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, Lari Ho
Version: 2.5.5

Table of Contents

1 Introduction

Java web development as it stands today is dramatically more complicated than it needs to be. Most moder[n]
Java space are over complicated and don't embrace the Don't Repeat Yourself (DRY) principles.

Dynamic frameworks like Rails, Django and TurboGears helped pave the way to a more modern wa[y]
applications. Grails builds on these concepts and dramatically reduces the complexity of building web
platform. What makes it different, however, is that it does so by building on already established Java tec[h]
Hibernate.

Grails is a full stack framework and attempts to solve as many pieces of the web development puzzle thr[ough]
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 Do[main-specific languages]
(DSLs)

This documentation will take you through getting started with Grails and building web applications with th[e]

1.1 What's new in Grails 2.5?

Grails 2.5.x is a small release that brings Grails up-to-date with the latest third-party dependencies includi[ng]

- Groovy 2.4.x
- Spring 4.1.x
- Spring Loaded 1.2.4

In addition Grails 2.5.x includes many small maintenance fixes to the platform.

1.2 What's new in Grails 2.4?

Groovy 2.3

Grails 2.4 comes with Groovy 2.3 which includes many new features and enhancements.

For more information on Groovy 2.3, see the [comprehensive release notes](#).

Spring 4.0

Grails 2.4 comes with Spring 4.0.4 which includes many new features and enhancements. See the [Spring d](#)

Hibernate 4.3

Grails 2.4 now uses Hibernate 4.3.5 by default (Hibernate 3 is still available as an optional install).

Standalone GORM and GSP

GORM and GSP can now be used outside of Grails. See the following guides / examples for more informa

- [Accessing Data with GORM](#)
- [Accessing MongoDB Data with GORM](#)
- [GSP in Spring Boot Example Application](#)

The Asset-Pipeline replaces Resources to serve static assets.

The asset-pipeline provides a new, easier to manage, faster means of managing your JavaScript, CSS
bringing compiled client languages in to the fray as first-class citizens (e.g. CoffeeScript, LESS, SASS).

All your assets should now live in the `grails-app/assets` subfolders. Three folders are made for you

- javascript
- stylesheets
- images

Now, defining manifests are done directly in your JavaScript files, or CSS by using require directives!

```javascript
//= require jquery
//= require_self
//= require file_a
//= require_tree .

console.log('some javascript');
```

Easily add your assets to your GSP files:

```
<asset:javascript src="application.js"/>
<asset:stylesheet href="application.css"/>
<asset:image src="grails_logo.png" height="60" />
```

Enjoy developing with on the fly asset processing, asset compiling on WAR, and much more. See the [docs](#)

Static Compilation

Groovy is a dynamically dispatched, dynamically typed language by default but also has great support for static compilation. See [these notes on Groovy static compilation](#). In general, Grails supports Groovy's st are a lot of special situations which are common in a Grails app which cannot be statically compiled. F marked with `@CompileStatic` contains code which invokes a GORM dynamic finder the code will Groovy compiler cannot verify that the dynamic finder is valid. Grails 2.4 improves on this by allow compiled and still do things like invoke GORM dynamic finders.

The [grails.compiler.GrailsCompileStatic](#) annotation behaves much like the [groovy.transform.Comp](#) provides special handling to recognize Grails specific constructs.

The following controller is marked with `@GrailsCompileStatic`. All of the code that can be st statically compiled. When the compiler encounters code which can not be statically validated, normal compile error. The Grails compiler will allow certain things to be considered valid and dynamically dispat

```groovy
// grails-app/controllers/com/demo/PersonController.groovy
package com.demo

import grails.compiler.GrailsCompileStatic

@GrailsCompileStatic
class PersonController {

def showKids() {
        def kids = Person.findAllByAgeLessThan(16)

// …
    }
}
```

There may be situations where most of the code in a class should be statically compiled but a specific dynamic compilation. See the following example.

```
import grails.compiler.GrailsCompileStatic
import groovy.transform.TypeCheckingMode

@GrailsCompileStatic
class SomeClass {

def update() {
      // this method will be statically compiled
    }

@GrailsCompileStatic(TypeCheckingMode.SKIP)
    def save() {
      // this method will not be statically compiled
    }

def delete() {
      // this method will be statically compiled
    }
}
```

The [grails.compiler.GrailsTypeChecked](#) annotation behaves much like the [groovy.transform.TypeChecked](#) special handling to recognize Grails specific constructs.

See the [static compilation and type checking](#) section for more details.

More Advanced Subqueries in GORM

The support for subqueries has been extended. You can now use `in` with nested subqueries:

```
def results = Person.where {
    firstName in where { age < 18 }.firstName
}.list()
```

Criteria and where queries can be seamlessly mixed:

```
def results = Person.withCriteria {
    notIn "firstName", Person.where { age < 18 }.firstName
}
```

Subqueries can be used with projections:

```
def results = Person.where {
    age > where { age > 18 }.avg('age')
}
```

Correlated queries that span two domain classes can be used:

```
def employees = Employee.where {
    region.continent in ['APAC', "EMEA"]
}.id()

def results = Sale.where {
    employee in employees && total > 100000
}.employee.list()
```

And support for aliases (cross query references) using simple variable declarations has been added to wher

```
def query = Employee.where {
    def em1 = Employee
    exists Sale.where {
        def s1 = Sale
        def em2 = employee
        return em2.id == em1.id
    }.id()
}
def results = query.list()
```

GORM for Hibernate in Unit tests

It is no longer necessary to create integration tests in order to test GORM interactions with Hibernate. Y
HibernateTestMixin:

```
import grails.test.mixin.TestMixin
import grails.test.mixin.gorm.Domain
import grails.test.mixin.hibernate.HibernateTestMixin
import spock.lang.Specification


@Domain(Person)
@TestMixin(HibernateTestMixin)
class PersonSpec extends Specification {

void "Test count people"() {
        expect: "Test execute Hibernate count query"
            Person.count() == 0
            sessionFactory != null
            transactionManager != null
            session != null
    }
}
```

This library dependency is required in grails-app/conf/BuildConfig.groovy for adding support for Hibernat

```
dependencies {
        test "org.grails:grails-datastore-test-support:1.0-grails-2.4"
    }
```

HibernateTestMixin is only supported with hibernate4 plugin versions >= 4.3.5.4 .

```
plugins {
        runtime ':hibernate4:4.3.5.4'
    }
```

Views For Namespaced Controllers

The views for namespaced controllers may now be defined in the grails-app/
name>/<controller name>/ directory. See the Models And Views section for more details.

Improved Programmatic Transactions

Transaction attributes may now be specified when invoking withTransaction.

```
    // the keys in the Map must correspond to properties
    // of org.springframework.transaction.support.DefaultTransactionDefinition

    Account.withTransaction([propagationBehavior: TransactionDefinition.PROPAGATION_R
                             isolationLevel: TransactionDefinition.ISOLATION_REPEATAB
        // …
    }
```

See the **withTransaction** docs for more information.

New Maven Plugin

The Maven plugin has been rewritten to use **Aether for dependency resolution** and can now be used w
Grails 2.4.x without releasing a new version of the plugin.

This means that the Maven plugin version number is no longer tied to the version number of Grails and n
plugin will not come out with each new Grails release. Instead, users can continue to use the 2.4.0 vers
version of Grails going forward.

Unit Testing improvements

There is a Grails "unit testing runtime" that is based on the previous TestMixin based solution. It now
classes and the actual runtime that handles the lifecycle of the Grails unit testing runtime. State of the ru
fields of the TestMixin classes anymore. The Groovy AST transformation behind the TestMixin annotatic
Spock test classes by adding JUnit Rule fields to the class. In the previous solution, Before/BeforeCl
annotations on AST added mix-in methods were used for the integration.

Some of the main features:

- The programming model remains the same for unit testing of Grails applications

- Setup/teardown method ordering is now deterministic because the integration is now using a single
  test runtime uses eventing internally to setup and teardown resources

- There are doWithSpring and doWithConfig callbacks for unit tests - these callback method
  grailsApplication instance in the unit test runtime gets initialized.

- It's possible to register a Spock Mock as a bean to the application context of the Grails unit test runti
  replace a collaborator bean with a mock

- It's possible to reuse a single application context for several test classes and control that so that tests
  required

- The Grails unit testing runtime has an event-based plugin architecture. It's possible to add new tes
  new **test runtime plugin** classes. The test runtime plugin API is due to change. Changes will be made
  the Grails community. The main interfaces of the API are currently documented in the
  TestEventInterceptor and **TestEvent**. Custom test plugins are currently limited since there isn't a s
  available test plugins. It's now possible to add custom test plugins in a static initialization block o
  TestRuntimeFactory.addPluginClass .

See the updated **unit testing chapter** in the manual for more information of the new features like doWithSp

Improved Unit Testing Support For allowedMethods

The allowedMethods property is now respected in unit tests.

```
// grails-app/controllers/com/demo/DemoController.groovypackage com.demo

class DemoController {

static allowedMethods = [save: 'POST', update: 'PUT', delete: 'DELETE']

def save() {
        render 'Save was successful!'
    }
// …
}
```

```
// test/unit/com/demo/DemoControllerSpec.groovy
package com.demo

import grails.test.mixin.TestFor
import spock.lang.Specification
import static javax.servlet.http.HttpServletResponse.*

@TestFor(DemoController)
class DemoControllerSpec extends Specification {

void "test a valid request method"() {
        when:
        request.method = 'POST'
        controller.save()

then:
        response.status == SC_OK
        response.text == 'Save was successful!'
    }

void "test an invalid request method"() {
        when:
        request.method = 'DELETE'
        controller.save()

then:
        response.status == SC_METHOD_NOT_ALLOWED
    }
}
```

1.3 What's new in Grails 2.3?

Improved Dependency Management

The default dependency resolution engine used by Grails has been changed to [Aether](#), the dependency r
Maven. Which engine you use can be configured in `BuildConfig`:

```
grails.project.dependency.resolver = "maven" // or ivy
```

Using Aether dependency resolution in Grails results in the same behavior as when using the Maven build snapshot handling, understanding of custom packaging types and so on.

In addition, the [dependency-report](#) command has been updated to print the dependency graph of the diagnosing dependency resolution failures. See the chapter on [Dependency Resolution](#) for more informatio

Data Binder

Grails 2.3 includes a new data binding mechanism which is more flexible and easier to maintain than previous versions. The new data binder includes numerous enhancements including:

- Custom date formats on a per field basis using [BindingFormat](#)

- User defined data converters using [ValueConverter](#)

- User defined formatted data converters using [BindingFormat](#) and [FormattedValueConverter](#)

- Custom binding on a per class basis using [BindUsing](#)

- Custom binding on a per field basis using [BindUsing](#)

- By default all blank and empty Strings will be converted to null during data binding (configurable)

See the [Data Binding](#) section for details.

The legacy data binder may be used by assigning `true` to the `grails.databinding.useSpri` `grails-app/conf/Config.groovy`. Note that the legacy binder does not support any of the new new data binder.

Binding Request Body To Command Objects

If a request is made to a controller action which accepts a command object and the request includes a body and used to do data binding to the command object. This simplifies use cases where a request includes a example) that can be bound to a command object. See the [Command Objects](#) documentation for more deta

Domain Classes As Command Objects

When a domain class is used as a command object and there is an `id` request parameter, the framework w: the domain class from the database using the `id` request parameter. See the [Command Objects](#) documentat

Forked Execution

All major commands can now be forked into a separate JVM, thus isolating the build path from the run execution can be controlled via the `BuildConfig`:

```
grails.project.fork = [
    test: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256, daemon:true]
settings for the test-app JVM
    run: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256], // configure
run-app JVM
    war: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256], // configure
run-war JVM
    console: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256]// configu
the Console UI JVM
]
```

See the documentation on [Forked Mode](#) for more information.

Test Runner Daemon

To speed up testing when using forked execution a new daemon will start-up in the background to run tes
mode. You can restart the daemon with the `restart-daemon` command from interactive mode:

```
$ grails> restart-daemon
```

Server-Side REST Improvements

Grails' [REST support](#) has been significantly improved with the addition of the following features:

- Rich [REST URL Mapping support](#) with supports for resource mappings, singular resource map
  versioning and more

- New extensible response rendering and binding APIs

- Support for HAL, Atom and Hypermedia (HATEAOS)

- Scaffolding for REST controllers

See the [user guide](#) for more information.

New Scaffolding 2.0 Plugin

Grails' Scaffolding feature has been split into a [separate plugin](#). Version 2.0 of the plugin includes supp
controllers, Async controllers, and Spock unit tests.

URL Mappings May Specify A Redirect

URL Mappings may now specify that a redirect should be triggered when the mapping matches an incomir

```
class UrlMappings {
    static mappings = {
        "/viewBooks"(redirect: '/books/list')
        "/viewAuthors"(redirect: [controller: 'author', action: 'list'])
        "/viewPublishers"(redirect: [controller: 'publisher', action: 'list', per

// …
    }
}
```

See the [user guide](#) for more information.

Async support

Grails 2.3 features new [Asynchronous Programming APIs](#) that allow for asynchronous processing seamlessly with GORM. Example:

```
import static grails.async.Promises.*
…
def index() {
    tasks books: Book.async.list(),
          totalBooks: Book.async.count(),
          otherValue: {
              // do hard work
          }
}
```

See [the documentation](#) for further information.

Encoding / Escaping Improvements

Grails 2.3 features dedicated support for [Cross Site Scripting (XSS) prevention](#), including :

- Defaulting to HTML escaping all GSP expressions and scriptlets

- Context sensitive encoding switching for tags

- Double encoding prevention

- Optional automatic encoding of all data in a GSP page not considered safe

See the documentation on [Cross Site Scripting (XSS) prevention](#) for more information.

Hibernate 3 and 4 support

The GORM for Hibernate 3 support for Grails has been extracted into a separate project, allowing new sup separate plugin.

Controller Exception Handling

Controllers may define exception handler methods which will automatically be invoked any time an action an exception.

```groovy
// grails-app/controllers/demo/DemoController.groovy
package demo

class DemoController {

def someAction() {
      // do some work
    }

def handleSQLException(SQLException e) {
      render 'A SQLException Was Handled'
    }

def handleBatchUpdateException(BatchUpdateException e) {
      redirect controller: 'logging', action: 'batchProblem'
    }

def handleNumberFormatException(NumberFormatException nfe) {
      [problemDescription: 'A Number Was Invalid']
    }
}
```

See the controller exception handling docs for more information.

Namespaced Controllers

Controllers may now be defined in a namespace which allows for multiple controllers to be defined with th packages.

```groovy
// grails-app/controllers/com/app/reporting/AdminController.groovy
package com.app.reporting

class AdminController {

static namespace = 'reports'

// …
}
```

21

```groovy
// grails-app/controllers/com/app/security/AdminController.groovy
package com.app.security

class AdminController {

static namespace = 'users'

// …
}
```

```groovy
// grails-app/conf/UrlMappings.groovy
class UrlMappings {

static mappings = {
        '/userAdmin' {
            controller = 'admin'
            namespace = 'users'
        }

'/reportAdmin' {
            controller = 'admin'
            namespace = 'reports'
        }

"/$namespace/$controller/$action?"()
    }
}
```

```groovy
<g:link controller="admin" namespace="reports">Click For Report Admin</g:link>
<g:link controller="admin" namespace="users">Click For User Admin</g:link>
```

See the namespaced controllers docs for more information.

Command Line

The `create-app` command will now by default generate the command line grailsw wrapper for newly (
`--skip-wrapper` switch may be used to prevent the wrapper from being generated.

```
grails create-app appname --skip-wrapper
```

1.4 What's new in Grails 2.2?

Namespace Support

Grails 2.2 includes improved support for managing naming conflicts between artifacts provided by an appl

Bean names for Service artifacts provided by a plugin are now prefixed with the plugin name. For exam
com.publishing.AuthorService is provided by a plugin named PublishingUtilities and
com.bookutils.AuthorService is provided by a plugin named BookUtilities, the bean nam
be publishingUtilitiesAuthorService and bookUtilitiesAuthorService respective
Service that does not have a name which conflicts with any other Service, then a bean alias will automati
not contain the prefix and the alias will refer to the bean referenced by the prefixed name. Service artifacts
application will have no prefix added to the relevant bean name. See the [dependency injection and services

Domain classes provided by a plugin will have their default database table name prefixed with t
grails.gorm.table.prefix.enabled config property is set to true. For example, if the Pul
plugin provides a domain class named Book, the default table name for that dor
PUBLISHING_UTILITIES_BOOK if the grails.gorm.table.prefix.enabled config propert

URL Mappings may now include a plugin attribute to indicate that the controller referenced in the n
particular plugin.

```groovy
static mappings = {
// requests to /bookAuthors will be handled by the
    // AuthorController provided by the BookUtilities plugin
    "/bookAuthors" {
        controller = 'author'
        plugin = 'bookUtilities'
    }
// requests to /publishingAuthors will be handled by the
    // AuthorController provided by the Publishing plugin
    "/publishingAuthors" {
        controller = 'author'
        plugin = 'publishing'
    }
}
```

See the [namespaced controllers](#) docs for more information.

Controller methods and GSP Tags which accept a controller name as a parameter now support an optional
the controller is provided by a specific plugin.

```
<g:link controller="user" plugin="springSecurity">Manage Users</g:link>
```

```
class DemoController {
    def index() {
        redirect controller: 'user', action: 'list', plugin: 'springSecurity'
    }
}
```

Forked Tomcat Execution

Grails 2.2 supports forked JVM execution of the Tomcat container in development mode. This has several

- Reduced memory consumption, since the Grails build system can exit

- Isolation of the build classpath from the runtime classpath

- The ability to deploy other Grails/Spring applications in parallel without conflicting dependencies

See the [documentation](#) on using forked mode for more information.

SQL Projections In Criteria Queries

Grails 2.2 adds new functionality to criteria queries to provide access to Hibernate's SQL projection API.

```
// Use SQL projections to retrieve the perimeter and area of all of the Box insta
def c = Box.createCriteria()

def results = c.list {
    projections {
      sqlProjection '(2 * (width + height)) as perimeter, (width * height) as are
'area'], [INTEGER, INTEGER]
    }
}
```

See the [Criteria](#) section for more information.

Groovy 2

Grails 2.2 ships with Groovy 2.0, which has a [bunch of new features](#) itself.

1.5 What's new in Grails 2.1?

Maven Improvements / Multi Module Build Support

Grails' Maven support has been improved in a number of significant ways. Firstly it is now possible to spe
`pom.xml` file:

```
<dependency>
    <groupId>org.grails.plugins</groupId>
    <artifactId>hibernate</artifactId>
    <version>2.1.0</version>
    <type>zip</type>
    <scope>compile</scope>
</dependency>
```

The Maven plugin now resolves plugins as well as jar dependencies (previously jar dependencies were
plugins by Ivy). Ivy is completely disabled leaving all dependency resolution up to Maven ensuring
expected.

There is also a new Grails `create-multi-project-build` script which features initial support for [
a future release). This script can be run from a parent directory containing Grails applications and plugi
Maven multi-module build.

Enabling Maven in a project has been made easier with the inclusion of the `create-pom` command:

```
grails create-app myapp
cd myapp
grails create-pom com.mycompany
mvn package
```

To create a multi-module Maven build follow these steps:

```
grails create-app myapp
grails create-plugin plugin-a
grails create-plugin plugin-b
grails create-multi-project-build com.mycompany:parent:1.0-SNAPSHOT
mvn install
```

Grails Wrapper

The Grails Wrapper allows a Grails application to build without having to install Grails and config environment variable. The wrapper includes a small shell script and a couple of small bootstrap jar file checked in to source code control along with the rest of the project. The first time the wrapper is execu configure a Grails installation. This wrapper makes it more simple to setup a development environment, upgrades to future versions of Grails. When the application is upgraded to the next version of Grails, the checked in to the source code control system and the next time developers update their workspace and r automatically be using the correct version of Grails.

See the **Wrapper Documentation** for more details.

Debug Option

The `grails` command now supports a `-debug` option which will startup the remote debug agent. provided by the `grails-debug` command. `grails-debug` is still available but is deprecated and future release.

```
grails -debug run-app
```

Grails Command Aliases

The `alias` command may be used to define aliases for grails commands.

The following command creates an alias named `rit` (short for "run integration tests"):

```
grails alias rit test-app integration:
```

See the **alias** docs for more info.

Cache Plugin

Grails 2.1 installs the **cache plugin** by default. This plugin provides powerful and easy to use cache functio plugins. The main plugin provides basic map backed caching support. For more robust caching options o plugins should be installed and configured. See the **cache-redis docs** and the **cache-ehcache docs** for details

See **the main plugin documentation** for details on how to configure and use the plugin.

New GORM Methods

In Grails 2.1.1 domain classes now have static methods named `first` and `last` to retrieve the first ar datastore. See the **first** and **last** documentation for details.
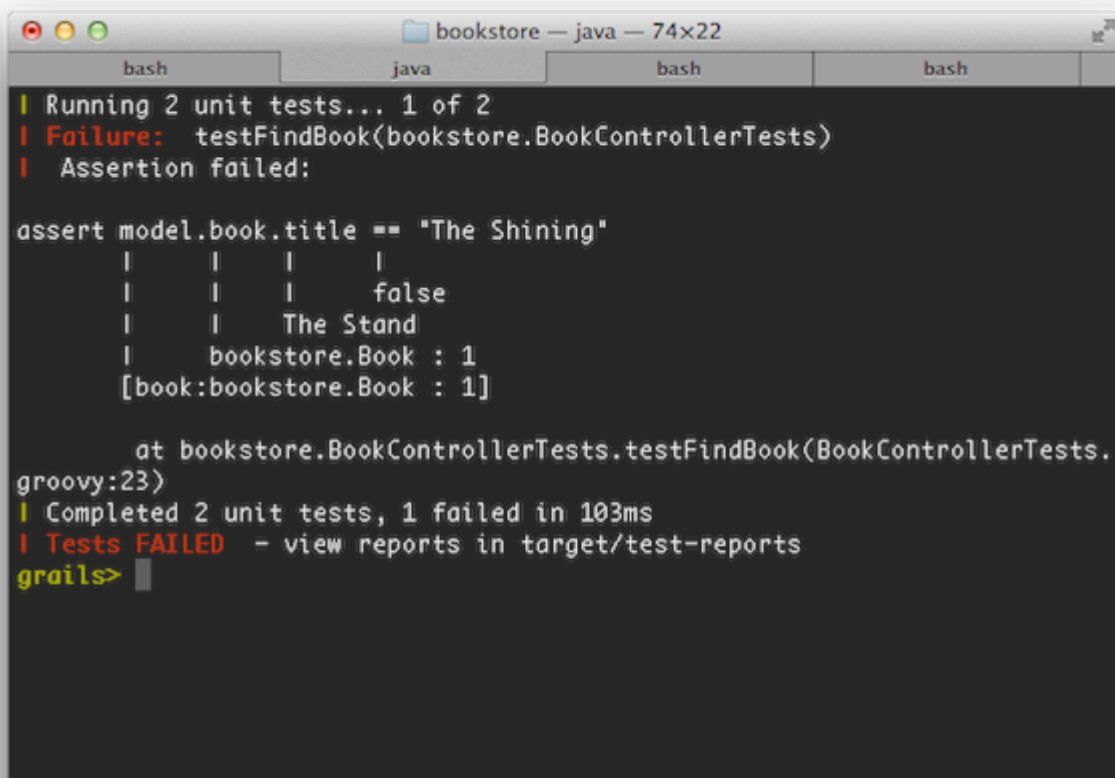
1.6 What's new in Grails 2.0?

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

1.6.1 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 example 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 the in This means that while in previous versions of Grails the war command produced many lines of output, in output is produced:

In addition simply typing 'grails' at the command line activates the new interactive mode which fe command history and keeps the JVM running to ensure commands execute much quicker than otherwise

28

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

Reloading Agent

Grails 2.0 reloading mechanism no longer uses class loaders, but instead uses a JVM agent to reload ch
results in greatly improved reliability when reloading changes and also ensures that the class files stored
with the class files loaded in memory, which reduces the need to run the clean command.
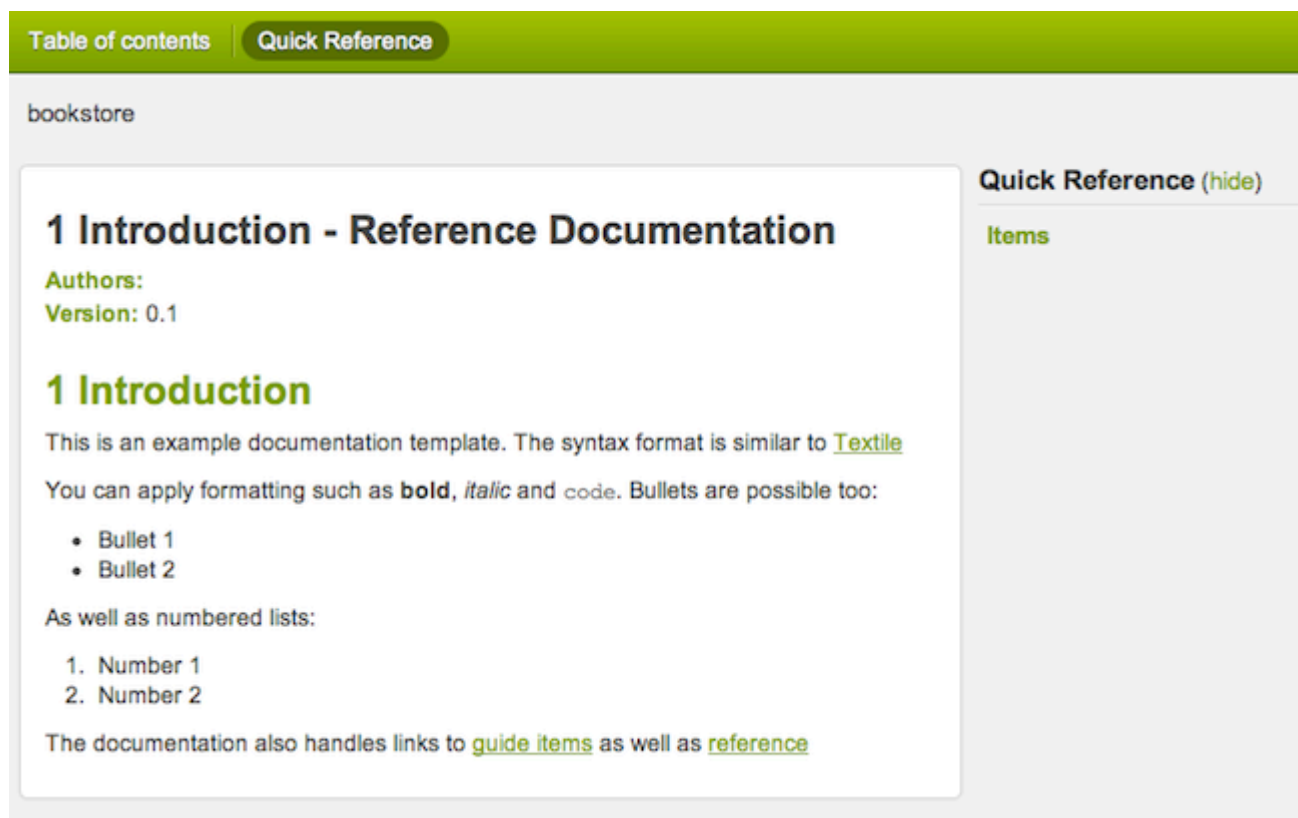
New Test Report and Documentation Templates

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

In addition, the Grails documentation engine has received a facelift with a new template for presentin
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. Although cc made it difficult to restructure your user guide by inserting new chapters and sections. In addition, a renaming of section titles resulted in breaking changes to the URLs.

You can now use logical names for your gdoc files and define the structure and section titles in a YAML described in the section on the [documentation engine](). The logical names appear in the URLs, so as long a your URLs will always remain the same no matter how much 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 sta displays problem areas in your code:

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 console in devel
/dbconsole) so that the in-memory database can be easily queried from the browser:

Plugin Usage Tracking

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

This will help drive the roadmap and increase support of key plugins while reducing the need to supp
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 agai BuildConfig.groovy.

- Plugins dependencies now appear in the dependency report generated by grails dependency-r

- Plugins published with the release plugin now publish their transitive plugin dependencies in the g 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 by

```
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
}
```

1.6.2 Core Features

Binary Plugins

Grails plugins can now be packaged as JAR files and published to standard maven repositories. This even resources (with resources plugin 1.0.1). See the section on [Binary plugins](#) for more 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 wh Grails environment called "production", a Spring profile of "production" is activated so that you configuration APIs to configure beans for a specific profile.

1.6.3 Web Features

Controller Actions as Methods

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

```
// 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 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 parameters

```
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 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 Grails appli the context of a controller. For example if you need to generate links in a service or an asynchronous ba 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 scop as in a scheduled job or web service. The `PageRenderer` class features a very similar API to the `rend` controllers:

```
grails.gsp.PageRenderer groovyPageRenderer

void welcomeUser(User user) {
    def contents = groovyPageRenderer.render(view:"/emails/welcomeLetter", model:
    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 wl filter should be applied.
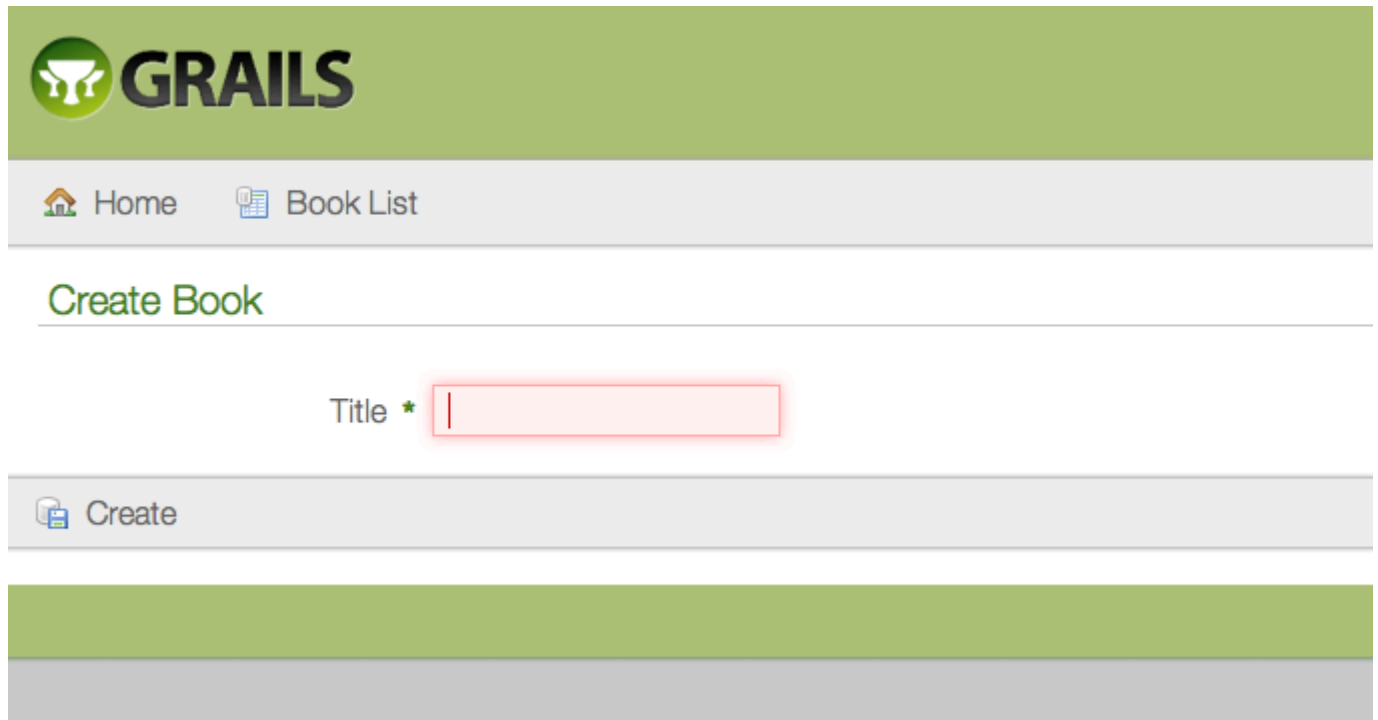
```
filter1(actionExclude: 'log*') {
    before = {
        // …
    }
}
filter2(controllerExclude: 'auth') {
    before = {
        // …
    }
}

filter3(uriExclude: '/secure*') {
    before = {
        // …
    }
}
```

Performance Improvements

Performance of GSP page rendering has once again been improved by optimizing the GSP compiler to 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 backwards plugin is available. Refer to the [documentation](#) on the Prototype plugin for 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 for ac addNumbers in a controller named MathHelperController would be something like /mathF Grails allows for the customization of this pattern and provides an implementation which replaces the cam hyphenated convention that would support URLs like /math-helper/add-numbers. To enable hy value of "hyphenated" to the grails.web.url.converter property in grails-app/conf/Conf

```
// 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 : that class to the Spring application context with the bean name of grails.web.UrlConverter.BE/ a bean in the context with that name, it will be used as the default converter and there is no need t 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
    }
}
```

```
// grails-app/conf/spring/resources.groovy

beans = {
    "${grails.web.UrlConverter.BEAN_NAME}"(com.myapplication.MyUrlConverterImpl)
}
```

Web Flow input and output

It is now possible to provide input arguments when calling a subflow. Flows can also return output val calling flow.

1.6.4 Persistence Features

The GORM API

The GORM API has been formalized into a set of classes (GormStaticApi, Gorm
GormValidationApi) that get statically wired into every domain class at the byte code level. T
completion for IDEs, better integration with Java and the potential for more GORM implementations for o

Detached Criteria and Where Queries

Grails 2.0 features support for **DetachedCriteria** which are criteria queries that are not associated with a
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 DS
greatly reduce the complexity of criteria queries:

```
def query = Person.where {
    (lastName != "Simpson" && firstName != "Fred") || (firstName == "Bart" && age
}
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 and findO
which behave just like findWhere and findBy methods except that they should never return null. If a mat
found in the database then a new instance is created, populated with values represented in the query param
case of findOrSaveWhere and findOrSaveBy, the instance is saved before being returned.

```
def book = Book.findOrCreateWhere(author: 'Douglas Adams', title: "The Hitchhiker
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 linkin

```groovy
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 more domain uses by default:

```groovy
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 a any regular GORM method:

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

For more information see the section on Multiple Data Sources in the user guide.

Database Migrations

A new [database migration plugin](#) has been designed and built for Grails 2.0 allowing you to apply mig rollback changes and diff your domain model with the current state of the database.

Database Reverse Engineering

A new [database reverse engineering](#) plugin has been designed and built for Grails 2.0 that allows you to from an existing database schema.

Hibernate 3.6

Grails 2.0 is now built on Hibernate 3.6

Bag Collections
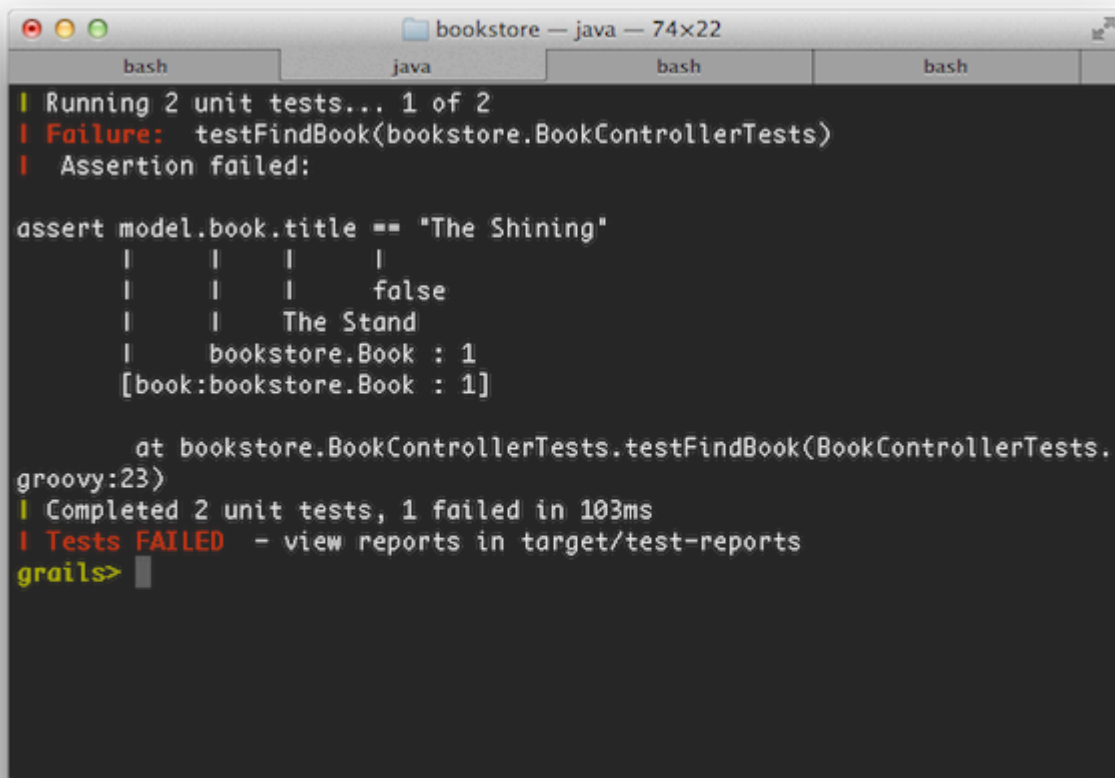
You can now use Hibernate [Bag](#)s for mapped collections to avoid the memory and performance issues of to enforce `Set` uniqueness or `List` order.

For more information see the section on [Sets, Lists and Maps](#) in the user guide.

1.6.5 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
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 A
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 unit

Unit Test Scaffolding

A unit test is now generated for scaffolded controllers

2 Getting Started

2.1 Installation Requirements

Before installing Grails you will need as a minimum a Java Development Kit (JDK) installed version 1.6 appropriate JDK for your operating system, run the installer, and then set up an environment variable call to the location of this installation. If you're unsure how to do this, we recommend the video in grailsexample.net:

- Windows

- Linux

- Mac OS X

These will show you how to install Grails too, not just the JDK.

> ⚠ A JDK is required in your Grails development environment. A JRE is not sufficient.

On some platforms (for example OS X) the Java installation is automatically detected. However in man manually configure the location of Java. For example:

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

if you're using bash or another variant of the Bourne Shell.

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 steps

- Download a binary distribution of Grails and extract the resulting zip file to a location of your choice

- Set the GRAILS_HOME environment variable to the location where you extracted the zip

  - On Unix/Linux based systems this is typically a matter of adding something like `GRAILS_HOME=/path/to/grails` to your profile

  - On Windows this is typically a matter of setting an environment `Computer/Advanced/Environment Variables`

- Then add the `bin` directory to your `PATH` variable:

  - On Unix/Linux based systems this can be done by adding `export PATH="$PATH:$GRAII` profile

  - On Windows this is done by modifying the `Path` environment `Computer/Advanced/Environment Variables`

If Grails is working correctly you should now be able to type `grails -version` in the terminal windo
to this:

```
Grails version: 2.0.0
```

2.3 Creating an Application

To create a Grails application you first need to familiarize yourself with the usage of the `grails` comm
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 directory i

```
cd helloworld
```

2.4 A Hello World Example

Let's now take the new project and turn it into the classic "Hello world!" example. First, change into the "h
just created and start the Grails interactive console:

```
$ cd helloworld
$ grails
```

You should see a prompt that looks like this:

```
Graeme-Rochers-iMac:helloworld graemerocher$ grails
| Enter a script name to run. Use TAB for completion:
grails>
ctory created in the previous section and activate interactive mode:
```

What we want is a simple page that just prints the message "Hello World!" to the browser. In Grails, w
page you just create a new controller action for it. Since we don't yet have a controller, let's cre
create-controller command:

```
grails> create-controller hello
```

Don't forget that in the interactive console, we have auto-completion on command names. So you can t
<tab> to get a list of all create-* commands. Type a few more letters of the command name and then <

The above command will create a new controller in the grails-app/controllers/hellow
HelloController.groovy. Why the extra helloworld directory? Because in Java land, it's stron
classes are placed into packages, so Grails defaults to the application name if you don't provide one.
create-controller provides more detail on this.

We now have a controller so let's add an action to generate the "Hello World!" page. The code looks like t

```groovy
package helloworld
class HelloController {
def index() {
        render "Hello World!"
    }
}
```

The action is simply a method. In this particular case, it calls a special method provided by Grails to render

Job done. To see your application in action, you just need to start up a server with another command called

```
grails> run-app
```

This will start an embedded server on port 8080 that hosts your application. You should now be able to a
the URL http://localhost:8080/helloworld/ - try it!

> ⚠ If you see the error "Server failed to start for port 8080: Address already in use", then it me
> server is running on that port. You can easily work around this by running your server on a di
> using `-Dserver.port=9090 run-app`. '9090' is just an example: you can pretty m
> anything within the range 1024 to 49151.

The result will look something like this:
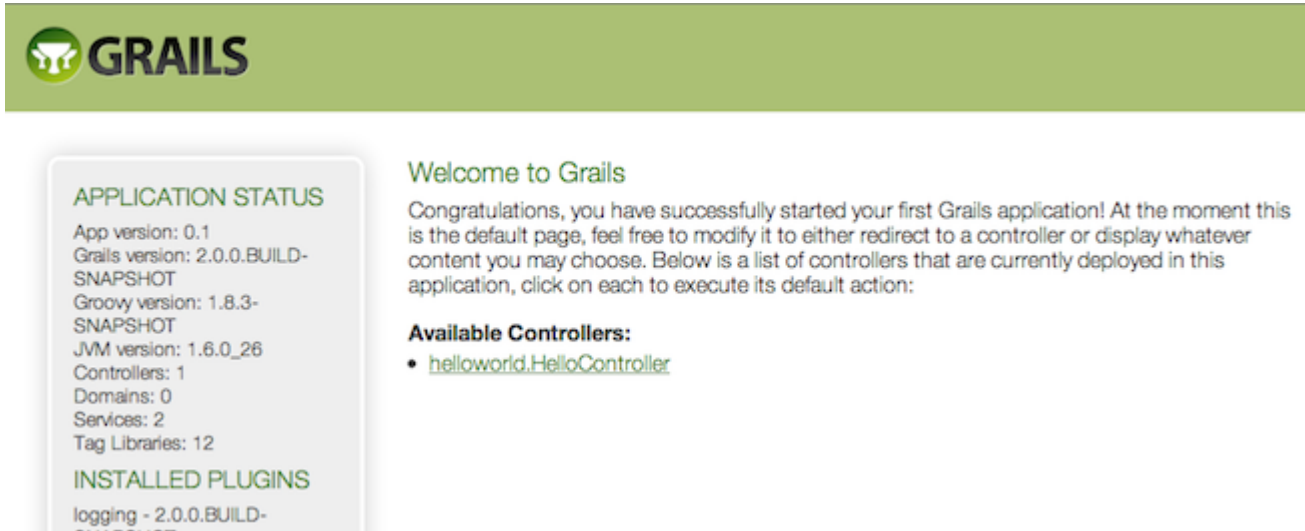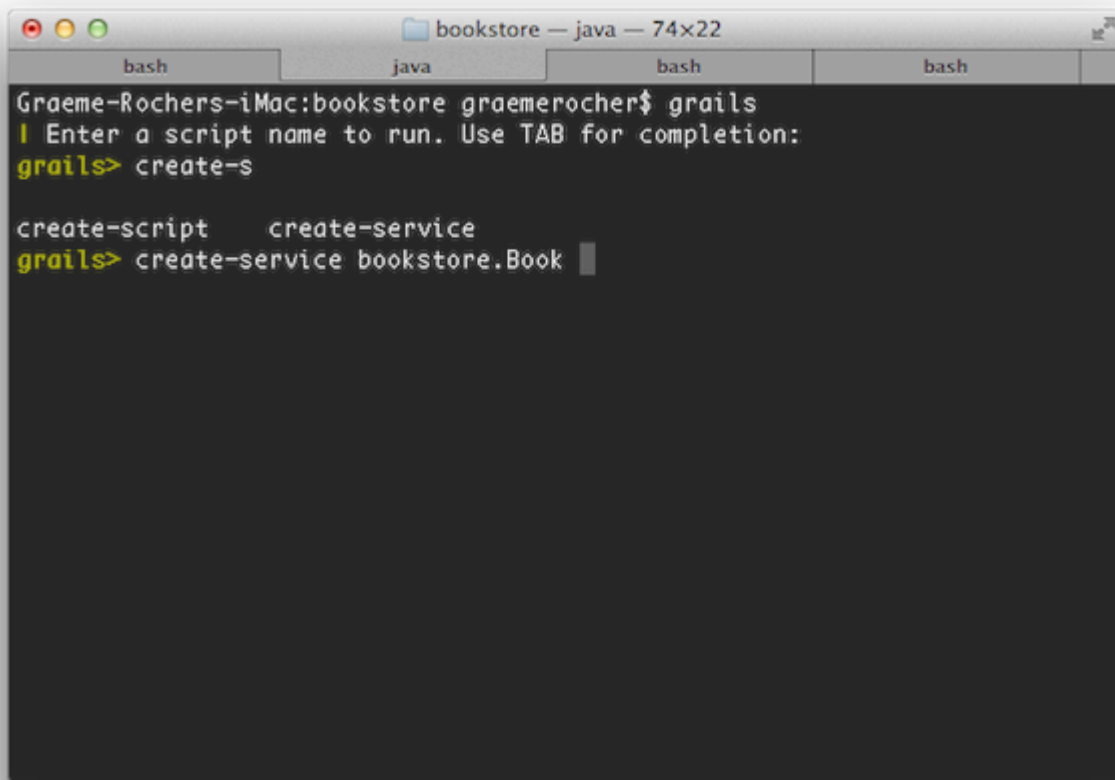
This is the Grails intro page which is rendered by the `grails-app/view/index.gsp` file. It dete
controllers and provides links to them. You can click on the "HelloController" link to see our custom
"Hello World!". Voila! You have your first working Grails application.

One final thing: a controller can contain many actions, each of which corresponds to a different page (igno
Each page is accessible via a unique URL that is composed from the controller name
/<appname>/<controller>/<action>. This means you can access the Hello World page via /helloworld/he
the controller name (remove the 'Controller' suffix from the class name and lower-case the first letter)
name. But you can also access the page via the same URL without the action name: this is because 'index'
the end of the controllers and actions section of the user guide to find out more on default actions.

2.5 Using Interactive Mode

Grails 2.0 features an interactive mode which makes command execution faster since the JVM doesn't hav
command. To use interactive mode simple type 'grails' from the root of any projects and use TAB co
available commands. See the screenshot below for an example:

For more information on the capabilities of interactive mode refer to the section on [Interactive Mode](#) in the

2.6 Getting Set Up in an IDE

IntelliJ IDEA

[IntelliJ IDEA](#) and the [JetGroovy](#) plugin offer good support for Groovy and Grails developers. Refer to th [Grails](#) support on the JetBrains website for a feature overview.

IntelliJ IDEA comes in two flavours; the open source "Community Edition" and the commercial "Ultim support for Groovy, but only Ultimate Edition offers Grails support.

With Ultimate Edition, there is no need to use the `grails integrate-with --intellij` comm understands Grails projects natively. Just open the project with `File -> New Project -> Cre existing sources`.

You can still use Community Edition for Grails development, but you will miss out on all the Grails specif classpath management, GSP editor and quick access to Grails commands. To integrate Grails with Co following command to generate appropriate project files:

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

Eclipse

We recommend that users of [Eclipse](#) looking to develop Grails application take a look at [Groovy/Grails](#) built in support for Grails including automatic classpath management, a GSP editor and quick access to G [STS Integration](#) page for an overview.

NetBeans

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

TextMate

Since Grails' focus is on simplicity it is often possible to utilize more simple editors and [TextMate](#) on Groovy/Grails bundle available from the [TextMate 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 follo root of your project:

```
mate .
```

2.7 Convention over Configuration

Grails uses "convention over configuration" to configure itself. This typically means that the name and instead of explicit configuration, hence you need to familiarize yourself with the directory structure provid

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](#).

2.8 Running an Application

Grails applications can be run with the built in Tomcat server using the [run-app](#) command which will load 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 'stop-app' to shutdown.
| Downloading: plugins-list.xml
grails> stop-app
| Stopping Grails server
grails> run-app
| Server running. Browse to http://localhost:8080/helloworld
| Application loaded in interactive mode. Type 'stop-app' to shutdown.
| Downloading: plugins-list.xml
```

More information on the <u>run-app</u> command can be found in the reference guide.

2.9 Testing an Application

The create-* commands in Grails automatically create unit or integration tests for you withi
test/integration directory. It is of course up to you to populate these tests with valid test logic, info
found in the section on <u>Testing</u>.

To execute tests you run the <u>test-app</u> command as follows:

```
grails test-app
```

2.10 Deploying an Application

Grails applications are deployed as Web Application Archives (WAR files), and Grails includes the <u>war</u>
this task:

```
grails war
```

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

Unlike most scripts which default to the development environment unless overridden, the war
production environment by default. You can override this like any script by specifying the environmen

```
grails dev war
```

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

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

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

2.11 Supported Java EE Containers

Grails runs on any container that supports Servlet 2.5 and above and is known to work on the following sp

- Tomcat 7

- Tomcat 6

- SpringSource tc Server

- Eclipse Virgo

- GlassFish 3

- GlassFish 2

- Resin 4

- Resin 3

- JBoss 6

- JBoss 5

- Jetty 8

- Jetty 7

- Jetty 6

- Oracle Weblogic 10.3

- Oracle Weblogic 10

- Oracle Weblogic 9

- IBM WebSphere 8.5

- IBM WebSphere 8.0

- IBM WebSphere 7.0

- IBM WebSphere 6.1

> ⚠ It's required to set "-Xverify:none" in "Application servers > server > Process Definition > J
> Machine > Generic JVM arguments" for older versions of WebSphere. This is no longer
> WebSphere version 8 or newer.

Some containers have bugs however, which in most cases can be worked around. A <u>list of known deploy</u>
on the Grails wiki.

