been removed from Grails core and now new Grails applications have jQuery plugin configured by default. This will only impact you if you are using Prototype with the adap AJAX tags in your application, e.g. <g:remoteLink/> etc, because those tags will break as soon as upgrade.

To resolve this issue, simply install the [Prototype plugin](#) in your application. You can also remove prototype files from your `web-app/js/prototype` directory if you want.

The Resources Plugin

The Resources plugin is a great new feature of Grails that allows you to manage static web resources b than before, but you do need to be aware that it adds an extra URL at `/static`. If you have access contr your application, this may mean that the static resources require an authenticated user to load them! Make your access rules take account of the `/static` URL.

Controller Public Methods

As of Grails 2.0, public methods of controllers are now treated as actions in addition to actions define traditional Closures. If you were relying on the use of methods for privacy controls or as helper methods this could result in unexpected behavior. To resolve this issue you should mark all methods of application that are not to be exposed as actions as `private` methods.

The redirect Method

The [redirect](#) method no longer commits the response. The result of this is code that relies of this behavior break in 2.0. For example:

```
redirect action: "next"
if (response.committed) {
    // do something
}
```

In this case in Grails 1.3.x and below the `response.committed` property would return true and the block will execute. In Grails 2.0 this is no longer the case and you should instead use the n `isRedirected()` method of the `request` object:

```
    redirect action: "next"
    if (request.redirected) {
        // do something
    }
```

Another side-effect of the changes to the redirect method is that it now always uses t
grails.serverURL configuration option if it's set. Previous versions of Grails included default value
all the environments, but when upgrading to Grails 2.0 those values more often than not break redirection
we recommend you remove the development and test settings for grails.serverURL or replace t
with something appropriate for your application.

Content Negotiation

As of Grails 2.0 the [withFormat](#) method of controllers no longer takes into account the request content
(dictated by the CONTENT_TYPE header), but instead deals exclusively with the response content
(dictated by the ACCEPT header or file extension). This means that if your application has code that relie
reading XML from the request using withFormat this will no longer work:

```
def processBook() {
    withFormat {
        xml {
            // read request XML
        }
        html {
            // read request parameters
        }
    }
}
```

Instead you use the withFormat method provided on the request object:

```
def processBook() {
    request.withFormat {
        xml {
            // read request XML
        }
        html {
            // read request parameters
        }
    }
}
```

Unit Test Framework

Grails 2 introduces a new unit testing framework that is simpler and behaves more consistently than the
one. The old framework based on the GrailsUnitTestCase class hierarchy is still available
backwards compatibility, but it does not work with the new annotations.

Migrating unit tests to the new approach is non-trivial, but recommended. Here are a set of mappings from
old style to the new:

1. Remove `extends *UnitTestCase` and add a `@TestFor` annotation to the class if you're testi core artifact (controller, tag lib, domain class, etc.) or `@TestMixin(GrailsUnitTestMixin)` non-core artifacts and non-artifact classes.

2. Add `@Mock` annotation for domain classes that must be mocked and use `new MyDomain().sav` in place of `mockDomain()`.

3. Replace references to `mockRequest`, `mockResponse` and `mockParams` with `reques` `response` and `params`.

4. Remove references to `renderArgs` and use the `view` and `model` properties for view rendering `response.text` for all others.

5. Replace references to `redirectArgs` with `response.redirectedUrl`. The latter takes account the URL mappings as is a string URL rather than a map of `redirect()` arguments.

6. The `mockCommandObject()` method is no longer needed as Grails automatically detects whethe action requires a command object or not.

There are other differences, but these are the main ones. We recommend that you read the chapter on tes thoroughly to understand everything that has changed.

Note that the Grails annotations don't need to be imported in your test cases to run them from the comn line, but your IDE may need them. So, here are the relevant classes with packages:

- `grails.test.mixin.TestFor`
- `grails.test.mixin.TestMixin`
- `grails.test.mixin.Mock`
- `grails.test.mixin.support.GrailsUnitTestMixin`
- `grails.test.mixin.domain.DomainClassUnitTestMixin`
- `grails.test.mixin.services.ServiceUnitTestMixin`
- `grails.test.mixin.web.ControllerUnitTestMixin`
- `grails.test.mixin.web.FiltersUnitTestMixin`
- `grails.test.mixin.web.GroovyPageUnitTestMixin`
- `grails.test.mixin.web.UrlMappingsUnitTestMixin`
- `grails.test.mixin.webflow/WebFlowUnitTestMixin`

Note that you're only ever likely to use the first two explicitly. The rest are there for reference.

Command Line Output

Ant output is now hidden by default to keep the noise in the terminal to a minimum. That means if you `ant.echo` in your scripts to communicate messages to the user, we recommend switching to an alterna mechanism.

For status related messages, you can use the event system:

```
event "StatusUpdate", ["Some message"]
event "StatusFinal",  ["Some message"]
event "StatusError",  ["Some message"]
```

For more control you can use the grailsConsole script variable, which gives you access to an instanc
GrailsConsole. In particular, you can log information messages with log() or info(), errors and warn
with error() and warning(), and request user input with userInput().

Custom Plugin Repositories

Many plugins have dependencies, both other plugins and straight JAR libraries. These are often locate
Maven Central, the Grails core repository or the Grails Central Plugin Repository in which case applicat
are largely unaffected if they upgrade to Grails 2. But sometimes such dependencies are located elsewhere
Grails must be told where they can be found.

Due to changes in the way Grails handles the resolution of dependencies, Grails 2.0 requires you to add
such custom repository locations to your project if an affected plugin is to install properly.

Ivy Cache Location Has Changed

The default Ivy cache location for Grails has changed. If the thought of yet another cache of JARs on y
disk horrifies you, then you can change this in your settings.groovy:

```
grails.dependency.cache.dir = "${userHome}/.ivy2/cache"
```

If you do this, be aware that you may run into problems running Grails 2 and earlier versions of G
side-by-side. These problems can be avoided by excluding "xml-apis" and "commons-digester" from
inherited global dependencies in Grails 1.3 and earlier projects.

URL Domain Properties

If your domain model has any properties of type java.net.URL, they may cease to work once you upg
to Grails 2. It seems that the default mapping of URL to database column has changed with the new versic
Hibernate. This is a tricky problem to solve, but in the long run it's best if you migrate your URL properti
strings. One technique is to use the database migration plugin to add a new text column and then execute s
code in BootStrap (using Grails 1.3.x or earlier) to fetch each row of the table as a domain insta
convert the URL properties to string URLs, and then write those values to the new column.

Updated Underlying APIs

Grails 2.0 contains updated dependencies including Servlet 3.0, Tomcat 7, Spring 3.1, Hibernate 3.6
Groovy 1.8. This means that certain plugins and applications that depend on earlier versions of these A
may no longer work. For example the Servlet 3.0 HttpServletRequest interface includes new meth
so if a plugin implements this interface for Servlet 2.5 but not for Servlet 3.0 then said plugin will break.
same can be said of any Spring interface.

Removal of release-plugin Command

The built in `release-plugin` command for releases plugins to the central Grails plugin repository been removed. The new [release](#) plugin should be used instead which provides an equivale `publish-plugin` command.

Removal of Deprecated Classes

The following deprecated classes have been removed: `grails.web.JsonBuilder` `grails.web.OpenRicoBuilder`

Upgrading from Grails 1.2.x

Plugin Repositories

As of Grails 1.3, Grails no longer natively supports resolving plugins against secured SVN repositories. plugin resolution mechanism in Grails 1.2 and below has been replaced by one built on [Ivy](#), the upsid which is that you can now resolve Grails plugins against Maven repositories as well as regular G repositories.

Ivy supports a much richer set of repository resolvers for resolving plugins, including support for Web HTTP, SSH and FTP. See the section on [resolvers](#) in the Ivy docs for all the available options and the sec 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](#) pr provides a set of resolvers for 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 resou 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 d change the above to:

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

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

Tag and Body return values

Tags no longer return `java.lang.String` instances but instead return a Grails `StreamCharBuf` instance. The `StreamCharBuffer` class implements all the same methods as `String` but doesn't ext `String`, so code like this will break:

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

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

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

New JSONBuilder

There is a new version of `JSONBuilder` which is semantically different from the one used in ea versions of Grails. However, if your application depends on the older semantics you can still use deprecated implementation by setting 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 tha invalid objects are persisted. If one of your constraints (such as a custom validator) executes a query then can cause an additional flush, resulting in a `StackOverflowError`. For 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 new Hibernate `session` (which is recommended in general as doing Hibernate work during flushing 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
have a library that was compiled with Groovy 1.5 you must recompile it against Groovy 1.6 or higher be
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
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`
`grails-app/conf/BuildConfig.groovy`:

- `grails.config.base.webXml`

- `grails.project.war.file` (renamed from `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
baseline (see above).

4) The use of jsessionid (now considered harmful) is disabled by default. If your application requ
jsessionid you can re-enable its usage by adding the following to `grails-app/conf/Config.groov`

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

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

Plugin Changes

As of version 1.1, Grails no longer stores plugins inside your `PROJECT_HOME/plugins` directory default. This may result in compilation errors in your application unless you either re-install all your plu 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 impor 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("_GrailsBootstrap")
```

2) Due to an upgrade of Gant all references to the variable `Ant` should be changed to `ant`.

3) The root directory of the project is no longer on the classpath, so loading a resource like this will no lo 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 b 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 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 a foreign key reference. You shouldn't need to change anything; however you should drop column on 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 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")
```

2.4 Creating an Application

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

```
grails [command name]
```

Run [create-app](#) to create an application:

```
grails create-app helloworld
```

This will create a new directory inside the current one that contains the project. Navigate to this director your console:

```
cd helloworld
```

2.5 A Hello World Example

To implement the typical "hello world!" example `cd` into the "helloworld" directory created in the prev
section and activate interactive mode:

```
$ cd helloworld
$ grails
```

Grails' interactive mode will be activated and you should see a prompt that looks like the following:

Now 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
grails-app/controllers directory called helloworld/HelloController.groovy.

> ⚠ If no package is specified with create-controller script, Grails automatically uses the
> application name as the package name. This default is configurable with the
> grails.project.groupId attribute in Config.groovy.

Controllers are capable of dealing with web requests and to fulfil the "hello world!" use case
implementation needs to look like the following:

```
package helloworld
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 U
`http://localhost:8080/helloworld`
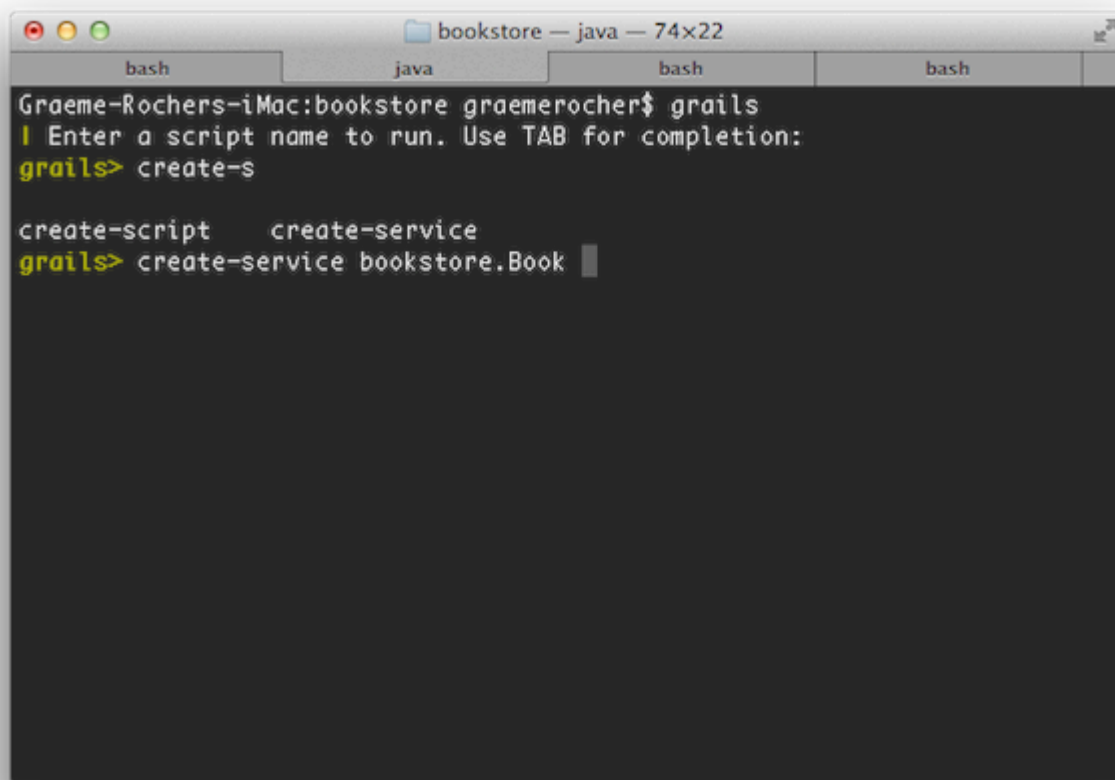
The result will look something like the following screenshot:
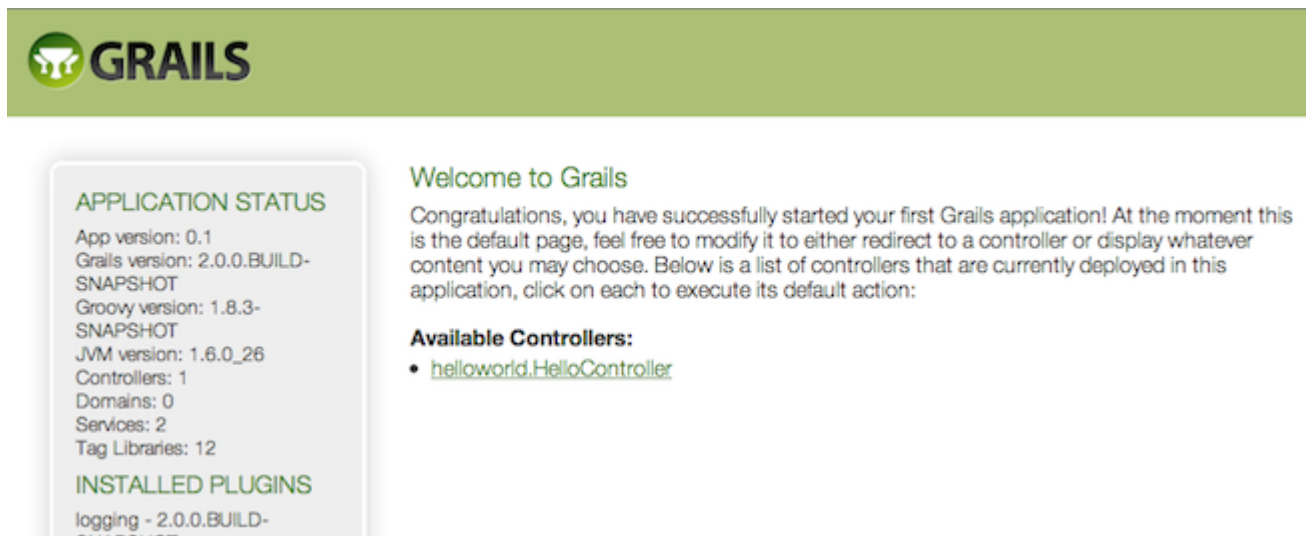
This is the Grails intro page which is rendered by the `web-app/index.gsp` file. You will note it h
detected the presence of your controller and clicking on the link to our controller we can see the text "H
World!" printed to the browser window.

2.6 Using Interactive Mode

Grails 2.0 features an interactive mode which makes command execution faster since the JVM doesn't hav
be restarted for each command. To use interactive mode simple type 'grails' from the root of any projects
use TAB completion to get a list of available commands. See the screenshot below for an example:

For more information on the capabilities of interactive mode refer to the section on <u>Interactive Mode</u> in user guide.

2.7 Getting Set Up in an IDE

IntelliJ IDEA

<u>IntelliJ IDEA</u> and the <u>JetGroovy</u> plugin offer good support for Groovy and Grails developers. Refer to section on <u>Groovy and Grails</u> support on the JetBrains website for a feature overview.

To integrate Grails with IntelliJ run the following command to generate appropriate project files:

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

Eclipse

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

NetBeans

NetBeans provides a Groovy/Grails plugin that automatically recognizes Grails projects and provides ability to run Grails applications in the IDE, code completion and integration with the Glassfish server. For overview of features see the [NetBeans Integration](#) guide on the Grails website which was written by NetBeans team.

TextMate

Since Grails' focus is on simplicity it is often possible to utilize more simple editors and [TextMate](#) on the has an excellent Groovy/Grails bundle available from the [Texmate bundles SVN](#).

To integrate Grails 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 follow command from the root of your project:

```
mate .
```

2.8 Convention over Configuration

Grails uses "convention over configuration" to configure itself. This typically means that the name location of files is used instead of explicit configuration, hence you need to familiarize yourself with 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](#).
    - `utils` - Grails specific utilities.
    - `views` - [Groovy Server Pages](#) - The V in MVC.
  - `scripts` - [Gant scripts](#).
  - `src` - Supporting sources
    - `groovy` - Other Groovy sources
    - `java` - Other Java sources
  - `test` - [Unit and integration tests](#).

2.9 Running an Application

Grails applications can be run with the built in Tomcat server using the [run-app](#) command which will lo
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
```

Note that it is better to start up the application in interactive mode since a container restart is much quicker

```
$ grails
grails> run-app
| Server running. Browse to http://localhost:8080/helloworld
| Application loaded in interactive mode. Type 'exit' to shutdown.
| Downloading: plugins-list.xml
grails> exit
| Stopping Grails server
grails> run-app
| Server running. Browse to http://localhost:8080/helloworld
| Application loaded in interactive mode. Type 'exit' to shutdown.
| Downloading: plugins-list.xml
```

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

2.10 Testing an Application

The `create-*` commands in Grails automatically create unit or integration tests for you within
`test/unit` or `test/integration` directory. It is of course up to you to populate these tests with v
test logic, information on which can be found in the section on [Testing](#).

To execute tests you run the [test-app](#) command as follows:

```
grails test-app
```

2.11 Deploying an Application

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

```
grails war
```

This will produce a WAR file under the `target` directory which can then be deployed as per your contai
instructions.

Unlike most scripts which default to the `development` environment unless overridden, the `war` comm
runs in the `production` environment by default. You can override this like any script by specifying
environment name, for example:

```
grails dev war
```

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

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

```
-server -Xmx512M -XX:MaxPermSize=256m
```

2.12 Supported Java EE Containers

Grails runs on any container that supports Servlet 2.5 and above and is known to work on the follov
specific container products:

- Tomcat 7

- Tomcat 6

- SpringSource tc Server

- Eclipse Virgo

- GlassFish 3

- GlassFish 2

- Resin 4

- Resin 3

- JBoss 6

- JBoss 5

- Jetty 7

- Jetty 6

- IBM Websphere 7.0

- IBM Websphere 6.1

- Oracle Weblogic 10.3

- Oracle Weblogic 10

- Oracle Weblogic 9

Some containers have bugs however, which in most cases can be worked around. A list of known deployı issues can be found on the Grails wiki.

2.13 Generating an Application

To get started quickly with Grails it is often useful to use a feature called Scaffolding to generate the skel of an application. To do this use one of the generate-* commands such as generate-all, which generate a controller (and its unit test) and the associated views:

```
grails generate-all Book
```

2.14 Creating Artefacts

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

> ⚠ These are just 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:

45

```
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.

> ⚠ To decrease the amount of time it takes to run Grails scripts, use the [interactive](#) mode.

3 Configuration

It may seem odd that in a framework that embraces "convention-over-configuration" that we tackle this t
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 an application without doing any configura
whatsoever. Grails ships with an embedded servlet container and in-memory H2 database, so there isn't ev
database to set up.

However, typically you should configure a more robust database at some point and that is described in
following section.

3.1 Basic Configuration

For general configuration Grails provides a file called `grails-app/conf/Config.groovy`. This
uses Groovy's [ConfigSlurper](#) which is very similar to Java properties files except it is pure Groovy hence
can reuse 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 from
[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
configuration object:

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

> ⊖ ConfigurationHolder and ApplicationHolder are deprecated and will be removed in a
> future version of Grails, so it is highly preferable to access the `GrailsApplication`
> and config from the `grailsApplication` variable.

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 should be merged with main configuration

- `grails.enable.native2ascii` - Set this to false if you do not require native2ascii conversic Grails i18n properties files

- `grails.views.default.codec` - Sets the default encoding regime for GSPs - can be on '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 typ [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 se name e.g. grails.serverURL="http://my.yourportal.com". See [createLink](#).

War generation

- `grails.project.war.file` - Sets the name and location of the WAR file generated by the command

- `grails.war.dependencies` - A closure containing Ant builder syntax or a list of JAR filenal Lets you customise what libaries are included in the WAR file.

- `grails.war.copyToWebApp` - A closure containing Ant builder syntax that is legal inside an copy, for example "fileset()". Lets you control what gets included in the WAR file from the "web-a directory.

- `grails.war.resources` - A closure containing Ant builder syntax. Allows the application to any other other work before building the final WAR file

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

3.1.2 Logging

The Basics

Grails uses its common configuration mechanism to provide the settings for the underlying [Log4j](#) log sys so all you have to do is add a `log4j` setting to the file `grails-app/conf/Config.groovy`.

So what does this `log4j` setting look like? Here's a basic example:

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

warn   'org.apache.catalina'
}
```

This says that for loggers whose name starts with 'org.codehaus.groovy.grails.web.servlet' 'org.codehaus.groovy.grails.web.pages', only messages logged at 'error' level and above will be sho Loggers with names starting with 'org.apache.catalina' logger only show messages at the 'warn' level above. What does that mean? First of all, you have to understand how levels work.

Logging levels

The are several standard logging levels, which are listed here in order of descending priority:

1. off

2. fatal

3. error

4. warn

5. info

6. debug

7. trace

8. all

When you log a message, you implicitly give that message a level. For example, the meth log.error(msg) will log a message at the 'error' level. Likewise, log.debug(msg) will log 'debug'. Each of the above levels apart from 'off' and 'all' have a corresponding log method of the same nam

The logging system uses that *message* level combined with the configuration for the logger (see next sect to determine whether the message gets written out. For example, if you have an 'org.example.domain' lo configured like so:

```
warn 'org.example.domain'
```

then messages with a level of 'warn', 'error', or 'fatal' will be written out. Messages at other levels wil ignored.

Before we go on to loggers, a quick note about those 'off' and 'all' levels. These are special in that they only be used in the configuration; you can't log messages at these levels. So if you configure a logger w level of 'off', then no messages will be written out. A level of 'all' means that you will see all messa Simple.

Loggers

Loggers are fundamental to the logging system, but they are a source of some confusion. For a start, wha they? Are they shared? How do you configure them?

A logger is the object you log messages to, so in the call log.debug(msg), log is a logger instance type Log). These loggers are cached and uniquely identified by name, so if two separate classes use log with the same name, those loggers are actually the same instance.

There are two main ways to get hold of a logger:

1. use the `log` instance injected into artifacts such as domain classes, controllers and services;

2. use the Commons Logging API directly.

If you use the dynamic `log` property, then the name of the logger is 'grails.app.<type>.<className>', wl type is the type of the artifact, for example 'controller' or 'service, and `className` is the fully qual name of the artifact. For example, if you have this service:

```
package org.example

class MyService {
    …
}
```

then the name of the logger will be 'grails.app.service.org.example.MyService'.

For other classes, the typical approach is to store a logger based on the class name in a constant static field

```
package org.other

import org.apache.commons.logging.LogFactory

class MyClass {
    private static final log = LogFactory.getLog(this)
    …
}
```

This will create a logger with the name 'org.other.MyClass' - note the lack of a 'grails.app.' prefix since class isn't an artifact. You can also pass a name to the `getLog()` method, such as "myLogger", but th less common because the logging system treats names with dots ('.') in a special way.

Configuring loggers

You have already seen how to configure loggers in Grails:

```
log4j = {
    error   'org.codehaus.groovy.grails.web.servlet'
}
```

This example configures loggers with names starting with 'org.codehaus.groovy.grails.web.servlet' to ig any messages sent to them at a level of 'warn' or lower. But is there a logger with this name in the applicat No. So why have a configuration for it? Because the above rule applies to any logger whose name *begins* 'org.codehaus.groovy.grails.servlet.' as well. For example, the rule applies to both tl `org.codehaus.groovy.grails.web.servlet.GrailsDispatcherServlet` class and `org.codehaus.groovy.grails.web.servlet.mvc.GrailsWebRequest` one.

In other words, loggers are hierarchical. This makes configuring them by package much simpler than it w otherwise be.

The most common things that you will want to capture log output from are your controllers, services, other artifacts. Use the convention mentioned earlier to do that: *grails.app.<artifactType>.<className>* particular the class name must be fully qualifed, i.e. with the package if there is one:

```
log4j = {
    // Set level for all application artifacts
    info "grails.app"

// Set for a specific controller in the default package
    debug "grails.app.controllers.YourController"

// Set for a specific domain class
    debug "grails.app.domain.org.example.Book"

// Set for all taglibs
    info "grails.app.taglib"
}
```

The standard artifact names used in the logging configuration are:

- `conf` - For anything under `grails-app/conf` such as `BootStrap.groovy` (but excluding filt
- `filters` - For filters
- `taglib` - For tag libraries
- `services` - For service classes
- `controllers` - For controllers
- `domain` - For domain entities

Grails itself generates plenty of logging information and it can sometimes be helpful to see that. Here are s useful loggers from Grails internals that you can use, especially when tracking down problems with application:

- `org.codehaus.groovy.grails.commons` - Core artifact information such as class loading e
- `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
- `grails.spring` - See what Spring beans Grails and plugins are defining
- `org.springframework` - See what Spring is doing
- `org.hibernate` - See what Hibernate is doing

So far, we've only looked at explicit configuration of loggers. But what about all those loggers that *don't* l an explicit configuration? Are they simply ignored? The answer lies with the root logger.

The Root Logger

All logger objects inherit their configuration from the root logger, so if no explicit configuration is prov for a given logger, then any messages that go to that logger are subject to the rules defined for the root log In other words, the root logger provides the default configuration for the logging system.

Grails automatically configures the root logger to only handle messages at 'error' level and above, and al messages are directed to the console (stdout for those with a C background). You can customise this behav by specifying a 'root' section in your logging configuration like so:

```
log4j = {
    root {
        info()
    }
    …
}
```

The above example configures the root logger to log messages at 'info' level and above to the default con
appender. You can also configure the root logger to log to one or more named appenders (which we'll
more about shortly):

```
log4j = {
    appenders {
        file name:'file', file:'/var/logs/mylog.log'
    }
    root {
        debug 'stdout', 'file'
    }
}
```

In the above example, the root logger will log to two appenders - the default 'stdout' (console) appender a
custom 'file' appender.

For power users there is an alternative syntax for configuring the root logger: the ro
`org.apache.log4j.Logger` instance is passed as an argument to the log4j closure. This lets you v
with the logger directly:

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

For more information on what you can do with this `Logger` instance, refer to the Log4j API documentatic

Those are the basics of logging pretty well covered and they are sufficient if you're happy to only send
messages to the console. But what if you want to send them to a file? How do you make sure that mess
from a particular logger go to a file but not the console? These questions and more will be answered a
look into appenders.

Appenders

Loggers are a useful mechanism for filtering messages, but they don't physically write the messages anywh
That's the job of the appender, of which there are various types. For example, there is the default one
writes messages to the console, another that writes them to a file, and several others. You can even create
own appender implementations!

This diagram shows how they fit into the logging pipeline:
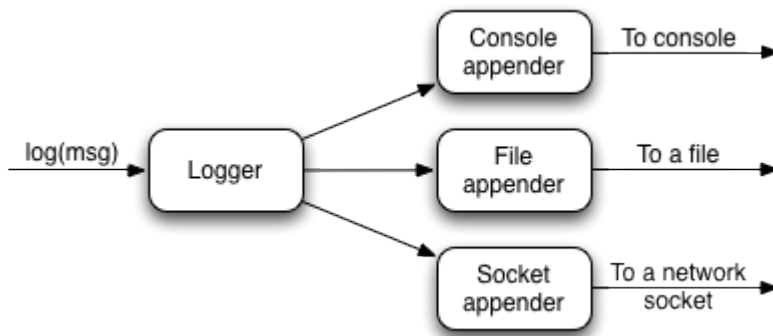
As you can see, a single logger may have several appenders attached to it. In a standard Grails configura
the console appender named 'stdout' is attached to all loggers through the default root logger configura
But that's the only one. Adding more appenders can be done within an 'appenders' block:

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

The following appenders are available by default:

Name	Class	Description
jdbc	JDBCAppender	Logs to a JDBC connection.
console	ConsoleAppender	Logs to the console.
file	FileAppender	Logs to a single file.
rollingFile	RollingFileAppender	Logs to rolling files, for example a new file each day.

Each named argument passed to an appender maps to a property of the underlying Appender implementa
So the previous example sets the `name`, `maxFileSize` and `file` properties of t
`RollingFileAppender` instance.

You can have as many appenders as you like - just make sure that they all have unique names. You can
have multiple instances of the same appender type, for example several file appenders that log to diffe
files.

If you prefer to create the appender programmatically or if you want to use an appender implementation t
not available in the above syntax, simply declare an `appender` entry with an instance of the appender
want:

```
import org.apache.log4j.*

log4j = {
    appenders {
        appender new RollingFileAppender(
                name: "myAppender",
                maxFileSize: 1024,
                file: "/tmp/logs/myApp.log")
    }
}
```

This approach can be used to configure JMSAppender, SocketAppender, SMTPAppender, and mo

Once you have declared your extra appenders, you can attach them to specific loggers by passing the nam
a key to one of the log level methods from the previous section:

```
error myAppender: "grails.app.controllers.BookController"
```

This will ensure that the 'grails.app.controller.BookController' logger sends log messages to 'myAppende
well as any appenders configured for the root logger. To add more than one appender to the logger, then
them to the same level declaration:

```
error myAppender:       "grails.app.controllers.BookController",
      myFileAppender:   ["grails.app.controllers.BookController",
                         "grails.app.services.BookService"],
      rollingFile:      "grails.app.controllers.BookController"
```

The above example also shows how you can configure more than one logger at a time for a given appen
myFileAppender) by using a list.

Be aware that you can only configure a single level for a logger, so if you tried this code:

```
error myAppender:       "grails.app.controllers.BookController"
debug myFileAppender:   "grails.app.controllers.BookController"
fatal rollingFile:      "grails.app.controllers.BookController"
```

you'd find that only 'fatal' level messages get logged for 'grails.app.controller.BookController'. That's bec
the last level declared for a given logger wins. What you probably want to do is limit what level of mess
an appender writes.

An appender that is attached to a logger configured with the 'all' level will generate a lot of log
information. That may be fine in a file, but it makes working at the console difficult. So we configure
console appender to only write out messages at 'info' level or above:

```
log4j = {
    appenders {
        console name: "stdout", threshold: org.apache.log4j.Level.INFO
    }
}
```

The key here is the `threshold` argument which determines the cut-off for log messages. This argume
available for all appenders, but do note that you currently have to specify a `Level` instance - a string su
"info" will not work.

Custom Layouts

By default the Log4j DSL assumes that you want to use a [PatternLayout](). However, there are other lay
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")
    }
}
```

Environment-specific configuration

Since the logging configuration is inside `Config.groovy`, you can put it inside an environment-spe
block. However, there is a problem with this approach: you have to provide the full logging configuration
time you define the `log4j` setting. In other words, you cannot selectively override parts of the configurati
it's all or nothing.

To get around this, the logging DSL provides its own environment blocks that you can put anywhere in
configuration:

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

    environments {
            production {
                rollingFile name: "myAppender", maxFileSize: 1024,
                            file: "/tmp/logs/myApp.log"
            }
        }
    }
    root {
        //…
    }

    // other shared config
    info "grails.app.controller"

    environments {
        production {
            // Override previous setting for 'grails.app.controller'
            error "grails.app.controllers"
        }
    }
}
```

The one place you can't put an environment block is *inside* the `root` definition, but you can put the r
definition inside an environment block.

Full stacktraces

When exceptions occur, there can be an awful lot of noise in the stacktrace from Java and Groovy inter
Grails filters these typically irrelevant details and restricts traces to non-core Grails/Groovy class packages

When this happens, the full trace is always logged to the `StackTrace` logger, which by default write
output to a file called `stacktrace.log`. As with other loggers though, you can change its behaviour in
configuration. For example if you prefer full stack traces to go to the console, add this entry:

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

This won't stop Grails from attempting to create the stacktrace.log file - it just redirects where stack traces
written to. An alternative approach is to change the location of the 'stacktrace' appender's file:

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

or, if you don't want to the 'stacktrace' appender at all, configure it as a 'null' appender:

```
log4j = {
    appenders {
        'null' name: "stacktrace"
    }
}
```

You can of course combine this with attaching the 'stdout' appender to the 'StackTrace' logger if you war
the output in the console.

Finally, you can completely disable stacktrace filtering by setting the grails.full.stacktrace
property to true:

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

Masking Request Parameters From Stacktrace Logs

When Grails logs a stacktrace, the log message may include the names and values of all of the req
parameters for the current request. To mask out the values of secure request parameters, specify the param
names in the grails.exceptionresolver.params.exclude config property:

```
grails.exceptionresolver.params.exclude = ['password', 'creditCard']
```

Request parameter logging may be turned off altogether by setting the
grails.exceptionresolver.logRequestParameters config property to false. The de
value is true when the application is running in DEVELOPMENT mode and false for all other modes.

```
grails.exceptionresolver.logRequestParameters=false
```

Logger inheritance

Earlier, we mentioned that all loggers inherit from the root logger and that loggers are hierarchical base
'.'-separated terms. What this means is that unless you override a parent setting, a logger retains the level
the appenders configured for that parent. So with this configuration:

```
log4j = {
    appenders {
        file name:'file', file:'/var/logs/mylog.log'
    }
    root {
        debug 'stdout', 'file'
    }
}
```

all loggers in the application will have a level of 'debug' and will log to both the 'stdout' and 'file' appenc
What if you only want to log to 'stdout' for a particular logger? Change the 'additivity' for a logger in that c

57

Additivity simply determines whether a logger inherits the configuration from its parent. If additivity is f then its not inherited. The default for all loggers is true, i.e. they inherit the configuration. So how do change this setting? Here's an example:

```
log4j = {
    appenders {
        …
    }
    root {
        …
    }
info additivity: false
        stdout: ["grails.app.controllers.BookController",
                 "grails.app.services.BookService"]
}
```

So when you specify a log level, add an 'additivity' named argument. Note that you when you specify additivity, you must configure the loggers for a named appender. The following syntax will *not* work:

```
info additivity: false, ["grails.app.controllers.BookController",
                         "grails.app.services.BookService"]
```

Customizing stack trace printing and filtering

Stacktraces in general and those generated when using Groovy in particular are quite verbose and con many stack frames that aren't interesting when diagnosing problems. So Grails uses a implementation of org.codehaus.groovy.grails.exceptions.StackTraceFilterer interface to filter irrelevant stack frames. To customize the approach used for filtering, implement that interface in a clas src/groovy or src/java and register it in Config.groovy:

```
grails.logging.stackTraceFiltererClass =
        'com.yourcompany.yourapp.MyStackTraceFilterer'
```

In addition, Grails customizes the display of the filtered stacktrace to make the information more readable customize this, implement the org.codehaus.groovy.grails.exceptions.StackTracePrinter interface in a clas src/groovy or src/java and register it in Config.groovy:

```
grails.logging.stackTracePrinterClass =
        'com.yourcompany.yourapp.MyStackTracePrinter'
```

Finally, to render error information in the error GSP, an HTML-generating printer implementation is nee
The default implementation is
`org.codehaus.groovy.grails.web.errors.ErrorsViewStackTracePrinter` and
registered as a Spring bean. To use your own implementation, either implement t
`org.codehaus.groovy.grails.exceptions.StackTraceFilterer` directly or subcla
`ErrorsViewStackTracePrinter` and register it in
`grails-app/conf/spring/resources.groovy` as:

```
import com.yourcompany.yourapp.MyErrorsViewStackTracePrinter

beans = {

errorsViewStackTracePrinter(MyErrorsViewStackTracePrinter,
                              ref('grailsResourceLocator'))
}
```

Alternative logging libraries

By default, Grails uses Log4J to do its logging. For most people this is absolutely fine, and many users of
even care what logging library is used. But if you're not one of those and want to use an alternative, suc
the [JDK logging package](#) or [logback](#), you can do so by simply excluding a couple of dependencies from
global set and adding your own:

```
grails.project.dependency.resolution = {
    inherits("global") {
        excludes "grails-plugin-logging", "log4j"
    }
    …
    dependencies {
        runtime "ch.qos.logback:logback-core:0.9.29"
        …
    }
    …
}
```

If you do this, you will get unfiltered, standard Java stacktraces in your log files and you won't be able to
the logging configuration DSL that's just been described. Instead, you will have to use the stan
configuration mechanism for the library you choose.

3.1.3 GORM

Grails provides the following GORM configuration options:

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

For example, to enable failOnError for all domain classes:

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

and to 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 session, replacing the need to explicitly flush using `save(flush: true)`.

3.2 Environments

Per Environment Configuration

Grails supports the concept of per environment configuration. The `Config.groovy`, `DataSource.groovy`, and `BootStrap.groovy` files in the `grails-app/conf` directory can per-environment configuration using the syntax provided by [ConfigSlurper]() As an example consider following default `DataSource` definition provided by Grails:

```
dataSource {
    pooled = false
    driverClassName = "org.h2.Driver"
    username = "sa"
    password = ""
}
environments {
    development {
        dataSource {
            dbCreate = "create-drop"
            url = "jdbc:h2:mem:devDb"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:prodDb"
        }
    }
}
```

Notice how the common configuration is provided at the top level and then an `environments` b specifies per environment settings for the `dbCreate` and `url` properties of the `DataSource`.

Packaging and Running for Different Environments

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

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

In addition, there are 3 preset environments known to Grails: `dev`, `prod`, and `test` for developme production and `test`. For example to create a WAR for the `test` environment you wound run:

```
grails test war
```

To target other environments 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 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 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 exec You can also use this class yourself to execute your own environment specific logic:

61

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

3.3 The DataSource

Since Grails is built on Java technology setting up a data source requires some knowledge of JDBC technology that doesn't stand for Java Database Connectivity).

If you use a database other than H2 you need a JDBC driver. For example for MySQL you would n Connector/J

Drivers typically come in the form of a JAR archive. It's best to use Ivy to resolve the jar if it's available Maven repository, for example you could add a dependency for the MySQL driver like this:

```
grails.project.dependency.resolution = {
    inherits("global")
    log "warn"
    repositories {
        grailsPlugins()
        grailsHome()
        grailsCentral()
        mavenCentral()
    }
    dependencies {
        runtime 'mysql:mysql-connector-java:5.1.16'
    }
}
```

Note that the built-in `mavenCentral()` repository is included here since that's a reliable location for library.

If you can't use Ivy then just put the JAR in your project's `lib` directory.

Once you have the JAR resolved you need to get familiar Grails' DataSource descriptor file located `grails-app/conf/DataSource.groovy`. This file contains the dataSource definition which incl 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 - one of 'create-d 'create', 'update' or 'validate'

- `pooled` - Whether to use a pool of connections (defaults to true)

- `logSql` - Enable SQL logging to stdout

- `formatSql` - Format logged SQL

- `dialect` - A String or Class that represents the Hibernate dialect used to communicate with database. See the [org.hibernate.dialect](#) package for available dialects.