2.12 Creating Artefacts

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

> ⚠ These are just for your convenience and you can just as easily use an IDE or your favourite te

For example to create the basis of an application you typically need a <u>domain model</u>:

```
grails create-app helloworld
cd helloworld
grails create-domain-class book
```

This will result in the creation of a domain class at `grails-app/domain/helloworld/Book.gro`

```
package helloworld

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 <u>interactive</u> mode.

2.13 Generating an Application

To get started quickly with Grails it is often useful to use a feature called <u>Scaffolding</u> to generate the skele
do this use one of the `generate-*` commands such as <u>generate-all</u>, which will generate a <u>controller</u> (
associated <u>views</u>:

```
grails generate-all helloworld.Book
```

3 Upgrading from Grails 2.2

A number of changes need to be considered when upgrading your application from Grails 2.2, some of quick list with more detail on each item following after:

- New improved data binding (no Spring property editors)
- Much improved XSS prevention with default HTML encoding
- A new dependency resolution engine
- Must be online to fetch Grails dependencies
- Grails core dependencies rearranged
- Tomcat and Hibernate plugins independently versioned now (breaking!)
- Scaffolding is now a separate plugin
- Spock included by default
- Dependency injection does not work in integration tests by default
- Forked execution for tests
- Reloading in `run-app` won't work by default on upgraded apps
- `grails-debug` doesn't work for forked execution

New Data Binder

There is a new data binding mechanism written from the ground up to meet Grails' needs. If you wish to data binding then you must set the `grails.databinding.useSpringBinder` pro `grails-app/conf/Config.groovy`

Encoding / Escaping (XSS) Changes

Grails 2.3 includes new features to help prevent XSS attacks. These are enabled by default for new applications will require manual intervention. See the section on [Cross Site Scripting (XSS) prevention](#) configure XSS prevention.

Dependency Resolution changes

Although dependency resolution using Ivy is still supported, the default for Grails 2.3 is to use Aether and be improved upon going forward. You may wish to consider using Aether instead for your existing ap following in `grails-app/conf/BuildConfig.groovy`:

```
grails.project.dependency.resolver = "maven" // or ivy
```

If you need to authenticate to a maven repository, you will want to change the definition of that repository

```
mavenRepo("http://artifactory.mycompany.com/repo") {
    authentication(username: "myusername", password: "secret")
}
```

Dependency Metadata Changes

In addition, the POM and dependency metadata for Grails 2.3 has been re-arranged and cleaned up so that
are specified for an application and all other dependencies are inherited transitively. This has implications
example, Ehcache is now a transitive dependency of the Hibernate plugin, whilst before it was a dir
compilation error related to Ehcache, it is most likely that you don't have the Hibernate plugin installed an
the Ehcache dependency:

```
compile "net.sf.ehcache:ehcache:2.8.1"
```

In addition, excludes may no longer work and may need adjusting when upgrading due to how the metada
dependency-report to see the new dependency metadata and make adjustments accordingly.

A common error that may occur when upgrading is:

```
| Configuring classpath
:: problems summary ::
:::: WARNINGS
    ::::::::::::::::::::::::::::::::::::::::::::::
    ::          UNRESOLVED DEPENDENCIES         ::
    ::::::::::::::::::::::::::::::::::::::::::::::
    :: org.springframework#spring-test;3.2.2.RELEASE: configuration not found in
org.springframework#spring-test;3.2.2.RELEASE: 'compile'. It was required from
org.grails#grails-plugin-testing;2.3.0.BUILD-SNAPSHOT compile
    ::::::::::::::::::::::::::::::::::::::::::::::
```

This is caused by a plugin that depends on an old version of spring-test (for example the Mail plu
grails dependency-report and search for plugins that have a transitive dependency on spring-
For example:

```
plugins {
  compile ':mail:1.0', {
    excludes 'spring-test'
  }
}
```

However, longer term to solve problems like this we recommend that users move away from Ivy an
dependency resolution:

```
grails.project.dependency.resolver="maven"
```

No initial offline mode with Aether

Aether does not support resolving dependencies from a flat file system. This means that the jars
GRAILS_HOME/lib are not used for the first resolve, but instead the jars are obtained from Maven cent
obtained from Maven central then Aether operates fine offline.

If however you do not have the necessary jars in your local Maven repository, then the only way to g
enable Ivy via BuildConfig (see above).

Changes to Core plugin versioning schemes and the Upgrade command

Core plugins like tomcat and hibernate are no longer versioned the same as the Grails version, in
according to the Tomcat and Hibernate version they target. If you are upgrading from Grails 2.2 you ne
the correct Tomcat and Hibernate plugins in BuildConfig. The upgrade command will not do this fo

```
plugins {
        // plugins for the build system only
        build ':tomcat:7.0.42'
// plugins needed at runtime but not for compilation
        runtime ':hibernate:3.6.10.2'
    }
```

Note that the upgrade command will be deprecated in 2.3 and replaced with
use-current-grails-version, which will make no attempts to automatically upgrade Grails appli

Scaffolding moved to a plugin and rewritten

If you have dynamically scaffolded controllers in your application then you will need to configure [Scaffolding plugin](#) in BuildConfig:

```
plugins {
  compile ':scaffolding:1.0.0'
}
```

By default for new applications the 2.0 version of the scaffolding plugin is used, which is not backwards co

Spock included by default

You no longer need to add the Spock plugin to your projects. Simply create Spock specifications as befo
unit tests. In fact, don't install the Spock plugin, otherwise your specifications will run twice and potenti
that the spock test type no longer exists. Specifications and JUnit tests run as the same type now.

Dependency Injection for Integration Tests

In order to support alternate JUnit4 test runners, Grails 2.3 no longer uses a special test runner to run t
should no longer extend GroovyTestCase.

This change requires that any JUnit integration tests that require dependency injection now need to be anno

```
@TestMixin(IntegrationTestMixin)
```

For Spock integration tests, extending IntegrationSpec also works.

Forked Execution for Testing

Tests are now by default executed in a forked JVM (although this can be disabled). One implication of
slower to execute when using:

```
grails test-app
```

The reason for this is the need to load a separate JVM to execute tests. To mitigate this Grails interactive r
load a background JVM that can be resumed. If you do:

```
$ grails // load interactive mode
$ grails -> test-app
$ grails -> test-app
```

Test execution will be noticeably faster and is the recommended way to run tests in Grails. On older hardv
multiple cores (to run the separate JVMs) it is recommended you disable forked execution for tests to ach
times:

```
forkConfig = [maxMemory: 1024, minMemory: 64, debug: false, maxPerm: 256]
grails.project.fork = [
    test: false, // disable forked execution for test-app
    run: forkConfig, // configure settings for the run-app JVM
    …
]
```

Forked Execution and the Reloading Agent

In Grails 2.3 the reloading agent is no longer on the build system path unless you pass the -reload:
command:

```
grails -reloading run-app
```

The reason for this is that the default in Grails 2.3 and above is to load Grails application in a forked JVM
the forked JVM. If you do not wish to use forked JVMs then you must ensure that you run Grails with
Alternatively, you can enable forking with the following configuration in BuildConfig:

```
forkConfig = [maxMemory: 1024, minMemory: 64, debug: false, maxPerm: 256]
grails.project.fork = [
    test: forkConfig, // configure settings for the test-app JVM
    run: forkConfig, // configure settings for the run-app JVM
    war: forkConfig, // configure settings for the run-war JVM
    console: forkConfig // configure settings for the Swing console JVM
]
```

Forked Execution and Remote Debugging

The `grails-debug` command will no longer work with Grails for remote debugging sessions. The enabled debugging for the build system JVM, but not the JVM used in forked execution. The soluti `debug-fork` command line argument:

```
grails --debug-fork run-app
```

Alternatively you can set the `debug` setting to `true` in `BuildConfig` and use the regular `grails` com

```
forkConfig = [maxMemory: 1024, minMemory: 64, debug: true, maxPerm: 256]
grails.project.fork = [
    run: forkConfig, // configure settings for the run-app JVM
    ...
```

Forked Execution and Functional Test plugins

Some existing plugins (Cucumber plugin for example) do not work with 2.3.x forked execution because t running in the same JVM as the application under tests. For example it is not possible to setup fixture inside a functional test and have that data visible to the application under test since the application under The solution to this is to provide the necessary fixture data in the `BootStrap` of the application (only fc course).

4 Upgrading from Grails 2.3

The upgrade Command

The upgrade command has been removed from Grails 2.4. The procedure for upgrading to the latest detailed in the user guide from now on. Below are steps that must be taken to upgrade an application from

The set-grails-version Command

The set-grails-version command should be run to update the application's metadata to indicate whic application is built with.

Update to latest Plugin versions

You should update your application's BuildConfig to use the latest plugins compatible with Grails 2.3.

```
plugins {
        // plugins for the build system only
        build ':tomcat:7.0.52.1'

// plugins for the compile step
        compile ':scaffolding:2.1.0'
        compile ':cache:1.1.3'
        compile ':asset-pipeline:1.8.3'

// plugins needed at runtime but not for compilation
        runtime ':hibernate4:4.3.5.2' // or ':hibernate:3.6.10.14'
        runtime ':database-migration:1.4.0'
        runtime ':jquery:1.11.0.2'
        …
    }
```

You may get compilation errors or incompatibility problems with older versions of the above plugins insta

grails-debug Script Has Been Removed

The grails-debug and grails-debug.bat scripts have been removed. To debug the build sy -debug <command> and to debug the forked JVM run grails --debug-fork <command>.

New Command Object Data Binding Behavior

The data binding behavior for command objects has changed in Grails 2.4. Request parameter names may name of the controller action argument name that the request parameter should be bound to. For example, i buy action in the controller below a request parameter named buyer.name will be bound to the name argument and a request parameter named seller.name will be bound to the name property of the se Command Objects documentation for more details.

New Behavior For Domain Class Command Objects

If a command object's type is a domain class and there is no `id` request parameter then `null` will be [passed to the]
action unless the HTTP request method is "POST", in which case a new instance of the domain class wi[ll be created with]
the domain class constructor. For all of the cases where the domain class instance is non-null, data bindin[g is only performed if the]
HTTP request method is "POST", "PUT" or "PATCH". See the [Command Objects](#) documentation for mor[e details.]

Nullable Command Object Properties

The behavior in Grails 2.3.x is such that constrained properties in command objects and other classes ma[rked with @Validateable]
are all configured with `nullable: false` by default. Unconstrained properties were not configured wi[th nullable: false]
. In Grails 2.4 all non-static unconstrained properties in command object classes and other classes marke[d with @Validateable are]
all configured with `nullable: false`.

```
class StoreController {
    def buy(Person buyer, Person seller) {
        // …
    }
}

class Person {
    String name
}
```

If you wish to retain the old behavior, you can do so on a per-command object basis by using the [Constraints constraint]
explicitly and passing the `nullable: true` argument:

```
@Validateable(nullable = true)
class Person {
    String name
}
```

This will cause all properties to be nullable by default unless a constraint is explicitly added (similar to th[e behavior before Grails]
2.4).

See the [Command Objects documentation](#) for more details.

Ajax Tags Have Been Deprecated

The [formRemote](#), [remoteField](#), [remoteFunction](#) and [remoteLink](#) Ajax tags have been deprecated and will [be removed in a future]
version of Grails. Applications may provide their own Ajax tags and/or Javascript plugins may provide Aja[x tags.]

The Spring Data Binder Has Been Deprecated

The `grails.databinding.useSpringBinder` config property may be set to `true` to tell Grai
binder instead of the Grails data binder. The Spring data binder has been deprecated and will be removed
Grails. It is recommended that when upgrading to Grails 2.4 that the Grails data binder be used.

The resources Plugin

As of Grails 2.4 the `resources` plugin has been replaced with the `asset-pipeline` plugin
management plugin for newly created applications. See the [static resource abstraction](#) section of the Use
When upgrading an application to Grails 2.4 if you choose to continue using the resources plugin you will
or later as previous versions of the plugin are not compatible with Grails 2.4.

Static Holder Classes

The following deprecated classes have been removed from Grails 2.4.x:

- `org.codehaus.groovy.grails.commons.ApplicationHolder`
- `org.codehaus.groovy.grails.commons.ConfigurationHolder`
- `org.codehaus.groovy.grails.plugins.PluginManagerHolder`
- `org.codehaus.groovy.grails.web.context.ServletContextHolder`
- `org.codehaus.groovy.grails.compiler.support.GrailsResourceLoaderHold`

If you or any plugins you have installed are using these classes you will get a compilation error. The pro
updating to new plugins and using `grails.util.Holders` instead.

> ⚠️ If your application uses the jquery plugin you will need to update to version 1.11.0.2 or later
> versions of the plugin made use of the `ApplicationHolder` class. If your applicatio
> resources plugin you will need to update to version 1.2.7 or later as previous versions of the p
> use of the `ConfigurationHolder` class.

Changes To applicationContext.xml

The `web-app/WEB-INF/applicationContext.xml` file contains a bean definition for a gra
bean which is an instance of `org.codehaus.groovy.grails.commons.GrailsResourceI`
That bean definition needs to be removed from the file. The `grailsApplication`
`grailsResourceLoader` bean injected into it as shown below.

```
<bean id="grailsApplication" class=
"org.codehaus.groovy.grails.commons.GrailsApplicationFactoryBean">
    <description>Grails application factory bean</description>
    <property name="grailsDescriptor" value="/WEB-INF/grails.xml" />
    <property name="grailsResourceLoader" ref="grailsResourceLoader" />
</bean>
```

The grailsApplication bean definition should be left in the file but the grailsResourceLoade
be removed as shown below.

```
<bean id="grailsApplication" class=
"org.codehaus.groovy.grails.commons.GrailsApplicationFactoryBean">
    <description>Grails application factory bean</description>
    <property name="grailsDescriptor" value="/WEB-INF/grails.xml" />
</bean>
```

Changes to web.xml

The Sitemesh servlet filter has been removed and the GSP layout feature is now handled by [GrailsLayou
are using a customized web.xml should apply the customizations to a web.xml file of Grails 2.4 . This ap
that have used the "install-templates" to install template files in src/templates folder of the application. It's
src/templates to a different name and use a diff tool to apply the possible application specific customiza
with Grails 2.4 install-templates command.

Data Binding Changes

Prior to Grails 2.4 when data binding was performed with the params object in a controller, if the requ
body would be parsed and used for data binding instead of the params object. In Grails 2.4 this behavi
binding is initiated with params, the binding will always be done with the params object, without reg
request has a body. If binding is done with the request object, if the request has a body then the body w
data binding, otherwise the request parameters will be used for data binding.

```
class SomeController {

def someAction() {
        // Prior to Grails 2.4 if the request contains a body
        // then obj1 will be populated with values parsed from
        // the body instead of with values in params.

// With Grails 2.4 obj1 will be populated with values
        // in params.
        def obj1 = new SomeDomainClass(params)

// the same is true for the following
        def obj2 = new SomeDomainClass()
        obj2.properties = params
    }

def someOtherAction() {
        // If the request contains a body then obj1 will be
        // populated with values parsed from the body, otherwise
        // obj1 will be populated with the request parameters.

// This is not a new change in behavior.
        def obj1 = new SomeDomainClass()
        obj1.properties = request
    }
}
```

> ⚠ There is one release in the 2.3.x chain which has the 2.4 behavior described above and th
> 2.3.8. None of the 2.3.x releases before or after 2.3.8 have this behavior.

The allowedMethods Property And Unit Tests

The unit testing environment now respects the [allowedMethods](#) property in controllers. Prior to Gra
accessed a controller action which is supposed to be restricted to certain request methods could have skipp
request method in the unit test because the allowedMethods property was ignored by the unit test. As of
action is limited to be accessed with certain request methods, the unit test must be constructed to deal with

```
// grails-app/controllers/com/demo/DemoController.groovypackage com.demo

class DemoController {

static allowedMethods = [save: 'POST', update: 'PUT', delete: 'DELETE']

def save() {
        render 'Save was successful!'
    }

// …
}
```

```groovy
// test/unit/com/demo/DemoControllerSpec.groovy
package com.demo

import grails.test.mixin.TestFor
import spock.lang.Specification
import static javax.servlet.http.HttpServletResponse.*

@TestFor(DemoController)
class DemoControllerSpec extends Specification {

void "test a valid request method"() {
        when:
        request.method = 'POST'
        controller.save()

then:
        response.status == SC_OK
        response.text == 'Save was successful!'
    }
void "test an invalid request method"() {
        when:
        request.method = 'DELETE'
        controller.save()

then:
        response.status == SC_METHOD_NOT_ALLOWED
    }
}
```

scanning for JSP taglibs has to be configured, no JSTL default dependency

JSP taglib tld files aren't scanned by default any more. This must be configured with the grails.g
setting. It accepts a comma separated String value. Spring's PathMatchingResourcePatternResolver is used

You can get the previous behaviour by adding this setting to Config.groovy:

```
grails.gsp.tldScanPattern='classpath*:/META-INF/*.tld,/WEB-INF/tld/*.tld'
```

JSTL standard library is no more added as a dependency. In case you are using JSTL, you should also add

```
runtime 'javax.servlet:jstl:1.1.2'
        runtime 'taglibs:standard:1.1.2'
```

5 Configuration

It may seem odd that in a framework that embraces "convention-over-configuration" that we tackle this
default settings you can actually develop an application without doing any configuration whatso
demonstrates, but it's important to learn where and how to override the conventions when you need to. I
guide will mention what configuration settings you can use, but not how to set them. The assumption is t
the first section of this chapter!

5.1 Basic Configuration

For general configuration Grails provides two files:

- `grails-app/conf/BuildConfig.groovy`
- `grails-app/conf/Config.groovy`

Both of them use Groovy's [ConfigSlurper](#) syntax. The first, `BuildConfig.groovy`, is for settings that
Grails commands, such as `compile`, `doc`, etc. The second file, `Config.groovy`, is for settings
application is running. This means that `Config.groovy` is packaged with your application, but `Bui`
not. Don't worry if you're not clear on the distinction: the guide will tell you which file to put a particular s

The most basic syntax is similar to that of Java properties files with dot notation on the left-hand side:

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

Note that the value is a Groovy string literal! Those quotes around 'world' are important. In fact, th
advantages of the ConfigSlurper syntax over properties files: the property values can be any valid Gro
integers, or arbitrary objects!

Things become more interesting when you have multiple settings with the same base. For example, you co

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

both of which have the same base: `foo.bar`. The above syntax works but it's quite repetitive and verbos
of that verbosity by nesting properties at the dots:

```
foo {
    bar {
        hello = "world"
        good = "bye"
    }
}
```

or by only partially nesting them:

```
foo {
    bar.hello = "world"
    bar.good = "bye"
}
```

However, you can't nest after using the dot notation. In other words, this **won't** work:

```
// Won't work!
foo.bar {
    hello = "world"
    good = "bye"
}
```

Within both `BuildConfig.groovy` and `Config.groovy` you can access several implicit varia
values:

Variable	Description
userHome	Location of the home directory for the account that is running the Grails application.
grailsHome	Location of the directory where you installed Grails. If the `GRAILS_HOME` environment vari
appName	The application name as it appears in application.properties.
appVersion	The application version as it appears in application.properties.

For example:

```
my.tmp.dir = "${userHome}/.grails/tmp"
```

In addition, `BuildConfig.groovy` has

Variable	Description
grailsVersion	The version of Grails used to build the project.
grailsSettings	An object containing various build related settings, such as `baseDir`. It's of type `BuildS`

and `Config.groovy` has

Variable	Description
grailsApplication	The `GrailsApplication` instance.

Those are the basics of adding settings to the configuration file, but how do you access those settings fro
That depends on which config you want to read.

The settings in `BuildConfig.groovy` are only available from **command scripts** and can
`grailsSettings.config` property like so:

```
target(default: "Example command") {
    def maxIterations = grailsSettings.config.myapp.iterations.max
    …
}
```

If you want to read runtime configuration settings, i.e. those defined in `Config.groovy`, use the
object, which is available as a variable in controllers and tag libraries:

```
class MyController {
    def hello() {
        def recipient = grailsApplication.config.foo.bar.hello

render "Hello ${recipient}"
    }
}
```

and can be easily injected into services and other Grails artifacts:

```
class MyService {
    def grailsApplication

String greeting() {
        def recipient = grailsApplication.config.foo.bar.hello
        return "Hello ${recipient}"
    }
}
```

As you can see, when accessing configuration settings you use the same dot notation as when you define th

5.1.1 Built in options

Grails has a set of core settings that are worth knowing about. Their defaults are suitable for most proj
understand what they do because you may need one or more of them later.

Build settings

Let's start with some important build settings. Although Grails requires JDK 6 when developing your app
deploy those applications to JDK 5 containers. Simply set the following in `BuildConfig.groovy`:

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

Note that source and target levels are different to the standard public version of JDKs, so JDK 5 -> 1.5, JD
1.7.

In addition, Grails supports Servlet versions 2.5 and above but defaults to 2.5. If you wish to use newer fe
(such as 3.0 async support) you should configure the `grails.servlet.version` setting appropriatel

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

Runtime settings

On the runtime front, i.e. `Config.groovy`, there are quite a few more core settings:

- `grails.config.locations` - The location of properties files or addition Grails Config files th
  main configuration. See the [section on externalised config](#).

- `grails.enable.native2ascii` - Set this to false if you do not require native2ascii conversion
  files (default: true).

- `grails.views.default.codec` - Sets the default encoding regime for GSPs - can be one of
  (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: 'utf-8').

- `grails.mime.file.extensions` - Whether to use the file extension to dictate the mime typ
  (default: true).

- `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, incl
  grails.serverURL="http://my.yourportal.com". See [createLink](#). Also used by redirects.

- `grails.views.gsp.sitemesh.preprocess` - Determines whether SiteMesh preprocessing
  slows down page rendering, but if you need SiteMesh to parse the generated HTML from a GSP vie
  right option. Don't worry if you don't understand this advanced property: leave it set to true.

- `grails.reload.excludes` and `grails.reload.includes` - Configuring these directiv
  behavior for project specific source files. Each directive takes a list of strings that are the class name
  that should be excluded from reloading behavior or included accordingly when running the applicat
  the `run-app` command. If the `grails.reload.includes` directive is configured, then only t
  be reloaded.

War generation

- `grails.project.war.file` - Sets the name and location of the WAR file generated by the [war](#)

- `grails.war.dependencies` - A closure containing Ant builder syntax or a list of JAR filena
  what libraries are included in the WAR file.

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

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

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

5.1.2 Logging

The Basics

Grails uses its common configuration mechanism to provide the settings for the underlying [Log4j](#) log syst
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.g
'org.codehaus.groovy.grails.web.pages', only messages logged at 'error' level and above will be shown. Log
with 'org.apache.catalina' logger only show messages at the 'warn' level and above. What does that mean'
understand how levels work.

Logging levels

There 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 method `log.e`
message at the 'error' level. Likewise, `log.debug(msg)` will log it at 'debug'. Each of the above level
have a corresponding log method of the same name.

The logging system uses that *message* level combined with the configuration for the logger (see next secti
the message gets written out. For example, if you have an 'org.example.domain' logger 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 will be ig

Before we go on to loggers, a quick note about those 'off' and 'all' levels. These are special in that they
configuration; you can't log messages at these levels. So if you configure a logger with a level of 'off', t
written out. A level of 'all' means that you will see all messages. Simple.

Loggers

Loggers are fundamental to the logging system, but they are a source of some confusion. For a start,
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 instar
loggers are cached and uniquely identified by name, so if two separate classes use loggers with the same
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>', w
the artifact, for example 'controllers' or 'services', and `className` is the fully qualified name of the arti
have this service:

```
package org.example
class MyService {
    …
}
```

then the name of the logger will be 'grails.app.services.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
You can also pass a name to the `getLog()` method, such as "myLogger", but this is less common bec
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 ign
them at a level of 'warn' or lower. But is there a logger with this name in the application? No. So why ha
Because the above rule applies to any logger whose name *begins with* 'org.codehaus.groovy.grails.w
example, the rule applies to both the org.codehaus.groovy.grails.web.servlet.Grails
class and the 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 wo

The most common things that you will want to capture log output from are your controllers, services, an
convention mentioned earlier to do that: *grails.app.<artifactType>.<className>* . In particular the c
qualified, 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
Grails internals that you can use, especially when tracking down problems with your application:

- `org.codehaus.groovy.grails.commons` - Core artifact information such as class loading et
- `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 tha
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 provide
any messages that go to that logger are subject to the rules defined for the root logger. In other words, the
default configuration for the logging system.

Grails automatically configures the root logger to only handle messages at 'error' level and above, and all
to the console (stdout for those with a C background). You can customise this behaviour by specifyin
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 co
also configure the root logger to log to one or more named appenders (which we'll talk 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) appe
appender.

For power users there is an alternative syntax for configuring the root logger: the root `org.apache.log`
passed as an argument to the log4j closure. This lets you work 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 log
But what if you want to send them to a file? How do you make sure that messages from a particular logg
console? These questions and more will be answered as we look into appenders.

Appenders

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

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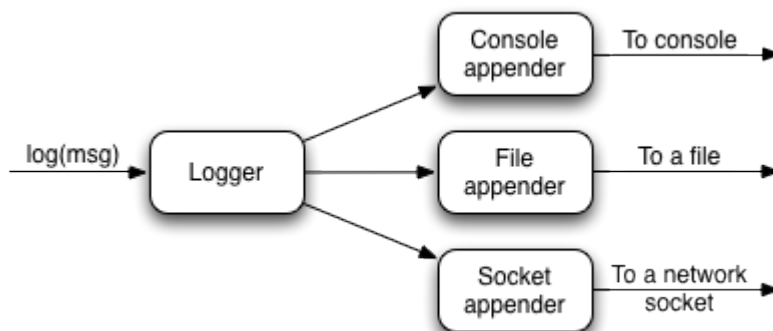
As you can see, a single logger may have several appenders attached to it. In a standard Grails configurati
named 'stdout' is attached to all loggers through the default root logger configuration. But that's the
appenders can be done within an 'appenders' block:

```
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 impleme example sets the `name`, `maxFileSize` and `file` properties of the `RollingFileAppender` instance

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

If you prefer to create the appender programmatically or if you want to use an appender implementation above syntax, simply declare an `appender` entry with an instance of the appender you 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 na log level methods from the previous section:

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

This will ensure that the 'grails.app.controllers.BookController' logger sends log messages to 'myA appenders configured for the root logger. To add more than one appender to the logger, then add them to th

```
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 appende 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.controllers.BookController'. That declared for a given logger wins. What you probably want to do is limit what level of messages an appende

An appender that is attached to a logger configured with the 'all' level will generate a lot of logging infor in a file, but it makes working at the console difficult. So we configure the console appender to only wr 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 argu appenders, but do note that you currently have to specify a `Level` instance - a string such as "info" will no

Custom Layouts

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

- `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-specifi
is a problem with this approach: you have to provide the full logging configuration each time you defin
other words, you cannot selectively override parts of the configuration - it's all or nothing.

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

```
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
environment block.

Full stacktraces

When exceptions occur, there can be an awful lot of noise in the stacktrace from Java and Groovy inte
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 writes
`stacktrace.log`. As with other loggers though, you can change its behaviour in the configuration. F
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
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 w
console.

Finally, you can completely disable stacktrace filtering by setting the grails.full.stacktrace VM

```
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 request p
request. To mask out the values of secure request parameters, specify the param
grails.exceptionresolver.params.exclude config property:

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

Request parameter logging may be turned off altogether by grails.exceptionresolver.logRequestParameters config property to false. The defaul 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 bas What this means is that unless you override a parent setting, a logger retains the level and the appenders c 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' appender to log to 'stdout' for a particular logger? Change the 'additivity' for a logger in that case.

Additivity simply determines whether a logger inherits the configuration from its parent. If additivity is fa The default for all loggers is true, i.e. they inherit the configuration. So how do you change this setting? He

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

81

So when you specify a log level, add an 'additivity' named argument. Note that you when you specify 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 contai aren't interesting when diagnosing problems. So Grails uses a impleme org.codehaus.groovy.grails.exceptions.StackTraceFilterer interface to filter out To customize the approach used for filtering, implement that interface in a class in src/groovy or src Config.groovy:

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

In addition, Grails customizes the display of the filtered stacktrace to make the information more read implement the org.codehaus.groovy.grails.exceptions.StackTracePrinter interface 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 implementation is org.codehaus.groovy.grails.web.errors.ErrorsViewStackTra registered as a Spring bean. To use your own implementation, either org.codehaus.groovy.grails.exceptions.StackTraceFilterer directly ErrorsViewStackTracePrinter and register it in grails-app/conf/spring/resources.

```
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 us
logging library is used. But if you're not one of those and want to use an alternative, such as the JDK log
you can do so by simply excluding a couple of dependencies from the 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
configuration DSL that's just been described. Instead, you will have to use the standard configuration mech
choose.

5.1.3 GORM

Grails provides the following GORM configuration options:

- `grails.gorm.failOnError` - If set to `true`, causes the `save()` method on domai
  `grails.validation.ValidationException` if validation fails during a save. This option n
  of Strings representing package names. If the value is a list of Strings then the failOnError behavio
  domain classes in those packages (including sub-packages). See the save method docs for more inform

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
  need to explicitly flush using `save(flush: true)`.

5.2 Environments

Per Environment Configuration

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

```
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` block s
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 specific env:

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

In addition, there are 3 preset environments known to Grails: `dev`, `prod`, and `test` for developme
`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 the En

```
import grails.util.Environment

...

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

Per Environment Bootstrapping

It's often desirable to run code when your application starts up on a per-environment basis. To 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 execu class yourself to execute your own environment specific logic:

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

5.3 The DataSource

Since Grails is built on Java technology setting up a data source requires some knowledge of JDBC (the
stand for Java Database Connectivity).

If you use a database other than H2 you need a JDBC driver. For example for MySQL you would need [Co

Drivers typically come in the form of a JAR archive. It's best to use the dependency resolution to resolve t
Maven repository, for example you could add a dependency for the MySQL driver like this:

```
dependencies {
        runtime 'mysql:mysql-connector-java:5.1.29'
    }
```

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

Once you have the JAR resolved you need to get familiar Grails' DataSource descri
`grails-app/conf/DataSource.groovy`. This file contains the dataSource definition which
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-drop', 'cr

- `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 org.hibernate.dialect package for available dialects.

- `readOnly` - If `true` makes the DataSource read-only, which results in the connection pool calling on each `Connection`

- `transactional` - If `false` leaves the DataSource's transactionManager bean outside the cha manager implementation. This only applies to additional datasources.

- `persistenceInterceptor` - The default datasource is automatically wired up to the persi datasources are not wired up automatically unless this is set to `true`

- `properties` - Extra properties to set on the DataSource bean. See the Tomcat Pool documentation format documentation of the properties.

- `jmxExport` - If `false`, will disable registration of JMX MBeans for all DataSources. By default for DataSources with `jmxEnabled = true` in properties.

A typical configuration for MySQL may be something like:

```
dataSource {
    pooled = true
    dbCreate = "update"
    url = "jdbc:mysql://localhost:3306/my_database"
    driverClassName = "com.mysql.jdbc.Driver"
    dialect = org.hibernate.dialect.MySQL5InnoDBDialect
    username = "username"
    password = "password"
    properties {
        jmxEnabled = true
        initialSize = 5
        maxActive = 50
        minIdle = 5
        maxIdle = 25
        maxWait = 10000
        maxAge = 10 * 60000
        timeBetweenEvictionRunsMillis = 5000
        minEvictableIdleTimeMillis = 60000
        validationQuery = "SELECT 1"
        validationQueryTimeout = 3
        validationInterval = 15000
        testOnBorrow = true
        testWhileIdle = true
        testOnReturn = false
        jdbcInterceptors = "ConnectionState;StatementCache(max=200)"
        defaultTransactionIsolation = java.sql.Connection.TRANSACTION_READ_COMMITT
    }
}
```

> 🚫 When configuring the DataSource do not include the type or the def keyword before
> configuration settings as Groovy will treat these as local variable definitions and they
> 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:3306/my_database"
    driverClassName = "com.mysql.jdbc.Driver"
    dialect = org.hibernate.dialect.MySQL5InnoDBDialect
    username = "username"
    password = "password"
    properties {
```

89

```
        // Documentation for Tomcat JDBC Pool
        // http://tomcat.apache.org/tomcat-7.0-doc/jdbc-pool.html#Common_Attribute
        //
https://tomcat.apache.org/tomcat-7.0-doc/api/org/apache/tomcat/jdbc/pool/PoolConf
        jmxEnabled = true
        initialSize = 5
        maxActive = 50
        minIdle = 5
        maxIdle = 25
        maxWait = 10000
        maxAge = 10 * 60000
        timeBetweenEvictionRunsMillis = 5000
        minEvictableIdleTimeMillis = 60000
        validationQuery = "SELECT 1"
        validationQueryTimeout = 3
        validationInterval = 15000
        testOnBorrow = true
        testWhileIdle = true
        testOnReturn = false
        ignoreExceptionOnPreLoad = true
        // http://tomcat.apache.org/tomcat-7.0-doc/jdbc-pool.html#JDBC_interceptor
        jdbcInterceptors = "ConnectionState;StatementCache(max=200)"
        defaultTransactionIsolation = java.sql.Connection.TRANSACTION_READ_COMMITT
        // controls for leaked connections
        abandonWhenPercentageFull = 100 // settings are active only when pool is f
        removeAbandonedTimeout = 120
        removeAbandoned = true
        // use JMX console to change this setting at runtime
        logAbandoned = false // causes stacktrace recording overhead, use only for
        // JDBC driver properties
        // Mysql as example
        dbProperties {
            // Mysql specific driver properties
            //
http://dev.mysql.com/doc/connector-j/en/connector-j-reference-configuration-prope
            // let Tomcat JDBC Pool handle reconnecting
            autoReconnect=false
            // truncation behaviour
            jdbcCompliantTruncation=false
            // mysql 0-date conversion
            zeroDateTimeBehavior='convertToNull'
            // Tomcat JDBC Pool's StatementCache is used instead, so disable mysql
            cachePrepStmts=false
            cacheCallableStmts=false
            // Tomcat JDBC Pool's StatementFinalizer keeps track
            dontTrackOpenResources=true
            // performance optimization: reduce number of SQLExceptions thrown in
            holdResultsOpenOverStatementClose=true
            // enable MySQL query cache - using server prep stmts will disable que
            useServerPrepStmts=false
            // metadata caching
            cacheServerConfiguration=true
            cacheResultSetMetadata=true
            metadataCacheSize=100
            // timeouts for TCP/IP
            connectTimeout=15000
            socketTimeout=120000
            // timer tuning (disable)
            maintainTimeStats=false
            enableQueryTimeouts=false
            // misc tuning
            noDatetimeStringSync=true
```

```
        }
    }
}
```

More on dbCreate

Hibernate can automatically create the database tables required for your domain model. You have some co
it does this through the dbCreate property, which can take these values:

- **create** - Drops the existing schema and creates the schema on startup, dropping existing tables, indexe

- **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
  this can't properly handle many schema changes like column renames (you're left with the old colum
  data).

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

- any other value - does nothing

You can also remove the dbCreate setting completely, which is recommended once your schema
definitely when your application and database are deployed in production. Database changes are then
migrations, either with SQL scripts or a migration tool like [Liquibase](#) (the [Database Migration](#) plugin use
integrated with Grails and GORM).

5.3.1 DataSources and Environments

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

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
        }
    }
}
```

5.3.2 JNDI DataSources

Referring to a JNDI DataSource

Most Java EE containers supply `DataSource` instances via Java Naming and Directory Interface (JN
definition of JNDI data sources as follows:

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

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

Configuring a Development time JNDI resource

The way in which you configure JNDI data sources at development time is plugin dependent. Using the
define JNDI resources using the `grails.naming.entries` setting in `grails-app/conf/Config`

```
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"
    ]
]
```

5.3.3 Automatic Database Migration

The `dbCreate` property of the `DataSource` definition is important as it dictates what Grails should d
to automatically generating the database tables from GORM classes. The options are described in the Data

- `create`
- `create-drop`
- `update`
- `validate`
- no value

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

It's tempting to switch to `update` so you retain existing data and only update the schema when your code update support is very conservative. It won't make any changes that could result in data loss, and doesn't de tables, so you'll be left with the old one and will also have the new one.

Grails supports migrations via the [Database Migration](#) plugin which can be installed by de `grails-app/conf/BuildConfig.groovy`:

```
grails.project.dependency.resolution = {
    …
    plugins {
        runtime ':database-migration:1.3.1'
    }
}
```

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

5.3.4 Transaction-aware DataSource Proxy

The actual `dataSource` bean is wrapped in a transaction-aware proxy so you will be given the connec 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 connect able to see changes that haven't been committed yet (assuming you have a sensible transaction `READ_COMMITTED` or better).

The "real" unproxied `dataSource` is still available to you if you need access to it; its bean name is `data`

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
```

5.3.5 Database Console

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

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

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

```
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}"
    }
}
```

⛔ If you enable the console in production be sure to guard access to it using a trusted security fra

Configuration

By default the console is configured for an H2 database which will work with the default settings if ye
external database - you just need to change the JDBC URL to `jdbc:h2:mem:devDB`. If you've config
(e.g. MySQL, Oracle, etc.) then you can use the Saved Settings dropdown to choose a settings templa
username/password information from your DataSource.groovy.

5.3.6 Multiple Datasources

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

Configuring Additional DataSources

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

```
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
another `dataSource` block (at the top level, in an environment block, or both, just like the standard D
with a custom name, separated by an underscore. For example, this configuration adds a second `DataSo`
the development environment and Oracle in 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 the
in the mapping block to configure a non-default DataSource. For example, if you want to use the Zip
'lookup' DataSource, configure it like this;

```
class ZipCode {

String code

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

96

A domain class can also use two or more `DataSources`. Use the `datasources` property with a list of 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 `'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 each `Dat` GORM calls for a particular `DataSource`. For example, consider this class which uses two `DataSourc`

```
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 w
you can call GORM methods on the 'auditing' `DataSource` with the `DataSource` 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.

Hibernate Mapped Domain Classes

You can also partition annotated Java classes into separate datasources. Classes using the default data
grails-app/conf/hibernate/hibernate.cfg.xml. To specify that an annotated class uses
create a `hibernate.cfg.xml` file for that datasource with the file name prefixed with the datasource n

For example if the `Book` class is in the default datasource, you would
grails-app/conf/hibernate/hibernate.cfg.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>
      <mapping class='org.example.Book'/>
   </session-factory>
</hibernate-configuration>
```

and if the `Library` class is in the "ds2" datasource, you would
grails-app/conf/hibernate/ds2_hibernate.cfg.xml:

```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>
      <mapping class='org.example.Library'/>
   </session-factory>
</hibernate-configuration>
```

The process is the same for classes mapped with hbm.xml files - just list them in the appropriate hibernate.

Services

Like Domain classes, by default Services use the default `DataSource` and `PlatformTransactionM`
Service to use a different `DataSource`, use the static `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
`DataSource` is the same as the Service.

Note that the datasource specified in a service has no bearing on which datasources are used for domain
by their declared datasources in the domain classes themselves. It's used to declare which transaction man.

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

Transactions across multiple datasources

Grails uses the Best Efforts 1PC pattern for handling transactions across multiple datasources.

The Best Efforts 1PC pattern is fairly general but can fail in some circumstances that the developer mu
non-XA pattern that involves a synchronized single-phase commit of a number of resources. Because th
never be as safe as an XA transaction, but is often good enough if the participants are aware of the compro

99

The basic idea is to delay the commit of all resources as late as possible in a transaction so that the only t[...] an infrastructure failure (not a business-processing error). Systems that rely on Best Efforts 1PC reason t[...] are rare enough that they can afford to take the risk in return for higher throughput. If business-pro[...] designed to be idempotent, then little can go wrong in practice.

The BE1PC implementation was added in Grails 2.3.6. . Before this change additional datasources didn'[...] initiated in Grails. The transactions in additional datasources were basically in auto commit mode. In som[...] wanted behavior. One reason might be performance: on the start of each new transaction, the BE1PC tran[...] new transaction to each datasource. It's possible to leave an additional datasource out of the BE1PC transa[...] `transactional = false` in the respective configuration block of the additional dataSource. Datasou[...] `true` will also be left out of the chained transaction manager (since 2.3.7).

By default, the BE1PC implementation will add all beans implementing the Spring <u>`PlatformTransact`</u>[...] to the chained BE1PC transaction manager. For example, a possible <u>`JMSTransactionManager`</u> bean[...] context would be added to the Grails BE1PC transaction manager's chain of transaction managers.

You can exclude transaction manager beans from the BE1PC implementation with the this configuration o[...]

```
grails.transaction.chainedTransactionManagerPostProcessor.blacklistPattern = '.*'
```

The exclude matching is done on the name of the transaction manager bean. The transaction manag[...] `transactional = false` or `readOnly = true` will be skipped and using this configuration opt[...] case.

XA and Two-phase Commit

When the Best Efforts 1PC pattern isn't suitable for handling transactions across multiple transactio[...] datasources), there are several options available for adding XA/2PC support to Grails applications.

The <u>Spring transactions documentation</u> contains information about integrating the JTA/XA transacti[...] application servers. In this case, you can configure a bean with the name `transactionM`[...] `resources.groovy` or `resources.xml` file.

There is also <u>Atomikos plugin</u> available for XA support in Grails applications.

5.4 Externalized Configuration

Some deployments require that configuration be sourced from more than one place and be changeable w[...] of the application. In order to support deployment scenarios such as these the configuration can be exte[...] Grails at the locations of the configuration files that should be used by adding a `grails.config.`[...] `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 ConfigSlurper conf 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 yo use for this is re-using configuration provided by plugins across multiple applications.

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

Values that have the same name as previously defined values will overwrite the existing values, and the 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.locations pr values defined in your application Config.groovy file which may not be what you want. You may wa values be be loaded that can be overridden in either your application's Config.groovy file or in a na this you can use the grails.config.defaults.locations property.

This property supports the same values as the grails.config.locations property (i.e. paths to con or classes), but the config described by grails.config.defaults.locations will be loaded *be can therefore be overridden. Some plugins use this mechanism to supply one or more sets of default co choose to include in your application config.

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

5.5 Versioning

Versioning Basics

Grails has built in support for application versioning. The version of the application is set to 0.1 application with the [create-app](#) command. The version is stored in the application meta data file applica 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 application created WAR file.

Detecting Versions at Runtime

You can detect the application version using Grails' support for application metadata using the [Gra](#) example within [controllers](#) there is an implicit [grailsApplication](#) variable that can be used:

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

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

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

or the GrailsUtil class:

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

5.6 Project Documentation

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

The documentation engine uses a variation on the [Textile](#) syntax to automatically create project documen
formatting etc.

Creating project documentation

To use the engine you need to follow a few conventions. First, you need to create a `src/docs/gui`
documentation source files will go. Then, you need to create the source docs themselves. Each chapter s
file as should all numbered sub-sections. You will end up with 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 put sub-s
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 yo
be. To do that, you add a `src/docs/guide/toc.yml` file that contains the structure and titles for ea
[YAML](#) format and basically represents the structure of the user guide in tree form. For example, t
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
.gdoc extension) followed by a colon. The next line should contain `title:` plus the title of the section
Every sub-section then has its own line after the title. Leaf nodes, i.e. those without any sub-sections, decl
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 generate
also make sure that all section names, i.e. the gdoc filenames, should be unique since they are used for c
for the HTML filenames. Don't worry though, the documentation engine will warn you of duplicate section

Creating reference items

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

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

Configuring Output Properties

There are various properties you can set within your `grails-app/conf/Config.groovy` file that 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, the applica

You can also customise the look of the documentation and provide images by setting a few other options:

- **grails.doc.css** - The location of a directory containing custom CSS files (type `java.io.File`)

- **grails.doc.js** - The location of a directory containing custom JavaScript files (type `java.io.File`)

- **grails.doc.style** - The location of a directory containing custom HTML templates for the guide (type

- **grails.doc.images** - The location of a directory containing image files for use in the style te documentation pages themselves (type `java.io.File`)

One of the simplest ways to customise the look of the generated guide is to provide a value for `grails.` custom.css file in the corresponding directory. Grails will automatically include this CSS file in the gui custom-pdf.css file in that directory. This allows you to override the styles for the PDF version of the guide

Generating Documentation

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

```
grails doc
```

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

Documentation Syntax

As mentioned the syntax is largely similar to Textile or Confluence style wiki markup. The following se
the syntax basics.

Basic Formatting

Monospace: `monospace`

```
@monospace@
```

Italic: *italic*

```
_italic_
```

Bold: **bold**

```
*bold*
```

Image:

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

You can also link to internal images like so:

```
!someFolder/my_diagram.png!
```

This will link to an image stored locally within your project. There is currently no default location for specify one with the `grails.doc.images` setting in Config.groovy like so:

```
grails.doc.images = new File("src/docs/images")
```

In this example, you would put the my_diagram.png file in the directory 'src/docs/images/someFolder'.

Linking

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

```
[Pivotal|http://www.pivotal.io/oss]
```

or

```
"Pivotal":http://www.pivotal.io/oss
```

For links to other sections inside the user guide you can use the `guide:` prefix with the name of the sectio

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

The section name comes from the corresponding gdoc filename. The documentation engine will warn you
your guide break.

To link to reference items you can use a special syntax:

```
[renderPDF|controllers]
```

In this case the category of the reference item is on the right hand side of the | and the name of the referenc

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 add addi
you can configure them in `grails-app/conf/Config.groovy`. For 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 Hibernate web

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 XM

```
<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}
```

5.7 Dependency Resolution

Grails features a dependency resolution DSL that lets you control how plugins and JAR dependencies are r

You can choose to use Aether (since Grails 2.3) or Apache Ivy as the dependency resolution engine. /
resolution library used by the Maven build tool, so if you are looking for Maven-like behavior then Aethe
allows more flexibility if you wish to resolve jars from flat file systems or none HTTP repositorie
dependency resolution engine for Grails applications since Grails 2.3.

> ⛔ As of Grails 2.4 the Ivy resolver is considered deprecated and no longer maintained. It is rec
> all users switch to using Aether.

To configure which dependency resolution engine to use you can specify the `grails.project.de`
setting in `grails-app/conf/BuildConfig.groovy`. The default setting is shown below:

```
grails.project.dependency.resolver = "maven" // or ivy
```

You can then specify a `grails.project.dependency.resolution` prop
`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.servlet.version = "3.0" // Change depending on target container compliance
grails.project.class.dir = "target/classes"
grails.project.test.class.dir = "target/test-classes"
grails.project.test.reports.dir = "target/test-reports"
grails.project.work.dir = "target/work"
grails.project.target.level = 1.6
grails.project.source.level = 1.6
//grails.project.war.file = "target/${appName}-${appVersion}.war"

grails.project.fork = [
    // configure settings for compilation JVM, note that if you alter the Groovy
compilation is required
    //  compile: [maxMemory: 256, minMemory: 64, debug: false, maxPerm: 256, daem

// configure settings for the test-app JVM, uses the daemon by default
    test: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256, daemon:true
    // configure settings for the run-app JVM
    run: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256, forkReserve:
    // configure settings for the run-war JVM
    war: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256, forkReserve:
    // configure settings for the Console UI JVM
    console: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256]
]

grails.project.dependency.resolver = "maven" // or ivy
grails.project.dependency.resolution = {
    // inherit Grails' default dependencies
    inherits("global") {
        // specify dependency exclusions here; for example, uncomment this to dis
        // excludes 'ehcache'
    }
    log "error" // log level of Ivy resolver, either 'error', 'warn', 'info', 'de
    checksums true // Whether to verify checksums on resolve
    legacyResolve false // whether to do a secondary resolve on plugin installati
and here for backwards compatibility

repositories {
        inherits true // Whether to inherit repository definitions from plugins

grailsPlugins()
        grailsHome()
        mavenLocal()
        grailsCentral()
        mavenCentral()
        // uncomment these (or add new ones) to enable remote dependency resoluti
Maven repositories
        //mavenRepo "http://repository.codehaus.org"
        //mavenRepo "http://download.java.net/maven/2/"
    }

dependencies {
        // specify dependencies here under either 'build', 'compile', 'runtime',
'provided' scopes e.g.
        runtime 'mysql:mysql-connector-java:5.1.24'
        compile 'org.springframework.integration:spring-integration-core:2.2.5.RE
    }

plugins {
        // plugins for the build system only
    }
}
```

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

5.7.1 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

- `optional` (Aether only): Dependencies considered optional and not required for the execution of th

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

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

This uses the string syntax: `group:name:version`.

If you are using Aether as the dependency resolution library, the Maven pattern of:

```
<groupId>:<artifactId>[:<extension>[:<classifier>]]:<version>
```

You can also use a Map-based syntax:

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

Possible settings to the map syntax are:

- group - The group / organization (or groupId in Maven terminology)

- name - The dependency name (or artifactId in Maven terminology)

- version - The version of the dependency

- extension (Aether only) - The file extension of the dependency

- classifier - The dependency classifier

- branch (Ivy only) - The branch of the dependency

- transitive (Ivy only) - Whether the dependency has transitive dependencies

As you can see from the list above some dependency configuration settings work only in Aether and some

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 transit usually what you want, but there are occasions where you want a dependency without all its baggage 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 dependencies or are unnecessary. For example, many Apache projects have 'commons-logging' as a transhouldn't be included in a Grails project (we use SLF4J). That's where the `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 group and artifact IDs (if you use the Map notation). You may also come across `exclude` as well, but that string or Map:

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

Dependency Management (Aether Only)

If you are using Aether then you can take advantage of Maven's notion of [Dependency Management](#).

To do so you use a `management` block, for example:

```
management {
    dependency "commons-logging:commons-logging:1.1.3"
}
```

The above declaration will force all any transitive dependencies on `commons-logging` to use the 1 having to declare an explicit dependency on `commons-logging` yourself. In addition to the version, scope and exclusion rules of a dependency.

Where are the JARs?

With all these declarative dependencies, you may wonder where all the JARs end up. They have to go
default Grails puts them into a directory, called the dependency cache, that resides on your
user.home/.grails/ivy-cache or user.home/.m2/repository when using Aether. You
the 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.

5.7.2 Dependency Repositories

Remote Repositories

Initially your BuildConfig.groovy does not use any remote public Maven repositories. There is a d
repository that will locate the JAR files Grails needs from your Grails installation. To use a public rep
repositories block:

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

In this case the default public Maven repository is specified.

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.

Offline Mode

There are times when it is not desirable to connect to any remote repositories (whilst working on the tr
case you can use the offline flag to execute Grails commands and Grails will not connect to any remote

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

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

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

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

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

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

Authentication with Aether

To authenticate with Aether you can either define the credentials on the repository definition:

```
mavenRepo(url:"http://localhost:8082/myrepo") {
    auth username: "foo", password: "bar"
}
```

Or you can specify an id on the repository:

```
mavenRepo(id:'myrepo', url:"http://localhost:8082/myrepo")
```

And then declare your credentials in USER_HOME/.grails/settings.groovy:

```
grails.project.dependency.authentication = {
    credentials {
        id = "myrepo"
        username = "admin"
        password = "password"
    }
}
```

Authentication with Ivy

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
grails.project.ivy.authentication setting:

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

5.7.3 Debugging Resolution

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

```
// log level of the Aether or 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 Ivy
This helps ensure that the dependencies are valid. But for a variety of reasons some dependencies
checksums in the repositories, even if they are valid JARs. To get round this, you can disable Ivy's depende

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

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

5.7.4 Inherited Dependencies

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

```
inherits "global"
```

Inside the `BuildConfig.groovy` file. To exclude specific inherited dependencies you use the `exclud`

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

5.7.5 Providing Default Dependencies

Most Grails applications have runtime dependencies on several jar files that are provided by the Grails f
libraries like Spring, Sitemesh, Hibernate etc. When a war file is created, all of these dependencies will
application may choose to exclude these jar files from the war. This is useful when the jar files will be pro
would normally be the case if multiple Grails applications are deployed to the same container.

The dependency resolution DSL provides a mechanism to express that all of the default dependencies
container. This is done by invoking the `defaultDependenciesProvided` method and passing `true`

```
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
> dependencies will be included in the war.

5.7.6 Snapshots and Other Changing Dependencies

Configuration Changing dependencies

Typically, dependencies are constant. That is, for a given combination of `group`, `name` and `version`
refers to will never change. The Grails dependency management system uses this fact to cache depenc
having to download them from the source repository each time. Sometimes this is not desirable. For exam
the convention of a *snapshot* (i.e. a dependency with a version number ending in "-SNAPSHOT") that
time while still retaining the same version number. We call this a "changing dependency".

Whenever you have a changing dependency, Grails will always check the remote repository for a new ve
when a changing dependency is encountered during dependency resolution its last modified timestar
compared against the last modified timestamp in the dependency repositories. If the version on the remo
newer than the version in the local cache, the new version will be downloaded and used.

> ⚠ Be sure to read the next section on "Dependency Resolution Caching" in addition to this one
> changing dependencies.

All dependencies (jars and plugins) with a version number ending in `-SNAPSHOT` are **implicitly** cons
Grails. You can also explicitly specify that a dependency is changing by setting the changing flag in the c
only required for Ivy, Aether does not support the 'changing' flag and treats dependencies that end with -SN

```
runtime ('org.my:lib:1.2.3') {
    changing = true
}
```

Aether and SNAPSHOT dependencies

The semantics for handling snapshots when using Aether in Grails are the same as those when using th
default snapshot check policy is to check once a day for a new version of the dependency. This means
published during the day to a remote repository you may not see that change unless you manually clear out

If you wish to change the snapshot update policy you can do so by configuring an `updatePolicy` for
snapshot was resolved from, for example:

```
repositories {
    mavenCentral {
        updatePolicy "interval:1"
    }
}
```

The above example configures an update policy that checks once a minute for changes. Note that that an u
above will seriously impact performance of dependency resolution. The possibly configuration values for
follows:

- `never` - Never check for new snapshots

- `always` - Always check for new snapshots

- `daily` - Check once a day for new snapshots (the default)

- `interval:x` - Check once every x minutes for new snapshots

Ivy and Changing dependencies

For those used to Maven snapshot handling, if you use Aether dependency management you can expe
Maven. If you choose to use Ivy there is a caveat to the support for changing dependencies that you sho
stop looking for newer versions of a dependency once it finds a remote repository that has the dependency.

Consider the following setup:

```
grails.project.dependency.resolution = {
    repositories {
        mavenLocal()
        mavenRepo "http://my.org/repo"
    }
    dependencies {
        compile "myorg:mylib:1.0-SNAPSHOT"
    }
```

In this example we are using the local maven repository and a remote network maven repository. As
dependency and the local Maven cache do not contain the dependency but the remote repository
dependency resolution the following actions will occur:

- maven local repository is searched, dependency not found

- maven network repository is searched, dependency is downloaded to the cache and used

Note that the repositories are checked in the order they are defined in the `BuildConfig.groovy` file.

If we perform dependency resolution again without the dependency changing on the remote server, the foll

- maven local repository is searched, dependency not found

- maven network repository is searched, dependency is found to be the same "age" as the version in
  updated (i.e. downloaded)

Later on, a new version of `mylib 1.0-SNAPSHOT` is published changing the version on the server. T
dependency resolution, the following will happen:

- maven local repository is searched, dependency not found

- maven network repository is searched, dependency is found to newer than version in the cache
  downloaded to the cache)

So far everything is working well.

Now we want to test some local changes to the `mylib` library. To do this we build it locally and install it
(how doesn't particularly matter). The next time we perform a dependency resolution, the following will oc

- maven local repository is searched, dependency is found to newer than version in the cache
  downloaded to the cache)

- maven network repository is NOT searched as we've already found the dependency

This is what we wanted to occur.

Later on, a new version of `mylib 1.0-SNAPSHOT` is published changing the version on the server. T
dependency resolution, the following will happen:

- maven local repository is searched, dependency is found to be the same "age" as the version in
  updated (i.e. downloaded)

- maven network repository is NOT searched as we've already found the dependency

This is likely to not be the desired outcome. We are now out of sync with the latest published snapshot using the version from the local maven repository.

The rule to remember is this: when resolving a dependency, Ivy will stop searching as soon as it finds dependency at the specified version number. It will **not** continue searching all repositories trying to find instance.

To remedy this situation (i.e. build against the *newer* version of mylib 1.0-SNAPSHOT in the remeither:

- Delete the version from the local maven repository, or
- Reorder the repositories in the BuildConfig.groovy file

Where possible, prefer deleting the version from the local maven repository. In general, when you have fi locally built SNAPSHOT always try to clear it from the local maven repository.

> ⚠ This changing dependency behaviour is an unmodifiable characteristic of the underlying management system Apache Ivy. It is currently not possible to have Ivy search all repositories newer versions (in terms of modification date) of the same dependency (i.e. the same coml group, name and version). If you want this behavior consider switching to Aet dependency manager.

5.7.7 Dependency Reports

As mentioned in the previous section a Grails application consists of dependencies inherited from the 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 (scope) you want a report for by passing an argument containing the configuration name:

```
grails dependency-report runtime
```

As of Grails 2.3 the dependency-report command will also output to the console a graph of application. Example output it shown below:

```
compile - Dependencies placed on the classpath for compilation (total: 73)
+--- org.codehaus.groovy:groovy-all:2.0.6
+--- org.grails:grails-plugin-codecs:2.3.0
|     --- org.grails:grails-web:2.3.0
|           --- commons-fileupload:commons-fileupload:1.2.2
|           --- xpp3:xpp3_min:1.1.4c
|           --- commons-el:commons-el:1.0
|           --- opensymphony:sitemesh:2.4
|           --- org.springframework:spring-webmvc:3.1.2.RELEASE
|     --- commons-codec:commons-codec:1.5
|     --- org.slf4j:slf4j-api:1.7.2
+--- org.grails:grails-plugin-controllers:2.3.0
|     --- commons-beanutils:commons-beanutils:1.8.3
|     --- org.grails:grails-core:2.3.0
...
```

5.7.8 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
plugin is installed into an application the application automatically inherits the dependencies of the plugin.

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

```
compile('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 application de
if you're using the Map syntax:

```
compile 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 recommen
> 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 ca 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 specifies `excludes` method, effectively excluding the javassist library as a dependency.

5.7.9 Maven Integration

When using the Grails Maven plugin with the Maven build tool, Grails' dependency resolution mecha assumed 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 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.

5.7.10 Deploying to a Maven Repository

If you use Maven to build your Grails project, you can use the standard Maven targets `mvn  install` a you can deploy a Grails project or plugin to a Maven repository using the [release]() plugin.

The plugin provides the ability to publish Grails projects and plugins to local and remote Maven reposi additional targets added by the plugin:

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

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

maven-install

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

```
grails maven-install
```

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

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 w
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 repository sp
your `grails-app/conf/BuildConfig.groovy` file or in your `$USER_HOME/.grails/setti`

```
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 example the following XML:

```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 (e.g. "http" from 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 conventions or plugin descriptor.

Projects

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

```
grails set-version 0.2
```

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

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

Plugins

With a Grails plugin the `groupId` and `version` are taken from the following properties in the Gr
descriptor:

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

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

5.7.11 Plugin Dependencies

You can declaratively specify plugins as dependencies via the dependency DSL instead of using the <u>install</u>

```
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.

Latest Integration

> ⊖ Only the Ivy dependency manager supports the "latest.integration" version. For Aether you ca
> similar effect with version ranges.

You can specify to use the latest version of a particular plugin by using "latest.integration" as the version n

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

Integration vs. Release

The "latest.integration" version label will also include resolving snapshot versions. To not include snaps
"latest.release" label:

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

> ⚠ The "latest.release" label only works with Maven compatible repositories. If you have
> 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. For

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

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

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

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

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

5.7.12 Caching of Dependency Resolution Results

As a performance optimisation, when using Ivy (this does not apply to Aether), Grails does not resolve
command invocation. Even with all the necessary dependencies downloaded and cached, resolution may
minimise this cost, Grails caches the result of dependency resolution (i.e. the location on the local file sys
dependencies, typically inside the dependency cache) and reuses this result for subsequent commands
expect that nothing has changed.

Grails only performs dependency resolution under the following circumstances:

- The project is clean (i.e. fresh checkout or after `grails clean`)

- The `BuildConfig.groovy` file has changed since the last command was run

- The `--refresh-dependencies` command line switch is provided to the command (any comman

- The `refresh-dependencies` command is the command being executed

Generally, this strategy works well and you can ignore dependency resolution caching. Every time you c
(i.e. modify `BuildConfig.groovy`) Grails will do the right thing and resolve your new dependencies.

However, when you have *changing* or *dynamic* dependencies you will have to consider dependency resolu

{info} A *changing* dependency is one whose version number does not change, but its contents do (like a S
dependency is one that is defined as one of many possible options (like a dependency with a version ra
number like `latest.integration`). {info}

Both *changing* and *dynamic* dependencies are influenced by the environment. With caching active, any ch
are effectively ignored. For example, your project may not automatically fetch the very latest version of a
`latest.integration`. Or if you declare a SNAPSHOT dependency, you may not automatically get the
the server.

To ensure you have the correct version of a *changing* or *dynamic* dependency in your project, you can:

- clean the project

- run the `refresh-dependencies` command

- run *any* command with the `--refresh-dependencies` switch; or

- make a change to `BuildConfig.groovy`

If you have your CI builds configured to not perform clean builds, it may be worth adding the `--ref`
switch to the command you use to build your projects.

6 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. So ty

```
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 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 guid
> information about individual commands

It's often useful to provide custom arguments to the JVM when running Grails commands, in particular w
may for example want to set a higher maximum heap size. The Grails command will use any JVM option
JAVA_OPTS environment variable, but you can also specify a Grails-specific environment variable too:

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

non-interactive mode
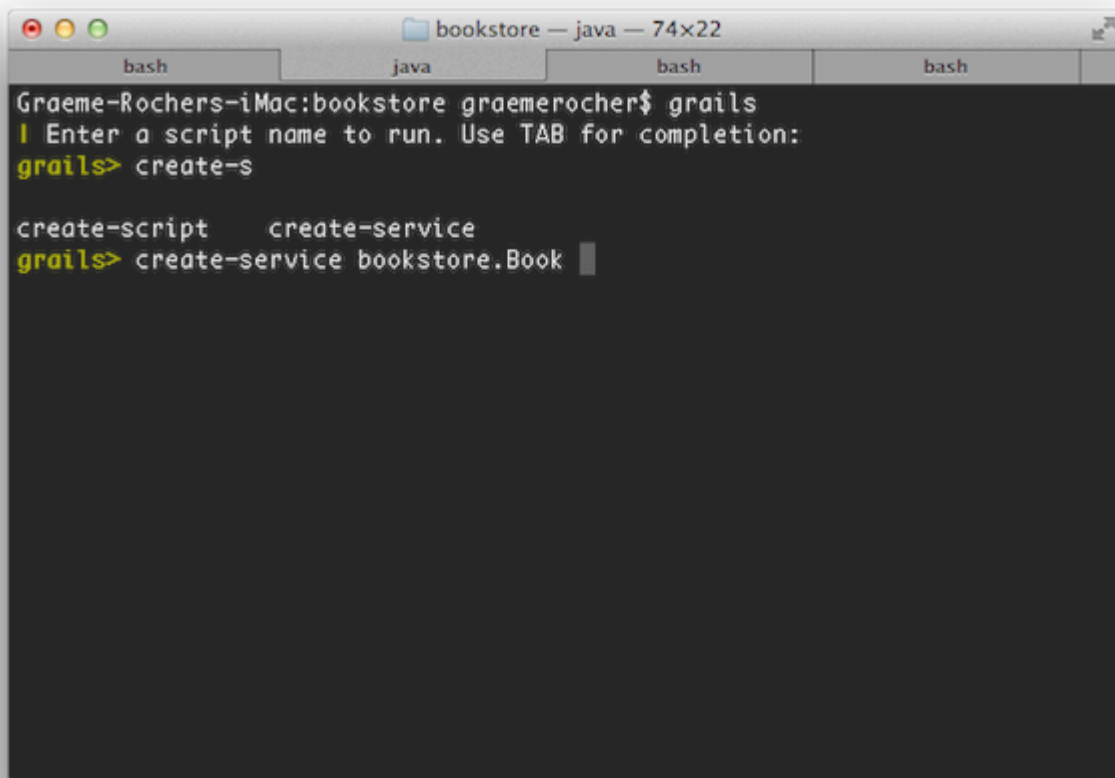
When you run a script manually and it prompts you for information, you can answer the questions and co
But when you run a script as part of an automated process, for example a continuous integration build
"answer" the questions. So you can pass the --non-interactive switch to the script command to
default answer for any questions, for example whether to install a missing plugin.

For example:

```
grails war --non-interactive
```
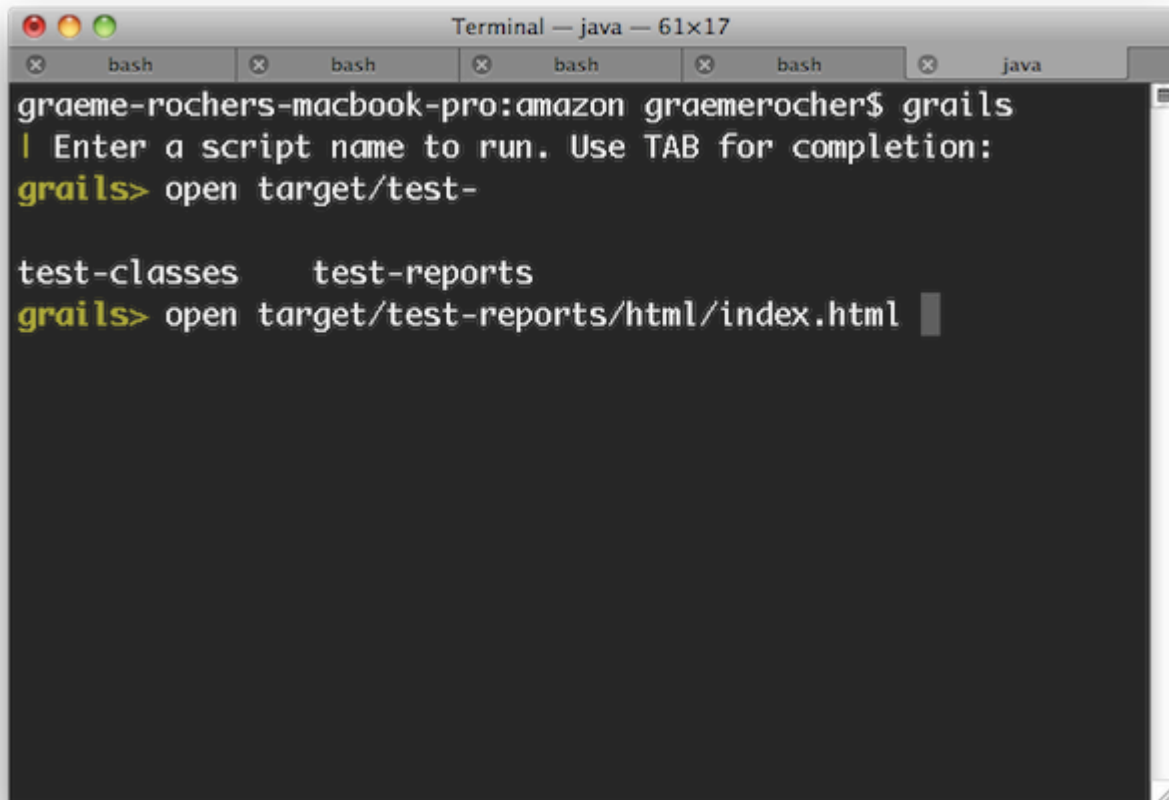
6.1 Interactive Mode

Interactive mode is the a feature of the Grails command line which keeps the JVM running and allows commands. To activate interactive mode type 'grails' at the command line and then use TAB completion to

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

Even better, the `open` command understands the logical aliases 'test-report' and 'dep-report', which will and dependency reports respectively. In other words, to open the test report in a browser simply execute You can even open multiple files at once: `open test-report test/unit/MyTests.groovy` report in your browser and the `MyTests.groovy` source file in your text editor.

TAB completion also works for class names after the `create-*` commands:
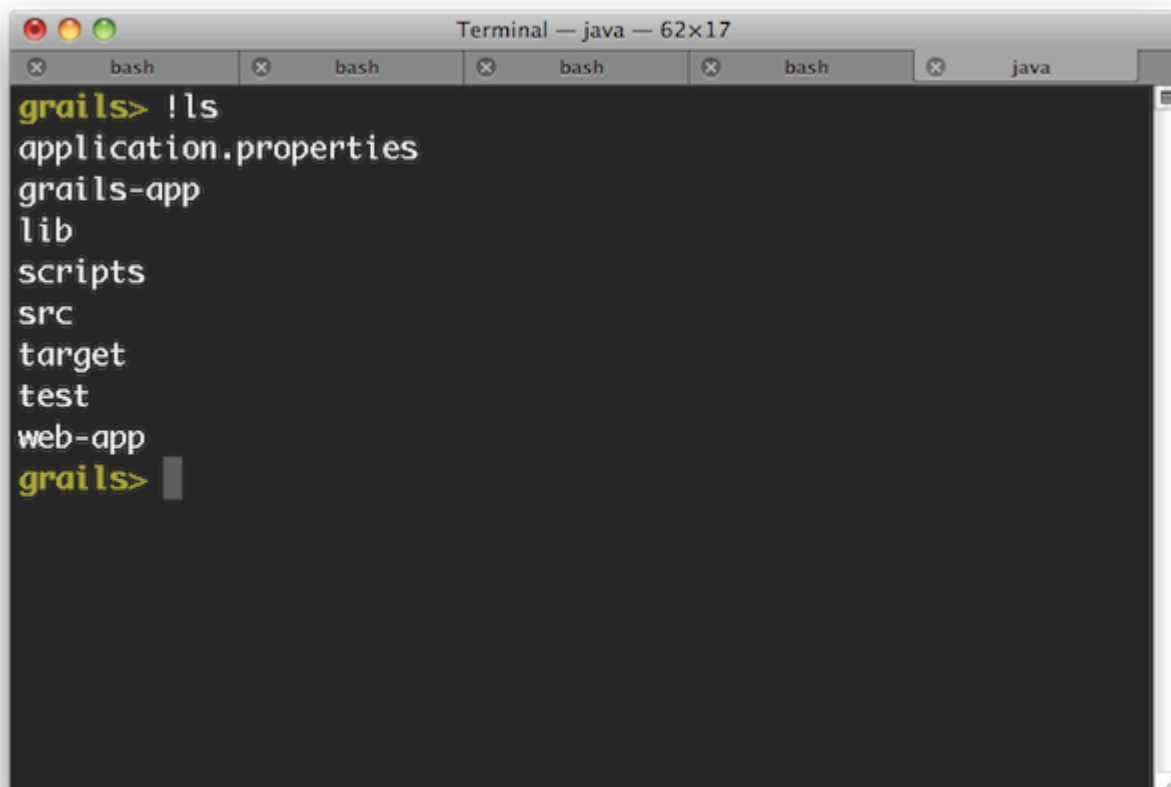
If you need to run an external process whilst interactive mode is running you can do so by starting the com

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

The `stop-app` command will stop an application that has been run with the `run-app` command.

To exit interactive mode enter the `exit` command. Note that if the Grails application has been run with r
terminate when the interactive mode console exits because the JVM will be terminated. An exception
application were running in forked mode which means the application is running in a different JVM. In
will be left running after the interactive mode console terminates. If you want to exit interactive mode and
is running in forked mode, use the `quit` command. The `quit` command will stop the running ap
interactive mode.

6.2 Forked Execution

Forked Execution

Since Grails 2.3, the `run-app`, `run-war`, `test-app` and `console` commands are now executed in
isolate the build classpath from the runtime classpath.

Forked execution is configured via the `grails-app/conf/BuildConfig.groovy` file. The
configuration:

```
grails.project.fork = [
    test: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256, daemon:true]
settings for the test-app JVM
    run: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256], // configure
run-app JVM
    war: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256], // configure
run-war JVM
    console: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256]// configu
the Console UI JVM
]
```

The memory requirements of the forked JVM can be tweaked as per the requirements of the application.

Forked Test Execution

When running the test-app command, a separate JVM is launched to execute this tests. This will have a no
of execution of the tests when running the command directly:

```
grails test-app
```

To mitigate this, Grails 2.3 and above include a feature that launches a background JVM on standby
interactive mode. In other words, running `test-app` from interactive mode will result in faster test execu

137

```
$ grails
$ grails> test-app
```

It is recommended that forked execution is used for tests, however it does require modern hardware d
JVMs. You can therefore disable forked execution by setting the grails.project.fork.test settin

```
forkConfig = [maxMemory: 1024, minMemory: 64, debug: false, maxPerm: 256]
grails.project.fork = [
    test: false,
    …
]
```

Using the Test Runner Daemon to Speed-up Test Execution

The default configuration for the testing is to activate a daemon to run tests using the daemon argument:

```
grails.project.fork = [
    test: [maxMemory: 768, minMemory: 64, debug: false, maxPerm: 256, daemon:true]
settings for the test-app JVM
    ...
```

This only works in interactive mode, so if you start Grails with the 'grails' command and then using test-
used:

```
$ grails
$ grails> test-app
```

This has the effect of speeding-up test executions times. You can disable the daemon by setting daemon t
becomes unresponsive you can restart it with restart-daemon:

```
$ grails> restart-daemon
```

Debugging and Forked Execution (--debug vs --debug-fork)

An important consideration when using forked execution is that the `debug` argument will allow a remote
to the build JVM but not the JVM that your application is running in. To debug your application you shou
argument:

```
grails test-app --debug-fork
```

Or for run-app:

```
grails run-app --debug-fork
```

Forked Tomcat Execution

Grails 2.2 and above support forked JVM execution of the Tomcat container in development mode.
including:

- Reduced memory consumption, since the Grails build system can exit

- Isolation of the build classpath from the runtime classpath

- The ability to deploy other Grails/Spring applications in parallels without conflicting dependencies

To enable forked execution you can set the `grails.project.fork.run` property to `true`:

```
grails.project.fork.run=true
```

139

Then just us the regular `run-app` command as per normal. Note that in forked mode the `grails` proce
container running in the background. To stop the server there is a new `stop-app` command:

```
grails stop-app
```

To customize the JVM arguments passed to the forked JVM you can specify a list instead:

```
grails.project.fork.run= [maxMemory:1024, minMemory:64, debug:false, maxPerm:256,
['-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005']]
```

Auto-deploying additional WAR files in Forked Mode

Since forked execution isolates classpaths more effectively than embedded execution you can deploy addit
other Grails or Spring applications) to the container.

The easiest way to do so is to drop the WAR files into the `src/autodeploy` directory (if it doesn't exis

You can customize the location of the autodeploy directory by specifying an alternative location in `Build`

```
grails.project.autodeploy.dir="/path/to/my/war/files"
```

Customizing the Forked Tomcat instance

If you want to programmatically customize the forked Tomcat instance you can do so by impler
`org.grails.plugins.tomcat.ForkedTomcatCustomizer` which provides a method with the

```
void customize(Tomcat tomcat) {
 // your code here
}
```

6.3 Creating Gant Scripts

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

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

Will create a script called `scripts/CompileSources.groovy`. A Gant script itself is similar to
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 of `groovy.util.An`
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

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

The default target

In the example above, we specified a target with the explicit name "default". This is one way of defining
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
setDefaultTarget() at the end of the script in this example, it can go anywhere as long as it comes
("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 m
case. One thing we would say is that if you want to allow other scripts to call your "default" target, you sho
script that doesn't have a default target at all. We'll talk some more about this in the next section.

6.4 Re-using Grails scripts

Grails ships with a lot of command line functionality out of the box that you may find useful in your own s
line reference in the reference guide for info on all the commands). Of particular use are the compile, pack

The bootstrap script for example lets you bootstrap a Spring ApplicationContext instance to get access to
(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 upon or [...] they had been defined in the current script. The mechanism for doing this is the `includeTargets` pro[...] 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 th[...]

Core Grails targets

As you saw in the example at the beginning of this section, you use neither the File- nor the [...] `includeTargets` when including core Grails targets. Instead, you should use the special `grailsSc`[...] provided by the Grails command launcher (note that this is not available in normal Gant scripts, just Grails[...]

The syntax for the `grailsScript()` method is pretty straightforward: simply pass it the name of the [...] 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 oth[...] _GrailsProxy, so you usually don't have to include it explicitly.
_GrailsEvents	Include this to fire events. Adds an `event(String eventName, List` [...] included by almost all other Grails scripts.
_GrailsClasspath	Configures compilation, test, and runtime classpaths. If you want to use or play with [...] 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 script to [...] your proxy.
_GrailsArgParsing	Provides a `parseArguments` target that does what it says on the tin: parses the ar[...] user when they run your script. Adds them to the `argsMap` property.
_GrailsTest	Contains all the shared test code. Useful if you want to add any extra tests.
_GrailsRun	Provides all you need to run the application in the configured servlet container, eit[...] `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 fin[...] 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' [...] script is *internal* , or in other words that it has not corresponding "command". So you can't run "grails _gr[...] 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 approach in your targets into an internal script that can be easily shared, and provide simple command scripts that arguments and delegate to the targets in the internal script. For example if you have a script that runs some split it like this:

```
./scripts/FunctionalTests.groovy:

includeTargets << new File("${basedir}/scripts/_FunctionalTests.groovy")

target(default: "Runs the functional tests for this project.") {
    depends(runFunctionalTests)
}

./scripts/_FunctionalTests.groovy:

includeTargets << grailsScript("_GrailsTest")

target(runFunctionalTests: "Run functional tests.") {
    depends(...)
    …
}
```

Here are a few general guidelines on writing scripts:

- Split scripts into a "command" script and an internal one.

- Put the bulk of the implementation in the internal script.

- Put argument parsing into the "command" script.

- To pass arguments to a target, create some script variables and initialise them before calling the target

- Avoid name clashes by using closures assigned to script variables instead of targets. You can then pa closures.

6.5 Hooking into Events

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

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

Defining event handlers

Event handlers are defined in scripts called _Events.groovy. Grails searches for these scripts in the fo

- 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 re performed automatically by Grails, so you just need to declare them in the relevant _Events.groovy fi

Event handlers are blocks defined in _Events.groovy, with a name beginning with "event". The follo in your /scripts directory to demonstrate the feature:

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

eventStatusUpdate = { msg ->
    println msg
}

eventStatusFinal = { msg ->
    println msg
}
```

You can see here the three handlers eventCreatedArtefact, eventStatusUpdate, event provides some standard events, which are documented in the command line reference guide. For examp 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
StatusFinal	message	Passed a string indicating the final script status message, target, even if the target does not exit the scripting environme
CreatedArtefact	artefactType,artefactName	Called when a create-xxxx script has completed and created
CreatedFile	fileName	Called whenever a project source filed is created, not in managed by Grails
Exiting	returnCode	Called when the scripting environment is about to exit cleanl
PluginInstalled	pluginName	Called after a plugin has been installed
CompileStart	kind	Called when compilation starts, passing the kind of compile
CompileEnd	kind	Called when compilation is finished, passing the kind of com
DocStart	kind	Called when documentation generation is about to start - java
DocEnd	kind	Called when documentation generation has ended - javadoc
SetClasspath	rootLoader	Called during classpath initialization so plugins can augr rootLoader.addURL(...). Note that this augments the classpa loaded so you cannot use this to load a class that your even 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 t started and after web.xml is generated)

6.6 Customising the build

Grails is most definitely an opinionated framework and it prefers convention to configuration, but this configure it. In this section, we look at how you can influence and modify the standard Grails build.

The defaults

The core of the Grails build configuration is the grails.util.BuildSettings class, which cont information. It controls where classes are compiled to, what dependencies the application has, and other su

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.
config	The configuration settings defined in the project's `BuildConfig.groovy` file. same way as you access runtime settings: `grailsSettings.config.foo.b`
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 `BuildSettings` is available to your scripts as the `grailsSettings` script variable. You can also ac using the `grails.util.BuildSettingsHolder` class, but this isn't recommended.

Overriding the defaults

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

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

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

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

Note that the default values take account of the property values they depend on, so setting the project w
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 proper
precedence over the `BuildConfig.groovy` file, which in turn takes precedence over the default values

The `BuildConfig.groovy` file is a sibling of `grails-app/conf/Config.groovy` - the form
only affect the build, whereas the latter contains those that affect the application at runtime. It's not limited
table either: you will find build configuration options dotted around the documentation, such as ones for sp
embedded servlet container runs on or for determining what files get packaged in the WAR file.

Available build settings

Name	Description
grails.server.port.http	Port to run the embedded servlet container on ("run-app" and "run-war"). Ir
grails.server.port.https	Port to run the embedded servlet container on for HTTPS ("run-app --https Integer.
grails.config.base.webXml	Path to a custom web.xml file to use for the application (alternative to using
grails.compiler.dependencies	Legacy approach to adding extra dependencies to the compiler classp containing "fileset()" entries. These entries will be processed by an `AntBu` the Groovy form of the corresponding XML elements in an Ant build file "$basedir/lib", includes: "**/*.class").
grails.testing.patterns	A list of Ant path patterns that let you control which files are included i should not include the test case suffix, which is set by the next property.
grails.testing.nameSuffix	By default, tests are assumed to have a suffix of "Tests". You can change it setting this option. For example, another common suffix is "Test".
grails.project.war.file	A string containing the file path of the generated WAR file, along witl extension). For example, "target/my-app.war".
grails.war.dependencies	A closure containing "fileset()" entries that allows you complete contro WAR's "WEB-INF/lib" directory.
grails.war.copyToWebApp	A closure containing "fileset()" entries that allows you complete control ov of the WAR. It overrides the default behaviour of including everything und
grails.war.resources	A closure that takes the location of the staging directory as its first argume tasks to do anything you like. It is typically used to remove files from the that directory is jar'd up into a WAR.
grails.project.web.xml	The location to generate Grails' web.xml to

Reloading Agent Cache Directory

Grails uses an agent based reloading system in the development environment that allows source code chan
the application is running. This reloading agent caches information needed to carry out the reloading ef
information is stored under `<USER_HOME_DIR>/.grails/.slcache/`. The `GRAILS_AGENT_C`
variable may be assigned a value to cause this cache information to be stored somewhere else. Note that th
environment variable, not a JVM system property or a property which may be defined in `BuildConfi`
must be defined as an environment variable because the agent cache directory must be configured very
process, before any Grails code is executed.

6.7 Ant and Maven

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

Maven Integration

Grails provides integration with [Maven 3](#) with a Maven plugin.

Preparation

In order to use the Maven plugin, all you need is Maven 3 installed and set up. This is because **you n
Grails separately to use it with Maven!**

> ⚠️ The Maven 3 integration for Grails has been designed and tested for Maven 3.1.0 and above
> work with earlier versions.

Creating a Grails Maven Project

Using the `create-pom` command you can generate a valid Maven `pom.xml` file for any existing G
presents an example:

```
$ grails create-app myapp
$ cd myapp
$ grails create-pom com.mycompany
```

The `create-pom` command expects a group id as an argument. The name and the version
`application.properties` of the application. The Maven plugin will keep the version in the po
version in `application.properties`.

The following standard Maven commands are then possible:

- `compile` - Compiles a Grails project

- `package` - Builds a WAR file from the Grails project.

- `install` - Builds a WAR file (or plugin zip/jar if a plugin) and installs it into your local Maven cacl

- `test` - Runs the tests of a Grails project

- `clean` - Cleans the Grails project

Other standard Maven commands will likely work too.

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-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

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

Defining Plugin Dependencies

All Grails plugins are published to a standard Maven repository located at . When using the Maven pl
ensure that this repository is declared in your list of remote repositories:

```xml
<repository>
    <id>grails-plugins</id>
    <name>grails-plugins</name>
    <url>http://repo.grails.org/grails/plugins</url>
</repository>
```

With this done you can declare plugin dependencies within your `pom.xml` file:

```xml
<dependency>
    <groupId>org.grails.plugins</groupId>
    <artifactId>database-migration</artifactId>
    <version>1.1</version>
    <scope>runtime</scope>
    <type>zip</type>
</dependency>
```

Note that the `type` element must be set to `zip`.

Specifying the Grails Version to Use

The 2.4.0 version of the Maven plugin works with different versions of Grails. By default it tries to auto-de
to use from the `grails-dependencies` dependency definition found in the `pom.xml`:

```xml
<dependency>
    <groupId>org.grails</groupId>
    <artifactId>grails-dependencies</artifactId>
    <version>2.4.0</version>
</dependency>
```

If you change the version of `grails-dependencies` then a different version of Grails will be used
explicitly define the Grails version to be used in the plugin `configuration`:

```
<plugin>
    <groupId>org.grails</groupId>
    <artifactId>grails-maven-plugin</artifactId>
    <version>2.4.0</version>
    <configuration>
        <grailsVersion>2.4.0</grailsVersion>
    </configuration>
    <extensions>true</extensions>
</plugin>
```

Debugging Grails Execution

The Maven plugin will run Grails commands in a separate process, meaning that the Grails process occup Maven process.

To debug the Grails process you need to configure the `forkDebug` option in the plugin's `configurati`

```
<plugin>
    <groupId>org.grails</groupId>
    <artifactId>grails-maven-plugin</artifactId>
    <version>2.4.0</version>
    <configuration>
        <forkDebug>true</forkDebug>
    </configuration>
    <extensions>true</extensions>
</plugin>
```

With this configuration in place the JVM executed in Maven will load in debug mode.

If you need to customize the memory of the forked process the following elements are available:

- `forkMaxMemory` - The maximum amount of heap (default 1024)

- `forkMinMemory` - The minimum amount of heap (default 512)

- `forkPermGen` - The amount of permgen (default 256)

Multi Module Maven Builds

The Maven plugin can be used to power multi-module Grails builds. The easiest way to set t `create-multi-project-build` command:

```
$ grails create-app myapp
$ grails create-plugin plugin1
$ grails create-plugin plugin2
$ grails create-multi-project-build org.mycompany:parent:1.0
```

Running `mvn install` will build all projects together. To enable the 'grails' command to read the BuildConfig.groovy to use the POM and resolve dependencies from your Maven local cache:

```
grails.project.dependency.resolution = {
    …
    pom true
    repositories {
        …
        mavenLocal()
    }
}
```

By reading the `pom.xml` file you can do an initial `mvn install` from the parent project to build all into your local maven cache and then `cd` into your project and use the regular `grails run-ap` application. All previously built plugins will be resolved from the local Maven cache.

Adding Grails commands to phases

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

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

```
    <plugin>
        <groupId>org.grails</groupId>
        <artifactId>grails-maven-plugin</artifactId>
        <version>2.4.0</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. Simp
command as the `command` system property, and optionally specify the arguments with the `args` property

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

Raising issues

If you come across any problems with the Maven integration, please raise a [JIRA issue](#).

Ant Integration

When you create a Grails application with the [create-app](#) command, Grails doesn't automatically create
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 will au the requisite Grails JAR files and other dependencies on demand. You don't even have to install Grails loc it particularly useful for continuous integration systems 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 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" J. to be on the classpath at least. You should also include the "groovy-all" JAR. With the task defined, yo 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
classpathref	Classpath to load Grails from. Must include the "grails-bootstrap" artifact and should include "grails-scripts".	Yes, unless a classpa
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. Default
environment	The Grails environment to run the script in.	No. Default
includeRuntimeClasspath	Advanced setting: adds the application's runtime classpath to the build classpath if true.	No. Default

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 @compileClasspath.

- `testClasspath` - Classpath used to compile and run the tests. Typically includes everything in ru

How you populate these paths is up to you. If you use the `home` attribute and put your own dependencies i you don't even need to use any of them. For an example of their use, take a look at the generated Ant build

6.8 Grails Wrapper

The Grails Wrapper allows a Grails application to built without having to install Grails and config environment variable. The wrapper includes a small shell script and a couple of small bootstrap jar file checked in to source code control along with the rest of the project. The first time the wrapper is execu configure a Grails installation. This wrapper makes it more simple to setup a development environment, upgrades to future versions of Grails. When the application is upgraded to the next version of Grails, the checked in to the source code control system and the next time developers update their workspace and r automatically be using the correct version of Grails.

Generating The Wrapper

The [wrapper](#) command can be used to generate the wrapper shell scripts and supporting jar files. Execute the top of an existing Grails project.

```
grails wrapper
```

In order to do this of course Grails must be installed and configured. This is only a requirement for bo Once the wrapper is generated there is no need to have a Grails installation configured in order to use the v

See the [wrapper](#) command documentation for details about command line arguments.

By default the wrapper command will generate a `grailsw` shell script and `grailsw.bat` batch file at addition to those, a `wrapper/` directory (the name of the directory is configurable via command line op contains some support files which are necessary to run the wrapper. All of these files should be chec control system along with the rest of the project. This allows developers to check the project out of immediately start using the wrapper to execute Grails commands without having to install and configure G

Using The Wrapper

The wrapper script accepts all of the same arguments as the normal grails command.

```
./grailsw create-domain-class com.demo.Person
./grailsw run-app
./grailsw test-app unit:

etc...
```

7 Object Relational Mapping (GORM)

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

GORM is Grails' object relational mapping (ORM) implementation. Under the hood it uses Hibernate 3 (a
open source ORM solution) and thanks to the dynamic nature of Groovy with its static and dynamic
convention of Grails, there is far less configuration involved in creating Grails domain classes.

You can also write Grails domain classes in Java. See the section on Hibernate Integration for how to wri
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()
```

7.1 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
> name as the package name.

This will create a class at the location `grails-app/domain/helloworld/Person.groovy` such

```
package helloworld

class Person {
}
```

⚠ If you have the `dbCreate` property set to "update", "create" or "create-drop" on your [
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 Spring Application

7.1.1 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 datab

object in a read-only state by using the [read](#) method:

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

In this case the underlying Hibernate engine will not do any dirty checking and the object will not be p

explicitly call the [save](#) method then the object is placed back into a read-write state.

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 the pr

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()
```

7.2 Domain Modelling in GORM

When building Grails applications you have to consider the problem domain you are trying to solve. I building an [Amazon](#)-style bookstore you would be thinking about books, authors, customers and publisher

These are modeled in GORM as Groovy classes, so a `Book` class may have a title, a release date, an ISBN 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 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 underscores. For example releaseDate maps onto a column release_date. The SQL types are au types, but can be customized with [Constraints](#) or the [ORM DSL](#).

7.2.1 Association in GORM

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

7.2.1.1 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 do 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 relatio the other side as follows (and see the section on controlling the ends of the association just below):

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 th
attach a `Nose` instance to it and when we save or delete the `Face` instance, GORM will save or delete th
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 erro
:

```
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 example A, so in this case the
stored in the `nose` table inside a column called `face_id`. Also, `hasOne` only works with bidirectional re

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
    }
}
```

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

Controlling the ends of the association

Occasionally you may find yourself with domain classes that have multiple properties of the same t
self-referential, i.e. the association property has the same type as the domain class it's in. Such situat
because Grails may guess incorrectly the type of the association. Consider this simple class:

```
class Person {
    String name
    Person parent

static belongsTo = [ supervisor: Person ]

static constraints = { supervisor nullable: true }
}
```

As far as Grails is concerned, the `parent` and `supervisor` properties are two directions of the same a
set the `parent` property on a `Person` instance, Grails will automatically set the `supervisor` proper
instance. This may be what you want, but if you look at the class, what we in fact have are two unidirection

To guide Grails to the correct mapping, you can tell it that a particular association is unidirectional
property:

```
class Person {
    String name
    Person parent

static belongsTo = [ supervisor: Person ]

static mappedBy = [ supervisor: "none", parent: "none" ]

static constraints = { supervisor nullable: true }
}
```

You can also replace "none" with any property name of the target class. And of course this works for no
not just self-referential ones. Nor is the `mappedBy` property limited to many-to-one and one-to-one asso
one-to-many and many-to-many associations as you'll see in the next section.

> ⛔ If you have a property called "none" on your domain class, this approach won't work cur
> "none" property will be treated as the reverse direction of the association (or the "back i
> Fortunately, "none" is not a common domain class property name.

7.2.1.2 One-to-many

A one-to-many relationship is when one class, example `Author`, has many instances of another class, ex
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 wit

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

Grails will automatically inject a property of type `java.util.Set` into the domain class based on the
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 wi
> 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

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

```
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 mapped
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"]
}
```

167

```
class Flight {
    Airport departureAirport
    Airport destinationAirport
}
```

7.2.1.3 Many-to-many

Grails supports many-to-many relationships by defining a `hasMany` on both sides of the relationship an 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 relationsh takes responsibility for persisting the relationship and is the only side that can cascade saves across.

For example this will work and cascade saves:

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

However 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 respon relationship.

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

7.2.1.4 Basic Collection Types

As well as associations between different domain classes, GORM also supports mapping of basic collectio following class creates a nicknames association that is a Set of String instances:

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

GORM will map an association like the above using a join table. You can alter various aspects of how 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

169

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

7.2.2 Composition in GORM

As well as [association](), Grails supports the notion of composition. In this case instead of mapping class
class can be "embedded" within the current table. For example:

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

class Address {
    String number
    String code
}
```

The resulting mapping would looking like this:

Person Table

id	home_address _number	home_address _code	work_address _number	work_address _code
1	47	343432	67	43545

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

7.2.3 Inheritance in GORM

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

```
class Content {
    String author
}
```

```
class BlogEntry extends Content {
    URL url
}
```

```
class Book extends Content {
    String ISBN
}
```

```
class PodCast extends Content {
    byte[] audioStream
}
```

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

Considerations

At the database level Grails by default uses table-per-hierarchy mapping with a discriminator column cal
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 with i
alternative is to use table-per-subclass which can be enabled with the <u>ORM DSL</u>

However, excessive use of inheritance and table-per-subclass can result in poor query performance due
queries. In general our advice is if you're going to use inheritance, don't abuse it and don't make your i
deep.

Polymorphic Queries

171

The upshot of inheritance is that you get the ability to polymorphically query. For example using the <u>list</u> 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
```

7.2.4 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 unorder 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 uniqueness but not order 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 implement jav 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 will be
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
collection type as a `List`:

```
class Author {

List books

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

In this case when you add new elements to the books collection the order is retained in a sequential list in
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 ind
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 v
`org.hibernate.HibernateException`: null index column for collection):

173

```
// 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 Hibern
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 mappe
load of all existing instances from the database, so this approach will perform better and require less memo
`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 **must**

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
the entire associations from the database. If you have a large numbers of entries in the association this
performance.

The same behavior is required for `List` types, since Hibernate needs to load the entire association to main
recommended that if you anticipate a large numbers of records in the association that you make the associa
the link can be created on the inverse side. For example consider the following code:

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

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

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

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

7.3 Persistence Basics

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

Grails automatically binds a Hibernate session to the currently executing request. This lets you use the [sa](#) 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 necessarily perform any SQL operations **at that point**. Hibernate batches up SQL statements and executes often at the end of the request when flushing and closing the session. This is typically done for you autom manages your Hibernate session.

Hibernate caches database updates where possible, only actually pushing the changes when it knows th when a flush is triggered programmatically. One common case where Hibernate will flush cached upd queries since the cached information might be included in the query results. But as long as you're doin updates, and deletes, they'll be batched until the session is flushed. This can be a significant performance do a lot of database writes.

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

7.3.1 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 occurs when you want to control when those statements are executed or, in Hibernate terminology, when the sess 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 synchro This also lets you catch any exceptions, which is typically useful in highly concurrent scenarios involving

```
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 that va
instance will *not* be persisted to the database. By default, save() will simply return null in this case,
to throw an exception you can use the failOnError 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 secti
remember that when you are saving domain instances that have been bound with data provided by th
validation exceptions is quite high and you won't want those exceptions propagating to the end user.

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

7.3.2 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 delete
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 may database constraint, although this is normally down to a programming or schema error. The following exa a `DataIntegrityViolationException` that is thrown when you violate the database constraints:

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

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

Note that Grails does not supply a `deleteAll` method as deleting data is discouraged and can often be 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"])
```

7.3.3 Understanding Cascading Updates and Deletes

It is critical that you understand how cascading updates and deletes work when using GORM. The key `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 updates ca class to its dependant (the other side of the relationship), and for many-/one-to-one and one-to-many relat cascade.

If you *do not* define `belongsTo` then no cascades will happen and you will have to manually save each of the one-to-many, in which case saves will cascade automatically if a new instance is in a `hasMany` coll

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 update flight, hence saving the whole object graph:

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

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

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

However, if I were to remove `belongsTo` then the above cascading deletion code **would not work**. To u
a look at the summaries below that describe the default behaviour of GORM with regards to specific asso
of the GORM Gotchas series of articles to get a deeper 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 cascade st
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 ca
"SAVE-UPDATE".

Bidirectional one-to-many, no belongsTo

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

```
class B { A a }
```

In the case of a bidirectional one-to-many where the many side does not define a `belongsTo` then the "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 strateg 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](#).

7.3.4 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
iteration over the `flights` association to get the current flight's destination. In other words you get N+
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 specified as foll

```
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, althoug executed to fetch the collection. You can also use `fetch: 'join'` instead of `lazy: false`, in whi execute a single query to get the airports and their flights. This works well for single-ended associations, b with one-to-manys. Queries will work as you'd expect right up to the moment you add a limit to the numb that point, you will likely end up with fewer results than you were expecting. The reason for this is quite the problem arises from GORM using a left outer join.

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

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

Using Batch Fetching

Although eager fetching is appropriate for some cases, it is not always desirable. If you made everythi possibly load your entire database into memory resulting in performance and memory problems. An altern to use batch fetching. You can configure Hibernate to lazily fetch 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 batches of 10. For example if you had an `Airport` that had 30 flights, if you didn't configure batch f query to fetch the `Airport` and then `30` queries to fetch each flight. With batch fetching you get 1 que and 3 queries to fetch each `Flight` in batches of 10. In other words, batch fetching is an optimization of t Batch fetching 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.

7.3.5 Pessimistic and Optimistic Locking

Optimistic Locking

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

The `version` column gets read into a `version` property that contains the current versioned state of j you can access:

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

When you perform updates Hibernate will automatically check the version property against the version co if they differ will throw a [StaleObjectException](#). This will roll back the transaction if one is active.

This is useful as it allows a certain level of atomicity without resorting to pessimistic locking that ha penalty. The downside is that you have to deal with this exception if you have highly concurrent writes. 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 mer 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 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 committed. H
what we are doing is "upgrading" from a regular SELECT to a SELECT..FOR UPDATE and anothe
updated the record in between the call to get() and the call to lock().

To get around this problem you can use the static lock method that takes an id just like get:

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

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

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

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

Or using criteria:

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

7.3.6 Modification Checking

Once you have loaded and possibly modified a persistent domain class instance, it isn't straightforward
values. If you try to reload the instance using [get](get) Hibernate will return the current modified instance
Reloading using another query would trigger a flush which could cause problems if your data isn't rea
GORM provides some methods to retrieve the original values that Hibernate caches when it loads the in
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 othe
> 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 emp

```
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
    }
}
```

7.4 Querying with GORM

GORM supports a number of powerful ways to query from dynamic finders, to criteria to Hibernate language HQL. Depending on the complexity of the query you have the following options in order of flexi

- 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 so o 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 `orde` 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)
```

7.4.1 Dynamic Finders

GORM supports the concept of **dynamic finders**. A dynamic finder looks like a static method inv
themselves don't actually exist in any form at the code level.

Instead, a method is auto-magically generated using code synthesis at runtime, based on the properties of
example the `Book` class:

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

```
class Author {
    String name
}
```

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

```
def book = Book.findByTitle("The Stand")

book = Book.findByTitleLike("Harry Pot%")

book = Book.findByReleaseDateBetween(firstDate, secondDate)

book = Book.findByReleaseDateGreaterThan(someDate)

book = Book.findByTitleLikeOrReleaseDateLessThan("%Something%", someDate)
```

Method Expressions

A method expression in GORM is made up of the prefix such as findBy followed by an expression that
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 `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

- `InRange` - Between the `from` and `to` values of a Groovy Range

- `Rlike` - Performs a Regexp LIKE in MySQL or Oracle otherwise falls back to `Like`

- `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 demo 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)
```

In this case we're using And in the middle of the query to make sure both conditions are satisfied, but you

```
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 n
Or or if the number of criteria creates a very long method name, just convert the query to a [Criteria](#) or [HQ](#)

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

Pagination and Sorting

The same pagination and sorting parameters available on the list method can also be used with dynamic fi
as the final parameter:

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

7.4.2 Where Queries

The `where` method, introduced in Grails 2.0, builds on the support for Detached Criteria by providing an
checked query DSL for common queries. The `where` method is more flexible than dynamic finders, less
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. 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
or session. This means you can use the `where` method to define common queries at the class level:

```
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 immediately there are variations of the `findAll` and `find` methods to accomplish this:

```
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 Groovy

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:

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

The Groovy regex matching operators map onto like and ilike queries unless the expression on the right 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 expre

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 opera

```
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

```
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 be 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](#) instance class.

Conjunction, Disjunction and Negation

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

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

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

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

195

Property Comparison Queries

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

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

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

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 to be

```
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 the a

```
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 t
following query can be used:

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

The following table lists the possible subqueries:

197

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 contain

```
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 exam
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 tab

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
> 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 execute
batch updates and deletes. For example, the following query will update all people with the surname
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
> associations) are not allowed.

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

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

7.4.3 Criteria

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

Criteria can be used either with the createCriteria or withCriteria methods. The builder uses Hibernate's C
this builder map the static methods found in the Restrictions class of the Hibernate Criteria API. For exam

```
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 `Ac`
`Transaction` objects:

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

We can query this association by using the property name `transactions` 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
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 recer
days.

Querying with Projections

Projections may be used to customise the results. Define a "projections" node within the criteria builde
There are equivalent methods within the projections node to the methods found in the Hibernate Projection

```
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 will b

SQL Projections

The criteria DSL provides access to Hibernate's SQL projection API.

```
// Box is a domain class…
class Box {
    int width
    int height
}
```

```
// Use SQL projections to retrieve the perimeter and area of all of the Box insta
def c = Box.createCriteria()

def results = c.list {
    projections {
      sqlProjection '(2 * (width + height)) as perimeter, (width * height) as are
'area'], [INTEGER, INTEGER]
    }
}
```

The first argument to the `sqlProjection` method is the SQL which defines the projections. The sec
Strings which represent column aliases corresponding to the projected values expressed in the SQL. The t
`org.hibernate.type.Type` instances which correspond to the projected values expressed in the SQ
`org.hibernate.type.Type` objects but constants like INTEGER, LONG, FLOAT etc. are prov
correspond to all of the types defined in `org.hibernate.type.StandardBasicTypes`.

Consider that the following table represents the data in the BOX table.

width	height
2	7
2	8
2	9
4	9

The query above would return results like this:

```
[[18, 14], [20, 16], [22, 18], [26, 36]]
```

Each of the inner lists contains the 2 projected values for each `Box`, perimeter and area.

> ⚠ Note that if there are other references in scope wherever your criteria query is expressed that
> that conflict with any of the type constants described above, the code in your criteria will re
> references, not the type constants provided by the DSL. In the unlikely event of that happeni
> disambiguate the conflict by referring to the fully qualified Hibernate type. For
> `StandardBasicTypes.INTEGER` instead of `INTEGER`.

If only 1 value is being projected, the alias and the type do not need to be included in a list.

```
def results = c.list {
    projections {
      sqlProjection 'sum(width * height) as totalArea', 'totalArea', INTEGER
    }
}
```

That query would return a single result with the value of 84 as the total area of all of the `Box` instances.

The DSL supports grouped projections with the `sqlGroupProjection` method.

```
def results = c.list {
    projections {
        sqlGroupProjection 'width, sum(height) as combinedHeightsForThisWidth', '
'combinedHeightsForThisWidth'], [INTEGER, INTEGER]
    }
}
```

The first argument to the `sqlGroupProjection` method is the SQL which defines the projection
represents the group by clause that should be part of the query. That string may be single column name or
column names. The third argument is a list of Strings which represent column aliases corresponding
expressed in the SQL. The fourth argument is a list of `org.hibernate.type.Type` instances
projected values expressed in the SQL.

The query above is projecting the combined heights of boxes grouped by width and would return results th

```
[[2, 24], [4, 9]]
```

Each of the inner lists contains 2 values. The first value is a box width and the second value is the sum o
boxes which have that width.

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 dynamic restri

```
def c = Person.createCriteria()

def peopleWithShortFirstNames = c.list {
    sqlRestriction "char_length(first_name) < ? AND char_length(first_name) > ?",
minValue]
}
```

⚠️ Note that the parameter there is SQL. The `first_name` attribute referenced in the example i
persistence model, not the object model like in HQL queries. The `Person` property named f.
is mapped to the `first_name` column in the database and you must refer to th
`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*
> *pattern is very similar to the JDBC PreparedStatement / ResultSet pattern and the semantics*
> *interface are similar to the similarly named methods on ResultSet.*

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

Setting properties in the Criteria instance

If a node within the builder tree doesn't match a particular criterion it will attempt to set a property on the C
allows full access to all the properties in this class. This example calls `setMaxResults` and `setFirst`
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 the
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 asse
instances. It's probably best not to use this for one-to-many associations though, because you will most lik
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
`maxResults` option.

> ⚠️ `fetchMode` and `join` are general settings of the query and can only be specified at the to
> 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, those assoc
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 been expli

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 on 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 listing only distinct entities and is equivalent to DISTINCT_ROOT_ENTITY of the CriteriaS
count	Returns the number of matching rows.

Combining Criteria

You can combine multiple criteria closures in the following way:

```
def emeaCriteria = {
    eq "region", "EMEA"
}
def results = Airport.withCriteria {
    emeaCriteria.delegate = delegate
    emeaCriteria()
    flights {
        like "number", "BA%"
    }
}
```

This technique requires that each criteria must refer to the same domain class (i.e. `Airport`). A more fl
Detached Criteria, as described in the following section.

7.4.4 Detached Criteria

Detached Criteria are criteria queries that are not associated with any given database session/connection
2.0, Detached Criteria queries have many uses including allowing you to create common reusable
subqueries and execute batch updates/deletes.

Building Detached Criteria Queries

The primary point of entry for using the Detached Criteria is the `grails.gorm.DetachedCriter`
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 <u>where</u> queries or crit
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 instea
other words, you have to use the return value of the `build` method to obtain the mutated 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.
query has been constructed then there are a number of useful query methods which are summarized in the t

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 can

```
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 G: 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. Fo find all people who are older than the average age the following query will accomplish that:

```
def results = Person.withCriteria {
    gt "age", new DetachedCriteria(Person).build {
        projections {
            avg "age"
        }
    }
    order "firstName"
}
```

211

Notice that in this case the subquery class is the same as the original criteria query class (i.e. `Person`) an 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). multiple results then there are different criteria methods that need to be used to compare the result. Fo 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 c
following query will update all people with the surname "Simpson" to have the surname "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
> 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()
```

7.4.5 Hibernate Query Language (HQL)

GORM classes also support Hibernate's query language HQL, a very complete reference for which can b
documentation of the Hibernate documentation.

GORM provides a number of methods that work with HQL including find, findAll and executeQuery. A
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

213

```
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 the pagin
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])
```

7.5 Advanced GORM Features

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

7.5.1 Events and Auto Timestamping

GORM supports the registration of events as methods that get fired when certain events occurs such as del
The following is a list of supported events:

- `beforeInsert` - Executed before an object is initially persisted to the database. If you return
  cancelled.

- `beforeUpdate` - Executed before an object is updated. If you return false, the update will be cance

- `beforeDelete` - Executed before an object is deleted. If you return false, the delete will be cancell

- `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 method with your domain class.

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

Event types

The beforeInsert event

Fired before an object is saved to the database

```groovy
class Person {
    private static final Date NULL_DATE = new Date(0)

String firstName
    String lastName
    Date signupDate = NULL_DATE

def beforeInsert() {
        if (signupDate == NULL_DATE) {
            signupDate = new Date()
        }
    }
}
```

The beforeUpdate event

Fired before an existing object is updated

```groovy
class Person {

def securityService

String firstName
    String lastName
    String lastUpdatedBy

static constraints = {
        lastUpdatedBy nullable: true
    }

def beforeUpdate() {
        lastUpdatedBy = securityService.currentAuthenticatedUsername()
    }
}
```

The beforeDelete event

Fired before an object is deleted.