- `readOnly` - If `true` makes the DataSource read-only, which results in the connection pool call `setReadOnly(true)` on each `Connection`

- `properties` - Extra properties to set on the DataSource bean. See the [Commons DI 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"
    dialect = org.hibernate.dialect.MySQL5InnoDBDialect
    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 ignored local variable
    …
}
```

Example of advanced configuration using extra properties:

```
dataSource {
    pooled = true
    dbCreate = "update"
    url = "jdbc:mysql://localhost/yourDB"
    driverClassName = "com.mysql.jdbc.Driver"
    dialect = org.hibernate.dialect.MySQL5InnoDBDialect
    username = "yourUser"
    password = "yourPassword"
    properties {
        maxActive = 50
        maxIdle = 25
        minIdle = 5
        initialSize = 5
        minEvictableIdleTimeMillis = 60000
        timeBetweenEvictionRunsMillis = 60000
        maxWait = 10000
        validationQuery = "/* ping */"
    }
}
```

More on dbCreate

Hibernate can automatically create the database tables required for your domain model. You have s
control over when and how it does this through the `dbCreate` property, which can take these values:

- **create** - Drops the existing schemaCreates the schema on startup, dropping existing tables, indexes, first.

- **create-drop** - Same as **create**, but also drops the tables when the application shuts down cleanly.

- **update** - Creates missing tables and indexes, and updates the current schema without dropping any ta or data. Note that this can't properly handle many schema changes like column renames (you're left the old column containing the existing data).

- **validate** - Makes no changes to your database. Compares the configuration with the existing data schema and reports warnings.

- any other value - does nothing

You can also remove the `dbCreate` setting completely, which is recommended once your schem
relatively stable and definitely when your application and database are deployed in production. Data
changes are then managed through proper migrations, either with SQL scripts or a migration tool l
[Liquibase](#) (the [Database Migration](#) plugin uses Liquibase and is tightly integrated with Grails and GORM)

3.3.1 DataSources and Environments

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

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

```
dataSource {
    pooled = true
    driverClassName = "com.mysql.jdbc.Driver"
    dialect = org.hibernate.dialect.MySQL5InnoDBDialect
    // other common settings here
}

environments {
    production {
        dataSource {
            url = "jdbc:mysql://liveip.com/liveDb"
            // other environment-specific settings here
        }
    }
}
```

3.3.2 JNDI DataSources

Referring to a JNDI DataSource

Most Java EE containers supply `DataSource` instances via [Java Naming and Directory Interface](#) (JN
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
`DataSource` in Grails 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
[Tomcat](#) plugin you can define JNDI resources using the `grails.naming.entries` setting
`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.h2.Driver",
          url: "jdbc:h2:mem: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
runtime with regards to automatically generating the database tables from [GORM](#) classes. The options
described in the [DataSource](#) section:

- create

- create-drop

- update

- validate

- no value

In [development](#) mode dbCreate is by default set to "create-drop", but at some point in development
certainly once you go to production) you'll need to stop dropping and re-creating the database every time
start up your server.

It's tempting to switch to update so you retain existing data and only update the schema when your
changes, but Hibernate's update support is very conservative. It won't make any changes that could resu
data loss, and doesn't detect renamed columns or tables, so you'll be left with the old one and will also l
the new one.

Grails supports Rails-style migrations via the [Database Migration](#) plugin which can be installed by running

```
grails install-plugin database-migration
```

The plugin uses [Liquibase](#) and and provides access to all of its functionality, and also has support for GC
(for example generating a change set by comparing your domain classes to a database).

3.3.4 Transaction-aware DataSource Proxy

The actual `dataSource` bean is wrapped in a transaction-aware proxy so you will be given the connec
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,
you wouldn't be able to see changes that haven't been committed yet (assuming you have a sens
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
`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
```

3.3.5 Database Console

The [H2 database console](#) is a convenient feature of H2 that provides a web-based interface to any data
that you have a JDBC driver for, and it's very useful to view the database you're developing against.
especially useful when running against an in-memory database.

You can access the console by navigating to **http://localhost:8080/appname/dbconsole** in a browser.
URI can be configured using the `grails.dbconsole.urlRoot` attribute in Config.groovy and defa
to `'/dbconsole'`.

The console is enabled by default in development mode and can be disabled or enabled in other environm
by using the `grails.dbconsole.enabled` attribute in Config.groovy. For example you could enable
console in production using

```
environments {
    production {
        grails.serverURL = "http://www.changeme.com"
        grails.dbconsole.enabled = true
        grails.dbconsole.urlRoot = '/admin/dbconsole'
    }
    development {
        grails.serverURL = "http://localhost:8080/${appName}"
    }
    test {
        grails.serverURL = "http://localhost:8080/${appName}"
    }
}
```

Configuration

By default the console is configured for an H2 database which will work with the default settings if haven't configured an external database - you just need to change the JDBC URL to `jdbc:h2:mem:dev`. If you've configured an external database (e.g. MySQL, Oracle, etc.) then you can use the Saved Sett dropdown to choose a settings template and fill in the url and username/password information from DataSource.groovy.

3.3.6 Multiple Datasources

By default all domain classes share a single `DataSource` and a single database, but you have the optic partition your domain classes into two or more `DataSources`.

Configuring Additional DataSources

The default `DataSource` configuration in `grails-app/conf/DataSource.groovy` looks somet like this:

```
dataSource {
    pooled = true
    driverClassName = "org.h2.Driver"
    username = "sa"
    password = ""
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = true
    cache.provider_class = 'net.sf.ehcache.hibernate.EhCacheProvider'
}

environments {
    development {
        dataSource {
            dbCreate = "create-drop"
            url = "jdbc:h2:mem:devDb"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:prodDb"
        }
    }
}
```

This configures a single `DataSource` with the Spring bean named `dataSource`. To configure ex
`DataSources`, add another `dataSource` block (at the top level, in an environment block, or both, just
the standard `DataSource` definition) with a custom name, separated by an underscore. For example,
configuration adds a second `DataSource`, using MySQL in the development environment and Oracl
production:

```
environments {
    development {
        dataSource {
            dbCreate = "create-drop"
            url = "jdbc:h2:mem:devDb"
        }
        dataSource_lookup {
            dialect = org.hibernate.dialect.MySQLInnoDBDialect
            driverClassName = 'com.mysql.jdbc.Driver'
            username = 'lookup'
            password = 'secret'
            url = 'jdbc:mysql://localhost/lookup'
            dbCreate = 'update'
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:mem:testDb"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:prodDb"
        }
        dataSource_lookup {
            dialect = org.hibernate.dialect.Oracle10gDialect
            driverClassName = 'oracle.jdbc.driver.OracleDriver'
            username = 'lookup'
            password = 'secret'
            url = 'jdbc:oracle:thin:@localhost:1521:lookup'
            dbCreate = 'update'
        }
    }
}
```

You can use the same or different databases as long as they're supported by Hibernate.

Configuring Domain Classes

If a domain class has no `DataSource` configuration, it defaults to the standard `'dataSource'`. Set
`datasource` property in the `mapping` block to configure a non-default `DataSource`. For example, if
want to use the `ZipCode` domain to use the `'lookup'` `DataSource`, configure it like this;

```
class ZipCode {

String code

static mapping = {
    datasource 'lookup'
  }
}
```

A domain class can also use two or more `DataSources`. Use the `datasources` property with a li:
names to configure more than one, for example:

```
class ZipCode {

String code

static mapping = {
    datasources(['lookup', 'auditing'])
  }
}
```

If a domain class uses the default `DataSource` and one or more others, use the special name `'DEFAU`
to indicate the default `DataSource`:

```
class ZipCode {

String code

static mapping = {
    datasources(['lookup', 'DEFAULT'])
  }
}
```

If a domain class uses all configured `DataSources` use the special value `'ALL'`:

```
class ZipCode {

String code

static mapping = {
    datasource 'ALL'
  }
}
```

Namespaces and GORM Methods

If a domain class uses more than one `DataSource` then you can use the namespace implied by e
`DataSource` name to make GORM calls for a particular `DataSource`. For example, consider this
which uses two `DataSources`:

```
class ZipCode {

String code

static mapping = {
    datasources(['lookup', 'auditing'])
  }
}
```

The first `DataSource` specified is the default when not using an explicit namespace, so in this case
default to 'lookup'. But you can call GORM methods on the 'auditing' `DataSource` with the `DataSou`
name, for example:

```
    def zipCode = ZipCode.auditing.get(42)
    …
    zipCode.auditing.save()
```

As you can see, you add the `DataSource` to the method call in both the static case and the instance case.

Services

Like Domain classes, by default Services use the default `DataSource` an
`PlatformTransactionManager`. To configure a Service to use a different `DataSource`, use the s
datasource property, for example:

```
class DataService {
    static datasource = 'lookup'
    void someMethod(...) {
        …
    }
}
```

A transactional service can only use a single `DataSource`, so be sure to only make changes for dor
classes whose `DataSource` is the same as the Service.

Note that the datasource specified in a service has no bearing on which datasources are used for dor
classes; that's determined by their declared datasources in the domain classes themselves. It's used to dec
which transaction manager to use.

What you'll see is that if you have a Foo domain class in dataSource1 and a Bar domain class in dataSour
and WahooService uses dataSource1, a service method that saves a new Foo and a new Bar will only
transactional for Foo since they share the datasource. The transaction won't affect the Bar instance. If
want both to be transactional you'd need to use two services and XA datasources for two-phase commit,
with the Atomikos plugin.

XA and Two-phase Commit

Grails has no native support for [XA](#) `DataSources` or [two-phase commit](#), but the [Atomikos plugin](#) mak
easy. See the plugin documentation for the simple changes needed in your `DataSource` definition
reconfigure them as XA `DataSources`.

3.4 Externalized Configuration

Some deployments require that configuration be sourced from more than one place and be changeable wit
requiring a rebuild of the application. In order to support deployment scenarios such as these the configura
can be externalized. To do so, point Grails at the locations of the configuration files that should be use
adding a `grails.config.locations` setting in `Config.groovy`, for example:

```
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 ConfigSlu configurations) from different places on the classpath and files located in USER_HOME.

It is also possible to load config by specifying a class that is a config script.

```
grails.config.locations = [com.my.app.MyConfig]
```

This can be useful in situations where the config is either coming from a plugin or some other part of application. A typical use for this is re-using configuration provided by plugins across multiple application

Ultimately all configuration files get merged into the config property of the GrailsApplication object are hence obtainable from there.

Values that have the same name as previously defined values will overwrite the existing values, and pointed to configuration sources are loaded in the order in which they are defined.

Config Defaults

The configuration values contained in the locations described by the grails.config.locati property will **override** any values defined in your application Config.groovy file which may not be you want. You may want to have a set of *default* values be be loaded that can be overridden in either application's Config.groovy file or in a named config location. For this you can use grails.config.defaults.locations property.

This property supports the same values as the grails.config.locations property (i.e. paths to co scripts, property files or classes), but the config described by grails.config.defaults.locati will be loaded *before* all other values and can therefore be overridden. Some plugins use this mechanis supply one or more sets of default configuration that you can choose to include in your application config.

> ⚠ 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. The version of the application is set to 0.1 when first create an application with the create-app command. The version is stored in the application meta data application.properties in the root of the project.

To change the version of your application you can edit the file manually, or 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 applica
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 t
[GrailsApplication](#) class. For example within [controllers](#) there is an implicit [grailsApplication](#) variable that
be used:

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

You can retrieve the the version of Grails that is running with:

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

or the `GrailsUtil` class:

```groovy
import grails.util.GrailsUtil
…
def grailsVersion = GrailsUtil.grailsVersion
```

3.6 Project Documentation

Since Grails 1.2, the documentation engine that powers the creation of this documentation has been avail
for your own Grails projects.

The documentation engine uses a variation on the [Textile](#) syntax to automatically create pro
documentation with smart linking, formatting etc.

Creating project documentation

To use the engine you need to follow a few conventions. First, you need to create a `src/docs/gu`
directory where your documentation source files will go. Then, you need to create the source docs themsel
Each chapter should have its own gdoc file as should all numbered sub-sections. You will end up
something like:

```
+ src/docs/guide/introduction.gdoc
+ src/docs/guide/introduction/changes.gdoc
+ src/docs/guide/gettingStarted.gdoc
+ src/docs/guide/configuration.gdoc
+ src/docs/guide/configuration/build.gdoc
+ src/docs/guide/configuration/build/controllers.gdoc
```

Note that you can have all your gdoc files in the top-level directory if you want, but you can also sub-sections in sub-directories named after the parent section - as the above example shows.

Once you have your source files, you still need to tell the documentation engine what the structure of user guide is going to be. To do that, you add a `src/docs/guide/toc.yml` file that contains structure and titles for each section. This file is in [YAML](#) format and basically represents the structure of user guide in tree form. For example, the above files could be represented as:

```
introduction:
  title: Introduction
  changes: Change Log
gettingStarted: Getting Started
configuration:
  title: Configuration
  build:
    title: Build Config
    controllers: Specifying Controllers
```

The format is pretty straightforward. Any section that has sub-sections is represented with the correspon filename (minus the .gdoc extension) followed by a colon. The next line should contain `title:` plus the of the section as seen by the end user. Every sub-section then has its own line after the title. Leaf nodes those without any sub-sections, declare their title on the same line as the section name but after the colon.

That's it. You can easily add, remove, and move sections within the `toc.yml` to restructure the gener user guide. You should also make sure that all section names, i.e. the gdoc filenames, should be unique s they are used for creating internal links and for the HTML filenames. Don't worry though, the documenta engine will warn you of duplicate section names.

Creating reference items

Reference items appear in the Quick Reference section of the documentation. Each reference item belongs category and a category is a directory located in the `src/docs/ref` directory. For example, suppose have defined a new controller method called `renderPDF`. That belongs to the `Controllers` categor you would create a gdoc text file at the following location:

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

Configuring Output Properties

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

- **grails.doc.title** - The title of the documentation

- **grails.doc.subtitle** - The subtitle of the documentation

- **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 version are pulled from your project itself. If a title is not specified, application name is used.

Generating Documentation

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

```
grails doc
```

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

Documentation Syntax

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

Basic Formatting

Monospace: `monospace`

```
@monospace@
```

Italic: *italic*

```
_italic_
```

Bold: **bold**

```
*bold*
```

Image:

```
!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 eithe 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 with the name of the sec
you want to link to:

```
[Intro|guide:introduction]
```

The section name comes from the corresponding gdoc filename. The documentation engine will warn yo
any links to sections in your guide break.

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 refere
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. To
additional APIs to the engine you can configure them in `grails-app/conf/Config.groovy`.
example:

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

The above example configures classes within the `org.hibernate` package to link to the Hiber
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 X markup:

```xml
<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

Grails features a dependency resolution DSL that lets you control how plugins and JAR dependencies resolved.

You specify a `grails.project.dependency.resolution` property inside the `grails-app/conf/BuildConfig.groovy` file that configures how dependencies are resolved:

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

The default configuration looks like the following:

```
    grails.project.class.dir = "target/classes"
    grails.project.test.class.dir = "target/test-classes"
    grails.project.test.reports.dir = "target/test-reports"
    //grails.project.war.file = "target/${appName}-${appVersion}.war"

    grails.project.dependency.resolution = {
        // inherit Grails' default dependencies
        inherits("global") {
            // uncomment to disable ehcache
            // excludes 'ehcache'
        }
        log "warn"
        repositories {
            grailsPlugins()
            grailsHome()
            grailsCentral()

// uncomment these to enable remote dependency resolution
            // from public Maven repositories
            //mavenCentral()
            //mavenLocal()
            //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 eg.

// runtime 'mysql:mysql-connector-java:5.1.16'
        }
    plugins {
            compile ":hibernate:$grailsVersion"
            compile ":jquery:1.6.1.1"
            compile ":resources:1.0"

    build ":tomcat:$grailsVersion"
        }
    }
```

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

3.7.1 Configurations and Dependencies

Grails features five dependency resolution configurations (or 'scopes'):

- `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 configuration calling the equivalent method. For example if your application requires the MySQL driver to function runtime you can specify that like this:

```
runtime 'com.mysql:mysql-connector-java:5.1.16'
```

This uses the string syntax: `group:name:version`. You can also use a Map-based syntax:

```
runtime group: 'com.mysql',
        name: 'mysql-connector-java',
        version: '5.1.16'
```

In Maven terminology, `group` corresponds to an artifact's `groupId` and `name` corresponds to `artifactId`.

Multiple dependencies can be specified by passing multiple arguments:

```
runtime 'com.mysql:mysql-connector-java:5.1.16',
        'net.sf.ehcache:ehcache:1.6.1'

// Or

runtime(
    [group:'com.mysql', name:'mysql-connector-java', version:'5.1.16'],
    [group:'net.sf.ehcache', name:'ehcache', version:'1.6.1']
)
```

Disabling transitive dependency resolution

By default, Grails will not only get the JARs and plugins that you declare, but it will also get their trans dependencies. This is usually what you want, but there are occasions where you want a dependency wit all its baggage. In such cases, you can disable transitive dependency resolution on a case-by-case basis:

```
runtime('com.mysql:mysql-connector-java:5.1.16',
        'net.sf.ehcache:ehcache:1.6.1') {
    transitive = false
}
// Or
runtime group:'com.mysql',
        name:'mysql-connector-java',
        version:'5.1.16',
        transitive:false
```

Excluding specific transitive dependencies

A far more common scenario is where you want the transitive dependencies, but some of them cause is with your own dependencies or are unnecessary. For example, many Apache projects have 'commons-log as a transitive dependency, but it shouldn't be included in a Grails project (we use SLF4J). That's where `excludes` option comes in:

```
runtime('com.mysql:mysql-connector-java:5.1.16',
        'net.sf.ehcache:ehcache:1.6.1') {
    excludes "xml-apis", "commons-logging"
}

// Or
runtime(group:'com.mysql', name:'mysql-connector-java', version:'5.1.16') {
    excludes([ group: 'xml-apis', name: 'xml-apis'],
             [ group: 'org.apache.httpcomponents' ],
             [ name: 'commons-logging' ])
```

As you can see, you can either exclude dependencies by their artifact ID (also known as a module name
any combination of group and artifact IDs (if you use the Map notation). You may also come acr
exclude as well, but that can only accept a single string or Map:

```
runtime('com.mysql:mysql-connector-java:5.1.16',
        'net.sf.ehcache:ehcache:1.6.1') {
    exclude "xml-apis"
}
```

Using Ivy module configurations

If you use Ivy module configurations and wish to depend on a specific configuration of a module, you can
the dependencyConfiguration method to specify the configuration to use.

```
provided("my.org:web-service:1.0") {
    dependencyConfiguration "api"
}
```

If the dependency configuration is not explicitly set, the configuration named "default" will be
(which is also the correct value for dependencies coming from Maven style repositories).

Where are the JARs?

With all these declarative dependencies, you may wonder where all the JARs end up. They have to
somewhere after all. By default Grails puts them into a directory, called the dependency cache, that reside
your local file system at user.home/.grails/ivy-cache. You can change this either via t
settings.groovy file:

```
grails.dependency.cache.dir = "${userHome}/.my-dependency-cache"
```

or in the dependency DSL:

```
grails.project.dependency.resolution = {
    …
    cacheDir "target/ivy-cache"
    …
}
```

The `settings.groovy` option applies to all projects, so it's the preferred approach.

3.7.2 Dependency Repositories

Remote Repositories

Initially your BuildConfig.groovy does not use any remote public Maven repositories. There is a def
`grailsHome()` repository that will locate the JAR files Grails needs from your Grails installation. To u
public repository, specify it in the `repositories` block:

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

In this case the default public Maven repository is specified. To use the SpringSource Enterprise Bu
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"
}
```

and even give it a name:

```
repositories {
    mavenRepo name: "Codehaus", root: "http://repository.codehaus.org"
}
```

so that you can easily identify it in logs.

Controlling Repositories Inherited from Plugins

A plugin you have installed may define a reference to a remote repository just as an application can
default your application will inherit this repository definition when you install the plugin.

If you do not wish to inherit repository definitions from plugins then you can disable repository inheritance

```
repositories {
    inherit false
}
```

In this case your application will not inherit any repository definitions from plugins and it is down to yo
provide appropriate (possibly internal) repository definitions.

Offline Mode

There are times when it is not desirable to connect to any remote repositories (whilst working on the trai
example!). In this case you can use the `offline` flag to execute Grails commands and Grails will
connect to any remote repositories:

```
grails --offline run-app
```

> ⚠ Note that this command will fail if you do not have the necessary dependencies in your
> local Ivy cache

You can also globally configure offline mode by setting `grails.offline.mode` to `true`
`~/.grails/settings.groovy` or in your project's `BuildConfig.groovy` file:

```
grails.offline.mode=true
```

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'
}
```

To specify your local Maven cache (`~/.m2/repository`) as a repository:

```
repositories {
    mavenLocal()
}
```

Custom Resolvers

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

```
/*
 * Configure our resolver.
 */
def libResolver = new org.apache.ivy.plugins.resolver.URLResolver()
['libraries', 'builds'].each {

libResolver.addArtifactPattern(
            "http://my.repository/${it}/" +
            "[organisation]/[module]/[revision]/[type]s/[artifact].[ext]")

libResolver.addIvyPattern(
            "http://my.repository/${it}/" +
            "[organisation]/[module]/[revision]/[type]s/[artifact].[ext]")
}

libResolver.name = "my-repository"
libResolver.settings = ivySettings

resolver libResolver
```

It's also possible to pull dependencies from a repository using SSH. Ivy comes with a dedicated resolver
you can configure and include in your project like so:

```
import org.apache.ivy.plugins.resolver.SshResolver
…
repositories {
    ...

def sshResolver = new SshResolver(
            name: "myRepo",
            user: "username",
            host: "dev.x.com",
            keyFile: new File("/home/username/.ssh/id_rsa"),
            m2compatible: true)

sshResolver.addArtifactPattern(
            "/home/grails/repo/[organisation]/[artifact]/" +
            "[revision]/[artifact]-[revision].[ext]")

sshResolver.latestStrategy =
            new org.apache.ivy.plugins.latest.LatestTimeStrategy()

sshResolver.changingPattern = ".*SNAPSHOT"

sshResolver.setCheckmodified(true)

resolver sshResolver
}
```

Download the JSch JAR and add it to Grails' classpath to use the SSH resolver. You can do this by passing
path in the Grails command line:

```
grails -classpath /path/to/jsch compile|run-app|etc.
```

You can also add its path to the CLASSPATH environment variable but be aware this it affects many
applications. An alternative on Unix is to create an alias for grails -classpath ... so that you
have to type the extra arguments each time.

Authentication

If your repository requires authentication you can configure this using a credentials block:

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

This can be placed in your USER_HOME/.grails/settings.groovy file using t
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
underlying engine using the log method:

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

A common issue is that the checksums for a dependency don't match the associated JAR file, and so
rejects the dependency. This helps ensure that the dependencies are valid. But for a variety of reasons s
dependencies simply don't have valid checksums in the repositories, even if they are valid JARs. To get re
this, you can disable Ivy's dependency checks like so:

```
grails.project.dependency.resolution = {
    …
    log "warn"
    checksums false
    …
}
```

This is a global setting, so only use it if you have to.

3.7.4 Inherited Dependencies

By default every Grails application inherits several framework dependencies. This is done through the line

```
inherits "global"
```

85

Inside the `BuildConfig.groovy` file. To exclude specific inherited dependencies you use the `exclu`
method:

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

3.7.5 Providing Default Dependencies

Most Grails applications have runtime dependencies on several jar files that are provided by the G
framework. These include libraries like Spring, Sitemesh, Hibernate etc. When a war file is created, a
these dependencies will be included in it. But, an application may choose to exclude these jar files from
war. This is useful when the jar files will be provided by the container, as would normally be the ca
multiple Grails applications are deployed to the same container.

The dependency resolution DSL provides a mechanism to express that all of the default dependencies wi
provided by the container. This is done by invoking the `defaultDependenciesProvided` method
passing `true` as an argument:

```
grails.project.dependency.resolution = {

defaultDependenciesProvided true // all of the default dependencies will
                                 // be "provided" by the container

inherits "global" // inherit Grails' default dependencies

repositories {
        grailsHome()
        …
    }
    dependencies {
        …
    }
}
```

> ⚠ `defaultDependenciesProvided` must come before `inherits`, otherwise the
> Grails dependencies will be included in the war.

3.7.6 Dependency Reports

As mentioned in the previous section a Grails application consists of dependencies inherited from
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
```

By default this will generate reports in the `target/dependency-report` directory. You can spe
which configuration (scope) you want a report for by passing an argument containing the configuration na

```
grails dependency-report runtime
```

3.7.7 Plugin JAR Dependencies

Specifying Plugin JAR dependencies

The way in which you specify dependencies for a [plugin](#) is identical to how you specify dependencies i application. When a plugin is installed into an application the application automatically inherits dependencies of the plugin.

To define a dependency that is resolved for use with the plugin but not *exported* to the application then can set the `export` property of the dependency:

```
test('org.spockframework:spock-core:0.5-groovy-1.8') {
    export = false
}
```

In this case the Spock dependency will be available only to the plugin and not resolved as an applica dependency. Alternatively, if you're using the Map syntax:

```
test group: 'org.spockframework', name: 'spock-core',
    version: '0.5-groovy-1.8', export: false
```

> ⚠ You can use `exported = false` instead of `export = false`, but we recommend the latter because it's consistent with the Map argument.

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 override how a plugin resolves its dependencies inside an application using exclusions. For example:

```
plugins {
    compile(":hibernate:$grailsVersion") {
        excludes "javassist"
    }
}
dependencies {
    runtime "javassist:javassist:3.4.GA"
}
```

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

3.7.8 Maven Integration

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

However, if you would like to continue using Grails regular commands like [run-app](#), [test-app](#) and so on 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.

3.7.9 Deploying to a Maven Repository

If you use Maven to build your Grails project, you can use the standard Maven targets `mvn install` `mvn deploy`. If not, you can deploy a Grails project or plugin to a Maven repository using [maven-publisher](#) plugin.

The plugin provides the ability to publish Grails projects and plugins to local and remote Maven reposito 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 alre 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 Ma cache:

```
grails maven-install
```

In the case of plugins, the plugin zip file will be installed, whilst for application the application WAR file 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 with `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. Configure the details of the reposi
specified by this 'id' within your `grails-app/conf/BuildConfig.groovy` file or in yo
`$USER_HOME/.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
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" f
"http://.." etc.), however to 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
Grails project conventions or plugin descriptor.

89

Projects

For applications this plugin will use the Grails application name and version provided by Grails v 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 on Config.groovy:

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

Plugins

With a Grails plugin the `groupId` and `version` are taken from the following properties in *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 call FeedsGrailsPlugin the `artifactId` will be "feeds". If your plugin does not specify a `groupId` this defaults to "org.grails.plugins".

3.7.10 Plugin Dependencies

As of Grails 1.3 you can declaratively specify plugins as dependencies via the dependency DSL instea using the [install-plugin](#) command:

```
grails.project.dependency.resolution = {
    …
    repositories {
        …
    }
plugins {
        runtime ':hibernate:1.2.1'
    }
dependencies {
        …
    }
    …
}
```

If you don't specify a group id the default plugin group id of `org.grails.plugins` is used. You 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. To not include snap
versions then 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 use 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.
example:

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

Here we have defined a dependency on the "weceem" plugin which transitively depends on the "searcha
plugin. By using the `excludes` method you can tell Grails *not* to transitively install the searchable pl
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 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 further through the use of convention and the `grails` command. When you type

```
grails [command name]
```

Grails searches 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
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 def
Grails will quit with an error.

To get a list of all commands 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 the Quick Reference menu of the reference
> guide for more information about individual commands

It's often useful to provide custom arguments to the JVM when running Grails commands, in particular v
`run-app` where you may for example want to set a higher maximum heap size. The Grails command will
any JVM options provided in the general `JAVA_OPTS` environment variable, but you can also speci
Grails-specific environment variable too:

```
export GRAILS_OPTS="-Xmx1G -Xms256m -XX:MaxPermSize=256m"
grails run-app
```

non-interactive mode
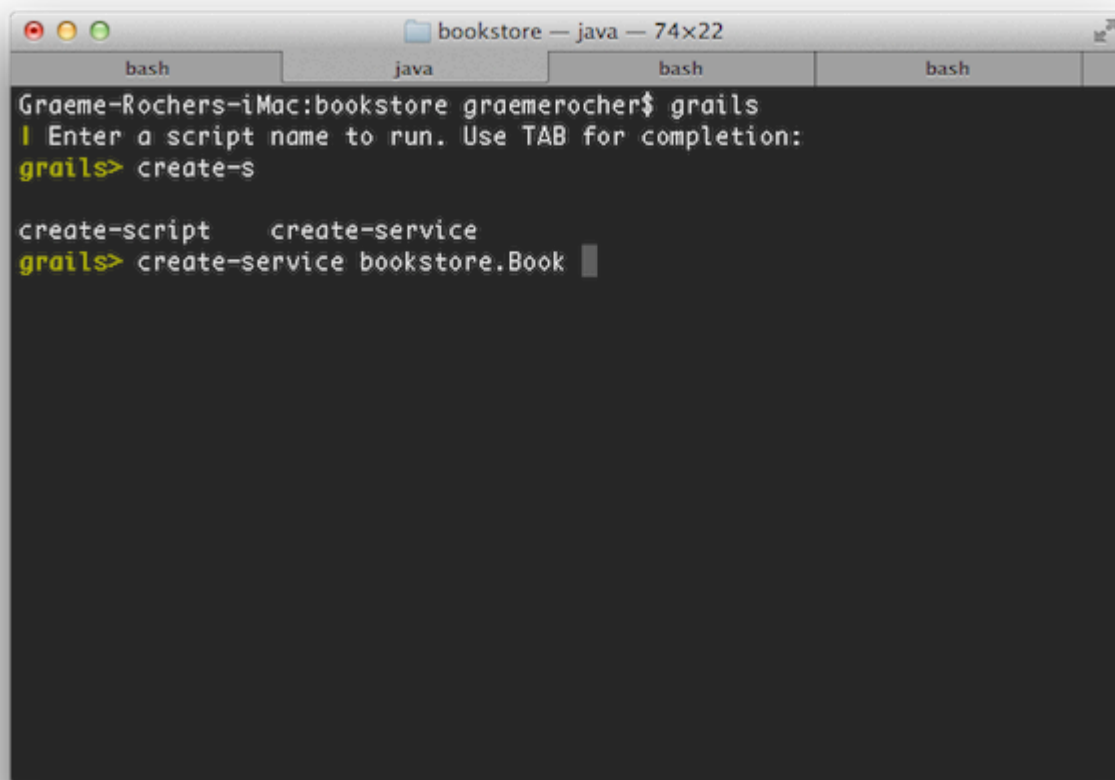
When you run a script manually and it prompts you for information, you can answer the questions
continue running the script. But when you run a script as part of an automated process, for examp
continuous integration build server, there's no way to "answer" the questions. So you can pass
`--non-interactive` switch to the script command to tell Grails to accept the default answer for
questions, for example whether to install a missing plugin.

For example:

```
grails war --non-interactive
```

4.1 Interactive Mode
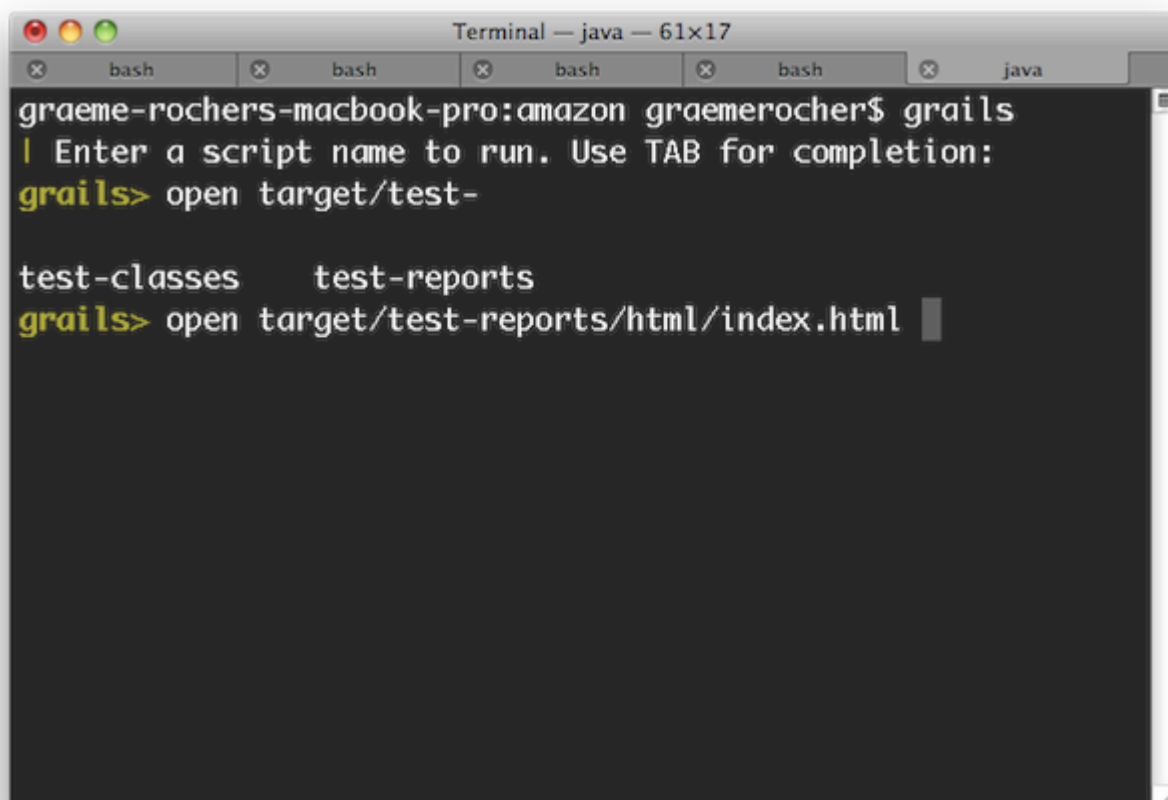
Interactive mode is the a feature of the Grails command line which keeps the JVM running and allows
quicker execution of commands. To activate interactive mode type 'grails' at the command line and then
TAB completion to get a list of commands:

If you need to open a file whilst within interactive mode you can use the `open` command which will
complete file paths:

Even better, the `open` command understands the logical aliases 'test-report' and 'dep-report', which will the most recent test and dependency reports respectively. In other words, to open the test report in a bro simply execute `open test-report`. You can even open multiple files at once: `open test-rep test/unit/MyTests.groovy` will open the HTML test report in your browser and t `MyTests.groovy` source file in your text editor.

TAB completion also works for class names after the `create-*` commands:

If you need to run an external process whilst interactive mode is running you can do so by starting command with a !:

Note that with ! (bang) commands, you get file path auto completion - ideal for external commands operate on the file system such as 'ls', 'cat', 'git', etc.

4.2 Creating Gant Scripts

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

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

Will create a script called `scripts/CompileSources.groovy`. A Gant script itself is similar 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 (an instance `groovy.util.AntBuilder`) 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` ta above.

The default target

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

```
target("clean-compile": "Performs a clean compilation on the app source") {
    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 lets you call the default target directly from other scripts if you wish. Also, although we have put the to `setDefaultTarget()` at the end of the script in this example, it can go anywhere as long as it co *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 n advantages in either case. One thing we would say is that if you want to allow other scripts to call y "default" target, you should move it into a shared script that doesn't have a default target at all. We'll talk s more about this in the next section.

4.3 Re-using Grails scripts

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

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

```
    includeTargets << grailsScript("_GrailsBootstrap")

    target ('default': "Database stuff") {
        depends(configureProxy, packageApp, classpath, loadApp, configureApp)

    Connection c
        try {
            c = appCtx.getBean('dataSource').getConnection()
            // do something with connection
        }
        finally {
            c?.close()
        }
    }
```

Pulling in targets from other scripts

Gant lets you pull in all targets (except "default") from another Gant script. You can then depend upo
invoke those targets as if they had been defined in the current script. The mechanism for doing this is
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
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 sy
for includeTargets when including core Grails targets. Instead, you should use the spec
grailsScript() method that is provided by the Grails command launcher (note that this is not avail
in normal Gant scripts, just Grails ones).

The syntax for the grailsScript() method is pretty straightforward: simply pass it the name of the G
script to include, without any path information. Here is a list of Grails scripts that you could reuse:

Script	Description
_GrailsSettings	You really should include this! Fortunately, it is included automatically by all ot Grails scripts except _GrailsProxy, so you usually don't have to include it explicitly.
_GrailsEvents	Include this to fire events. Adds an `event(String eventName, List arg` method. Again, included by almost all other Grails scripts.
_GrailsClasspath	Configures compilation, test, and runtime classpaths. If you want to use or play w them, include this script. Again, included by almost all other Grails scripts.
_GrailsProxy	If you don't have direct access to the internet and use a proxy, include this scrip configure access through your proxy.
_GrailsArgParsing	Provides a `parseArguments` target that does what it says on the tin: parses arguments provided by the user when they run your script. Adds them to the `argsM` 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, eit 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 what kind of targets are available. Anything that starts with an "_" is designed for reuse.

Script architecture

You maybe wondering what those underscores are doing in the names of the Grails scripts. That is Grails' of determining that a script is *internal* , or in other words that it has not corresponding "command". So 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 reuse. In fact, we recommend you take a similar approac your own scripts: put all your targets into an internal script that can be easily shared, and provide sir command scripts that parse any command line arguments and delegate to the targets in the internal script. example if you have a script that runs some functional tests, you can split it like this:

```groovy
./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 arguments direct to the closures.

4.4 Hooking into Events

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

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

Defining event handlers

Event handlers are defined in scripts called _Events.groovy. Grails searches for these scripts in 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 registratio handlers is performed automatically by Grails, so you just need to declare them in the relev _Events.groovy file.

Event handlers are blocks defined in _Events.groovy, with a name beginning with "event". The follov 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, eventStatusUpdat eventStatusFinal. Grails provides some standard events, which are documented in the command 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, when completing a target, even if the target does not exit scripting environment
CreatedArtefact	artefactType,artefactName	Called when a create-xxxx script has completed and created 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 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 groovydoc
SetClasspath	rootLoader	Called during classpath initialization so plugins can augment the classpath with rootLoader.addURL(...). Note that augments the classpath **after** event scripts are loaded so cannot use this to load a class that your event script needs 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 Tomcat server being started and after web.xml is generated)

4.5 Customising the build

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

The defaults

The core of the Grails build configuration is the `grails.util.BuildSettings` class, which cont quite a bit of useful information. It controls where classes are compiled to, what dependencies the applica 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 d[...] instance of `BuildSettings` is available to your scripts as the `grailsSettings` script variable. You[...] also access it from your code by using the `grails.util.BuildSettingsHolder` class, but this [...] recommended.

Overriding the defaults

All of the properties in the first table can be overridden by a system property or a configuration opti[...] simply use the "config option" name. For example, to change the project working directory, you could e[...] 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 work 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 because it takes precedence over the `BuildConfig.groovy` file, which in turn takes precedence over default values.

The `BuildConfig.groovy` file is a sibling of `grails-app/conf/Config.groovy` - the for contains options that only affect the build, whereas the latter contains those that affect the applicatic runtime. It's not limited to the options in the first table either: you will find build configuration options d around the documentation, such as ones for specifying the port that the embedded servlet container runs c 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-wa Integer.
grails.server.port.https	Port to run the embedded servlet container on for HTTPS ("run-app --htt and "run-war --https"). Integer.
grails.config.base.webXml	Path to a custom web.xml file to use for the application (alternative to us the web.xml template).
grails.compiler.dependencies	Legacy approach to adding extra dependencies to the compiler classpath. S to a closure containing "fileset()" entries. These entries will be processed an `AntBuilder` so the syntax is the Groovy form of the corresponding X elements in an Ant build file, e.g. `fileset(dir: "$basedir/lik include: "**/*.class)`.
grails.testing.patterns	A list of Ant path patterns that let you control which files are included in tests. The patterns should not include the test case suffix, which is set by next property.
grails.testing.nameSuffix	By default, tests are assumed to have a suffix of "Tests". You can change i anything you like but setting this option. For example, another common su is "Test".
grails.project.war.file	A string containing the file path of the generated WAR file, along with its name (include extension). For example, "target/my-app.war".
grails.war.dependencies	A closure containing "fileset()" entries that allows you complete control c what goes in the WAR's "WEB-INF/lib" directory.
grails.war.copyToWebApp	A closure containing "fileset()" entries that allows you complete control c what goes in the root of the WAR. It overrides the default behaviour including everything under "web-app".
grails.war.resources	A closure that takes the location of the staging directory as its first argum You can use any Ant tasks to do anything you like. It is typically usec remove files from the staging directory before that directory is jar'd up in WAR.
grails.project.web.xml	The location to generate Grails' web.xml to

4.6 Ant and Maven

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

Ant Integration

When you create a Grails application with the create-app command, Grails doesn't automatically create an `build.xml` file but you can generate one with the integrate-with command:

```
grails integrate-with --ant
```

This creates a `build.xml` file 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 configured to use Apache Ivy for dependency management, which means that it automatically download all the requisite Grails JAR files and other dependencies on demand. You don't have to install Grails locally to use it! That makes it particularly useful for continuous integration syst such as CruiseControl or Jenkins.