```
class Person {
    String name

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

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

Fortunately the `withNewSession` method lets you share the same transactional JDBC connection ev 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.

> ⚠️ Validation may run more often than you think. It is triggered by the `validate()` an
> methods as you'd expect, but it is also typically triggered just before the view is rendered
> when writing `beforeValidate()` implementations, make sure that they can handle b
> multiple times with the same property values.

GORM supports an overloaded version of `beforeValidate` which accepts a `List` parameter which m the properties which are about to be validated. This version of `beforeValidate` will be called when has been invoked and passed a `List` of property names as an argument.

217

```
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 metho
> validate method is being invoked with no arguments, not a List that includes all of t
> names.

Either or both versions of beforeValidate may be defined in a domain class. GORM will prefer the L
passed to validate but will fall back on the no-arg version if the List version does not exist. Likewis
no-arg version if no arguments are passed to validate but will fall back on the List version if the no-a
In that case, null is passed to 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 even
Hibernate and also works for other persistence plugins such as the [MongoDB plugin for GORM](#).

To use this API you need to subclass `AbstractPersistenceEventLister`
org.grails.datastore.mapping.engine.event) and implement the methods `onPersistenceEvent` and s
You also must provide a reference to the datastore to the listener. The simplest possible implementation ca

219

```
public MyPersistenceListener(final Datastore datastore) {
    super(datastore)
}

@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;
    }
}

@Override
public boolean supportsEventType(Class<? extends ApplicationEvent> eventType) {
    return true
}
```

The `AbstractPersistenceEvent` class has many subclasses (`PreInsertEvent`, `PostInsert` further information specific to the event. A `cancel()` method is also provided on the event which allo update or delete operation.

Once you have created your event listener you need to register it with the `ApplicationContex` `BootStrap.groovy`:

```
def init = {
    application.mainContext.eventTriggeringInterceptor.datastores.each { k, datas
        applicationContext.addApplicationListener new MyPersistenceListener(datas
    }
}
```

or use this in a plugin:

```
def doWithApplicationContext = { applicationContext ->
    application.mainContext.eventTriggeringInterceptor.datastores.each { k, datas
        applicationContext.addApplicationListener new MyPersistenceListener(datas
    }
}
```

Hibernate Events

It is generally encouraged to use the non-Hibernate specific API described above, but if you need access to
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/resour`
`doWithSpring` closure in a plugin descriptor by registering a Spring bean named `hibernateEvent`
has one property, `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. Y
implements all of the required interfaces, or one concrete class per interface, or any combination. T
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 implements `PostInse`
`PostUpdateEventListener`, and `PostDeleteEventListener` using the following in an applic

```
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

If you define a `dateCreated` property it will be set to the current date for you when you create new in
define a `lastUpdated` property it will be automatically be updated for you when you change persistent i

If this is not the behaviour you want you can disable this feature with:

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

223

> 🚫 If you have `nullable: false` constraints on either `dateCreated` or `lastUpda`
> domain instances will fail validation - probably not what you want. Omit constraints
> properties unless you disable automatic timestamping.

7.5.2 Custom ORM Mapping

Grails domain classes can be mapped onto many legacy schemas with an Object Relational Mappin
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 t
> column names and so on. You only needs this functionality if you need to tailor the way G
> onto legacy schemas or configures caching

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

```groovy
class Person {
    …
    static mapping = {
        version false
        autoTimestamp false
    }
}
```

You can also configure global mappings in Config.groovy (or an external config file) using this setting:

```groovy
grails.gorm.default.mapping = {
    version false
    autoTimestamp false
}
```

It has the same syntax as the standard `mapping` block but it applies to all your domain classes! You
defaults within the `mapping` block of a domain class.

7.5.2.1 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 chan

```
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. Si
to a domain class persistent field, the parameter values (in this case just `"column"`) are used to config
property.

Column type

GORM supports configuration of Hibernate types with the DSL using the type attribute. This includes s
implement the Hibernate org.hibernate.usertype.UserType interface, which allows complete customi
persisted. As an example if you had a `PostCodeType` you could use it as follows:

```
class Address {

String number
    String postCode

static mapping = {
        postCode type: PostCodeType
    }
}
```

225

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

```groovy
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 usin 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 one-to-one association this is exactly the same as any regular column. For example consider the following:

```groovy
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 have changed the name of the foreign key column to `Person_Adress_Id`.

One-to-Many Mapping

With a bidirectional one-to-many you can change the foreign key column used by changing the column n the association as per the example in the previous section on one-to-one associations. However, with unidi foreign key needs to be specified on the association itself. For example given a unidirectional one-to-ma `Person` and `Address` the following code will change the foreign key in the `address` table:

```
class Person {
String firstName

static hasMany = [addresses: Address]

static mapping = {
      table 'people'
      firstName column: 'First_Name'
      addresses column: 'Person_Address_Id'
   }
}
```

If you don't want the column to be in the address table, but instead some intermediate join table you
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 this many-to-

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

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

In this case Grails will create a join table called `group_person` containing foreign keys called `pers`
referencing the `person` and `group` tables. To change the column names you can specify a column with
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'
    }
}
```

7.5.2.2 Caching Strategy

Setting up caching

[Hibernate](#) features a second-level cache with a customizable cache provider. This needs to
`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.

> ⚠ For further reading on caching and in particular Hibernate's second-level cache, refer to the
> [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 customize

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

Caching associations

229

As well as the ability to use Hibernate's second level cache to cache instances you can also cache coll
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

```
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 a
> in the previous section.

You can also cache criteria queries:

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

Cache usages

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

- `read-only` - If your application needs to read but never modify instances of a persistent class, a 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 (i.e. if it transactions would try to update the same item simultaneously) and strict transaction isolat `nonstrict-read-write` cache might be appropriate.

- `transactional` - The `transactional` cache strategy provides support for fully transactional JBoss TreeCache. Such a cache may only be used in a JTA environment and `hibernate.transaction.manager_lookup_class` in the `grails-app/conf/DataS` `hibernate` config.

7.5.2.3 Inheritance Strategies

By default GORM classes use `table-per-hierarchy` inheritance mapping. This has the disadvant have a `NOT-NULL` constraint applied to them at the database level. If you would prefer to use a t 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-hierarch`
classes.

7.5.2.4 Custom Database Identity

You can customize how GORM generates identifiers for the database using the DSL. By default GO
database mechanism for generating ids. This is by far the best approach, but there are still many sch
approaches to identity.

To deal with this Hibernate defines the concept of an id generator. You can customize the id generator and
follows:

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

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

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

Although you don't typically specify the `id` field (Grails adds it for you) you can still configure its
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'
    }
}
```

7.5.2.5 Composite Primary Keys

GORM supports the concept of composite identifiers (identifiers composed from 2 or more properties).
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. T
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 a and `hashCode` methods, using the properties in the composite key for the calculations. The `HashCodeBuilder` for convenience but it's fine to implement it yourself.

Another important consideration when using composite primary keys is associations. If for example association where the foreign keys are stored in the associated table then 2 columns will be present in the a

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` and If you wish the change the mapping of these columns then you can do so using the following technique:

```
class Address {
    Person person
    static mapping = {
        columns {
            person {
                column name: "FirstName"
                column name: "LastName"
            }
        }
    }
}
```

7.5.2.6 Database Indices

To get the best performance out of your queries it is often necessary to tailor the table index definitions domain specific and a matter of monitoring usage patterns of your queries. With GORM's DSL you can sp 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 example
`Address_Index'` will cause an error.

7.5.2.7 Optimistic Locking and Versioning

As discussed in the section on [Optimistic and Pessimistic Locking](#), by default GORM uses optimistic l
injects a `version` property into every class which is in turn mapped to a `version` column at the databas

If you're mapping to a legacy schema that doesn't have version columns (or there's some other reason why
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
> are open to the risk of users losing data (due to data overriding) unless you use [pessimistic loc](#)

Version columns types

By default Grails maps the `version` property as a `Long` that gets incremented by one each time an
Hibernate also supports using a `Timestamp`, for example:

235

```
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 t
but this risk is very low. One benefit of using a Timestamp instead of a Long is that you combine th
last-updated semantics into a single column.

7.5.2.8 Eager and Lazy Fetching

Lazy Collections

As discussed in the section on [Eager and Lazy fetching](#), GORM collections are lazily loaded by default
behaviour with the ORM DSL. There are several options available to you, but the most 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'
    }
}
```

236

```
class Address {
    String street
    String postCode
}
```

```
class Pet {
    String name
}
```

The first option, `lazy: false` , ensures that when a `Person` instance is loaded, its addresses c
same time with a second SELECT. The second option is basically the same, except the collection is loade
another SELECT. Typically you want to reduce the number of queries, so `fetch: 'join'` is the mo
the other hand, it could feasibly be the more expensive approach if your domain model and data result i
than would otherwise be 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 examp
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 wi
four next ones. You can get the same behaviour with eager loading by combining `batchSize` with the
You can find out more about these options in the [Hibernate user guide](#) and this [primer on fetching strateg](#)
does not currently support the "subselect" fetching strategy.

Lazy Single-Ended Associations

237

In GORM, one-to-one and many-to-one associations are by default lazy. Non-lazy single ended associa when you load many entities because each non-lazy association will result in an extra SELECT statement also have non-lazy associations, the number of queries grows significantly!

Use the same technique as for lazy collections to make a one-to-one or many-to-one association non-lazy/e

```
class Person {
    String firstName
}
```

```
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) w loaded.

Lazy Single-Ended Associations and Proxies

Hibernate uses runtime-generated proxies to facilitate single-ended lazy associations; Hibernate dynamic class to create the proxy.

Consider the previous example but with a lazily-loaded `person` association: Hibernate will set the `per` that is a subclass of `Person`. When you call any of the getters (except for the `id` property) or setters on t load the entity from the database.

Unfortunately this technique can produce surprising results. Consider the following example classes:

```
class Pet {
    String name
}
```

```
class Dog extends Pet {
}
```

```
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 work

```
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 Hibern you load the `Person` instance, Hibernate creates a proxy for its `pet` relation and attaches it to the ses whenever you retrieve that `Pet` instance with a query, a `get()`, or the `pet` relation *within the same sess* the proxy.

239

Fortunately for us, GORM automatically unwraps the proxy when you use `get()` and `findBy*()`, or
the relation. That means you don't have to worry at all about proxies in the majority of cases. But GORM
returned with a query that returns a list, such as `list()` and `findAllBy*()`. However, if Hibernate ha
the session, those queries will return the 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 a `ClassCa`
the first pet in the list is a proxy instance with a class that is neither `Dog` 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
`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
an error occurs and be able to work around it.

7.5.2.9 Custom Cascade Behaviour

As described in the section on <u>cascading updates</u>, the primary mechanism to control the way updates and
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

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 c association then it should be automatically deleted. Children are also deleted when the parent is.

> ⚠ It is advisable to read the section in the Hibernate documentation on transitive persistence 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 settin

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

```
class Address {
    String street
    String postCode
}
```

7.5.2.10 Custom Hibernate Types

You saw in an earlier section that you can use composition (with the `embedded` property) to break a ta You can achieve a similar effect with Hibernate's custom user types. These are not domain classes then Groovy classes. Each of these types also has a corresponding "meta-type" class that implements org.hibern

The Hibernate reference manual has some information on custom types, but here we will focus on how to start by taking a look at a simple domain class that uses an old-fashioned (pre-Java 1.5) type-safe enum cla

```
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
corresponding `UserType` implementation. That's all you have to do to start using your custom type. If yo
the other column settings such as "column" to change the column name and "index" to add it to an index.

Custom types aren't limited to just a single column - they can be mapped to as many columns as you
explicitly define in the mapping what columns to use, since Hibernate can only use the property na
Fortunately, Grails lets you map multiple columns to a property using this syntax:

```
class Book {

String title
    Name author
    Rating rating

static mapping = {
        author type: NameUserType, {
            column name: "first_name"
            column name: "last_name"
        }
        rating type: RatingUserType
    }
}
```

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

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

The column definitions do *not* support the following attributes: `type`, `cascade`, `lazy`, `cache`, and `joi`

242

One thing to bear in mind with custom types is that they define the *SQL types* for the corresponding data[take the burden of configuring them yourself, but what happens if you have a legacy database that uses a d[of the columns? In that case, override the column's SQL type using the `sqlType` attribute:

```
class Book {
String title
    Name author
    Rating rating

static mapping = {
        author type: NameUserType, {
            column name: "first_name", sqlType: "text"
            column name: "last_name", sqlType: "text"
        }
        rating type: RatingUserType, sqlType: "text"
    }
}
```

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

7.5.2.11 Derived Properties

A derived property is one that takes its value from a SQL expression, often but not necessarily based on 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 probably n[property. The SQL used to derive the value of a derived property may be expressed in the ORM DSL like [

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

static mapping = {
        tax formula: 'PRICE * TAX_RATE'
    }
}
```

243

Note that the formula expressed in the ORM DSL is SQL so references to other properties should relate
not the object model, which is why the example refers to PRICE and TAX_RATE instead of price and t

With that in place, when a Product is retrieved with something like Product.get(42), the SQL that is
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 eff
adding a method like getTax() to the Product class that simply returns the product of the taxRat
With an approach like that you would give up the ability query the database based on the value of the
derived property allows exactly that. To retrieve all Product objects that have a tax value greater than
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 e
> SQL code, derived properties may not have GORM constraints applied to them. If con
> specified for a derived property, they will be ignored.

7.5.2.12 Custom Naming Strategy

By default Grails uses Hibernate's `ImprovedNamingStrategy` to convert domain class Class and f
and column names by converting from camel-cased Strings to ones that use underscores as word sepapara
these on a per-class basis in the `mapping` closure but if there's a consistent pattern you ca
`NamingStrategy` class to use.

Configure the class name to be used in `grails-app/conf/DataSource.groovy` in the `hibernat`

```
dataSource {
    pooled = true
    dbCreate = "create-drop"
    …
}

hibernate {
    cache.use_second_level_cache = true
    …
    naming_strategy = com.myco.myproj.CustomNamingStrategy
}
```

You can also specify the name of the class and it will be loaded for you:

245

```
hibernate {
    …
    naming_strategy = 'com.myco.myproj.CustomNamingStrategy'
}
```

A third option is to provide an instance if there is some configuration required beyond calling the default c

```
hibernate {
    …
    def strategy = new com.myco.myproj.CustomNamingStrategy()
    // configure as needed
    naming_strategy = strategy
}
```

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

```
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)
    }
}
```

7.5.3 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 `Airport` instances will by default be sorted by the airport na
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 re
> because they involve a join table. See [this issue](#) for more details. Consider using a `Sort`
> queries with sort parameters to fetch the data you need.

7.6 Programmatic Transactions

Grails is built on Spring and uses Spring's Transaction abstraction for dealing with programmatic transa
classes have been enhanced to make this simpler with the [withTransaction](#) method. This method has a sin
which has a single parameter which is a Spring [TransactionStatus](#) 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 unchecked
(but not a checked `Exception`, even though Groovy doesn't require that you catch checked exceptio
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 wa
transaction. This can be achieved through the use of Spring's [SavePointManager](#) interface.

The `withTransaction` method deals with the begin/commit/rollback logic for you within the scope of

7.7 GORM and Constraints

Although constraints are covered in the [Validation](#) section, it is important to mention them here as son
affect the way in which the database schema is generated.

Where feasible, Grails uses a domain class's constraints to influence the database columns generated for t
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 characters i case, we would likely define the column as follows *if* we were creating the table with 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 *before* we persist any records. In Grails, we achieve this validation with [constraints](). We would add declaration to the domain class.

```
static constraints = {
    description maxSize: 1000
}
```

This constraint would provide both the application-based validation we want and it would also cause the s shown above. Below is a description of the other constraints that influence schema generation.

Constraints Affecting String Properties

- [inList]()
- [maxSize]()
- [size]()

If either the `maxSize` or the `size` constraint is defined, Grails sets the maximum column length based on

In general, it's not advisable to use both constraints on the same domain class property. However, if both and the `size` constraint are defined, then Grails sets the column length to the minimum of the `maxSize` bound of the size constraint. (Grails uses the minimum of the two, because any length that exceeds that validation error.)

If the `inList` constraint is defined (and the `maxSize` and the `size` constraints are not defined), then column length based on the length of the longest string in the list of valid values. For example, given a list "Groovy", and "C++", Grails would set the column length to 6 (i.e., the number of characters in the string '

Constraints Affecting Numeric Properties

249

- [min](#)

- [max](#)

- [range](#)

If the `max`, `min`, or `range` constraint is defined, Grails attempts to set the column precision based on t success of this attempted influence is largely dependent on how Hibernate interacts with the underlying DI

In general, it's not advisable to combine the pair `min`/ `max` and `range` constraints together on the sam However, if both of these constraints is defined, then Grails uses the minimum precision value from the co minimum of the two, because any length that exceeds that minimum precision will result in a validation er

- [scale](#)

If the scale constraint is defined, then Grails attempts to set the column [scale](#) based on the constraint value floating point numbers (i.e., `java.lang.Float`, `java.Lang.Double`, `java.lang.BigDeci` `java.lang.BigDecimal`). The success of this attempted influence is largely dependent on how Hib underlying DBMS.

The constraints define the minimum/maximum numeric values, and Grails derives the maximum numbe precision. Keep in mind that specifying only one of `min`/ `max` constraints will not affect schema generat large negative value of property with max:100, for example), unless the specified constraint value require: Hibernate column precision is (19 at the moment). For example:

```
someFloatValue max: 1000000, scale: 3
```

would yield:

```
someFloatValue DECIMAL(19, 3) // precision is default
```

but

```
someFloatValue max: 12345678901234567890, scale: 5
```

would yield:

```
someFloatValue DECIMAL(25, 5) // precision = digits in max + scale
```

and

```
someFloatValue max: 100, min: -100000
```

would yield:

```
someFloatValue DECIMAL(8, 2) // precision = digits in min + default scale
```

8 The Web Layer

8.1 Controllers

A controller handles requests and creates or prepares the response. A controller can generate the response view. To create a controller, simply create a class whose name ends with `Controller` in the `grail` 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 t defined within your controller maps to URIs within the controller name URI.

8.1.1 Understanding Controllers and Actions

Creating a controller

Controllers can be created with the [create-controller](#) or [generate-controller](#) command. For example t 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/BookCon`

```
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 conve 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 pre 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 with super.ac

- Methods can be intercepted with standard proxying mechanisms, something that is complicated to they're fields.

If you prefer the Closure syntax or have older controller classes created in earlier versions of Grails and st using methods, you can set the grails.compile.artefacts.closures.convert BuildConfig.groovy:

```
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 un grails-app/controllers/ directory, methods inherited from that class are not c controller actions. If the intent is to expose those inherited methods as controller actions t may be overridden in the subclass and the subclass method may invoke the method in the supe

The Default Action

A controller has the concept of a default URI that maps to the root URI of the controller, fo: BookController. The action that is called when the default URI is requested is dictated by the followir

- 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"
```

8.1.2 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 enti servletContext is an instance of [ServletContext](#)
- [session](#) - The session allows associating state with a given user and typically uses cookies to associa The session object is an instance of [HttpSession](#)
- [request](#) - The request object allows the storage of objects for the current request only. The request [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 o 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 req
only. Afterwards the attributes are cleared. This is useful for setting a message directly before redirecting,

```
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 ar
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 display i
object type.

Scoped Controllers

Supported controller scopes are:

- `prototype` (default) - A new controller will be created for each request (recommended for actions a

- `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 values

255

```
static scope = "singleton"
```

You can define the default strategy under in `Config.groovy` with the `grails.controllers.d`
example:

```
grails.controllers.defaultScope = "singleton"
```

Newly created applications have the `grails.controllers.defaultScope`
`grails-app/conf/Config.groovy` with a value of "singleton". You may change this value to any
listed above. If the property is not assigned a value at all, controllers will default to "prototype" scope.

> ⚠ Use scoped controllers wisely. For instance, we don't recommend having any prop
> singleton-scoped controller since they will be shared for *all* requests.

8.1.3 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
view. There are a couple of ways to return a model. First, you can explicitly return a Map instance:

```
def show() {
    [book: Book.get(params.id)]
}
```

> ⚠ The above does *not* reflect what you should use with the scaffolding views - see the scaffol
> for more details.

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 of

Selecting the View

In both of the previous two examples there was no code that specified which view to render. So how doe
to pick? The answer lies in the conventions. Grails will look for a view at the location `grails-app/vi`
for this `show` 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`
automatically qualifies the view location with the `book` directory of the `grails-app/views` director
to access shared 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/displa

Grails also supports JSPs as views, so if a GSP isn't found in the expected location but a JSP is, it will be u

Selecting Views For Namespaced Controllers

If a controller defines a namespace for itself with the namespace property that will affect the root direc
look for views which are specified with a relative path. The default root directory for views rendered by a
grails-app/views/<namespace name>/<controller name>/. If the view is not found in t
then Grails will fallback to looking for the view in the non-namespaced directory.

See the example below.

```
class ReportingController {
    static namespace = 'business'

def humanResources() {
        // This will render grails-app/views/business/reporting/humanResources.gs
        // if it exists.

// If grails-app/views/business/reporting/humanResources.gsp does not
        // exist the fallback will be grails-app/views/reporting/humanResources.g

// The namespaced GSP will take precedence over the non-namespaced GSP.

[numberOfEmployees: 9]
    }

    def accountsReceivable() {
        // This will render grails-app/views/business/reporting/accounting.gsp
        // if it exists.

// If grails-app/views/business/reporting/accounting.gsp does not
        // exist the fallback will be grails-app/views/reporting/accounting.gsp.

// The namespaced GSP will take precedence over the non-namespaced GSP.

render view: 'numberCrunch', model: [numberOfEmployees: 13]
    }
}
```

Rendering a Response

Sometimes it's easier (for example with Ajax applications) to render snippets of text or code to the re
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` meth 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 `MarkupBuild`
a `<form>` element, use the following:

```
def login() {
    // …
    body {
        h1 'Hello'
        builder.form {
        }
    }
    // …
}
```

8.1.4 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:

```
// 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 para
specified with the same name as a request parameter, the request parameter is overridden and the controlle

Since the `params` object is a Map, you can use it to pass the current request parameters from one action to

```
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. For ex
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. Th
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"])
```

8.1.5 Controller Interceptors

Often it is useful to intercept processing based on either request, session or application state. This can
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
> writing a [Filter](). Filters can be applied to multiple controllers or URIs without the need to
> logic of each controller

Before Interception

The `beforeInterceptor` intercepts processing before the action is executed. If it returns `false` th
will not be executed. The interceptor can be defined for all actions in a controller as follows:

```
def beforeInterceptor = {
    println "Tracing action ${actionUri}"
}
```

The above is declared inside the body of the controller definition. It will be executed before all actions an
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
The `beforeInterceptor` then defines an interceptor that is used on all actions *except* the login action
method. The `auth` method is referenced using Groovy's method pointer syntax. Within the method it d
user in the session, and if not it redirects to the `login` action and returns `false`, causing the inte
processed.

After Interception

Use the `afterInterceptor` property to define an interceptor that is executed after an action:

264

```
def afterInterceptor = { model ->
    println "Tracing action ${actionUri}"
}
```

The after interceptor takes the resulting model as an argument and can hence manipulate the model or resp

An after interceptor may also modify the Spring MVC [ModelAndView](#) object prior to rendering. In this
becomes:

```
def afterInterceptor = { model, modelAndView ->
    println "Current view is ${modelAndView.viewName}"
    if (model.someVar) modelAndView.viewName = "/mycontroller/someotherview"
    println "View is now ${modelAndView.viewName}"
}
```

This allows the view to be changed based on the model returned by the current action. Note that the m
null if the action being intercepted called `redirect` or `render`.

Interception Conditions

Rails users will be familiar with the authentication example and how the 'except' condition was used when
(interceptors are called 'filters' in Rails; this terminology conflicts with Servlet filter 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 defined

```
def beforeInterceptor = [action: this.&auth, except: ['login', 'register']]
```

The other supported condition is 'only', this executes the interceptor for only the specified action(s):

265

```
    def beforeInterceptor = [action: this.&auth, only: ['secure']]
```

8.1.6 Data Binding

Data binding is the act of "binding" incoming request parameters onto the properties of an object or an ent
binding should deal with all necessary type conversion since request parameters, which are typica
submission, are always strings whilst the properties of a Groovy or Java object may well not be.

Map Based Binding

The data binder is capable of converting and assigning values in a Map to properties of an object. The bi
in the Map to properties of the object using the keys in the Map that have values which correspond to prop
The following code demonstrates the basics:

```
// grails-app/domain/Person.groovy
class Person {
    String firstName
    String lastName
    Integer age
}
```

```
def bindingMap = [firstName: 'Peter', lastName: 'Gabriel', age: 63]

def person = new Person(bindingMap)

assert person.firstName == 'Peter'
assert person.lastName == 'Gabriel'
assert person.age == 63
```

To update properties of a domain object you may assign a Map to the properties property of the doma

```
def bindingMap = [firstName: 'Peter', lastName: 'Gabriel', age: 63]

def person = Person.get(someId)
person.properties = bindingMap

assert person.firstName == 'Peter'
assert person.lastName == 'Gabriel'
assert person.age == 63
```

The binder can populate a full graph of objects using Maps of Maps.

```
class Person {
    String firstName
    String lastName
    Integer age
    Address homeAddress
}
class Address {
    String county
    String country
}
```

```
def bindingMap = [firstName: 'Peter', lastName: 'Gabriel', age: 63, homeAddress:
country: 'England'] ]

def person = new Person(bindingMap)

assert person.firstName == 'Peter'
assert person.lastName == 'Gabriel'
assert person.age == 63
assert person.homeAddress.county == 'Surrey'
assert person.homeAddress.country == 'England'
```

Binding To Collections And Maps

The data binder can populate and update Collections and Maps. The following code shows a simple examp
of objects in a domain class:

267

```groovy
class Band {
    String name
    static hasMany = [albums: Album]
    List albums
}
class Album {
    String title
    Integer numberOfTracks
}
```

```groovy
def bindingMap = [name: 'Genesis',
                  'albums[0]': [title: 'Foxtrot', numberOfTracks: 6],
                  'albums[1]': [title: 'Nursery Cryme', numberOfTracks: 7]]

def band = new Band(bindingMap)

assert band.name == 'Genesis'
assert band.albums.size() == 2
assert band.albums[0].title == 'Foxtrot'
assert band.albums[0].numberOfTracks == 6
assert band.albums[1].title == 'Nursery Cryme'
assert band.albums[1].numberOfTracks == 7
```

That code would work in the same way if albums were an array instead of a List.

Note that when binding to a Set the structure of the Map being bound to the Set is the same as that of
List but since a Set is unordered, the indexes don't necessarily correspond to the order of elements
example above, if albums were a Set instead of a List, the bindingMap could look exactly the sa
the first album in the Set or it might be the second. When updating existing elements in a Set the Map b
must have id elements in it which represent the element in the Set being updated, as in the following exa

```
/*
 * The value of the indexes 0 and 1 in albums[0] and albums[1] are arbitrary
 * values that can be anything as long as they are unique within the Map.
 * They do not correspond to the order of elements in albums because albums
 * is a Set.
 */
def bindingMap = ['albums[0]': [id: 9, title: 'The Lamb Lies Down On Broadway']
                  'albums[1]': [id: 4, title: 'Selling England By The Pound']]

def band = Band.get(someBandId)

/*
 * This will find the Album in albums that has an id of 9 and will set its title
 * to 'The Lamb Lies Down On Broadway' and will find the Album in albums that has
 * an id of 4 and set its title to 'Selling England By The Pound'.  In both
 * cases if the Album cannot be found in albums then the album will be retrieved
 * from the database by id, the Album will be added to albums and will be updated
 * with the values described above.  If a Album with the specified id cannot be
 * found in the database, then a binding error will be created and associated
 * with the band object.  More on binding errors later.
 */
band.properties = bindingMap
```

When binding to a Map the structure of the binding Map is the same as the structure of a Map used for bin
and the index inside of square brackets corresponds to the key in the Map being bound to. See the followin

```
class Album {
    String title
    static hasMany = [players: Player]
    Map players
}

class Player {
    String name
}
```

```
def bindingMap = [title: 'The Lamb Lies Down On Broadway',
                  'players[guitar]': [name: 'Steve Hackett'],
                  'players[vocals]': [name: 'Peter Gabriel'],
                  'players[keyboards]': [name: 'Tony Banks']]

def album = new Album(bindingMap)

assert album.title == 'The Lamb Lies Down On Broadway'
assert album.players.size() == 3
assert album.players.guitar.name == 'Steve Hackett'
assert album.players.vocals.name == 'Peter Gabriel'
assert album.players.keyboards.name == 'Tony Banks'
```

269

When updating an existing Map, if the key specified in the binding Map does not exist in the Map being b
will be created and added to the Map with the specified key as in the following example:

```
def bindingMap = [title: 'The Lamb Lies Down On Broadway',
                   'players[guitar]': [name: 'Steve Hackett'],
                   'players[vocals]': [name: 'Peter Gabriel']
                   'players[keyboards]': [name: 'Tony Banks']]

def album = new Album(bindingMap)

assert album.title == 'The Lamb Lies Down On Broadway'
assert album.players.size() == 3
assert album.players.guitar == 'Steve Hackett'
assert album.players.vocals == 'Peter Gabriel'
assert album.players.keyboards == 'Tony Banks'

def updatedBindingMap = ['players[drums]': [name: 'Phil Collins'],
                         'players[keyboards]': [name: 'Anthony George Banks']]

album.properties = updatedBindingMap

assert album.title == 'The Lamb Lies Down On Broadway'
assert album.players.size() == 4
assert album.players.guitar.name == 'Steve Hackett'
assert album.players.vocals.name == 'Peter Gabriel'
assert album.players.keyboards.name == 'Anthony George Banks'
assert album.players.drums.name == 'Phil Collins'
```

Binding Request Data to the Model

The params object that is available in a controller has special behavior that helps convert dotted reque
nested Maps that the data binder can work with. For example, if a request includes reque
person.homeAddress.country and person.homeAddress.city with values 'USA' and '
params would include entries like these:

```
[person: [homeAddress: [country: 'USA', city: 'St. Louis']]]
```

There are two ways to bind request parameters onto the properties of a domain class. The first involves
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 the Grails automatically recognizes that you are trying to bind from request parameters. So if we had an incom

```
/book/save?title=The%20Stand&author=Stephen%20King
```

Then the `title` and `author` request parameters would automatically be set on the domain class. You property to perform data binding onto an existing instance:

```
def save() {
    def b = Book.get(params.id)
    b.properties = params
    b.save()
}
```

This has the same effect as using the implicit constructor.

When binding an empty String (a String with no characters in it, not even spaces), the data binder will co null. This simplifies the most common case where the intent is to treat an empty form field as having the v a way to actually submit a null as a request parameter. When this behavior is not desirable the applicati directly.

The mass property binding mechanism will by default automatically trim all Strings at binding time. To the `grails.databinding.trimStrings` property to false in `grails-app/conf/Config.gr`

```
// the default value is true
grails.databinding.trimStrings = false

// ...
```

The mass property binding mechanism will by default automatically convert all empty Strings to null at this behavior set the `grails.databinding.convertEmptyStringsToNull` prop grials-app/conf/Config.groovy.

271

```
// the default value is true
grails.databinding.convertEmptyStringsToNull = false

// ...
```

The order of events is that the String trimming happens and then null conversion happens so if `trim`
`convertEmptyStringsToNull` is `true`, not only will empty Strings be converted to null but als
String is any String such that the `trim()` method returns an empty String.

> ⛔ These forms of data binding in Grails are very convenient, but also indiscriminate. In other
> will bind *all* non-transient, typed instance properties of the target object, including ones that y
> want bound. Just because the form in your UI doesn't submit all the properties, an attacker ca
> malign data via a raw HTTP request. Fortunately, Grails also makes it easy to protect a
> attacks - see the section titled "Data Binding and Security concerns" for more information.

Data binding and Single-ended Associations

If you have a `one-to-one` or `many-to-one` association you can use Grails' data binding ca
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` instar
doing data binding such as:

```
def b = new Book(params)
```

An association property can be set to `null` by passing the literal `String` "null". For example:

```
/book/save?author.id=null
```

Data Binding and Many-ended Associations

If you have a one-to-many or many-to-many association there are different techniques for data binding dep
type.

If you have a `Set` based association (the default for a `hasMany`) then the simplest way to populate an as
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 v
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 th
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
This is because a `Set` has no concept of order, so although we're referring to `books0` and `books1` it i
order of the association will be correct on the server side unless you apply some explicit sorting yourself.

This is not a problem if you use `List` based associations, since a `List` has a defined order and an index
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 th
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.

273

You can bind existing instances of the associated type to a `List` using the same `.id` syntax as you would 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 `books0` 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 replaced

```
<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"`.

When binding to Maps, Arrays and Collections the data binder will automatically grow the size of the coll default limit to how large the binder will grow a collection is 256. If the data binder encounters an entry th be grown beyond that limit, the entry is ignored. The limit may be configured by assig `grails.databinding.autoGrowCollectionLimit` property in `Config.groovy`.

```
// grails-app/conf/Config.groovy
// the default value is 256
grails.databinding.autoGrowCollectionLimit = 128
// ...
```

Data binding with Multiple domain classes

It is possible to bind data to multiple domain objects from the [params](#) object.

For example so you have an incoming request to:

```
/book/save?book.title=The%20Stand&author.name=Stephen%20King
```

You'll notice the difference with the above request is that each parameter has a prefix such as `author.` or
isolate which parameters belong to which type. Grails' `params` object is like a multi-dimensional hash an
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
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 con
The first category is command objects. Complex types are treated as command objects. See the [Comma]
user guide for details. The other category is basic object types. Supported types are the 8 primitives,
wrappers and [java.lang.String](). The default behavior is to map request parameters to action arguments by n

```
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
carried out before the request parameter value can be bound to the action argument. The type conversion h
a case like the example shown above, the `params.accountType` request parameter has to be conv
conversion fails for any reason, the argument will have its default value per normal Java behavior (null for
false for booleans and zero for numbers) and a corresponding error will be added to the `errors`
controller.

```
/accounting/displayInvoice?accountNumber=B59786&accountType=bogusValue
```

Since "bogusValue" cannot be converted to type int, the value of accountType will be
`errors.hasErrors()` will be true, the controller's `errors.errorCount` will be equal to
`errors.getFieldError('accountType')` will contain the corresponding error.

If the argument name does not match the name of the request parameter then the `@grails.web`
annotation may be applied to an argument to express the name of the request parameter which should be b

```
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
{
        // …
    }
}
```

Data binding and type conversion errors

Sometimes when performing data binding it is not possible to convert a particular String into a particular t
a type conversion error. Grails will retain type conversion errors inside the [errors](#) property of a Grails doma

```
class Book {
    …
    URL publisherURL
}
```

Here we have a domain class `Book` that uses the `java.net.URL` class to represent URLs. Given an inco

```
/book/save?publisherURL=a-bad-url
```

it is not possible to bind the string `a-bad-url` to the `publisherURL` property as a type mismatch err
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 ty
would want a message from the `grails-app/i18n/messages.properties` file to use for the err
error message handler such as:

```
typeMismatch.java.net.URL=The field {0} is not a valid URL
```

Or a more specific one:

277

```
typeMismatch.Book.publisherURL=The publisher URL you specified is not a valid URL
```

The BindUsing Annotation

The BindUsing annotation may be used to define a custom binding mechanism for a particular field i
binding is being applied to the field the closure value of the annotation will be invoked with 2 arguments.
object that data binding is being applied to and the second argument is DataBindingSource which is the
binding. The value returned from the closure will be bound to the property. The following example woul
version of the name value in the source being applied to the name field during data binding.

```
import org.grails.databinding.BindUsing

class SomeClass {
    @BindUsing({obj, source ->

//source is DataSourceBinding which is similar to a Map
        //and defines getAt operation but source.name cannot be used here.
        //In order to get name from source use getAt instead as shown below.

source['name']?.toUpperCase()
    })
    String name
}
```

> ⚠ Note that data binding is only possible when the name of the request parameter matches wi
> name in the class. Here, name from request parameters matches with name from SomeClas

The BindUsing annotation may be used to define a custom binding mechanism for all of the fields on a p
annotation is applied to a class, the value assigned to the annotation should be a class which impler
interface. An instance of that class will be used any time a value is bound to a property in the class that
applied to.

```
@BindUsing(SomeClassWhichImplementsBindingHelper)
class SomeClass {
    String someProperty
    Integer someOtherProperty
}
```

Custom Data Converters

The binder will do a lot of type conversion automatically. Some applications may want to define th
converting values and a simple way to do this is to write a class which implements [ValueConverter](#) and r
class as a bean in the Spring application context.

```groovy
package com.myapp.converters

import org.grails.databinding.converters.ValueConverter

/**
 * A custom converter which will convert String of the
 * form 'city:state' into an Address object.
 */
class AddressValueConverter implements ValueConverter {

boolean canConvert(value) {
        value instanceof String
    }

def convert(value) {
        def pieces = value.split(':')
        new com.myapp.Address(city: pieces[0], state: pieces[1])
    }

Class<?> getTargetType() {
        com.myapp.Address
    }
}
```

An instance of that class needs to be registered as a bean in the Spring application context. The bean na
beans that implemented ValueConverter will be automatically plugged in to the data binding process.

```groovy
// grails-app/conf/spring/resources.groovy

beans = {

addressConverter com.myapp.converters.AddressValueConverter

// ...

}
```

279

```
class Person {
    String firstName
    Address homeAddress
}
class Address {
    String city
    String state
}

def person = new Person()
person.properties = [firstName: 'Jeff', homeAddress: "O'Fallon:Missouri"]
assert person.firstName == 'Jeff'
assert person.homeAddress.city = "O'Fallon"
assert person.homeAddress.state = 'Missouri'
```

Date Formats For Data Binding

A custom date format may be specified to be used when binding a String to a Date value by applying the ~~E~~ to a Date field.

```
import org.grails.databinding.BindingFormat

class Person {
    @BindingFormat('MMddyyyy')
    Date birthDate
}
```

A global setting may be configured in `Config.groovy` to define date formats which will be used applie to Date.

```
// grails-app/conf/Config.groovy

grails.databinding.dateFormats = ['MMddyyyy', 'yyyy-MM-dd HH:mm:ss.S', "yyyy-MM-d
```

The formats specified in `grails.databinding.dateFormats` will be attempted in the order in w the List. If a property is marked with @BindingFormat, the @BindingFormat will take precedence ove grails.databinding.dateFormats.

The default formats that are used are "yyyy-MM-dd HH:mm:ss.S", "yyyy-MM-dd'T'hh:mm:ss'Z'" and "yy z".

Custom Formatted Converters

You may supply your own handler for the [BindingFormat](#) annotation by writing a class w
[FormattedValueConverter](#) interface and registering an instance of that class as a bean in the Spring applica
example of a trivial custom String formatter that might convert the case of a String based on th
BindingFormat annotation.

```groovy
package com.myapp.converters

import org.grails.databinding.converters.FormattedValueConverter

class FormattedStringValueConverter implements FormattedValueConverter {
    def convert(value, String format) {
        if('UPPERCASE' == format) {
            value = value.toUpperCase()
        } else if('LOWERCASE' == format) {
            value = value.toLowerCase()
        }
        value
    }

Class getTargetType() {
        // specifies the type to which this converter may be applied
        String
    }
}
```

An instance of that class needs to be registered as a bean in the Spring application context. The bean na
beans that implemented FormattedValueConverter will be automatically plugged in to the data binding pro

```groovy
// grails-app/conf/spring/resources.groovy

beans = {

formattedStringConverter com.myapp.converters.FormattedStringValueConverter

// ...

}
```

With that in place the BindingFormat annotation may be applied to String fields to inform the data bi
the custom converter.

```
import org.grails.databinding.BindingFormat

class Person {
    @BindingFormat('UPPERCASE')
    String someUpperCaseString

@BindingFormat('LOWERCASE')
    String someLowerCaseString

String someOtherString
}
```

Localized Binding Formats

The `BindingFormat` annotation supports localized format strings by using the optional `code` attribute
the code attribute that value will be used as the message code to retrieve the binding format string from
bean in the Spring application context and that lookup will be localized.

```
import org.grails.databinding.BindingFormat

class Person {
    @BindingFormat(code='date.formats.birthdays')
    Date birthDate
}
```

```
# grails-app/conf/i18n/messages.properties
date.formats.birthdays=MMddyyyy
```

```
# grails-app/conf/i18n/messages_es.properties
date.formats.birthdays=ddMMyyyy
```

Structured Data Binding Editors

A structured data binding editor is a helper class which can bind structured request parameters to a proper for structured binding is binding to a `Date` object which might be constructed from several smaller pieces in several request parameters with names like `birthday_month`, `birthday_date` and `birthda` editor would retrieve all of those individual pieces of information and use them to construct a `Date`.

The framework provides a structured editor for binding to `Date` objects. An application may register its o whatever types are appropriate. Consider the following classes:

```groovy
// src/groovy/databinding/Gadget.groovy
package databinding

class Gadget {
    Shape expandedShape
    Shape compressedShape
}
```

```groovy
// src/groovy/databinding/Shape.groovy
package databinding

class Shape {
    int area
}
```

A `Gadget` has 2 `Shape` fields. A `Shape` has an `area` property. It may be that the application wants to a like `width` and `height` and use those to calculate the `area` of a `Shape` at binding time. A structur suited for that.

The way to register a structured editor with the data binding process is to add org.grails.databinding.TypedStructuredBindingEditor interface to the Spring application context. The easie `TypedStructuredBindingEditor` interface is to exte org.grails.databinding.converters.AbstractStructuredBindingEditor abstract class and override the getPr as shown below:

```
// src/groovy/databinding/converters/StructuredShapeEditor.groovy
package databinding.converters

import databinding.Shape

import org.grails.databinding.converters.AbstractStructuredBindingEditor

class StructuredShapeEditor extends AbstractStructuredBindingEditor<Shape> {

public Shape getPropertyValue(Map values) {
        // retrieve the individual values from the Map
        def width = values.width as int
        def height = values.height as int

// use the values to calculate the area of the Shape
        def area = width * height

// create and return a Shape with the appropriate area
        new Shape(area: area)
    }
}
```

An instance of that class needs to be registered with the Spring application context:

```
// grails-app/conf/spring/resources.groovy
beans = {
    shapeEditor databinding.converters.StructuredShapeEditor

// …
}
```

When the data binder binds to an instance of the Gadget class it will check to see if there are request
compressedShape and expandedShape which have a value of "struct" and if they do exist, that w
StructuredShapeEditor. The individual components of the structure need to have parame
propertyName_structuredElementName. In the case of the Gadget class above that would mean that tl
request parameter should have a value of "struct" and the compressedShape_width and compr
parameters should have values which represent the width and the height of the compressed S
expandedShape request parameter should have a value of "struct" and the expanded
expandedShape_height parameters should have values which represent the width and the height of t

```
// grails-app/controllers/demo/DemoController.groovy
class DemoController {

def createGadget(Gadget gadget) {
        /*

/demo/createGadget?expandedShape=struct&expandedShape_width=80&expandedShape_heig
                        &compressedShape=struct&compressedShape_width=10&compre

*/

// with the request parameters shown above gadget.expandedShape.area would be 240
        // and gadget.compressedShape.area would be 30

// ...

}
}
```

Typically the request parameters with "struct" as their value would be represented by hidden form fields.

Data Binding Event Listeners

The [DataBindingListener](#) interface provides a mechanism for listeners to be notified of data binding events this:

```
package org.grails.databinding.events;

import org.grails.databinding.errors.BindingError;

public interface DataBindingListener {
/**
     * @return true if the listener is interested in events for the specified typ
     */
    boolean supports(Class<?> clazz);
/**
     * Called when data binding is about to start.
     *
     * @param target The object data binding is being imposed upon
     * @param errors the Spring Errors instance (a org.springframework.validation
     * @return true if data binding should continue
     */
    Boolean beforeBinding(Object target, Object errors);
/**
     * Called when data binding is about to imposed on a property
     *
     * @param target The object data binding is being imposed upon
     * @param propertyName The name of the property being bound to
     * @param value The value of the property being bound
     * @param errors the Spring Errors instance (a org.springframework.validation
     * @return true if data binding should continue, otherwise return false
     */
    Boolean beforeBinding(Object target, String propertyName, Object value, Objec
/**
     * Called after data binding has been imposed on a property
     *
     * @param target The object data binding is being imposed upon
     * @param propertyName The name of the property that was bound to
     * @param errors the Spring Errors instance (a org.springframework.validation
     */
    void afterBinding(Object target, String propertyName, Object errors);
/**
     * Called after data binding has finished.
     *
     * @param target The object data binding is being imposed upon
     * @param errors the Spring Errors instance (a org.springframework.validation
     */
    void afterBinding(Object target, Object errors);
/**
     * Called when an error occurs binding to a property
     * @param error encapsulates information about the binding error
     * @param errors the Spring Errors instance (a org.springframework.validation
     * @see BindingError
     */
    void bindingError(BindingError error, Object errors);
}
```

Any bean in the Spring application context which implements that interface will automatically be registe
The [DataBindingListenerAdapter](#) class implements the DataBindingListener interface and provides
for all of the methods in the interface so this class is well suited for subclassing so your listener clas
implementations for the methods your listener is interested in.

The Grails data binder has limited support for the older [BindEventListener](#) style listeners. BindEventLi

```
package org.codehaus.groovy.grails.web.binding;

import org.springframework.beans.MutablePropertyValues;
import org.springframework.beans.TypeConverter;

public interface BindEventListener {

/**
     * @param target The target to bind to
     * @param source The source of the binding, typically a Map
     * @param typeConverter The type converter to be used
     */
    void doBind(Object target, MutablePropertyValues source, TypeConverter typeCo
}
```

Support for `BindEventListener` is disabled by default. To enable support assign a va
`grails.databinding.enableSpringEventAdapter` property in `grails-app/conf/Conf`

```
// grails-app/conf/Config.groovy
grails.databinding.enableSpringEventAdapter=true

...
```

With `enableSpringEventAdapter` set to `true` instances of `BindEventListener` which are
context will automatically be registered with the data binder. Notice that the `MutablePr`
`TypeConverter` arguments to the `doBind` method in `BindEventListener` are Spring specific cl
to the current data binder. The event adapter will pass `null` values for those arguments. The only re
`doBind` method will be the object being bound to. This limited support is provided for backward compa
for a subset of scenarios. Developers are encouraged to migrate their `BindEventListener`
`DataBindingListener` model.

Using The Data Binder Directly

There are situations where an application may want to use the data binder directly. For example, to do bind
arbitrary object which is not a domain class. The following will not work because the `properties` prope

```
// src/groovy/bindingdemo/Widget.groovy
package bindingdemo

class Widget {
    String name
    Integer size
}
```

```
// grails-app/services/bindingdemo/WidgetService.groovy
package bindingdemo

class WidgetService {

def updateWidget(Widget widget, Map data) {
        // this will throw an exception because
        // properties is read-only
        widget.properties = data
    }
}
```

An instance of the data binder is in the Spring application context with a bean name of grailsWebD
implements the [DataBinder](#) interface. The following code demonstrates using the data binder directly.

```
// grails-app/services/bindingdmeo/WidgetService
package bindingdemo

import org.grails.databinding.SimpleMapDataBindingSource

class WidgetService {

// this bean will be autowired into the service
    def grailsWebDataBinder

def updateWidget(Widget widget, Map data) {
        grailsWebDataBinder.bind widget, data as SimpleMapDataBindingSource
    }
}
```

See the [DataBinder](#) documentation for more information about overloaded versions of the bind method.

Data Binding and Security Concerns

When batch updating properties from request parameters you need to be careful not to allow clients t
domain classes and be persisted in the database. You can limit what properties are bound to a given
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 class 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 fie subject to binding if they are not explicitly excluded.

8.1.7 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 produce a

```
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: "application/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 [domain classes](#) 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>
```

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` wit

```
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"}
  ]
```

8.1.8 More on JSONBuilder

The previous section on on XML and JSON responses covered simplistic examples of rendering XML and
the XML builder used by Grails is the standard [XmlSlurper](#) found in Groovy, the JSON builder is a custom
to Grails.

JSONBuilder and Grails versions

JSONBuilder behaves different depending on the version of Grails you use. For version belo [grails.web.JSONBuilder](#) class is used. This section covers the usage of the Grails 1.2 JSONBuilder

For backwards compatibility the old `JSONBuilder` class is used with the `render` method for older 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: "application/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: "application/json") {
    categories = ['a', 'b', 'c']
}
```

This will produce:

```
{"categories":["a","b","c"]}
```

You can also render lists of complex objects, for example:

```
render(contentType: "application/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: "application/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: "application/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: "application/json") {
    categories = [ { a = "A" }, { b = "B" } ]
}
```

You can use the `array` method to build them up dynamically:

```
def results = Book.list()
render(contentType: "application/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 direct

```
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
```

8.1.9 Uploading Files

Programmatic File Uploads

Grails supports file uploads using Spring's [MultipartHttpServletRequest](#) interface. The first step for file
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 the s

There are then a number of ways to handle the file upload. One is to work with the Spring [MultipartFile](#) in

```
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 o
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 will au
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 with
can't handle reasonably sized files. For example, both H2 and MySQL default to a blob size of 255 bytes fo

It is also possible to set the contents of the file as a string by changing the type of the `myFile` property
type:

297

```
class Image {
    String myFile
}
```

8.1.10 Command Objects

Grails controllers support the concept of command objects. A command object is a class that is used
binding, usually to allow validation of data that may not fit into an existing domain class.

> ⚠ Note: A class is only considered to be a command object when it is used as a parameter of an

Declaring Command Objects

Command object classes are defined just like any other class.

```
@grails.validation.Validateable
class LoginCommand {
    String username
    String password

static constraints = {
        username(blank: false, minSize: 6)
        password(blank: false, minSize: 6)
    }
}
```

In this example, the command object is marked with the Validateable annotation. The Validateab
definition of constraints just like in domain classes. If the command object is defined in the same source fi
using it, Grails will automatically mark it as Validateable. It is not required that command object clas

By default, all Validateable object properties are nullable: false which matches the beh
objects. If you want a Validateable that has nullable: true properties by default, you can spe
on the annotation:

```
@grails.validation.Validateable(nullable=true)
class AuthorSearchCommand {
    String  name
    Integer age
}
```

In this example, both `name` and `age` will allow null values during validation.

Using Command Objects

To use command objects, controller actions may optionally specify any number of command object pa
types must be supplied so that Grails knows what objects to create and initialize.

Before the controller action is executed Grails will automatically create an instance of the command obj
properties by binding the request parameters. If the command object class is marked with `Validatea`
object will be validated. For example:

```
class LoginController {

def login(LoginCommand cmd) {
        if (cmd.hasErrors()) {
            redirect(action: 'loginForm')
            return
        }
// work with the command object data
    }
}
```

If the command object's type is that of a domain class and there is an `id` request parameter then instead
class constructor to create a new instance a call will be made to the static `get` method on the domain clas
parameter will be passed as an argument. Whatever is returned from that call to `get` is what will be
action. This means that if there is an `id` request parameter and no corresponding record is found in the d
the command object will be `null`. If an error occurs retrieving the instance from the database then `null`
argument to the controller action and an error will be added the controller's `errors` property. If the co
domain class and there is no `id` request parameter then `null` will be passed into the controller action
method is "POST", in which case a new instance of the domain class will be created by invoking the dom
all of the cases where the domain class instance is non-null, data binding is only performed if the HTTP re
"PUT" or "PATCH".

Command Objects And Request Parameter Names

Normally request parameter names will be mapped directly to property names in the command object. ~~may~~ may be used to bind down the object graph in an intuitive way. In the example below a request parame~~ter~~ bound to the name property of the Person instance and a request parameter named address.city w~~ill~~ property of the address property in the Person.

```
class StoreController {
    def buy(Person buyer) {
        // …
    }
}

class Person {
    String name
    Address address
}

class Address {
    String city
}
```

A problem may arise if a controller action accepts multiple command objects which happen to contain ~~properties with the same name.~~ Consider the following example.

```
class StoreController {
    def buy(Person buyer, Product product) {
        // …
    }
}

class Person {
    String name
    Address address
}

class Address {
    String city
}

class Product {
    String name
}
```

If there is a request parameter named name it isn't clear if that should represent the name of the Prod~~uct or the~~ Person. Another version of the problem can come up if a controller action accepts 2 command objects o~~f the same type as shown~~ below.

```
class StoreController {
    def buy(Person buyer, Person seller, Product product) {
        // …
    }
}

class Person {
    String name
    Address address
}

class Address {
    String city
}

class Product {
    String name
}
```

To help deal with this the framework imposes special rules for mapping parameter names to command ob
object data binding will treat all parameters that begin with the controller action parameter name as belong
command object. For example, the product.name request parameter will be bound to the name p
argument, the buyer.name request parameter will be bound to the name property in the
seller.address.city request parameter will be bound to the city property of the address
argument, etc...

Command Objects and Dependency Injection

Command objects can participate in dependency injection. This is useful if your command object has som
which uses a Grails service:

```
@grails.validation.Validateable
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
ApplicationContext.

Binding The Request Body To Command Objects

When a request is made to a controller action which accepts a command object and the request contains a
to parse the body of the request based on the request content type and use the body to do data binding on
the following example.

```groovy
// grails-app/controllers/bindingdemo/DemoController.groovy
package bindingdemo

class DemoController {

def createWidget(Widget w) {
        render "Name: ${w?.name}, Size: ${w?.size}"
    }
}

class Widget {
    String name
    Integer size
}
```

```
$ curl -H "Content-Type: application/json" -d '{"name":"Some Widget","size":"42"}
localhost:8080/myapp/demo/createWidget
 Name: Some Widget, Size: 42
~ $
$ curl -H "Content-Type: application/xml" -d '<widget><name>Some Other
Widget</name><size>2112</size></widget>' localhost:8080/bodybind/demo/createWidge
 Name: Some Other Widget, Size: 2112
~ $
```

Note that the body of the request is being parsed to make that work. Any attempt to read the body of the
since the corresponding input stream will be empty. The controller action can either use a command object
of the request on its own (either directly, or by referring to something like request.JSON), but cannot do bo

```groovy
// grails-app/controllers/bindingdemo/DemoController.groovy
package bindingdemo

class DemoController {

def createWidget(Widget w) {
        // this will fail because it requires reading the body,
        // which has already been read.
        def json = request.JSON

// ...

}
}
```

8.1.11 Handling Duplicate Form Submissions

Grails has built-in support for handling duplicate form submissions using the "Synchronizer Token Pat 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 defa invalid token in a `flash.invalidToken` variable and redirect the request back to the original page. 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 used in a cluster.

8.1.12 Simple Type Converters

Type Conversion Methods

If you prefer to avoid the overhead of [Data Binding](#) and simply want to convert incoming parameter another more appropriate type the [params](#) object has a number of convenience methods for each type:

```
def total = params.int('total')
```

The above example uses the int method, and there are also methods for boolean, long, char, sh
these methods is null-safe and safe from any parsing errors, so you don't have to perform any additional ch

Each of the conversion methods allows a default value to be passed as an optional second argument. T
returned if a corresponding entry cannot be found in the map or if an error occurs during the conversion. E

```
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
as ?name=Bob&name=Judy.

In this case dealing with one parameter and dealing with many has different semantics since Groovy's
String iterate over each character. To avoid this problem the params object provides a list method tha

```
for (name in params.list('name')) {
    println name
}
```

8.1.13 Declarative Controller Exception Handling

Grails controllers support a simple mechanism for declarative exception handling. If a controller declare
single argument and the argument type is java.lang.Exception or some subclass of java.l
method will be invoked any time an action in that controller throws an exception of that type. See the follo

```groovy
// grails-app/controllers/demo/DemoController.groovy
package demo

class DemoController {

def someAction() {
        // do some work
    }

def handleSQLException(SQLException e) {
        render 'A SQLException Was Handled'
    }

def handleBatchUpdateException(BatchUpdateException e) {
        redirect controller: 'logging', action: 'batchProblem'
    }

def handleNumberFormatException(NumberFormatException nfe) {
        [problemDescription: 'A Number Was Invalid']
    }
}
```

That controller will behave as if it were written something like this...

```groovy
// grails-app/controllers/demo/DemoController.groovy
package demo

class DemoController {

def someAction() {
        try {
            // do some work
        } catch (BatchUpdateException e) {
           return handleBatchUpdateException(e)
        } catch (SQLException e) {
           return handleSQLException(e)
        } catch (NumberFormatException e) {
           return handleNumberFormatException(e)
        }
    }

def handleSQLException(SQLException e) {
        render 'A SQLException Was Handled'
    }

def handleBatchUpdateException(BatchUpdateException e) {
        redirect controller: 'logging', action: 'batchProblem'
    }

def handleNumberFormatException(NumberFormatException nfe) {
        [problemDescription: 'A Number Was Invalid']
    }
}
```

The exception handler method names can be any valid method name. The name is not what makes t
handler, the Exception argument type is the important part.

305

The exception handler methods can do anything that a controller action can do including invoking `rende` a model, etc.

One way to share exception handler methods across multiple controllers is to use inheritance. Except inherited into subclasses so an application could define the exception handlers in an abstract class that m from. Another way to share exception handler methods across multiple controllers is to use a trait, as show

```groovy
// src/groovy/com/demo/DatabaseExceptionHandler.groovy
package com.demo

trait DatabaseExceptionHandler {
    def handleSQLException(SQLException e) {
        // handle SQLException
    }

def handleBatchUpdateException(BatchUpdateException e) {
        // handle BatchUpdateException
    }
}
```

```groovy
// grails-app/controllers/com/demo/DemoController.groovy
package com.demo

class DemoController implements DatabaseExceptionHandler {

// all of the exception handler methods defined
    // in DatabaseExceptionHandler will be added to
    // this class at compile time
}
```

Exception handler methods must be present at compile time. Specifically, exception handler meth metaprogrammed onto a controller class are not supported.

8.2 Groovy Server Pages

Groovy Servers Pages (or GSP for short) is Grails' view technology. It is designed to be familiar for user 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 conver method such as:

```groovy
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
> this document, the practice is strongly discouraged. Mixing mark-up and code is a **bad** thin
> GSP pages contain no code and needn't do so.

A GSP typically has a "model" which is a set of variables that are used for view rendering. The model is
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 ke
within the GSP view using the name `book`:

```
${book.title}
```

> ⛔ Embedding data received from user input has the risk of making your application vulnerable
> Site Scripting (XSS) attack. Please read the documentation on [XSS prevention](#) for information
> prevent XSS attacks.

8.2.1 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 cov
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) a
demonstrates:

```
<html>
    <body>
        <%-- This is my comment --%>
        <%="Hello GSP!" %>
    </body>
</html>
```

> ⛔ Embedding data received from user input has the risk of making your application vulnerable
> Site Scripting (XSS) attack. Please read the documentation on [XSS prevention](#) for informatio
> prevent XSS attacks.

8.2.1.1 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

8.2.1.2 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>
```

8.2.1.3 Page Directives

GSP also supports a few JSP-style page directives.

309

The import directive lets you import classes into the page. However, it is rarely needed due to Groovy's
[Tags](#):

```
<%@ page import="java.awt.*" %>
```

GSP also supports the contentType directive:

```
<%@ page contentType="application/json" %>
```

The contentType directive allows using GSP to render other formats.

8.2.1.4 Expressions

In GSP the `<%= %>` syntax introduced earlier is rarely used due to the support for GSP expressions. A GS
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.

> 🚫 Embedding data received from user input has the risk of making your application vulnerable
> Site Scripting (XSS) attack. Please read the documentation on [XSS prevention](#) for informatio
> prevent XSS attacks.

8.2.2 GSP Tags

Now that the less attractive JSP heritage has been set aside, the following sections cover GSP's built-in tag
way to define GSP pages.

> ⚠ The section on [Tag Libraries](#) covers how to add your own custom tag libraries.

310

All built-in GSP tags start with the prefix g:. Unlike JSP, you don't specify any tag library imports. If automatically assumed to be a GSP tag. An example GSP tag would look like:

```
<g:example />
```

GSP tags can also have a body such as:

```
<g:example>
    Hello world
</g:example>
```

Expressions can be passed into GSP tag attributes, if an expression is not used it will be assumed to be a St

```
<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.

8.2.2.1 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 n 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>
```

The assigned value can also be a bean from the applicationContext:

```
<g:set var="bookService" bean="bookService" />
```

Variables can also be placed in one of the following scopes:

- page - Scoped to the current page (default)

- request - Scoped to the current request

- flash - Placed within flash scope and hence available for the next request

- session - Scoped for the user session

- application - Application-wide scope.

To specify the scope, use the scope attribute:

```
<g:set var="now" value="${new Date()}" scope="request" />
```

8.2.2.2 Logic and Iteration

GSP also supports logical and iterative tags out of the box. For logic there are [if](#), [else](#) and [elseif](#) tags for use

```
<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>
```

8.2.2.3 Search and Filtering

If you have collections of objects you often need to sort and filter them. Use the [findAll](#) and [grep](#) tags for t

```
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
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 Groovy. is a collection of Book instances. Since each Book has a title, you can obtain a list of Book titl books.title. Groovy will auto-magically iterate the collection, obtain each title, and return a new list!

8.2.2.4 Links and Resources

GSP also features tags to help you manage linking to controllers and actions. The link tag lets you spec name pairing and it will automatically work out the link based on the URL Mappings, even if you change t

```
<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>
```

8.2.2.5 Forms and Fields

Form Basics

GSP supports many different tags for working with HTML forms and fields, the most basic of which i controller/action aware version of the regular HTML form tag. The url attribute lets you specify which 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. Be
HTML attributes apply.

Form Fields

In addition to easy construction of forms, GSP supports custom tags for dealing with different types of fiel

- **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
currencySelect and **timeZoneSelect** (for selecting locales, currencies and time zones respectively).

Multiple Submit Buttons

The age old problem of dealing with multiple submit buttons is also handled elegantly with Grails using t
just like a regular submit, but lets you specify an alternative action to submit to:

```
<g:actionSubmit value="Some update label" action="update" />
```

8.2.2.6 Tags as Method Calls

One major different between GSP tags and other tagging technologies is that GSP tags can be called as 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 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 me an adverse effect of WYSIWYG tools such as Dreamweaver that attempt to render the mark-up as it is not

```
<img src="<g:createLinkTo dir="images" file="logo.jpg" />" />
```

Tags as method calls from Controllers and Tag Libraries

You can also invoke tags from controllers and tag libraries. Tags within the default `g:` [namespace]() can 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 Edit]

```
def editor = fckeditor.editor(name: "text", width: "100%", height: "400")
```

8.2.3 Views and Templates

Grails also has the concept of templates. These are useful for partitioning your views into maintainable ch[u]
[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.
have a template that renders Books located at `grails-app/views/book/_bookTemplate.gsp`:

```
<div class="book" id="${book?.id}">
    <div>Title: ${book?.title}</div>
    <div>Author: ${book?.author?.name}</div>
</div>
```

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 mul[t]
can also render the template for each `Book` using the render tag with a `collection` attribute:

```
<g:render template="bookTemplate" var="book" collection="${bookList}" />
```

Shared Templates

In the previous example we had a template that was specific to the BookControlle
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 belo
with the template attribute use an absolute location starting with / instead of a relative location. For exam
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 make
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 [Aja](#) generate small HTML or data responses to partially update the current page instead of performing new req

```
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 in 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 the

8.2.4 Layouts with Sitemesh

Creating Layouts

Grails leverages [Sitemesh](#), a decorator engine, to support view layouts. Layouts a 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 aspe

Triggering Layouts

There are a few ways to trigger a layout. The simplest is to add a meta tag to the view:

```
<html>
    <head>
        <title>An Example Page</title>
        <meta name="layout" content="main" />
    </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 pa
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 "layout"
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
`BookController` delegates to. The value of the "layout" property may contain a directory st
`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/cu`
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 applie
`BookController` delegates to.

Alternatively, you can create a layout called `grails-app/views/layouts/book/list.gsp` whi
the `list` action within the `BookController`.

If you have both the above mentioned layouts in place the layout specific to the action will take precedent
executed.

If a layout may not be located using any of those conventions, the convention of last resort is to look f
layout which is `grails-app/views/layouts/application.gsp`. The name of the applicatic
changed by defining a property in `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/myl`

Inline Layouts

Grails' also supports Sitemesh's concept of inline layouts with the [applyLayout](#) tag. This can be used to app
URL or arbitrary section of content. This lets you even further modularize your view structure by "de
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 includ 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:

```
<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.

8.2.5 Static Resources

Grails 2.0 integrates with the **Asset Pipeline plugin** to provide sophisticated static asset management. T default in new Grails applications.

The basic way to include a link to a static asset in your application is to use the **resource** tag. This simpl pointing to the file.

However modern applications with dependencies on multiple JavaScript and CSS libraries and 1 dependencies on multiple Grails plugins) require something more powerful.

The issues that the Asset-Pipeline plugin tackles are:

- Reduced Dependence - The plugin has compression, minification, and cache-digests built in.

- Easy Debugging - Makes for easy debugging by keeping files separate in development mode.

- Asset Bundling using require **directives**.

- Web application performance tuning is difficult.

- The need for a standard way to expose static assets in plugins and applications.

- The need for extensible processing to make languages like LESS or Coffee first class citizens.

323

The asset-pipeline allows you to define your javascript or css requirements right at the top of the file and t
creation.

Take a look at the [documentation](#) for the asset-pipeline to get started.

8.2.5.1 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 t
exposes a "jquery" resource module, to use jQuery in any page on your site you simply add:

```html
<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
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 a

```html
<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 some J
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:layoutRes
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 resources at the correct time.

To achieve this correctly, you must include the r:layoutResources tag twice in your page, or more common

```
<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 resource should be included.

The default disposition applied depends on the type of resource. All CSS must be rendered in <head> in default for all CSS, and will be rendered by the first r:layoutResources. Page load times are improved w after the page content, so the default for JavaScript files is "defer", which means it is rendered when the s is invoked.

Note that both your GSP page and your Sitemesh layout (as well as any GSP template fragments) can ca resources. The only limitation is that you must call r:require before the r:layoutResources that should rende

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 reco
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 <he
body, based on the disposition.

Given a Sitemesh layout like this:

```
<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 demonstratic

Note that such r:script code fragments **always** load after any modules that you have used, to ensure that a 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 f - 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 declaratior 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 reso is Resources-aware and will delegate to `r:img` if found. However it is recommended that you use `r:` 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.

8.2.5.2 Other resource tags

r:resource

This is equivalent to the Grails [resource](#) tag, returning a link to the processed static resource. Grails'
delegates to this implementation if found, but if your code requires the Resources plugin, you should use r

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
resource such as CSS, JS or a favicon.

See [r:resource documentation](#) for full details.

8.2.5.3 Declaring resources

A DSL is provided for declaring resources and modules. This can go either in your `Config.g`
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 must to e
other resources-specific attributes. Any resource that is not declared is called "ad-hoc" and will still be pro
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 will be auto the box according to the module name, resulting in fewer files. You can override this with `bundle:'som` resource, or call `defaultBundle` on the module (see [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 resource 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 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 `r:external` tag when the engine renders the link to the resource, so you can customize the HTML a link.

There is no limit to the number of modules or xxxResources.groovy artefacts you can provide, and plu 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 `grails.resources.modules` Config variable.

For full details of the resource DSL please see the [resources plugin documentation](#).

8.2.5.4 Overriding plugin resources

Because a resource module can define the bundle groupings and other attributes of resources, you m 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 is trade-off here, but often it is the case that you'd like to override some of these settings.

To do this, the DSL supports an "overrides" clause, within which you can change the `defaultBundle` 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 files required for head and defer dispositions, and JavaScript and CSS files are bundled separately.

Note that overriding individual resources requires the original declaration to have included a unique id for

For full details of the resource DSL please see the [resources plugin documentation](#).

8.2.5.5 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 ov 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 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 from a def the module. Modules have an implicit default bundle name the same as the name of the module.

Files of the same kind will be aggregated into this bundle file. Bundles operate across module boundaries:

329

```
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 change 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 th also sets the caching headers on every response for those resources. To use, simply install the cached-resou

Note that the caching headers can only be set if your resources are being served by your application. If serving the static content from your app (e.g. Apache HTTPD), configure it to send caching header 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, exclu 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 co none are publicly released but releases are imminent.

8.2.5.6 Debugging

When your resources are being moved around, renamed and otherwise mutated, it can be hard to debug c
browsers, especially Safari, Chrome and Firefox have excellent tools that let you view all the resourc
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 adde
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 byp
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
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 it

```
grails.resources.debug = true
```

You can of course set this per-environment.

8.2.5.7 Preventing processing of resources

Sometimes you do not want a resource to be processed in a particular way, or even at all. Occasional
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
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. Mappers
these based on their activity, for example the zipped-resources plugin's "zip" mapper is set to exclude imag

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 fo
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 Grails
(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 A
mappings, which the Resources filter will use to detect such "ad-hoc resource" requests.

By default this is set to:

```
grails.resources.adhoc.patterns = ['images/*', '*.js', '*.css']
```

8.2.5.8 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](#)

8.2.6 Sitemesh Content Blocks

Although it is useful to decorate an entire page sometimes you may find the need to decorate independent
do this you can use content blocks. To get started, partition the page to be decorated using the `<content`

```
<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>
```

8.2.7 Making Changes to a Deployed Application

One of the main issues with deploying a Grails application (or typically any servlet-based one) is that
requires that you redeploy your whole application. If all you want to do is fix a typo on a page, or change a
like a lot of unnecessary work. For such simple requirements, Grails does have a solution: the gra
configuration setting.

How does this work? The first step is to decide where the GSP files should go. Let's say we want to k
/var/www/grails/my-app directory. We add these two lines to grails-app/conf/Config.g:

```
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
many changes as you like but they won't be reflected in the running application until you restart. The seco
to load the views and layouts from.

> ⚠ The trailing slash on the grails.gsp.view.dir value is important! Without it, Grails v
> views in the parent directory.

Setting "grails.gsp.view.dir" is optional. If it's not specified, you can update files directly to the applicati
directory. Depending on the application server, these files might get overwritten when the server is re
servers support "exploded war deployment" which is recommended in this case.

With those settings in place, all you need to do is copy the views from your web application to the 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 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 spa will eventually hit "out of permgen space" errors unless you restart the server. So this technique is not recc large changes to the views.

There are also some System properties to control GSP reloading:

Name	Description
grails.gsp.enable.reload	alternative system property for enabling the GSP reload mode wit Config.groovy
grails.gsp.reload.interval	interval between checking the lastmodified time of the gsp source file, unit is
grails.gsp.reload.granularity	the number of milliseconds leeway to give before deciding a file is out needed because different roundings usually cause a 1000ms difference times

GSP reloading is supported for precompiled GSPs since Grails 1.3.5 .

8.2.8 GSP Debugging

Viewing the generated source code

- Adding "?showSource=true" or "&showSource=true" to the url shows the generated Groovy source of rendering it. It won't show the source code of included templates. This only works in development

- The saving of all generated source code can be activated by setting the property "grails.views.g Config.groovy) . It must point to a directory that exists and is writable.

- During "grails war" gsp pre-compilation, the generated source code is stored in grails.project.work.d ~/.grails/(grails_version)/projects/(project 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 template single page. It might be hard to find out what GSP template actually renders the html seen in the res -feature adds html comments to the output. The comments contain debug information about gsp templates

Usage is simple: append "?debugTemplates" or "&debugTemplates" to the url and view the source of th "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.

8.3 Tag Libraries

Like [Java Server Pages](#) (JSP), GSP supports the concept of custom tag libraries. Unlike JSP, Grails' simple, elegant and completely reloadable at runtime.

Quite simply, to create a tag library create a Groovy class that ends with the convention `TagLib` grails-app/taglib directory:

```
class SimpleTagLib {

}
```

Now to create a tag create a Closure property that takes two arguments: the tag attributes and the body con

```
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 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
the response. Then you can reference the tag inside your GSP; no imports are necessary:

```
<g:emoticon happy="true">Hi John</g:emoticon>
```

⚠ To help IDEs like Spring Tool Suite (STS) and others autocomplete tag attributes, you
Javadoc comments to your tag closures with `@attr` descriptions. Since taglibs use Groovy
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)
    }
}
```

8.3.1 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

8.3.2 Simple Tags

As demonstrated in the previous example it is easy to write simple tags that have no body and just output c
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.
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
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 <u>rend</u>

```
def formatBook = { attrs, body ->
    out << render(template: "bookTemplate", model: [book: attrs.book])
}
```

And then have a separate GSP template that does the actual rendering.

8.3.3 Logical Tags

You can also create logical tags where the body of the tag is only output once a set of conditions have been 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 privileges:

```
<g:isAdmin user="${myUser}">
    // some restricted content
</g:isAdmin>
```

8.3.4 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 Groovy's
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 b
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
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 invocatic

```
out << body((var):num)
```

> ⚠ Note the usage of the parenthesis around the variable name. If you omit these Groovy assur
> 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 referenc
body of the tag.

8.3.5 Tag Namespaces

By default, tags are added to the default Grails namespace and are used with the `g:` prefix in GSP pages. I
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 fro

```
<my:example name="..." />
```

where the prefix is the same as the value of the static `namespace` property. Namespaces are particularly

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

8.3.6 Using JSP Tag Libraries

In addition to the simplified tag library mechanism provided by GSP, you can also use JSP tags from GSP the JSP to use with the `taglib` directive:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

Besides this you have to configure Grails to scan for the JSP tld files. This is `grails.gsp.tldScanPattern` setting. It accepts a comma separated Stri PathMatchingResourcePatternResolver is used to resolve the patterns.

For example you could scan for all available tld files by adding this to Config.groovy:

```
grails.gsp.tldScanPattern='classpath*:/META-INF/*.tld,/WEB-INF/tld/*.tld'
```

JSTL standard library is no more added as a dependency by default. In case you are using JSTL, yc dependencies to BuildConfig.groovy:

```
runtime 'javax.servlet:jstl:1.1.2'
        runtime 'taglibs:standard:1.1.2'
```

Then you can use JSP tags 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")}
```

8.3.7 Tag return value

A taglib can be used in a GSP as an ordinary tag or it might be used as a function in other taglibs or GSP e

Internally Grails intercepts calls to taglib closures. The "out" that is available in a taglib is mapped t implementation that writes to a buffer that "captures" the output of the taglib call. This buffer is the return when it's used as a function.

If the tag is listed in the library's static `returnObjectForTags` array, then its return value will writt used as a normal tag. The return value of the tag lib closure will be returned as-is if it's used as a functio other taglibs.

If the tag is not included in the returnObjectForTags array, then its return value will be discarded. Using returnObjectForTags is not supported.

Example:

```
class ObjectReturningTagLib {
    static namespace = "cms"
    static returnObjectForTags = ['content']

def content = { attrs, body ->
        CmsContent.findByCode(attrs.code)?.content
    }
}
```

Given this example cmd.content(code:'something') call in another taglib or GSP expression v "CmsContent.content" directly to the caller without wrapping the return value in a buffer. It might be wo of performance optimization reasons. There is no need to wrap the tag return value in an output buffer in st

8.4 URL Mappings

Throughout the documentation so far the convention used for URLs has been the default of /cont
However, this convention is not hard wired into Grails and is in fact controlled by a URL Map
grails-app/conf/UrlMappings.groovy.

The UrlMappings class contains a single property called mappings that has been assigned a block of

```
class UrlMappings {
    static mappings = {
    }
}
```

8.4.1 Mapping to Controllers and Actions

To create a simple mapping simply use a relative URL as the method name and specify named paramet
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. Om
map to the default action of the controller:

```
"/product"(controller: "product")
```

An alternative syntax is to assign the controller and action to use within a block passed to the method:

```
"/product" {
    controller = "product"
    action = "list"
}
```

Which syntax you use is largely dependent on personal preference.

If you have mappings that all fall under a particular path you can group mappings with the `group` method

```
group "/product", {
    "/apple"(controller:"product", id:"apple")
    "/htc"(controller:"product", id:"htc")
}
```

To rewrite one URI onto another explicit 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.

8.4.2 Mapping to REST resources

Since Grails 2.3, it possible to create RESTful URL mappings that map onto controllers by convention. follows:

```
"/books"(resources:'book')
```

You define a base URI and the name of the controller to map to using the `resources` parameter. The a in the following URLs:

HTTP Method	URI	Grails Action
GET	/books	index
GET	/books/create	create
POST	/books	save
GET	/books/${id}	show
GET	/books/${id}/edit	edit
PUT	/books/${id}	update
DELETE	/books/${id}	delete

If you wish to include or exclude any of the generated URL mappings you can do so with the `in` parameter, which accepts the name of the Grails action to include or exclude:

```
"/books"(resources:'book', excludes:['delete', 'update'])
or
"/books"(resources:'book', includes:['index', 'show'])
```

Single resources

A single resource is a resource for which there is only one (possibly per user) in the system. You can crea the `resource` parameter (as oppose to `resources`):

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

This results in the following URL mappings:

HTTP Method	URI	Grails Action
GET	/book/create	create
POST	/book	save
GET	/book	show
GET	/book/edit	edit
PUT	/book	update
DELETE	/book	delete

The main difference is that the id is not included in the URL mapping.

Nested Resources

You can nest resource mappings to generate child resources. For example:

```
"/books"(resources:'book') {
   "/authors"(resources:"author")
}
```

The above will result in the following URL mappings:

HTTP Method	URL	Grails Action
GET	/books/${bookId}/authors	index
GET	/books/${bookId}/authors/create	create
POST	/books/${bookId}/authors	save
GET	/books/${bookId}/authors/${id}	show
GET	/books/${bookId}/authors/edit/${id}	edit
PUT	/books/${bookId}/authors/${id}	update
DELETE	/books/${bookId}/authors/${id}	delete

You can also nest regular URL mappings within a resource mapping:

```
"/books"(resources: "book") {
    "/publisher"(controller:"publisher")
}
```

This will result in the following URL being available:

HTTP Method	URL	Grails Action
GET	/books/1/publisher	index

Linking to RESTful Mappings

You can link to any URL mapping created with the `g:link` tag provided by Grails simply by referencing
to link to:

```
<g:link controller="book" action="index">My Link</g:link>
```

As a convenience you can also pass a domain instance to the `resource` attribute of the `link` tag:

```
<g:link resource="${book}">My Link</g:link>
```

This will automatically produce the correct link (in this case "/books/1" for an id of "1").

The case of nested resources is a little different as they typically required two identifiers (the id of the nested within). For example given the nested resources:

```
"/books"(resources:'book') {
    "/authors"(resources:"author")
}
```

If you wished to link to the `show` action of the `author` controller, you would write:

```
// Results in /books/1/authors/2
<g:link controller="author" action="show" method="GET" params="[bookId:1]" id="2"
Author</g:link>
```

However, to make this more concise there is a `resource` attribute to the link tag which can be used inste

```
// Results in /books/1/authors/2
<g:link resource="book/author" action="show" bookId="1" id="2">My Link</g:link>
```

The resource attribute accepts a path to the resource separated by a slash (in this case "book/author"). The be used to specify the necessary `bookId` parameter.

8.4.3 Redirects In URL Mappings

Since Grails 2.3, it is possible to define URL mappings which specify a redirect. When a URL mapping time that mapping matches an incoming request, a redirect is initiated with information provided by the ma

When a URL mapping specifies a redirect the mapping must either supply a String representing a UI provide a Map representing the target of the redirect. That Map is structured just like the Map that may be the `redirect` method in a controller.

```
"/viewBooks"(redirect: '/books/list')
"/viewAuthors"(redirect: [controller: 'author', action: 'list'])
"/viewPublishers"(redirect: [controller: 'publisher', action: 'list', permanent:
```

Request parameters that were part of the original request will be included in the redirect.

8.4.4 Embedded Variables

Simple Variables

The previous section demonstrated how to map simple URLs with concrete "tokens". In URL mappi sequence of characters between each slash, '/'. A concrete token is one which is well defined such as as many circumstances you don't know what the value of a particular token will be until runtime. In this c 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 secon (available via the params object) called `id`. For example given the URL `/product/MacBook`, the fo "MacBook" to the response:

```
class ProductController {
    def index() { render params.id }
}
```

You can of course construct more complex examples of mappings. For example the traditional blog URL
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 for
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
this technique:

```
static mappings = {
    "/$controller/$action?/$id?"()
}
```

Here the name of the controller, action and id are implicitly obtained from the variables control
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 }
    }
}
```

351

Optional Variables

Another characteristic of the default mapping is the ability to append a ? at the end of a variable to make
further example this technique could be applied to the blog URL mapping to have more flexible linking:

```
static mappings = {
    "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

With this mapping all of these URLs would match with only the relevant parameters being populated in the

```
/graemerocher/2007/01/10/my_funky_blog_entry
/graemerocher/2007/01/10
/graemerocher/2007/01
/graemerocher/2007
/graemerocher
```

Optional File Extensions

If you wish to capture the extension of a particular path, then a special case mapping exists:

```
"/$controller/$action?/$id?(.$format)?"()
```

By adding the (.$format)? mapping you can access the file extension using the response.format

```
def index() {
    render "extension is ${response.format}"
}
```

Arbitrary Variables

You can also pass arbitrary parameters from the URL mapping into the controller by just setting them i
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 va
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
with all sorts of logic.

8.4.5 Mapping to Views

You can resolve a URL to a view without a controller or action involved. For example to map the roo
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
}
```

8.4.6 Mapping to Response Codes

Grails also lets you map HTTP response codes to controllers, actions or views. Just use a method name t
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 illegal
`ErrorsController`, a `NullPointerException` will be handled by the `nullPointer` action, a
be handled by the `customException` action. Other exceptions will be handled by the catch
`/errors/serverError` view.

You can access the exception from your custom error handing view or controller action using the reques
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 u
> `StackOverflowException`.

8.4.7 Mapping to HTTP methods

URL mappings can also be configured to map based on the HTTP method (GET, POST, PUT or DELETE
RESTful APIs and for restricting mappings based on HTTP method.

As an example the following mappings provide a RESTful API URL mappings for the `ProductContro`

```
static mappings = {
    "/product/$id"(controller:"product", action: "update", method: "PUT")
}
```

8.4.8 Mapping Wildcards

Grails' URL mappings mechanism also supports wildcard mappings. For example consider the following m

```
static mappings = {
    "/images/*.jpg"(controller: "image")
}
```

This mapping will match all paths to images such as /image/logo.jpg. Of course you can achiev
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. E
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

```
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 mappin
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`.

8.4.9 Automatic Link Re-Writing

Another great feature of URL mappings is that they automatically customize the behaviour of the link
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.
as the blog one from an earlier section:

```
static mappings = {
    "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

357

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>
```

8.4.10 Applying Constraints

URL Mappings also support Grails' unified [validation constraints](#) mechanism, which lets you further "c matched. For example, if we revisit the blog sample code from earlier, the mapping currently looks like thi

```
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 Mappi
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 pat
that burden later on.

8.4.11 Named URL Mappings

URL Mappings also support named mappings, that is mappings which have a name associated with them.
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:

359

```
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 attrs a
will be applied directly to the href, not passed through to be used as request parameters.

```
<link:accountDetails attrs="[class: 'fancy']" acctNumber="8675309">
    Show Account
</link:accountDetails>
```

That would result in:

361

```
<a href="/details/8675309" class="fancy">Show Account</a>
```

8.4.12 Customizing URL Formats

The default URL Mapping mechanism supports camel case names in the URLs. The default URL for ac
addNumbers in a controller named MathHelperController would be something like /mathF
Grails allows for the customization of this pattern and provides an implementation which replaces the cam
hyphenated convention that would support URLs like /math-helper/add-numbers. To enable hy
value of "hyphenated" to the grails.web.url.converter property in grails-app/conf/Conf

```
// grails-app/conf/Config.groovy

grails.web.url.converter = 'hyphenated'
```

Arbitrary strategies may be plugged in by providing a class which implements the [UrlConverter](link) interface a
that class to the Spring application context with the bean name of grails.web.UrlConverter.BEA
a bean in the context with that name, it will be used as the default converter and there is no need t
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
    }
}
```

```
// grails-app/conf/spring/resources.groovy

beans = {
    "${grails.web.UrlConverter.BEAN_NAME}"(com.myapplication.MyUrlConverterImpl)
}
```

8.4.13 Namespaced Controllers

If an application defines multiple controllers with the same name in different packages, the controller namespace. The way to define a namespace for a controller is to define a static property named namespace assign a String to the property that represents the namespace.

```groovy
// grails-app/controllers/com/app/reporting/AdminController.groovy
package com.app.reporting

class AdminController {

static namespace = 'reports'

// …
}
```

```groovy
// grails-app/controllers/com/app/security/AdminController.groovy
package com.app.security

class AdminController {

static namespace = 'users'

// …
}
```

When defining url mappings which should be associated with a namespaced controller, the namespace of the URL mapping.

```groovy
// grails-app/conf/UrlMappings.groovy
class UrlMappings {

static mappings = {
        '/userAdmin' {
            controller = 'admin'
            namespace = 'users'
        }

'/reportAdmin' {
            controller = 'admin'
            namespace = 'reports'
        }

"/$namespace/$controller/$action?"()
    }
}
```

Reverse URL mappings also require that the `namespace` be specified.

```
<g:link controller="admin" namespace="reports">Click For Report Admin</g:link>
<g:link controller="admin" namespace="users">Click For User Admin</g:link>
```

When resolving a URL mapping (forward or reverse) to a namespaced controller, a mapping will only n has been provided. If the application provides several controllers with the same name in different package be defined without a `namespace` property. If there are multiple controllers with the same name that do n property, the framework will not know how to distinguish between them for forward or reverse mapping re

It is allowed for an application to use a plugin which provides a controller with the same name as a c application and for neither of the controllers to define a `namespace` property as long as the controllers For example, an application may include a controller named `com.accounting.ReportingControl` may use a plugin which provides a controller named `com.humanresources.ReportingControl` that is the URL mapping for the controller provided by the plugin needs to be explicit in specifying that th `ReportingController` which is provided by the plugin.

See the following example.

```
static mappings = {
    "/accountingReports" {
        controller = "reporting"
    }
    "/humanResourceReports" {
        controller = "reporting"
        plugin = "humanResources"
    }
}
```

With that mapping in place, a request to `/accountingReports` will be handled by the `Reporting` defined in the application. A request to `/humanResourceReports` will be handled by the `Reportin` provided by the `humanResources` plugin.

There could be any number of `ReportingController` controllers provided by any number of pl provide more than one `ReportingController` even if they are defined in separate packages.

Assigning a value to the `plugin` variable in the mapping is only required if there are multiple control available at runtime provided by the application and/or plugins. If the `humanResource` `ReportingController` and there is no other `ReportingController` available at runtime, the f work.

```
static mappings = {
    "/humanResourceReports" {
        controller = "reporting"
    }
}
```

It is best practice to be explicit about the fact that the controller is being provided by a plugin.

8.5 Filters

Although Grails [controllers](#) support fine grained interceptors, these are only really useful when applied
become difficult to manage with larger applications. Filters on the other hand can be applied across a who
URI space or to a specific action. Filters are far easier to plugin and maintain completely separately to y
and are useful for all sorts of cross cutting concerns such as security, logging, and so on.

8.5.1 Applying Filters

To create a filter create a class that ends with the convention `Filters` in the `grails-app/conf` di
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
using named arguments. For example to define a filter that applies to all controllers and all actions you can

```
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

365

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 regex is
- `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.fin`
- `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*'

```
actionBeginningWithBButNotBad(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 order in
To control the order of execution between `Filters` classes, you can use the `dependsOn` prop
[dependencies](#) section.

> ⚠ Note: When exclude patterns are used they take precedence over the matching patterns. For
> action is 'b*' and actionExclude is 'bad*' then actions like 'best' and 'bien' will have that filter
> actions like 'bad' and 'badlands' will not.

8.5.2 Filter Types

367

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 and the action should not execute

- after - Executed after an action. Takes a first argument as the view model to allow modificat... rendering the view

- afterView - Executed after view rendering. Takes an Exception as an argument which will be n... occurs during processing. Note: this Closure is called before the layout is applied.

For example to fulfill the common simplistic authentication use case you could define a filter as follows:

```
class SecurityFilters {
    def filters = {
        loginCheck(controller: '*', action: '*') {
            before = {
                if (!session.user && !actionName.equals('login')) {
                    redirect(action: 'login')
                    return false
                }
            }
        }
    }
}
```

Here the `loginCheck` filter uses a `before` interceptor to execute a block of code that checks if a user i... redirects to the login action. Note how returning false ensure that the action itself is not executed.

Here's a more involved example that demonstrates all three filter types:

```groovy
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_NU

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: $controllerNam
                "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 a̶
you need to add or remove items from the model map you can do that in the `after` filter.

8.5.3 Variables and Scopes

Filters support all the common properties available to [controllers](#) and [tag libraries](#), plus the application con

- [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

8.5.4 Filter Dependencies

In a `Filters` class, you can specify any other `Filters` classes that should first be executed using th
This is used when a `Filters` class depends on the behavior of another `Filters` class (e.g. sett
modifying the request/session, etc.) and is defined as an array of `Filters` classes.

Take the following example `Filters` classes:

```
class MyFilters {
    def dependsOn = [MyOtherFilters]

def filters = {
        checkAwesome(uri: "/*") {
            before = {
                if (request.isAwesome) { // do something awesome }
            }
        }

checkAwesome2(uri: "/*") {
            before = {
                if (request.isAwesome) { // do something else awesome }
            }
        }
    }
}
```

```
class MyOtherFilters {
    def filters = {
        makeAwesome(uri: "/*") {
            before = {
                request.isAwesome = true
            }
        }
        doNothing(uri: "/*") {
            before = {
                // do nothing
            }
        }
    }
}
```

MyFilters specifically `dependsOn` MyOtherFilters. This will cause all the filters in MyOtherFilters v
current request to be executed before those in MyFilters. For a request of "/test", which will match the so
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 M
order between `Filters` classes are enabled and the execution order of filters within each `Filters` class

If any cyclical dependencies are detected, the filters with cyclical dependencies will be added to the er
processing will continue. Information about any cyclical dependencies that are detected will be written to t
root logging level is set to at least WARN or configure an appender for the Grai
`org.codehaus.groovy.grails.plugins.web.filters.FiltersGrailsPlugin)` wl
dependency issues.

8.6 Ajax

Ajax is the driving force behind the shift to richer web applications. These types of applications in general
dynamic frameworks written in languages like [Groovy] and [Ruby] Grails provides support for building Aja
Ajax tag library. For a full list of these see the Tag Library Reference.

> ⚠ Note: JavaScript examples use the jQuery library.

8.6.1 Ajax Support

By default Grails ships with the [jQuery] library, but through the [Plugin system] provides support for ot
[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 y

```
<g:javascript library="jquery" />
```

You can replace `jQuery` with any other library supplied by a plugin you have installed. This works beca
adaptive tag libraries. Thanks to Grails' plugin system there is support for a number of different Ajax lik
limited to):

* jQuery
* Prototype
* Dojo
* YUI
* MooTools

8.6.1.1 Remoting Linking

Remote content can be loaded in a number of ways, the most commons way is through the [remoteLink]
creation of HTML anchor tags that perform an asynchronous request and optionally set the response in
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

8.6.1.2 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 th
`deleted"`. This is done by the `update` attribute on the tag, which can also take a Map to indicate wl
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.

373

8.6.1.3 Remote Form Submission

An HTML form can also be submitted asynchronously in one of two ways. Firstly using the [formRemote](#)
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
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>
```

8.6.1.4 Ajax Events

Specific JavaScript can be called if certain events occur, all the events start with the "on" prefix and let
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 whate
events include:

- `onSuccess` - The JavaScript function to call if successful

- `onFailure` - The JavaScript function to call if the call failed

- `onERROR_CODE` - The JavaScript function to call to handle specified error codes (e.g. 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 any

You can simply refer to the `XMLHttpRequest` variable to obtain the request:

```
<g:javascript>
    function fireMe(event) {
        alert("XmlHttpRequest = " + event)
    }
}
</g:javascript>
<g:remoteLink action="example"
              update="success"
              onFailure="fireMe(XMLHttpRequest)">Ajax Link</g:remoteLink>
```

8.6.2 Ajax with Prototype

Grails features an external plugin to add [Prototype](#) support to Grails. To install the plugin, list it in BuildC

```
runtime ":prototype:latest.release"
```

This will download the current supported version of the Prototype plugin and install it into your Grails pro 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

8.6.3 Ajax with Dojo

Grails features an external plugin to add [Dojo](#) support to Grails. To install the plugin, list it in BuildConfig

```
compile ":dojo:latest.release"
```

This will download the current supported version of Dojo and install it into your Grails project. With th
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.

8.6.4 Ajax with GWT

Grails also features support for the [Google Web Toolkit](#) through a plugin. There is comprehensive [docum](#)
Grails wiki.

8.6.5 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 and prog
  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, l
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 typic⟨…⟩ template with the [render](#) method:

```groovy
def showBook() {
    def b = Book.get(params.id)

render(template: "bookTemplate", model: [book: b])
}
```

Calling this on the client involves using the [remoteLink](#) tag:

```groovy
<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 programmatical⟨…⟩ with Grails you would typically use Grails' [JSON marshalling](#) capability:

```groovy
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(data) {
    $("#book" + data.id + "_title").html( data.title );
}
</g:javascript>
<g:remoteLink action="showBook" id="${book.id}" onSuccess="updateBook(data)">
    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(data) {
    var id = $(data).find("book").attr("id");
    $("#book" + id + "_title").html( $(data).find("title").text() );
}
</g:javascript>
<g:remoteLink action="showBook" id="${book.id}" onSuccess="updateBook(data)">
    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 exa
below:

```
def showBook() {
    def b = Book.get(params.id)

response.contentType = "text/javascript"
    String title = b.title.encodeAsJavaScript()
    render "$('#book${b.id}_title').html('${title}');"
}
```

The important thing to remember is to set the `contentType` to `text/javascript`. If you use Pr
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 o
server. This is one of the reasons Rails has something like RJS. Although Grails does not currently have a
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
method to `HttpServletRequest` which can be used to identify Ajax requests. For example you coul
using a template for Ajax requests or the full page for regular 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]
    }
}
```

8.7 Content Negotiation

Grails has built in support for [Content negotiation](#) using either the HTTP `Accept` header, an explicit for
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 wi
Grails comes configured with a number of different content types within `grails-app/conf/Conf`
`grails.mime.types` setting:

```
grails.mime.types = [ // the first one is the default format
    all:           '*/*', // 'all' maps to '*' or the first available format in w
    atom:          'application/atom+xml',
    css:           'text/css',
    csv:           'text/csv',
    form:          'application/x-www-form-urlencoded',
    html:          ['text/html','application/xhtml+xml'],
    js:            'text/javascript',
    json:          ['application/json', 'text/json'],
    multipartForm: 'multipart/form-data',
    rss:           'application/rss+xml',
    text:          'text/plain',
    hal:           ['application/hal+json','application/hal+xml'],
    xml:           ['text/xml', 'application/xml']
]
```

The above bit of configuration allows Grails to detect to format of a request containing either the 'text/
media types as simply 'xml'. You can add your own types by simply adding new entries into the map. T
format.

Content Negotiation using the format parameter

Let's say a controller action can return a resource in a variety of formats: HTML, XML, and JSON. What f
The easiest and most reliable way for the client to control this is through a `format` URL parameter.

So if you, as a browser or some other client, want a resource as XML, you can use a URL like this:

```
http://my.domain.org/books?format=xml
```

The result of this on the server side is a `format` property on the `response` object with the value xm
controller action to return XML based on this property, but you can also make use of the controller-sp
method:

```
import grails.converters.JSON
import grails.converters.XML

class BookController {

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

withFormat {
            html bookList: books
            json { render books as JSON }
            xml { render books as XML }
            '*' { render books as JSON }
        }
    }
}
```

In this example, Grails will only execute the block inside `withFormat()` that matches the requeste[...] preferred format is `html` then Grails will execute the `html()` call only. Each 'block' can either [...] corresponding view (as we are doing for 'html' in the above example) or a closure. The closure can co[...] code, for example it can return a model or render content directly.

When no format matches explicitly, a **(wildcard) block can be used to handle all other formats.**

There is a special format, "all", that is handled differently from the explicit formats. If "all" is [...] happens through the Accept header - see below), then the first block of `withFormat()` is exec[...] (wildcard) block available.

You should not add an explicit "all" block. In this example, a format of "all" will trigger the `html` handle[...] and there is no `*` block).

```
withFormat {
            html bookList: books
            json { render books as JSON }
            xml { render books as XML }
        }
```

> ⚠ When using [withFormat](#) make sure it is the last call in your controller action as the return v[...]
> `withFormat` method is used by the action to dictate what happens next.

Using the Accept header

Every incoming HTTP request has a special [Accept](#) header that defines what media types (or mime types[...] older browsers this is typically:

381

```
*/*
```

which simply means anything. However, newer browsers send more interesting values such as this one sen

```
text/xml, application/xml, application/xhtml+xml, text/html;q=0.9,
    text/plain;q=0.8, image/png, */*;q=0.5
```

This particular accept header is unhelpful because it indicates that XML is the preferred response format
expecting HTML. That's why Grails ignores the accept header by default for browsers. However, non-bro
more specific in their requirements and can send accept headers such as

```
application/json
```

As mentioned the default configuration in Grails is to ignore the accept header for browsers. This is d
setting `grails.mime.disable.accept.header.userAgents`, which is configured to det
engines and ignore their ACCEPT headers. This allows Grails' content negotiation to continue to work for

```
grails.mime.disable.accept.header.userAgents = ['Gecko', 'WebKit', 'Presto', 'Tri
```

For example, if it sees the accept header above ('application/json') it will set `format` to `json` as you'd e
works with the `withFormat()` method in just the same way as when the `format` URL parameter i
parameter takes precedence).

An accept header of '*/*' results in a value of `all` for the `format` property.

> ⚠ If the accept header is used but contains no registered content types, Grails will assume a brok
> is making the request and will set the HTML format - note that this is different from ho
> content negotiation modes work as those would activate the "all" format!

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 1 CONTENT_TYPE header and is typically used to detect if the incoming request can be parsed into X response format uses the file extension, format parameter or ACCEPT header to attempt to deliver an ap client.

The [withFormat](#) available on controllers deals specifically with the response format. If you wish to add request format then you can do so using a separate `withFormat` method available on the 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 speci 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
```

This works as a result of the default URL Mapping definition which is:

```
"/$controller/$action?/$id?(.$format)?"{
```

Note the inclusion of the `format` variable in the path. If you do not wish to use content negotiation v simply remove this part of the URL mapping:

```
"/$controller/$action?/$id?"{
```

Testing Content Negotiation

To test content negotiation in a unit or integration test (see the section on [Testing](#)) you can either manipu 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
}
```

9 Web Services

Web Services are all about providing a web API onto your web application and are typically implemented

9.1 REST

REST is not really a technology in itself, but more an architectural pattern. REST is very simple and just in or JSON as a communication medium, combined with URL patterns that are "representational" of the HTTP methods such as GET, PUT, POST and DELETE.

Each HTTP method maps to an action type. For example GET for retrieving data, POST for creating data, on.

Grails includes flexible features that make it easy to create RESTful APIs. Creating a RESTful resource line of code, as demonstrated in the next section.

9.1.1 Domain classes as REST resources

The easiest way to create a RESTful API in Grails is to expose a domain class as a REST resource. This ca `grails.rest.Resource` transformation to any domain class:

```groovy
import grails.rest.*

@Resource(uri='/books')
class Book {

String title

static constraints = {
        title blank:false
    }
}
```

Simply by adding the `Resource` transformation and specifying a URI, your domain class will automa REST resource in either XML or JSON formats. The transformation will automatically register the r mapping and create a controller called `BookController`.

You can try it out by adding some test data to `BootStrap.groovy`:

```groovy
def init = { servletContext ->

new Book(title:"The Stand").save()
        new Book(title:"The Shining").save()
    }
```

And then hitting the URL http://localhost:8080/myapp/books/1, which will render the response like:

```
<?xml version="1.0" encoding="UTF-8"?>
<book id="1">
    <title>The Stand</title>
</book>
```

If you change the URL to `http://localhost:8080/myapp/books/1.json` you will get a JSON

```
{"id":1,"title":"The Stand"}
```

If you wish to change the default to return JSON instead of XML, you can do this by setting the `fc`
`Resource` transformation:

```
import grails.rest.*

@Resource(uri='/books', formats=['json', 'xml'])
class Book {
    …
}
```

With the above example JSON will be prioritized. The list that is passed should contain the names of the
should expose. The names of formats are defined in the `grails.mime.types` setting of `Config.gro`

```
grails.mime.types = [
    …
    json:           ['application/json', 'text/json'],
    …
    xml:            ['text/xml', 'application/xml']
]
```

See the section on [Configuring Mime Types](#) in the user guide for more information.

Instead of using the file extension in the URI, you can also obtain a JSON response using the ACCEPT h
using the Unix `curl` tool:

387

```
$ curl -i -H "Accept: application/json" localhost:8080/myapp/books/1
{"id":1,"title":"The Stand"}
```

This works thanks to Grails' [Content Negotiation](#) features.

You can create a new resource by issuing a POST request:

```
$ curl -i -X POST -H "Content-Type: application/json" -d '{"title":"Along Came A
localhost:8080/myapp/books
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
...
```

Updating can be done with a PUT request:

```
$ curl -i -X PUT -H "Content-Type: application/json" -d '{"title":"Along Came A S
localhost:8080/myapp/books/1
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
...
```

Finally a resource can be deleted with DELETE request:

```
$ curl -i -X DELETE localhost:8080/myapp/books/1
HTTP/1.1 204 No Content
Server: Apache-Coyote/1.1
...
```

As you can see, the Resource transformation enables all of the HTTP method verbs on the resource
read-only capabilities by setting the readOnly attribute to true:

```
import grails.rest.*
@Resource(uri='/books', readOnly=true)
class Book {
    …
}
```

In this case POST, PUT and DELETE requests will be forbidden.

9.1.2 Mapping to REST resources

If you prefer to keep the declaration of the URL mapping in your `UrlMappings.groovy` file then s
attribute of the `Resource` transformation and adding the following line to `UrlMappings.groovy` wil

```
"/books"(resources:"book")
```

Extending your API to include more end points then becomes trivial:

```
"/books"(resources:"book") {
    "/publisher"(controller:"publisher", method:"GET")
}
```

The above example will expose the URI `/books/1/publisher`.

A more detailed explanation on [creating RESTful URL mappings](#) can be found in the [URL Mappings secti](#)

9.1.3 Linking to REST resources

The `link` tag offers an easy way to link to any domain class resource:

```
<g:link resource="${book}">My Link</g:link>
```

389

However, currently you cannot use g:link to link to the DELETE action and most browsers do not supp
method directly.

The best way to accomplish this is to use a form submit:

```
<form action="/book/2" method="post">
    <input type="hidden" name="_method" value="DELETE"/>
</form>
```

Grails supports overriding the request method via the hidden _method parameter. This is for browser con
is useful when using restful resource mappings to create powerful web interfaces. To make a link fire th
capture all click events for links with a `data-method` attribute and issue a form submit via javascript.

9.1.4 Versioning REST resources

A common requirement with a REST API is to expose different versions at the same time. There are
achieved in Grails.

Versioning using the URI

A common approach is to use the URI to version APIs (although this approach is discouraged in fav
example, you can define the following URL mappings:

```
"/books/v1"(resources:"book", namespace:'v1')
"/books/v2"(resources:"book", namespace:'v2')
```

That will match the following controllers:

```
package myapp.v1

class BookController {
    static namespace = 'v1'
}

package myapp.v2

class BookController {
    static namespace = 'v2'
}
```

This approach has the disadvantage of requiring two different URI namespaces for your API.

Versioning with the Accept-Version header

As an alternative Grails supports the passing of an `Accept-Version` header from clients. For exa
following URL mappings:

```
"/books"(version:'1.0', resources:"book", namespace:'v1')
"/books"(version:'2.0', resources:"book", namespace:'v2')
```

Then in the client simply pass which version you need using the `Accept-Version` header:

```
$ curl -i -H "Accept-Version: 1.0" -X GET http://localhost:8080/myapp/books
```

Versioning using Hypermedia / Mime Types

Another approach to versioning is to use Mime Type definitions to declare the version of your custom me
on "Hypermedia as the Engine of Application State" for more information about Hypermedia cond
`Config.groovy` you can declare a custom Mime Type for your resource that includes a version parame

```
grails.mime.types = [
    all: '*/*',
    book: "application/vnd.books.org.book+json;v=1.0",
    bookv2: "application/vnd.books.org.book+json;v=2.0",
    …
}
```

> ⛔ It is critical that place your new mime types after the 'all' Mime Type because if the Content
> request cannot be established then the first entry in the map is used for the response. If you
> new Mime Type at the top then Grails will always try and send back your new Mime
> requested Mime Type cannot be established.