It uses the Grails Ant task to hook into the existing Grails build system. The task lets you run any Grails s 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" artifact, so that needs to be on the classpath at least. You should also include the "groovy-all" JAR. Witl 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 specified.
classpathref	Classpath to load Grails from. Must include the "grails-bootstrap" artifact and should include "grails-scripts".	Yes, unless `home` is se you use a `classpa` 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 sc 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 incl everything in @compileClasspath.
- `testClasspath` - Classpath used to compile and run the tests. Typically includes everything `runtimeClasspath`.

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

Maven Integration

Grails provides integration with [Maven 2](#) with a Maven plugin. The current Maven plugin is based on supersedes the version created by [Octo](#), who did a great job with the original.

Preparation

In order to use the new plugin, all you need is Maven 2 installed and set up. This is because **you no lor 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.

> ⚠ The default mvn setup DOES NOT supply sufficient memory to run the Grails environment. We recommend that you add the following environment variable setting to prevent poor performance:
>
> ```
> export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=256"
> ```

Creating a Grails Maven Project

To create a Mavenized Grails project simply run the following command:

```
mvn archetype:generate -DarchetypeGroupId=org.grails \
    -DarchetypeArtifactId=grails-maven-archetype \
    -DarchetypeVersion=1.3.2 \
    -DgroupId=example -DartifactId=my-app
```

Choose whichever grails version, group ID and artifact ID you want for your application, but everything must be as written. This will create a new Maven project with a POM and a couple of other files. What won't see is anything that looks like a Grails application. So, the next step is to create the project structure you're used to. But first, to set target JDK to Java 6, do that now. Open my-app/pom.xml and change

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>
```

to

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
  </configuration>
</plugin>
```

Then you're ready to create the project structure:

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

⚠ if you see a message similar to this:

```
Resolving plugin JAR dependencies …
:: problems summary ::
:::: WARNINGS
        module not found: org.hibernate#hibernate-core;3.3.1.GA
```

you need to add the plugins manually to application.properties:

```
plugins.hibernate=2.0.0
plugins.tomcat=2.0.0
```

then run

```
mvn compile
```

and the hibernate and tomcat plugins will be installed.

Now you have a Grails application all ready to go. The plugin integrates into the standard build cycle, so can use the standard Maven phases to build and package your app: `mvn clean`, `mvn compile`, `test`, `mvn package`, `mvn install`.

You can also use 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

For a complete, up to date list, run `mvn grails:help`

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 project and then copy the contents of the old one over. The solution is to create a POM for the existing pro using this Maven command (substitute the version number with the grails version of your existing project)

```
mvn org.grails:grails-maven-plugin:1.3.2:create-pom -DgroupId=com.mycompany
```

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

You may also want to set target JDK to Java 6; see above.

Adding Grails commands to phases

The standard POM created for you by Grails already attaches the appropriate core Grails commands to 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 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 execution block:

```xml
<plugin>
    <groupId>org.grails</groupId>
    <artifactId>grails-maven-plugin</artifactId>
    <version>1.3.2</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 the name of the command as the `command` system property, and optionally specify the arguments with `args` property:

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

Debugging a Grails Maven Project

Maven can be launched in debug mode using the "mvnDebug" command. To launch your Grails application debug, simply run:

```
mvnDebug grails:run-app
```

The process will be suspended on startup and listening for a debugger on port 8000.

If you need more control of the debugger, this can be specified using the MAVEN_OPTS environment variable, and launch Maven with the default "mvn" command:

```
MAVEN_OPTS="-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005
mvn grails:run-app
```

Raising issues

If you come across any problems with the Maven integration, please raise a JIRA issue as a sub-task
GRAILS-3547.

5 Object Relational Mapping (GORM)

Domain classes are core to any business application. They hold state about business processes and hopef also implement behavior. They are linked together through relationships; one-to-one, one-to-many many-to-many.

GORM is Grails' object relational mapping (ORM) implementation. Under the hood it uses Hibernate very popular and flexible open source ORM solution) and thanks to the dynamic nature of Groovy wit static and dynamic typing, along with the convention of Grails, there is far less configuration involve creating Grails domain classes.

You can also write Grails domain classes in Java. See the section on Hibernate Integration for how to v 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 helloworld.Person
```

> ⚠ If no package is specified with the create-domain-class script, Grails automatically uses the application name as the package name.

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

```
package helloworld
class Person {
}
```

> ⚠ If you have the dbCreate property set to "update", "create" or "create-drop" on your [DataSource](#), Grails will automatically generate/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 with the shell or console by typing:

```
grails console
```

This loads an interactive GUI where you can run Groovy commands with access to the Sp
ApplicationContext, GORM, etc.

5.1.1 Basic CRUD

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

Create

To create a domain class use Map constructor to 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 datal
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 no
persisted. Note that if you explicitly call the save method then the object is placed back into a read-write st

In addition, you can also load a 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
proxied instance, or throws an exception if no record is found for the specified id.

Update

To update an instance, change some properties and then 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.
example if you were building an Amazon-style bookstore you would be thinking about books, auth
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 I$
number and so on. The next few sections show how to model the domain in GORM.

To create a domain class you run the create-domain-class command as follows:

```
grails create-domain-class org.bookstore.Book
```

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

```
package org.bookstore

class Book {
}
```

This class will map automatically to a table in the database called book (the same name as the class).
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:

```
package org.bookstore

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 l
case separated by underscores. For example `releaseDate` maps onto a column `release_date`. The
types are auto-detected from the Java types, but can be customized with [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 en
relationship exists only in the direction it is defined.

5.2.1.1 Many-to-one and one-to-one

A many-to-one relationship is the simplest kind, and is defined with a property of the type of another dor
class. Consider this example:

Example A

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

```
class Nose {
}
```

In this case we have a unidirectional many-to-one relationship from `Face` to `Nose`. To make this relation
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 tha[t] can create a `Face`, attach a `Nose` instance to it and when we save or delete the `Face` instance, GORM save or delete the `Nose`. In other words, saves and deletes will cascade from `Face` to the associated `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 e[rror] due to a transient `Face`:

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

Now if we delete the `Face` instance, the `Nose` will go too:

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

To make the relationship a true one-to-one, use the `hasOne` property on the owning side, e.g. `Face`:

Example C

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

```
class Nose {
    Face face
}
```

Note that using this property puts the foreign key on the inverse table to the previous example, so in this the foreign key column is stored in the `nose` table inside a column called `face_id`. Also, `hasOne` works with bidirectional relationships.

Finally, it's a good idea to add a unique constraint on one side of the one-to-one relationship:

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

117

```
class Nose {
    Face face
}
```

5.2.1.2 One-to-many

A one-to-many relationship is when one class, example `Author`, has many instances of a another c
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 w
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
`hasMany` setting. This can be used to iterate over the collection:

```
def a = Author.get(1)

for (book in a.books) {
    println book.title
}
```

> ⚠ The default fetch strategy used by Grails is "lazy", which means that the collection will
> be lazily initialized on first access. 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
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 mappedE
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: "destinationAirport"]
}
```

```
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
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 this case `Author`, takes responsibility for persisting the relationship and is the only side that can case 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 this 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 responsib 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 collec types. For example, the following class creates a `nicknames` association that is a `Set` of `String` instan

```
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 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 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

121

⚠ 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. 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 spe behaviour.

Considerations

At the database level Grails by default uses table-per-hierarchy mapping with a discriminator column ca `class` so the parent class (`Content`) and its subclasses (`BlogEntry`, `Book` etc.), share the **same** table

Table-per-hierarchy mapping has a down side in that you **cannot** have non-nullable properties inheritance mapping. An alternative is to use table-per-subclass which can be enabled with the [ORM DSL]

However, excessive use of inheritance and table-per-subclass can result in poor query performance due t use of outer join queries. In general our advice is if you're going to use inheritance, don't abuse it and c 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
method on the `Content` super class will return all subclasses of `Content`:

```
def content = Content.list() // list all blog entries, books and podcasts
content = Content.findAllByAuthor('Joe Bloggs') // find all by author

def podCasts = PodCast.list() // list only podcasts
```

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 unord
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`. Sets guarantee uniquenes but not order, w
may not be what you want. To have custom ordering you configure the Set as a `SortedSet`:

```
class Author {

SortedSet books

static hasMany = [books: Book]
}
```

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

```
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 collection of the Author class wil
ordered by their release date.

Lists of Objects

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

123

```
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 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 in of the elements in the collection to retain this order at the database level.

When using a `List`, elements must be added to the collection before being saved, otherwise Hibernate 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()
```

Bags of Objects

If ordering and uniqueness aren't a concern (or if you manage these explicitly) then you can use the Hiber Bag type to represent mapped collections.

The only change required for this is to define the collection type as a `Collection`:

```
class Author {

Collection books

static hasMany = [books: Book]
}
```

Since uniqueness and order aren't managed by Hibernate, adding to or removing from collections mapped Bag don't trigger a load of all existing instances from the database, so this approach will perform better require less memory than using a `Set` or a `List`.

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 **mu** strings.

A Note on Collection Types and Performance

The Java Set type doesn't allow duplicates. To ensure uniqueness when adding an entry to a Set associat Hibernate has to load the entire associations from the database. If you have a large numbers of entries in 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 to mai order. Therefore it is recommended that if you anticipate a large numbers of records in the association that make the association bidirectional so that the link can be created on the inverse side. For example consider 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 necessar manipulate the collection directly resulting in fewer queries and more efficient code. Given an Author w large number of associated Book instances if you were to write code like the following you would se impact on performance:

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

You could also model the collection as a Hibernate Bag as described above.

5.3 Persistence Basics

A key thing to remember about Grails is that under the surface Grails is using [Hibernate](#) for persistence. If are coming from a background of using [ActiveRecord](#) or [iBatis](#) Hibernate's "session" model may feel a strange.

Grails automatically binds a Hibernate session to the currently executing request. This lets you use the and [delete](#) methods as well as other GORM methods transparently.

Transactional Write-Behind

A useful feature of Hibernate over direct JDBC calls and even other frameworks is that when you call [sav](#) [delete](#) it does not necessarily perform any SQL operations **at that point**. Hibernate batches up SQL statem and executes them as late as possible, often at the end of the request when flushing and closing the ses This is typically done for you automatically by Grails, which manages your Hibernate session.

Hibernate caches database updates where possible, only actually pushing the changes when it knows tl flush is required, or when a flush is triggered programmatically. One common case where Hibernate will f cached updates is when performing queries since the cached information might be included in the q results. But as long as you're doing non-conflicting saves, updates, and deletes, they'll be batched unti session is flushed. This can be a significant performance boost for applications that do a lot of database wr

Note that flushing is not the same as committing a transaction. If your actions are performed in the conte a transaction, flushing will execute SQL updates but the database will save the changes in its transac queue and only finalize the updates when the transaction commits.

5.3.1 Saving and Updating

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

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

This save will be not be pushed to the database immediately - it will be pushed when the next flush oc But there are occasions when you want to control when those statements are executed or, in Hiber 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, deletes, etc. will be synchron with the database. This also lets you catch any exceptions, which is typically useful in highly concur scenarios involving [optimistic locking](#):

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

Another thing to bear in mind is that Grails validates a domain instance every time you save it. If validation fails the domain instance will *not* be persisted to the database. By default, save() will si: return null in this case, but if you would prefer it to throw an exception you can use the failOnEr argument:

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

You can even change the default behaviour with a setting in Config.groovy, as described in the sectio configuration. Just remember that when you are saving domain instances that have been bound with provided by the user, the likelihood of validation exceptions is quite high and you won't want those except propagating to the end user.

You can find out more about the subtleties of saving data in this article - a must read!

5.3.2 Deleting Objects

An example of the delete method can be seen below:

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

As with saves, Hibernate will use transactional write-behind to perform the delete; to perform the d in-place you can use the flush argument:

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

Using the flush argument lets you catch any errors that occur during a delete. A common error that occur is if you violate a database constraint, although this is normally down to a programming or sch error. The following example shows how to catch a DataIntegrityViolationException th 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 ofte
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 pa
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, defining belongsTo will result in upd
cascading from the owning class to its dependant (the other side of the relationship), and for many-/one-to
and one-to-many relationships deletes will also cascade.

If you *do not* define belongsTo then no cascades will happen and you will have to manually save
object (except in the case of the one-to-many, in which case saves will cascade automatically if a new insta
is in a hasMany collection).

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 upd
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**
understand this better take a look at the summaries below that describe the default behaviour of GORM
regards to specific associations. Also read [part 2](#) of the GORM Gotchas series of articles to get a de
understanding of relationships and cascading.

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 cas
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 cas
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 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
    Location destination
    static belongsTo = [airport: Airport]
}
```

```
class Location {
    String city
    String country
}
```

Given the above domain classes and the following code:

```
def airport = Airport.findByName("Gatwick")
for (flight in airport.flights) {
    println flight.destination.city
}
```

GORM will execute a single SQL query to fetch the `Airport` instance, another to get its flights, and th
extra query for *each iteration* over the `flights` association to get the current flight's destination. In c
words you get N+1 queries (if you exclude the original one to get the airport).

Configuring Eager Fetching

An alternative approach that avoids the N+1 queries is to use eager fetching, which can be specifie
follows:

```
class Airport {
    String name
    static hasMany = [flights: Flight]
    static mapping = {
        flights lazy: false
    }
}
```

In this case the `flights` association will be loaded at the same time as its `Airport` instance, althou
second query will be executed to fetch the collection. You can also use `fetch: 'join'` instead of la
`false`, in which case GORM will only execute a single query to get the airports and their flights. This w
well for single-ended associations, but you need to be careful with one-to-manys. Queries will work as y
expect right up to the moment you add a limit to the number of results you want. At that point, you will li
end up with fewer results than you were expecting. The reason for this is quite technical but ultimately
problem arises from GORM using a left outer join.

So, the recommendation is currently to use `fetch: 'join'` for single-ended associations and la
`false` for one-to-manys.

Be careful how and where you use eager loading because you could load your entire database into men
with too many eager associations. You can find more information on the mapping options in the [section o
ORM DSL](#).

Using Batch Fetching

Although eager fetching is appropriate for some cases, it is not always desirable. If you made everything e
you could quite possibly load your entire database into memory resulting in performance and men
problems. An alternative to eager fetching is to use batch fetching. You can configure Hibernate to lazily f
results in "batches". For example:

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

In this case, due to the `batchSize` argument, when you iterate over the `flights` association, Hiber will fetch results in batches of 10. For example if you had an `Airport` that had 30 flights, if you d configure batch fetching you would get 1 query to fetch the `Airport` and then 30 queries to fetch flight. With batch fetching you get 1 query to fetch the `Airport` and 3 queries to fetch each `Fligh` batches of 10. In other words, batch fetching is an optimization of the lazy fetching strategy. Batch fetc can also be configured at the class level as follows:

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

Check out part 3 of the GORM Gotchas series for more in-depth coverage of this tricky topic.

5.3.5 Pessimistic and Optimistic Locking

Optimistic Locking

By default GORM classes are configured for optimistic locking. Optimistic locking is a feature of Hiber which involves storing a version value in a special `version` column in the database that is incremented each update.

The `version` column gets read into a `version` property that contains the current versioned stat 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 ver column in the database and if they differ will throw a StaleObjectException. This will roll back the transac if one is active.

This is useful as it allows a certain level of atomicity without resorting to pessimistic locking that ha inherit performance penalty. The downside is that you have to deal with this exception if you have hi 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 merg 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.

> ⚠ The `version` will only be updated after flushing the session.

Pessimistic Locking

Pessimistic locking is equivalent to doing a SQL "SELECT * FOR UPDATE" statement and locking a ro 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 with 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 commi However, in the above case what we are doing is "upgrading" from a regular SELECT to a SELECT..I UPDATE and another thread could still have updated the record in between the call to `get()` and the ca `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.

As well as the [lock](#) method you can also obtain a pessimistic locking using queries. For example usi 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 straightforwar retrieve the original values. If you try to reload the instance using [get](#) Hibernate will return the cu modified instance from its Session cache. Reloading using another query would trigger a flush which c cause problems if your data isn't ready to be flushed yet. So GORM provides some methods to retrieve 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
}
```

> ⚠ isDirty() does not currently check collection associations, but it does check all other persistent properties and associations.

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 er 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 ob
oriented query language HQL. Depending on the complexity of the query you have the following option
order of flexibility and power:

- Dynamic Finders

- Where Queries

- Criteria Queries

- Hibernate Query Language (HQL)

In addition, Groovy's ability to manipulate collections with [GPath](#) and methods like sort, findAll and s
combined with GORM results in a powerful combination.

However, let's start with the basics.

Listing instances

Use the [list](#) method to obtain all instances of a given class:

```
    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 `or`
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 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 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 [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 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` comparato equivalent to a SQL `like` expression.

The possible comparators include:

- `InList` - In the list of given values
- `LessThan` - less than a 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 take an argument)
- `IsNull` - Is a null value (doesn't take an argument)

Notice that the last three require different numbers of method arguments compared to the rest, as demonstr 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 or more criteria:

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

137

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

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

You can combine as many criteria as you like, but they must all be combined with `And` or all `Or`. If you want to combine `And` and `Or` or if the number of criteria creates a very long method name, just convert the query to a [Criteria](#) or [HQL](#) query.

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 and 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 Where Queries

The `where` method, introduced in Grails 2.0, builds on the support for [Detached Criteria](#) by providing enhanced, compile-time checked query DSL for common queries. The `where` method is more flexible dynamic finders, less verbose than criteria and provides a powerful mechanism to compose queries.

Basic Querying

The `where` method accepts a closure that looks very similar to Groovy's regular collection methods. closure should define the logical criteria in regular Groovy syntax, for example:

```
def query = Person.where {
    firstName == "Bart"
}
Person bart = query.find()
```

The returned object is a `DetachedCriteria` instance, which means it is not associated with any partic database connection or session. This means you can use the `where` method to define common queries a class level:

```groovy
class Person {
    static simpsons = where {
        lastName == "Simpson"
    }
    …
}
…
Person.simpsons.each {
    println it.firstname
}
```

Query execution is lazy and only happens upon usage of the [DetachedCriteria](#) instance. If you want to exe a where-style query immediately there are variations of the `findAll` and `find` methods to accomplish t

```groovy
def results = Person.findAll {
    lastName == "Simpson"
}
def results = Person.findAll(sort:"firstName") {
    lastName == "Simpson"
}
Person p = Person.find { firstName == "Bart" }
```

Each Groovy operator maps onto a regular criteria method. The following table provides a map of Gro operators to methods:

Operator	Criteria Method	Description
==	eq	Equal to
!=	ne	Not equal to
>	gt	Greater than
<	lt	Less than
>=	ge	Greater than or equal to
<=	le	Less than or equal to
in	inList	Contained within the given list
==~	like	Like a given string
=~	ilike	Case insensitive like

It is possible use regular Groovy comparison operators and logic to formulate complex queries:

```groovy
def query = Person.where {
    (lastName != "Simpson" && firstName != "Fred") || (firstName == "Bart" && age
    > 9)
}
def results = query.list(sort:"firstName")
```

139

The Groovy regex matching operators map onto like and ilike queries unless the expression on the right side is a Pattern object, in which case they map onto an rlike query:

```
def query = Person.where {
    firstName ==~ ~/B.+/
}
```

> ⚠ Note that rlike queries are only supported if the underlying database supports regular expressions

A between criteria query can be done by combining the in keyword with a range:

```
def query = Person.where {
    age in 18..65
}
```

Finally, you can do isNull and isNotNull style queries by using null with regular comparison operators:

```
def query = Person.where {
    middleName == null
}
```

Query Composition

Since the return value of the where method is a [DetachedCriteria](#) instance you can compose new queries from the original query:

```
def query = Person.where {
    lastName == "Simpson"
}
def bartQuery = query.where {
    firstName == "Bart"
}
Person p = bartQuery.find()
```

Note that you cannot pass a closure defined as a variable into the where method unless it has been explicitly cast to a DetachedCriteria instance. In other words the following will produce an error:

```
def callable = {
    lastName == "Simpson"
}
def query = Person.where(callable)
```

The above must be written as follows:

```
import grails.gorm.DetachedCriteria

def callable = {
    lastName == "Simpson"
} as DetachedCriteria<Person>
def query = Person.where(callable)
```

As you can see the closure definition is cast (using the Groovy `as` keyword) to a [DetachedCriteria](#) inst
targeted at the `Person` class.

Conjunction, Disjunction and Negation

As mentioned previously you can combine regular Groovy logical operators (`||` and `&&`) to form conjunct
and disjunctions:

```
def query = Person.where {
    (lastName != "Simpson" && firstName != "Fred") || (firstName == "Bart" && age
> 9)
}
```

You can also negate a logical comparison using `!`:

```
def query = Person.where {
    firstName == "Fred" && !(lastName == 'Simpson')
}
```

Property Comparison Queries

If you use a property name on both the left hand and right side of a comparison expression then
appropriate property comparison criteria is automatically used:

```
def query = Person.where {
    firstName == lastName
}
```

The following table described how each comparison operator maps onto each criteria property compar
method:

Operator	Criteria Method	Description
==	eqProperty	Equal to
!=	neProperty	Not equal to
>	gtProperty	Greater than
<	ltProperty	Less than
>=	geProperty	Greater than or equal to
<=	leProperty	Less than or equal to

Querying Associations

Associations can be queried by using the dot operator to specify the property name of the association t
queried:

```
def query = Pet.where {
    owner.firstName == "Joe" || owner.firstName == "Fred"
}
```

You can group multiple criterion inside a closure method call where the name of the method matches
association name:

```
def query = Person.where {
    pets { name == "Jack" || name == "Joe" }
}
```

This technique can be combined with other top-level criteria:

```
def query = Person.where {
    pets { name == "Jack" } || firstName == "Ed"
}
```

For collection associations it is possible to apply queries to the size of the collection:

```
def query = Person.where {
    pets.size() == 2
}
```

The following table shows which operator maps onto which criteria method for each size() comparison:

Operator	Criteria Method	Description
==	sizeEq	The collection size is equal to
!=	sizeNe	The collection size is not equal to
>	sizeGt	The collection size is greater than
<	sizeLt	The collection size is less than
>=	sizeGe	The collection size is greater than or equal to
<=	sizeLe	The collection size is less than or equal to

Subqueries

It is possible to execute subqueries within where queries. For example to find all the people older than average age the following query can be used:

```
final query = Person.where {
  age > avg(age)
}
```

The following table lists the possible subqueries:

Method	Description
avg	The average of all values
sum	The sum of all values
max	The maximum value
min	The minimum value
count	The count of all values
property	Retrieves a property of the resulting entities

You can apply additional criteria to any subquery by using the `of` method and passing in a closure containing the criteria:

```
def query = Person.where {
  age > avg(age).of { lastName == "Simpson" } && firstName == "Homer"
}
```

Since the `property` subquery returns multiple results, the criterion used compares all results. For example the following query will find all people younger than people with the surname "Simpson":

```
Person.where {
    age < property(age).of { lastName == "Simpson" }
}
```

Other Functions

There are several functions available to you within the context of a query. These are summarized in the table below:

Method	Description
second	The second of a date property
minute	The minute of a date property
hour	The hour of a date property
day	The day of the month of a date property
month	The month of a date property
year	The year of a date property
lower	Converts a string property to upper case
upper	Converts a string property to lower case
length	The length of a string property
trim	Trims a string property

> ⚠ Currently functions can only be applied to properties or associations of domain classes. You cannot, for example, use a function on a result of a subquery.

For example the following query can be used to find all pet's born in 2011:

```
def query = Pet.where {
    year(birthDate) == 2011
}
```

You can also apply functions to associations:

```
def query = Person.where {
    year(pets.birthDate) == 2009
}
```

Batch Updates and Deletes

Since each `where` method call returns a [DetachedCriteria](#) instance, you can use `where` queries to exe batch operations such as batch updates and deletes. For example, the following query will update all pe with the surname "Simpson" to have the surname "Bloggs":

```
def query = Person.where {
    lastName == 'Simpson'
}
int total = query.updateAll(lastName:"Bloggs")
```

> ⚠ Note that one limitation with regards to batch operations is that join queries (queries that query associations) are not allowed.

To batch delete records you can use the `deleteAll` method:

```
def query = Person.where {
    lastName == 'Simpson'
}
int total = query.deleteAll()
```

5.4.3 Criteria

Criteria is an advanced way to query that uses a Groovy builder to construct potentially complex queries. a much better approach than building up query strings using a `StringBuffer`.

Criteria can be used either with the createCriteria or withCriteria methods. The builder uses Hibern Criteria API. The nodes on this builder map the static methods found in the Restrictions class of the Hiber Criteria API. For example:

```
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 in a List 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 order by `holderLastName`.

If no records are found with the above criteria, an empty List is returned.

Conjunctions and Disjunctions

As demonstrated in the previous example you can group criteria in a logical OR using an `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 Acco 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 la 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 rece
created in the last 10 days.

Querying with Projections

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

```
def c = Account.createCriteria()

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

When multiple fields are specified in the projection, a List of values will be returned. A single value wi
returned otherwise.

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"
}
```

SQL Restrictions may be parameterized to deal with SQL injection vulnerabilities related to dyna
restrictions.

```
def c = Person.createCriteria()

def peopleWithShortFirstNames = c.list {
    sqlRestriction "char_length(first_name) < ? AND char_length(first_name) > ?",
[maxValue, minValue]
}
```

> ⚠ Note that the parameter there is SQL. The `first_name` attribute referenced in the example refers to the persistence model, not the object model like in HQL queries. The `Person` property named `firstName` is mapped to the `first_name` column in the database and you must refer to that in the `sqlRestriction` string.
>
> 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 Criteria object itself. This allows full access to all the properties in this class. This example ca setMaxResults and setFirstResult on the [Criteria](#) instance:

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

Querying with Eager Fetching

In the section on [Eager and Lazy Fetching](#) we discussed how to declaratively specify fetching to avoid 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: it tells the criteria API to use a `JOIN` to fetch the named associat
with the `Task` instances. It's probably best not to use this for one-to-many associations though, because
will most likely end up with duplicate results. Instead, use the 'select' fetch mode:

```
import org.hibernate.FetchMode as FM
…
def results = Airport.withCriteria {
    eq "region", "EMEA"
    fetchMode "flights", FM.SELECT
}
```

Although this approach triggers a second query to get the `flights` association, you will get reliable resu
even with the `maxResults` option.

> ⚠ `fetchMode` and `join` are general settings of the query and can only be specified at
> the top-level, i.e. you cannot use them inside projections or association constraints.

An important point to bear in mind is that if you include associations in the query constraints, th
associations will automatically be eagerly loaded. For example, in this query:

```
def results = Airport.withCriteria {
    eq "region", "EMEA"
    flights {
        like "number", "BA%"
    }
}
```

the `flights` collection would be loaded eagerly via a join even though the fetch mode has not I
explicitly set.

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 c 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 result set, this allows listing only distinct entities and is equivalent t `DISTINCT_ROOT_ENTITY` of the [CriteriaSpecification](#) class.
count	Returns the number of matching rows.

5.4.4 Detached Criteria

Detached Criteria are criteria queries that are not associated with any given database session/connec Supported since Grails 2.0, Detached Criteria queries have many uses including allowing you to cr common reusable criteria queries, execute subqueries and execute batch updates/deletes.

Building Detached Criteria Queries

The primary point of entry for using the Detached Criteria is the `grails.gorm.DetachedCrite` class which accepts a domain class as the only argument to its constructor:

```
import grails.gorm.*
…
def criteria = new DetachedCriteria(Person)
```

Once you have obtained a reference to a detached criteria instance you can execute [where](#) queries or cri queries to build up the appropriate query. To build a normal criteria query you can use the `build` method

```
def criteria = new DetachedCriteria(Person).build {
    eq 'lastName', 'Simpson'
}
```

Note that methods on the `DetachedCriteria` instance **do not** mutate the original object but instead re a new query. In other words, you have to use the return value of the `build` method to obtain the mut criteria object:

```
def criteria = new DetachedCriteria(Person).build {
    eq 'lastName', 'Simpson'
}
def bartQuery = criteria.build {
    eq 'firstName', 'Bart'
}
```

Executing Detached Criteria Queries

Unlike regular criteria, Detached Criteria are lazy, in that no query is executed at the point of definition. Once a Detached Criteria query has been constructed then there are a number of useful query methods which summarized in the table below:

Method	Description
list	List all matching entities
get	Return a single matching result
count	Count all matching records
exists	Return true if any matching records exist
deleteAll	Delete all matching records
updateAll(Map)	Update all matching records with the given properties

As an example the following code will list the first 4 matching records sorted by the `firstName` property

```
def criteria = new DetachedCriteria(Person).build {
    eq 'lastName', 'Simpson'
}
def results = criteria.list(max:4, sort:"firstName")
```

You can also supply additional criteria to the list method:

```
def results = criteria.list(max:4, sort:"firstName") {
    gt 'age', 30
}
```

To retrieve a single result you can use the `get` or `find` methods (which are synonyms):

```
Person p = criteria.find() // or criteria.get()
```

The `DetachedCriteria` class itself also implements the `Iterable` interface which means that it ca treated like a list:

```
def criteria = new DetachedCriteria(Person).build {
    eq 'lastName', 'Simpson'
}
criteria.each {
    println it.firstName
}
```

In this case the query is only executed when the `each` method is called. The same applies to all other Gro collection iteration methods.

You can also execute dynamic finders on `DetachedCriteria` just like on domain classes. For example

```
def criteria = new DetachedCriteria(Person).build {
    eq 'lastName', 'Simpson'
}
def bart = criteria.findByFirstName("Bart")
```

Using Detached Criteria for Subqueries

Within the context of a regular criteria query you can use DetachedCriteria to execute subquery.
example if you want to find all people who are older than the average age the following query will accomp
that:

```
def results = Person.withCriteria {
    gt "age", new DetachedCriteria(Person).build {
        projections {
            avg "age"
        }
    }
    order "firstName"
}
```

Notice that in this case the subquery class is the same as the original criteria query class (ie. Person)
hence the query can be shortened to:

```
def results = Person.withCriteria {
    gt "age", {
        projections {
            avg "age"
        }
    }
    order "firstName"
}
```

If the subquery class differs from the original criteria query then you will have to use the original syntax.

In the previous example the projection ensured that only a single result was returned (the average age). If
subquery returns multiple results then there are different criteria methods that need to be used to compare
result. For example to find all the people older than the ages 18 to 65 a gtAll query can be used:

```
def results = Person.withCriteria {
    gtAll "age", {
        projections {
            property "age"
        }
        between 'age', 18, 65
    }
order "firstName"
}
```

The following table summarizes criteria methods for operating on subqueries that return multiple results:

Method	Description
gtAll	greater than all subquery results
geAll	greater than or equal to all subquery results
ltAll	less than all subquery results
leAll	less than or equal to all subquery results
eqAll	equal to all subquery results
neAll	not equal to all subquery results

Batch Operations with Detached Criteria

The `DetachedCriteria` class can be used to execute batch operations such as batch updates and del
For example, the following query will update all people with the surname "Simpson" to have the surn
"Bloggs":

```
def criteria = new DetachedCriteria(Person).build {
    eq 'lastName', 'Simpson'
}
int total = criteria.updateAll(lastName:"Bloggs")
```

> ⚠️ Note that one limitation with regards to batch operations is that join queries (queries that
> query associations) are not allowed within the `DetachedCriteria` instance.

To batch delete records you can use the `deleteAll` method:

```
def criteria = new DetachedCriteria(Person).build {
    eq 'lastName', 'Simpson'
}
int total = criteria.deleteAll()
```

5.4.5 Hibernate Query Language (HQL)

GORM classes also support Hibernate's query language HQL, a very complete reference for which ca
found [in the Hibernate documentation](#) of the Hibernate documentation.

GORM provides a number of methods that work with HQL including [find](#), [findAll](#) and [executeQuery](#)
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

Use the line continuation character to separate the query across multiple lines:

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

> ⚠ Triple-quoted 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 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 events.

5.5.1 Events and Auto Timestamping

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

- `beforeInsert` - Executed before an object is initially persisted to the database

- `beforeUpdate` - Executed before an object is updated

- `beforeDelete` - Executed before an object is deleted

- `beforeValidate` - Executed before an object is validated

- `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 deleted

- `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 database

```
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 Hibernat flushing using persistence methods like save() and delete() won't result in objects being saved ur you run your operations with a new Session.

Fortunately the withNewSession method lets you share the same transactional JDBC connection though you're using a different underlying Session.

The beforeValidate event

Fired before an object is validated.

```
class Person {
    String name

    static constraints = {
        name size: 5..45
    }

    def beforeValidate() {
        name = name?.trim()
    }
}
```

The beforeValidate method is run before any validators are run.

GORM supports an overloaded version of `beforeValidate` which accepts a `List` parameter which include the names of the properties which are about to be validated. This version of `beforeValidate` be called when the `validate` method has been invoked and passed a `List` of property names a argument.

```
class Person {
    String name
    String town
    Integer age

static constraints = {
        name size: 5..45
        age range: 4..99
    }

def beforeValidate(List propertiesBeingValidated) {
        // do pre validation work based on propertiesBeingValidated
    }
}

def p = new Person(name: 'Jacob Brown', age: 10)
p.validate(['age', 'name'])
```

> ⚠️ Note that when `validate` is triggered indirectly because of a call to the `save` method that the `validate` method is being invoked with no arguments, not a `List` that includes all of the property names.

Either or both versions of `beforeValidate` may be defined in a domain class. GORM will prefer `List` version if a `List` is passed to `validate` but will fall back on the no-arg version if the `List` ver does not exist. Likewise, GORM will prefer the no-arg version if no arguments are passed to `validate` will fall back on the `List` version if the no-arg version does not exist. In that case, `null` is passed `beforeValidate`.

The onLoad/beforeLoad event

Fired immediately before an object is loaded from the database:

```
class Person {
    String name
    Date dateCreated
    Date lastUpdated

def onLoad() {
        log.debug "Loading ${id}"
    }
}
```

`beforeLoad()` is effectively a synonym for `onLoad()`, so only declare one or the other.

The afterLoad event

Fired immediately after an object is loaded from the database:

```
class Person {
    String name
    Date dateCreated
    Date lastUpdated

def afterLoad() {
        name = "I'm loaded"
    }
}
```

Custom Event Listeners

As of Grails 2.0 there is a new API for plugins and applications to register and listen for persistence ev
This API is not tied to Hibernate and also works for other persistence plugins such as the MongoDB plugi
GORM.

To use this API you need to subclass the
org.grails.datastore.mapping.engine.event.AbstractPersistenceEventListe
and implement a single method called onPersistenceEvent. The simplest possible implementation
be seen below:

```
@Override
protected void onPersistenceEvent(final AbstractPersistenceEvent event) {
    switch(event.eventType) {
        case PreInsert:
            println "PRE INSERT ${event.entityObject}"
        break
        case PostInsert:
            println "POST INSERT ${event.entityObject}"
        break
        case PreUpdate:
            println "PRE UPDATE ${event.entityObject}"
        break;
        case PostUpdate:
            println "POST UPDATE ${event.entityObject}"
        break;
        case PreDelete:
            println "PRE DELETE ${event.entityObject}"
        break;
        case PostDelete:
            println "POST DELETE ${event.entityObject}"
        break;
        case PreLoad:
            println "PRE LOAD ${event.entityObject}"
        break;
        case PostLoad:
            println "POST LOAD ${event.entityObject}"
        break;
    }
}
```

The AbstractPersistenceEvent class has many subclasses (PreInsertEvent
PostInsertEvent etc.) that provide further information specific to the event. A cancel() methc
also provided on the event which allows you to veto an insert, update or delete operation.

Once you have created your event listener you need to register it with the ApplicationContext. This
be done in BootStrap.groovy:

```
def init = {
    applicationContext.addApplicationListener(new MyPersistenceListener())
}
```

Hibernate Events

It is generally encouraged to use the non-Hibernate specific API described above, but if you need acces
more detailed Hibernate events then you can define custom Hibernate-specific event listeners.

You can also register event handler classes in an application's
`grails-app/conf/spring/resources.groovy` or in the `doWithSpring` closure in a pl
descriptor by registering a Spring bean named `hibernateEventListeners`. This bean has one prope
`listenerMap` which specifies the listeners to register for various Hibernate events.

The values of the Map are instances of classes that implement one or more Hibernate listener interfaces.
can use one class that implements all of the required interfaces, or one concrete class per interface, or
combination. The valid Map keys and corresponding interfaces are listed here:

Name	Interface
auto-flush	AutoFlushEventListener
merge	MergeEventListener
create	PersistEventListener
create-onflush	PersistEventListener
delete	DeleteEventListener
dirty-check	DirtyCheckEventListener
evict	EvictEventListener
flush	FlushEventListener
flush-entity	FlushEntityEventListener
load	LoadEventListener
load-collection	InitializeCollectionEventListener
lock	LockEventListener
refresh	RefreshEventListener
replicate	ReplicateEventListener
save-update	SaveOrUpdateEventListener
save	SaveOrUpdateEventListener
update	SaveOrUpdateEventListener
pre-load	PreLoadEventListener
pre-update	PreUpdateEventListener
pre-delete	PreDeleteEventListener
pre-insert	PreInsertEventListener
pre-collection-recreate	PreCollectionRecreateEventListener
pre-collection-remove	PreCollectionRemoveEventListener
pre-collection-update	PreCollectionUpdateEventListener
post-load	PostLoadEventListener
post-update	PostUpdateEventListener
post-delete	PostDeleteEventListener
post-insert	PostInsertEventListener
post-commit-update	PostUpdateEventListener
post-commit-delete	PostDeleteEventListener
post-commit-insert	PostInsertEventListener
post-collection-recreate	PostCollectionRecreateEventListener
post-collection-remove	PostCollectionRemoveEventListener
post-collection-update	PostCollectionUpdateEventListener

For example, you could register a class `AuditEventListener` which implemen
`PostInsertEventListener`, `PostUpdateEventListener`, and `PostDeleteEventListe`
using the following in an application:

```
beans = {

auditListener(AuditEventListener)

hibernateEventListeners(HibernateEventListeners) {
    listenerMap = ['post-insert': auditListener,
                   'post-update': auditListener,
                   'post-delete': auditListener]
    }
}
```

or use this in a plugin:

```
def doWithSpring = {

auditListener(AuditEventListener)

hibernateEventListeners(HibernateEventListeners) {
    listenerMap = ['post-insert': auditListener,
                   'post-update': auditListener,
                   'post-delete': auditListener]
    }
}
```

Automatic timestamping

The examples above demonstrated using events to update a `lastUpdated` and `dateCreated` proper
keep track of updates to objects. However, this is actually not necessary. By defining a `lastUpdated`
`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
   }
}
```

> ⛔ If you put `nullable: false` constraints on either `dateCreated` or
> `lastUpdated`, your domain instances will fail validation - probably not what you
> want. Leave constraints off these properties unless you have disabled automatic
> timestamping.

5.5.2 Custom ORM Mapping

Grails domain classes can be mapped onto many legacy schemas with an Object Relational Mapping (domain specific language). The following sections takes you through what is possible with the ORM DSL

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

```
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 to cha the name you can do:

```
class Person {

String firstName

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

Here `firstName` is a dynamic method within the `mapping` Closure that has a single Map parameter. S its name corresponds to a domain class persistent field, the parameter values (in this case just `"column"` used to configure the mapping for that property.

Column type

GORM supports configuration of Hibernate types with the DSL using the type attribute. This inclu
specifing user types that implement the Hibernate [org.hibernate.usertype.UserType](#) interface, which all
complete customization of how a type is persisted. As an example if you had a PostCodeType you c
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 chose
Grails you could use:

```
class Address {

String number
    String postCode

static mapping = {
        postCode type: 'text'
    }
}
```

This would make the postCode column map to the default large-text type for the database you're using
example TEXT or CLOB).

See the Hibernate documentation regarding [Basic Types](#) for further information.

Many-to-One/One-to-One Mappings

In the case of associations it is also possible to configure the foreign keys used to map associations. In the
of a many-to-one or one-to-one association this is exactly the same as any regular column. For exar
consider the following:

```
class Person {

String firstName
    Address address

static mapping = {
        table 'people'
        firstName column: 'First_Name'
        address column: 'Person_Address_Id'
    }
}
```

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

163

One-to-Many Mapping

With a bidirectional one-to-many you can change the foreign key column used by changing the column n
on the many side of the association as per the example in the previous section on one-to-one associati
However, with unidirectional associations the foreign key needs to be specified on the association itself.
example given a unidirectional one-to-many relationship between `Person` and `Address` the following
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
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
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 cal
`person_id` and `group_id` referencing the `person` and `group` tables. To change the column names
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
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 customize any of these settings, for example to use a distributed caching mechanism.

165

> ⚠ For further reading on caching and in particular Hibernate's second-level cache, refer to the [Hibernate documentation](#) on the subject.

Caching instances

Call the `cache` method in your mapping block to enable caching with the default settings:

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

This will configure a 'read-write' cache that includes both lazy and non-lazy properties. You can custom this further:

```
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 collect (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 `cache` argument:

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

> ⚠ In order for the results of the query to be cached, you must 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 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 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 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 use `table-per-hierarchy` inheritance mapping. This has the disadvantage that columns cannot have a `NOT-NULL` constraint applied to them at the database level. If you would prefer use a `table-per-subclass` inheritance strategy you can do so as follows:

```
class Payment {
    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-hierar`
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 r
on the native database mechanism for generating ids. This is by far the best approach, but there are still n
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
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 <u>Hibernate</u>
> <u>reference documentation</u>

Although you don't typically specify the `id` field (Grails adds it for you) you can still configure its map
like the other properties. For example to customise the column for the id property 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).
not an approach we recommend, but is available to you if you need it:

```
import org.apache.commons.lang.builder.HashCodeBuilder

class Person implements Serializable {

String firstName
    String lastName

boolean equals(other) {
        if (!(other instanceof Person)) {
            return false
        }
other.firstName == firstName && other.lastName == lastName
    }

int hashCode() {
        def builder = new HashCodeBuilder()
        builder.append firstName
        builder.append lastName
        builder.toHashCode()
    }

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

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

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

Domain classes mapped with composite primary keys must implement the `Serializable` interface
override the `equals` and `hashCode` methods, using the properties in the composite key for the calculati
The example above uses a `HashCodeBuilder` for convenience but it's fine to implement it yourself.