Then override the renderer (see the section on "Customizing Response Rendering" for more information
send back the custom Mime Type in `grails-app/conf/spring/resourses.groovy`:

```
import grails.rest.render.json.*
import org.codehaus.groovy.grails.web.mime.*

beans = {
    bookRendererV1(JsonRenderer, myapp.v1.Book, new MimeType("application/vnd.boo
[v:"1.0"]))
    bookRendererV2(JsonRenderer, myapp.v2.Book, new MimeType("application/vnd.boo
[v:"2.0"]))
}
```

Then using the `Accept` header you can specify which version you need using the Mime Type:

```
$ curl -i -H "Accept: application/vnd.books.org.book+json;v=1.0" -X GET
http://localhost:8080/myapp/books
```

9.1.5 Implementing REST controllers

The `Resource` transformation is a quick way to get started, but typically you'll want to customize rendering of the response or extend the API to include additional actions.

9.1.5.1 Extending the RestfulController super class

The easiest way to get started doing so is to create a new controller for your resour grails.rest.RestfulController super class. For example:

```
class BookController extends RestfulController {
    static responseFormats = ['json', 'xml']
    BookController() {
        super(Book)
    }
}
```

To customize any logic you can just override the appropriate action. The following table provides the na and the URIs they map to:

HTTP Method	URI	Controller Action
GET	/books	index
GET	/books/create	create
POST	/books	save
GET	/books/${id}	show
GET	/books/${id}/edit	edit
PUT	/books/${id}	update
DELETE	/books/${id}	delete

⚠️ Note that the `create` and `edit` actions are only needed if the controller exposes an HTML i

As an example, if you have a nested resource then you would typically want to query both the parent and
example, given the following URL mapping:

```
"/authors"(resources:'author') {
    "/books"(resources:'book')
}
```

You could implement the nested controller as follows:

```
class BookController extends RestfulController {
    static responseFormats = ['json', 'xml']
    BookController() {
        super(Book)
    }

@Override
    protected Book queryForResource(Serializable id) {
        Book.where {
            id == id && author.id = params.authorId
        }.find()
    }

}
```

The example above subclasses `RestfulController` and overrides the protected `queryForResour`
the query for the resource to take into account the parent resource.

Customizing Data Binding In A RestfulController Subclass

The RestfulController class contains code which does data binding for actions like `save` and `update`. getObjectToBind() method which returns a value which will be used as the source for data binding. action does something like this...

```
class RestfulController<T> {
def update() {
        T instance = // retrieve instance from the database...
instance.properties = getObjectToBind()
// …
    }
// …
}
```

By default the getObjectToBind() method returns the [request](#) object. When the `request` object source, if the request has a body then the body will be parsed and its contents will be used to do the da request parameters will be used to do the data binding. Subclasses of RestfulController may override the method and return anything that is a valid binding source, including a [Map](#) or a [DataBindingSource](#). For n request is appropriate but the getObjectToBind() method allows for changing that behavior where de

Using custom subclass of RestfulController with Resource annotation

You can also customize the behaviour of the controller that backs the Resource annotation.

The class must provide a constructor that takes a domain class as it's argument. The second constructor i Resource annotation with readOnly=true.

This is a template that can be used for subclassed RestfulController classes used in Resource annotations:

```
class SubclassRestfulController<T> extends RestfulController<T> {
    SubclassRestfulController(Class<T> domainClass) {
        this(domainClass, false)
    }
SubclassRestfulController(Class<T> domainClass, boolean readOnly) {
        super(domainClass, readOnly)
    }
}
```

You can specify the super class of the controller that backs the Resource annotation with the superClas

```
import grails.rest.*

@Resource(uri='/books', superClass=SubclassRestfulController)
class Book {
String title

static constraints = {
        title blank:false
    }
}
```

9.1.5.2 Implementing REST Controllers Step by Step

If you don't want to take advantage of the features provided by the `RestfulController` super class, each HTTP verb yourself manually. The first step is to create a controller:

```
$ grails create-controller book
```

Then add some useful imports and enable readOnly by default:

```
import grails.transaction.*
import static org.springframework.http.HttpStatus.*
import static org.springframework.http.HttpMethod.*

@Transactional(readOnly = true)
class BookController {
    …
}
```

Recall that each HTTP verb matches a particular Grails action according to the following conventions:

HTTP Method	URI	Controller Action
GET	/books	index
GET	/books/${id}	show
GET	/books/create	create
GET	/books/${id}/edit	edit
POST	/books	save
PUT	/books/${id}	update
DELETE	/books/${id}	delete

> ⚠ The 'create' and 'edit' actions are already required if you plan to implement an HTML interf
> REST resource. They are there in order to render appropriate HTML forms to create and edit
> If this is not a requirement they can be discarded.

The key to implementing REST actions is the [respond](#) method introduced in Grails 2.3. The `respond` m
most appropriate response for the requested content type (JSON, XML, HTML etc.)

Implementing the 'index' action

For example, to implement the `index` action, simply call the `respond` method passing the list of objects

```
def index(Integer max) {
    params.max = Math.min(max ?: 10, 100)
    respond Book.list(params), model:[bookCount: Book.count()]
}
```

Note that in the above example we also use the `model` argument of the `respond` method to supply the
required if you plan to support pagination via some user interface.

The `respond` method will, using [Content Negotiation](#), attempt to reply with the most appropriate respo
requested by the client (via the ACCEPT header or file extension).

If the content type is established to be HTML then a model will be produced such that the action above w
writing:

```
def index(Integer max) {
    params.max = Math.min(max ?: 10, 100)
    [bookList: Book.list(params), bookCount: Book.count()]
}
```

By providing an `index.gsp` file you can render an appropriate view for the given model. If the conten than HTML then the `respond` method will attempt to lookup an appropriate `grails.rest.render.` is capable of rendering the passed object. This is done by inspecting the `grails.rest.render.Rend`

By default there are already renderers configured for JSON and XML, to find out how to register a custom on "Customizing Response Rendering".

Implementing the 'show' action

The `show` action, which is used to display and individual resource by id, can be implemented in o (excluding the method signature):

```
def show(Book book) {
    respond book
}
```

By specifying the domain instance as a parameter to the action Grails will automatically attempt to loo using the `id` parameter of the request. If the domain instance doesn't exist, then `null` will be pass `respond` method will return a 404 error if null is passed otherwise once again it will attempt to render a the format is HTML then an appropriate model will produced. The following action is functionally equival

```
def show(Book book) {
    if(book == null) {
        render status:404
    }
    else {
        return [book: book]
    }
}
```

Implementing the 'save' action

The `save` action creates new resource representations. To start off, simply define an action that accep argument and mark it as `Transactional` with the `grails.transaction.Transactional` trans

```
@Transactional
def save(Book book) {
    …
}
```

Then the first thing to do is check whether the resource has any [validation errors](#) and if so respond with the

```
if(book.hasErrors()) {
    respond book.errors, view:'create'
}
else {
    …
}
```

In the case of HTML the 'create' view will be rendered again so the user can correct the invalid input. In (JSON, XML etc.), the errors object itself will be rendered in the appropriate format and (UNPROCESSABLE_ENTITY) returned.

If there are no errors then the resource can be saved and an appropriate response sent:

```
book.save flush:true
    withFormat {
        html {
            flash.message = message(code: 'default.created.message', args: [messa
'book.label', default: 'Book'), book.id])
            redirect book
        }
        '*' { render status: CREATED }
    }
```

In the case of HTML a redirect is issued to the originating resource and for other formats a status cod returned.

Implementing the 'update' action

The `update` action updates an existing resource representations and is largely similar to the `save` action signature:

```
@Transactional
def update(Book book) {
    …
}
```

If the resource exists then Grails will load the resource, otherwise null we passed. In the case of null, you s

```
if(book == null) {
        render status: NOT_FOUND
    }
    else {
        …
    }
```

Then once again check for errors [validation errors](#) and if so respond with the errors:

```
if(book.hasErrors()) {
    respond book.errors, view:'edit'
}
else {
    …
}
```

In the case of HTML the 'edit' view will be rendered again so the user can correct the invalid input. In (JSON, XML etc.) the errors object itself will be rendered in the appropriate format and (UNPROCESSABLE_ENTITY) returned.

If there are no errors then the resource can be saved and an appropriate response sent:

```
book.save flush:true
withFormat {
    html {
        flash.message = message(code: 'default.updated.message', args: [message(c
default: 'Book'), book.id])
        redirect book
    }
    '*' { render status: OK }
}
```

In the case of HTML a redirect is issued to the originating resource and for other formats a status code of 2

Implementing the 'delete' action

The `delete` action deletes an existing resource. The implementation is largely similar to the upd `delete()` method is called instead:

```
book.delete flush:true
withFormat {
    html {
        flash.message = message(code: 'default.deleted.message', args: [message(c
default: 'Book'), book.id])
        redirect action:"index", method:"GET"
    }
    '*'{ render status: NO_CONTENT }
}
```

Notice that for an HTML response a redirect is issued back to the index action, whilst for other content t (NO_CONTENT) is returned.

9.1.5.3 Generating a REST controller using scaffolding

To see some of these concepts in action and help you get going the [Scaffolding plugin](#), version 2.0 and abc ready controller for you, simply run the command:

```
$ grails generate-controller [Domain Class Name]
```

9.1.6 Customizing Response Rendering

There are several ways to customize response rendering in Grails.

9.1.6.1 Customizing the Default Renderers

The default renderers for XML and JSON can be found in the grails.rest. grails.rest.render.json packages respectively. These use the Grails converters (grails.c grails.converters.JSON) by default for response rendering.

You can easily customize response rendering using these default renderers. A common change you may w or exclude certain properties from rendering.

Including or Excluding Properties from Rendering

As mentioned previously, Grails maintains a registry of grails.rest.render.Renderer instance configured renderers and the ability to register or override renderers for a given domain class or even fc classes. To include a particular property from rendering you need to register a custom renderer grails-app/conf/spring/resources.groovy:

```
import grails.rest.render.xml.*

beans = {
    bookRenderer(XmlRenderer, Book) {
        includes = ['title']
    }
}
```

> ⚠ The bean name is not important (Grails will scan the application context for all registere
> beans), but for organizational and readability purposes it is recommended you name it
> meaningful.

To exclude a property, the `excludes` property of the `XmlRenderer` class can be used:

```
import grails.rest.render.xml.*

beans = {
    bookRenderer(XmlRenderer, Book) {
        excludes = ['isbn']
    }
}
```

Customizing the Converters

As mentioned previously, the default renders use the `grails.converters` package under the covers.
covers they essentially do the following:

```
import grails.converters.*

…
render book as XML

// or render book as JSON
```

Why the separation between converters and renderers? Well a renderer has more flexibility to use whate
you chose. When implementing a custom renderer you could use Jackson, Gson or any Java library to
Converters on the other hand are very much tied to Grails' own marshalling implementation.

9.1.6.2 Registering Custom Objects Marshallers

Grails' Converters feature the notion of an [ObjectMarshaller](#) and each type can have a registered `Object` register custom `ObjectMarshaller` instances to completely customize response rendering. For exam following in `BootStrap.init`:

```
XML.registerObjectMarshaller Book, { Book book, XML xml ->
  xml.attribute 'id', book.id
  xml.build {
    title(book.title)
  }
}
```

You can customize the formatting of an individual value this way too. For example the [JodaTime plug](#) support rendering of JodaTime dates in JSON output:

```
JSON.registerObjectMarshaller(DateTime) {
    return it?.toString("yyyy-MM-dd'T'HH:mm:ss'Z'")
}
```

In the case of JSON it's often simple to use a map to customize output:

```
JSON.registerObjectMarshaller(Book) {
  def map= [:]
  map['titl'] = it.title
  map['auth'] = it.author
  return map
}
```

Registering Custom Marshallers via Spring

Note that if you have many custom marshallers it is recommended you split the registration of these into a

```
class CustomMarshallerRegistrar {

@javax.annotation.PostConstruct
    void registerMarshallers() {
        JSON.registerObjectMarshaller(DateTime) {
            return it?.toString("yyyy-MM-dd'T'HH:mm:ss'Z'")
        }
    }
}
```

Then define this class as Spring bean in `grails-app/conf/spring/resources.groovy`:

```
beans = {
    myCustomMarshallerRegistrar(CustomMarshallerRegistrar)
}
```

The `PostConstruct` annotation will get triggered on startup of your application.

9.1.6.3 Using Named Configurations for Object Marshallers

It is also possible to register named configurations. For example:

```
XML.createNamedConfig('publicApi') {
  it.registerObjectMarshaller(Book) { Book book, XML xml ->
    // do public API
  }
}
XML.createNamedConfig('adminApi') {
  it.registerObjectMarshaller(Book) { Book book, XML xml ->
    // do admin API
  }
}
```

Then when you use either the `render` or `respond` methods you can wrap the call in a named conf
customize rendering per request:

403

```
XML.use( isAdmin ? 'adminApi' : 'publicApi') {
    render book as XML
}
```

or

```
XML.use( isAdmin ? 'adminApi' : 'publicApi') {
    respond book
}
```

9.1.6.4 Implementing the ObjectMarshaller Interface

For more complex marshallers it is recommended you implement the [ObjectMarshaller](#) interface. For
class:

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

By default the output when using:

```
render book as XML
```

Would look like:

```
<book id="1">
    <title>The Stand</title>
</book>
```

To write a custom marshaller you can do the following:

```
class BookMarshaller implements ObjectMarshaller<XML> {

public boolean supports(Object object) {
    return object instanceof Book
  }

public void marshalObject(Object object, XML converter) {
    Book book  = (Book)object
    converter.chars book.title
  }
}
```

And then register the marshaller with:

```
XML.registerObjectMarshaller(new BookMarshaller())
```

With the custom `ObjectMarshaller` in place, the output is now:

```
<book>The Stand</book>
```

Customizing the Name of the Root Element

If you wish the customize the name of the surrounding element, you can implement [NameAwareMarshalle](#)

```
class BookMarshaller implements ObjectMarshaller<XML>,NameAwareMarshaller {

...

String getElementName(Object o) {
    return 'custom-book'
  }

}
```

405

With the above change the output would now be:

```
<custom-book>The Stand</custom-book>
```

Outputting Markup Using the Converters API or Builder

With the passed Converter object you can explicitly code to the Converters API to stream markup to the re

```
public void marshalObject(Object object, XML converter) {
  Book book = (Book)object

converter.attribute 'id', book.id.toString()
  converter.attribute 'date-released', book.dateReleased.toString()

converter.startNode 'title'
  converter.chars book.title
  converter.end()
}
```

The above code results in:

```
<book id="1" date-released="...">
    <title>The Stand</title>
</book>
```

You can also use a builder notation to achieve a similar result (although the builder notation does not work

```
public void marshalObject(Object object, XML converter) {
  Book b = (Book)object

converter.build {
    book(id: b.id) {
        title b.title
    }
  }
}
```

Using the convertAnother Method to Recursively Convert Objects

To create more complex responses you can use the `convertAnother` method to convert associations an

```
public void marshalObject(Object object, XML converter) {
  Book book = (Book)object

converter.startNode 'title'
  converter.chars book.title
  converter.end()

if (book.authors) {
    converter.startNode 'authors'
    for(author in book.authors) {
      converter.convertAnother author
    }
    converter.end()
  }
}
```

9.1.6.5 Implementing a Custom Renderer

If you want even more control of the rendering or prefer to use your own marshalling techniques then you
`Renderer` instance. For example below is a simple implementation that customizes the rendering of the I

```
package myapp
import grails.rest.render.*
import org.codehaus.groovy.grails.web.mime.MimeType

class BookXmlRenderer extends AbstractRenderer<Book> {
    BookXmlRenderer() {
        super(Book, [MimeType.XML,MimeType.TEXT_XML] as MimeType[])
    }

void render(Book object, RenderContext context) {
        context.contentType = MimeType.XML.name

def xml = new groovy.xml.MarkupBuilder(context.writer)
        xml.book(id: object.id, title:object.title)
    }
}
```

The `AbstractRenderer` super class has a constructor that takes the class that it renders and the `MimeT`
(via the ACCEPT header or file extension) for the renderer.

To configure this renderer, simply add it is a bean to `grails-app/conf/spring/resources.grc`

```
beans = {
    bookRenderer(myapp.BookXmlRenderer)
}
```

The result will be that all `Book` instances will be rendered in the following format:

```
<book id="1" title="The Stand"/>
```

> ⚠️ Note that if you change the rendering to a completely different format like the above, then yo
> to change the binding if you plan to support POST and PUT requests. Grails will not automat
> how to bind data from a custom XML format to a domain class otherwise. See the
> "Customizing Binding of Resources" for further information.

Container Renderers

A `grails.rest.render.ContainerRenderer` is a renderer that renders responses for containe
collections etc.). The interface is largely the same as the `Renderer` interface except for
`getComponentType()` method, which should return the "contained" type. For example:

```
class BookListRenderer implements ContainerRenderer<List, Book> {
    Class<List> getTargetType() { List }
    Class<Book> getComponentType() { Book }
    MimeType[] getMimeTypes() { [ MimeType.XML] as MimeType[] }
    void render(List object, RenderContext context) {
        ....
    }
}
```

9.1.6.6 Using GSP to Customize Rendering

You can also customize rendering on a per action basis using Groovy Server Pages (GSP). For exampl
mentioned previously:

```
def show(Book book) {
    respond book
}
```

You could supply a `show.xml.gsp` file to customize the rendering of the XML:

```
<%@page contentType="application/xml"%>
<book id="${book.id}" title="${book.title}"/>
```

9.1.7 Hypermedia as the Engine of Application State

HATEOS, an abbreviation for Hypermedia as the Engine of Application State, is a common pattern appli
that uses hypermedia and linking to define the REST API.

Hypermedia (also called Mime or Media Types) are used to describe the state of a REST resource, and
transition to the next state. The format of the response is typically JSON or XML, although standard form
HAL are frequently used.

9.1.7.1 HAL Support

HAL is a standard exchange format commonly used when developing REST APIs that follow HATEOA
HAL document representing a list of orders can be seen below:

```
{
    "_links": {
        "self": { "href": "/orders" },
        "next": { "href": "/orders?page=2" },
        "find": {
            "href": "/orders{?id}",
            "templated": true
        },
        "admin": [{
            "href": "/admins/2",
            "title": "Fred"
        }, {
            "href": "/admins/5",
            "title": "Kate"
        }]
    },
    "currentlyProcessing": 14,
    "shippedToday": 20,
    "_embedded": {
        "order": [{
            "_links": {
                "self": { "href": "/orders/123" },
                "basket": { "href": "/baskets/98712" },
                "customer": { "href": "/customers/7809" }
            },
            "total": 30.00,
            "currency": "USD",
            "status": "shipped"
        }, {
            "_links": {
                "self": { "href": "/orders/124" },
                "basket": { "href": "/baskets/97213" },
                "customer": { "href": "/customers/12369" }
            },
            "total": 20.00,
            "currency": "USD",
            "status": "processing"
        }]
    }
}
```

Exposing Resources Using HAL

To return HAL instead of regular JSON for a resource you can simply overrid
`grails-app/conf/spring/resources.groovy` with an inst
`grails.rest.render.hal.HalJsonRenderer` (or `HalXmlRenderer` for the XML variation):

```
import grails.rest.render.hal.*
beans = {
    halBookRenderer(HalJsonRenderer, rest.test.Book)
}
```

With the bean in place requesting the HAL content type will return HAL:

```
$ curl -i -H "Accept: application/hal+json" http://localhost:8080/myapp/books/1

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/hal+json;charset=ISO-8859-1

{
  "_links": {
    "self": {
      "href": "http://localhost:8080/myapp/books/1",
      "hreflang": "en",
      "type": "application/hal+json"
    }
  },
  "title": ""The Stand""
}
```

To use HAL XML format simply change the renderer:

```
import grails.rest.render.hal.*
beans = {
    halBookRenderer(HalXmlRenderer, rest.test.Book)
}
```

Rendering Collections Using HAL

To return HAL instead of regular JSON for a list of resources you can simply overr
grails-app/conf/spring/resources.groovy with an insta
grails.rest.render.hal.HalJsonCollectionRenderer:

```
import grails.rest.render.hal.*
beans = {
    halBookCollectionRenderer(HalJsonCollectionRenderer, rest.test.Book)
}
```

With the bean in place requesting the HAL content type will return HAL:

411

```
$ curl -i -H "Accept: application/hal+json" http://localhost:8080/myapp/books
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/hal+json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 17 Oct 2013 02:34:14 GMT

{
  "_links": {
    "self": {
      "href": "http://localhost:8080/myapp/books",
      "hreflang": "en",
      "type": "application/hal+json"
    }
  },
  "_embedded": {
    "book": [
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/myapp/books/1",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "The Stand"
      },
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/myapp/books/2",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "Infinite Jest"
      },
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/myapp/books/3",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "Walden"
      }
    ]
  }
}
```

Notice that the key associated with the list of `Book` objects in the rendered JSON is `book` which is
objects in the collection, namely `Book`. In order to customize the value of this key assign a value to
property on the `HalJsonCollectionRenderer` bean as shown below:

```
import grails.rest.render.hal.*
beans = {
    halBookCollectionRenderer(HalCollectionJsonRenderer, rest.test.Book) {
        collectionName = 'publications'
    }
}
```

With that in place the rendered HAL will look like the following:

```
$ curl -i -H "Accept: application/hal+json" http://localhost:8080/myapp/books
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/hal+json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 17 Oct 2013 02:34:14 GMT

{
  "_links": {
    "self": {
      "href": "http://localhost:8080/myapp/books",
      "hreflang": "en",
      "type": "application/hal+json"
    }
  },
  "_embedded": {
    "publications": [
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/myapp/books/1",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "The Stand"
      },
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/myapp/books/2",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "Infinite Jest"
      },
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/myapp/books/3",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "Walden"
      }
    ]
  }
}
```

Using Custom Media / Mime Types

If you wish to use a custom Mime Type then you first need to declare the
grails-app/conf/Config.groovy:

```
grails.mime.types = [
    all:      "*/*",
    book:     "application/vnd.books.org.book+json",
    bookList: "application/vnd.books.org.booklist+json",
    …
]
```

⛔ It is critical that place your new mime types after the 'all' Mime Type because if the Content
request cannot be established then the first entry in the map is used for the response. If you
new Mime Type at the top then Grails will always try and send back your new Mime
requested Mime Type cannot be established.

Then override the renderer to return HAL using the custom Mime Types:

```
import grails.rest.render.hal.*
import org.codehaus.groovy.grails.web.mime.*

beans = {
    halBookRenderer(HalJsonRenderer, rest.test.Book, new MimeType(
"application/vnd.books.org.book+json", [v:"1.0"]))
    halBookListRenderer(HalJsonCollectionRenderer, rest.test.Book, new MimeType(
"application/vnd.books.org.booklist+json", [v:"1.0"]))
}
```

In the above example the first bean defines a HAL renderer for a single book instance that ret
`application/vnd.books.org.book+json`. The second bean defines the Mime Type used to ren
(in this case `application/vnd.books.org.booklist+json`).

With this in place issuing a request for the new Mime Type returns the necessary HAL:

```
$ curl -i -H "Accept: application/vnd.books.org.book+json" http://localhost:8080/

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/vnd.books.org.book+json;charset=ISO-8859-1


{
  "_links": {
    "self": {
      "href": "http://localhost:8080/myapp/books/1",
      "hreflang": "en",
      "type": "application/vnd.books.org.book+json"
    }
  },
  "title": ""The Stand""
}
```

Customizing Link Rendering

An important aspect of HATEOAS is the usage of links that describe the transitions the client can use to
API. By default the `HalJsonRenderer` will automatically create links for you for associations and to
the "self" relationship).

However you can customize link rendering using the `link` method that is added to all domain
`grails.rest.Resource` or any class annotated with `grails.rest.Linkable`. For example,
modified as follows to provide a new link in the resulting output:

```
def show(Book book) {
    book.link rel:'publisher', href: g.link(resource:"publisher", params:[bookId:
    respond book
}
```

Which will result in output such as:

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/myapp/books/1",
      "hreflang": "en",
      "type": "application/vnd.books.org.book+json"
    }
    "publisher": {
        "href": "http://localhost:8080/myapp/books/1/publisher",
        "hreflang": "en"
    }
  },
  "title": ""The Stand""
}
```

The link method can be passed named arguments that match the properties of the grails.rest.Linl

9.1.7.2 Atom Support

Atom is another standard interchange format used to implement REST APIs. An example of Atom output o

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">

<title>Example Feed</title>
 <link href="http://example.org/"/>
 <updated>2003-12-13T18:30:02Z</updated>
 <author>
   <name>John Doe</name>
 </author>
 <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>

<entry>
   <title>Atom-Powered Robots Run Amok</title>
   <link href="http://example.org/2003/12/13/atom03"/>
   <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
   <updated>2003-12-13T18:30:02Z</updated>
   <summary>Some text.</summary>
 </entry>

</feed>
```

To use Atom rendering again simply define a custom renderer:

```
import grails.rest.render.atom.*
beans = {
    halBookRenderer(AtomRenderer, rest.test.Book)
    halBookListRenderer(AtomCollectionRenderer, rest.test.Book)
}
```

9.1.7.3 Vnd.Error Support

Vnd.Error is a standardised way of expressing an error response.

By default when a validation error occurs when attempting to POST new resources then the errors objec
with a 422 respond code:

```
$ curl -i -H "Accept: application/json"  -H "Content-Type: application/json" -X P
http://localhost:8080/myapp/books

HTTP/1.1 422 Unprocessable Entity
Server: Apache-Coyote/1.1
Content-Type: application/json;charset=ISO-8859-1

{"errors":[{"object":"rest.test.Book", "field":"title", "rejected-value":null, "m
[title] of class [class rest.test.Book] cannot be null"}]}
```

If you wish to change the format to Vnd.Error then si
grails.rest.render.errors.VndErrorJsonRenderer bean
grails-app/conf/spring/resources.groovy:

```
beans = {
    vndJsonErrorRenderer(grails.rest.render.errors.VndErrorJsonRenderer)
    // for Vnd.Error XML format
    vndXmlErrorRenderer(grails.rest.render.errors.VndErrorXmlRenderer)
}
```

Then if you alter the client request to accept Vnd.Error you get an appropriate response:

```
$ curl -i -H "Accept: application/vnd.error+json,application/json" -H "Content-Ty
application/json" -X POST -d "" http://localhost:8080/myapp/books
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/vnd.error+json;charset=ISO-8859-1

[
    {
        "logref": ""book.nullable"",
        "message": "Property [title] of class [class rest.test.Book] cannot be nu
        "_links": {
            "resource": {
                "href": "http://localhost:8080/rest-test/books"
            }
        }
    }
]
```

9.1.8 Customizing Binding of Resources

The framework provides a sophisticated but simple mechanism for binding REST requests to domain objec
One way to take advantage of this is to bind the request property in a controller the properties of a
following XML as the body of the request, the createBook action will create a new Book and assign "
property and "Stephen King" to the authorName property.

```
<?xml version="1.0" encoding="UTF-8"?>
<book>
    <title>The Stand</title>
    <authorName>Stephen King</authorName>
</book>
```

```
class BookController {

def createBook() {
        def book = new Book()
        book.properties = request

// …
    }
}
```

If the root element of the XML document contains an id attribute, the id value will be used to re
persistent instance from the database and then the rest of the document will be bound to the instance. If n
found in the database, the command object reference will be null.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book>
    <title>The Stand</title>
    <authorName>Stephen King</authorName>
</book>
```

Command objects will automatically be bound with the body of the request:

```groovy
class BookController {
    def createBook(BookCommand book) {

// …
    }
}

class BookCommand {
    String title
    String authorName
}
```

If the command object type is a domain class and the root element of the XML document contains an i will be used to retrieve the corresponding persistent instance from the database and then the rest of the d the instance. If no corresponding record is found in the database, the command object reference will be nul

```xml
<?xml version="1.0" encoding="UTF-8"?>
<book id="42">
    <title>Walden</title>
    <authorName>Henry David Thoreau</authorName>
</book>
```

```
    class BookController {
        def updateBook(Book book) {
            // The book will have been retrieved from the database and updated
            // by doing something like this:
            //
            // book == Book.get('42')
            // if(book != null) {
            //     book.properties = request
            // }
            //
            // the code above represents what the framework will
            // have done. There is no need to write that code.

    // ...

    }
    }
```

The data binding depends on an instance of the [DataBindingSource](#) interface created by
[DataBindingSourceCreator](#) interface. The specific implementation of `DataBindingSourceCreator`
the `contentType` of the request. Several implementations are provided to handle common con
implementations will be fine for most use cases. The following table lists the content types which an
framework and which `DataBindingSourceCreator` implementations are used for each. All of the in
in the `org.codehaus.groovy.grails.web.binding.bindingsource` package.

Content Type(s)	Bean Name	DataBindingSourceCreator Impl.
application/xml, text/xml	xmlDataBindingSourceCreator	XmlDataBindingSourceCreator
application/json, text/json	jsonDataBindingSourceCreator	JsonDataBindingSourceCreator
application/hal+json	halJsonDataBindingSourceCreator	HalJsonDataBindingSourceCreator
application/hal+xml	halXmlDataBindingSourceCreator	HalXmlDataBindingSourceCreator

In order to provide your own `DataBindingSourceCreator` for any of those content types, write a
`DataBindingSourceCreator` and register an instance of that class in the Spring application context
of the existing helpers, use the corresponding bean name from above. If you are providing a helper for ;
those accounted for by the core framework, the bean name may be anything that you like but you shoulc
with one of the bean names above.

The `DataBindingSourceCreator` interface defines just 2 methods:

```groovy
package org.grails.databinding.bindingsource

import org.codehaus.groovy.grails.web.mime.MimeType
import org.grails.databinding.DataBindingSource

/**
 * A factory for DataBindingSource instances
 *
 * @since 2.3
 * @see DataBindingSourceRegistry
 * @see DataBindingSource
 *
 */
interface DataBindingSourceCreator {

/**
     * return All of the {link MimeType} supported by this helper
     */
    MimeType[] getMimeTypes()

/**
     * Creates a DataBindingSource suitable for binding bindingSource to bindingT
     *
     * @param mimeType a mime type
     * @param bindingTarget the target of the data binding
     * @param bindingSource the value being bound
     * @return a DataBindingSource
     */
    DataBindingSource createDataBindingSource(MimeType mimeType, Object bindingTa
bindingSource)
}
```

[AbstractRequestBodyDataBindingSourceCreator](#) is an abstract class designed to be extended to si DataBindingSourceCreator classes. Classes which extend AbstractRequestbodyDatabin need to implement a method named createBindingSource which accepts an InputStream as an DataBindingSource as well as implementing the getMimeTypes method DataBindingSourceCreator interface above. The InputStream argument to createBin access to the body of the request.

The code below shows a simple implementation.

```
// MyCustomDataBindingSourceCreator.groovy in
// src/groovy/com/demo/myapp/databinding
package com.demo.myapp.databinding

import org.codehaus.groovy.grails.web.mime.MimeType
import org.grails.databinding.DataBindingSource
import org...databinding.SimpleMapDataBindingSource
import org...databinding.bindingsource.AbstractRequestBodyDataBindingSourceCreato

/**
 * A custom DataBindingSourceCreator capable of parsing key value pairs out of
 * a request body containing a comma separated list of key:value pairs like:
 *
 * name:Herman,age:99,town:STL
 *
 */
class MyCustomDataBindingSourceCreator extends AbstractRequestBodyDataBindingSour

@Override
    public MimeType[] getMimeTypes() {
        [new MimeType('text/custom+demo+csv')] as MimeType[]
    }

@Override
    protected DataBindingSource createBindingSource(InputStream inputStream) {
        def map = [:]

def reader = new InputStreamReader(inputStream)

// this is an obviously naive parser and is intended
        // for demonstration purposes only.

reader.eachLine { line ->
            def keyValuePairs = line.split(',')
            keyValuePairs.each { keyValuePair ->
                if(keyValuePair?.trim()) {
                    def keyValuePieces = keyValuePair.split(':')
                    def key = keyValuePieces[0].trim()
                    def value = keyValuePieces[1].trim()
                    map[key] = value
                }
            }
        }
// create and return a DataBindingSource which contains the parsed data
        new SimpleMapDataBindingSource(map)
    }
}
```

An instance of `MyCustomDataSourceCreator` needs to be registered in the spring application contex

```
// grails-app/conf/spring/resources.groovy
beans = {

myCustomCreator com.demo.myapp.databinding.MyCustomDataBindingSourceCreator

// …
}
```

423

With that in place the framework will use the `myCustomCreator` bean any time a `DataBindingSou` to deal with a request which has a `contentType` of "text/custom+demo+csv".

9.2 SOAP

Grails does not feature SOAP support out-of-the-box, but there are several plugins that can help for both and calling SOAP web services.

SOAP Clients

To call SOAP web services there are generally 2 approaches taken, one is to use a tool to generate cli manually construct the SOAP calls. The former can be easier to use, but the latter provides more flexibility

The [CXF client plugin](#) uses the CXF framework, which includes a `wsdl2java` tool for generating a Groovy/Grails specific here in the generated code as it simply provides a Java API which you can in services.

See the documentation on the [CXF client plugin](#) for further information.

Alternatively, if you prefer more control over your SOAP calls the [WS-Lite library](#) is an excellent cho [plugin](#). You have more control over the SOAP requests sent, and since Groovy has fantastic support for b it can be very productive approach.

Below is an example of a SOAP call with wslite:

```
withSoap(serviceURL: 'http://www.holidaywebservice.com/Holidays/US/Dates/USHolida
    def response = send {
        body {
            GetMothersDay(xmlns: 'http://www.27seconds.com/Holidays/US/Dates/') {
                year(2011)
            }
        }
    }
    println response.GetMothersDayResponse.GetMothersDayResult.text()
}
```

It is not recommended that you use the [GroovyWS](#) library, it pulls in many dependencies which inconflicts. The WSlite library provides a far simpler and easier to use solution.

SOAP Servers

Again, Grails does not have direct support for exposing SOAP web services, however if you wish to expo your application then the [CXF plugin](#) (not to be confused with the cxf-client plugin), provides an easy way

Typically it involves taking a Grails service and adding 'expose'-style configuration, such as the below:

```
    static expose = EndpointType.JAX_WS_WSDL
      //your path (preferred) or url to wsdl
      static wsdl = 'org/grails/cxf/test/soap/CustomerService.wsdl'
```

Please refer to the <u>documentation of the plugin</u> for more information.

9.3 RSS and Atom

No direct support is provided for RSS or Atom within Grails. You could construct RSS or ATOM feeds
XML capability. There is however a <u>Feeds plugin</u> available for Grails that provides a RSS and Atom b
<u>ROME</u> 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
            }
        }
    }
}
```

425

10 Asynchronous Programming

With modern hardware featuring multiple cores, many programming languages have been adding
programming APIs, Groovy being no exception.

The excellent [GPars](#) project features a whole range of different APIs for asynchronous programming tec
promises, STM and data flow concurrency.

Added Grails 2.3, the Async features of Grails aim to simplify concurrent programming within the fra
concept of Promises and a unified event model.

10.1 Promises

A Promise is a concept being embraced by many concurrency frameworks. The
java.util.concurrent.Future instances, but include a more user friendly exception handling m
chaining and the ability to attach listeners.

Promise Basics

In Grails the grails.async.Promises class provides the entry point to the Promise API:

```
import static grails.async.Promises.*
```

To create promises you can use the task method, which returns an instance of the grails.async.Pro

```
def p1 = task { 2 * 2 }
def p2 = task { 4 * 4 }
def p3 = task { 8 * 8 }
assert [4,16,64] == waitAll(p1, p2, p3)
```

The waitAll method waits synchronously, blocking the current thread, for all of the concurrent tasks to
results.

If you prefer not to block the current thread you can use the onComplete method:

```
onComplete([p1,p2,p3]) { List results ->
    assert [4,16,64] == results
}
```

The waitAll method will throw an exception if an error occurs executing one of the promises. The orig
thrown. The onComplete method, however, will simply not execute the passed closure if an exception
an onError listener if you wish to handle exceptions without blocking:

```
onError([p1,p2,p3]) { Throwable t ->
    println "An error occured ${t.message}"
}
```

If you have just a single long running promise then the grails.async.Promise interface provic
promise itself. For example:

```
import static java.util.concurrent.TimeUnit.*
import static grails.async.Promises.*

Promise p = task {
    // Long running task
}
p.onError { Throwable err ->
    println "An error occured ${err.message}"
}
p.onComplete { result ->
    println "Promise returned $result"
}
// block until result is called
def result = p.get()
// block for the specified time
def result = p.get(1,MINUTES)
```

Promise Chaining

It is possible to chain several promises and wait for the chain to complete using the then method:

```
final polish = { … }
final transform = { … }
final save = { … }
final notify = { … }

Promise promise = task {
    // long running task
}
promise.then polish then transform then save then {
    // notify end result
}
```

If an exception occurs at any point in the chain it will be propagated back to the caller and the next step called.

Promise Lists and Maps

Grails' async API also features the concept of a promise lists and maps. These are grails.async.PromiseList and grails.async.PromiseMap classes respectively.

The easiest way to create a promise list or map is via the tasks method of the Promises class:

```
import static grails.async.Promises.*
def promiseList = tasks([{ 2 * 2 }, { 4 * 4}, { 8 * 8 }])

assert [4,16,64] == promiseList.get()
```

The tasks method, when passed a list of closures, returns a PromiseList. You can also construct a Pr

```
import grails.async.*

def list = new PromiseList()
list << { 2 * 2 }
list << { 4 * 4 }
list << { 8 * 8 }
list.onComplete { List results ->
  assert [4,16,64] == results
}
```

428

> ⚠ The `PromiseList` class does not implement the java.util.List interface, but instead
> java.util.List from the get() method

Working with `PromiseMap` instances is largely similar. Again you can either use the `tasks` method:

```
import static grails.async.Promises.*

def promiseList = tasks one:{ 2 * 2 },
                        two:{ 4 * 4},
                        three:{ 8 * 8 }

assert [one:4,two:16,three:64] == promiseList.get()
```

Or construct a `PromiseMap` manually:

```
import grails.async.*

def map = new PromiseMap()
map['one'] = { 2 * 2 }
map['two'] = { 4 * 4 }
map['three'] = { 8 * 8 }
map.onComplete { Map results ->
  assert [one:4,two:16,three:64] == results
}
```

Promise Factories

The `Promises` class uses a `grails.async.PromiseFactory` instance to create `Promise` instance

The default implementation uses the GPars concurrency library a
`org.grails.async.factory.gpars.GparsPromiseFactory`, however it is possible to sy
setting the `Promises.promiseFactory` variable.

One common use case for this is unit testing, typically you do not want promises to execute asynchron
this makes tests harder to write. For this purpose Grails s
`org.grails.async.factory.SynchronousPromiseFactory` instance that makes it easier to

```
import org.grails.async.factory.*
import grails.async.*

Promises.promiseFactory = new SynchronousPromiseFactory()
```

Using the `PromiseFactory` mechanism is theoretically possible to plug in other concurrency framework.

DelegateAsync Transformation

It is quite common to require both synchronous and asynchronous versions of the same API. Develop maintenance problem as typically the asynchronous API would simply delegate to the synchronous version

The `DelegateAsync` transformation is designed to mitigate this problem by transforming any sy asynchronous one.

For example, consider the following service:

```
class BookService {
    List<Book> findBooks(String title) {
        // implementation
    }
}
```

The `findBooks` method executes synchronously in the same thread as the caller. To make an asynchro you can define another class as follows:

```
import grails.async.*

class AsyncBookService {
    @DelegateAsync BookService bookService
}
```

The `DelegateAsync` transformation will automatically add a new method that looks like `AsyncBookService` class:

```
Promise<List<Book>> findBooks(String title) {
    Promises.task {
        bookService.findBooks(title)
    }
}
```

As you see the transform adds equivalent methods that return a Promise and execute asynchronously.

The `AsyncBookService` can then be injected into other controllers and services and used as follows:

```
AsyncBookService asyncBookService
def findBooks(String title) {
    asyncBookService.findBooks(title)
        .onComplete { List results ->
            println "Books = ${results}"
        }
}
```

10.2 Asynchronous GORM

Since Grails 2.3, GORM features an asynchronous programming model that works across all supporte
MongoDB etc.).

Async Namespace

The Asynchronous GORM API is available on every domain class via the `async` namespace.

For example, the following code listing reads 3 objects from the database asynchronously:

```
import static grails.async.Promises.*

def p1 = Person.async.get(1L)
def p2 = Person.async.get(2L)
def p3 = Person.async.get(3L)
def results = waitAll(p1, p2, p3)
```

Using the `async` namespace, all the regular GORM methods are available (even dynamic finders),
synchronously, the query is run in the background and a `Promise` instance is returned.

The following code listing shows a few common examples of GORM queries executed asynchronously:

431

```
import static grails.async.Promises.*

Person.async.list().onComplete { List results ->
  println "Got people = ${results}"
}
def p = Person.async.getAll(1L, 2L, 3L)
List results = p.get()

def p1 = Person.async.findByFirstName("Homer")
def p2 = Person.async.findByFirstName("Bart")
def p3 = Person.async.findByFirstName("Barney")
results = waitAll(p1, p2, p3)
```

Async and the Session

When using GORM async each promise is executed in a different thread. Since the Hibernate session is
new session is bound per thread.

This is an important consideration when using GORM async (particularly with Hibernate as the persiste
returned from asynchronous queries will be detached entities.

This means you cannot save objects returned from asynchronous queries without first merging them back i
the following will not work:

```
def promise = Person.async.findByFirstName("Homer")
def person = promise.get()
person.firstName = "Bart"
person.save()
```

Instead you need to merge the object with the session bound to the calling thread. The above code needs to

```
def promise = Person.async.findByFirstName("Homer")
def person = promise.get()
person.merge()
person.firstName = "Bart"
```

Note that merge() is called first because it may refresh the object from the cache or database, which w
being lost. In general it is not recommended to read and write objects in different threads and you sho
unless absolutely necessary.

Finally, another issue with detached objects is that association lazy loading **will not** work and LazyInitializationException errors if you do so. If you plan to access the associated object asynchronous queries you should use eager queries (which is recommended anyway to avoid N+1 problem

Multiple Asynchronous GORM calls

As discussed in the previous section you should avoid reading and writing objects in different threads inefficient.

However, if you wish to do more complex GORM work asynchronously then the GORM async name method that makes this possible. For example:

```
def promise = Person.async.task {
    withTransaction {
        def person = findByFirstName("Homer")
        person.firstName = "Bart"
        person.save(flush:true)
    }
}
Person updatedPerson = promise.get()
```

Note that the GORM task method differs from the static Promises.task method in that it deals with the asynchronous thread for you. If you do not use the GORM version and do asynchronous work with GO this manually. Example:

```
import static grails.async.Promises.*

def promise = task {
    Person.withNewSession {
        // your logic here
    }
}
```

Async DetachedCriteria

The DetachedCriteria class also supports the async namespace. For example you can do the follow

```
DetachedCriteria query = Person.where {
    lastName == "Simpson"
}
def promise = query.async.list()
```

10.3 Asynchronous Request Handling

If you are deploying to a Servlet 3.0 container such as Tomcat 7 and above then it is possible
asynchronously.

In general for controller actions that execute quickly there is little benefit in handling requests asynchron
running controller actions it is extremely beneficial.

The reason being that with an asynchronous / non-blocking response, the one thread == one request == on
broken. The container can keep a client response open and active, and at the same time return the thread
deal with another request, improving scalability.

For example, if you have 70 available container threads and an action takes a minute to complete, if the a
a non-blocking fashion the likelihood of all 70 threads being occupied and the container not being able to
you should consider asynchronous request processing.

Since Grails 2.3, Grails features a simplified API for creating asynchronous responses built on the `Promi`
previously.

The implementation is based on Servlet 3.0 async so to enable the async features you need to set your se
in BuildConfig.groovy:

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

Async Models

A typical activity in a Grails controller is to produce a model (a map of key/value pairs) that can be rendere

If the model takes a while to produce then the server could arrive at a blocking state, impacting scalability
the model asynchronously by returning a `grails.async.PromiseMap` via the `Promises.tasks`

```
import static grails.async.Promises.*
…
def index() {
    tasks books: Book.async.list(),
          totalBooks: Book.async.count(),
          otherValue: {
            // do hard work
          }
}
```

Grails will handle the response asynchronously, waiting for the promises to complete before rendering synchronous action of the above is:

```
def index() {
    def otherValue = …
    [ books: Book.list() ,
      totalBooks: Book.count(),
      otherValue: otherValue  ]
}
```

You can even render different view by passing the PromiseMap to the model attribute of the render r

```
import static grails.async.Promises.*
…
def index() {
    render view:"myView", model: tasks( one:{ 2 * 2 },
                                        two:{ 3 * 3 } )
}
```

Async Response Rendering

You can also write to the response asynchronously using promises in Grails 2.3 and above:

435
```

```
import static grails.async.Promises.*
class StockController {

def stock(String ticker) {
      task {
          ticker = ticker ?: 'GOOG'
          def url = new URL(
"http://download.finance.yahoo.com/d/quotes.csv?s=${ticker}&f=nsl1op&e=.csv")
          Double price = url.text.split(',')[-1] as Double
          render "ticker: $ticker, price: $price"
      }
    }
}
```

The above example using Yahoo Finance to query stock prices, executing asynchronously and only render
result has been obtained. This is done by returning a `Promise` instance from the controller action.

If the Yahoo URL is unresponsive the original request thread will not be blocked and the container will no

10.4 Servlet 3.0 Async

In addition to the higher level async features discussed earlier in the section, you can access the raw Serv
from a Grails application.

Servlet 3.0 Asynchronous Rendering

You can render content (templates, binary data etc.) in an asynchronous manner by calling the `star`
returns an instance of the Servlet 3.0 `AsyncContext`. Once you have a reference to the `AsyncCont`
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 the `di`
`AsyncContext` class:

```
def index() {
    def ctx = startAsync()
    ctx.start {
        // do working
        …
        // render view
        ctx.dispatch()
    }
}
```

11 Validation

Grails validation capability is built on [Spring's Validator API](#) and data binding capabilities. However Gra provides a unified way to define validation "constraints" with its constraints mechanism.

Constraints in Grails are a way to declaratively specify validation rules. Most commonly they are ap however [URL Mappings](#) and [Command Objects](#) also support constraints.

11.1 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 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 can unique. We've also applied other constraints to the `password`, `email` and `age` properties.

> ⚠ By default, all domain class properties are not nullable (i.e. they have an implicit `nullable` constraint).

A complete reference for the available constraints can be found in the Quick Reference section under the C

Note that constraints are only evaluated once which may be relevant for a constraint that relies on a v
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
you do so, you will get a MissingPropertyException for your trouble. For 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, use a

```
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 validat
survey property and return a boolean to indicate whether the new value for the answer property, val,

11.2 Validating Constraints

Validation Basics

Call the <u>validate</u> 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 <u>Errors</u> interface. The `Errors` inter
navigate the validation errors and also retrieve the original values.

Validation Phases

Within Grails there are two phases of validation, the first one being <u>data binding</u> which occurs when you
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 con
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 <u>validate</u> or <u>save</u>. This is when Grails will validate
the <u>constraints</u> you defined. For example, by default the <u>save</u> method calls `validate` before executir
code like:

```
    if (user.save()) {
        return user
    }
    else {
        user.errors.allErrors.each {
            println it
        }
    }
```

11.3 Sharing Constraints Between Classes

A common pattern in Grails is to use [command objects](#) for validating user-submitted data and then co
command object to the relevant domain classes. This often means that your command objects and domai
and their constraints. You could manually copy and paste the constraints between the two, but that's a ve
Instead, make use of Grails' global constraints and import mechanism.

Global Constraints

In addition to defining constraints in domain classes, command objects and [other validateable classes](#), you
grails-app/conf/Config.groovy:

```
grails.gorm.default.constraints = {
    '*'(nullable: true, size: 1..20)
    myShared(nullable: false, blank: false)
}
```

These constraints are not attached to any particular classes, but they can be easily referenced from any vali

```
class User {
    ...
static constraints = {
        login shared: "myShared"
    }
}
```

Note the use of the shared argument, whose value is the name of one of the co
grails.gorm.default.constraints. Despite the name of the configuration setting, you cai
constraints from any validateable class, such as command objects.

441

The '*' constraint is a special case: it means that the associated constraints ('nullable' and 'size' in the
applied to all properties in all validateable classes. These defaults can be overridden by the constraints c
class.

Importing Constraints

Grails 2 introduced an alternative approach to sharing constraints that allows you to import a set of const
another.

Let's say you have a domain class like so:

```
class User {
    String firstName
    String lastName
    String passwordHash

static constraints = {
        firstName blank: false, nullable: false
        lastName blank: false, nullable: false
        passwordHash blank: false, nullable: false
    }
}
```

You then want to create a command object, UserCommand, that shares some of the properties of t
corresponding constraints. You do this with the importFrom() method:

```
class UserCommand {
    String firstName
    String lastName
    String password
    String confirmPassword

static constraints = {
        importFrom User

password blank: false, nullable: false
        confirmPassword blank: false, nullable: false
    }
}
```

This will import all the constraints from the User domain class and apply them to UserCommand. Th
constraints in the source class (User) that don't have corresponding properties in the importing class (
above example, only the 'firstName' and 'lastName' constraints will be imported into UserCommand be
properties shared by the two classes.

If you want more control over which constraints are imported, use the include and exclude argumen
list of simple or regular expression strings that are matched against the property names in the source const
you only wanted to import the 'lastName' constraint you would use:

```
…
static constraints = {
    importFrom User, include: ["lastName"]

    …
}
```

or if you wanted all constraints that ended with 'Name':

```
…
static constraints = {
    importFrom User, include: [/.*Name/]

    …
}
```

Of course, `exclude` does the reverse, specifying which constraints should *not* be imported.

11.4 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 errors. Grails supports a rich set of tags for dealing with errors. To render the errors as a list you can use re

```
<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>
```

443

Highlighting Errors

It is often useful to highlight using a red box or some indicator when a field has been incorrectly input. T[
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 c[
you to use CSS rules to highlight the `div`.

Retrieving Input Values

Each error is actually an instance of the [FieldError]() class in Spring, which retains the original input value [
you can use the error object to restore the value input by the user using the [fieldValue]() tag:

```
<input type="text" name="login" value="${fieldValue(bean:user,field:'login')}"/>
```

This code will check for an existing `FieldError` in the `User` bean and if there is obtain the originally i[
field.

11.5 Validation and Internationalization

Another important thing to note about errors in Grails is that error messages are not hard coded anywher[
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 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 prec
com.mycompany.myapp.User.login.blank will be used before user.login.blank. This allows for cases v
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 more
can handle this yourself:

445

```
<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 <u>eachError</u> tag we use the <u>message</u> tag in combination with its err
message for the given error.

11.6 Applying Validation to Other Classes

<u>Domain classes</u> and <u>command objects</u> support validation by default. Other classes may be made validatea
constraints property in the class (as described above) and then telling the framework about them
application register the validateable classes with the framework. Simply defining the constraints prop

The Validateable Annotation

Classes which define the static constraints property and are annotated with @Validateable can be :
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 b
The steps required to do this are to define the static constraints property
described above) and then telling the framework about the class by ass:
the grails.validateable.classes property in Config.groovy@:

```
grails.validateable.classes = [com.mycompany.myapp.User, com.mycompany.dto.Accoun
```

12 The Service Layer

Grails defines the notion of a service layer. The Grails team discourages the embedding of core applicatio
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 controllers
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 in a t

```
grails create-service helloworld.simple
```

⚠ If no package is specified with the create-service script, Grails automatically uses the applic
as the package name.

The above example will create a service at the location `grails-app/services/helloworld/Sim`
. A service's name ends with the convention `Service`, other than that a service is a plain Groovy class:

```
package helloworld
class SimpleService {
}
```

12.1 Declarative Transactions

Default Declarative Transactions

Services are typically involved with coordinating logic between [domain classes](#), and hence often invol
spans large operations. Given the nature of services, they frequently require transactional behaviour. Yo
transactions with the [withTransaction](#) method, however this is repetitive and doesn't fully leverage the pow
transaction abstraction.

Services enable transaction demarcation, which is a declarative way of defining which methods are to b
services are transactional by default. To disable this set the `transactional` property to `false`:

```
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 wi
> 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 thr (i.e. one that extends `RuntimeException`) or an `Error`. The propagation level of the transacti [PROPAGATION_REQUIRED](#).

> ⛔ Checked exceptions do **not** roll back transactions. Even though Groovy blurs the distincti
> checked and unchecked exceptions, Spring isn't aware of this and its default behaviour is u
> important to understand the distinction between checked and unchecked exceptions.