Another important consideration when using composite primary keys is associations. If for example you I
a many-to-one association where the foreign keys are stored in the associated table then 2 columns wil
present in the associated table.

For example consider the following domain class:

```
class Address {
    Person person
}
```

In this case the `address` table will have an additional two columns called `person_first_name`
`person_last_name`. If you wish the change the mapping of these columns then you can do so using
following technique:

```
class Address {
    Person person
    static mapping = {
        person {
            column: "FirstName"
            column: "LastName"
        }
    }
}
```

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 DSL you can specify which columns are used 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 examp index:'Name_Idx, Address_Index' will cause an error.

5.5.2.7 Optimistic Locking and Versioning

As discussed in the section on <u>Optimistic and Pessimistic Locking</u>, by default GORM uses optimistic loc and automatically injects a version property into every class which is in turn mapped to a vers column at the database level.

If you're mapping to a legacy schema that doesn't have version columns (or there's some other reason why don't want/need this feature) you can disable this with the version method:

```
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 data (due to data overriding) unless you use <u>pessimistic locking</u>

Version columns types

By default Grails maps the `version` property as a `Long` that gets incremented by one each time an inst:
is updated. But Hibernate also supports using a `Timestamp`, for example:

```
import java.sql.Timestamp
class Person {

…
    Timestamp version
static mapping = {
        table 'people'
    }
}
```

There's a slight risk that two updates occurring at nearly the same time on a fast server can end up with
same timestamp value but this risk is very low. One benefit of using a `Timestamp` instead of a `Long` is
you combine the optimistic locking and last-updated semantics into a single column.

5.5.2.8 Eager and Lazy Fetching

Lazy Collections

As discussed in the section on [Eager and Lazy fetching](), GORM collections are lazily loaded by default
you can change this behaviour with the ORM DSL. There are several options available to you, but the
common ones are:

- lazy: false
- fetch: 'join'

and they're used like this:

```
class Person {

String firstName
    Pet pet

static hasMany = [addresses: Address]

static mapping = {
        addresses lazy: false
        pet fetch: 'join'
    }
}
```

```
class Address {
    String street
    String postCode
}
```

```
class Pet {
    String name
}
```

The first option, `lazy: false`, ensures that when a `Person` instance is loaded, its addres
collection is loaded at the same time with a second SELECT. The second option is basically the same, ex
the collection is loaded with a JOIN rather than another SELECT. Typically you want to reduce the numbe
queries, so `fetch: 'join'` is the more appropriate option. On the other hand, it could feasibly be the r
expensive approach if your domain model and data result in more and larger results than would otherwis
necessary.

For more advanced users, the other settings available are:

1. batchSize: N

2. lazy: false, batchSize: N

where N is an integer. These let you fetch results in batches, with one query per batch. As a simple exam
consider this mapping for `Person`:

```
class Person {

String firstName
    Pet pet

static mapping = {
        pet batchSize: 5
    }
}
```

If a query returns multiple `Person` instances, then when we access the first `pet` property, Hibernate
fetch that `Pet` plus the four next ones. You can get the same behaviour with eager loading by combin
`batchSize` with the `lazy: false` option. You can find out more about these options in the [Hiber
user guide](#) and this [primer on fetching strategies](#). Note that ORM DSL does not currently support
"subselect" fetching strategy.

Lazy Single-Ended Associations

In GORM, one-to-one and many-to-one associations are by default lazy. Non-lazy single ended associat
can be problematic when you load many entities because each non-lazy association will result in an e
SELECT statement. If the associated entities also have non-lazy associations, the number of queries gr
significantly!

Use the same technique as for lazy collections to make a one-to-one or many-to-one associa
non-lazy/eager:

```
class Person {
    String firstName
}
```

```groovy
class Address {

String street
    String postCode

static belongsTo = [person: Person]

static mapping = {
        person lazy: false
    }
}
```

Here we configure GORM to load the associated `Person` instance (through the `person` property) when an `Address` is loaded.

Lazy Single-Ended Associations and Proxies

Hibernate uses runtime-generated proxies to facilitate single-ended lazy associations; Hibernate dynamic subclasses the entity class to create the proxy.

Consider the previous example but with a lazily-loaded `person` association: Hibernate will set the `per` property to a proxy that is a subclass of `Person`. When you call any of the getters (except for the property) or setters on that proxy, Hibernate will load the entity from the database.

Unfortunately this technique can produce surprising results. Consider the following example classes:

```groovy
class Pet {
    String name
}
```

```groovy
class Dog extends Pet {
}
```

```groovy
class Person {
    String name
    Pet pet
}
```

and assume that we have a single `Person` instance with a `Dog` as the `pet`. The following code will wor you would expect:

```groovy
def person = Person.get(1)
assert person.pet instanceof Dog
assert Pet.get(person.petId) instanceof Dog
```

But this won't:

```
def person = Person.get(1)
assert person.pet instanceof Dog
assert Pet.list()[0] instanceof Dog
```

The second assertion fails, and to add to the confusion, this will work:

```
assert Pet.list()[0] instanceof Dog
```

What's going on here? It's down to a combination of how proxies work and the guarantees that the Hiber
session makes. When you load the `Person` instance, Hibernate creates a proxy for its `pet` relation
attaches it to the session. Once that happens, whenever you retrieve that `Pet` instance with a query, a `get`
or the `pet` relation *within the same session* , Hibernate gives you the proxy.

Fortunately for us, GORM automatically unwraps the proxy when you use `get()` and `findBy*()`, or v
you directly access the relation. That means you don't have to worry at all about proxies in the majorit
cases. But GORM doesn't do that for objects returned with a query that returns a list, such as `list()`
`findAllBy*()`. However, if Hibernate hasn't attached the proxy to the session, those queries will return
real instances - hence why the last example works.

You can protect yourself to a degree from this problem by using the `instanceOf` method by GORM:

```
def person = Person.get(1)
assert Pet.list()[0].instanceOf(Dog)
```

However, it won't help here if casting is involved. For example, the following code will throw
`ClassCastException` because the first pet in the list is a proxy instance with a class that is neither
nor a sub-class of `Dog`:

```
def person = Person.get(1)
Dog pet = Pet.list()[0]
```

Of course, it's best not to use static types in this situation. If you use an untyped variable for the pet inst
you can access any `Dog` properties or methods on the instance without any problems.

These days it's rare that you will come across this issue, but it's best to be aware of it just in case. At least
will know why such an error occurs and be able to work around it.

5.5.2.9 Custom Cascade Behaviour

As described in the section on <u>cascading updates</u>, the primary mechanism to control the way updates
deletes cascade from one association to another is the static <u>belongsTo</u> property.

However, the ORM DSL gives you complete access to Hibernate's <u>transitive persistence</u> capabilities using
`cascade` attribute.

Valid settings for the cascade attribute include:

- `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

- `all-delete-orphan` - Applies only to one-to-many associations and indicates that when a chi
  removed from an association then it should be automatically deleted. Children are also deleted when
  parent is.

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

To specify the cascade attribute simply define one or more (comma-separated) of the aforementioned sett
as its value:

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

```groovy
class Address {
    String street
    String postCode
}
```

5.5.2.10 Custom Hibernate Types

You saw in an earlier section that you can use composition (with the `embedded` property) to break a
into multiple objects. You can achieve a similar effect with Hibernate's custom user types. These are
domain classes themselves, but plain Java or Groovy classes. Each of these types also has a correspon
"meta-type" class that implements <u>org.hibernate.usertype.UserType</u>.

The <u>Hibernate reference manual</u> has some information on custom types, but here we will focus on how to
them in Grails. Let's start by taking a look at a simple domain class that uses an old-fashioned (pre-Java
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 cus
mapping to the corresponding `UserType` implementation. That's all you have to do to start using
custom type. If you want, you can also use the other column settings such as "column" to change the col
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 wan
such cases you explicitly define in the mapping what columns to use, since Hibernate can only use
property name for a single column. Fortunately, Grails lets you map multiple columns to a property using
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'
pleased to know that you can also use some of the normal column/property mapping attributes in the col
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,`
`joinTable`.

One thing to bear in mind with custom types is that they define the *SQL types* for the corresponding data
columns. That helps take the burden of configuring them yourself, but what happens if you have a leg
database that uses a different SQL type for one of the columns? In that case, override the column's SQL
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 defau
"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 one that takes its value from a SQL expression, often but not necessarily based or
value of one or more other persistent properties. 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 is probabl
need to persist the tax property. The SQL used to derive the value of a derived property may be expresse
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
persistence model not the object model, which is why the example refers to PRICE and TAX_RATE instea
price and taxRate.

With that in place, when a Product is retrieved with something like Product.get(42), the SQL th
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 derived at runtime and not stored in the database it might seem that the same e
could be achieved by adding a method like `getTax()` to the `Product` class that simply returns the pro
of the `taxRate` and `price` properties. With an approach like that you would give up the ability quer
database based on the value of the `tax` property. Using a derived property allows exactly that. To retrieve
`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>?
```

> ⚠ 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
names to SQL table and column names by converting from camel-cased Strings to ones that use undersc
as word separators. You can customize these on a per-instance basis in the `mapping` closure but if ther
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 `hibern` 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 col 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 query arguments such as those found in the list method:

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

However, you can also declare the default sort order for a collection in the mapping:

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

The above means that all collections of `Airports` will by default be sorted by the airport name. If you want to change the sort *order* , use this syntax:

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

Finally, you can configure sorting at the association level:

```
class Airport {
    …
    static hasMany = [flights: Flight]

static mapping = {
        flights sort: 'number', order: 'desc'
    }
}
```

In this case, the `flights` collection will always be sorted in descending order of flight number.

> ⛔ These mappings will not work for default unidirectional one-to-many or many-to-many relationships because they involve a join table. See this issue for more details. Consider using a `SortedSet` or queries with sort parameters to fetch the data you need.

5.6 Programmatic Transactions

Grails is built on Spring and uses Spring's Transaction abstraction for dealing with programmatic transacti
However, GORM classes have been enhanced to make this simpler with the withTransaction method.
method has a single parameter, a Closure, which has a single parameter which is a Spring TransactionSt
instance.

Here's an example of using `withTransaction` in a controller methods:

```
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) {
            if (dest.active) {
                source.balance -= amount
                dest.amount += amount
            }
            else {
                status.setRollbackOnly()
            }
        }
    }
}
```

In this example we rollback the transaction if the destination account is not active. Also, if an unchec
`Exception` or `Error` (but not a checked `Exception`, even though Groovy doesn't require that you c
checked exceptions) is 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 wan
rollback the entire transaction. This can be achieved through the use of Spring's [SavePointManager](#) interfac

The `withTransaction` method deals with the begin/commit/rollback logic for you within the scope o
block.

5.7 GORM and Constraints

Although constraints are covered in the [Validation](#) section, it is important to mention them here as some o
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 fo
corresponding domain class properties.

Consider the following example. Suppose we have a domain model with the following properties:

```
String name
String description
```

By default, in MySQL, Grails would define these columns as

Column	Data Type
name	varchar(255)
description	varchar(255)

But perhaps the business rules for this domain class state that a description can be up to 1000 characte
length. If that were the case, we would likely define the column as follows *if* we were creating the table
an SQL script.

Column	Data Type
description	TEXT

Chances are we would also want to have some application-based validation to make sure we don't exceed
1000 character limit *before* we persist any records. In Grails, we achieve this validation with [constraints](#).
would 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
schema to be generated as shown above. Below is a description of the other constraints that influence sch
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 the constraint value.

In general, it's not advisable to use both constraints on the same domain class property. However, if both `maxSize` constraint and the `size` constraint are defined, then Grails sets the column length to the minir of the `maxSize` constraint and the upper bound of the size constraint. (Grails uses the minimum of the 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 G sets the maximum column length based on the length of the longest string in the list of valid values. example, given a list including values "Java", "Groovy", and "C++", Grails would set the column length (i.e., the number of characters in the string "Groovy").

Constraints Affecting Numeric Properties

- min

- max

- range

If the `max`, `min`, or `range` constraint is defined, Grails attempts to set the column precision based on constraint value. (The success of this attempted influence is largely dependent on how Hibernate interacts the underlying DBMS.)

In general, it's not advisable to combine the pair `min`/`max` and `range` constraints together on the s domain class property. However, if both of these constraints is defined, then Grails uses the minir precision value from the constraints. (Grails uses the minimum of the two, because any length that exc 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 va This rule only applies to floating point numbers (i.e., `java.lang.Float`, `java.Lang.Doub` `java.lang.BigDecimal`, or subclasses of `java.lang.BigDecimal`). The success of this attem 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 numbe digits for use in the precision. Keep in mind that specifying only one of `min`/`max` constraints will not a schema generation (since there could be large negative value of property with max:100, for example), ur the specified constraint value requires more digits than default Hibernate column precision is (19 at 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. A controller can generate the response directly or delegate to a view. To create a controller, simply create a class whose name ends with Controller in the `grails-app/controllers` directory (in a subdirectory if it's in a package).

The default [URL Mapping](#) configuration ensures that the first part of your controller name is mapped to a and each action defined within your controller maps to URIs within the controller name URI.

6.1.1 Understanding Controllers and Actions

Creating a controller

Controllers can be created with the [create-controller](#) or [generate-controller](#) command. For example try run the following command from the root of a Grails project:

```
grails create-controller book
```

The command will create a controller at the location grails-app/controllers/myapp/BookController.groovy:

```groovy
package myapp
class BookController {
def index() { }
}
```

where "myapp" will be the name of your application, the default package name if one isn't specified.

BookController by default maps to the /book URI (relative to your application root).

> ⚠ The `create-controller` and `generate-controller` commands are just for convenience and you can just as easily create controllers using your favorite text editor or IDE

Creating Actions

A controller can have multiple public action methods; each one 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.

Public Methods as Actions

In earlier versions of Grails actions were implemented with Closures. This is still supported, but the prefe
approach is to use methods.

Leveraging methods instead of Closure properties has some advantages:

- Memory efficient

- Allow use of stateless controllers (singleton scope)

- You can override actions from subclasses and call the overridden superclass method w
  super.actionName()

- Methods can be intercepted with standard proxying mechanisms, something that is complicated t
  with Closures since they're fields.

If you prefer the Closure syntax or have older controller classes created in earlier versions of Grails and
want   the   advantages   of   using   methods,   you   can   set   the
grails.compile.artefacts.closures.convert property to true in BuildConfig.groov

```
grails.compile.artefacts.closures.convert = true
```

and a compile-time AST transformation will convert your Closures to methods in the generated bytecode.

> ⚠ If a controller class extends some other class which is not defined under the
> grails-app/controllers/ directory, methods inherited from that class are not
> converted to controller actions. If the intent is to expose those inherited methods as
> controller actions the methods may be overridden in the subclass and the subclass
> method may invoke the method in the super class.

The Default Action

A controller has the concept of a default URI that maps to the root URI of the controller, for example /b
for BookController. The action that is called when the default URI is requested is dictated by
following rules:

- If there is only one action, it's the default

- If you have an action named `index`, it's the default

- Alternatively you can set it explicitly with the `defaultAction` property:

```
static defaultAction = "list"
```

6.1.2 Controllers and Scopes

Available Scopes

Scopes are hash-like objects where you can store variables. The following scopes are available to controlle

- [servletContext](#) - Also known as application scope, this scope lets you share state across the entire application. The servletContext is an instance of [ServletContext](#)

- [session](#) - The session allows associating state with a given user and typically uses cookies to associa 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 ob is an instance of [HttpServletRequest](#)

- [params](#) - Mutable map of incoming request query string or POST parameters

- [flash](#) - See below

Accessing Scopes

Scopes can be accessed using the variable names above in combination with Groovy's array index oper 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 also access values within scopes using the de-reference operator, making the syntax even more cl

```
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 as a temporary store to make attributes available for this request the next request only. Afterwards the attributes are cleared. This is useful for setting a message directly before redirecting, 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
}
```

When the `list` action is requested, the `message` value will be in scope and can be used to display information message. It will be removed from the `flash` scope after this second request.

Note that the attribute name can be anything you want, and the values are often strings used to dis messages, but can be any object type.

Scoped Controllers

By default, a new controller instance is created for each request. In fact, because the controller is `protot` scoped, it is thread-safe since each request happens on its own thread.

You can change this behaviour by placing a controller in a particular scope. The supported scopes are:

- `prototype` (default) - A new controller will be created for each request (recommended for actior Closure properties)

- `session` - One controller is created for the scope of a user session

- `singleton` - Only one instance of the controller ever exists (recommended for actions as methods)

To enable one of the scopes, add a static `scope` property to your class with one of the valid scope va listed above, for example

```
static scope = "singleton"
```

You can define the default strategy under in `Config.groovy` with th `grails.controllers.defaultScope` key, for example:

```
grails.controllers.defaultScope = "singleton"
```

> ⚠ Use scoped controllers wisely. For instance, we don't recommend having any properties in a singleton-scoped controller since they will be shared for *all* requests. Setting a default scope other than `prototype` may also lead to unexpected behaviors if you have controllers provided by installed plugins that expect that the scope is `prototype`.

6.1.3 Models and Views

Returning the Model

A model is a Map that the view uses when rendering. The keys within that Map correspond to variable na accessible by the view. There are a couple of ways to return a model. First, you can explicitly return a I instance:

```
def show() {
    [book: Book.get(params.id)]
}
```

> ⚠ The above does *not* reflect what you should use with the scaffolding views - see the [scaffolding section](scaffolding) for more details.

If no explicit model is returned the controller's properties will be used as the model, thus allowing you to 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, and all users would share the same data.

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() {
    // get some books just for the index page, perhaps your favorites
    def favoriteBooks = ...

// forward to the list view to show them
    return new ModelAndView("/book/list", [ bookList : favoriteBooks ])
}
```

One thing to bear in mind is that certain variable names can not be used in your model:

- `attributes`

- `application`

Currently, no error will be reported if you do use them, but this will hopefully change in a future version Grails.

Selecting the View

In both of the previous two examples there was no code that specified which [view](#) to render. So how Grails know which one to pick? The answer lies in the conventions. Grails will look for a view at the location `grails-app/views/book/show.gsp` for this `list` action:

```
class BookController {
    def show() {
        [book: Book.get(params.id)]
    }
}
```

To render a different view, use the [render](#) method:

```
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` directory of the `grails-app/views` directory. This is convenient, but to access sh views you need instead you can use an absolute path instead of a relative one:

```
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`.

Grails also supports JSPs as views, so if a GSP isn't found in the expected location but a JSP is, it will be used instead.

Rendering a Response

Sometimes it's easier (for example 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, be careful of naming clashes between HTML elements and Grails tags, for example:

```
import groovy.xml.MarkupBuilder
…
def login() {
    def writer = new StringWriter()
    def builder = new MarkupBuilder(writer)
    builder.html {
        head {
            title 'Log in'
        }
        body {
            h1 'Hello'
            form {
            }
        }
    }

def html = writer.toString()
    render html
}
```

This will actually [call the form tag](#) (which will return some text that will be ignored by the `MarkupBuil`
). To correctly output a `<form>` element, use the following:

```
def login() {
    // …
    body {
        h1 'Hello'
        builder.form {
        }
    }
    // …
}
```

6.1.4 Redirects and Chaining

Redirects

Actions can be redirected using the [redirect](#) controller method:

```
class OverviewController {

def login() {}

def find() {
        if (!session.user)
            redirect(action: 'login')
            return
        }
        …
    }
}
```

Internally the [redirect](#) method uses the [HttpServletResponse](#) object's `sendRedirect` method.

The `redirect` method expects one of:

- Another closure within the same controller class:

191

```
// Call the login action within the same class
redirect(action: login)
```

- The name of an action (and controller name if the redirect isn't to an action in the current controller):

```
// 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 optionally be passed from one action to the next using the `params` argument of the metho

```
redirect(action: 'myaction', params: [myparam: "myvalue"])
```

These parameters are made available through the [params](#) dynamic property that accesses request parame
If a parameter is specified with the same name as a request parameter, the request parameter is overridden
the controller parameter is used.

Since the `params` object is a Map, you can use it to pass the current request parameters from one actio
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")
```

which 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.
example calling the `first` action in this 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 using the `chainModel` map.
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 ca:
achieved with action interceptors. There are currently two 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` the:
intercepted action will not be executed. The interceptor can be defined for all actions in a controlle
follows:

193
```

```
def beforeInterceptor = {
    println "Tracing action ${actionUri}"
}
```

The above is declared inside the body of the controller definition. It will be executed before all actions does not interfere with processing. A common use case is very simplistic authentication:

```
def beforeInterceptor = [action: this.&auth, except: 'login']

// defined with private scope, so it's not considered an action
private auth() {
    if (!session.user) {
        redirect(action: 'login')
        return false
    }
}

def login() {
    // display login page
}
```

The above code defines a method called `auth`. A private method is used so that it is not exposed as an ac to the outside world. The `beforeInterceptor` then defines an interceptor that is used on all acti *except* the login action and it executes the `auth` method. The `auth` method is referenced using Groo method pointer syntax. Within the method it detects whether there is a user in the session, and if n redirects to the `login` action and returns `false`, causing the intercepted action to not be processed.

After Interception

Use the `afterInterceptor` property to define an interceptor that is executed after an action:

```
def afterInterceptor = { model ->
    println "Tracing action ${actionUri}"
}
```

The after interceptor takes the resulting model as an argument and can hence manipulate the mode response.

An after interceptor may also modify the Spring MVC [ModelAndView](#) object prior to rendering. In this 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 `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 w executing the interceptor (interceptors are called 'filters' in Rails; this terminology conflicts with Servlet f terminology in Java):

```
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 define follows:

```
def beforeInterceptor = [action: this.&auth, except: ['login', 'register']]
```

The other supported condition is 'only', this executes the interceptor for only the specified action(s):

```
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 e graph of objects. Data binding should deal with all necessary type conversion since request parameters, w are typically delivered by a form submission, are always strings whilst the properties of a Groovy or 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 usi domain classes' Map 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 domain class constructor Grails automatically recognizes that you are trying to bind from request parame 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 be set on the domain class. You use the properties property to perform data binding onto an existing instance:

195

```
def save() {
    def b = Book.get(params.id)
    b.properties = params
    b.save()
}
```

This has 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 capabilit 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 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 bin depending of the association type.

If you have a `Set` based association (the default for a `hasMany`) then the simplest way to populat association is to 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 lets you select multiple values. In this case if you submit the form Grails 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 technique won't work. Instead you 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 to do the update in. This is because a `Set` has no concept of order, so although we're referring to `books0` `books1` it is not guaranteed that the order of the association will be correct on the server side unless 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 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 two and you refer to an element that is out 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 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 crea 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 with a single-ended association. For example:

```
<g:select name="books[0].id" from="${bookList}"
          value="${author?.books[0]?.id}" />

<g:select name="books[1].id" from="${bookList}"
          value="${author?.books[1]?.id}" />

<g:select name="books[2].id" from="${bookList}"
          value="${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}"
          noSelection="['null': '']"/>
```

Will render a select box that will remove the association at books 0 if the empty option is chosen.

Binding to a `Map` property works the same way except that the list index in the parameter name is replace the map key:

```
<g:select name="images[cover].id"
          from="${Image.list()}"
          value="${book?.images[cover]?.id}"
          noSelection="['null': '']"/>
```

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. book. which is used to isolate which parameters belong to which type. Grails' `params` object is li multi-dimensional hash and you can index into it 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 param below this level to bind. We could do the same with an `Author` domain class:

```
def a = new Author(params.author)
```

Data Binding and Action Arguments

Controller action arguments are subject to request parameter data binding. There are 2 categories of contr action arguments. The first category is command objects. Complex types are treated as command objects. the [Command Objects](#) section of the user guide for details. The other category is basic object types. Suppc types are the 8 primitives, their corresponding type wrappers and [java.lang.String](#). The default behavior map request parameters to action arguments by name:

```
class AccountingController {

// accountNumber will be initialized with the value of params.accountNumber
    // accountType will be initialized with params.accountType
    def displayInvoice(String accountNumber, int accountType) {
        // …
    }
}
```

For primitive arguments and arguments which are instances of any of the primitive type wrapper classes a conversion has to be carried out before the request parameter value can be bound to the action argument. type conversion happens automatically. In a case like the example shown above, t params.accountType request parameter has to be converted to an int. If type conversion fails for reason, the argument will have its default value per normal Java behavior (null for type wrapper referer false for booleans and zero for numbers) and a corresponding error will be added to the errors propert the defining controller.

```
/accounting/displayInvoice?accountNumber=B59786&accountType=bogusValue
```

Since "bogusValue" cannot be converted to type int, the value of accountType will be ze controller.errors.hasErrors() will be true, controller.errors.errorCount wil equal to 1 and controller.errors.getFieldError('accountType') will contain corresponding error.

If the argument name does not match the name of the request parameter then t @grails.web.RequestParameter annotation may be applied to an argument to express the nam the request parameter which should be bound to that argument:

```
import grails.web.RequestParameter

class AccountingController {

// mainAccountNumber will be initialized with the value of params.accountNumber
    // accountType will be initialized with params.accountType
    def displayInvoice(@RequestParameter('accountNumber') String mainAccountNumber
int accountType) {
        // …
    }
}
```

Data binding and type conversion errors

Sometimes when performing data binding it is not possible to convert a particular String into a partic target type. This results in a type conversion error. Grails will retain type conversion errors inside the e property of a Grails domain class. For example:

```
class Book {
    …
    URL publisherURL
}
```

Here we have a domain class Book that uses the java.net.URL class to represent URLs. Giver incoming request such as:

```
/book/save?publisherURL=a-bad-url
```

199

it is not possible to bind the string `a-bad-url` to the `publisherURL` property as a type mismatch 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 URL!"
}
```

Although we have not yet covered error codes (for more information see the section on [Validation](#)), for conversion errors you would want a message from the `grails-app/i18n/messages.propert` file to use for the error. 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 malicious data to domain classes and be persisted in the database. You can limit what properties are boun 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 is to use [Command Objects](#) as the target of data binding instead of domain clas 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)
```

The `bindData` method also lets you 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]])
```

> ⚠ Note that if an empty List is provided as a value for the `include` parameter then all fields will be subject to binding if they are not explicitly excluded.

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 is 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>
```

Be careful to avoid naming conflicts when using mark-up building. For example this code would produc
error:

```
def list() {

def books = Book.list()  // naming conflict here

render(contentType: "text/xml") {
        books {
            for (b in results) {
                book(title: b.title)
            }
        }
    }
}
```

This is because 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"}
]
```

The same dangers with naming conflicts described above for XML also apply to JSON building.

Automatic XML Marshalling

Grails also supports automatic marshalling of <u>domain classes</u> to XML using 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 provi
encodeAsXML and encodeAsJSON methods:

```
def xml = Book.list().encodeAsXML()
render xml
```

For more information on XML marshalling see the section on REST

Automatic JSON Marshalling

Grails also supports automatic marshalling to JSON using the same mechanism. Simply substitute XML v
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
JSON responses. Whilst the XML builder used by Grails is the standard XmlSlurper found in Groovy
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 deprecated [grails.web.JSONBuilder](#) class is used. This section covers the usage of the Grails 1.2 JSONBui

For backwards compatibility the old `JSONBuilder` class is used with the `render` method for applications; to use the newer/better `JSONBuilder` class set 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"}] }
```

Use the special `element` method to return a list as the root:

```
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" } ]
}
```

You can use the `array` method to build them up dynamically:

```
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 direc

```
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 using Spring's [MultipartHttpServletRequest](#) interface. The first step for
uploading is to create a multipart form like this:

```
Upload Form: <br />
    <g:uploadForm action="upload">
        <input type="file" name="myFile" />
        <input type="submit" />
    </g:uploadForm>
```

The `uploadForm` tag conveniently adds the `enctype="multipart/form-data"` attribute to
standard `<g:form>` tag.

There are then a number of ways to handle the file upload. One is to work with the Spring [Multipart](#)
instance directly:

```
def upload() {
    def f = request.getFile('myFile')
    if (f.empty) {
        flash.message = 'file cannot be empty'
        render(view: 'uploadForm')
        return
    }

f.transferTo(new File('/some/local/dir/myfile.txt'))
    response.sendError(200, 'Done')
}
```

This is convenient for doing transfers to other destinations and manipulating the file directly as you can ob
an `InputStream` and so on with the [MultipartFile](#) interface.

File Uploads through Data Binding

File uploads can also be performed using data binding. Consider this `Image` domain class:

```
class Image {
    byte[] myFile

static constraints = {
        // Limit upload file size to 2MB
        myFile maxSize: 1024 * 1024 * 2
    }
}
```

If you create an image using the `params` object in the constructor as in the example below, Grails automatically bind the file's contents as a `byte` to the `myFile` property:

```
def img = new Image(params)
```

It's important that you set the [size](#) or [maxSize](#) constraints, otherwise your database may be created wi small column size that can't handle reasonably sized files. For example, both H2 and MySQL default to a size of 255 bytes for `byte` properties.

It is also possible to set the contents of the file as a string by changing the type of the `myFile` property of 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 framework like Struts, and they are useful for populating a subset of the properties needed to update a dom class. Or where there is no domain class required for the interaction, but you need features such as [binding](#) and [validation](#).

Declaring Command Objects

Command objects are typically declared in the same source file as a controller, directly below the contr 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 this example shows, you can define constraints in command objects just like in domain classes.

Using Command Objects

To use command objects, controller actions may optionally specify any number of command ob
parameters. The parameter types must be supplied so that Grails knows what objects to create, populate
validate.

Before the controller action is executed Grails will automatically create an instance of the command ob
class, populate its properties with by binding the request parameters, and validate the command object.
example:

```
class LoginController {

def login = { LoginCommand cmd ->
        if (cmd.hasErrors()) {
            redirect(action: 'loginForm')
            return
        }

// work with the command object data
    }
}
```

When using methods instead of Closures for actions, you can specify command objects in arguments:

```
class LoginController {
    def login(LoginCommand cmd) {
        if (cmd.hasErrors()) {
            redirect(action: 'loginForm')
            return
        }

// work with the command object data
    }
}
```

Command Objects and Dependency Injection

Command objects can participate in dependency injection. This is useful if your command object has s
custom validation logic uses 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 the loginService bean which is injected by name 1
the Spring ApplicationContext.

6.1.11 Handling Duplicate Form Submissions

Grails has built-in support for handling duplicate form submissions using the "Synchronizer Token Patt
To get started you 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 G
will store the invalid token in a flash.invalidToken variable and redirect the request back to
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 or
> clustered sessions 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 parame
(typically Strings) into another more appropriate type the [params](#) object has a number of convenience metl
for each type:

```
def total = params.int('total')
```

The above example uses the `int` method, and there are also methods for `boolean`, `long`, `char`, `sh`
and so on. Each of these methods is null-safe and safe from any parsing errors, so you don't have to perf
any additional checks on the parameters.

Each of the conversion methods allows a default value to be passed as an optional second argument.
default value will be returned if a corresponding entry cannot be found in the map or if an error occurs du
the conversion. Example:

```
def total = params.int('total', 42)
```

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 coul
a query string such as `?name=Bob&name=Judy`.

In this case dealing with one parameter and dealing with many has different semantics since Groovy's itera
mechanics for `String` iterate over each character. To avoid this problem the [params](#) object provides a `l`
method that always returns a list:

```
for (name in params.list('name')) {
    println name
}
```

6.1.13 Asynchronous Request Processing

Grails support asynchronous request processing as provided by the Servlet 3.0 specification. To enable
async features you need to set your servlet target version to 3.0 in BuildConfig.groovy:

```
grails.servlet.version = "3.0"
```

With that done ensure you do a clean re-compile as some async features are enabled at compile time.

> ⚠ With a Servlet target version of 3.0 you can only deploy on Servlet 3.0 containers such
> as Tomcat 7 and above.

Asynchronous Rendering

You can render content (templates, binary data etc.) in an asynchronous manner by calling the `startAs` method which returns an instance of the Servlet 3.0 `AsyncContext`. Once you have a reference to `AsyncContext` you can use Grails' regular render method to render content:

```
def index() {
    def ctx = startAsync()
    ctx.start {
        new Book(title:"The Stand").save()
        render template:"books", model:[books:Book.list()]
        ctx.complete()
    }
}
```

Note that you must call the `complete()` method to terminate the connection.

Resuming an Async Request

You resume processing of an async request (for example to delegate to view rendering) by using `dispatch` method of the `AsyncContext` class:

```
def index() {
    def ctx = startAsync()
    ctx.start {
        // do working
        …
        // render view
        ctx.dispatch()
    }
}
```

6.2 Groovy Server Pages

Groovy Servers Pages (or GSP for short) is Grails' view technology. It is designed to be familiar for user technologies such as ASP and JSP, but to be far more flexible and intuitive.

GSPs live in the `grails-app/views` directory and are typically rendered automatically (by convention with 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 mod 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
then be referenced 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 c
some basic syntax that users of JSP and ASP should be familiar with.

GSP supports the usage of `<% %>` scriptlet blocks to embed Groovy code (again this is discouraged):

```
<html>
    <body>
        <% out << "Hello GSP!" %>
    </body>
</html>
```

You can also use the `<%= %>` syntax to output values:

```
<html>
    <body>
        <%="Hello GSP!" %>
    </body>
</html>
```

GSP also supports JSP-style server-side comments (which are not rendered in the HTML response) as
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 declare variables:

```
<% now = new Date() %>
```

and then access those variables later in the page:

```
<%=now%>
```

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 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.

213

The import directive lets you import classes into the page. However, it is rarely needed due to Groc 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. A (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 `${..}` block. Variables within `${..}` block are **not** escaped by default, so any HTML in the variable's string is rendered directly to the p To reduce the risk of Cross-site-scripting (XSS) attacks, you can enable automatic HTML escaping with `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 which are the preferred 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 specify any tag library imports. If a 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 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 r java.util.Date instance). You can also use the body of the <g:set> tag to define a variable:

```
<g:set var="myHTML">
    Some re-usable code on: ${new Date()}
</g:set>
```

Variables can also be placed in one of the following scopes:

215

- `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 specify the scope, 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](#) tags for
with branching:

```
<g:if test="${session.role == 'admin'}">
    <%-- show administrative functions --%>
</g:if>
<g:else>
    <%-- show basic functions --%>
</g:else>
```

Use the [each](#) and [while](#) tags for iteration:

```
<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. Use the [findAll](#) and [grep](#) tags
these tasks:

```
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. The [grep](#) tag does a similar
for example filtering by class:

```
<g:grep in="${books}" filter="NonFictionBooks.class">
    <p>Title: ${it.title}</p>
</g:grep>
```

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 an XPath-like language in Gro
The books variable is a collection of Book instances. Since each Book has a title, you can obtain a li
Book titles using the expression books.title. Groovy will auto-magically iterate the collection, ol
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 lets you spe
controller and action name pairing and it will automatically work out the link based on the URL Mappi
even if you change them! For example:

```
<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 params="[sort: 'title', order: 'asc', author: currentBook.author]"
        action="list">Book List</g:link>
```

6.2.2.5 Forms and Fields

Form Basics

GSP supports many different tags for working with HTML forms and fields, the most basic of which is
form tag. This is a controller/action aware version of the regular HTML form tag. The url attribute lets
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. Bey
that all of the usual HTML attributes apply.

Form Fields

217

In addition to easy construction of forms, GSP supports custom tags for dealing with different types of fi[elds] including:

- [textField](#) - For input fields of type 'text'
- [passwordField](#) - For input fields of type 'password'
- [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 allows GSP expressions for 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 [r] tags), [localeSelect](#), [currencySelect](#) and [timeZoneSelect](#) (for selecting locales, currencies and time z[ones] respectively).

Multiple Submit Buttons

The age old problem of dealing with multiple submit buttons is also handled elegantly with Grails using [actionSubmit](#) tag. It is just like a regular submit, but lets you 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 e[ither] regular tags or as method calls from [controllers](#), [tag libraries](#) or GSP views.

Tags as method calls from GSPs

Tags return their results as a String-like object (a `StreamCharBuffer` which has all of the same met[hods] as String) instead of writing directly to the response when called as methods. For example:

```
Static Resource: ${createLinkTo(dir: "images", file: "logo.jpg")}
```

This is particularly useful for using 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 m quickly and often has an adverse effect of WYSWIG tools such as Dreamweaver that attempt to render 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](#) ca invoked without the prefix and a StreamCharBuffer result is returned:

```
def imageLocation = createLinkTo(dir:"images", file:"logo.jpg").toString()
```

Prefix the namespace to avoid naming conflicts:

```
def imageLocation = g.createLinkTo(dir:"images", file:"logo.jpg").toString()
```

For tags that use a [custom namespace](#), use that prefix for the method call. For example (from the [FCK E](#) [plugin](#)):

```
def editor = fckeditor.editor(name: "text", width: "100%", height: "400")
```

6.2.3 Views and Templates

Grails also has the concept of templates. These are useful for partitioning your views into maintain chunks, and combined with [Layouts](#) provide a highly re-usable mechanism for structured views.

Template Basics

Grails uses the convention of placing an underscore before the name of a view to identify it as a template. example, you might have a template that renders Books located a grails-app/views/book/_bookTemplate.gsp:

```
<div class="book" id="${book?.id}">
   <div>Title: ${book?.title}</div>
   <div>Author: ${book?.author?.name}</div>
</div>
```

Use the [render](#) tag to render this template from one of the views in grails-app/views/book:

```
<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 mult
`Book` instances you can also render the template for each `Book` using the render tag with a `collect`
attribute:

```
<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 view
`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
location, and then with the template attribute use an absolute location starting with / instead of a rela
location. For example if you had a template called
`grails-app/views/shared/_mySharedTemplate.gsp`, you would reference it as:

```
<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 u
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](#) controller method. This is useful for
applications where you generate small HTML or data responses to partially update the current page instea
performing new request:

```
def bookData() {
    def b = Book.get(params.id)
    render(template:"bookTemplate", model:[book:b])
}
```

The [render](#) controller method writes directly to the response, which is the most common behaviour. To ins
obtain the result of template as a String you can use the [render](#) tag:

```
def bookData() {
    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
[render](#) method.

6.2.4 Layouts with Sitemesh

Creating Layouts

Grails leverages [Sitemesh](#), a decorator engine, to support view layouts. Layouts are located in
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 invocations:

- layoutTitle - outputs the target page's title

- layoutHead - outputs the target page's head tag contents

- layoutBody - outputs the target page's body tag contents

The previous example also demonstrates the [pageProperty](#) tag which can be used to inspect and return asp
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:

221

```
<html>
    <head>
        <title>An Example Page</title>
        <meta name="layout" content="main" />
    </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 pag
we were to use the layout from the previous section the output would resemble this:

```
<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 "lay
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 app
to all views that the `BookController` delegates to. The value of the "layout" property may conta
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 this controller

```
class BookController {
    def list() { … }
}
```

You can create a layout called `grails-app/views/layouts/book.gsp`, which will be applied t
views that the `BookController` delegates to.

Alternatively, you can create a layout called `grails-app/views/layouts/book/list.gsp` w
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 preced
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 fo
application default layout which is `grails-app/views/layouts/application.gsp`. The nam
the application default layout may be changed by defining a property i
`grails-app/conf/Config.groovy` as follows:

```
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. This can be used to a
a layout to a template, URL or arbitrary section of content. This lets you even further modularize your v
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 inc
external content in the current page you use the [include](#) tag:

```
<g:include controller="book" action="list" />
```

You can even combine the [include](#) tag and the [applyLayout](#) tag for added flexibility:

223

```
<g:applyLayout name="myLayout">
    <g:include controller="book" action="list" />
</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 Static Resources

Grails 2.0 integrates with the [Resources plugin](#) to provide sophisticated static resource management. plugin is installed by default in new Grails applications.

The basic way to include a link to a static resource in your application is to use the [resource](#) tag. This si approach creates a URI pointing to the file.

However modern applications with dependencies on multiple JavaScript and CSS libraries and framework well as dependencies on multiple Grails plugins) require something more powerful.

The issues that the Resources framework tackles are:

- Web application performance tuning is difficult

- Correct ordering of resources, and deferred inclusion of JavaScript

- Resources that depend on others that must be loaded first

- The need for a standard way to expose static resources in plugins and applications

- The need for an extensible processing chain to optimize resources

- Preventing multiple inclusion of the same resource

The plugin achieves this by introducing new artefacts and processing the resources using the server's local system.

It adds artefacts for declaring resources, for declaring "mappers" that can process resources, and a servlet to serve processed resources.

What you get is an incredibly advanced resource system that enables you to easily create highly optim web applications that run the same in development and in production.

The Resources plugin documentation provides a more detailed overview of the [concepts](#) which wil beneficial when reading the following guide.

6.2.5.1 Including resources using the resource tags

Pulling in resources with r:require

To use resources, your GSP page must indicate which resource modules it requires. For example with [jQuery plugin](#), which exposes a "jquery" resource module, to use jQuery in any page on your site you si add:

```
<html>
    <head>
        <r:require module="jquery"/>
        <r:layoutResources/>
    </head>
    <body>
        …
        <r:layoutResources/>
    </body>
</html>
```

This will automatically include all resources needed for jQuery, including them at the correct locations in page. By default the plugin sets the disposition to be "head", so they load early in the page.

You can call `r:require` multiple times in a GSP page, and you use the "modules" attribute to provide of modules:

```
<html>
    <head>
        <r:require modules="jquery, main, blueprint, charting"/>
        <r:layoutResources/>
    </head>
    <body>
        …
        <r:layoutResources/>
    </body>
</html>
```

The above may result in many JavaScript and CSS files being included, in the correct order, with s JavaScript files loading at the end of the body to improve the apparent page load time.

However you cannot use r:require in isolation - as per the examples you must have the <r:layoutResourc tag to actually perform the render.

Rendering the links to resources with r:layoutResources

When you have declared the resource modules that your GSP page requires, the framework needs to re the links to those resources at the correct time.

To achieve this correctly, you must include the r:layoutResources tag twice in your page, or more commo in your GSP layout:

```
<html>
    <head>
        <g:layoutTitle/>
        <r:layoutResources/>
    </head>
    <body>
        <g:layoutBody/>
        <r:layoutResources/>
    </body>
</html>
```

This represents the simplest Sitemesh layout you can have that supports Resources.

The Resources framework has the concept of a "disposition" for every resource. This is an indication of w
in the page the resource should be included.

The default disposition applied depends on the type of resource. All CSS must be rendered in <head
HTML, so "head" is the default for all CSS, and will be rendered by the first r:layoutResources. Page
times are improved when JavaScript is loaded after the page content, so the default for JavaScript file
"defer", which means it is rendered when the second r:layoutResources is invoked.

Note that both your GSP page and your Sitemesh layout (as well as any GSP template fragments) can
r:require to depend on resources. The only limitation is that you must call r:require before
r:layoutResources that should render it.

Adding page-specific JavaScript code with r:script

Grails has the [javascript](#) tag which is adapted to defer to Resources plugin if installed, but it is recommen
that you call `r:script` directly when you need to include fragments of JavaScript code.

This lets you write some "inline" JavaScript which is actually **not** rendered inline, but either in the <head
at the end of the body, based on the disposition.

Given a Sitemesh layout like this:

```html
<html>
    <head>
        <g:layoutTitle/>
        <r:layoutResources/>
    </head>
    <body>
        <g:layoutBody/>
        <r:layoutResources/>
    </body>
</html>
```

...in your GSP you can inject some JavaScript code into the head or deferred regions of the page like this:

```html
<html>
    <head>
        <title>Testing r:script magic!</title>
    </head>
    <body>
        <r:script disposition="head">
          window.alert('This is at the end of <head>');
        </r:script>
        <r:script disposition="defer">
          window.alert('This is at the end of the body, and the page has loaded.')
        </r:script>
    </body>
</html>
```

The default disposition is "defer", so the disposition in the latter r:script is purely included for demonstratio

Note that such r:script code fragments **always** load after any modules that you have used, to ensure that
required libraries have loaded.

Linking to images with r:img

This tag is used to render `<img>` markup, using the Resources framework to process the resource on the (if configured to do so - e.g. make it eternally cacheable).

This includes any extra attributes on the `<img>` tag if the resource has been previously declared in a modu

With this mechanism you can specify the width, height and any other attributes in the resource declaratic the module, and they will be pulled in as necessary.

Example:

```html
<html>
    <head>
        <title>Testing r:img</title>
    </head>
    <body>
        <r:img uri="/images/logo.png"/>
    </body>
</html>
```

Note that Grails has a built-in `g:img` tag as a shortcut for rendering `<img>` tags that refer to a static resou The Grails [img](#) tag is Resources-aware and will delegate to `r:img` if found. However it is recommended you use `r:img` directly if using the Resources plugin.

Alongside the regular Grails [resource](#) tag attributes, this also supports the "uri" attribute for increased brev

See [r:resource documentation](#) for full details.

6.2.5.2 Other resource tags

r:resource

This is equivalent to the Grails [resource](#) tag, returning a link to the processed static resource. Grails' c `g:resource` tag delegates to this implementation if found, but if your code requires the Resources plu you should use `r:resource` directly.

Alongside the regular Grails [resource](#) tag attributes, this also supports the "uri" attribute for increased brev

See [r:resource documentation](#) for full details.

r:external

This is a resource-aware version of Grails [external](#) tag which renders the HTML markup necessary to inc an external file resource such as CSS, JS or a favicon.

See [r:resource documentation](#) for full details.

6.2.5.3 Declaring resources

A DSL is provided for declaring resources and modules. This can go either in your `Config.groovy` ir case of application-specific resources, or more commonly in a resources artefact in `grails-app/conf.`

Note that you do not need to declare all your static resources, especially images. However you mu: establish dependencies or other resources-specific attributes. Any resource that is not declared is ca "ad-hoc" and will still be processed using defaults for that resource type.

Consider this example resource configuration file, `grails-app/conf/MyAppResources.groovy`

```
modules = {
    core {
        dependsOn 'jquery, utils'

resource url: '/js/core.js', disposition: 'head'
        resource url: '/js/ui.js'
        resource url: '/css/main.css',
        resource url: '/css/branding.css'
        resource url: '/css/print.css', attrs: [media: 'print']
    }

utils {
        dependsOn 'jquery'

resource url: '/js/utils.js'
    }

forms {
        dependsOn 'core,utils'

resource url: '/css/forms.css'
        resource url: '/js/forms.js'
    }
}
```

This defines three resource modules; 'core', 'utils' and 'forms'. The resources in these modules wil
automatically bundled out of the box according to the module name, resulting in fewer files. You can ove
this with `bundle:'someOtherName'` on each resource, or call `defaultBundle` on the module (
resources plugin documentation).

It declares dependencies between them using `dependsOn`, which controls the load order of the resources.

When you include an `<r:require module="forms"/>` in your GSP, it will pull in all the resou
from 'core' and 'utils' as well as 'jquery', all in the correct order.

You'll also notice the `disposition:'head'` on the `core.js` file. This tells Resources that while it
defer all the other JS files to the end of the body, this one must go into the `<head>`.

The CSS file for print styling adds custom attributes using the `attrs` map option, and these are pa
through to the `r:external` tag when the engine renders the link to the resource, so you can customiz
HTML attributes of the generated link.

There is no limit to the number of modules or xxxResources.groovy artefacts you can provide, and plugins
supply them to expose modules to applications, which is exactly how the jQuery plugin works.

To define modules like this in your application's Config.groovy, you simply assign the DSL closure to
`grails.resources.modules` Config variable.

For full details of the resource DSL please see the resources plugin documentation.

6.2.5.4 Overriding plugin resources

Because a resource module can define the bundle groupings and other attributes of resources, you may
that the settings provided are not correct for your application.

For example, you may wish to bundle jQuery and some other libraries all together in one file. There
load-time and caching trade-off here, but often it is the case that you'd like to override some of these settin

To do this, the DSL supports an "overrides" clause, within which you can change the `defaultBun
setting for a module, or attributes of individual resources that have been declared with a unique id:

```
modules = {
    core {
        dependsOn 'jquery, utils'
        defaultBundle 'monolith'

resource url: '/js/core.js', disposition: 'head'
        resource url: '/js/ui.js'
        resource url: '/css/main.css',
        resource url: '/css/branding.css'
        resource url: '/css/print.css', attrs: [media: 'print']
    }
    utils {
        dependsOn 'jquery'
        defaultBundle 'monolith'

resource url: '/js/utils.js'
    }
    forms {
        dependsOn 'core,utils'
        defaultBundle 'monolith'

resource url: '/css/forms.css'
        resource url: '/js/forms.js'
    }
    overrides {
        jquery {
            defaultBundle 'monolith'
        }
    }
}
```

This will put all code into a single bundle named 'monolith'. Note that this can still result in multiple file
separate bundles are required for head and defer dispositions, and JavaScript and CSS files are bun
separately.

Note that overriding individual resources requires the original declaration to have included a unique id fo
resource.

For full details of the resource DSL please see the [resources plugin documentation](#).

6.2.5.5 Optimizing your resources

The Resources framework uses "mappers" to mutate the resources into the final format served to the user.

The resource mappers are applied to each static resource once, in a specific order. You can create your
resource mappers, and several plugins provide some already for zipping, caching and minifying.

Out of the box, the Resources plugin provides bundling of resources into fewer files, which is achieved w
few mappers that also perform CSS re-writing to handle when your CSS files are moved into a bundle.

Bundling multiple resources into fewer files

The 'bundle' mapper operates by default on any resource with a "bundle" defined - or inherited fron
`defaultBundle` clause on the module. Modules have an implicit default bundle name the same as
name of the module.

Files of the same kind will be aggregated into this bundle file. Bundles operate across module boundaries:

```
modules = {
    core {
        dependsOn 'jquery, utils'
        defaultBundle 'common'

resource url: '/js/core.js', disposition: 'head'
        resource url: '/js/ui.js', bundle: 'ui'
        resource url: '/css/main.css', bundle: 'theme'
        resource url: '/css/branding.css'
        resource url: '/css/print.css', attrs: [media: 'print']
    }

utils {
        dependsOn 'jquery'

resource url: '/js/utils.js', bundle: 'common'
    }

forms {
        dependsOn 'core,utils'

resource url: '/css/forms.css', bundle: 'ui'
        resource url: '/js/forms.js', bundle: 'ui'
    }
}
```

Here you see that resources are grouped into bundles; 'common', 'ui' and 'theme' - across module boundarie

Note that auto-bundling by module does **not** occur if there is only one resource in the module.

Making resources cache "eternally" in the client browser

Caching resources "eternally" in the client is only viable if the resource has a unique name that cha
whenever the contents change, and requires caching headers to be set on the response.

The [cached-resources](#) plugin provides a mapper that achieves this by hashing your files and renaming t
based on this hash. It also sets the caching headers on every response for those resources. To use, sim
install the cached-resources plugin.

Note that the caching headers can only be set if your resources are being served by your application. If
have another server serving the static content from your app (e.g. Apache HTTPD), configure it to s
caching headers. Alternatively you can configure it to request and proxy the resources from your container

Zipping resources

Returning gzipped resources is another way to reduce page load times and reduce bandwidth.

The [zipped-resources](#) plugin provides a mapper that automatically compresses your content, excludin
default already compressed formats such as gif, jpeg and png.

Simply install the zipped-resources plugin and it works.

Minifying

There are a number of CSS and JavaScript minifiers available to obfuscate and reduce the size of your c
At the time of writing none are publicly released but releases are imminent.

6.2.5.6 Debugging

When your resources are being moved around, renamed and otherwise mutated, it can be hard to de client-side issues. Modern browsers, especially Safari, Chrome and Firefox have excellent tools that let view all the resources requested by a page, including the headers and other information about them.

There are several debugging features built in to the Resources framework.

X-Grails-Resources-Original-Src Header

Every resource served in development mode will have the X-Grails-Resources-Original-Src: header ad indicating the original source file(s) that make up the response.

Adding the debug flag

If you add a query parameter **_debugResources=y** to your URL and request the page, Resources will by any processing so that you can see your original source files.

This also adds a unique timestamp to all your resource URLs, to defeat any caching that browsers may This means that you should always see your very latest code when you reload the page.

Turning on debug all the time

You can turn on the aforementioned debug mechanism without requiring a query parameter, but turning in Config.groovy:

```
grails.resources.debug = true
```

You can of course set this per-environment.

6.2.5.7 Preventing processing of resources

Sometimes you do not want a resource to be processed in a particular way, or even at all. Occasionally may also want to disable all resource mapping.

Preventing the application of a specific mapper to an individual resource

All resource declarations support a convention of noXXXX:true where XXXX is a mapper name.

So for example to prevent the "hashandcache" mapper from being applied to a resource (which renames moves it, potentially breaking relative links written in JavaScript code), you would do this:

```
modules = {
    forms {
        resource url: '/css/forms.css', nohashandcache: true
        resource url: '/js/forms.js', nohashandcache: true
    }
}
```

Excluding/including paths and file types from specific mappers

Mappers have includes/excludes Ant patterns to control whether they apply to a given resource. Mapper
sensible defaults for these based on their activity, for example the zipped-resources plugin's "zip" mapp
set to exclude images by default.

You can configure this in your `Config.groovy` using the mapper name e.g:

```
// We wouldn't link to .exe files using Resources but for the sake of example:
grails.resources.zip.excludes = ['**/*.zip', '**/*.exe']

// Perhaps for some reason we want to prevent bundling on "less" CSS files:
grails.resources.bundle.excludes = ['**/*.less']
```

There is also an "includes" inverse. Note that settings these replaces the default includes/excludes for
mapper - it is not additive.

Controlling what is treated as an "ad-hoc" (legacy) resource

Ad-hoc resources are those undeclared, but linked to directly in your application **without** using the Grai
Resources linking tags (resource, img or external).

These may occur with some legacy plugins or code with hardcoded paths in.

There is a Config.groovy setting **grails.resources.adhoc.patterns** which defines a list of Servlet
compliant filter URI mappings, which the Resources filter will use to detect such "ad-hoc resource" reques

By default this is set to:

```
grails.resources.adhoc.patterns = ['images/*', '*.js', '*.css']
```

6.2.5.8 Other Resources-aware plugins

At the time of writing, the following plugins include support for the Resources framework:

- [jquery](#)
- [jquery-ui](#)
- [blueprint](#)
- [lesscss-resources](#)
- [zipped-resources](#)
- [cached-resources](#)

6.2.6 Sitemesh Content Blocks

Although it is useful to decorate an entire page sometimes you may find the need to decorate indepen
sections of your site. To do this you can use content blocks. To get started, partition the page to be decor
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>
```

6.2.7 Making Changes to a Deployed Application

One of the main issues with deploying a Grails application (or typically any servlet-based one) is that
change to the views requires that you redeploy your whole application. If all you want to do is fix a typo
page, or change an image link, it can seem like a lot of unnecessary work. For such simple requireme
Grails does have a solution: the grails.gsp.view.dir configuration setting.

How does this work? The first step is to decide where the GSP files should go. Let's say we want to keep t
unpacked in a /var/www/grails/my-app directory. We add these two lines
grails-app/conf/Config.groovy:

```
grails.gsp.enable.reload = true
grails.gsp.view.dir = "/var/www/grails/my-app/"
```

The first line tells Grails that modified GSP files should be reloaded at runtime. If you don't have this set you can make as many changes as you like but they won't be reflected in the running application until restart. The second line tells Grails where to load the views and layouts from.

> ⚠ The trailing slash on the `grails.gsp.view.dir` value is important! Without it, Grails will look for views in the parent directory.

Setting "grails.gsp.view.dir" is optional. If it's not specified, you can update files directly to the applica server's deployed war directory. Depending on the application server, these files might get overwritten v the server is restarted. Most application servers support "exploded war deployment" which is recommende this case.

With those settings in place, all you need to do is copy the views from your web application to the exte directory. On a Unix-like system, this would look something like this:

```
mkdir -p /var/www/grails/my-app/grails-app/views
cp -R grails-app/views/* /var/www/grails/my-app/grails-app/views
```

The key point here is that you must retain the view directory structure, including the grails-app/vi bit. So you end up with the path `/var/www/grails/my-app/grails-app/views/....`

One thing to bear in mind with this technique is that every time you modify a GSP, it uses up permgen sp So at some point you will eventually hit "out of permgen space" errors unless you restart the server. So technique is not recommended for frequent or large changes to the views.

There are also some System properties to control GSP reloading:

Name	Description	Def
grails.gsp.enable.reload	altervative system property for enabling the GSP reload mode without changing Config.groovy	
grails.gsp.reload.interval	interval between checking the lastmodified time of the gsp source file, unit is milliseconds	5000
grails.gsp.reload.granularity	the number of milliseconds leeway to give before deciding a file is out of date. this is needed because different roundings usually cause a 1000ms difference in lastmodified times	100

GSP reloading is supported for precompiled GSPs since Grails 1.3.5 .

6.2.8 GSP Debugging

Viewing the generated source code

- Adding "?showSource=true" or "&showSource=true" to the url shows the generated Groovy source for the view instead of rendering it. It won't show the source code of included templates. This only w in development mode

- The saving of all generated source code can be activated by setting the prop "grails.views.gsp.keepgenerateddir" (in Config.groovy) . It must point to a directory that exists a writable.

- During "grails war" gsp pre-compilation, the generated source code is stored grails.project.work.dir/gspcompile (usually in ~/.grails/(grails_version)/projects/(pr name)/gspcompile).

Debugging GSP code with a debugger

- See [Debugging GSP in STS](#)

Viewing information about templates used to render a single url

GSP templates are reused in large web applications by using the `g:render` taglib. Several small temp can be used to render a single page. It might be hard to find out what GSP template actually renders the seen in the result. The debug templates -feature adds html comments to the output. The comments cor debug information about gsp templates used to render the page.

Usage is simple: append "?debugTemplates" or "&debugTemplates" to the url and view the source of result in your browser. "debugTemplates" is restricted to development mode. It won't work in production.

Here is an example of comments added by debugTemplates :

```
<!-- GSP #2 START template: /home/.../views/_carousel.gsp
     precompiled: false lastmodified: … -->
.
.
.
<!-- GSP #2 END template: /home/.../views/_carousel.gsp
     rendering time: 115 ms -->
```

Each comment block has a unique id so that you can find the start & end of each template call.

6.3 Tag Libraries

Like [Java Server Pages](#) (JSP), GSP supports the concept of custom tag libraries. Unlike JSP, Grails' tag lib mechanism is simple, elegant and completely reloadable at runtime.

Quite simply, to create a tag library create a Groovy class that ends with the convention `TagLib` and pla within the `grails-app/taglib` directory:

```
class SimpleTagLib {
}
```

Now to create a tag create a Closure property that takes two arguments: the tag attributes and the content:

```
class SimpleTagLib {
    def simple = { attrs, body ->

    }
}
```

The `attrs` argument is a Map of the attributes of the tag, whilst the `body` argument is a Closure that ret
the body content when invoked:

```
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
to append content to the response. Then you can reference the tag inside your GSP; no imports are necessa

```
<g:emoticon happy="true">Hi John</g:emoticon>
```

⚠ To help IDEs like SpringSource Tool Suite (STS) and others autocomplete tag attributes, you should add Javadoc comments to your tag closures with `@attr` descriptions. Since taglibs use Groovy code it can be difficult to reliably detect all usable attributes.

For example:

```groovy
class SimpleTagLib {

/**
    * Renders the body with an emoticon.
    *
    * @attr happy whether to show a happy emoticon ('true') or
    * a sad emoticon ('false')
    */
    def emoticon = { attrs, body ->
        out << body() << (attrs.happy == 'true' ? " :-)" : " :-(")
    }
}
```

and any mandatory attributes should include the REQUIRED keyword, e.g.

```groovy
class SimpleTagLib {

/**
    * Creates a new password field.
    *
    * @attr name REQUIRED the field name
    * @attr value the field value
    */
    def passwordField = { attrs ->
        attrs.type = "password"
        attrs.tagName = "passwordField"
        fieldImpl(out, attrs)
    }
}
```

6.3.1 Variables and Scopes

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 easy to write simple tags that have no body and just ou 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 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 b 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 reuse the [re](#) tag:

238

```
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 l met. An example of this may be a set of security tags:

```
def isAdmin = { attrs, body ->
    def user = attrs.user
    if (user && 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 co set of access privileges:

```
<g:isAdmin user="${myUser}">
    // some restricted content
</g:isAdmin>
```

6.3.4 Iterative Tags

Iterative tags are easy 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 Groov times method to iterate the specified number of times:

```
<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 bec 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 to conflicts, so you should should instead name the variables that the body uses:

```
def repeat = { attrs, body ->
    def 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 invoca 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 refer 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 pa 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 f GSP pages like this:

```
<my:example name="..." />
```

where the prefix is the same as the value of the static `namespace` property. Namespaces are particu[...]
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[...]
do so simply declare the JSP to use with 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"/>
```

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 cha[...]
improves performance by reducing object creation and optimizing buffering during request processing[...]
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[...]
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 `/controller/action/id`. However, this convention is not hard wired into Grails and is in 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 bloc 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 parameter 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`. Omi 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. To rewrite one URI onto another exp URI (rather than a controller/action pair) do something like this:

```
"/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 simple URLs with concrete "tokens". In URL mapping speak tokens are the sequence of characters between each slash, '/'. A concrete token is one which is well def such as as /product. However, in many circumstances you don't know what the value of a particular to 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 to into a parameter (available via the [params](#) object) called id. For example given the UI /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 U format could be mapped as follows:

```
static mappings = {
   "/$blog/$year/$month/$day/$id"(controller: "blog", action: "show")
}
```

The above mapping would let you 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 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 G 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 controll 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 optional token. In a further example this technique could be applied to the blog URL mapping to have flexible linking:

```
static mappings = {
    "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

With this mapping all of these URLs would match with only the relevant parameters being populated in 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 just setting them in 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 vari 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 ca
used in combination with all sorts of logic.

6.4.3 Mapping to Views

You can resolve a URL to a view without a controller or action involved. For example to map the root UF
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 lets you map HTTP response codes to controllers, actions or views. Just use a method name
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 you can specify 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: IllegalArgumentException)
    "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 illegalArgum
action in ErrorsController, a NullPointerException will be handled by the nullPoin
action, and a MyException will be handled by the customException action. Other exceptions wi
handled by the catch-all rule and use the /errors/serverError view.

You can access the exception from your custom error handing view or controller action using the reque
exception attribute like so:

```
class ErrorController {
    def handleError() {
        def exception = request.exception
        // perform desired processing to handle the exception
    }
}
```

> ⊖ If your error-handling controller action throws an exception as well, you'll end up with a
>   StackOverflowException.

6.4.5 Mapping to HTTP methods

URL mappings can also be configured to map based on the HTTP method (GET, POST, PUT or DELE'
This is very 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 t
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 follow
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
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. I
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 par
object:

```
def name = params.name
println name // prints "logo" or "other/logo"
```

If you use wildcard URL mappings then you may want to exclude certain URIs from Grails' URL map
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

247

Another great feature of URL mappings is that they automatically customize the behaviour of the [link](#) ta
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
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 - October 2007 Posts
</g:link>
```

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 [validation constraints](#) mechanism, which lets you fur
"constrain" how a URL is matched. For example, if we revisit the blog sample code from earlier, the map
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 Mapp
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 pa
thus relieving you of that burden later on.

6.4.9 Named URL Mappings

URL Mappings also support named mappings, that is mappings which have a name associated with them.
name may be used to refer to a specific mapping when links are generated.

The syntax for defining a named mapping is as follows:

```
static mappings = {
    name <mapping name>: <url pattern> {
        // …
    }
}
```

For 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>
```

To specify attributes that should be applied to the generated href, specify a Map value to the at
attribute. These attributes will be applied directly to the href, not passed through to be used as req
parameters.

```
<link:accountDetails attrs="[class: 'fancy']" acctNumber="8675309">
    Show Account
</link:accountDetails>
```

That would result in:

```
<a href="/details/8675309" class="fancy">Show Account</a>
```

6.4.10 Customizing URL Formats

The default URL Mapping mechanism supports camel case names in the URLs. The default URL accessing an action named `addNumbers` in a controller named `MathHelperController` would something like `/mathHelper/addNumbers`. Grails allows for the customization of this pattern provides an implementation which replaces the camel case convention with a hyphenated convention would support URLs like `/math-helper/add-numbers`. To enable hyphenated URLs assign a value "hyphenated" to the `grails.web.url.converter` property in `grails-app/conf/Config.groovy`.

```
// grails-app/conf/Config.groovy
grails.web.url.converter = 'hyphenated'
```

Arbitrary strategies may be plugged in by providing a class which implements the [UrlConverter](#) interface adding an instance of that class to the Spring application context with the bean name `grails.web.UrlConverter.BEAN_NAME`. If Grails finds a bean in the context with that name, it be used as the default converter and there is no need to assign a value to th `grails.web.url.converter` config property.

```
// src/groovy/com/myapplication/MyUrlConverterImpl.groovy
package com.myapplication
class MyUrlConverterImpl implements grails.web.UrlConverter {

String toUrlElement(String propertyOrClassName) {
        // return some representation of a property or class name that should be
used in URLs…
    }
}
```

```
// grails-app/conf/spring/resources.groovy
beans = {
    "${grails.web.UrlConverter.BEAN_NAME}"(com.myapplication.MyUrlConverterImpl)
}
```

6.5 Web Flow

Overview

Grails supports the creation of web flows built on the [Spring Web Flow](#) project. A web flow is a conversa that spans multiple requests and retains state for the scope of the flow. A web flow also has a defined start end state.

Web flows don't require an HTTP session, but instead store their state in a serialized form, which is restored using a flow execution key that Grails passes around as a request parameter. This makes flow more scalable than other forms of stateful application that use the HttpSession and its inherit memory clustering concerns.

Web flow is essentially an advanced state machine that manages the "flow" of execution from one state to next. Since the state is managed for you, you don't have to be concerned with ensuring that users ente action in the middle of some multi step flow, as web flow manages that for you. This makes web flow per for use cases such as shopping carts, hotel booking and any application that has multi page work flows.

> ⚠ From Grails 1.2 onwards Webflow is no longer in Grails core, so you must install the Webflow plugin to use this feature: `grails install-plugin webflow`

Creating a Flow

To create a flow create a regular Grails controller and add an action that ends with the convention `Flow`. 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 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 wh user first initiates a conversation (or flow). The start state of a Grails flow is the first method call that tak 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 actio redirect it is assumed be a [view state](#) that, by convention, refers to the vie `grails-app/views/book/shoppingCart/showCart.gsp`.

Notice that unlike regular controller actions, the views are stored within a directory that matches the nam the flow: `grails-app/views/book/shoppingCart`.

The `shoppingCart` flow also has two possible end states. The first is `displayCatalogue` w[]
performs an external redirect to another controller and action, thus exiting the flow. The second
`displayInvoice` which is an end state as it has no events at all and will simply render a view ca[]
`grails-app/views/book/shoppingCart/displayInvoice.gsp` whilst ending the flow a[]
same time.

Once a flow has ended it can only be resumed from the start state, in this case `showCart`, and not from []
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 this 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 []
the `enterPersonalDetails` state defines two events: `submit` and `return`. The view is responsibl[]
[triggering] these events. Use the `render` method to change the view to be rendered:

```
enterPersonalDetails {
    render(view: "enterDetailsView")
    on("submit").to "enterShipping"
    on("return").to "showCart"
}
```

Now it will look for `grails-app/views/book/shoppingCart/enterDetailsView.gsp`. []
the `view` parameter with a `/` to use a shared view:

```
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 a view. The result of the action is use[]
dictate flow transition. To create an action state you define an action to to be executed. This is done by ca[]
the `action` method and passing it a block of code to be executed:

253

```
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 reuse controller actio
you want. If the action successfully returns with no errors the success event will be triggered. In this
since we return a Map, which 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"
```

This makes 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)
        for (item in cartItems) { o.addToItems item }
        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 create
Order object. It then returns the order as the model. The important thing to note here is the interaction
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 pri
state transition once an [event](#) has been triggered. A simple example of a transition action can be seen belo

```
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 very 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* 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 `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 compon 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 name="checkout" value="Checkout" />
</g:form>
```

The form must submit back to the `shoppingCart` flow. The name attribute of each [submitButton]() signals which event will be triggered. If you don't have a form you can also trigger an event with the [link]() as follows:

```
<g:link action="shoppingCart" event="checkout" />
```

Triggering Events from an Action

To trigger an event from an `action` you invoke a method. For example there is the built in `error()` `success()` methods. The example below triggers the `error()` event on validation failure in a trans: action:

255

```
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 t 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 "scope". Grails flows have five 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 ne subflows

- `session` - Stores objects in the user's session

> ⚠ Grails service classes can be automatically scoped to a web flow scope. See the documentation on <u>Services</u> for more information.

Returning a model Map from an action will automatically result in the model being placed in flow scope. 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 ac
state (for example) will not be available in a subsequent view state. Use one of the other scopes to pass ob
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
   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 impleme
`java.io.Serializable` or an exception will be thrown. This has an impact on domain classes in
domain classes are typically placed within a scope so that they can be rendered in a view. For exam
consider the following domain class:

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

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 clos
used in GORM events such as `onLoad`, `onSave` and so on. The following domain class will cause an err
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 should declare all events as transient:

```
class Book implements Serializable {

String title

transient onLoad = {
        println "I'm loading"
    }
}
```

or as methods:

```
class Book implements Serializable {

String title

def onLoad() {
        println "I'm loading"
    }
}
```

> ⚠ The flow scope contains a reference to the Hibernate session. As a result, any object loaded into the session through a GORM query will also be in the flow and will need to implement Serializable.
>
> If you don't want your domain class to be Serializable or stored in the flow, then you will need to evict the entity manually before the end of the state:
>
> ```
> flow.persistenceContext.evict(it)
> ```

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 enterPersonalDetails state. This state renders a view and waits for the user to enter the requ 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 capture the form info we 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 f scope. Also interesting is that we perform [validation](#) and invoke the error() method if validation fails. signals to the flow that the transition should halt and return to the enterPersonalDetails view so entries can be entered by the user, otherwise the transition should continue and go to the enterShipp state.

Like regular actions, flow actions also support the notion of [Command Objects](#) by defining the first argur 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 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 {
        // Extended search subflow
        subflow(controller: "searchExtensions", action: "extendedSearch")
        on("moreResults").to "displayMoreResults"
        on("noResults").to "displayNoMoreResults"
    }
    displayMoreResults()
    displayNoMoreResults()
}
```

It references a subflow in the `extendedSearch` state. The controller parameter is optional if the subflo
defined in the same controller as the calling flow.

> ⚠ Prior to 1.3.5, the previous subflow call would look like
> `subflow(extendedSearchFlow)`, with the requirement that the name of the
> subflow state be the same as the called subflow (minus `Flow`). This way of calling a
> subflow is deprecated and only supported for backward compatibility.

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
lets you share state that spans the whole conversation not just the flow. Also notice that the end state (ei
`moreResults` or `noResults` of the subflow triggers the events in the main flow:

```
extendedSearch {
    // Extended search subflow
    subflow(controller: "searchExtensions", action: "extendedSearch")
    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
few controllers and become difficult to manage with larger applications. Filters on the other hand ca
applied across a whole group of controllers, a URI space or to a specific action. Filters are far easier to pl
and maintain completely separately to your main controller logic and are useful for all sorts of cross cu
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/c`
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
the scope is defined using named arguments. For example to define a filter that applies to all controllers
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

- `controllerExclude` - controller exclusion pattern, by default * is replaced with .* and a rege compiled

- `action` - action matching pattern, by default * is replaced with .* and a regex is compiled

- `actionExclude` - action exclusion 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/**)

- `uriExclude` - a uri pattern to exclude, 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`

```
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) {

}
```

- All actions starting with the letter 'b' except for actions beginning with the phrase 'bad*'

```
    actionBeginningWithBBbutNotBad(action: 'b*', actionExclude: 'bad*', 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 ord
which they are executed. To control the order of execution between `Filters` classes, you can use
`dependsOn` property discussed in <u>filter dependencies</u> section.

> ⚠ Note: When exclude patterns are used they take precedence over the matching patterns.
> For example, if action is 'b*' and actionExclude is 'bad*' then actions like 'best' and
> 'bien' will have that filter applied but actions like 'bad' and 'badlands' will not.

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 fi

- `before` - Executed before the action. Return `false` to indicate that the response has been handled
  that all future filters and the action should not execute

- `after` - Executed after an action. Takes a first argument as the view model to allow modification of
  model before rendering the view

- `afterView` - Executed after view rendering. Takes an Exception as an argument which will be
  `null` if an exception occurs during processing. Note: this Closure is called before the layout is applie

For example to fulfill the common simplistic authentication use case you could define a filter as follows:

263

```
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 the session and if not redirects to the login action. Note how returning false ensure that the action itself is executed.

Here's a more involved example that demonstrates all three filter types:

```
import java.util.concurrent.atomic.AtomicLong

class LoggingFilters {

private static final AtomicLong REQUEST_NUMBER_COUNTER = new AtomicLong()
   private static final String START_TIME_ATTRIBUTE = 'Controller__START_TIME__'
   private static final String REQUEST_NUMBER_ATTRIBUTE =
'Controller__REQUEST_NUMBER__'

def filters = {

logFilter(controller: '*', action: '*') {

before = {
            if (!log.debugEnabled) return true

long start = System.currentTimeMillis()
            long currentRequestNumber = REQUEST_NUMBER_COUNTER.incrementAndGet()

request[START_TIME_ATTRIBUTE] = start
            request[REQUEST_NUMBER_ATTRIBUTE] = currentRequestNumber

log.debug "preHandle request #$currentRequestNumber : " +
                "'$request.servletPath'/'$request.forwardURI', " +
                "from $request.remoteHost ($request.remoteAddr) " +
                " at ${new Date()}, Ajax: $request.xhr, controller:
$controllerName, " +
                "action: $actionName, params: ${new TreeMap(params)}"

return true
          }

after = { Map model ->

if (!log.debugEnabled) return true

long start = request[START_TIME_ATTRIBUTE]
            long end = System.currentTimeMillis()
            long requestNumber = request[REQUEST_NUMBER_ATTRIBUTE]

def msg = "postHandle request #$requestNumber: end ${new Date()}, " +
                    "controller total time ${end - start}ms"
            if (log.traceEnabled) {
                log.trace msg + "; model: $model"
            }
            else {
                log.debug msg
            }
          }

afterView = { Exception e ->

if (!log.debugEnabled) return true

long start = request[START_TIME_ATTRIBUTE]
            long end = System.currentTimeMillis()
            long requestNumber = request[REQUEST_NUMBER_ATTRIBUTE]

def msg = "afterCompletion request #$requestNumber: " +
                    "end ${new Date()}, total time ${end - start}ms"
            if (e) {
                log.debug "$msg \n\texception: $e.message", e
            }
            else {
                log.debug msg
            }
          }
      }
    }
}
```

In this logging example we just log various request information, but note that the model map in the af
filter is mutable. If you need to add or remove items from the model map you can do that in the after fil

6.6.3 Variables and Scopes

Filters support all the common properties available to [controllers](#) and [tag libraries](#), plus the applica
context:

- [request](#) - The HttpServletRequest object
- [response](#) - The HttpServletResponse object
- [session](#) - The HttpSession object
- [servletContext](#) - The ServletContext object
- [flash](#) - The flash object
- [params](#) - The request parameters object
- [actionName](#) - The action name that is being dispatched to
- [controllerName](#) - The controller name that is being dispatched to
- [grailsApplication](#) - The Grails application currently running
- [applicationContext](#) - The ApplicationContext object

However, filters only support a subset of the methods available to controllers and tag libraries. These inclu

- [redirect](#) - For redirects to other controllers and actions
- [render](#) - 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
`dependsOn` property. This is used when a `Filters` class depends on the behavior of another `Filt`
class (e.g. setting up the environment, modifying the request/session, etc.) and is defined as an array
`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 w
scope matches the current request to be executed before those in MyFilters. For a request of "/test", which
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 MyFi
class. Execution order between `Filters` classes are enabled and the execution order of filters within e
`Filters` class are preserved.

If any cyclical dependencies are detected, the filters with cyclical dependencies will be added to the end o
filter chain and processing will continue. Information about any cyclical dependencies that are detected wi
written to the logs. Ensure that your root logging level is set to at least WARN or configure an appende
the   Grails   Filters   Plugin
org.codehaus.groovy.grails.plugins.web.filters.FiltersGrailsPlugin)   v
debugging filter dependency issues.

6.7 Ajax

Ajax is the driving force behind the shift to richer web applications. These types of applications in genera
better suited to agile, dynamic frameworks written in languages like [Groovy](#) and [Ruby](#) Grails provides sup
for building Ajax applications through its Ajax tag library. For a full list of these see the Tag Lib
Reference.

6.7.1 Ajax Support

By default Grails ships with the [jQuery](#) library, but through the [Plugin system](#) provides support for c
frameworks such as [Prototype](#), , and the [Google Web Toolkit](#).

This section covers Grails' support for Ajax in general. To get started, add this line to the `<head>` tag of
page:

```
<g:javascript library="jquery" />
```

You can replace `jQuery` with any other library supplied by a plugin you have installed. This works bec
of Grails' support for adaptive tag libraries. Thanks to Grails' plugin system there is support for a number
different Ajax libraries including (but not limited to):

- jQuery

- 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]
This tag allows the creation of HTML anchor tags that perform an asynchronous request and optionally se
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

6.7.1.2 Updating Content

This is great, but usually you provide feedback to the user about what 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
`"Book 1 was deleted"`. This is done by the `update` attribute on the tag, which can also take a Ma
indicate what should be updated on failure:

```
<div id="message"></div>
<div id="error"></div>
<g:remoteLink update="[success: 'message', failure: 'error']"
              action="delete" id="1">
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](#)
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 button
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 let
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 whatev
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
  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 respo

- `onComplete` - The JavaScript function to call when the remote function is complete, including
  updates

If you need a reference to the `XmlHttpRequest` object you can use the implicit event parameter e to ob
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 Prototype

Grails features an external plugin to add [Prototype](#) support to Grails. To install the plugin type the follow
command from the root of your project in a terminal window:

```
grails install-plugin prototype
```

This will download the current supported version of the Prototype plugin and install it into your Grails pro
With that done you can add the following reference to the top of your page:

```
<g:javascript library="prototype" />
```

If you require [Scriptaculous](#) too you can do the following instead:

```
<g:javascript library="scriptaculous" />
```

Now all of Grails tags such as [remoteLink](#), [formRemote](#) and [submitToRemote](#) work with Prototype remoti

6.7.3 Ajax with Dojo

Grails features an external plugin to add [Dojo](#) support to Grails. To install the plugin type the follow
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 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.