Custom Transaction Configuration

Grails also provides `@Transactional` and `@NotTransactional` annotations for cases where you control over transactions at a per-method level or need to specify an alternative propagation le `@NotTransactional` annotation can be used to mark a particular method to be skipped when a `@Transactional`.

> ⚠ The `grails.transaction.Transactional` annotation was first introduced in Grail
> to 2.3, Spring's @Transactional annotation was used.

> ⚠ Annotating a service method with `Transactional` disables the default Grails transaction
> for that service (in the same way that adding `transactional=false` does) so if y
> annotations you must annotate all methods that require transactions.

In this example `listBooks` uses a read-only transaction, `updateBook` uses a default read-write transa 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 t
per-method. For example, this service is equivalent to one that has no annotations (since the
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
overrides this to use a read-only transaction:

```groovy
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
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 annota
will detect them up automatically.

12.1.1 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
persistence session is handled by Hibernate. When a transaction is rolled back the Hibernate session us
This means any objects within the session become detached and accessing uninitialized lazy-loaded
`LazyInitializationExceptions`.

To understand why it is important that the Hibernate session is cleared. Consider the following example:

```groovy
class Author {
    String name
    Integer age

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

If you were to save two authors using consecutive transactions as follows:

451

```
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 cleari
If the Hibernate session were not cleared then both author instances would be persisted and it would
results.

It can, however, be frustrating to get `LazyInitializationExceptions` due to the session being cle

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 maximu `isTooOld()` method by throwing an `AuthorException`. The `AuthorException` references t books association is accessed a `LazyInitializationException` will be thrown because the und has been cleared.

To solve this problem you have a number of options. One is to ensure you query eagerly to get the data you

```
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 is to `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 this ser

```
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 with a rendering:

454

```
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]
      }
    }
}
```

12.2 Scoped Services

By default, access to service methods is not synchronised, so nothing prevents concurrent execution o
because the service is a singleton and may be used concurrently, you should be very careful about storing
the easy (and better) road and never store state in a service.

You can change this behaviour by placing a service in a particular scope. The supported scopes are:

- `prototype` - A new service is created every time it is injected into another class

- `request` - A new service will be created per request

- `flash` - A new service will be created for the current and next request only

- `flow` - In web flows the service will exist for the scope of the flow

- `conversation` - In web flows the service will exist for the scope of the conversation. ie a root flow

- `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 in
> `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, for

```
static scope = "flow"
```

455

12.3 Dependency Injection and Services

Dependency Injection Basics

A key aspect of Grails services is the ability to use [Spring Framework](#)'s dependency injection features. Gr injection by convention". In other words, you can use the property name representation of the cla 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` in

```
class BookController {
    def bookService
    …
}
```

In this case, the Spring container will automatically inject an instance of that service based on its configure injection is done by name. You can also specify the type as follows:

```
class AuthorService {
    BookService bookService
}
```

Dependency Injection and Services

You can inject services in other services with the same technique. If you had an `AuthorService` `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 views:

```
class Book {
    …
    def bookService

def buyBook() {
        bookService.buyBook(this)
    }
}
```

Service Bean Names

The default bean name which is associated with a service can be problematic if there are multiple serv defined in different packages. For example consider the situation where an application defines com.demo.ReportingService and the application uses a plugin named ReportingUtilities a service class named com.reporting.util.ReportingService. The default bean name for reportingService so they would conflict with each other. Grails manages this by changing the defau provided by plugins by prefixing the bean name with the plugin name. In the scenario described above the bean would be an instance of the com.demo.ReportingService class defined in the reportingUtilitiesReportingService bean would be an instal com.reporting.util.ReportingService class provided by the ReportingUtilities plu provided by plugins, if there are no other services with the same name within the application or other plugi a bean alias will be created which does not include the plugin name and that alias points to the bean ref does include the plugin name prefix. For example, if the ReportingUtilities plugin prov com.reporting.util.AuthorService and there is no other AuthorService in the applicatio that the application is using then there will be a bean named reportingUtilitiesAuthorServic this com.reporting.util.AuthorService class and there will be a bean alias defined authorService which points to that same bean.

12.4 Using Services from Java

457

One of the powerful things about services is that since they encapsulate re-usable logic, you can use including Java classes. There are a couple of ways you can reuse a service from Java. The simplest way is a package within the `grails-app/services` directory. The reason this is important is that it is not into Java from the default package (the package used when no package declaration is present). So for exam below cannot be used from Java as it stands:

```
class BookService {
    void buyBook(Book book) {
        // logic
    }
}
```

However, this can be rectified by placing this class in a package, by moving the class into a `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 (although it's always a good idea to use packages). Either way, the goal of this exercise to enable Java to s (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 us of the bean in Spring:

```java
// 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 in `grails-app/conf/spring` more information see the section on [Grails and Spring](#)):

```xml
<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")
    }
}
```

13 Static Type Checking And Compilation

Groovy is a dynamic language and by default Groovy uses a dynamic dispatch mechanism to carry out n access. This dynamic dispatch mechanism provides a lot of flexibility and power to the language. For e dynamically add methods to classes at runtime and it is possible to dynamically replace existing methods these are important and provide a lot of power to the language. However, there are times when you n dynamic dispatch in favor of a more static dispatch mechanism and Groovy provides a way to do that. Th compiler that a particular class should compiled statically is to mark the class with the groovy.transform.C as shown below.

```
import groovy.transform.CompileStatic

@CompileStatic
class MyClass {

// this class will be statically compiled...

}
```

See these notes on Groovy static compilation for more details on how CompileStatic works and why y

One limitation of using CompileStatic is that when you use it you give up access to the power a dynamic dispatch. For example, in Grails you would not be able to invoke a GORM dynamic finder fr with CompileStatic because the compiler cannot verify that the dynamic finder method exists, be compile time. It may be that you want to take advantage of Groovy's static compilation benefits wit dynamic dispatch for Grails specific things like dynamic finders and this is where grails.compiler.Grails GrailsCompileStatic behaves just like CompileStatic but is aware of certain Grails features a specific features to be accessed dynamically.

13.1 The GrailsCompileStatic Annotation

GrailsCompileStatic

The GrailsCompileStatic annotation may be applied to a class or methods within a class.

```
import grails.compiler.GrailsCompileStatic

@GrailsCompileStatic
class SomeClass {

// all of the code in this class will be statically compiled

def methodOne() {
        // …
    }

def methodTwo() {
        // …
    }

def methodThree() {
        // …
    }
}
```

```
import grails.compiler.GrailsCompileStatic

class SomeClass {

// methodOne and methodThree will be statically compiled
    // methodTwo will be dynamically compiled

@GrailsCompileStatic
    def methodOne() {
        // …
    }

def methodTwo() {
        // …
    }

@GrailsCompileStatic
    def methodThree() {
        // …
    }
}
```

It is possible to mark a class with GrailsCompileStatic and exclude specific methods b
GrailsCompileStatic and specifying that the type checking should be skipped for that particular me

```
import grails.compiler.GrailsCompileStatic
import groovy.transform.TypeCheckingMode

@GrailsCompileStatic
class SomeClass {

// methodOne and methodThree will be statically compiled
    // methodTwo will be dynamically compiled

def methodOne() {
        // …
    }

@GrailsCompileStatic(TypeCheckingMode.SKIP)
    def methodTwo() {
        // …
    }

def methodThree() {
        // …
    }
}
```

Code that is marked with GrailsCompileStatic will all be statically compiled except for Grails
cannot be statically compiled but that GrailsCompileStatic can identify as permissible for dynami
things like invoking dynamic finders and DSL code in configuration blocks like constraints and map
classes.

Care must be taken when deciding to statically compile code. There are benefits associated with static co
take advantage of those benefits you are giving up the power and flexibility of dynamic dispatch. For exa
compiled it cannot take advantage of runtime metaprogramming enhancements which may be provided by

13.2 The GrailsTypeChecked Annotation

GrailsTypeChecked

The grails.compiler.GrailsTypeChecked annotation works a lot like the GrailsCompileStatic anno
enables static type checking, not static compilation. This affords compile time feedback for expressions w
statically at compile time while still leaving dynamic dispatch in place for the class.

```
import grails.compiler.GrailsTypeChecked

@GrailsTypeChecked
class SomeClass {

// all of the code in this class will be statically type
    // checked and will be dynamically dispatched at runtime

def methodOne() {
        // …
    }

def methodTwo() {
        // …
    }

def methodThree() {
        // …
    }
}
```

14 Testing

Automated testing is a key part of Grails. Hence, Grails provides many ways to making testing easier from 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 in a JUnit 3 style. Grails 2.0.x and above deprecates these test harnesses in favour of mixins 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 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/SimpleControll` unit test at `test/unit/com/acme/app/SimpleControllerTests.groovy`. What Grails won 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 supp

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
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
faster on the second and subsequent runs. Second, a shortcut is available to open the HTML reports in you

```
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 named Si
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 pattern 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:typ

> ⚠ Grails organises tests by phase and by type. A test phase relates to the state of the Grails
> during the tests, and the type relates to the testing mechanism.
>
> Grails comes with support for 4 test phases (unit, integration, functional and c
> JUnit test types for the unit and integration phases. These test types have the same r
> phase.
>
> Testing plugins may provide new test phases or new test types for existing phases. Refer to
> documentation.

To execute the JUnit integration tests you can run:

```
grails test-app integration:integration
```

Both phase and type are optional. Their absence acts as a wildcard. The following command will run a
phase:

```
grails test-app unit:
```

The Grails [Spock Plugin](#) is one plugin that adds new test types to Grails. It adds a spock test type to the
and functional phases. To run all spock tests in all phases you would run the 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` or a s

14.1 Unit Testing

Unit testing are tests at the "unit" level. In other words you are testing individual methods or blocks of co
for surrounding infrastructure. Unit tests are typically run without the presence of physical resource
databases, socket connections or files. This is to ensure they run as quick as possible since quick feedback

The Test Mixins

Since Grails 2.0, a collection of unit testing mixins is provided by Grails that lets you enhance the beha
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
> backwards compatibility, but is now deprecated. The previous documentation on the sub
> 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 yo
example can't find the classes, here they all are:

- `grails.test.mixin.TestFor`

- `grails.test.mixin.Mock`

- `grails.test.mixin.TestMixin`

- `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.hibernate.HibernateTestMixin`

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

Test Mixin Basics

Most testing can be achieved via the `TestFor` annotation in combination with the `Mock` annotation fo
For example, to test a controller and associated domains you would define the following:

```
@TestFor(BookController)
@Mock([Book, Author, BookService])
```

The `TestFor` annotation defines the class under test and will automatically create a field for the type
example in the above case a "controller" field will be present, however if `TestFor` was defined for a
would be created and so on.

The `Mock` annotation creates mock version of any collaborators. There is an in-memory implementa
simulate most interactions with the GORM API. For those interactions that are not automatically mocked
support for defining mocks and stubs programmatically. For example:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(BookController)
@Mock(Book)
class BookControllerSpec extends Specification {

void "test search"() {
        given:
        def searchMock = mockFor(SearchService)
        searchMock.demand.searchWeb { String q -> ['first result', 'second result
        searchMock.demand.static.logResults { List results ->   }
        controller.searchService = searchMock.createMock()

when:
        controller.search()

then:
        controller.response.text.contains "Found 2 results"
    }
}
```

doWithSpring and doWithConfig callback methods, FreshRuntime annotation

The doWithSpring callback method can be used to add beans with the BeanBuilder DSL. There callback method for changing the grailsApplication.config values before the grailsApplication instance initialized.

```
import grails.test.mixin.support.GrailsUnitTestMixin

import org.junit.ClassRule
import org.junit.rules.TestRule

import spock.lang.Ignore;
import spock.lang.IgnoreRest
import spock.lang.Shared;
import spock.lang.Specification

@TestMixin(GrailsUnitTestMixin)
class StaticCallbacksSpec extends Specification {
    static doWithSpring = {
        myService(MyService)
    }

static doWithConfig(c) {
        c.myConfigValue = 'Hello'
    }

def "grailsApplication is not null"() {
        expect:
        grailsApplication != null
    }

def "doWithSpring callback is executed"() {
        expect:
        grailsApplication.mainContext.getBean('myService') != null
    }

def "doWithConfig callback is executed"(){
        expect:
        config.myConfigValue == 'Hello'
    }
}
```

You can also use these callbacks without "static" together with the grails.test.runtime.Fresh
this case, a clean application context and grails application instance is initialized for each test method call.

```
import grails.test.mixin.support.GrailsUnitTestMixin
import grails.test.runtime.FreshRuntime;

import org.junit.ClassRule
import org.junit.rules.TestRule

import spock.lang.Ignore;
import spock.lang.IgnoreRest
import spock.lang.Shared;
import spock.lang.Specification

@FreshRuntime
@TestMixin(GrailsUnitTestMixin)
class TestInstanceCallbacksSpec extends Specification {
    def doWithSpring = {
        myService(MyService)
    }

def doWithConfig(c) {
        c.myConfigValue = 'Hello'
    }

def "grailsApplication is not null"() {
        expect:
        grailsApplication != null
    }

def "doWithSpring callback is executed"() {
        expect:
        grailsApplication.mainContext.getBean('myService') != null
    }

def "doWithConfig callback is executed"(){
        expect:
        config.myConfigValue == 'Hello'
    }
}
```

You can use `org.codehaus.groovy.grails.commons.InstanceFactoryBean` together wi[t]
FreshRuntime annotation to mock beans in tests.

```
import grails.test.mixin.support.GrailsUnitTestMixin
import grails.test.runtime.FreshRuntime

import org.codehaus.groovy.grails.commons.InstanceFactoryBean
import org.junit.ClassRule

import spock.lang.Shared
import spock.lang.Specification

@FreshRuntime
@TestMixin(GrailsUnitTestMixin)
class MockedBeanSpec extends Specification {
    def myService=Mock(MyService)

def doWithSpring = {
        myService(InstanceFactoryBean, myService, MyService)
    }

def "doWithSpring callback is executed"() {
        when:
        def myServiceBean=grailsApplication.mainContext.getBean('myService')
        myServiceBean.prova()
        then:
        1 * myService.prova() >> { true }
    }
}
```

The DirtiesRuntime annotation

Test methods may be marked with the grails.test.runtime.DirtiesRuntime annotation
modifies the runtime in ways which might be problematic for other tests and as such the runtime should b
method runs.

```
import grails.test.mixin.TestFor
import spock.lang.Specification
import grails.test.runtime.DirtiesRuntime

@TestFor(PersonController)
class PersonControllerSpec extends Specification {

@DirtiesRuntime
    void "a test method which modifies the runtime"() {
        when:
        Person.metaClass.someMethod = { … }
        // ...

then:
        // …
    }
void "a test method which should not be affected by the previous test method"() {
        // …
    }
}
```

Sharing test runtime grailsApplication instance and beans for several test classes

It's possible to share a single grailsApplication instance and beans for several test classes. This feat
`SharedRuntime` annotation. This annotation takes an optional class parameter implements `Shared`
interface. All test classes referencing the same SharedRuntimeConfigurer implementation class will share
a single test run. The value class for SharedRuntimeConfigurer annotation can also implement `TestEv`
this case the instance of the class will be registered as a test event interceptor for the test runtime.

Loading application beans in unit tests

Adding `static loadExternalBeans = true` field definition to a unit test class makes the Grails
bean definitions from `grails-app/conf/spring/resources.gro`
`grails-app/conf/spring/resources.xml` files.

```
import spock.lang.Issue
import spock.lang.Specification
import grails.test.mixin.support.GrailsUnitTestMixin

@TestMixin(GrailsUnitTestMixin)
class LoadExternalBeansSpec extends Specification {
    static loadExternalBeans = true

void "should load external beans"(){
        expect:
        applicationContext.getBean('simpleBean') == 'Hello world!'
    }
}
```

14.1.1 Unit Testing Controllers

The Basics

You use the `grails.test.mixin.TestFor` annotation to unit test controllers. Using `TestFor` in
`grails.test.mixin.web.ControllerUnitTestMixin` and its associated API. For example:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void "test something"() {
    }
}
```

Adding the `TestFor` annotation to a controller causes a new `controller` field to be automatically under test.

> ⚠ The `TestFor` annotation will also automatically annotate any public methods starting with
> JUnit 4's @Test annotation. If any of your test method don't start with "test" just add this man

To test the simplest "Hello World"-style example you can do the following:

```
// Test class
class SimpleController {
    def hello() {
        render "hello"
    }
}
```

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void "test hello"() {
        when:
        controller.hello()

then:
        response.text == 'hello'
    }
}
```

The `response` object is an instance of `GrailsMockHttpServletResponse` (
`org.codehaus.groovy.grails.plugins.testing`) which extends Spring's `MockHttpSe`
and has a number of useful methods for inspecting the state of the response.

For example to test a redirect you can use the `redirectedUrl` property:

```
class SimpleController {
    def index() {
        redirect action: 'hello'
    }
    …
}
```

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test index'() {
        when:
        controller.index()

then:
        response.redirectedUrl == '/simple/hello'
    }
}
```

Many actions make use of the parameter data associated with the request. For example, the 'sort', 'max', a
quite common. Providing these in the test is as simple as adding appropriate values to a special params va

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(PersonController)
class PersonControllerSpec extends Specification {

void 'test list'() {
        when:
        params.sort = 'name'
        params.max = 20
        params.offset = 0
        controller.list()

then:
        // …
    }
}
```

You can even control what type of request the controller action sees by setting the method property of the

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(PersonController)
class PersonControllerSpec extends Specification {

void 'test save'() {
        when:
        request.method = 'POST'
        controller.save()

then:
        // …
    }
}
```

This is particularly important if your actions do different things depending on the type of the request. request as AJAX like so:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(PersonController)
class PersonControllerSpec extends Specification {

void 'test list'() {
        when:
        request.method = 'POST'
        request.makeAjaxRequest()
        controller.getPage()

then:
        // …
    }
}
```

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` prop
`org.springframework.web.servlet.ModelAndView`) or you can use the `view` and `model` p
mixin:

```
class SimpleController {
    def home() {
        render view: "homePage", model: [title: "Hello World"]
    }
    …
}
```

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test home'() {
        when:
        controller.home()

then:
        view == '/simple/homePage'
        model.title == 'Hello World'
    }
}
```

Note that the view string is the absolute view path, so it starts with a '/' and will include path elements, su
after the action's controller.

Testing Template Rendering

Unlike view rendering, template rendering will actually attempt to write the template directly to the respo
`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.`
follows:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test display'() {
        when:
        controller.display()

then:
        response.text == 'contents of the template'
    }
}
```

However, you may not want to render the real template, but just test that is was rendered. In this cas
Groovy Pages:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test display with mock template'() {
        when:
        views['/simple/_snippet.gsp'] = 'mock template contents'
        controller.display()

then:
        response.text == 'mock template contents'
    }
}
```

Testing Actions Which Return A Map

When a controller action returns a `java.util.Map` that `Map` may be inspected directly to assert that
data:

```
class SimpleController {
    def showBookDetails() {
        [title: 'The Nature Of Necessity', author: 'Alvin Plantinga']
    }
}
```

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test show book details'() {
        when:
        def model = controller.showBookDetails()

then:
        model.author == 'Alvin Plantinga'
    }
}
```

Testing XML and JSON Responses

XML and JSON response are also written directly to the response. Grails' mocking capabilities provide 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:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test render xml'() {
        when:
        controller.renderXml()

then:
        response.text == "<book title='Great'/>"
        response.xml.@title.text() == 'Great'
    }
}
```

The xml property is a parsed result from Groovy's XmlSlurper class which is very convenient for parsing

481
```

Testing JSON responses is pretty similar, instead you use the `json` property:

```
// controller action
def renderJson() {
    render(contentType:"application/json") {
        book = "Great"
    }
}
```

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test render json'() {
        when:
        controller.renderJson()

then:
        response.text == '{"book":"Great"}'
        response.json.book == 'Great'
    }
}
```

The `json` property is an instance of `org.codehaus.groovy.grails.web.json.JSONEleme` 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. For incoming JSON or XML requests using Grails' data binding:

```
def consumeBook(Book b) {
    render "The title is ${b.title}."
}
```

To test this Grails provides an easy way to specify an XML or JSON packet via the `xml` or `json` prc above action can be tested by specifying a String containing the XML:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
@Mock([Book])
class SimpleControllerSpec extends Specification {
    void 'test consume book xml'() {
        when:
        request.xml = '<book><title>Wool</title></book>'
        controller.consumeBook()

then:
        response.text == 'The title is Wool.'
    }
}
```

Or alternatively a domain instance can be specified and it will be auto-converted into the appropriate XML

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
@Mock([Book])
class SimpleControllerSpec extends Specification {

void 'test consume book xml'() {
        when:
        request.xml = new Book(title: 'Shift')
        controller.consumeBook()

then:
        response.text == 'The title is Shift.'
    }
}
```

The same can be done for JSON requests:

```groovy
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
@Mock([Book])
class SimpleControllerSpec extends Specification {

void 'test consume book json'() {
        when:
        request.json = new Book(title: 'Shift')
        controller.consumeBook()

then:
        response.text == 'The title is Shift.'
    }
}
```

If you prefer not to use Grails' data binding but instead manually parse the incoming XML or JSON th
example consider the controller action below:

```groovy
def consume() {
    request.withFormat {
        xml {
            render "The XML Title Is ${request.XML.@title}."
        }
        json {
            render "The JSON Title Is ${request.JSON.title}."
        }
    }
}
```

To test the XML request you can specify the XML as a string:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test consume xml'() {
        when:
        request.xml = '<book title="The Stand"/>'
        controller.consume()

then:
        response.text == 'The XML Title Is The Stand.'
    }
void 'test consume json'() {
        when:
        request.json = '{title:"The Stand"}'
        controller.consume()

then:
        response.text == 'The JSON Title Is The Stand.'
    }
}
```

Testing Mime Type Handling

You can test mime type handling and the `withFormat` method quite simply by setting the request's con

```
// controller action
def sayHello() {
    def data = [Hello:"World"]
    request.withFormat {
        xml { render data as grails.converters.XML }
        json { render data as grails.converters.JSON }
        html data
    }
}
```

485

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test say hello xml'() {
        when:
        request.contentType = 'application/xml'
        controller.sayHello()

then:
        response.text == '<?xml version="1.0" encoding="UTF-8"?><map><entry key="
>World</entry></map>'
    }

void 'test say hello json'() {
        when:
        request.contentType = 'application/json'
        controller.sayHello()

then:
        response.text == '{"Hello":"World"}'
    }
}
```

There are constants provided by `ControllerUnitTestMixin` for all of the common common content

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test say hello xml'() {
        when:
        request.contentType = XML_CONTENT_TYPE
        controller.sayHello()

then:
        response.text == '<?xml version="1.0" encoding="UTF-8"?><map><entry key="
>World</entry></map>'
    }

void 'test say hello json'() {
        when:
        request.contentType = JSON_CONTENT_TYPE
        controller.sayHello()

then:
        response.text == '{"Hello":"World"}'
    }
}
```

The defined constants are listed below:

Constant	Value
ALL_CONTENT_TYPE	*/*
FORM_CONTENT_TYPE	application/x-www-form-urlencoded
MULTIPART_FORM_CONTENT_TYPE	multipart/form-data
HTML_CONTENT_TYPE	text/html
XHTML_CONTENT_TYPE	application/xhtml+xml
XML_CONTENT_TYPE	application/xml
JSON_CONTENT_TYPE	application/json
TEXT_XML_CONTENT_TYPE	text/xml
TEXT_JSON_CONTENT_TYPE	text/json
HAL_JSON_CONTENT_TYPE	application/hal+json
HAL_XML_CONTENT_TYPE	application/hal+xml
ATOM_XML_CONTENT_TYPE	application/atom+xml

Testing Duplicate Form Submissions

Testing duplicate form submissions is a little bit more involved. For example if you have an action that har

```
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 or Testing the bad submission is simple. Just invoke the controller:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test duplicate form submission'() {
        when:
        controller.handleForm()

then:
        response.text == 'Bad'
    }
}
```

Testing the successful submission requires providing an appropriate SynchronizerToken:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

import org.codehaus.groovy.grails.web.servlet.mvc.SynchronizerTokensHolder

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test valid form submission'() {
        when:
        def tokenHolder = SynchronizerTokensHolder.store(session)

params[SynchronizerTokensHolder.TOKEN_URI] = '/controller/handleForm'
        params[SynchronizerTokensHolder.TOKEN_KEY] =
tokenHolder.generateToken(params[SynchronizerTokensHolder.TOKEN_URI])
        controller.handleForm()

then:
        response.text == 'Good'
    }
}
```

If you test both the valid and the invalid request in the same test be sure to reset the response between exec

```
import grails.test.mixin.TestFor
import spock.lang.Specification

import org.codehaus.groovy.grails.web.servlet.mvc.SynchronizerTokensHolder

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test form submission'() {
        when:
        controller.handleForm()

then:
        response.text == 'Bad'

when:
        response.reset()
        def tokenHolder = SynchronizerTokensHolder.store(session)

params[SynchronizerTokensHolder.TOKEN_URI] = '/controller/handleForm'
        params[SynchronizerTokensHolder.TOKEN_KEY] =
tokenHolder.generateToken(params[SynchronizerTokensHolder.TOKEN_URI])
        controller.handleForm()

then:
        response.text == 'Good'
    }
}
```

Testing File Upload

You use the `GrailsMockMultipartFile` class to test file uploads. For example consider the followin

```
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:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

import org.codehaus.groovy.grails.plugins.testing.GrailsMockMultipartFile

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test file upload'() {
        when:
        def file = new GrailsMockMultipartFile('myFile', 'some file contents'.byt
        request.addFile file
        controller.uploadFile()

then:
        file.targetFileLocation.path == '/local/disk/myFile'
    }
}
```

The GrailsMockMultipartFile constructor arguments are the name and contents of the file. It has
of the transferTo method that simply records the targetFileLocation and doesn't write to disk.

Testing Command Objects

Special support exists for testing command object handling with the mockCommandObject method. I
following action:

```
class SimpleController {
    def handleCommand(SimpleCommand simple) {
        if(simple.hasErrors()) {
            render 'Bad'
        } else {
            render 'Good'
        }
    }
}

class SimpleCommand {
    String name

static constraints = {
        name blank: false
    }
}
```

To test this you mock the command object, populate it and then validate it as follows:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test valid command object'() {
        given:
        def simpleCommand = new SimpleCommand(name: 'Hugh')
        simpleCommand.validate()

when:
        controller.handleCommand(simpleCommand)

then:
        response.text == 'Good'
    }
void 'test invalid command object'() {
        given:
        def simpleCommand = new SimpleCommand(name: '')
        simpleCommand.validate()

when:
        controller.handleCommand(simpleCommand)

then:
        response.text == 'Bad'
    }
}
```

The testing framework also supports allowing Grails to create the command object instance automaticall
no-arg version of the controller action method. Grails will create an instance of the command object, pe
using the request parameters and validate the object just like it does in when the application is running. See

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test valid command object'() {
        when:
        params.name = 'Hugh'
        controller.handleCommand()

then:
        response.text == 'Good'
    }
void 'test invalid command object'() {
        when:
        params.name = ''
        controller.handleCommand()

then:
        response.text == 'Bad'
    }
}
```

491

Testing allowedMethods

The unit testing environment respects the <u>allowedMethods</u> property in controllers. If a controller action with certain request methods, the unit test must be constructed to deal with that.

```
// grails-app/controllers/com/demo/DemoController.groovypackage com.demo

class DemoController {

static allowedMethods = [save: 'POST', update: 'PUT', delete: 'DELETE']

def save() {
        render 'Save was successful!'
    }
// …
}
```

```
// test/unit/com/demo/DemoControllerSpec.groovy
package com.demo

import grails.test.mixin.TestFor
import spock.lang.Specification
import static javax.servlet.http.HttpServletResponse.*

@TestFor(DemoController)
class DemoControllerSpec extends Specification {

void "test a valid request method"() {
        when:
        request.method = 'POST'
        controller.save()

then:
        response.status == SC_OK
        response.text == 'Save was successful!'
    }

void "test an invalid request method"() {
        when:
        request.method = 'DELETE'
        controller.save()

then:
        response.status == SC_METHOD_NOT_ALLOWED
    }
}
```

Testing Calling Tag Libraries

You can test calling tag libraries using `ControllerUnitTestMixin`, although the mechanism for te[...] from tag to tag. For example to test a call to the `message` tag, add a message to the `messageSource` [...] action:

```
def showMessage() {
    render g.message(code: "foo.bar")
}
```

This can be tested as follows:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
class SimpleControllerSpec extends Specification {

void 'test render message tag'() {
        given:
        messageSource.addMessage 'foo.bar', request.locale, 'Hello World'

when:
        controller.showMessage()

then:
        response.text == 'Hello World'
    }
}
```

See unit testing tag libraries for more information.

14.1.2 Unit Testing Tag Libraries

The Basics

Tag libraries and GSP pages can be tested with the `grails.test.mixin.web.GroovyPageUni[...]` use the mixin declare which tag library is under test with the `TestFor` annotation:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleTagLib)
class SimpleTagLibSpec extends Specification {

void "test something"() {
    }
}
```

Adding the TestFor annotation to a TagLib class causes a new tagLib field to be automatically crea under test. The tagLib field can be used to test calling tags as function calls. The return value of a StreamCharBuffer instance or the object returned from the tag closure when returnObjectForTags feature i

Note that if you are testing invocation of a custom tag from a controller you can combine the Contro and the GroovyPageUnitTestMixin using the Mock annotation:

```
import spock.lang.Specification

@TestFor(SimpleController)
@Mock(SimpleTagLib)
class SimpleControllerSpec extends Specification {

}
```

Testing Custom Tags

The core Grails tags don't need to be enabled during testing, however custom tag GroovyPageUnitTestMixin class provides a mockTagLib() method that you can use to mock example consider the following tag library:

```
class SimpleTagLib {

static namespace = 's'

def hello = { attrs, body ->
        out << "Hello ${attrs.name ?: 'World'}"
    }

def bye = { attrs, body ->
        out << "Bye ${attrs.author.name ?: 'World'}"
    }
}
```

494

You can test this tag library by using `TestFor` and supplying the name of the tag library:

```groovy
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleTagLib)
class SimpleTagLibSpec extends Specification {

void "test hello tag"() {
        expect:
        applyTemplate('<s:hello />') == 'Hello World'
        applyTemplate('<s:hello name="Fred" />') == 'Hello Fred'
        applyTemplate('<s:bye author="${author}" />', [author: new Author(name: '
Fred'
    }

void "test tag calls"() {
        expect:
        tagLib.hello().toString() == 'Hello World'
        tagLib.hello(name: 'Fred').toString() == 'Hello Fred'
        tagLib.bye(author: new Author(name: 'Fred')).toString == 'Bye Fred'
    }
}
```

Alternatively, you can use the `TestMixin` annotation and mock multiple tag libraries using the `mockTa`

```groovy
import spock.lang.Specification
import grails.test.mixin.TestMixin
import grails.test.mixin.web.GroovyPageUnitTestMixin

@TestMixin(GroovyPageUnitTestMixin)
class MultipleTagLibSpec extends Specification {

void "test multiple tags"() {
        given:
        mockTagLib(SomeTagLib)
        mockTagLib(SomeOtherTagLib)

expect:
        // …
    }
}
```

The `GroovyPageUnitTestMixin` provides convenience methods for asserting that the template out
expected value.

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleTagLib)
class SimpleTagLibSpec extends Specification {

void "test hello tag"() {
        expect:
        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` `GroovyPageUnitTestMixin`:

```
import spock.lang.Specification
import grails.test.mixin.TestMixin
import grails.test.mixin.web.GroovyPageUnitTestMixin

@TestMixin(GroovyPageUnitTestMixin)
class RenderingSpec extends Specification {

void "test rendering template"() {
        when:
        def result = render(template: '/simple/hello')

then:
        result == 'Hello World!'
    }
}
```

This will attempt to render a template found at the location `grails-app/views/simple/_hell` template depends on any custom tag libraries you need to call `mockTagLib` as described in the previous s

Some core tags use the active controller and action as input. In GroovyPageUnitTestMixin tests, you ca controller and action name by setting controllerName and actionName properties on the webRequest objec

```
webRequest.controllerName = 'simple'
    webRequest.actionName = 'hello'
```

14.1.3 Unit Testing Domains

Overview

Domain class interaction can be tested without involving a real database connection using `DomainCla` by using the `HibernateTestMixin`.

The GORM implementation in DomainClassUnitTestMixin is using a simple in-memory `ConcurrentH` Note that this has limitations compared to a real GORM implementation.

A large, commonly-used portion of the GORM API can be mocked using `DomainClassUnitTestMix`

- Simple persistence methods like `save()`, `delete()` etc.

- Dynamic Finders

- Named Queries

- Query-by-example

- GORM Events

`HibernateTestMixin` uses Hibernate 4 and a H2 in-memory database. This makes it possible to use a Grails unit tests.

All features of GORM for Hibernate can be tested within a `HibernateTestMixin` unit test including:

- String-based HQL queries

- composite identifiers

- dirty checking methods

- any direct interaction with Hibernate

The implementation behind `HibernateTestMixin` takes care of setting up the Hibernate with the in only configures the given domain classes for use in a unit test. The @Domain annotation is used to te should be configured.

DomainClassUnitTestMixin Basics

`DomainClassUnitTestMixin` is typically used in combination with testing either a controller, servic domain is a mock collaborator defined by the `Mock` annotation:

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(BookController)
@Mock(Book)
class BookControllerSpec extends Specification {
    // …
}
```

The example above tests the `SimpleController` class and mocks the behavior of the `Simple d⟨` 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
import spock.lang.Specification

@TestFor(BookController)
@Mock(Book)
class BookControllerSpec extends Specification {
   void "test saving an invalid book"() {
        when:
        controller.save()

then:
        model.bookInstance != null
        view == '/book/create'
    }

void "test saving a valid book"() {
        when:
        params.title = "The Stand"
        params.pages = "500"

controller.save()

then:
        response.redirectedUrl == '/book/show/1'
        flash.message != null
        Book.count() == 1
    }
}
```

`Mock` annotation also supports a list of mock collaborators if you have more than one domain to mock:

498

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(BookController)
@Mock([Book, Author])
class BookControllerSpec extends Specification {
    // …
}
```

Alternatively you can also use the `DomainClassUnitTestMixin` directly with the `TestMixin` ann mockDomain method to mock domains during your test:

```
import grails.test.mixin.TestFor
import grails.test.mixin.TestMixin
import spock.lang.Specification
import grails.test.mixin.domain.DomainClassUnitTestMixin

@TestFor(BookController)
@TestMixin(DomainClassUnitTestMixin)
class BookControllerSpec extends Specification {

void setupSpec() {
        mockDomain(Book)
    }

void "test saving an invalid book"() {
        when:
        controller.save()

then:
        model.bookInstance != null
        view == '/book/create'
    }

void "test saving a valid book"() {
        when:
        params.title = "The Stand"
        params.pages = "500"

controller.save()

then:
        response.redirectedUrl == '/book/show/1'
        flash.message != null
        Book.count() == 1
    }
}
```

The `mockDomain` method also includes an additional parameter that lets you pass a Map of Maps to con useful for fixture-like data:

```
mockDomain(Book, [
            [title: "The Stand", pages: 1000],
            [title: "The Shining", pages: 400],
            [title: "Along Came a Spider", pages: 300] ])
```

Testing Constraints

There are 4 types of validateable classes:

1. Domain classes

2. Classes marked with the `Validateable` annotation

3. Command Objects which have been made validateable automatically

4. Classes configured to be validateable via the `grails.validateable.classes` property in Con

The first 3 are easily testable in a unit test with no special configuration necessary as long as the test
`TestFor` or explicitly applies the `GrailsUnitTestMixin` using `TestMixin`. See the examples belo

```
// src/groovy/com/demo/MyValidateable.groovy
package com.demo

@grails.validation.Validateable
class MyValidateable {
    String name
    Integer age

static constraints = {
        name matches: /[A-Z].*/
        age range: 1..99
    }
}
```

```
// grails-app/domain/com/demo/Person.groovy
package com.demo

class Person {
    String name

static constraints = {
        name matches: /[A-Z].*/
    }
}
```

```groovy
// grails-app/controllers/com/demo/DemoController.groovy
package com.demo

class DemoController {

def addItems(MyCommandObject co) {
        if(co.hasErrors()) {
            render 'something went wrong'
        } else {
            render 'items have been added'
        }
    }
}

class MyCommandObject {
    Integer numberOfItems

static constraints = {
        numberOfItems range: 1..10
    }
}
```

```groovy
// test/unit/com/demo/PersonSpec.groovy
package com.demo

import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(Person)
class PersonSpec extends Specification {

void "Test that name must begin with an upper case letter"() {
        when: 'the name begins with a lower letter'
        def p = new Person(name: 'jeff')

then: 'validation should fail'
        !p.validate()

when: 'the name begins with an upper case letter'
        p = new Person(name: 'Jeff')

then: 'validation should pass'
        p.validate()
    }
}
```

501

```groovy
// test/unit/com/demo/DemoControllerSpec.groovy
package com.demo

import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(DemoController)
class DemoControllerSpec extends Specification {

void 'Test an invalid number of items'() {
        when:
        params.numberOfItems = 42
        controller.addItems()

then:
        response.text == 'something went wrong'
    }
void 'Test a valid number of items'() {
        when:
        params.numberOfItems = 8
        controller.addItems()

then:
        response.text == 'items have been added'
    }
}
```

```groovy
// test/unit/com/demo/MyValidateableSpec.groovy
package com.demo

import grails.test.mixin.TestMixin
import grails.test.mixin.support.GrailsUnitTestMixin
import spock.lang.Specification


@TestMixin(GrailsUnitTestMixin)
class MyValidateableSpec extends Specification {

void 'Test validate can be invoked in a unit test with no special configuration'(
        when: 'an object is valid'
        def validateable = new MyValidateable(name: 'Kirk', age: 47)

then: 'validate() returns true and there are no errors'
        validateable.validate()
        !validateable.hasErrors()
        validateable.errors.errorCount == 0

when: 'an object is invalid'
        validateable.name = 'kirk'

then: 'validate() returns false and the appropriate error is created'
        !validateable.validate()
        validateable.hasErrors()
        validateable.errors.errorCount == 1
        validateable.errors['name'].code == 'matches.invalid'

when: 'the clearErrors() is called'
        validateable.clearErrors()

then: 'the errors are gone'
        !validateable.hasErrors()
        validateable.errors.errorCount == 0

when: 'the object is put back in a valid state'
        validateable.name = 'Kirk'

then: 'validate() returns true and there are no errors'
        validateable.validate()
        !validateable.hasErrors()
        validateable.errors.errorCount == 0
    }
}
```

```groovy
// test/unit/com/demo/MyCommandObjectSpec.groovy
package com.demo

import grails.test.mixin.TestMixin
import grails.test.mixin.support.GrailsUnitTestMixin
import spock.lang.Specification

@TestMixin(GrailsUnitTestMixin)
class MyCommandObjectSpec extends Specification {

void 'Test that numberOfItems must be between 1 and 10'() {
        when: 'numberOfItems is less than 1'
        def co = new MyCommandObject()
        co.numberOfItems = 0

then: 'validation fails'
        !co.validate()
        co.hasErrors()
        co.errors['numberOfItems'].code == 'range.toosmall'

when: 'numberOfItems is greater than 10'
        co.numberOfItems = 11

then: 'validation fails'
        !co.validate()
        co.hasErrors()
        co.errors['numberOfItems'].code == 'range.toobig'

when: 'numberOfItems is greater than 1'
        co.numberOfItems = 1

then: 'validation succeeds'
        co.validate()
        !co.hasErrors()

when: 'numberOfItems is greater than 10'
        co.numberOfItems = 10

then: 'validation succeeds'
        co.validate()
        !co.hasErrors()
    }
}
```

For validateable classes which are not one of the first 3 types listed above but are grails.validateable.classes property in Config.groovy, one additional step is requ GrailsUnitTestMixin provides a method named mockForConstraintsTests that will mod these classes. See the example below.

```groovy
// src/groovy/com/demo/Book.groovy
package com.demo

class Book {
    String title
    String author

    static constraints = {
        author minSize: 5
    }
}
```

```groovy
// grails-app/conf/Config.groovy
grails.validateable.classes = [com.demo.Book]

// ...
```

```groovy
// test/unit/com/demo/BookSpec.groovy
package com.demo

import grails.test.mixin.TestMixin
import grails.test.mixin.support.GrailsUnitTestMixin
import spock.lang.Specification

@TestMixin(GrailsUnitTestMixin)
class BookSpec extends Specification {

void 'Test validation'() {
        given:
        mockForConstraintsTests Book

when: 'the author name has only 4 characters'
        def book = new Book()
        book.author = 'Jeff'

then: 'validation should fail'
        !book.validate()
        book.hasErrors()
        book.errors['author'] == 'minSize'

when: 'the author name has 5 characters'
        book.author = 'Jacob'

then: 'validation should pass'
        book.validate()
        !book.hasErrors()
    }
}
```

Note that the `mockForConstraintsTests` method changes the behavior of the errors object su

`book.errors'author'` will evaluate to the name of the failed co
`org.springframework.validation.FieldError` object. This is convenient for unit tests. If

want a reference to the `org.springframework.validation.FieldError` object
`book.errors.getFieldError('author')`.

That's it for testing constraints. One final thing we would like to say is that testing the constraints in thi
error: typos in the "constraints" property name which is a mistake that is easy to make and equally easy to
your constraints will highlight the problem straight away.

HibernateTestMixin Basics

`HibernateTestMixin` allows Hibernate 4 to be used in Grails unit tests. It uses a H2 in-memory datab

```groovy
import grails.test.mixin.TestMixin
import grails.test.mixin.gorm.Domain
import grails.test.mixin.hibernate.HibernateTestMixin
import spock.lang.Specification


@Domain(Person)
@TestMixin(HibernateTestMixin)
class PersonSpec extends Specification {

void "Test count people"() {
        expect: "Test execute Hibernate count query"
            Person.count() == 0
            sessionFactory != null
            transactionManager != null
            session != null
    }
}
```

This library dependency is required in grails-app/conf/BuildConfig.groovy for adding support for `Hibern`

```groovy
dependencies {
        test 'org.grails:grails-datastore-test-support:1.0-grails-2.4'
    }
```

HibernateTestMixin is only supported with hibernate4 plugin versions >= 4.3.5.4 .

```
plugins {
        runtime ':hibernate4:4.3.5.4'
    }
```

Configuring domain classes for HibernateTestMixin tests

The `grails.test.mixin.gorm.Domain` annotation is used to configure the list of domain classes t
sessionFactory instance that gets configured when the unit test runtime is initialized.

`Domain` annotations will be collected from several locations:

- the annotations on the test class

- the package annotations in the package-info.java/package-info.groovy file in the package of the test cl

- each super class of the test class and their respective package annotations

- the possible [SharedRuntime](#) class

`Domain` annotations can be shared by adding them as package annotations to package-info.java/packag
adding them to a [SharedRuntime](#) class which has been added for the test.

It's not possible to use DomainClassUnitTestMixin's `Mock` annotation in HibernateTestMixin tests. Use t
the place of `Mock` in HibernateTestMixin tests.

14.1.4 Unit Testing Filters

Unit testing filters is typically a matter of testing a controller where a filter is a mock collaborator. F
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. T
off with a test that targets the `SimpleController` class and add the `CancellingFilters` as a mo

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
@Mock(CancellingFilters)
class SimpleControllerSpec extends Specification {

// ...

}
```

You can then implement a test that uses the `withFilters` method to wrap the call to an action in filter e

```
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(SimpleController)
@Mock(CancellingFilters)
class SimpleControllerSpec extends Specification {

void "test list action is filtered"() {
        when:
        withFilters(action:"list") {
            controller.list()
        }

then:
        response.redirectedUrl == '/book'
    }
}
```

Note that the `action` parameter is required because it is unknown what the action to invoke is until the
The `controller` parameter is optional and taken from the controller under test. If it is another controller
can specify it:

```
withFilters(controller:"book",action:"list") {
    controller.list()
}
```

14.1.5 Unit Testing URL Mappings

The Basics

Testing URL mappings can be done with the `TestFor` annotation testing a particular URL mappings clas default URL mappings you can do the following:

```
import com.demo.SimpleController
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(UrlMappings)
@Mock(SimpleController)
class UrlMappingsSpec extends Specification {
    // …
}
```

As you can see, any controller that is the target of a URL mapping that you're testing *must* be added to the

> ⚠ Note that since the default `UrlMappings` class is in the default package your test must al default package

With that done there are a number of useful methods that are `grails.test.mixin.web.UrlMappingsUnitTestMixin` for testing URL mappings. These inc

- `assertForwardUrlMapping` - Asserts a URL mapping is forwarded for the given 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 and action

- `assertUrlMapping` - Asserts a URL mapping is valid for the given URL. `assertForwardUrlMapping` and `assertReverseUrlMapping` assertions

Asserting Forward URL Mappings

You use `assertForwardUrlMapping` to assert that a given URL maps to a given controller. Fc following URL mappings:

```
static mappings = {
    "/actionOne"(controller: "simple", action: "action1")
    "/actionTwo"(controller: "simple", action: "action2")
}
```

The following test can be written to assert these URL mappings:

```
import com.demo.SimpleController
import grails.test.mixin.TestFor
import spock.lang.Specification

@TestFor(UrlMappings)
@Mock(SimpleController)
class UrlMappingsSpec extends Specification {

void "test forward mappings"() {
        expect:
        assertForwardUrlMapping("/actionOne", controller: 'simple', action: "acti
        assertForwardUrlMapping("/actionTwo", controller: 'simple', action: "acti
    }
}
```

Assert Reverse URL Mappings

You use `assertReverseUrlMapping` to check that correct links are produced for your URL mappi
tag in GSP views. An example test is largely identical to the previous listing except you use `assert`
instead of `assertForwardUrlMapping`. Note that you can combine these 2 assertions with `assert`

14.1.6 Mocking Collaborators

The Spock Framework manual has a chapter on [Interaction Based Testing](#) which also explains mocking co

14.1.7 Mocking Codecs

The `GrailsUnitTestMixin` provides a `mockCodec` method for mocking [custom codecs](#) which ma
test is running.

```
mockCodec(MyCustomCodec)
```

Failing to mock a codec which is invoked while a unit test is running may result in a MissingMethodExcep

14.1.8 Unit Test Metaprogramming

If runtime metaprogramming needs to be done in a unit test it needs to be done early in the proces
environment is fully initialized. This should be done when the unit test class is being initialized. For a Spo
be done in the `setupSpec()` method. For a JUnit test this should be done in a method marked with @Be

```groovy
package myapp

import grails.test.mixin.*
import spock.lang.Specification

@TestFor(SomeController)
class SomeControllerSpec extends Specification {

    def setupSpec() {
        SomeClass.metaClass.someMethod = { ->
            // do something here
        }
    }

// …
}
```

```groovy
package myapp

import grails.test.mixin.*
import org.junit.*

@TestFor(SomeController)
class SomeControllerTests {

@BeforeClass
    static void metaProgramController() {
        SomeClass.metaClass.someMethod = { ->
            // do something here
        }
    }

// ...

}
```

14.2 Integration Testing

Integration tests differ from unit tests in that you have full access to the Grails environment within in-memory H2 database for integration tests and clears out all the data from the database between tests.

One thing to bear in mind is that logging is enabled for your application classes, but it is different from l have something like this:

511

```
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 example above is an instance of java.util.logging.Logger (inherited from the base class, not in doesn't have the same methods as the log property injected into your application artifacts. For example, or trace() methods, and the equivalent of 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 t saved during a test is not persisted to the database. Add a transactional property to your test cla 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 me 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](#), ar you can use in your tests. For example consider the following controller:

512

```
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 obt
with contentAsString (when writing to the response) or the redirected URL. These mocked version
completely mutable (unlike the real versions) and hence you can set properties on the request such as the
on.

Grails **does not** invoke interceptors or servlet filters when calling actions during integration testing. You
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 fr

Given a controller using a service:

513

```
class FilmStarsController {
    def popularityService

def update() {
        // do something with popularityService
    }
}
```

The test for this would be:

```
class FilmStarsTests extends GroovyTestCase {
    def popularityService

void testInjectedServiceInController () {
        def fsc = new FilmStarsController()
        fsc.popularityService = popularityService
        fsc.update()
    }
}
```

Testing Controller Command Objects

With command objects you just supply parameters to the request and it will automatically do the comm
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:
```

514

```
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
request parameters. During controller testing, the `params` are mutable with a mocked request supplied by

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 instan
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 inst
`modelAndView` property of the controller. The `modelAndView` property is an instance of Spring MV
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
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 = "application/json"
    controller.request.content =
            '{"id":1,"class":"Book","title":"The Stand"}'.getBytes()

def model = controller.create()
    assert model.book
    assertEquals "The Stand", model.book.title
}
```

⚠️ With JSON don't forget the `class` property to specify the name the target type to bind to. I
    is implicit within the name of the `<book>` node, but this property is required as part of the JS

For more information on the subject of REST web services see the section on [REST](#).

516

Testing Tag Libraries

Testing tag libraries is simple because when a tag is invoked as a method it returns its result as
`StreamCharBuffer` but this class implements all of the methods of `String`). So for example if you h

```
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 ta
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
`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
the output of GSP rendering.

> ⚠️ `GroovyPagesTestCase` can only be used in an integration test.

517

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 `GroovyPagesTest`

```
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 to b
testing queries you often need to "flush" to ensure the correct state has been persisted to the databas
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 <u>save</u> does not actually persist the `Book` instances when called. Calling
Hibernate that at some point in the future these instances should be persisted. To commit changes immedia

```
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 persiste
will be available to the query later on.

14.3 Functional Testing

Functional tests involve making HTTP requests against the running application and verifying the r
functional testing phase differs from the integration phase in that the Grails application is now listening
HTTP requests. This is useful for end-to-end testing scenarios, such as making REST calls against a JSON

Grails does not ship with any support for writing functional tests directly, but there are several plugins ava

- `Canoo Webtest` - <u>http://grails.org/plugin/webtest</u>

- `G-Func` - <u>http://grails.org/plugin/functional-test</u>

- `Geb` - <u>http://grails.org/plugin/geb</u>

- `Selenium-RC` - <u>http://grails.org/plugin/selenium-rc</u>

- `WebDriver` - <u>http://grails.org/plugin/webdriver</u>

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
application in a production-like state, but it has a longer startup time than the `-inline` option. It also
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 the W

https

The `-https` option results in the application being able to receive https requests as well as http requests.
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 `-httpsBaseUrl`

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 a
base url but still test against the local Grails application you **must** specify one of either the -inline or -

15 Internationalization

Grails supports Internationalization (i18n) out of the box by leveraging the underlying Spring MVC inte
With Grails you are able to customize the text that appears in a view based on the user's Locale. To q
Locale class:

*A Locale object represents a specific geographical, political, or cultural region. An operation that r
perform its task is called locale-sensitive and uses the Locale to tailor information for the u
displaying a number is a locale-sensitive operation--the number should be formatted
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 E
the code for British English.

15.1 Understanding Message Bundles

Now that you have an idea of locales, to use them in Grails you create message bundle file containing th
you wish to render. Message bundles in Grails are located inside the `grails-app/i18n` directory and a
files.

Each bundle starts with the name `messages` by convention and ends with the locale. Grails ships with
for a whole range of languages within the `grails-app/i18n` directory. For 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 loc
own message bundle by simply creating a new properties file that ends with the locale you are i
`messages_en_GB.properties` for British English.

15.2 Changing Locales

By default the user locale is detected from the incoming `Accept-Language` header. However, yo
capability to switch locales by simply passing a parameter called `lang` to Grails as a request parameter:

```
/book/list?lang=es
```

Grails will automatically switch the user's locale and store it in a cookie so subsequent requests will have t

15.3 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 t
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 <u>tag libraries</u>, but if your tag library uses a custom <u>namespace</u> then you g.:

```
def myTag = { attrs, body ->
    def msg = g.message(code: "my.localized.content", args: ['Juan', 'lunes'])
}
```

15.4 Scaffolding and i18n

Grails <u>scaffolding</u> templates for controllers and views are fully i18n-aware. The GSPs use the <u>message</u> ta and controller `flash` messages use i18n to resolve locale-specific messages.

The scaffolding includes locale specific labels for domain classes and domain fields. For example, if y class with a `title` field:

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

The scaffolding will use labels with the following keys:

```
book.label = Libro
book.title.label = Ttulo del libro
```

You can use this property pattern if you'd like or come up with one of your own. There is nothing special a `label` as part of the key other than it's the convention used by the scaffolding.

16 Security

Grails is no more or less secure than Java Servlets. However, Java servlets (and hence Grails) are extr
immune to common buffer overrun and malformed URL exploits due to the nature of the Java Virtual M
code.

Web security problems typically occur due to developer naivety or mistakes, and there is a little Grails
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 SQI

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 escapin
   code injection

4. Grails provides codecs to let you trivially escape data when rendered as HTML, JavaScript and U
   attacks here.

16.1 Securing Against Attacks

SQL injection

Hibernate, which is the technology underlying GORM domain classes, automatically escapes data when c
this is not an issue. However it is still possible to write bad dynamic HQL code that uses unchecked
example doing the following is vulnerable to HQL injection attacks:

```
def vulnerable() {
    def books = Book.find("from Book as b where b.title ='" + params.title + "'")
}
```

or the analogous 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 declared commu 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 not from another site. It is also important to ensure that all data values rendered into views are escaped cor rendering to HTML or XHTML you must ensure that people cannot maliciously inject JavaScript or othe viewed by others.

Grails 2.3 and above include special support for automatically encoded data placed into GSP pages. Se Cross Site Scripting (XSS) prevention for further information.

You must also avoid the use of request parameters or data fields for determining the next URL to redirect successURL parameter for example to determine where to redirect a user to after a successful login, a login procedure using your own site, and then redirect the user back to their own site once logged JavaScript code to then exploit the logged-in account on the site.

Cross-site request forgery

CSRF involves unauthorized commands being transmitted from a user that a website trusts. A typical ex website embedding a link to perform an action on your website if the user is still authenticated.

The best way to decrease risk against these types of attacks is to use the useToken attribute on yo Duplicate Form Submissions for more information on how to use it. An additional measure would be 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 wi 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 sup [encodeAsURL](#) where appropriate. If you create your own tags that generate URLs you will need to be min

Denial of service

Load balancers and other appliances are more likely to be useful here, but there are also issues relating example where a link is created by an attacker to set the maximum value of a result set so that a query c limits of the server or slow the system down. The solution here is to always sanitize request parameter dynamic finders or other GORM query methods:

```
int limit = 100
def safeMax = Math.min(params.max?.toInteger() ?: limit, limit) // limit to 100 r
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 else 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 befc to the user.