6.7.4 Ajax with GWT

Grails also features support for the [Google Web Toolkit](#) through a plugin. There is comprehens [documentation](#) available on the Grails wiki.

6.7.5 Ajax on the Server

There are a number of different ways to implement Ajax which are typically broken down into:

- Content Centric Ajax - Where you just 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 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 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 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. a JSON response with Grails you would typically use Grails' [JSON marshalling](#) capability:

```
import grails.converters.JSON

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 simple:

```
import grails.converters.XML

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
this can be seen below:

```
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 use Proto
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
client breaking the server. This is one of the reasons Rails has something like RJS. Although Grails does
currently have a feature such as RJS there is a **Dynamic JavaScript Plugin** that offers similar capabilities.

Responding to both Ajax and non-Ajax requests

It's straightforward to have the same Grails controller action handle both Ajax and non-Ajax requests. G
adds the isXhr() method to HttpServletRequest which can be used to identify Ajax requests.
example you could render a page fragment using a template for Ajax requests or the full page for reg
HTTP requests:

```
def listBooks() {
    def books = Book.list(params)
    if (request.xhr) {
        render template: "bookTable", model: [books: books]
    } else {
        render view: "list", model: [books: books]
    }
}
```

6.8 Content Negotiation

Grails has built in support for **Content negotiation** using either the HTTP Accept header, an explicit fo
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 wis
support. By default Grails comes configured with a number of different content types with
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',
                      csv: '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/xm 'application/xml' media types as simply 'xml'. You can add your own types by simply adding new entries the map.

Content Negotiation using the Accept header

Every incoming HTTP request has a special [Accept](#) header that defines what media types (or mime type client can "accept". In older browsers this is typically:

```
*/*
```

Which simply means anything. However, on newer browser something all together more useful is sent suc (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, */*;q=0.5
```

Grails parses this incoming format and adds a `property` to the [response](#) object that outlines the prefe response format. For the above example the following assertion would pass:

```
assert 'html' == response.format
```

Why? The `text/html` media type has the highest "quality" rating of 0.9, therefore is the highest priorit you have an older browser as mentioned previously the result is slightly different:

```
assert 'all' == response.format
```

In this case 'all' possible formats are accepted by the client. To deal with different kinds of requests fr [Controllers](#) you can use the [withFormat](#) method that acts as kind of a switch statement:

```
import grails.converters.XML

class BookController {

def list() {
        def books = Book.list()
        withFormat {
            html bookList: books
            js { render "alert('hello')" }
            xml { render books as XML }
        }
    }
}
```

If the preferred format is `html` then Grails will execute the `html()` call only. This causes Grails to look 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 an XML response rendered.

How do we handle the "all" format? Simply order the content-types within your `withFormat` block so whichever one you want executed comes first. So in the above example, "all" will trigger the `html` handle

> ⚠ 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.

Request format vs. Response format

As of Grails 2.0, there is a separate notion of the *request* format and the *response* format. The request fo is dictated by the `CONTENT_TYPE` header and is typically used to detect if the incoming request ca parsed into XML or JSON, whilst the response format uses the file extension, format parameter or ACC header to attempt to deliver an appropriate response to the client.

The **withFormat** available on controllers deals specifically with the response format. If you wish to add l that deals with the request format then you can do so using a separate `withFormat` method available or request:

```
request.withFormat {
    xml {
        // read XML
    }
    json {
        // read JSON
    }
}
```

Content Negotiation with the format Request Parameter

If fiddling with request headers if not your favorite activity you can override the format used by specifyin `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 using URI extensions. For example given the following URI:

```
/book/list.xml
```

Grails will remove the extension and map it to /book/list instead whilst simultaneously setting content format to xml based on this extension. This behaviour is enabled by default, so if you wish to tu 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 a unit or integration test (see the section on [Testing](#)) you can either manipu 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
}
```

276

7 Validation

Grails validation capability is built on [Spring's Validator API](#) and data binding capabilities. However G takes this further and provides a unified way to define validation "constraints" with its constraints mechani

Constraints in Grails are a way to declaratively specify validation rules. Most commonly they are applied [domain classes](#), however [URL Mappings](#) and [Command Objects](#) also support constraints.

7.1 Declaring Constraints

Within a domain class [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 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
    }
}
```

In this example we've declared that the `login` property must be between 5 and 15 characters long, it ca be blank and must be unique. We've also applied other constraints to the `password`, `email` and properties.

> ⚠ By default, all domain class properties are not nullable (i.e. they have an implicit `nullable: false` constraint). The same is not true for command object properties, which are nullable by default.

A complete reference for the available constraints can be found in the Quick Reference section under Constraints heading.

Note that constraints are only evaluated once which may be relevant for a constraint that relies on a value an instance of `java.util.Date`.

```
class User {
    ...

static constraints = {
        // this Date object is created when the constraints are evaluated, not
        // each time an instance of the User class is validated.
        birthDate max: new Date()
    }
}
```

A word of warning - referencing domain class properties from constraints

It's very easy to attempt to reference instance variables from the static constraints block, but this isn't leg
Groovy (or Java). If you do so, you will get a MissingPropertyException for your trouble.
example, you may try

```
class Response {
    Survey survey
    Answer answer

static constraints = {
        survey blank: false
        answer blank: false, inList: survey.answers
    }
}
```

See how the inList constraint references the instance property survey? That won't work. Instead, u
custom [validator](#):

```
class Response {
    …
    static constraints = {
        survey blank: false
        answer blank: false, validator: { val, obj -> val in obj.survey.answers }
    }
}
```

In this example, the obj argument to the custom validator is the domain *instance* that is being validated
we can access its survey property and return a boolean to indicate whether the new value for the ans
property, val, is valid.

7.2 Validating Constraints

Validation Basics

Call the [validate](#) method to validate a domain class 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` inter provides methods to navigate the validation errors and also retrieve the original values.

Validation Phases

Within Grails there are two phases of validation, the first one being [data binding](#) which occurs when you 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 conver 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 bound values agaits the [constraints](#) you defined. For example, by default the [save](#) method calls `valid` before executing, 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 redirect back to the view for rendering. Once there you need s way of displaying errors. Grails supports a rich set of tags for dealing with errors. To render the errors as a you can use [renderErrors](#):

279

```
  <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. 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>
```

This code checks if the `login` field of the `user` bean has any errors and if so it adds an `errors` CSS to the `div`, 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 within it. This is useful as you can use the error object to restore the value input by the user using [fieldValue](#) tag:

```
<input type="text" name="login" value="${fieldValue(bean:user,field:'login')}"/>
```

This code will check for an existing `FieldError` in the `User` bean and if there is obtain the originally i value for the `login` field.

7.4 Validation and Internationalization

Another important thing to note about errors in Grails is that error messages are not hard coded anywhere. [FieldError](#) class in Spring 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 earl

```
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
    }
}
```

If a constraint is violated Grails will by convention look for a message code of 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 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 preceden So for example, com.mycompany.myapp.User.login.blank will be used before user.login.blank. This all for cases where your domain class message codes clash with a plugin's.

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 look up messages for you using the [message](#) tag. If you need n control of rendering you can handle 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 er 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 validate by defining the static `constraints` property in the class (as described above) and then telling framework about them. It is important that the application register the validateable classes with framework. Simply defining the `constraints` property is not sufficient.

281

The Validateable Annotation

Classes which define the static `constraints` property and are annotated with @Validateable can be n
validateable by the framework. Consider this example:

```
// src/groovy/com/mycompany/myapp/User.groovy
package com.mycompany.myapp

import 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
    }
}
```

Registering Validateable Classes

If a class is not marked with `Validateable`, it may still be made validateable by
framework. The steps required to do this are to define the static constra
property in the class (as described above) and then telling the framew
about the class by assigning a value to the grails.validateable.classes property
Config.groovy@:

```
grails.validateable.classes = [com.mycompany.myapp.User,
com.mycompany.dto.Account]
```

8 The Service Layer

Grails defines the notion of a service layer. The Grails team discourages the embedding of core applica
logic inside controllers, as it does not promote reuse and a clean separation of concerns.

Services in Grails are the place to put the majority of the logic in your application, leaving contro
responsible for handling request flow with 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
terminal window:

```
grails create-service helloworld.simple
```

> ⚠️ If no package is specified with the create-service script, Grails automatically uses the
> application name as the package name.

The above example will create a service at the location
`grails-app/services/helloworld/SimpleService.groovy`. A service's name ends with
convention `Service`, other than that a service is a plain Groovy class:

```groovy
package helloworld
class SimpleService {
}
```

8.1 Declarative Transactions

Default Declarative Transactions

Services are typically involved with coordinating logic between [domain classes](#), and hence often invo
with persistence that spans large operations. Given the nature of services, they frequently require transacti
behaviour. You can use programmatic transactions with the [withTransaction](#) method, however th
repetitive and doesn't fully leverage the power of Spring's underlying transaction abstraction.

Services enable transaction demarcation, which is a declarative way of defining which methods are to be n
transactional. All services are transactional by default. To disable this set the `transactional` propert
false:

```groovy
class CountryService {
    static transactional = false
}
```

You may also set this property to `true` 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 that all methods are wrapped in a transaction and automatic rollback occurs if a method throws runtime exception (i.e. one that extends `RuntimeException`) or an `Error`. The propagation level of transaction is by default set to [PROPAGATION_REQUIRED](#).

> ⛔ Checked exceptions do **not** roll back transactions. Even though Groovy blurs the distinction between checked and unchecked exceptions, Spring isn't aware of this and its default behaviour is used, so it's important to understand the distinction between checked and unchecked exceptions.

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.

> ⚠️ Annotating a service method with `Transactional` disables the default Grails transactional behavior for that service (in the same way that adding `transactional=false` does) so if you use any annotations you must annotate all methods that require transactions.

In this example `listBooks` uses a read-only transaction, `updateBook` uses a default read-write transaction, and `deleteBook` is not transactional (probably not a good idea given its name).

```
import org.springframework.transaction.annotation.Transactional

class BookService {

@Transactional(readOnly = true)
    def listBooks() {
        Book.list()
    }

@Transactional
    def updateBook() {
        // …
    }

def deleteBook() {
        // …
    }
}
```

You can also annotate the class to define the default transaction behavior for the whole service, and override that default per-method. For example, this service is equivalent to one that has no annotations (s the default is implicitly `transactional=true`):

```
import org.springframework.transaction.annotation.Transactional

@Transactional
class BookService {

def listBooks() {
      Book.list()
    }

def updateBook() {
      // …
    }

def deleteBook() {
      // …
    }
}
```

This version defaults to all methods being read-write transactional (due to the class-level annotation), but listBooks method overrides this to use a read-only transaction:

```
import org.springframework.transaction.annotation.Transactional

@Transactional
class BookService {

@Transactional(readOnly = true)
    def listBooks() {
        Book.list()
    }

def updateBook() {
      // …
    }

def deleteBook() {
      // …
    }
}
```

Although updateBook and deleteBook aren't annotated in this example, they inherit the configura from the class-level annotation.

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 annot as needed and Grails will detect them up automatically.

8.1.1 Transactions Rollback and the Session

Understanding Transactions and the Hibernate Session

When using transactions there are important considerations you must take into account with regards to the underlying persistence session is handled by Hibernate. When a transaction is rolled back the Hiber session used by GORM is cleared. This means any objects within the session become detached and acces uninitialized lazy-loaded collections will lead to LazyInitializationExceptions.

To understand why it is important that the Hibernate session is cleared. Consider the following example:

285

```
class Author {
    String name
    Integer age

static hasMany = [books: Book]
}
```

If you were to save two authors using consecutive transactions as follows:

```
Author.withTransaction { status ->
    new Author(name: "Stephen King", age: 40).save()
    status.setRollbackOnly()
}

Author.withTransaction { status ->
    new Author(name: "Stephen King", age: 40).save()
}
```

Only the second author would be saved since the first transaction rolls back the author save() by clea
the Hibernate session. If the Hibernate session were not cleared then both author instances would be persi
and it would lead to very unexpected results.

It can, however, be frustrating to get LazyInitializationExceptions due to the session b
cleared.

For example, consider the following example:

```
class AuthorService {

void updateAge(id, int age) {
        def author = Author.get(id)
        author.age = age
        if (author.isTooOld()) {
            throw new AuthorException("too old", author)
        }
    }
}
```

```
class AuthorController {

def authorService

def updateAge() {
        try {
            authorService.updateAge(params.id, params.int("age"))
        }
        catch(e) {
            render "Author books ${e.author.books}"
        }
    }
}
```

In the above example the transaction will be rolled back if the Author's age exceeds the maximum v
defined in the isTooOld() method by throwing an AuthorException. The AuthorExcept
references the author but when the books association is accessed a LazyInitializationExcept
will be thrown because the underlying Hibernate session has been cleared.

To solve this problem you have a number of options. One is to ensure you query eagerly to get the data
will need:

```
class AuthorService {
    …
    void updateAge(id, int age) {
        def author = Author.findById(id, [fetch:[books:"eager"]])
        ...
```

In this example the `books` association will be queried when retrieving the `Author`.

> ⚠ This is the optimal solution as it requires fewer queries then the following suggested
> solutions.

Another solution is to redirect the request after a transaction rollback:

```
class AuthorController {

AuthorService authorService

def updateAge() {
    try {
        authorService.updateAge(params.id, params.int("age"))
    }
    catch(e) {
        flash.message "Can't update age"
        redirect action:"show", id:params.id
    }
  }
}
```

In this case a new request will deal with retrieving the `Author` again. And, finally a third solution
retrieve the data for the `Author` again to make sure the session remains in the correct state:

```
class AuthorController {

def authorService

def updateAge() {
    try {
        authorService.updateAge(params.id, params.int("age"))
    }
    catch(e) {
        def author = Author.read(params.id)
        render "Author books ${author.books}"
    }
  }
}
```

Validation Errors and Rollback

A common use case is to rollback a transaction if there are validation errors. For example consider
service:

287

```
import grails.validation.ValidationException

class AuthorService {

void updateAge(id, int age) {
        def author = Author.get(id)
        author.age = age
        if (!author.validate()) {
            throw new ValidationException("Author is not valid", author.errors)
        }
    }
}
```

To re-render the same view that a transaction was rolled back in you can re-associate the errors wi
refreshed instance before rendering:

```
import grails.validation.ValidationException

class AuthorController {

def authorService

def updateAge() {
        try {
            authorService.updateAge(params.id, params.int("age"))
        }
        catch (ValidationException e) {
            def author = Author.read(params.id)
            author.errors = e.errors
            render view: "edit", model: [author:author]
        }
    }
}
```

8.2 Scoped Services

By default, access to service methods is not synchronised, so nothing prevents concurrent execution of t
methods. In fact, because the service is a singleton and may be used concurrently, you should be very ca
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 :
  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 must 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 example

```
static scope = "flow"
```

8.3 Dependency Injection and Services

Dependency Injection Basics

A key aspect of Grails services is the ability to use [Spring Framework](#)'s dependency injection features. G supports "dependency injection by convention". In other words, you can use the property name representa 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 define a property called `bookService` controller as follows:

```
class BookController {
    def bookService
    …
}
```

In this case, the Spring container will automatically inject an instance of that service based on its config scope. All dependency injection is done by name. You can also specify the type as follows:

```
class AuthorService {
    BookService bookService
}
```

> ⚠ NOTE: Normally the property name is generated by lower casing the first letter of the type. For example, an instance of the `BookService` class would map to a property named `bookService`.
>
> To be consistent with standard JavaBean conventions, if the first 2 letters of the class name are upper case, the property name is the same as the class name. For example, the property name of the `JDBCHelperService` class would be `JDBCHelperService`, not `jDBCHelperService` or `jdbcHelperService`.
>
> See section 8.8 of the JavaBean specification for more information on de-capitalization rules.

Dependency Injection and Services

You can inject services in other services with the same technique. If you had an AuthorService needed to use the BookService, declaring the AuthorService as follows would allow that:

```
class AuthorService {
    def bookService
}
```

Dependency Injection and Domain Classes / Tag Libraries

You can even inject services into domain classes and tag libraries, which can aid in the development of domain models and views:

```
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 t from other classes, including Java classes. There are a couple of ways you can reuse a service from Java. simplest way is to move your service into a package within the grails-app/services directory. reason this is important is that it is not possible to import classes into Java from the default package package used when no package declaration is present). So for example the BookService below cannc 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 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 implementation class (although it's always a good idea to use packages). Either way, the goal of this exer 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` directory and add a setter that use type and the name of the bean in Spring:

```
// src/java/bookstore/BookConsumer.java
package bookstore;

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 i `grails-app/conf/spring/resources.xml` (for more information see the section on Grails Spring):

```
<bean id="bookConsumer" class="bookstore.BookConsumer">
    <property name="bookStore" ref="bookService" />
</bean>
```

or in `grails-app/conf/spring/resources.groovy`:

```
import bookstore.BookConsumer

beans = {
    bookConsumer(BookConsumer) {
        bookStore = ref("bookService")
    }
}
```

291

9 Testing

Automated testing is a key part of Grails. 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 for testing.

> ⚠ Grails 1.3.x and below used the `grails.test.GrailsUnitTestCase` class hierarchy for testing in a JUnit 3 style. Grails 2.0.x and above deprecates these test harnesses in favour of mixins that can be applied to a range of different kinds of tests (JUnit 3, Junit 4, Spock etc.) without subclassing

The first thing to be aware of is that all of the `create-*` and `generate-*` commands create unit or integration tests automatically. For example if you run the [create-controller](#) command as follows:

```
grails create-controller com.acme.app.simple
```

Grails will create a controller at `grails-app/controllers/com/acme/app/SimpleController.groovy`, and also a unit test at `test/unit/com/acme/app/SimpleControllerTests.groovy`. What Grails won't do however populate the logic inside the test! That is left up to you.

> ⚠ The default class name suffix is `Tests` but as of Grails 1.2.2, the suffix of `Test` is also supported.

Running Tests

Tests are run with the [test-app](#) command:

```
grails test-app
```

The 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 showing the reason for each test failure.

> ⚠ You can force a clean before running tests by passing `-clean` to the `test-app` command.

Grails writes both plain text and HTML test reports to the `target/test-reports` directory, along the original XML files. The HTML reports are generally the best ones to look at.

Using Grails' [interactive mode](#) confers some distinct advantages when executing tests. First, the tests execute significantly faster on the second and subsequent runs. Second, a shortcut is available to open HTML reports in your browser:

```
open test-report
```

You can also run your unit tests from within most IDEs.

Targeting Tests

You can selectively target the test(s) to be run in different ways. To run all tests for a controller nam `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 patter
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:t`
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 ru
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
`unit`, `integration` and `functional` phases. To run all spock tests in all phases you would ru
following:

```
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 package `some.org` subpackage.

9.1 Unit Testing

Unit testing are tests at the "unit" level. In other words you are testing individual methods or blocks of without consideration for surrounding infrastructure. Unit tests are typically run without the presenc physical resources that involve I/O such databases, socket connections or files. This is to ensure they ru quick as possible since quick feedback is important.

Since Grails 2.0, a collection of unit testing mixins is provided by Grails that lets you enhance the behavio a typical JUnit 3, JUnit 4 or Spock test. The following sections cover the usage of these mixins.

> ⚠ The previous JUnit 3-style `GrailsUnitTestCase` class hierarchy is still present in
> Grails for backwards compatibility, but is now deprecated. The previous documentation
> on the subject can be found in the [Grails 1.3.x documentation](#)

You won't normally have to import any of the testing classes because Grails does that for you. But if you that your IDE for example can't find the classes, here they all are:

- `grails.test.mixin.TestFor`
- `grails.test.mixin.TestMixin`
- `grails.test.mixin.Mock`
- `grails.test.mixin.support.GrailsUnitTestMixin`
- `grails.test.mixin.domain.DomainClassUnitTestMixin`
- `grails.test.mixin.services.ServiceUnitTestMixin`
- `grails.test.mixin.web.ControllerUnitTestMixin`
- `grails.test.mixin.web.FiltersUnitTestMixin`
- `grails.test.mixin.web.GroovyPageUnitTestMixin`
- `grails.test.mixin.web.UrlMappingsUnitTestMixin`
- `grails.test.mixin.webflow/WebFlowUnitTestMixin`

Note that you're only ever likely to use the first two explicitly. The rest are there for reference.

9.1.1 Unit Testing Controllers

The Basics

You use the `grails.test.mixin.TestFor` annotation to unit test controllers. Using `TestFor` in manner activates the `grails.test.mixin.web.ControllerUnitTestMixin` and its associ API. For example:

```
import grails.test.mixin.TestFor

@TestFor(SimpleController)
class SimpleControllerTests {
    void testSomething() {

    }
}
```

Adding the `TestFor` annotation to a controller causes a new `controller` field to be automatically cre for the controller under test.

> ⚠ The `TestFor` annotation will also automatically annotate any public methods starting with "test" with JUnit 4's @Test annotation. If any of your test method don't start with "test" just add this manually

To test the simplest "Hello World"-style example you can do the following:

```
// Test class
class SimpleController {
    def hello() {
        render "hello"
    }
}
```

```
void testHello() {
    controller.hello()

assert response.text == 'hello'
}
```

The `response` object is an instance of `GrailsMockHttpServletResponse` (from the packa `org.codehaus.groovy.grails.plugins.testing`) which extends Spring's `MockHttpServletResponse` class and has a number of useful methods for inspecting the state of response.

For example to test a redirect you can use the `redirectedUrl` property:

```
// Test class
class SimpleController {
    def index() {
        redirect action: 'hello'
    }
    …
}
```

```
void testIndex() {
    controller.index()

    assert response.redirectedUrl == '/simple/hello'
}
```

Many actions make use of the parameter data associated with the request. For example, the 'sort', 'max',
'offset' parameters are quite common. Providing these in the test is as simple as adding appropriate values
special `params` variable:

```
void testList() {
    params.sort = "name"
    params.max = 20
    params.offset = 0

    controller.list()
    …
}
```

You can even control what type of request the controller action sees by setting the `method` property or
mock request:

```
void testSave() {
    request.method = "POST"
    controller.save()
    …
}
```

This is particularly important if your actions do different things depending on the type of the request. Fin
you can mark a request as AJAX like so:

```
void testGetPage() {
    request.method = "POST"
    request.makeAjaxRequest()
    controller.getPage()
    …
}
```

You only need to do this though if the code under test uses the `xhr` property on the request.

Testing View Rendering

To test view rendering you can inspect the state of the controller's `modelAndView` property (an instance

`org.springframework.web.servlet.ModelAndView`) or you can use the `view` and `mo`

properties provided by the mixin:

```
// Test class
class SimpleController {
    def home() {
        render view: "homePage", model: [title: "Hello World"]
    }
    …
}
```

```
void testIndex() {
    controller.home()

assert view == "/simple/homePage"
    assert model.title == "Hello World"
}
```

Note that the view string is the absolute view path, so it starts with a '/' and will include path elements, suc

the directory named after the action's controller.

Testing Template Rendering

Unlike view rendering, template rendering will actually attempt to write the template directly to the resp

rather than returning a `ModelAndView` hence it requires a different approach to testing.

Consider the following controller action:

```
class SimpleController {
    def display() {
        render template:"snippet"
    }
}
```

In this example the controller will look for a template in `grails-app/views/simple/_snippet.`

. You can test this as follows:

```
void testDisplay() {
    controller.display()
    assert response.text == 'contents of template'
}
```

However, you may not want to render the real template, but just test that is was rendered. In this case you

provide mock Groovy Pages:

298

```
void testDisplay() {
    views['/simple/_snippet.gsp'] = 'mock contents'
    controller.display()
    assert response.text == 'mock contents'
}
```

Testing Actions Which Return A Map

When a controller action returns a `java.util.Map` that `Map` may be inspected directly to assert th
contains the expected data:

```
class SimpleController {
    def showBookDetails() {
        [title: 'The Nature Of Necessity', author: 'Alvin Plantinga']
    }
}
```

```
import grails.test.mixin.*

@TestFor(SimpleController)
class SimpleControllerTests {

void testShowBookDetails() {
        def model = controller.showBookDetails()

assert model.author == 'Alvin Plantinga'
    }
}
```

Testing XML and JSON Responses

XML and JSON response are also written directly to the response. Grails' mocking capabilities provide s
conveniences for testing XML and JSON response. For example consider the following action:

```
def renderXml() {
    render(contentType:"text/xml") {
        book(title:"Great")
    }
}
```

This can be tested using the `xml` property of the response:

```
void testRenderXml() {
    controller.renderXml()
    assert "<book title='Great'/>" == response.text
    assert "Great" == response.xml.@title.text()
}
```

The `xml` property is a parsed result from Groovy's XmlSlurper class which is very convenient for par
XML.

Testing JSON responses is pretty similar, instead you use the `json` property:

```
// controller action
def renderJson() {
    render(contentType:"text/json") {
        book = "Great"
    }
}
```

```
// test
void testRenderJson() {

controller.renderJson()

assert '{"book":"Great"}' == response.text
    assert "Great" == response.json.book
}
```

The `json` property is an instance of `org.codehaus.groovy.grails.web.json.JSONElem` which is a map-like structure that is useful for parsing JSON responses.

Testing XML and JSON Requests

Grails provides various convenient ways to automatically parse incoming XML and JSON packets. example you can bind incoming JSON or XML requests using Grails' data binding:

```
def consumeBook() {
    def b = new Book(params['book'])

render b.title
}
```

To test this Grails provides an easy way to specify an XML or JSON packet via the `xml` or `json` proper For example the above action can be tested by specifying a String containing the XML:

```
void testConsumeBookXml() {
    request.xml = '<book><title>The Shining</title></book>'
    controller.consumeBook()

assert response.text == 'The Shining'
}
```

Or alternatively a domain instance can be specified and it will be auto-converted into the appropriate X request:

```
void testConsumeBookXml() {
    request.xml = new Book(title:"The Shining")
    controller.consumeBook()

assert response.text == 'The Shining'
}
```

The same can be done for JSON requests:

```
void testConsumeBookJson() {
    request.json = new Book(title:"The Shining")
    controller.consumeBook()

assert response.text == 'The Shining'
}
```

If you prefer not to use Grails' data binding but instead manually parse the incoming XML or JSON that
be tested too. For example consider the controller action below:

```
def consume() {
    request.withFormat {
        xml {
            render request.XML.@title
        }
        json {
            render request.JSON.title
        }
    }
}
```

To test the XML request you can specify the XML as a string:

```
void testConsumeXml() {
    request.xml = '<book title="The Stand" />'

controller.consume()

assert response.text == 'The Stand'
}
```

And, of course, the same can be done for JSON:

```
void testConsumeJson() {
    request.json = '{title:"The Stand"}'
    controller.consume()

assert response.text == 'The Stand'
}
```

Testing Spring Beans

When using `TestFor` only a subset of the Spring beans available to a running Grails application
available. If you wish to make additional beans available you can do so with the `defineBeans` method
`GrailsUnitTestMixin`:

```
class SimpleController {
    SimpleService simpleService
    def hello() {
        render simpleService.sayHello()
    }
}
```

```
void testBeanWiring() {
    defineBeans {
        simpleService(SimpleService)
    }

controller.hello()

assert response.text == "Hello World"
}
```

The controller is auto-wired by Spring just like in a running Grails application. Autowiring even occurs if instantiate subsequent instances of the controller:

```
void testAutowiringViaNew() {
    defineBeans {
        simpleService(SimpleService)
    }

def controller1 = new SimpleController()
    def controller2 = new SimpleController()

assert controller1.simpleService != null
    assert controller2.simpleService != null
}
```

Testing Mime Type Handling

You can test mime type handling and the `withFormat` method quite simply by setting the respon `format` attribute:

```
// controller action
def sayHello() {
    def data = [Hello:"World"]
    withFormat {
        xml { render data as XML }
        html data
    }
}
```

```
// test
void testSayHello() {
    response.format = 'xml'
    controller.sayHello()

    String expected = '<?xml version="1.0" encoding="UTF-8"?>' +
                         '<map><entry key="Hello">World</entry></map>'

    assert expected == response.text
}
```

Testing Duplicate Form Submissions

Testing duplicate form submissions is a little bit more involved. For example if you have an action
handles a form such as:

```
def handleForm() {
    withForm {
        render "Good"
    }.invalidToken {
        render "Bad"
    }
}
```

you want to verify the logic that is executed on a good form submission and the logic that is executed
duplicate submission. Testing the bad submission is simple. Just invoke the controller:

```
void testDuplicateFormSubmission() {
    controller.handleForm()
    assert "Bad" == response.text
}
```

Testing the successful submission requires providing an appropriate SynchronizerToken:

```
import org.codehaus.groovy.grails.web.servlet.mvc.SynchronizerToken
...
void testValidFormSubmission() {
    def token = SynchronizerToken.store(session)
    params[SynchronizerToken.KEY] = token.currentToken.toString()

    controller.handleForm()
    assert "Good" == response.text
}
```

If you test both the valid and the invalid request in the same test be sure to reset the response betv
executions of the controller:

```
controller.handleForm() // first execution
…
response.reset()
…
controller.handleForm() // second execution
```

303

Testing File Upload

You use the GrailsMockMultipartFile class to test file uploads. For example consider the follow
controller action:

```
def uploadFile() {
    MultipartFile file = request.getFile("myFile")
    file.transferTo(new File("/local/disk/myFile"))
}
```

To test this action you can register a GrailsMockMultipartFile with the request:

```
void testFileUpload() {
    final file = new GrailsMockMultipartFile("myFile", "foo".bytes)
    request.addFile(file)
    controller.uploadFile()

    assert file.targetFileLocation.path == "/local/disk/myFile"
}
```

The GrailsMockMultipartFile constructor arguments are the name and contents of the file. It h
mock implementation of the transferTo method that simply records the targetFileLocation
doesn't write to disk.

Testing Command Objects

Special support exists for testing command object handling with the mockCommandObject method.
example consider the following action:

```
def handleCommand(SimpleCommand simple) {
    if (simple.hasErrors()) {
        render "Bad"
    }
    else {
        render "Good"
    }
}
```

To test this you mock the command object, populate it and then validate it as follows:

```
void testInvalidCommand() {
    def cmd = mockCommandObject(SimpleCommand)
    cmd.name = '' // doesn't allow blank names

cmd.validate()
    controller.handleCommand(cmd)

assert response.text == 'Bad'
}
```

Testing Calling Tag Libraries

You can test calling tag libraries using `ControllerUnitTestMixin`, although the mechanism for te the tag called varies from tag to tag. For example to test a call to the `message` tag, add a message to `messageSource`. Consider the following action:

```
def showMessage() {
    render g.message(code: "foo.bar")
}
```

This can be tested as follows:

```
void testRenderBasicTemplateWithTags() {
    messageSource.addMessage("foo.bar", request.locale, "Hello World")

controller.showMessage()

assert response.text == "Hello World"
}
```

9.1.2 Unit Testing Tag Libraries

The Basics

Tag libraries and GSP pages can be tested with the `grails.test.mixin.web.GroovyPageUnitTestMixin` mixin. To use the mixin declare which library is under test with the `TestFor` annotation:

```
@TestFor(SimpleTagLib)
class SimpleTagLibTests {

}
```

Note that if you are testing invocation of a custom tag from a controller you can combine `ControllerUnitTestMixin` and the `GroovyPageUnitTestMixin` using the `Mock` annotation:

```
@TestFor(SimpleController)
@Mock(SimpleTagLib)
class GroovyPageUnitTestMixinTests {

}
```

Testing Custom Tags

The core Grails tags don't need to be enabled during testing, however custom tag libraries do. `GroovyPageUnitTestMixin` class provides a `mockTagLib()` method that you can use to mo custom tag library. For example consider the following tag library:

305

```
class SimpleTagLib {

static namespace = 's'

def hello = { attrs, body ->
        out << "Hello ${attrs.name ?: 'World'}"
    }
}
```

You can test this tag library by using `TestFor` and supplying the name of the tag library:

```
@TestFor(SimpleTagLib)
class SimpleTagLibTests {
    void testHelloTag() {
        assert applyTemplate('<s:hello />') == 'Hello World'
        assert applyTemplate('<s:hello name="Fred" />') == 'Hello Fred'
    }
}
```

Alternatively, you can use the `TestMixin` annotation and mock multiple tag libraries using mockTagLib() method:

```
@grails.test.mixin.TestMixin(GroovyPageUnitTestMixin)
class MultipleTagLibraryTests {

@Test
    void testMuliple() {
        mockTagLib(FirstTagLib)
        mockTagLib(SecondTagLib)

…
    }
}
```

The `GroovyPageUnitTestMixin` provides convenience methods for asserting that the template out equals or matches an expected value.

```
@grails.test.mixin.TestMixin(GroovyPageUnitTestMixin)
class MultipleTagLibraryTests {

@Test
    void testMuliple() {
        mockTagLib(FirstTagLib)
        mockTagLib(SecondTagLib)
        assertOutputEquals ('Hello World', '<s:hello />')
        assertOutputMatches (/.*Fred.*/, '<s:hello name="Fred" />')
    }
}
```

Testing View and Template Rendering

You can test rendering of views and templates in `grails-app/views` via the `render(Map)` me provided by `GroovyPageUnitTestMixin`:

```
def result = render(template: "/simple/hello")
assert result == "Hello World"
```

This will attempt to render a template found at the location `grails-app/views/simple/_hello.`
. Note that if the template depends on any custom tag libraries you need to call `mockTagLib` as describe
the previous section.

9.1.3 Unit Testing Domains

Overview

> ⚠️ The mocking support described here is best used when testing non-domain artifacts that use domain classes, to let you focus on testing the artifact without needing a database. But when testing persistence it's best to use integration tests which configure Hibernate and use a database.

Domain class interaction can be tested without involving a database connection usi
`DomainClassUnitTestMixin`. This implementation mimics the behavior of GORM agains
in-memory `ConcurrentHashMap` implementation. Note that this has limitations compared to a real GC
implementation. The following features of GORM for Hibernate can only be tested within an integration t

- String-based HQL queries

- composite identifiers

- dirty checking methods

- any direct interaction with Hibernate

However a large, commonly-used portion of the GORM API can be mocked usi
`DomainClassUnitTestMixin` including:

- Simple persistence methods like `save()`, `delete()` etc.

- Dynamic Finders

- Named Queries

- Query-by-example

- GORM Events

If something isn't supported then `GrailsUnitTestMixin`'s `mockFor` method can come in handy to m
the missing pieces. Alternatively you can write an integration test which bootstraps the complete G
environment at a cost of test execution time.

The Basics

`DomainClassUnitTestMixin` is typically used in combination with testing either a controller, servic
tag library where the domain is a mock collaborator defined by the `Mock` annotation:

```
import grails.test.mixin.*

@TestFor(SimpleController)
@Mock(Simple)
class SimpleControllerTests {

}
```

The example above tests the `SimpleController` class and mocks the behavior of the `Simple` do
class as well. For example consider a typical scaffolded `save` controller action:

```
class BookController {
    def save() {
        def book = new Book(params)
        if (book.save(flush: true)) {
            flash.message = message(
                    code: 'default.created.message',
                    args: [message(code: 'book.label',
                                        default: 'Book'), book.id])}"
            redirect(action: "show", id: book.id)
        }
        else {
            render(view: "create", model: [bookInstance: book])
        }
    }
}
```

Tests for this action can be written as follows:

```
import grails.test.mixin.*

@TestFor(BookController)
@Mock(Book)
class BookControllerTests {

void testSaveInvalidBook() {
        controller.save()

assert model.bookInstance != null
        assert view == '/book/create'
    }

void testSaveValidBook() {
        params.title = "The Stand"
        params.pages = "500"

controller.save()

assert response.redirectedUrl == '/book/show/1'
        assert flash.message != null
        assert Book.count() == 1
    }
}
```

`Mock` annotation also supports a list of mock collaborators if you have more than one domain to mock:

```
@TestFor(BookController)
@Mock([Book, Author])
class BookControllerTests {
    …
}
```

Alternatively you can also use the `DomainClassUnitTestMixin` directly with the `TestMi`
annotation:

```
import grails.test.mixin.domain.DomainClassUnitTestMixin

@TestFor(BookController)
@TestMixin(DomainClassUnitTestMixin)
class BookControllerTests {
    …
}
```

And then call the `mockDomain` method to mock domains during your test:

```
void testSave() {
    mockDomain(Author)
    mockDomain(Book)
}
```

The `mockDomain` method also includes an additional parameter that lets you pass a Map of Map
configure a domain, which is useful for fixture-like data:

```
void testSave() {
    mockDomain(Book, [
            [title: "The Stand", pages: 1000],
            [title: "The Shining", pages: 400],
            [title: "Along Came a Spider", pages: 300] ])
}
```

Testing Constraints

Your constraints contain logic and that logic is highly susceptible to bugs - the kind of bugs that can be tr
to track down (particularly as by default `save()` doesn't throw an exception when it fails). If your answ
that it's too hard or fiddly, that is no longer an excuse. Enter the `mockForConstraintsTests()` meth

This method is like a much reduced version of the `mockDomain()` method that simply adds
`validate()` method to a given domain class. All you have to do is mock the class, create an instance
populated data, and then call `validate()`. You can then access the `errors` property to determin
validation failed. So if all we are doing is mocking the `validate()` method, why the optional list of
instances? That is so that we can test the `unique` constraint as you will soon see.

So, suppose we have a simple domain class:

```
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 (they're not!), they are for demonstration only. To
these constraints we can do the following:

```
@TestFor(Book)
class BookTests {
    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()

assert !book.validate()
        assert "nullable" == book.errors["title"]
        assert "nullable" == book.errors["author"]

// So let's demonstrate the unique and minSize constraints

book = new Book(title: "Misery", author: "JK")
        assert !book.validate()
        assert "unique" == book.errors["title"]
        assert "minSize" == book.errors["author"]

// Validation should pass!
        book = new Book(title: "The Shining", author: "Stephen King")
        assert book.validate()
    }
}
```

You can probably look at that code and work out what's happening without any further explanation. The
thing we will explain is the way the `errors` property is used. First, is a real Spring `Errors` instance, so
can access all the properties and methods you would normally expect. Second, this particular `Errors` ol
also has map/property access as shown. Simply specify the name of the field you are interested in and
map/property access will return the name of the constraint that was violated. Note that it is the const
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
catches a common error: typos in the "constraints" property name! It is currently one of the hardest bug
track down normally, and yet a unit test for your constraints will highlight the problem straight away.

9.1.4 Unit Testing Filters

Unit testing filters is typically a matter of testing a controller where a filter is a mock collaborator.
example consider the following filters class:

```
class CancellingFilters {
    def filters = {
        all(controller:"simple", action:"list") {
            before = {
                redirect(controller:"book")
                return false
            }
        }
    }
}
```

This filter interceptors the `list` action of the `simple` controller and redirects to the `book` controller. To this filter you start off with a test that targets the `SimpleController` class and add CancellingFilters as a mock collaborator:

```
@TestFor(SimpleController)
@Mock(CancellingFilters)
class SimpleControllerTests {

}
```

You can then implement a test that uses the `withFilters` method to wrap the call to an action in execution:

```
void testInvocationOfListActionIsFiltered() {
    withFilters(action:"list") {
        controller.list()
    }
    assert response.redirectedUrl == '/book'
}
```

Note that the `action` parameter is required because it is unknown what the action to invoke is until action is actually called. The `controller` parameter is optional and taken from the controller under te it is a another controller you are testing then you can specify it:

```
withFilters(controller:"book",action:"list") {
    controller.list()
}
```

9.1.5 Unit Testing URL Mappings

The Basics

Testing URL mappings can be done with the `TestFor` annotation testing a particular URL mappings c For example to test the default URL mappings you can do the following:

```
@TestFor(UrlMappings)
class UrlMappingsTests {

}
```

311

> ⚠️ Note that since the default `UrlMappings` class is in the default package your test must also be in the default package

With that done there are a number of useful methods that are defined by the `grails.test.mixin.web.UrlMappingsUnitTestMixin` for testing URL mappings. These include:

- `assertForwardUrlMapping` - Asserts a URL mapping is forwarded for the given controller (note that controller will need to be defined as a mock collaborate for this to work)

- `assertReverseUrlMapping` - Asserts that the given URL is produced when reverse mapping a to a given controller and action

- `assertUrlMapping` - Asserts a URL mapping is valid for the given URL. This combines `assertForwardUrlMapping` and `assertReverseUrlMapping` assertions

Asserting Forward URL Mappings

You use `assertForwardUrlMapping` to assert that a given URL maps to a given controller. example, consider the following URL mappings:

```
static mappings = {
    "/action1"(controller: "simple", action: "action1")
    "/action2"(controller: "simple", action: "action2")
}
```

The following test can be written to assert these URL mappings:

```
void testUrlMappings() {
assertForwardUrlMapping("/action1", controller: 'simple',
                                    action: "action1")

assertForwardUrlMapping("/action2", controller: 'simple',
                                    action: "action2")

shouldFail {
        assertForwardUrlMapping("/action2", controller: 'simple',
                                    action: "action1")
    }
}
```

Assert Reverse URL Mappings

You use `assertReverseUrlMapping` to check that correct links are produced for your URL map when using the `link` tag in GSP views. An example test is largely identical to the previous listing except use `assertReverseUrlMapping` instead of `assertForwardUrlMapping`. Note that you combine these 2 assertions with `assertUrlMapping`.