Not doing this is "security through obscurity" which is inevitably breached, just like having a default pass on.

You must assume that every unprotected URL is publicly accessible one way or another.

16.2 Cross Site Scripting (XSS) Prevention

Cross Site Scripting (XSS) attacks are a common attack vector for web applications. They typically invo Javascript code in a form such that when that code is displayed, the browser does something nasty. It coul up an alert box, or it could be much worse. The solution is to escape all untrusted user input when it is example,

```
<script>alert('Got ya!');</script>
```

will become

```
&lt;script&gt;alert('Got ya!');&lt;/script&gt;
```

when rendered, nullifying the effects of the malicious input.

By default, Grails plays it safe and escapes all content in ${} expressions in GSPs. All the standard G default, escaping any relevant attribute values.

So what happens when you want to stop Grails from escaping some content? There are valid use cases fc database and rendering it as-is, as long as that content is **trusted**. In such cases, you can tell Grails that the be rendered raw, i.e. without any escaping:

```
<section>${raw(page.content)}</section>
```

The raw() method you see here is available from controllers, tag libraries and GSP pages.

XSS prevention is hard and requires a lot of developer attention

> 🚫 Although Grails plays it safe by default, that is no guarantee that your application will be invu an XSS-style attack. Such an attack is less likely to succeed than would otherwise be th developers should always be conscious of potential attack vectors and attempt to uncover vu in the application during testing. It's also easy to switch to an unsafe default, thereby increas of a vulnerability being introduced.

There are more details about the XSS in OWASP - XSS prevention rules and OWASP - Types of Cross XSS are: Stored XSS, Reflected XSS and DOM based XSS. DOM based XSS prevention is coming more popularity of Javascript client side templating and Single Page Apps.

Grails codecs are mainly for preventing stored and reflected XSS type of attacks. Grails 2.4 includes HTN preventing some DOM based XSS attacks.

It's difficult to make a solution that works for everyone, and so Grails provides a lot of flexibility with r escaping works, allowing you to keep most of your application safe while switching off default escapir used for pages, tags, page fragments, and more.

Configuration

It is recommended that you review the configuration of a newly created Grails application to garner a prevention works in Grails.

GSP features the ability to automatically HTML encode GSP expressions, and as of Grails 2.3 this is the d default configuration (found in Config.groovy) for a newly created Grails application can be seen belc

```
grails {
      views {
          gsp {
              encoding = 'UTF-8'
              htmlcodec = 'xml' // use xml escaping instead of HTML4 escaping
              codecs {
                  expression = 'html' // escapes values inside ${}
                  scriptlet = 'html' // escapes output from scriptlets in GSPs
                  taglib = 'none' // escapes output from taglibs
                  staticparts = 'none' // escapes output from static template p
              }
          }
          // escapes all not-encoded output at final stage of outputting
          // filteringCodecForContentType.'text/html' = 'html'
          }
      }
  }
```

GSP features several codecs that it uses when writing the page to the response. The codecs are configur
and are described below:

- `expression` - The expression codec is used to encode any code found within ${..} expression
  created application is `html` encoding.

- `scriptlet` - Used for output from GSP scriplets (<% %>, <%= %> blocks). The default for newl
  `html` encoding

- `taglib` - Used to encode output from GSP tag libraries. The default is `none` for new applicatio
  responsibility of the tag author to define the encoding of a given tag and by specifying `none` G
  compatible with older tag libraries.

- `staticparts` - Used to encode the raw markup output by a GSP page. The default is `none`.

Double Encoding Prevention

Versions of Grails prior to 2.3, included the ability to set the default codec to `html`, however enablin
proved problematic when using existing plugins due to encoding being applied twice (once by the `html` co
plugin manually called `encodeAsHTML`).

Grails 2.3 includes double encoding prevention so that when an expression is evaluated, it will not enco
been encoded (Example ${foo.encodeAsHTML()}).

Raw Output

If you are 100% sure that the value you wish to present on the page has not been received from user input
value to be encoded then you can use the `raw` method:

```
${raw(book.title)}
```

The 'raw' method is available in tag libraries, controllers and GSP pages.

Per Plugin Encoding

Grails also features the ability to control the codecs used on a per plugin basis. For example if you ha
installed, then placing the following configuration in your application's `Config.groovy` will disable en
plugin

```
foo.grails.views.gsp.codecs.expression = "none"
```

Per Page Encoding

You can also control the various codecs used to render a GSP page on a per page basis, using a page direct

```
<%@page expressionCodec="none" %>
```

Per Tag Library Encoding

Each tag library created has the opportunity to specify a default codec used to encode output from
"defaultEncodeAs" property:

```
static defaultEncodeAs = 'html'
```

Encoding can also be specified on a per tag basis using "encodeAsForTags":

```
static encodeAsForTags = [tagName: 'raw']
```

Context Sensitive Encoding Switching

Certain tags require certain encodings and Grails features the ability to enable a codec only a certain part (
the "withCodec" method. Consider for example the "<g:javascript>"" tag which allows you to embed Jav
This tag requires JavaScript encoding, not HTML coding for the execution of the body of the tag (but n
output):

```
out.println '<script type="text/javascript">'
    withCodec("JavaScript") {
        out << body()
    }
    out.println()
    out.println '</script>'
```

Forced Encoding for Tags

If a tag specifies a default encoding that differs from your requirements you can force the encoding fo
optional 'encodeAs' attribute:

```
<g:message code="foo.bar" encodeAs="JavaScript" />
```

Default Encoding for All Output

The default configuration for new applications is fine for most use cases, and backwards compatible with
libraries. However, you can also make your application even more secure by configuring Grails to alway
end of a response. This is done using the filteringCodecForContentType configuration in Conf

```
grails.views.gsp.filteringCodecForContentType.'text/html' = 'html'
```

Note that, if activated, the `staticparts` codec typically needs to be set to `raw` so that static markup is

```
codecs {
        expression = 'html' // escapes values inside ${}
        scriptlet = 'html' // escapes output from scriptlets in GSPs
        taglib = 'none' // escapes output from taglibs
        staticparts = 'raw' // escapes output from static template parts
    }
```

16.3 Encoding and Decoding Objects

Grails supports the concept of dynamic encode/decode methods. A set of standard codecs are bundled supports a simple mechanism for developers to contribute their own codecs that will be recognized at runti

Codec Classes

A Grails codec class is one that may contain an encode closure, a decode closure or both. When a Grails Grails framework dynamically loads codecs from the `grails-app/utils/` directory.

The framework looks under `grails-app/utils/` for class names that end with the convention `Codec` 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 met with a name representing the codec that defined the encode closure. For example, the `HTMLCodec` c closure, so Grails attaches it with the name `encodeAsHTML`.

The `HTMLCodec` and `URLCodec` classes also define a `decode` closure, so Grails attaches those with t and `decodeURL` respectively. Dynamic codec methods may be invoked from anywhere in a Grails a consider a case where a report contains a property called 'description' which may contain special character be presented in an HTML document. One way to deal with that in a GSP is to encode the description pr encode method as shown below:

```
${report.description.encodeAsHTML()}
```

Decoding is performed using `value.decodeHTML()` syntax.

Encoder and Decoder interfaces for staticly compiled code

A preferred way to use codecs is to use the codecLookup bean to get hold of `Encoder` and `Decoder` ins

```
package org.codehaus.groovy.grails.support.encoding;

public interface CodecLookup {
    public Encoder lookupEncoder(String codecName);
    public Decoder lookupDecoder(String codecName);
}
```

example of using `CodecLookup` and `Encoder` interface

```
import org.codehaus.groovy.grails.support.encoding.CodecLookup

class CustomTagLib {
    CodecLookup codecLookup

def myTag = { Map attrs, body ->
        out << codecLookup.lookupEncoder('HTML').encode(attrs.something)
    }
}
```

Standard Codecs

HTMLCodec

This codec performs HTML escaping and unescaping, so that values can be rendered safely in an HTML [
HTML tags or damaging the page layout. For example, given a value "Don't you know that $2 > 1$?" you
this safely within an HTML page because the $>$ will look like it closes a tag, which is especially bad if yo
an attribute, such as the value attribute of an input field.

Example of usage:

```
<input name="comment.message" value="${comment.message.encodeAsHTML()}"/>
```

> ⚠ Note that the HTML encoding does not re-encode apostrophe/single quote so you must use do
> on attribute values to avoid text with apostrophes affecting your page.

HTMLCodec defaults to HTML4 style escaping (legacy HTMLCodec implementation in Grails versi
escapes non-ascii characters.

You can use plain XML escaping instead of HTML4 escaping by setting this config property in Config.gro

533

```
grails.views.gsp.htmlcodec = 'xml'
```

XMLCodec

This codec performs XML escaping and unescaping. It escapes & , < , > , " , ' , \ , @ , ` , non break
separator (\u2028) and paragraph separator (\u2029).

HTMLJSCodec

This codec performs HTML and JS encoding. It is used for preventing some DOM-XSS vulnerabilities. Se
XSS Prevention Cheat Sheet for guidelines of preventing DOM based XSS attacks.

URLCodec

URL encoding is required when creating URLs in links or form actions, or any time data is used to create
characters from getting into the URL and changing its meaning, for example "Apple & Blackberry" is no
parameter in a GET request as the ampersand will break 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
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
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
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 defaul
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
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 defaul
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 sy
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 cu
defined in the `grails-app/utils/` directory and the class name must end with `Codec`. The codec
encode closure, a `static decode` closure or both. The closure must accept a single argument which
dynamic method was invoked on. For 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()}
```

16.4 Authentication

Grails has no default mechanism for authentication as it is possible to implement authentication in m
however, easy to implement a simple authentication mechanism using either interceptors or filters. This is
cases but it's highly preferable to use an established security framework, for example by using the Spr
plugin.

Filters let you apply authentication across all controllers or across a URI space. For example you can crea
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 session then redirect to the `login` action.

The `login` action itself is simple too:

```
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"])
    }
}
```

16.5 SSL in Development

You can configure the embedded Tomcat server to use SSL in development mode. Typically with Tomcat server.xml file to define these properties. However in Grails you need to hook into Tomcat grails-app/scripts/_Events.groovy. In the example below I am using environment variable the keystore and truststore. Notice also that we must add the keystore/truststore properties twice. Using

```
System.setProperty("javax.net.ssl.keyStore", "$keystorepath")
```

will define the keystore path for Grails. You will also need to also define the keystore/truststore paths fo
Here is an example in `grails-app/scripts/_Events.groovy`:

```groovy
import org.apache.catalina.connector.*
import grails.util.Environment

eventConfigureTomcat = { org.apache.catalina.startup.Tomcat tomcat ->
        if (Environment.getCurrent() == Environment.DEVELOPMENT) {
            String keystorepass = System.getenv("KEY_STORE_PASS")
            String keystorepath = System.getenv("KEY_STORE_PATH")
            String truststorepass = System.getenv("TRUST_STORE_PASS")
            String truststorepath = System.getenv("TRUST_STORE_PATH")
System.setProperty("javax.net.debug", "ssl") //use this to confirm grails adds pr
keystore/truststore settings
            System.setProperty("javax.net.ssl.keyStoreType", "jks")
            System.setProperty("javax.net.ssl.keyStore", "$keystorepath")
            System.setProperty("javax.net.ssl.keyStorePassword", "$keystorepass")
            System.setProperty("javax.net.ssl.trustStoreType", "jks")
            System.setProperty("javax.net.ssl.trustStore", "$truststorepath")
            System.setProperty("javax.net.ssl.trustStorePassword", "$truststorepa
def connector = new org.apache.catalina.connector.Connector()
            connector.port = 8443
            connector.protocol = 8443
            connector.scheme = "https"
            connector.setProperty("maxThreads", "150")
            connector.setProperty("SSLEnabled", "true")
            connector.setProperty("secure", "true")
            connector.setProperty("clientAuth", "true")
            connector.setProperty("sslProtocol", "TLS")
            connector.setProperty("keystoreType", "JKS")
            connector.setProperty("keystoreFile", "$keystorepath")
            connector.setProperty("keystorePass", "$keystorepass")
            connector.setProperty("truststoreType", "JKS")
            connector.setProperty("truststoreFile", "$truststorepath")
            connector.setProperty("truststorePass", "$truststorepass")
tomcat.service.addConnector connector
            println "SSL configuration complete"
        }
    }
```

16.6 Security Plugins

If you need more advanced functionality beyond simple authentication such as authorization, roles etc. t
using one of the available security plugins.

16.6.1 Spring Security

The Spring Security plugins are built on the <u>Spring Security</u> project which provides a flexible, extensibl
all sorts of authentication and authorization schemes. The plugins are modular so you can install just the f
for your application. The Spring Security plugins are the official security plugins for Grails and are
supported.

There is a <u>Core plugin</u> which supports form-based authentication, encrypted/salted passwords, HTTP Basi
secondary dependent plugins provide alternate functionality such as <u>OpenID authentication</u>, <u>ACL support</u>,
<u>CAS</u>, <u>LDAP authentication</u>, <u>Kerberos authentication</u>, and a plugin providing <u>user interface extensions</u> and

See the [Core plugin page](#) for basic information and the [user guide](#) for detailed information.

16.6.2 Shiro

[Shiro](#) is a Java POJO-oriented security framework that provides a default domain model that models permissions. With Shiro you extend a controller base class called `JsecAuthBase` in each controller yo provide an `accessControl` block to setup the roles. An example below:

```
class ExampleController extends JsecAuthBase {
    static accessControl = {
        // All actions require the 'Observer' role.
        role(name: 'Observer')

// The 'edit' action requires the 'Administrator' role.
        role(name: 'Administrator', action: 'edit')

// Alternatively, several actions can be specified.
        role(name: 'Administrator', only: [ 'create', 'edit', 'save', 'update' ])
    }
    …
}
```

For more information on the Shiro plugin refer to the [documentation](#).

17 Plugins

Grails is first and foremost a web application framework, but it is also a platform. By exposing a number of you extend anything from the command line interface to the runtime configuration engine, Grails can be 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 incredibl how to build a Grails application, you'll know how to create a plugin for <u>sharing a data model</u> or some stat

17.1 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-pl` create a new plugin project called `example`.

Make sure the plugin name does not contain more than one capital in a row, or it won't work. Camel case i

The structure of a Grails plugin is very nearly the same as a Grails application project's except that directory you will find a plugin Groovy file called the "plugin descriptor".

> ⚠ The only plugins included in a new plugin project are Tomcat and Release. Hibernate is not default.

Being a regular Grails project has a number of benefits in that you can immediately test your plugin by run

```
grails run-app
```

> ⚠ Plugin projects don't provide an index.gsp by default since most plugins don't need it. So, i view the plugin running in a browser right after creating it, you will receive a page not found can easily create a `grails-app/views/index.gsp` for your plugin if you'd like.

The plugin descriptor name ends with the convention `GrailsPlugin` and is found in the root of the plug

```
class ExampleGrailsPlugin {
    def version = "0.1"

…
}
```

All plugins must have this class in the root of their directory structure. The plugin class defines the versic 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", "1.1

- `grailsVersion` - The version of version range of Grails that the plugin supports. eg. "1.2 > *" (in

- `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 Local Plugins

To make your plugin available for use in a Grails application run the `maven-install` command:

```
  grails maven-install
```

This will install the plugin into your local Maven cache. Then to use the plugin within an application dec
plugin in your `grails-app/conf/BuildConfig.groovy` file:

```
  compile ":quartz:0.1"
```

Notes on excluded Artefacts

Although the [create-plugin](#) command creates certain files for you so that the plugin can be run as a Gra
these files are included when packaging a plugin. The following is a list of artefacts created, but not includ

- `grails-app/conf/BootStrap.groovy`

- `grails-app/conf/BuildConfig.groovy` (although it is used to generate `dependencies.`

- `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 (
executed when a plugin is installed, to provide such artefacts. In addition, although `UrlMappings.gro`
allowed to include a `UrlMappings` definition with a different name, such as `MyPluginUrlMappings`

Customizing the plugin contents

You can specify what to exclude in addition to the default excludes by adding elements to the `plug:`
property (described below). In addition, there are two ways to configure the contents of the plugin ZIP or J

One is to create an event handler for the `CreatePluginArchiveStart` event, which is fired after a
been copied to the staging directory. By adding an event handler you can add, modify, or delete files as ne
`_Events.groovy` in the `scripts` directory, for example

```
eventCreatePluginArchiveStart = { stagingDir ->
    // update staging directory contents here
}
```

You can customize the location of the staging directory with the `grails.project.plugin.sta`
`BuildConfig.groovy` or as as system property.

Note that there is also a `CreatePluginArchiveEnd` event which is fired after the ZIP or JAR is packa

You can also do this work in a Closure in `BuildConfig.groovy` with the property `grails.plugin`
analogous to the `grails.war.resources` property, e.g.

```
grails.plugin.resources = { stagingDir ->
    // update staging directory contents here
}
```

Specifying Plugin Locations

An application can load plugins from anywhere on the file system, even if they have not been installed. S
(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 installing i

- You have split an application into a set of plugins and an application, all in the same "super-project" c

> ⚠️ The Artifactory repository for Grails now includes all the dependencies for published plugins
> are using inline plugins that have dependencies, it is necessary to do a secondary resolve be
> dependencies might not be in the repository. Therefore, you should set `legacyResolve`
> your `BuildConfig.groovy` if you are using inline plugins with dependencies.

17.2 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. Th
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-inf](#)

```
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,
> instructions for getting an account on [this wiki page](#).

When you have access to the Grails Plugin repository, install the [Release Plugin](#) by declaring it as a 'buil
`grails-app/conf/BuildConfig.groovy` file:

```
grails.project.dependency.resolution = {
    …
    plugins {
        build ':release:3.0.0'
    }
}
```

And execute the `publish-plugin` command to release your plugin:

```
grails publish-plugin
```

This will automatically publish the plugin to the central repository. If the command is successful, it will
on the plugin portal at http://grails.org/plugin/<pluginName>. You can find out more about the Release plu
in its user guide.

Configuring Additional Repositories

The process for configuring repositories in Grails differs between versions. For version of Grails 1.2 and of
Grails 1.2 documentation on the subject. The following sections cover Grails 1.3 and above.

Grails 1.3 and above use Ivy under the hood to resolve plugin dependencies. The mechanism for d
repositories is largely the same as defining repositories for JAR dependencies. For example you can
repository that contains Grails plugins using the following syntax in grails-app/conf/BuildConf:

```
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)
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 Gra
be searched last:

```
repositories {
    grailsRepo "http://myserver/mygrailsrepo"
    grailsCentral()
}
```

Publishing to Maven Compatible Repositories

In general it is recommended for Grails 1.3 and above to use standard Maven-style repositories to self hos
doing so include the ability for existing tooling and repository managers to interpret the structure of a Mav

You use the Release plugin to publish a plugin to a Maven repository. Please refer to the section of the
guide on the subject.

17.3 Understanding a Plugin's Structure

As as mentioned previously, a plugin is basically a regular Grails application with a plugin descriptor. Hov
structure of a plugin differs slightly. For example, take a look at this plugin directory structure:

```
+ grails-app
     + controllers
     + domain
     + taglib
     etc.
 + lib
 + src
     + java
     + groovy
 + web-app
     + js
     + css
```

547

When a plugin is installed the contents of the grails-app directory will go into a plugins/example-1.0/grails-app. They **will not** be copied into the main source tree. A plugin project's primary source tree.

Dealing with static resources is slightly different. When developing a plugin, just like an application, all s web-app directory. You can then link to static resources just like in an application. This example links to

```
<g:resource dir="js" file="mycode.js" />
```

When you run the plugin in development mode the link to the resource will resolve to something However, when the plugin is installed into an application the path will automatically chang /plugin/example-0.1/js/mycode.js and Grails will deal with making sure the resources are in

There is a special pluginContextPath variable that can be used whilst both developing the plugin a installed into the application to find out what the correct path to the plugin is.

At runtime the pluginContextPath variable will either evaluate to an empty string or /plugins/ whether the plugin is running standalone or has been installed in an application

Java and Groovy code that the plugin provides within the lib and src/java and src/groovy director the main project's web-app/WEB-INF/classes directory so that they are made available at runtime.

17.4 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 t 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
application through plugins. Grails' view resolution mechanism will first look for the view in the applicati
if that fails will attempt to look for the view within the plugin. This means that you can override views
creating corresponding GSPs in the application's `grails-app/views` directory.

For example, consider a controller called `BookController` that's provided by an 'amazon' plugin. If the
list, Grails will first look for a view called `grails-app/views/book/list.gsp` then if that fail
view relative to the plugin.

However if the view uses templates that are also provided by the plugin then the following syntax may be

```
  <g:render template="fooTemplate" plugin="amazon"/>
```

Note the usage of the `plugin` attribute, which contains the name of the plugin where the template resid
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 generate dependencies.

- `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 yo scripts/_Install.groovy Gant script to install these artefacts into the target project's directory tre

In addition, the default `UrlMappings.groovy` file is excluded to avoid naming conflicts, howeve UrlMappings definition under a different name which **will** be included. For exam `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 them i

17.5 Evaluating Conventions

Before looking at providing runtime configuration based on conventions you first need to understan conventions from a plugin. Every plugin has an implicit `application` variable which is an instance interface.

The `GrailsApplication` interface provides methods to evaluate the conventions within the proj references to all artifact classes within your application.

Artifacts implement the [GrailsClass](#) interface, which represents a Grails resource such as a controller or a to get all `GrailsClass` instances you can do:

550

```
for (grailsClass in application.allClasses) {
    println grailsClass.name
}
```

GrailsApplication has a few "magic" properties to narrow the type of artefact you are interested i
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 example `application.c`

- `get*Class` - Retrieves a named class for a particular artefact.
  `application.getControllerClass("PersonController")`

- `is*Class` - Returns true if the given class is of the given artefact t
  `application.isControllerClass(PersonController)`

The GrailsClass interface has a number of useful methods that let you further evaluate and work wit
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 pa

- `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 p
  name

- `getPropertyName` - Returns the name of the class as a property name

- `getLogicalPropertyName` - Returns the logical property name of the class in the applica
  convention part if applicable

- `getNaturalName` - Returns the name of the property in natural terms (e.g. 'lastName' becomes 'La

- `getPackageName` - Returns the package name

For a full reference refer to the [javadoc API](#).

551

17.6 Hooking into Build Events

Post-Install Configuration and Participating in Upgrades

Grails plugins can do post-install configuration. This is achieved using a specially named script under th the plugin - `_Install.groovy`.

`_Install.groovy` is executed after the plugin has been installed.

This scripts is a [Gant](#) script, so you can use the full power of Gant. An addition to the standard Gant `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-ap` configuration template:

```
ant.mkdir(dir: "${basedir}/grails-app/jobs")

ant.copy(file: "${pluginBasedir}/src/samples/SamplePluginConfig.groovy",
         todir: "${basedir}/grails-app/conf")
```

The `pluginBasedir` variable is not available in custom scripts, but you can use `fooPluginDir`, w your plugin.

Scripting events

It is also possible to hook into command line scripting events. These are events triggered during execu plugin scripts.

For example, you can hook into status update output (i.e. "Tests passed", "Server running") and the creatio

A plugin just has to provide an `_Events.groovy` script to listen to the required events. Refer the do [into Events](#) for further information.

17.7 Hooking into Runtime Configuration

Grails provides a number of hooks to leverage the different parts of the system and perform runtime config

Hooking into the Grails Spring configuration

First, you can hook in Grails runtime configuration by providing a property called `doWithSpring` whi code. For example the following snippet is from one of the core Grails plugins that provides [i18n](#) support:

```
import org.springframework.web.servlet.i18n.CookieLocaleResolver
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor
import org.springframework.context.support.ReloadableResourceBundleMessageSource

class I18nGrailsPlugin {

def version = "0.1"

def doWithSpring = {
        messageSource(ReloadableResourceBundleMessageSource) {
            basename = "WEB-INF/grails-app/i18n/messages"
        }
        localeChangeInterceptor(LocaleChangeInterceptor) {
            paramName = "lang"
        }
        localeResolver(CookieLocaleResolver)
    }
}
```

This plugin configures the Grails `messageSource` bean and a couple of other beans to manage Locale
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 th
participate in the generation of the file. A plugin can provide a `doWithWebDescriptor` property th
code that gets passed the `web.xml` as an `XmlSlurper` `GPathResult`.

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 a
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 doesn't
important, so it's simplest to insert your custom filter definition immediately after the last `<context-p`
important for mappings, but the usual approach is to add it immediately after the last `<filter>` element

```
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
encoding filter or the SiteMesh filter. Fortunately you can insert filter mappings immediately after
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 b
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
    }
}
```

17.8 Adding Dynamic Methods at Runtime

The Basics

Grails plugins let you register dynamic methods with any Grails-managed or other class at runtime.
`doWithDynamicMethods` closure.

For Grails-managed classes like controllers, tag libraries and so forth you can add methods, cons
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
new method called `myNewMethod` to each controller. If you know beforehand the class you wish the
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
interact with objects within it. For example if you were implementing a method to interact with Hibe.
SessionFactory instance in combination with a 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 can in
dynamic constructors that use the application context to wire dependencies into 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!

17.9 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 advanced reloading of application state at runtime. For example, consider this simplified sni 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 refe resources to watch. If the watched resources specify a Groovy file, when it is changed it will automaticall 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 the "Se new service bean is re-registered with the `ApplicationContext` when one of the service classes chang

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 the will occur when you try to auto-wire the reloaded service into an older controller Class.

To get around this, you can specify which plugins another plugin "influences". This means that when one will reload itself and then reload its influenced plugins. For example consider this snippet from the `Servi`

```
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 can use the "observe" property:

```
def observe = ["controllers"]
```

In this case when a controller is changed you will also receive the event chained from the controllers plugin

It is also possible for a plugin to observe all loaded plugins by using a wildcard:

```
def observe = ["*"]
```

The Logging plugin does exactly this so that it can add the `log` property back to *any* artefact that change running.

17.10 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. two properties. The first is called `dependsOn`. For example, take a look at this snippet from the Hibernate

```
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`, `domainClass`, i

The dependencies will be loaded before the Hibernate plugin and if all dependencies do not load, then the

The `dependsOn` property also supports a mini expression language for specifying version ranges. A few can be seen below:

```
def dependsOn = [foo: "* > 1.0"]
def dependsOn = [foo: "1.0 > 1.1"]
def dependsOn = [foo: "1.0 > *"]
```

When the wildcard * character is used it denotes "any" version. The expression syntax also excludes any -ALPHA etc. so for example the expression "1.0 > 1.1" would match any of the following versions:

- 1.1

- 1.0

- 1.0.1

- 1.0.3-SNAPSHOT

- 1.1-BETA2

Controlling Load Order

559

Using `dependsOn` establishes a "hard" dependency in that if the dependency is not resolved, the plugin will not load. It is possible though to have a weaker dependency using the `loadAfter` and `loadBefore` properties.

```
def loadAfter = ['controllers']
```

Here the plugin will be loaded after the `controllers` plugin if it exists, otherwise it will just be loaded. adapt to the presence of the other plugin, for example the Hibernate plugin has this code in its `doWithSp`:

```
if (manager?.hasGrailsPlugin("controllers")) {
    openSessionInViewInterceptor(OpenSessionInViewInterceptor) {
        flushMode = HibernateAccessor.FLUSH_MANUAL
        sessionFactory = sessionFactory
    }
    grailsUrlHandlerMapping.interceptors << openSessionInViewInterceptor
}
```

Here the Hibernate plugin will only register an `OpenSessionInViewInterceptor` if the `contro` loaded. The `manager` variable is an instance of the [GrailsPluginManager](#) interface and it provides meth plugins.

You can also use the `loadBefore` property to specify one or more plugins that your plugin should load b

```
def loadBefore = ['rabbitmq']
```

Scopes and Environments

It's not only plugin load order that you can control. You can also specify which environments your plugin which scopes (stages of a build). Simply declare one or both of these properties in 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 pack because it's excluded from the 'war' phase. This allows `development-only` plugins to not be packaged

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,

```
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.

17.11 The Artefact API

You should by now understand that Grails has the concept of artefacts: special types of classes that it kn differently from normal Groovy and Java classes, for example by enhancing them with extra properties an artefacts include domain classes and controllers. What you may not be aware of is that Grails allow developers access to the underlying infrastructure for artefacts, which means you can find out what artefac enhance them yourself. You can even provide your own custom artefact types.

17.11.1 Asking About Available Artefacts

As a plugin developer, it can be important for you to find out about what domain classes, controllers, or c available in an application. For example, the Searchable plugin needs to know what domain classes exist any `searchable` properties and index the appropriate ones. So how does it do it? The `grailsApplication` object, and instance of GrailsApplication that's available automatically in contr be injected everywhere else.

The `grailsApplication` object has several important properties and methods for querying artel 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 [GrailsClass](#) that has some particularly useful properties and methods, including one for the underlying Cla

- shortName - the class name of the artefact without the package (equivalent of Class.simpleNa

- logicalPropertyName - the artefact name in property form without the 'type' suffix. So 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 or an best if the property is initialised on declaration, e.g. static 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 whether a class is a particular type of artefact. For example, grailsApplication.isControllerClass(org.example.MyGreatController) to MyGreatController is in fact a controller.

17.11.2 Adding Your Own Artefact Types

Plugins can easily provide their own artefacts so that they can easily find out what implementations are a reloading. All you need to do is create an ArtefactHandler implementation and register it in your ma

```
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 [ArtefactHandl](#) a bit easier, there is a skeleton implementation that can readily be extended: [ArtefactHandlerAdapter](#).

In addition to the handler itself, every new artefact needs a corresponding wrapper class that implem skeleton implementations are available such as [AbstractInjectableGrailsClass](#), which is particularly usefu into a Spring bean that is auto-wired, just like controllers and 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.

17.12 Binary Plugins

Regular Grails plugins are packaged as zip files containing the full source of the plugin. This has som being an open distribution system (anyone can see the source), in addition to avoiding problems with the s used for compilation.

As of Grails 2.0 you can pre-compile Grails plugins into regular JAR files known as "binary plugins". Th (and some disadvantages as discussed in the advantages of source plugins above) 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 example `target/f`
. There are two ways to incorporate a binary plugin into an application.

One is simply placing the plugin JAR file in your application's `lib` directory. The other is to publ
compatible Maven repository and declare it as a dependency in `grails-app/conf/BuildConfig.g`

```
dependencies {
    compile "mycompany:myplugin:0.1"
}
```

> ⚠️ Since binary plugins are packaged as JAR files, they are declared as dependencie
> `dependencies` block, *not* in the `plugins` block as you may be naturally inclined t
> `plugins` block is used for declaring traditional source plugins packaged as zip files

18 Grails and Spring

This section is for advanced users and those who are interested in how Grails integrates with and builds o
It is also useful for [plugin developers](#) considering doing runtime configuration Grails.

18.1 The Underpinnings of Grails

Grails is actually a [Spring MVC](#) application in disguise. Spring MVC is the Spring framework's built-i
framework. Although Spring MVC suffers from some of the same difficulties as frameworks like Struts in
it is superbly designed and architected and was, for Grails, the perfect framework to build another framewo

Grails leverages Spring MVC in the following areas:

- Basic controller logic - Grails subclasses Spring's [DispatcherServlet](#) and uses it to delegate to Grails c
- Data Binding and Validation - Grails' [validation](#) and [data binding](#) capabilities are built on those provi
- Runtime configuration - Grails' entire runtime convention based system is wired together by a Spring
- Transactions - Grails uses Spring's transaction management in [GORM](#)

In other words Grails has Spring embedded running all the way through it.

The Grails ApplicationContext

Spring developers are often keen to understand how the Grails `ApplicationContext` instance is con
are as follows.

- Grails constructs a parent `ApplicationContext` from the `web-app/WEB-INF/applicati`
  This `ApplicationContext` configures the [GrailsApplication](#) instance and the [GrailsPluginManag](#)
- Using this `ApplicationContext` as a parent Grails' analyses the conventions with the `Grails`
  and constructs a child `ApplicationContext` that is used as the root `ApplicationContext` o

Configured Spring Beans

Most of Grails' configuration happens at runtime. Each [plugin](#) may configure Spring beans that
`ApplicationContext`. For a reference as to which beans are configured, refer to the reference guide
the Grails plugins and which beans they configure.

18.2 Configuring Additional Beans

Using the Spring Bean DSL

You can easily register new (or override existing) beans by config
`grails-app/conf/spring/resources.groovy` which uses the Grails [Spring DSL](#). Beans are
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 dependenc
BootStrap.groovy and integration tests) by declaring a public field whose name is your bean's name (

```
class ExampleController {

def myBean
    …
}
```

Using the DSL has the advantage that you can mix bean declarations and logic, for example based on the e

```
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 u
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 re
> Grails or an installed plugin, your bean will replace the previous registration. This is a conver
> customize behavior without resorting to editing plugin code or other approaches that w
> maintainability.

Using XML

Beans can also be configured using a `grails-app/conf/spring/resources.xml`. In earlier v
was automatically generated for you by the `run-app` script, but the DSL in `resources.groovy` is th
so it isn't automatically generated now. But it is still 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 to co

The `myBean` bean that we configured using the DSL would be configured with this syntax in the XML fil

```xml
<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:

```groovy
class ExampleController {

def myBean
}
```

Referencing Existing Beans

Beans declared in `resources.groovy` or `resources.xml` can reference other beans by convention
a `BookService` class its Spring bean name would be `bookService`, so your bean would reference it l

```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 Groo 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 exist yet. As long as it's in place when the final application context configuration occurs, everything will be

For a full reference of the available beans see the plugin reference in the reference guide.

18.3 Runtime Spring with the Beans DSL

This Bean builder in Grails aims to provide a simplified way of wiring together dependencies that uses Spr

In addition, Spring's regular way of configuration (via XML and annotations) is static and difficult to runtime, other than programmatic XML creation which is both error prone and verbose. Grails' BeanBu making it possible to programmatically wire together components at runtime, allowing you to adapt th properties or environment variables.

This enables the code to adapt to its environment and avoids unnecessary duplication of code (having dif test, development and production environments)

The BeanBuilder class

Grails provides a grails.spring.BeanBuilder class that uses dynamic Groovy to construct bean definitions. T

```groovy
import org.apache.commons.dbcp.BasicDataSource
import org.codehaus.groovy.grails.orm.hibernate.ConfigurableLocalSessionFactoryBe
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 cr
> instance of BeanBuilder. Instead the DSL is implicitly available inside the doWithSp
> beans blocks respectively.

This example shows how you would configure Hibernate with a data source with the BeanBuilder clas

Each method call (in this case dataSource and sessionFactory calls) maps to the name of the argument to the method is the bean's class, whilst the last argument is a block. Within the body of the blo 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 sessionFactory bean resolves the dataSource reference.

Certain special properties related to bean management can also be set by the builder, as seen in the followi

```
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
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 sar
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` a
definitions. An example script can be seen below:

```
import org.apache.commons.dbcp.BasicDataSource
import org.codehaus.groovy.grails.orm.hibernate.ConfigurableLocalSessionFactoryBe

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)

573

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.

18.4 The BeanBuilder DSL Explained

Using Constructor Arguments

Constructor arguments can be defined using parameters to each bean-defining method. Put them after the f

```
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 use to c
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 of these factories. In this case the bean has no Class argument and instead you must pass the name of the 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 named parameter syntax:

575

```
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 myBe

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 strir
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. The e
value but would work just as well with a name that is generated programmatically based on configuration,

Furthermore, because sometimes bean names are not known until runtime you may need to reference the
together other beans, in this case using the ref method:

```
def beanName = "example"
bb.beans {

"${beanName}Bean"(MyExampleBean) {
        someProperty = [1, 2, 3]
    }

anotherBean(AnotherBean) {
        example = ref("${beanName}Bean")
    }
}
```

Here the example property of `AnotherBean` is set using a runtime reference to the `exampleBean`. The
used to refer to beans from a parent `ApplicationContext` that is provided in the constructor of the Be

```
ApplicationContext parent = ...//
def 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 tha

```
bb.beans {

marge(Person) {
        name = "Marge"
        husband = { Person p ->
            name = "Homer"
            age = 45
            props = [overweight: true, height: "1.8m"]
        }
        children = [ref('bart'), ref('lisa')]
    }

bart(Person) {
        name = "Bart"
        age = 11
    }

lisa(Person) {
        name = "Lisa"
        age = 9
    }
}
```

In the above example we set the `marge` bean's husband property to a block that creates an inner bean r
you have a factory bean you can omit the type and just use the specified bean definition instead to setup th

577

```
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 = [ref('bart'), ref('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 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 othe
> 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 argu
Later we define a knights bean that has no `Class` defined, but inherits the `Class` from 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
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
With Spring namespaces you also get full access to all of the powerful AOP support in Spring from B
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)"
        }
    }
}
```

18.5 Property Placeholder Configuration

Grails supports the notion of property placeholder configuration through an extended 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 in `grails-app/conf/spring/resources.xml` and `grails-app/conf/spring/reso` example given the following entries in `grails-app/conf/Config.groovy` (or an externalized conf

```
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>
```

To specify placeholders in `resources.groovy` you need to use single quotes:

```
dataSource(org.springframework.jdbc.datasource.DriverManagerDataSource) {
    driverClassName = '${database.driver}'
    url = 'jdbc:${database.dbname}'
}
```

This sets the property value to a literal string which is later resolved against the config by Spring's Property

A better option for `resources.groovy` is to access properties through the `grailsApplication` va

```
dataSource(org.springframework.jdbc.datasource.DriverManagerDataSource) {
    driverClassName = grailsApplication.config.database.driver
    url = "jdbc:${grailsApplication.config.database.dbname}"
}
```

Using this approach will keep the types as defined in your config.

18.6 Property Override Configuration

Grails supports setting of bean properties via configuration. This is often useful when used in combi
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
```

19 Grails and Hibernate

If [GORM](#) (Grails Object Relational Mapping) is not flexible enough for your liking you can alternatively using Hibernate, either with XML mapping files or JPA annotations. You will be able to map Grails do range of legacy systems and have more flexibility in the creation of your database schema. Best of all, yo all of the dynamic persistent and query methods provided by GORM!

19.1 Using Hibernate XML Mapping Files

Mapping your domain classes with XML is pretty straightforward. Simply create a `hibernate.cfg.x` `grails-app/conf/hibernate` directory, either manually or with the [create-hibernate-cfg-xml](#) co 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, `grails-app/conf/hibernate` directory. To find out how to map domain classes with XML, check .

If the default location of the `hibernate.cfg.xml` file doesn't suit you, you can change it by specifyi in `grails-app/conf/DataSource.groovy`:

```groovy
hibernate {
    config.location = "file:/path/to/my/hibernate.cfg.xml"
}
```

or even a list of locations:

```
hibernate {
    config.location = ["file:/path/to/one/hibernate.cfg.xml",
                       "file:/path/to/two/hibernate.cfg.xml"]
}
```

Grails also lets you write your domain model in Java or reuse an existing one that already has Hiberna
place the mapping files into grails-app/conf/hibernate and either put the Java files in src/j
project's lib directory if the domain model is packaged as a JAR. You still need the hibernate.cfg.:

19.2 Mapping with Hibernate Annotations

To map a domain class with annotations, create a new class in src/java and use the annotations defin
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 releva
`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
domain class see the section on [Scaffolding](#).

19.3 Adding Constraints

You can still use GORM validation even if you use a Java domain model. Grails lets you define constraint
in the `src/java` directory. The script must be in a directory that matches the package of the correspon
name must have a *Constraints* suffix. For example, if you had a domain class `org.example.Book`, th
script `src/java/org/example/BookConstraints.groovy`.

Add a standard GORM `constraints` block to the script:

```groovy
constraints = {
    title blank: false
    author blank: false
}
```

Once this is in place you can validate instances of your domain class!

20 Scaffolding

Scaffolding lets you generate some basic CRUD interfaces for a domain class, including:

- The necessary [views](#)
- Controller actions for create/read/update/delete (CRUD) operations

As of Grails 2.3, the scaffolding feature has been moved to a plugin. By default this is configured applications, but if you are upgrading from a previous version of Grails you will need to add the followi BuildConfig.groovy file:

```
plugins {
        …
        compile ":scaffolding:2.0.0"
        …
    }
```

Version 1.0.0 of the plugin provides the same scaffolding seen in Grails 2.2.x and below. Version 2.0.x includes different scaffolding templates that are aligned with the new REST APIs introduced in Grails 2.3

Dynamic Scaffolding

The simplest way to get started with scaffolding is to enable it with the scaffold property. Set the sc 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 cl: 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 actions are dynamically implemented by default by the runtime scaffolding mechanism:

- index
- show
- edit
- delete
- create
- save
- update

A CRUD interface will also be generated. To access this open `http://localhost:8080/app/book`

If you prefer to keep your domain model in Java and [mapped with Hibernate](#) you can still use scaffo 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 index() {
        [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 dynamically

> ⚠ By default, the size of text areas in scaffolded views is defined in the CSS, so adding 'row
> attributes will have no effect.
>
> Also, the standard scaffold views expect model variables of the
> <propertyName>InstanceList for collections and <propertyName>Instance
> 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
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:

589
```

```
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 custo
Grails lets you generate a controller and the views used to create the above interface from the comm
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 i
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
[install-templates](#) command.

21 Deployment

Grails applications can be deployed in a number of ways, each of which has its pros and cons.

"grails run-app"

You should be very familiar with this approach by now, since it is the most common method of running development phase. An embedded Tomcat server is launched that loads the web application from the de allowing it to pick up an changes to application files.

This approach is not recommended at all for production deployment because the performance is poor. C changes places a sizable overhead on the server. Having said that, `grails prod run-app` removes and lets you fine tune how frequently the regular check takes place.

Setting the system property "disable.auto.recompile" to `true` disables this regular check complet "recompile.frequency" controls the frequency. This latter property should be set to the number of second 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 t Hot-reloading is disabled, so you get good performance without the hassle of having to deploy the WAR fi

WAR file

When it comes down to it, current java infrastructures almost mandate that web applications are deployed by far the most common approach to Grails application deployment in production. Creating a WAR file the war command:

```
grails war
```

There are also many ways in which you can customise the WAR file that is created. For example, you (absolute or relative) to the command that instructs it where to place the file and what name to give it:

```
grails war /opt/java/tomcat-5.5.24/foobar.war
```

Alternatively, you can add a line to `grails-app/conf/BuildConfig.groovy` that changes 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 directory. The default behavior is to include in the WAR file all libraries required by Grails, plus any libr "lib" directories, plus any libraries contained in the application's "lib" directory. As an alternative to the explicitly specify the complete list of libraries to include in the WAR file by setting the property grails in BuildConfig.groovy to either lists of Ant include patterns or closures containing AntBuilder syntax. C within an Ant "copy" step, so only elements like "fileset" can be included, whereas each item in a patt closure or pattern assigned to the latter property will be included in addition to grails.war.depender

Be careful with these properties: if any of the libraries Grails depends on are missing, the application v 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 y application will probably not work. You can find a list of dependencies required by Grails in the "depende directory of the unpacked distribution. You can also find a list of the default dependencies included in "War.groovy" script - see the DEFAULT_DEPS and DEFAULT_J5_DEPS variables.

The remaining two configuration options available to you are grails.war.copyToWebApp and grai The first of these lets you customise what files are included in the WAR file from the "web-app" directory any extra processing you want before the WAR file is finally created.

```
// This closure is passed the command line arguments used to start the
// war process.
grails.war.copyToWebApp = { args ->
    fileset(dir:"web-app") {
        include(name: "js/**")
        include(name: "css/**")
        include(name: "WEB-INF/**")
    }
}

// This closure is passed the location of the staging directory that
// is zipped up to make the WAR file, and the command line arguments.
// Here we override the standard web.xml with our own.
grails.war.resources = { stagingDir, args ->
    copy(file: "grails-app/conf/custom-web.xml",
        tofile: "${stagingDir}/WEB-INF/web.xml")
}
```

Application servers

Ideally you should be able to simply drop a WAR file created by Grails into any application server and
away. However, things are rarely ever this simple. The [Grails website](#) contains a list of application ser
tested with, along with any additional steps required to get a Grails WAR file working.

22 Contributing to Grails

Grails is an open source project with an active community and we rely heavily on that community to hel such, there are various ways in which people can contribute to Grails. One of these is by writing useful publicly available. In this chapter, we'll look at some of the other options.

22.1 Report Issues in JIRA

Grails uses JIRA to track issues in the core framework, its documentation, its website, and many of the found a bug or wish to see a particular feature added, this is the place to start. You'll need to create a (free) 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 versions of Grails and various plugins you are using. Also, an issue is much more likely to be dealt with if sample application (which can be packaged up using the `grails bug-report` command).

Reviewing issues

There are quite a few old issues in JIRA, some of which may no longer be valid. The core team can't trad very simple contribution that you can make is to verify one or two issues occasionally.

Which issues need verification? A shared JIRA filter will display all issues that haven't been resolved and someone else in the last 6 months. Just pick one or two of them and check whether they are still relevant.

Once you've verified an issue, simply edit it and set the "Last Reviewed" field to today. If you think the also check the "Flagged" field and add a short comment explaining why. Once those changes are saved from the results of the above filter. If you've flagged it, the core team will review and close if it really is no

One last thing: you can easily set the above filter as a favourite on this JIRA screen so that it appears in Just click on the star next to a filter to make it a favourite.

22.2 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 source, build it and test it with your own applications. Before you start, make sure you 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 GitHub in several repositories owned by the "grails" GitHub user. This is a simple case of cloning the re in. For example, to get the 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 source get a Grails installation from the source.

595

Creating a Grails installation

If you look at the project structure, you'll see that it doesn't look much like a standard GRAILS_HOME i
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
skips the extensive collection of Grails test classes, which can take some time to complete.

Once the above command has finished, simply set the GRAILS_HOME environment variable to the check
"bin" directory to your path. When you next type run the grails command, you'll be using the version yo

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. For exa
BinaryPluginSpec simply execute the following command:

```
./gradlew :grails-core:test --tests *.BinaryPluginSpec
```

Note that you need to specify the sub-project that the test case resides in, because the top-level "test" targe

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 them do the following:

* Add the springloaded-core JAR file in $GRAILS_HOME/lib/org.springsource.springloaded/s grails-core's classpath.

* Remove "src/test/groovy" from grails-plugin-testing's source path GRECLIPSE-1067

* Add the jsp-api JAR file in $GRAILS_HOME/lib/javax.servlet.jsp/jsp-api/jars to the classpath of grai

* Fix the source path of grails-scripts. Add linked source folder linking to "../scripts". If you get build e "../gradlew cleanEclipse eclipse" in that directory and edit the .classpath file again (remove th kind="src" path="../scripts"/>"). Remove possible empty "scripts" directory under grails-scripts if yo 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 an Share project (not "Share projects"). Choose "Git". Then check "Use or create repository in parent fo "Finish".

* Get the recommended code style settings from the mailing list thread (final style not decided ye Import the code style xml file to STS in Window->Preferences->Java->Code Style->Formatter->I spaces instead of tabs for indenting.

Debugging Grails or a Grails application

To enable debugging, run:

```
grails --debug-fork run-app
```

By default Grails forks a JVM to run the application in. The `--debug-fork` argument causes the debug
the forked JVM. In order to instead attach the debugger to the build system which is going to fork th
option:

```
grails -debug run-app
```

22.3 Submit Patches to Grails Core

If you want to submit patches to the project, you simply need to fork the repository on GitHub rather tha
you will commit your changes to your fork and send a pull request for a core team 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 rep](#)
[requests](#) with your changes.

What follows are some guidelines to help ensure that your pull requests are speedily dealt with and pro
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 exampl
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
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 w
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 date why a change was made. Include the ID in any and all commits that relate to that issue. If a commi 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 y 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 pu changes are currently on the local "mine" branch but not on "master". The first step involves pulling an repository that have been added since you last fetched and merged:

```
git checkout master
git pull upstream
```

This should complete without any problems or conflicts. Next, rebase your local branch against the now up

```
git checkout mine
git rebase master
```

What this does is rearrange the commits such that all of your changes come after the most recent one in m 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 develope 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 t
please specify the IDs of all issues that the request relates to. Also give a brief description of the work y
refactored the data binder and added support for custom number editors (GRAILS-xxxx)".

22.4 Submit Patches to Grails Documentation

Contributing to the documentation is simpler for the core framework because there is a
http://github.com/grails/grails-doc project that anyone can request commit access to. So, if you want
documentation, simply request commit access to the following repository http://github.com/pledbrook
GitHub message to 'pledbrook' and then commit your patches just as you would to any other GitHub repos

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 Gradle m
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 you to s
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
don't specify a grails.home property, then the build will fetch the Grails source - a download of 10s o
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

```
./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-directory
part and the ref folder containing the reference material. To view the user guide, simply open build/do

Publishing

The publishing system for the user guide is the same as [the one for Grails projects](). You write your cha
gdoc wiki format which is then converted to HTML for the final guide. Each chapter is a top-
src/<lang>/guide directory. Sections and sub-sections then go into directories with the same nam
without the suffix.

The structure of the user guide is defined in the src/<lang>/guide/toc.yml file, which is a Y
defines the (language-specific) section titles. If you add or remove a gdoc file, you must update the TOC a

601

The `src/<lang>/ref` directory contains the source for the reference sidebar. Each directory is the na
also appears in the docs. Hence the directories need different names for the different languages. Inside th
files, whose names match the names of the methods, commands, properties or whatever that the files descr

Translations

This project can host multiple translations of the user guide, with `src/en` being the main one. To add and
new language directory under `src` and copy into it all the files under `src/en`. The build will take care of

Once you have a copy of the original guide, you can use the `{hidden}` macro to wrap the English tex
rather than remove it. This makes it easier to compare changes to the English guide against your translation

```
{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 aplicao Grails com o comando [create-app|commandLine], Grails
no cria automaticamente um ficheiro de construo 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. Yo
of `diff` to see which bits of your translation need updating. On top of that, the `{hidden}` macro ensure
not displayed in the browser, although you can display it by adding this URL
`javascript:toggleHidden();` (requires you to build the user guide 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 nee
like so:

```
./left_to_do.groovy es
```

This will then print out a recursive diff of the given translation against the reference English user guide.
blocks that hasn't changed since being translated will *not* appear in the diff output. In other words, all yo
hasn't been translated yet and content that has changed since it was translated. Note that `{code}` blocks
need to include them inside `{hidden}` macros.

To provide translations for the headers, such as the user guide title and subtitle, just add languag
'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 above title will be El Grails Framework for the Spanish translation. Also, translators can be credited by add property:

```
fr.translators=Stphane Maldini
```

This should be a comma-separated list of names (or the native language equivalent) and it will be displa header in the user guide itself.

You can build specific translations very easily using the `publishGuide_*` and `publishPdf_*` tas 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 in `build/` 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 without having to build the docs yourself.