Simulating Controller Mapping

In addition to the assertions to check the validity of URL mappings you can also simulate mapping controller by using your `UrlMappings` as a mock collaborator and the `mapURI` method. For example:

```
@TestFor(SimpleController)
@Mock(UrlMappings)
class SimpleControllerTests {

void testControllerMapping() {

SimpleController controller = mapURI('/simple/list')
        assert controller != null

def model = controller.list()
        assert model != null
    }
}
```

9.1.6 Mocking Collaborators

Beyond the specific targeted mocking APIs there is also an all-purpose `mockFor()` method that is avail when using the `TestFor` annotation. The signature of `mockFor` is:

```
mockFor(class, loose = false)
```

This is general-purpose mocking that lets you 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 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 -> … }
strictControl.demand.static.aStaticMethod {-> … }
```

Notice that you can mock static as well as instance methods by using the "static" property. You then spe the name of the method to mock, with an optional range argument. This range determines how many ti you expect the method to be called, and if the number of invocations falls outside of that range (either too or too many) then an assertion error will be thrown. If no range is specified, a default of "1..1" is assumed 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 clo arguments must match the number and types of the mocked method, but otherwise you are free to whatever you want in the body.

As we mentioned before, call `mockControl.createMock()` to get an actual mock instance of the that you are mocking. You can call this multiple times to create as many mock instances as you need. once you have executed the test method, call `mockControl.verify()` to check that the expe methods were called.

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 not matter.

9.2 Integration Testing

Integration tests differ from unit tests in that you have full access to the Grails environment within the Grails uses an in-memory H2 database for integration tests and clears out all the data from the data between tests.

One thing to bear in mind is that logging is enabled for your application classes, but it is different f logging in tests. So if you have something like this:

```
class MyServiceTests extends GroovyTestCase {
    void testSomething() {
        log.info "Starting tests"
        …
    }
}
```

the "starting tests" message is logged using a different system than the one used by the application. The property in the example above is an instance of `java.util.logging.Logger` (inherited from the class, not injected by Grails), which doesn't have the same methods as the `log` property injected into application artifacts. For example, it doesn't have `debug()` or `trace()` methods, and the equivalent `warn()` is in fact `warning()`.

Transactions

Integration tests run inside a database transaction by default, which is rolled back at the end of the each This means that data saved during a test is not persisted to the database. Add a `transactional` proper your test class to check transactional behaviour:

```
class MyServiceTests extends GroovyTestCase {
    static transactional = false

    void testMyTransactionalServiceMethod() {
            …
    }
}
```

Be sure to remove any persisted data from a non-transactional test, for example in the `tearDown` method these tests don't interfere with standard transactional tests that expect a clean database.

Testing Controllers

To test controllers you first have to understand the Spring Mock Library.

Grails automatically configures each test with a [MockHttpServletRequest](), [MockHttpServletResponse](), [MockHttpSession]() that you can use in 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:

```
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 `response` is an instance of `MockHttpServletResponse` which we can use to ob
the generated content with `contentAsString` (when writing to the response) or the redirected URL. T
mocked versions of the Servlet API are completely mutable (unlike the real versions) and hence you car
properties on the request such as the `contextPath` and so on.

Grails **does not** invoke [interceptors](#) or servlet filters when calling actions during integration testing.
should test interceptors and filters in isolation, using [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 f
your test.

Given a controller using a service:

```
class FilmStarsController {
    def popularityService

def update() {
        // do something with popularityService
    }
}
```

The test for this would be:

315

```
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 comm 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 ok from the mocked request parameters. During controller testing, the params are mutable with a moc request supplied by Grails.

Testing Controllers and the render Method

The render method lets you render a custom view at any point within the body of an action. For insta 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 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

You can use the Spring [MockHttpServletRequest](#) to test an action that requires request data, for example REST web service. For example consider this action which performs data binding from an incoming request

```
def create() {
    [book: new Book(params.book)]
}
```

To 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>
            '''.stripIndent().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
}
```

317

> ⚠ With JSON don't forget the `class` property to specify the name the target type to bind to. In XML this is implicit within the name of the `<book>` node, but this property is required 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` w subclasses Spring Web Flow's [AbstractFlowExecutionTests](#) class.

> ⚠ Subclasses of `WebFlowTestCase` **must** be integration tests

For example given this simple flow:

```
class ExampleController {

def exampleFlow() {
        start {
            on("go") {
                flow.hello = "world"
            }.to "next"
        }
        next {
            on("back").to "start"
            on("go").to "subber"
        }
        subber {
            subflow(action: "sub")
            on("end").to("end")
        }
        end()
    }
def subFlow() {
        subSubflowState {
            subflow(controller: "other", action: "otherSub")
            on("next").to("next")
        }
        …
    }
}
```

You need to tell the test harness what to use for the "flow definition". This is done via overriding the abst `getFlow` method:

```
import grails.test.WebFlowTestCase

class ExampleFlowTests extends WebFlowTestCase {
    def getFlow() { new ExampleController().exampleFlow }
    …
}
```

You can specify the flow id by overriding the `getFlowId` method, otherwise the default is `test`:

```
import grails.test.WebFlowTestCase

class ExampleFlowTests extends WebFlowTestCase {
    String getFlowId() { "example" }
    …
}
```

If the flow under test calls any subflows, these (or mocks) must be registered before the calling the flow:

```
protected void setUp() {
    super.setUp()

registerFlow("other/otherSub") { // register a simplified mock
        start {
            on("next").to("end")
        }
        end()
    }

// register the original subflow
    registerFlow("example/sub", new ExampleController().subFlow)
}
```

Then you kick off the flow with the `startFlow` method:

```
void testExampleFlow() {
    def viewSelection = startFlow()
    …
}
```

Use the `signalEvent` method to trigger an event:

```
void testExampleFlow() {
    …
    signalEvent("go")
    assert "next" == flowExecution.activeSession.state.id
    assert "world" == flowScope.hello
}
```

Here we have signaled to the flow to execute the event "go" which causes a transition to the "next" state the example a transition action placed a `hello` variable into the flow scope.

Testing Tag Libraries

Testing tag libraries is simple because when a tag is invoked as a method it returns its result as a st (technically a `StreamCharBuffer` but this class implements all of the methods of `String`). So 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).toString()
    }

void testBodyTag() {
        assertEquals "<p>Hello World!</p>",
                     new FooTagLib().bodyTag(name: "p") {
                         "Hello World!"
                     }.toString()
    }
}
```

Notice that for the second example, `testBodyTag`, we pass a block that returns the body of the tag. Th
convenient to representing the body as a String.

Testing Tag Libraries with GroovyPagesTestCase

In addition to doing simple testing of tag libraries like in the above examples, you can also use
`grails.test.GroovyPagesTestCase` class to test tag libraries with integration tests.

The `GroovyPagesTestCase` class is a subclass of the standard `GroovyTestCase` class and adds u
methods for testing the output of GSP rendering.

> ⚠ `GroovyPagesTestCase` can only be used in an integration test.

For example, consider this date formatting tag library:

```
import java.text.SimpleDateFormat

class FormatTagLib {
    def dateFormat = { attrs, body ->
        out << new 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](), but there are a few things t
aware of. Firstly, when testing queries you often need to "flush" to ensure the correct state has been persi
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 fail because calling [save]() does not actually persist the `Book` instances when called. Calling s
only indicates to Hibernate that at some point in the future these instances should be persisted. To con
changes immediately you "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 persi
immediately and hence will be available to the query later on.

9.3 Functional Testing

Functional tests involve making HTTP requests against the running application and verifying the resu
behaviour. Grails does not ship with any support for writing functional tests directly, but there are sev
plugins available for this.

- `Canoo Webtest` - http://grails.org/plugin/webtest

- `G-Func` - http://grails.org/plugin/functional-test

- `Geb` - http://grails.org/plugin/geb

- `Selenium-RC` - http://grails.org/plugin/selenium-rc

- `WebDriver` - http://grails.org/plugin/webdriver

Consult the documentation for each plugin for its capabilities.

Common Options

There are options that are common to all plugins that control how the Grails application is launched, if at a

inline

The `-inline` option specifies that the grails application should be started inline (i.e. like `run-app`).

This option is implicitly set unless the `baseUrl` or `war` options are set

war

The `-war` option specifies that the grails application should be packaged as a war and started. This is us
as it tests your application in a production-like state, but it has a longer startup time than the `-inline` op
It also runs the war in a forked JVM, meaning that you cannot access any internal application objects.

```
grails test-app functional: -war
```

Note that the same build/config options for the run-war command apply to functional testing against
WAR.

https

The `-https` option results in the application being able to receive https requests as well as http requests.
compatible with both the `-inline` and `-war` options.

```
grails test-app functional: -https
```

Note that this does not change the test *base url* to be https, it will still be http unless the `-httpsBase`
option is also given.

httpsBaseUrl

The `-httpsBaseUrl` causes the implicit base url to be used for tests to be a https url.

```
grails test-app functional: -httpsBaseUrl
```

This option is ignored if the `-baseUrl` option is specified.

baseUrl

The `baseUrl` option allows the base url for tests to be specified.

```
grails test-app functional: -baseUrl=http://mycompany.com/grailsapp
```

This option will prevent the local grails application being started unless `-inline` or `-war` are given as v
To use a custom base url but still test against the local Grails application you **must** specify one of either
`-inline` or `-war` options.

323

10 Internationalization

Grails supports Internationalization (i18n) out of the box by leveraging the underlying Spring M internationalization support. With Grails you are able to customize the text that appears in a view based o user's Locale. To quote the javadoc for the [Locale](#) class:

> *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 eng whilst "en_GB" is the for British English.

10.1 Understanding Message Bundles

Now that you have an idea of locales, to use them in Grails you create message bundle file containing different languages that you wish to render. Message bundles in Grails are located inside t `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 several message bundles for a whole range of languages within the `grails-app/i18n` directory. example:

- messages.properties
- messages_da.properties
- messages_de.properties
- messages_es.properties
- messages_fr.properties
- ...

By default Grails looks in `messages.properties` for messages unless the user has specified a lo You can create your own message bundle by simply creating a new properties file that ends with the l 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 provide users the capability to switch locales by simply passing a parameter called `lang` to Grails as a req parameter:

```
/book/list?lang=es
```

Grails will automatically switch the user's locale and store it in a cookie so subsequent requests will have new header.

10.3 Reading Messages

Reading Messages in the View

The most common place that you need messages is inside the view. Use the message tag for this:

```
<g:message code="my.localized.content" />
```

As long as you have a key in your `messages.properties` (with appropriate locale suffix) such as the below then Grails will look up the message:

```
my.localized.content=Hola, Me llamo John. Hoy es domingo.
```

Messages can also include arguments, for example:

```
<g:message code="my.localized.content" args="${ ['Juan', 'lunes'] }" />
```

The message declaration specifies positional parameters which are dynamically specified:

```
my.localized.content=Hola, Me llamo {0}. Hoy es {1}.
```

Reading Messages in Controllers and Tag Libraries

It's simple to read messages in a controller since you can invoke tags as methods:

```
def show() {
    def msg = message(code: "my.localized.content", args: ['Juan', 'lunes'])
}
```

The same technique can be used in tag libraries, but if your tag library uses a custom namespace then must prefix the call with `g.`:

```
def myTag = { attrs, body ->
    def msg = g.message(code: "my.localized.content", args: ['Juan', 'lunes'])
}
```

10.4 Scaffolding and i18n

Grails scaffolding templates for controllers and views are fully i18n-aware. The GSPs use the message tag labels, buttons etc. and controller `flash` messages use i18n to resolve locale-specific messages.

11 Security

Grails is no more or less secure than Java Servlets. However, Java servlets (and hence Grails) are extrem... secure and largely immune to common buffer overrun and malformed URL exploits due to the nature of... Java Virtual Machine underpinning the code.

Web security problems typically occur due to developer naivety or mistakes, and there is a little Grails ca... 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 !... 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 esca... mechanisms to prevent code injection

4. Grails provides [codecs](#) to let you trivially escape data when rendered as HTML, JavaScript and URI... prevent injection attacks here.

11.1 Securing Against Attacks

SQL injection

Hibernate, which is the technology underlying GORM domain classes, automatically escapes data w... committing to database so this is not an issue. However it is still possible to write bad dynamic HQL code... uses unchecked request parameters. For example doing the following is vulnerable to HQL injection attack...

```
def vulnerable() {
    def books = Book.find("from Book as b where b.title ='" + params.title + "'")
}
```

or the analagous call using a GString:

```
def vulnerable() {
    def books = Book.find("from Book as b where b.title ='${params.title}'")
}
```

Do **not** do this. Use named or positional parameters instead to pass in parameters:

```
def safe() {
    def books = Book.find("from Book as b where b.title = ?",
                          [params.title])
}
```

or

```
def safe() {
    def books = Book.find("from Book as b where b.title = :title",
                          [title: params.title])
}
```

Phishing

This really a public relations issue in terms of avoiding hijacking of your branding and a decl
communication policy with your customers. Customers need to know how to identify valid emails.

XSS - cross-site scripting injection

It is important that your application verifies as much as possible that incoming requests were originated f
your application and not from another site. Ticketing and page flow systems can help this and Grails' sup
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 v
rendering to HTML or XHTML you must call encodeAsHTML on every object to ensure that people ca
maliciously inject JavaScript or other HTML into data or tags viewed by others. Grails supplies sev
Dynamic Encoding Methods for this purpose and if your output escaping format is not supported you
easily write your own codec.

You must also avoid the use of request parameters or data fields for determining the next URL to redirec
user to. If you use a `successURL` parameter for example to determine where to redirect a user to af
successful login, attackers can imitate your login procedure using your own site, and then redirect the
back to their own site once logged in, potentially allowing JavaScript code to then exploit the logge
account on the site.

Cross-site request forgery

CSRF involves unauthorized commands being transmitted from a user that a website trusts. A typical exan
would be another website embedding a link to perform an action on your website if the user is
authenticated.

The best way to decrease risk against these types of attacks is to use the `useToken` attribute on your fo
See Handling Duplicate Form Submissions for more information on how to use it. An additional mea
would be to not use remember-me cookies.

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 wil
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 supplie
Grails all use encodeAsURL where appropriate. If you create your own tags that generate URLs you will i
to be mindful of doing this too.

Denial of service

327

Load balancers and other appliances are more likely to be useful here, but there are also issues relatin
excessive queries for example where a link is created by an attacker to set the maximum value of a resul
so that a query could exceed the memory limits of the server or slow the system down. The solution here
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 elsewh
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 be
returning the response to the user.

Not doing this is "security through obscurity" which is inevitably breached, just like having a default passv
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
Grails. Grails also supports a simple mechanism for developers to contribute their own codecs that wil
recognized at runtime.

Codec Classes

A Grails codec class is one that may contain an encode closure, a decode closure or both. When a G
application starts up the Grails framework dynamically loads codecs from the `grails-app/uti`
directory.

The framework looks under `grails-app/utils/` for class names that end with the convention Coc
For example one of the standard codecs that ships with Grails is `HTMLCodec`.

If a codec contains an `encode` closure Grails will create a dynamic `encode` method and add that methc
the `Object` class with a name representing the codec that defined the encode closure. For example,
`HTMLCodec` class defines an `encode` closure, so Grails attaches it with the name `encodeAsHTML`.

The `HTMLCodec` and `URLCodec` classes also define a `decode` closure, so Grails attaches those with
names `decodeHTML` and `decodeURL` respectively. Dynamic codec methods may be invoked t
anywhere in a Grails application. For example, consider a case where a report contains a property ca
'description' which may contain special characters that must be escaped to be presented in an H1
document. One way to deal with that in a GSP is to encode the description property using the dynamic enc
method as shown below:

```
${report.description.encodeAsHTML()}
```

Decoding is performed using `value.decodeHTML()` syntax.

Standard Codecs

HTMLCodec

This codec performs HTML escaping and unescaping, so that values can be rendered safely in an HTML [without creating any HTML tags or damaging the page layout. For example, given a value "Don't you k that 2 > 1?" you wouldn't be able to show this safely within an HTML page because the > will look li closes a tag, which is especially bad if you render this data within an attribute, such as the value attribute of 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 affecting your page.

URLCodec

URL encoding is required when creating URLs in links or form actions, or any time data is used to cre URL. It prevents illegal characters from getting into the URL and changing its meaning, for example "A & Blackberry" is not going to work well as a parameter in a GET request as the ampersand will b parameter parsing.

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

Escapes Strings so they can be used as valid JavaScript strings. For example:

```
Element.update('${elementId}',
    '${render(template: "/common/message").encodeAsJavaScript()}')
```

HexCodec

Encodes byte arrays or lists of integers to lowercase hexadecimal strings, and can decode hexadecimal str
into byte arrays. For example:

```
Selected colour: #${[255,127,255].encodeAsHex()}
```

MD5Codec

Uses the MD5 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default sys
encoding), as a lowercase hexadecimal string. Example of usage:

```
Your API Key: ${user.uniqueID.encodeAsMD5()}
```

MD5BytesCodec

Uses the MD5 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default sys
encoding), as a byte array. Example of usage:

```
byte[] passwordHash = params.password.encodeAsMD5Bytes()
```

SHA1Codec

Uses the SHA1 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default sys
encoding), as a lowercase hexadecimal string. Example of usage:

```
Your API Key: ${user.uniqueID.encodeAsSHA1()}
```

SHA1BytesCodec

Uses the SHA1 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default sys
encoding), as a byte array. Example of usage:

```
byte[] passwordHash = params.password.encodeAsSHA1Bytes()
```

SHA256Codec

Uses the SHA256 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default sys
encoding), as a lowercase hexadecimal string. Example of usage:

```
Your API Key: ${user.uniqueID.encodeAsSHA256()}
```

SHA256BytesCodec

Uses the SHA256 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default sys
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 cus
codec class must be defined in the `grails-app/utils/` directory and the class name must end w
`Codec`. The codec may contain a `static encode` closure, a `static decode` closure or both.
closure must accept a single argument which will be the object that the dynamic method was invoked on.
Example:

```
class PigLatinCodec {
  static encode = { str ->
    // convert the string to pig latin and return the result
  }
}
```

With the above codec in place an application could do something like this:

```
${lastName.encodeAsPigLatin()}
```

11.3 Authentication

Grails has no default mechanism for authentication as it is possible to implement authentication in n
different ways. It is however, easy to implement a simple authentication mechanism using either intercep
or filters. This is sufficient for simple use cases but it's highly preferable to use an established sec
framework, for example by using the Spring Security or the Shiro plugin.

Filters let you apply authentication across all controllers or across a URI space. For example you can crea
new set of filters in a class called `grails-app/conf/SecurityFilters.groovy` by running:

```
grails create-filters security
```

and implement your interception logic there:

```
class SecurityFilters {
    def filters = {
        loginCheck(controller: '*', action: '*') {
            before = {
                if (!session.user && actionName != "login") {
                    redirect(controller: "user", action: "login")
                    return false
                }
            }
        }
    }
}
```

Here the `loginCheck` filter intercepts execution *before* all actions except `login` are executed, and if t
is no user in the session then redirect to the `login` action.

The `login` action itself is simple too:

```groovy
def login() {
    if (request.get) {
        return // render the login view
    }
    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 Plugins

If you need more advanced functionality beyond simple authentication such as authorization, roles etc.
you should consider using one of the available security plugins.

11.4.1 Spring Security

The Spring Security plugins are built on the [Spring Security](#) project which provides a flexible, extens
framework for building all sorts of authentication and authorization schemes. The plugins are modular so
can install just the functionality that you need for your application. The Spring Security plugins are the off
security plugins for Grails and are actively maintained and supported.

There is a [Core plugin](#) which supports form-based authentication, encrypted/salted passwords, HTTP E
authentication, etc. and secondary dependent plugins provide alternate functionality such as [Ope](#)
[authentication](#), [ACL support](#), [single sign-on with Jasig CAS](#), [LDAP authentication](#), [Kerberos authentica](#)
and a plugin providing [user interface extensions](#) and security workflows.

See the [Core plugin page](#) for basic information and the [user guide](#) for detailed information.

11.4.2 Shiro

[Shiro](#) is a Java POJO-oriented security framework that provides a default domain model that models rea
users, roles and permissions. With Shiro you extend a controller base class called called `JsecAuthBas`
each controller you want secured and then provide an `accessControl` block to setup the roles. An exan
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 Shiro plugin refer to the documentation.

12 Plugins

Grails is first and foremost a web application framework, but it is also a platform. By exposing a number of extension points that let you extend anything from the command line interface to the runtime configuration engine, Grails can be customised to suit almost any needs. To hook into this platform, all you need to do is create a plugin.

Extending the platform may sound complicated, but plugins can range from trivially simple to incredibly powerful. If you know how to build a Grails application, you'll know how to create a plugin for sharing a model or some static resources.

12.1 Creating and Installing Plugins

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. For example running `grails create-plugin example` would create a new plugin project called `example`.

The structure of a Grails plugin is very nearly the same as a Grails application project's 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 test your plugin by running:

```
grails run-app
```

The plugin descriptor name 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. The plugin class defines the version of the plugin and other metadata, 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 plugin. Valid values include example "0.1", "0.2-SNAPSHOT", "1. etc.

- `grailsVersion` - The version of version range of Grails that the plugin supports. eg. "1.2 (indicating 1.2 or higher)

- `author` - plugin author's name

- `authorEmail` - plugin author's contact e-mail

- `description` - full multi-line description of plugin's features

- `documentation` - URL of the plugin's documentation

Here is an example from the [Quartz Grails plugin](#):

```
class QuartzGrailsPlugin {
    def version = "0.1"
    def grailsVersion = "1.1 > *"
    def author = "Sergey Nebolsin"
    def authorEmail = "nebolsin@gmail.com"
    def title = "Quartz Plugin"
    def description = '''\
The 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/plugin/quartz"

    …
}
```

Installing and Distributing Plugins

To distribute a plugin you navigate to its root directory in a console and run:

```
grails package-plugin
```

This will create a zip file of the plugin starting with `grails-` then the plugin name and version. For exam with the example plugin created earlier this would be `grails-example-0.1.zip`. T `package-plugin` command will also generate a `plugin.xml` file which contains machine-read 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 run:

```
grails install-plugin /path/to/grails-example-0.1.zip
```

If the plugin is hosted on an HTTP server you can install it with:

```
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 plugin can be run as a G
application, not all of these files are included when packaging a plugin. The following is a list of artef
created, but not included by [package-plugin](#):

- `grails-app/conf/BootStrap.groovy`

- `grails-app/conf/BuildConfig.groovy` (although it is used to generat
  `dependencies.groovy`)

- `grails-app/conf/Config.groovy`

- `grails-app/conf/DataSource.groovy` (and any other `*DataSource.groovy`)

- `grails-app/conf/UrlMappings.groovy`

- `grails-app/conf/spring/resources.groovy`

- Everything within `/web-app/WEB-INF`

- Everything within `/web-app/plugins/**`

- Everything within `/test/**`

- SCM management files within `**/.svn/**` and `**/CVS/**`

If you need artefacts within `WEB-INF` it is recommended you use the `_Install.groovy` script (cov
later), which is executed when a plugin is installed, to provide such artefacts. In addition, althou
`UrlMappings.groovy` is excluded you are allowed to include a `UrlMappings` definition wi
different name, such as `MyPluginUrlMappings.groovy`.

Specifying Plugin Locations

An application can load plugins from anywhere on the file system, even if they have not been insta
Specify the location of the (unpacked) plugin in the application's
`grails-app/conf/BuildConfig.groovy` file:

```
// Useful to test plugins you are developing.
grails.plugin.location.shiro =
        "/home/dilbert/dev/plugins/grails-shiro"

// 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 installi[ng] first.

- You have split an application into a set of plugins and an application, all in the same "super-proj[ect]" directory.

Global plugins

Plugins can also be installed globally for all applications for a particular version of Grails using the `-glo[bal]` flag, for example:

```
grails install-plugin webtest -global
```

The default location is $USER_HOME/.grails/<grailsVersion>/global-plugins but this can be customized [by] the `grails.global.plugins.dir` setting in `BuildConfig.groovy`.

12.2 Plugin Repositories

Distributing Plugins in the Grails Central Plugin Repository

The preferred way to distribute plugin is to publish to the official Grails Central Plugin Repository. This [will] make your plugin visible to the [list-plugins](#) command:

```
grails list-plugins
```

which lists all plugins that are in the central repository. Your plugin will also be available to the [plugin-[info]](#) command:

```
grails plugin-info [plugin-name]
```

which prints extra information about it, such as its description, who wrote, etc.

> ⚠ If you have created a Grails plugin and want it to be hosted in the central repository, you'll find instructions for getting an account on [this wiki page](#).

When you have access to the Grails Plugin repository, install the [Release Plugin](#) and execute [the] `publish-plugin` command to release your plugin:

```
grails install-plugin release
grails publish-plugin
```

This will automatically commit any remaining source code changes to your SCM provider and then publish the plugin to the central repository. If the command is successful, it will immediately be available on the plugin portal at http://grails.org/plugin/<pluginName>. You can find out more about the Release plugin and other features in [its user guide](#).

Configuring Additional Repositories

The process for configuring repositories in Grails differs between versions. For version of Grails 1.2 earlier please refer to the [Grails 1.2 documentation](#) on the subject. The following sections cover Grails 1.3 above.

Grails 1.3 and above use Ivy under the hood to resolve plugin dependencies. The mechanism for defi additional plugin repositories is largely the same as [defining repositories for JAR dependencies](#). For exar you can define a remote Maven repository that contains Grails plugins using the following syntax `grails-app/conf/BuildConfig.groovy`:

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

// ...or with a name
    mavenRepo name: "myRepo",
              root: "http://myserver:8081/artifactory/plugins-snapshots-local"
}
```

You can also define a SVN-based Grails repository (such as the one hosted at http://plugins.grails.org) u the `grailsRepo` method:

```
repositories {
    grailsRepo "http://myserver/mygrailsrepo"

// ...or with a name
    grailsRepo "http://myserver/svn/grails-plugins", "mySvnRepo"
}
```

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 in this case the G central repository will be searched last:

```
repositories {
    grailsRepo "http://myserver/mygrailsrepo"
    grailsCentral()
}
```

All of the above examples use HTTP; however you can specify any [Ivy resolver](#) to resolve plugins v 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-[artifact]-[revision].[ext]")
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 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 plugins. The benefits of doing so include the ability for existing tooling and repository managers to inter the structure of a Maven repository. In addition Maven compatible repositories are not tied to SVN as G repositories are.

You use the Maven publisher plugin to publish a plugin to a Maven repository. Please refer to the sectio the [Maven deployment](#) user guide on the subject.

Publishing to Grails Compatible Repositories

Specify the `grails.plugin.repos.distribution.myRepository` setting within grails-app/conf/BuildConfig.groovy file to publish a Grails plugin to a Grails-compatible repository:

```
grails.plugin.repos.distribution.myRepository =
        "https://svn.codehaus.org/grails/trunk/grails-test-plugin-repo"
```

You can also provide this settings in the $USER_HOME/.grails/settings.groovy file if you prefer to share same settings across multiple projects.

Once this is done use the `repository` argument of the `release-plugin` command to specify repository to release the plugin into:

```
grails release-plugin -repository = myRepository
```

12.3 Understanding a Plugin's Structure

As as mentioned previously, a plugin is basically a regular Grails application with a plugin descri However when installed, the structure of a plugin differs slightly. For example, take a look at this pl directory structure:

```
   + grails-app
        + controllers
        + domain
        + taglib
        etc.
    + lib
    + src
        + java
        + groovy
    + web-app
        + js
        + css
```

When a plugin is installed the contents of the `grails-app` directory will go into a directory such
plugins/example-1.0/grails-app. They **will not** be copied into the main source tree. A pl
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 s
resources go in the `web-app` directory. You can then link to static resources just like in an application.
example 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 l
/js/mycode.js. However, when the plugin is installed into an application the path will automatic
change to something like /plugin/example-0.1/js/mycode.js and Grails will deal with ma
sure the resources are in the right place.

There is a special `pluginContextPath` variable that can be used whilst both developing the plugin
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
/plugins/example depending on whether the plugin is running standalone or has been installed i
application

Java and Groovy code that the plugin provides within the lib and `src/java` and `src/groovy` direct
will be compiled into the main project's `web-app/WEB-INF/classes` directory so that they are n
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 in its scripts directory:

```
  + MyPlugin.groovy
    + scripts      <-- additional scripts here
    + grails-app
        + controllers
        + services
        + etc.
     + lib
```

Adding a new grails-app artifact (Controller, Tag Library, Service, etc.)

A plugin can add new artifacts by creating the relevant file within the `grails-app` tree. Note that the pl
is 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 plugin provides a controller it may also provide default views to be rendered. This is an excellent
to modularize your application through plugins. Grails' view resolution mechanism will first look for the
in the application it is installed into and if that fails will attempt to look for the view within the plugin.
means that you can override views provided by a plugin by creating corresponding GSPs in the applicati
`grails-app/views` directory.

For example, consider a controller called `BookController` that's provided by an 'amazon' plugin. If
action being executed is `list`, Grails will first look for a view calle
`grails-app/views/book/list.gsp` then if that fails it will look for the same view relative to
plugin.

However if the view uses templates that are also provided by the plugin then the following syntax ma
necessary:

```
  <g:render template="fooTemplate" plugin="amazon"/>
```

Note the usage of the `plugin` attribute, which contains the name of the plugin where the template reside
this is not specified then Grails will look for the template relative to the application.

Excluded Artefacts

By default Grails excludes the following files during the packaging process:

- `grails-app/conf/BootStrap.groovy`

- `grails-app/conf/BuildConfig.groovy` (although it is used to generat
  `dependencies.groovy`)

- `grails-app/conf/Config.groovy`

- `grails-app/conf/DataSource.groovy` (and any other `*DataSource.groovy`)

- `grails-app/conf/UrlMappings.groovy`

- `grails-app/conf/spring/resources.groovy`

- Everything within `/web-app/WEB-INF`

- Everything within `/web-app/plugins/**`

- Everything within `/test/**`

- SCM management files within `**/.svn/**` and `**/CVS/**`

If your plugin requires files under the `web-app/WEB-INF` directory it is recommended that you modify
plugin's `scripts/_Install.groovy` Gant script to install these artefacts into the target proj
directory tree.

In addition, the default `UrlMappings.groovy` file is excluded to avoid naming conflicts, however you
free to add a UrlMappings definition under a different name which **will** be included. For example a file ca
`grails-app/conf/BlogUrlMappings.groovy` is fine.

The list of excludes is extensible with the `pluginExcludes` property:

```
// resources that are excluded from plugin packaging
def pluginExcludes = [
    "grails-app/views/error.gsp"
]
```

This is useful for example to include demo or test resources in the plugin repository, but not include the
the final distribution.

12.5 Evaluating Conventions

Before looking at providing runtime configuration based on conventions you first need to understand ho
evaluate those conventions from a plugin. Every plugin has an implicit `application` variable which i
instance of the [GrailsApplication](#) interface.

The `GrailsApplication` interface provides methods to evaluate the conventions within the project
internally stores references to all artifact classes within your application.

Artifacts implement the [GrailsClass](#) interface, which represents a Grails resource such as a controller or
library. For example to get all `GrailsClass` instances you can do:

```
for (grailsClass in application.allClasses) {
    println grailsClass.name
}
```

`GrailsApplication` has a few "magic" properties to narrow the type of artefact you are interested in. example to access controllers you can use:

```
for (controllerClass in application.controllerClasses) {
    println controllerClass.name
}
```

The dynamic method conventions are as follows:

- `*Classes` - Retrieves all the classes for a particular artefact name. For examp application.controllerClasses.

- `get*Class` - Retrieves a named class for a particular artefact. For examp application.getControllerClass("PersonController")

- `is*Class` - Returns `true` if the given class is of the given artefact type. For examp application.isControllerClass(PersonController)

The `GrailsClass` interface has a number of useful methods that let you further evaluate and work wit 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 p 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 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 applica without the trailing convention part if applicable

- `getNaturalName` - Returns the name of the property in natural terms (eg. 'lastName' becomes ' 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 plugins can do post-install configuration and participate in application upgrade process (the upg command). This is achieved using two specially named scripts under the `scripts` directory of the plug `_Install.groovy` and `_Upgrade.groovy`.

`_Install.groovy` is executed after the plugin has been installed and `_Upgrade.groovy` is exec
each time the user upgrades the application (but not the plugin) with [upgrade](#) command.

These scripts are [Gant](#) scripts, so you can use the full power of Gant. An addition to the standard (
variables there is also a `pluginBasedir` variable which points at the plugin installation basedir.

As an example this `_Install.groovy` script will create a new directory type under the `grails-`
directory and install a configuration template:

```
ant.mkdir(dir: "${basedir}/grails-app/jobs")

ant.copy(file: "${pluginBasedir}/src/samples/SamplePluginConfig.groovy",
         todir: "${basedir}/grails-app/conf")
```

Scripting events

It is also possible to hook into command line scripting events. These are events triggered during executio
Grails target and plugin scripts.

For example, you can hook into status update output (i.e. "Tests passed", "Server running") and the creatio
files or artefacts.

A plugin just has to provide an `_Events.groovy` script to listen to the required events. Refer
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 run
configuration by convention.

Hooking into the Grails Spring configuration

First, you can hook in Grails runtime configuration by providing a property called `doWithSpring` whic
assigned a block of code. For example the following snippet is from one of the core Grails plugins
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 configures the Grails `messageSource` bean and a couple of other beans to manage Lc 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 directly, they can participate in the generation of the file. A plugin can provide a `doWithWebDescrip` property that is assigned a block of code that gets passed the `web.xml` as an XmlSlurper GPathResu

Add `servlet` and `servlet-mapping`

Consider this 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 gets a reference to the last `<servlet-mapping>` element and appends Grails' servlet it using XmlSlurper's ability to programmatically modify XML using closures and blocks.

Add `filter` and `filter-mapping`

Adding a filter with its mapping works a little differently. The location of the `<filter>` element do matter since order is not important, so it's simplest to insert your custom filter definition immediately afte last `<context-param>` element. Order *is* important for mappings, but the usual approach is to ac immediately after the last `<filter>` element like so:

```
def doWithWebDescriptor = { webXml ->

def contextParam = webXml.'context-param'

contextParam[contextParam.size() - 1] + {
        'filter' {
            'filter-name'('springSecurityFilterChain')
            'filter-class'(DelegatingFilterProxy.name)
        }
    }

def filter = webXml.'filter'
    filter[filter.size() - 1] + {
        'filter-mapping'{
            'filter-name'('springSecurityFilterChain')
            'url-pattern'('/*')
        }
    }
}
```

In some cases you need to ensure that your filter comes after one of the standard Grails filters, such as the Spring character encoding filter or the SiteMesh filter. Fortunately you can insert filter mappings immediately after the standard ones (more accurately, any that are in the template web.xml file) like so:

```
def doWithWebDescriptor = { webXml ->
    ...
// Insert the Spring Security filter after the Spring
    // character encoding filter.
    def filter = webXml.'filter-mapping'.find {
        it.'filter-name'.text() == "charEncodingFilter"
    }
filter + {
        'filter-mapping'{
            'filter-name'('springSecurityFilterChain')
            'url-pattern'('/*')
        }
    }
}
```

Doing Post Initialisation Configuration

Sometimes it is useful to be able do some runtime configuration after the Spring [ApplicationContext](#) has built. In this case you can define a doWithApplicationContext closure property.

```
class SimplePlugin {

def name = "simple"
    def version = "1.1"

def doWithApplicationContext = { appCtx ->
        def sessionFactory = appCtx.sessionFactory
        // do something here with session factory
    }
}
```

12.8 Adding Dynamic Methods at Runtime

The Basics

Grails plugins let you register dynamic methods with any Grails-managed or other class at runtime. This work is done in a doWithDynamicMethods closure.

For Grails-managed classes like controllers, tag libraries and so forth you can add methods, constructors using the [ExpandoMetaClass](#) mechanism by accessing each controller's [MetaClass](#):

```
class ExamplePlugin {
    def doWithDynamicMethods = { applicationContext ->
        for (controllerClass in application.controllerClasses) {
            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' MetaC
instances and add a new method called `myNewMethod` to each controller. If you know beforehand the c
you wish the add a method to you can simply reference its `metaClass` property.

For example we can add a new method `swapCase` to `java.lang.String`:

```
class ExamplePlugin {

def doWithDynamicMethods = { applicationContext ->
        String.metaClass.swapCase = {->
            def sb = new StringBuilder()
            delegate.each {
                sb << (Character.isUpperCase(it as char) ?
                        Character.toLowerCase(it as char) :
                        Character.toUpperCase(it as char))
            }
            sb.toString()
        }
assert "UpAndDown" == "uPaNDdOWN".swapCase()
    }
}
```

Interacting with the ApplicationContext

The `doWithDynamicMethods` closure gets passed the Spring `ApplicationContext` instance. Th
useful as it lets you interact with objects within it. For example if you were implementing a method to int
with Hibernate you could use the `SessionFactory` instance in combination with
`HibernateTemplate`:

```
import org.springframework.orm.hibernate3.HibernateTemplate

class ExampleHibernatePlugin {

def doWithDynamicMethods = { applicationContext ->

for (domainClass in application.domainClasses) {

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
implement more powerful dynamic constructors that use the application context to wire dependencies
your object at runtime:

```
class MyConstructorPlugin {

def doWithDynamicMethods = { applicationContext ->
        for (domainClass in application.domainClasses) {
            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 perform some action when they occur. This is
Grails implements advanced reloading of application state at runtime. For example, consider this simple
snippet from the Grails `ServicesPlugin`:

```
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))
            }
        }
    }
}
```

First it defines `watchedResources` as either a String or a List of strings that contain either the refere
or patterns of the resources to watch. If the watched resources specify a Groovy file, when it is changed it
automatically be reloaded and passed into the `onChange` closure in the `event` object.

The `event` object defines a number of useful properties:

- `event.source` - The source of the event, 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

- `event.manager` - The `GrailsPluginManager` instance

These objects are available to help you apply the appropriate changes based on what changed. In "Services" example above, a new service bean is re-registered with the `ApplicationContext` when of the service classes changes.

Influencing Other Plugins

In addition to reacting to changes, sometimes a plugin needs to "influence" another.

Take for example the Services and Controllers plugins. When a service is reloaded, unless you reload controllers too, problems will occur when you try to auto-wire the reloaded service into an older contro Class.

To get around this, you can specify which plugins another plugin "influences". This means that when plugin detects a change, it will reload itself and then reload its influenced plugins. For example consider 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 resou 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 plugi

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 cha while the application is running.

12.10 Understanding Plugin Load Order

Controlling Plugin Dependencies

Plugins often depend on the presence of other plugins and can adapt depending on the presence of others. is implemented with two properties. The first is called `dependsOn`. For example, take a look at this sni from the Hibernate plugin:

```
class HibernateGrailsPlugin {

def version = "1.0"

def dependsOn = [dataSource: "1.0",
                 domainClass: "1.0",
                 i18n: "1.0",
                 core: "1.0"]
}
```

The Hibernate plugin is dependent on the presence of four plugins: the `dataSource`, `domainCla`
`i18n` and `core` plugins.

The dependencies will be loaded before the Hibernate plugin and if all dependencies do not load, then
plugin will not load.

The `dependsOn` property also supports a mini expression language for specifying version ranges. A
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
suffixes such as -BETA, -ALPHA etc. so for example the expression "1.0 > 1.1" would match any of
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
give up and won't load. It is possible though to have a weaker dependency using the `loadAfter`
`loadBefore` properties:

```
def loadAfter = ['controllers']
```

Here the plugin will be loaded after the `controllers` plugin if it exists, otherwise it will just be loa
The plugin can then adapt to the presence of the other plugin, for example the Hibernate plugin has this
in its `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 t
controllers plugin has been loaded. The manager variable is an instance of the [GrailsPluginMan](#)
interface and it provides methods to interact with other plugins.

You can also use the loadBefore property to specify one or more plugins that your plugin should
before:

```
def loadBefore = ['rabbitmq']
```

Scopes and Environments

It's not only plugin load order that you can control. You can also specify which environments your pl
should be loaded in and which scopes (stages of a build). Simply declare one or both of these propertie
your plugin descriptor:

```
def environments = ['development', 'test', 'myCustomEnv']
def scopes = [excludes:'war']
```

In this example, the plugin will only load in the 'development' and 'test' environments. Nor will it be packa
into the WAR file, because it's excluded from the 'war' phase. This allows development-only plugi
not be packaged for production use.

The full list of available scopes are defined by the enum [BuildScope](#), but here's a summary:

- test - when running tests
- functional-test - when running functional tests
- run - for run-app and run-war
- war - when packaging the application as a WAR file
- all - plugin applies to all scopes (default)

Both properties can be one of:

- a string - a sole inclusion
- a list - a list of environments or scopes to include
- a map - for full control, with 'includes' and/or 'excludes' keys that can have string or list values

For example,

351

```
def environments = "test"
```

will only include the plugin in the test environment, whereas

```
def environments = ["development", "test"]
```

will include it in both the development *and* test environments. Finally,

```
def environments = [includes: ["development", "test"]]
```

will do the same thing.

12.11 The Artefact API

You should by now understand that Grails has the concept of artefacts: special types of classes that it kn
about and can treat differently from normal Groovy and Java classes, for example by enhancing them
extra properties and methods. Examples of artefacts include domain classes and controllers. What you may
be aware of is that Grails allows application and plugin developers access to the underlying infrastructure
artefacts, which means you can find out what artefacts are available and even enhance them yourself. You
even provide your own custom artefact types.

12.11.1 Asking About Available Artefacts

As a plugin developer, it can be important for you to find out about what domain classes, controllers, or c
types of artefact are available in an application. For example, the [Searchable plugin](#) needs to know
domain classes exist so it can check them for any `searchable` properties and index the appropriate ones
how does it do it? The answer lies with the `grailsApplication` object, and instance of [GrailsApplica](#)
that's available automatically in controllers and GSPs and can be [injected](#) everywhere else.

The `grailsApplication` object has several important properties and methods for querying artef
Probably the most common is the one that gives you all the classes of a particular artefact type:

```
for (cls in grailsApplication.<artefactType>Classes) {
    …
}
```

In this case, `artefactType` is the property name form of the artefact type. With core Grails you have:

- domain
- controller
- tagLib
- service
- codec
- bootstrap
- urlMappings

So for example, if you want to iterate over all the domain classes, you use:

```
for (cls in grailsApplication.domainClasses) {
    …
}
```

and for URL mappings:

```
for (cls in grailsApplication.urlMappingsClasses) {
    …
}
```

You need to be aware that the objects returned by these properties are not instances of [Class](). Instead, they instances of [GrailsClass]() that has some particularly useful properties and methods, including one fo underlying `Class`:

- `shortName` - the class name of the artefact without the package (equivalent of `Class.simpleN`).
- `logicalPropertyName` - the artefact name in property form without the 'type' suffix. `MyGreatController` becomes 'myGreat'.
- `isAbstract()` - a boolean indicating whether the artefact class is abstract or not.
- `getPropertyValue(name)` - returns the value of the given property, whether it's a static o instance one. This works best if the property is initialised on declaration, e.g. `sta transactional = true`.

The artefact API also allows you to fetch classes by name and check whether a class is an artefact:

- get<type>Class(String name)
- is<type>Class(Class clazz)

The first method will retrieve the `GrailsClass` instance for the given name, e.g. 'MyGreatController'. second will check whether a class is a particular type of artefact. For example, you can u `grailsApplication.isControllerClass(org.example.MyGreatController)` to cl whether `MyGreatController` is in fact a controller.

353

12.11.2 Adding Your Own Artefact Types

Plugins can easily provide their own artefacts so that they can easily find out what implementations available and take part in reloading. All you need to do is create an `ArtefactHandler` implementation register it in your main plugin class:

```
class MyGrailsPlugin {
    def artefacts = [ org.somewhere.MyArtefactHandler ]
    …
}
```

The `artefacts` list can contain either handler classes (as above) or instances of handlers.

So, what does an artefact handler look like? Well, put simply it is an implementation of the [ArtefactHandler](#) interface. To make life a bit easier, there is a skeleton implementation that can readily be extend [ArtefactHandlerAdapter](#).

In addition to the handler itself, every new artefact needs a corresponding wrapper class that impleme [GrailsClass](#). Again, skeleton implementations are available such as [AbstractInjectableGrailsClass](#), which particularly useful as it turns your artefact into a Spring bean that is auto-wired, just like controllers services.

The best way to understand how both the handler and wrapper classes work is to look at the Quartz plugin:

- [GrailsJobClass](#)

- [DefaultGrailsJobClass](#)

- [JobArtefactHandler](#)

Another example is the [Shiro plugin](#) which adds a realm artefact.

12.12 Binary Plugins

Regular Grails plugins are packaged as zip files containing the full source of the plugin. This has s advantages in terms of being an open distribution system (anyone can see the source), in addition to avoi problems with the source compatibility level used for compilation.

As of Grails 2.0 you can pre-compile Grails plugins into regular JAR files known as "binary plugins". This several advantages (and some disadvantages as discussed in the advantages of source plugins ab including:

- Binary plugins can be published as standard JAR files to a Maven repository

- Binary plugins can be declared like any other JAR dependency

- Commercial plugins are more viable since the source isn't published

- IDEs have a better understanding since binary plugins are regular JAR files containing classes

Packaging

To package a plugin in binary form you can use the package-plugin command and the `--binary` flag:

```
grails package-plugin --binary
```

Supported artefacts include:

- Grails artifact classes such as controllers, domain classes and so on

- I18n Message bundles

- GSP Views, layouts and templates

You can also specify the packaging in the plugin descriptor:

```
def packaging = "binary"
```

in which case the packaging will default to binary.

Using Binary Plugins

The packaging process creates a JAR file in the `target` directory of the plugin, for examp
`target/foo-plugin-0.1.jar`. There are two ways to incorporate a binary plugin into an applicatio

One is simply placing the plugin JAR file in your application's `lib` directory. The other is to publish
plugin JAR to a compatible Maven repository and declare it as a dependency
`grails-app/conf/BuildConfig.groovy`:

```
dependencies {
    compile "mycompany:myplugin:0.1"
}
```

> ⚠️ Since binary plugins are packaged as JAR files, they are declared as dependencies in the
> `dependencies` block, *not* in the `plugins` block as you may be naturally inclined to
> do. The `plugins` block is used for declaring traditional source plugins packaged as zip
> files

13 Web Services

Web services are all about providing a web API onto your web application and are typically implemente
either [REST](#) or [SOAP](#) .

13.1 REST

REST is not really a technology in itself, but more an architectural pattern. REST is very simple and
involves using plain XML or JSON as a communication medium, combined with URL patterns that
"representational" of the underlying system, and HTTP methods such as GET, PUT, POST and DELETE.

Each HTTP method maps to an action type. For example GET for retrieving data, PUT for creating
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")
}
```

This maps the URI /product onto a ProductController. Each HTTP method such as GET, F
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

In addition, Grails provides automatic XML or JSON marshalling for you.

You can alter how HTTP methods are handled by using 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 auton
XML or JSON marshalling unless you specify the parseRequest argument:

```
"/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 (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 ->
        for (book in json.books) {
            println book.title
        }
    }
  }
```

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 Marshalling - Reading

The controller can use Grails' XML marshalling support to implement the GET method:

```
import grails.converters.XML

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
        }
    }
    ..
}
```

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 MacBook.

XML Marshalling - Updating

To support updates such as `PUT` and `POST` you can use the [params](#) object which Grails enhances with 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, using [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 `product` key we can automatically create bind the XML using the `Product` constructor. 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, or an XML requ or a JSON request.

> ⚠ 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 u Grails' [validation](#) capabilities in the form:

```
<error>
    <message>The property 'title' of class 'Person' must be specified</message>
</error>
```

REST with JAX-RS

There also is a [JAX-RS Plugin](#) which can be used to build web services based on the Java API for RES Web Services ([JSR 311: JAX-RS](#)).

13.2 SOAP

There are several plugins that add SOAP support to Grails depending on your preferred approach. Contract First SOAP services there is a [Spring WS](#) plugin, whilst if you want to generate a SOAP API Grails services there are several plugins that do this including:

- [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. This exam is taken from the CXF plugin:

```
class BookService {
static expose = ['cxf']

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 CXF plugin refer to [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 the [render](#) method's XML capability. There is however a [Feeds plugin](#) available for Grails that provides a 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"

for (article in Article.list()) {
            entry(article.title) {
                link = "http://your.test.server/article/${article.id}"
                article.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 Spring Framework It is also useful for plugin 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 M web application framework. Although Spring MVC suffers from some of the same difficulties as framewo like Struts in terms of its ease of use, it is superbly designed and architected and was, for Grails, the per 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 to Gr controllers

- Data Binding and Validation - Grails' validation and data binding capabilities are built on those prov by Spring

- Runtime configuration - Grails' entire runtime convention based system is wired together by a Spr 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 constructed. The basics of it are as follows.

- Grails constructs a parent `ApplicationContext` from the `web-app/WEB-INF/applicationContext.xml` file. This `ApplicationContext` config the GrailsApplication instance and the GrailsPluginManager.

- Using this `ApplicationContext` as a parent Grails' analyses the conventions with t `GrailsApplication` instance and constructs a child `ApplicationContext` that is used as root `ApplicationContext` of the web application

Configured Spring Beans

Most of Grails' configuration happens at runtime. Each plugin may configure Spring beans that are regist in the `ApplicationContext`. For a reference as to which beans are configured, refer to the refere guide which describes each of the Grails plugins and which beans they configure.

14.2 Configuring Additional Beans

Using the Spring Bean DSL

You can easily register new (or override existing) beans by configuring them `grails-app/conf/spring/resources.groovy` which uses the Grails Spring DSL. Beans defined inside a `beans` property (a Closure):

```
beans = {
    // beans here
}
```

As a simple example you can configure a bean with the following syntax:

```
import my.company.MyBeanImpl

beans = {
    myBean(MyBeanImpl) {
        someProperty = 42
        otherProperty = "blue"
    }
}
```

Once configured, the bean can be auto-wired into Grails artifacts and other classes that support depend
injection (for example `BootStrap.groovy` and integration tests) by declaring a public field whose nam
your bean's name (in this case `myBean`):

```
class ExampleController {

def myBean
    …
}
```

Using the DSL has the advantage that you can mix bean declarations and logic, for example based on
environment:

```
import grails.util.Environment
import my.company.mock.MockImpl
import my.company.MyBeanImpl

beans = {
    switch(Environment.current) {
        case Environment.PRODUCTION:
            myBean(MyBeanImpl) {
                someProperty = 42
                otherProperty = "blue"
            }
            break

case Environment.DEVELOPMENT:
            myBean(MockImpl) {
                someProperty = 42
                otherProperty = "blue"
            }
            break
    }
}
```

The `GrailsApplication` object can be accessed with the `application` variable and can be use
access the Grails configuration (amongst other things):

```
import grails.util.Environment
import my.company.mock.MockImpl
import my.company.MyBeanImpl

beans = {
    if (application.config.my.company.mockService) {
        myBean(MockImpl) {
            someProperty = 42
            otherProperty = "blue"
        }
    } else {
        myBean(MyBeanImpl) {
            someProperty = 42
            otherProperty = "blue"
        }
    }
}
```

> ⚠ If you define a bean in `resources.groovy` with the same name as one previously registered by Grails or an installed plugin, your bean will replace the previous registration. This is a convenient way to customize behavior without resorting to editing plugin code or other approaches that would affect maintainability.

Using XML

Beans can also be configured using a `grails-app/conf/spring/resources.xml`. In ea versions of Grails this file was automatically generated for you by the `run-app` script, but the DSL `resources.groovy` is the preferred approach now so it isn't automatically generated now. But it is supported - you just need to create it yourself.

This file is typical Spring XML file and the Spring documentation has an [excellent reference](#) on how configure Spring beans.

The `myBean` bean that we configured using the DSL would be configured with this syntax in the XML file

```
<bean id="myBean" class="my.company.MyBeanImpl">
    <property name="someProperty" value="42" />
    <property name="otherProperty" value="blue" />
</bean>
```

Like the other bean it can be auto-wired into any class that supports dependency injection:

```
class ExampleController {

def myBean
}
```

Referencing Existing Beans

Beans declared in `resources.groovy` or `resources.xml` can reference other beans by convention.
example if you had a `BookService` class its Spring bean name would be `bookService`, so your b
would reference it like this in the DSL:

```groovy
beans = {
    myBean(MyBeanImpl) {
        someProperty = 42
        otherProperty = "blue"
        bookService = ref("bookService")
    }
}
```

or like this in XML:

```xml
<bean id="myBean" class="my.company.MyBeanImpl">
    <property name="someProperty" value="42" />
    <property name="otherProperty" value="blue" />
    <property name="bookService" ref="bookService" />
</bean>
```

The bean needs a public setter for the bean reference (and also the two simple properties), which in Gro
would be defined like this:

```groovy
package my.company

class MyBeanImpl {
    Integer someProperty
    String otherProperty
    BookService bookService // or just "def bookService"
}
```

or in Java like this:

```java
package my.company;

class MyBeanImpl {

private BookService bookService;
    private Integer someProperty;
    private String otherProperty;

public void setBookService(BookService theBookService) {
        this.bookService = theBookService;
    }

public void setSomeProperty(Integer someProperty) {
        this.someProperty = someProperty;
    }

public void setOtherProperty(String otherProperty) {
        this.otherProperty = otherProperty;
    }
}
```

Using `ref` (in XML or the DSL) is very powerful since it configures a runtime reference, so the referen
bean doesn't have to exist yet. As long as it's in place when the final application context configuration occ
everything will be resolved correctly.

For a full reference of the available beans see the plugin reference in the reference guide.

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 and annotations) is static and 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, 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
import grails.spring.BeanBuilder

def bb = new BeanBuilder()

bb.beans {

dataSource(BasicDataSource) {
        driverClassName = "org.h2.Driver"
        url = "jdbc:h2:mem:grailsDB"
        username = "sa"
        password = ""
    }

sessionFactory(ConfigurableLocalSessionFactoryBean) {
        dataSource = ref('dataSource')
        hibernateProperties = ["hibernate.hbm2ddl.auto": "create-drop",
                               "hibernate.show_sql":    "true"]
    }
}

ApplicationContext appContext = bb.createApplicationContext()
```

> ⚠ Within plugins 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.

This example shows how you would configure Hibernate with a data source with the `BeanBuilder` class.

Each method call (in this case `dataSource` and `sessionFactory` calls) maps 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 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 ->
    // Autowiring behaviour. The other option is 'byType'. [autowire]
    bean.autowire = 'byName'
    // Sets the initialisation method to 'init'. [init-method]
    bean.initMethod = 'init'
    // Sets the destruction method to 'destroy'. [destroy-method]
    bean.destroyMethod = 'destroy'
    // Sets the scope of the bean. [scope]
    bean.scope = 'request'
    dataSource = ref('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

Include the `grails-spring-<version>.jar` file in your classpath to use BeanBuilder in a regular Spring MVC application. Then add the following `<context-param>` values to 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
    </param-value>
</context-param>
```

Then create a /WEB-INF/applicationContext.groovy file that does the rest:

```
import org.apache.commons.dbcp.BasicDataSource

beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.h2.Driver"
        url = "jdbc:h2: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 matching syntax defined here. For example:

```
def bb = new BeanBuilder()
bb.loadBeans("classpath:*SpringBeans.groovy")

def applicationContext = bb.createApplicationContext()
```

Here the `BeanBuilder` loads all Groovy files on the classpath ending with `SpringBeans.groovy` parses them into bean definitions. An example script can be seen below:

```
import org.apache.commons.dbcp.BasicDataSource
import org.codehaus.groovy.grails.orm.hibernate.ConfigurableLocalSessionFactoryBean

beans {

dataSource(BasicDataSource) {
        driverClassName = "org.h2.Driver"
        url = "jdbc:h2: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`:

```
def binding = new Binding()
binding.maxSize = 10000
binding.productGroup = 'finance'

def bb = new BeanBuilder()
bb.binding = binding
bb.loadBeans("classpath:*SpringBeans.groovy")

def ctx = bb.createApplicationContext()
```

Then you can access the `maxSize` and `productGroup` properties in your DSL files.

14.4 The BeanBuilder DSL Explained

Using Constructor Arguments

Constructor arguments can be defined using parameters to each bean-defining method. Put them after the argument (the Class):

```
bb.beans {
    exampleBean(MyExampleBean, "firstArgument", 2) {
        someProperty = [1, 2, 3]
    }
}
```

This configuration corresponds to a `MyExampleBean` with a constructor that looks like this:

```
MyExampleBean(String foo, int bar) {
    …
}
```

Configuring the BeanDefinition (Using factory methods)

The first argument to the closure is a reference to the bean configuration instance, which you can us
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 directly from a new instance
Class, but from one of these factories. In this case the bean has no Class argument and instead you must
the name of the factory bean to the bean defining method:

```
bb.beans {

myFactory(ExampleFactoryBean) {
        someProperty = [1, 2, 3]
    }

myBean(myFactory) {
        name = "blah"
    }
}
```

Another common approach is provide the name of the factory method to call on the factory bean. This ca
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 to create the myB
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 st
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.
example has a hard-coded value but would work just as well with a name that is generated programmatic
based on configuration, system properties, etc.

Furthermore, because sometimes bean names are not known until runtime you may need to reference ther
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`. '
`ref` method can also be used to refer to beans from a parent `ApplicationContext` that is provided ir
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 th
the bean type:

```
bb.beans {

marge(Person) {
        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 b
reference. Alternatively if you have a factory bean you can omit the type and just use the specified b
definition instead to setup the factory:

```
bb.beans {

personFactory(PersonFactory)

marge(Person) {
        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 without a `Class` parameter:

```
class HolyGrailQuest {
    def start() { println "lets begin" }
}
```

```
class KnightOfTheRoundTable {

String name
    String leader
    HolyGrailQuest quest

KnightOfTheRoundTable(String name) {
        this.name = name
    }

def embarkOnQuest() {
        quest.start()
    }
}
```

```
import grails.spring.BeanBuilder

def bb = new BeanBuilder()
bb.beans {
    abstractBean {
        leader = "Lancelot"
    }
    …
}
```

Here we define an abstract bean that has a `leader` property with the value of `"Lancelot"`. To use abstract bean set it as the parent of the child bean:

```
bb.beans {
    …
    quest(HolyGrailQuest)

knights(KnightOfTheRoundTable, "Camelot") { bean ->
        bean.parent = abstractBean
        quest = ref('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` specified you can do it this way:

```
import grails.spring.BeanBuilder

def bb = new BeanBuilder()
bb.beans {

abstractBean(KnightOfTheRoundTable) { bean ->
        bean.'abstract' = true
        leader = "Lancelot"
    }
quest(HolyGrailQuest)

knights("Camelot") { bean ->
        bean.parent = abstractBean
        quest = quest
    }
}
```

In this example we create an abstract bean of type `KnightOfTheRoundTable` and use the bean argum
to set it to abstract. Later we define a knights bean that has no `Class` defined, but inherits the `Class` 1
the parent bean.

Using Spring Namespaces

Since Spring 2.0, users of Spring have had easier access to key features via XML namespaces. You can u
Spring namespace in BeanBuilder by declaring it with this syntax:

```
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 attribut

```
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")
```

This example will create a Spring bean with the identifier `dataSource` by performing a JNDI lookup o
given JNDI name. With Spring namespaces you also get full access to all of the powerful AOP suppo
Spring from BeanBuilder. For example given these 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 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(person)"
        }
    }
}
```

14.5 Property Placeholder Configuration

Grails supports the notion of property placeholder configuration through an extended version of Spring
PropertyPlaceholderConfigurer, which is typically useful in combination with externalized configuration.

Settings defined in either ConfigSlurper scripts or Java properties files can be used as placeholder values
Spring configuration in `grails-app/conf/spring/resources.xml`. For example given
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 combina
with [externalized configuration]().

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
your domain classes using Hibernate, either with XML mapping files or JPA annotations. You will be ab
map Grails domain classes onto a wider range of legacy systems and have more flexibility in the creatio
your database schema. Best of all, you will still be able to call all of the dynamic persistent and query met
provided by GORM!

15.1 Using Hibernate XML Mapping Files

Mapping your domain classes with XML is pretty straightforward. Simply create a `hibernate.cfg.`
file in your project's `grails-app/conf/hibernate` directory, either manually or with t
[create-hibernate-cfg-xml](#) command, that contains the following:

```xml
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Example mapping file inclusion -->
        <mapping resource="org.example.Book.hbm.xml"/>
        …
    </session-factory>
</hibernate-configuration>
```

The individual mapping files, like 'org.example.Book.hbm.xml' in the above example, also go into
`grails-app/conf/hibernate` directory. To find out how to map domain classes with XML, check
the [Hibernate manual](#).

If the default location of the `hibernate.cfg.xml` file doesn't suit you, you can change it by specifyin
alternative location in `grails-app/conf/DataSource.groovy`:

```groovy
hibernate {
    config.location = "file:/path/to/my/hibernate.cfg.xml"
}
```

or even a list of locations:

```groovy
hibernate {
    config.location = ["file:/path/to/one/hibernate.cfg.xml",
                        "file:/path/to/two/hibernate.cfg.xml"]
}
```

Grails also lets you write your domain model in Java or reuse an existing one that already has Hiber
mapping files. Simply place the mapping files into `grails-app/conf/hibernate` and either pu
Java files in `src/java` or the classes in the project's `lib` directory if the domain model is packaged
JAR. You still need the `hibernate.cfg.xml` though!

15.2 Mapping with Hibernate Annotations

To map a domain class with annotations, create a new class in `src/java` and use the annotations defined part of the EJB 3.0 spec (for more info on this see the [Hibernate Annotations Docs](#)):

```java
package com.books;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@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;
    }
}
```

Then register the class with the Hibernate `sessionFactory` by adding relevant entries to `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>
```

See the previous section for more information on the `hibernate.cfg.xml` file.

When Grails loads it will register the necessary dynamic methods with the class. To see what else you ca with a Hibernate domain class see the section on [Scaffolding](#).

15.3 Adding Constraints

375

You can still use GORM validation even if you use a Java domain model. Grails lets you define constraints through separate scripts in the `src/java` directory. The script must be in a directory that matches package of the corresponding domain class and its name must have a *Constraints* suffix. For example, if had a domain class `org.example.Book`, then you would create the script `src/java/org/example/BookConstraints.groovy`.

Add a standard GORM `constraints` block to the script:

```
constraints = {
    title blank: false
    author blank: false
}
```

Once this is in place you can validate instances of your domain class!

16 Scaffolding

Scaffolding lets you auto-generate a whole application for a given domain class including:

- The necessary [views](#)
- Controller actions for create/read/update/delete (CRUD) operations

Dynamic Scaffolding

The simplest way to get started with scaffolding is to enable it with the `scaffold` property. Set `scaffold` property in the controller to `true` for the `Book` domain class:

```
class BookController {
    static scaffold = true
}
```

This works because the `BookController` follows the same naming convention as the `Book` domain c To scaffold a specific domain class we could reference the class directly in the scaffold property:

```
class SomeController {
    static scaffold = Author
}
```

With this configured, when you start your application the actions and views will be auto-generated at runt The following actions are dynamically implemented by default by the runtime scaffolding mechanism:

- list
- show
- edit
- delete
- create
- save
- update

A CRUD interface will also be generated. To access this open `http://localhost:8080/app/boc` a browser.

If you prefer to keep your domain model in Java and [mapped with Hibernate](#) you can still use scaffold simply import the domain class and set its name as the `scaffold` argument.

You can add new actions to a scaffolded controller, for example:

```
class BookController {

static 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:

```
class BookController {

static scaffold = Book

// overrides scaffolded action to return both authors and books
    def list() {
        [bookInstanceList: Book.list(),
         bookInstanceTotal: Book.count(),
         authorInstanceList: Author.list()]
    }
def show() {
        def book = Book.get(params.id)
        log.error(book)
        [bookInstance : book]
    }
}
```

All of this is what is known as "dynamic scaffolding" where the CRUD interface is generated dynamical
runtime.

> ⚠ By default, the size of text areas in scaffolded views is defined in the CSS, so adding
> 'rows' and 'cols' attributes will have no effect.
>
> Also, the standard scaffold views expect model variables of the form
> `<propertyName>InstanceList` for collections and
> `<propertyName>Instance` for single instances. It's tempting to use properties like
> 'books' and 'book', but those won't work.

Customizing the Generated Views

The views adapt to [Validation constraints](#). For example you can change the order that fields appear in
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:

378

```
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 with a constraint also effects how many characters can be entered in the generated view

```
def constraints = {
    name(size:0..30)
}
```

Static Scaffolding

Grails also supports "static" scaffolding.

The above scaffolding features are useful but in real world situations it's likely that you will want to custor the logic and views. Grails lets you generate a controller and the views used to create the above interface f the command line. To generate a controller type:

```
grails generate-controller Book
```

or to generate the views:

```
grails generate-views Book
```

or to generate everything:

```
grails generate-all Book
```

If you have a domain class in a package or are generating from a [Hibernate mapped class](#) remember to inc 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 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 runnin
application during the development phase. An embedded Tomcat server is launched that loads the
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 p
Checking for and loading changes places a sizable overhead on the server. Having said that, `grails p`
`run-app` removes the per-request overhead and lets you fine tune how frequently the regular check t
place.

Setting the system property "disable.auto.recompile" to `true` disables this regular check completely, w
the property "recompile.frequency" controls the frequency. This latter property should be set to the numbe
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
development sources. Hot-reloading is disabled, so you get good performance without the hassle of havir
deploy the WAR file elsewhere.

WAR file

When it comes down to it, current java infrastructures almost mandate that web applications are deploye
WAR files, so this is by far the most common approach to Grails application deployment in produc
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
specify a path (either absolute or relative) to the command that instructs it where to place the file and v
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
default location and filename:

```
grails.project.war.file = "foobar-prod.war"
```

Any command line argument that you provide overrides this setting.

It is also possible to control what libraries are included in the WAR file, for example to avoid conflicts libraries in a shared directory. The default behavior is to include in the WAR file all libraries require Grails, plus any libraries contained in plugin "lib" directories, plus any libraries contained in the applicat "lib" directory. As an alternative to the default behavior you can explicitly specify the complete list of libra to include in the WAR file by setting the property `grails.war.dependencies` in BuildConfig.groo either lists of Ant include patterns or closures containing AntBuilder syntax. Closures are invoked from w an Ant "copy" step, so only elements like "fileset" can be included, whereas each item in a pattern li included. Any closure or pattern assigned to the latter property will be included in addition `grails.war.dependencies`.

Be careful with these properties: if any of the libraries Grails depends on are missing, the application 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") {
        for (pattern in deps) {
            include(name: pattern)
        }
    }
}
```

This example only exists to demonstrate the syntax for the properties. If you attempt to use it as is in your application, the application will probably not work. You can find a list of dependencies required by Grai the "dependencies.txt" file in the root directory of the unpacked distribution. You can also find a list of default dependencies included in WAR generation in the "War.groovy" script - see the `DEFAULT_DEPS` `DEFAULT_J5_DEPS` variables.

The remaining two configuration options available to you are `grails.war.copyToWebApp` a `grails.war.resources`. The first of these lets you customise what files are included in the WAR from the "web-app" directory. The second lets you do any extra processing you want before the WAR fi 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
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.

18 Contributing to Grails

Grails is an open source project with an active community and we rely heavily on that community to make Grails better. As such, there are various ways in which people can contribute to Grails. One of the by [writing useful plugins](#) and making them publicly available. In this chapter, we'll look at some of the options.

18.1 Report Issues in JIRA

Grails uses [JIRA](#) to track issues in both the core framework, its documentation, its website, and in many public plugins. If you've found a bug or wish to see a particular feature added, this is the place to start. Yo need to create a (free) JIRA account in order to either submit an issue or comment on an existing one.

When submitting issues, please provide as much information as possible and in the case of bugs, make you explain which versions of Grails and various plugins you are using. Also, an issue is much more like be dealt with if you attach a reproducible sample application (which can be packaged up using the `gra bug-report` command).

18.2 Build From Source and Run Tests

If you're interested in contributing fixes and features to the core framework, you will have to learn how to hold of the project's source, build it and test it with your own applications. Before you start, make sure have:

- A JDK (1.6 or above)

- A git client

Once you have all the pre-requisite packages installed, the next step is to download the Grails source c which is hosted at [GitHub](#) in several repositories owned by the ["grails" GitHub user](#). This is a simple cas cloning the repository you're interested in. For example, to getthe core framework run:

```
git clone http://github.com/grails/grails-core.git
```

This will create a "grails-core" directory in your current working directory containing all the project so files. The next step is to get a Grails installation from the source.

Creating a Grails installation

If you look at the project structure, you'll see that it doesn't look much like a standard GRAILS_H installation. But, it's very simple to turn it into one. Just run this from the root directory of the project:

```
./gradlew install
```

This will fetch all the standard dependencies required by Grails and then build a `GRAILS_HOME` installa Note that this target skips the extensive collection of Grails test classes, which can take some tim complete.

Once the above command has finished, simply set the `GRAILS_HOME` environment variable to the chec directory and add the "bin" directory to your path. When you next type run the `grails` command, you' using the version you just built.

Running the test suite

All you have to do to run the full suite of tests is:

```
./gradlew test
```

These will take a while (15-30 mins), so consider running individual tests using the command line. example, to run the test case `MappingDslTests` simply execute the following command:

```
./gradlew -Dtest.single=MappingDslTest :grails-test-suite-persistence:test
```

Note that you need to specify the sub-project that the test case resides in, because the top-level "test" ta won't work....

Developing in IntelliJ IDEA

You need to run the following gradle task:

```
./gradlew idea
```

Then open the project file which is generated in IDEA. Simple!

Developing in STS / Eclipse

You need to run the following gradle task:

```
./gradlew cleanEclipse eclipse
```

Before importing projects to STS do the following action:

- Edit grails-scripts/.classpath and remove the line "<classpathentry kind="src" path="../scripts"/>".

Use "Import->General->Existing Projects into Workspace" to import all projects to STS. There will be a build errors. To fix them do the following:

- A d d
  "~/.gradle/cache/com.springsource.springloaded/springloaded-core/jars/springloaded-core-XXXX.jar
  grails-core's classpath.

- Remove "src/test/groovy" from grails-plugin-testing's source path GRECLIPSE-1067

- Add "~/.gradle/cache/javax.servlet.jsp/jsp-api/jars/jsp-api-2.1.jar" to the classpath of grails-web

- Fix the source path of grails-scripts. Add linked source folder linking to "../scripts". If you get b
  errors in grails-scripts, do "../gradlew cleanEclipse eclipse" in that directory and edit the .classpath
  again (remove the line "<classpathentry kind="src" path="../scripts"/>"). Remove possible er
  "scripts" directory under grails-scripts if you are not able to add the linked folder.

- Do a clean build for the whole workspace.

- To use Eclipse GIT scm team provider: Select all projects (except "Servers") in the navigation and n
  click -> Team -> Share project (not "Share projects"). Choose "Git". Then check "Use or cr
  repository in parent folder of project" and click "Finish".

- Get the recommended code style settings from the [mailing list thread](#) (final style not decided
  currently [profile.xml](#)). Import the code style xml file to STS in Window->Preferences->Java->C
  Style->Formatter->Import . Grails code uses spaces instead of tabs for indenting.

Debugging Grails or a Grails application

To enable debugging, run:

```
grails-debug <command>
```

and then connect to the JVM remotely via the IDE ("remote debugging") using the port 5005. Of cours
you have modified the `grails-debug` script to use a different port number, connect using that one.

If you need to debug stuff that happens during application startup, then you should modify the "grails-del
script and change the "suspend" option from 'n' to 'y'. You can read more about the JPDA connection sett
TODO here: http://java.sun.com/j2se/1.5.0/docs/guide/jpda/conninv.html#Invocation.

It's also possible to get Eclipse to wait for incoming debugger connections and instead of u
"-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005" you could use
"-Xrunjdwp:transport=dt_socket,server=n,address=8000" (which assumes the Eclipse default port for rer
java applications) Inside eclipse you create a new "Remote Java Application" launch configuration and ch
the connection type to "Standard (Socket Listen)" and click debug. This allows you to start a debugger ses
in eclipse and just leave it running and you're free to debug anything without having to keep rememberir
relaunch a "Socket Attach" launch configuration. You might find it handy to have 2 scripts, one ca
"grails-debug", and another called "grails-debug-attach"

18.3 Submit Patches to Grails Core

If you want to submit patches to the project, you simply need to fork the repository on GitHub rather
clone it directly. Then you will commit your changes to your fork and send a pull request for a core t
member to review.

Forking and Pull Requests

One of the benefits of [GitHub](#) is the way that you can easily contribute to a project by [forking the reposi](#) and [sending pull requests](#) with your changes.

What follows are some guidelines to help ensure that your pull requests are speedily dealt with and pro the information we need. They will also make your life easier!

Create a local branch for your changes

Your life will be greatly simplified if you create a local branch to make your changes on. For example, as as you fork a repository and clone the fork locally, execute

```
git checkout -b mine
```

This will create a new local branch called "mine" based off the "master" branch. Of course, you can name branch whatever you like - you don't have to use "mine".

Create JIRAs for non-trivial changes

For any non-trivial changes, raise a JIRA issue if one doesn't already exist. That helps us keep track of changes go into each new version of Grails.

Include JIRA issue ID in commit messages

This may not seem particularly important, but having a JIRA issue ID in a commit message means that we find out at a later date why a change was made. Include the ID in any and all commits that relate to that is If a commit isn't related to an issue, then there's no need to include an issue ID.

Make sure your fork is up to date

Since the core developers must merge your commits into the main repository, it makes life much easier if fork on GitHub is up to date before you send a pull request.

Let's say you have the main repository set up as a remote called "upstream" and you want to submit a request. Also, all your changes are currently on the local "mine" branch but not on "master". The first involves pulling any changes from the main repository that have been added since you last fetched merged:

```
git checkout master
git pull upstream
```

This should complete without any problems or conflicts. Next, rebase your local branch against the up-to-date master:

```
git checkout mine
git rebase master
```

What this does is rearrange the commits such that all of your changes come after the most recent on master. Think adding some cards to the top of a deck rather than shuffling them into the pack.

You'll now be able to do a clean merge from your local branch to master:

```
git checkout master
git merge mine
```

Finally, you must push your changes to your remote repository on GitHub, otherwise the core develo
won't be able to pick them up:

```
git push
```

You're now ready to send the pull request from the GitHub user interface.

Say what your pull request is for

A pull request can contain any number of commits and it may be related to any number of issues. In the
request message, please specify the IDs of all issues that the request relates to. Also give a brief descriptic
the work you have done, such as: "I refactored the data binder and added support for custom number ed
(GRAILS-xxxx)".

18.4 Submit Patches to Grails Documentation

Contributing to the documentation is simpler than the core framework because there is a public fork of
http://github.com/grails/grails-doc project that anyone can request commit access to. So, if you want to su
patches to the documentation, simply request commit access to the following reposito
http://github.com/pledbrook/grails-doc by sending a GitHub message to 'pledbrook' and then commit
patches just as you would to any other GitHub repository.

Building the Guide

To build the documentation, simply type:

```
./gradlew docs
```

Be warned: this command can take a while to complete and you should probably increase your Gr
memory settings by giving the GRADLE_OPTS environment variable a value like

```
export GRADLE_OPTS="-Xmx512m -XX:MaxPermSize=384m"
```

Fortunately, you can reduce the overall build time with a couple of useful options. The first allows yo
specify the location of the Grails source to use:

```
./gradlew -Dgrails.home=/home/user/projects/grails-core docs
```

The Grails source is required because the guide links to its API documentation and the build needs to en it's generated. If you don't specify a `grails.home` property, then the build will fetch the Grails source download of 10s of megabytes. It must then compile the Grails source which can take a while too.

Additionally you can create a local.properties file with this variable set:

```
grails.home=/home/user/projects/grails-core
```

or

```
grails.home=../grails-core
```

The other useful option allows you to disable the generation of the API documentation, since you only nee do it once:

```
./gradlew -Ddisable.groovydocs=true docs
```

Again, this can save a significant amount of time and memory.

The main English user guide is generated in the `build/docs` directory, with the `guide` sub-direc containing the user guide part and the `ref` folder containing the reference material. To view the user gu simply open `build/docs/index.html`.

Publishing

The publishing system for the user guide is the same as [the one for Grails projects](). You write your chap and sections in the gdoc wiki format which is then converted to HTML for the final guide. Each chapter top-level gdoc file in the `src/<lang>/guide` directory. Sections and sub-sections then go into direct with the same name as the chapter gdoc but without the suffix.

The structure of the user guide is defined in the `src/<lang>/guide/toc.yml` file, which is a YA file. This file also defines the (language-specific) section titles. If you add or remove a gdoc file, you i update the TOC as well!

The `src/<lang>/ref` directory contains the source for the reference sidebar. Each directory is the nam a category, which also appears in the docs. Hence the directories need different names for the diffe languages. Inside the directories go the gdoc files, whose names match the names of the methods, comma properties or whatever that the files describe.

Translations

This project can host multiple translations of the user guide, with `src/en` being the main one. To add and one, simply create a new language directory under `src` and copy into it all the files under `src/en`. The b will take care of the rest.

Once you have a copy of the original guide, you can use the `{hidden}` macro to wrap the English text you have replaced, rather than remove it. This makes it easier to compare changes to the English guide aga your translation. For example:

```
{hidden}
When you create a Grails application with the [create-app|commandLine] command,
Grails doesn't automatically create an Ant build.xml file but you can generate
one with the [integrate-with|commandLine] command:
{hidden}

Quando crias uma aplicação Grails com o comando [create-app|commandLine], Grails
não cria automaticamente um ficheiro de construção Ant build.xml mas podes gerar
um com o comando [integrate-with|commandLine]:
```

Because the English text remains in your gdoc files, `diff` will show differences on the English lines. You then use the output of `diff` to see which bits of your translation need updating. On top of that, {hidden} macro ensures that the text inside it is not displayed in the browser, although you can display adding this URL as a bookmark: `javascript:toggleHidden();` (requires you to build the user g with Grails 2.0 M2 or later).

Even better, you can use the `left_to_do.groovy` script in the root of the project to see what still n translating. You run it like so:

```
./left_to_do.groovy es
```

This will then print out a recursive diff of the given translation against the reference English user gu Anything in {hidden} blocks that hasn't changed since being translated will *not* appear in the diff outpu other words, all you will see is content that hasn't been translated yet and content that has changed since it translated. Note that {code} blocks are ignored, so you *don't* need to include them inside {hidd macros.

To provide translations for the headers, such as the user guide title and subtitle, just add language spe entries in the 'resources/doc.properties' file like so:

```
es.title=El Grails Framework
es.subtitle=...
```

For each language translation, properties beginning `<lang>.` will override the standard ones. In the al example, the user guide title will be El Grails Framework for the Spanish translation. Also, translators ca credited by adding a '<lang>.translators' property:

```
fr.translators=Stéphane Maldini
```

This should be a comma-separated list of names (or the native language equivalent) and it will be displaye a "Translated by" header in the user guide itself.

You can build specific translations very easily using the `publishGuide_*` and `publishPdf_*` tasks. example, to build both the French HTML and PDF user guides, simply execute

```
./gradlew publishPdf_fr
```

Each translation is generated in its own directory, so for example the French guide will end up
`build/docs/fr`. You can then view the translated guide by opening
`build/docs/<lang>/index.html`.

All translations are created as part of the [Hudson CI build for the grails-doc](#) project, so you can easily see
the current state is without having to build the docs yourself.