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: 3.2.0.M2

Table of Contents

1 Introduction

Java web development as it stands today is dramatically more complicated than it needs to be. Most moc complicated and don't embrace the Don't Repeat Yourself (DRY) principles.

Dynamic frameworks like Rails, Django and TurboGears helped pave the way to a more modern way of tł these concepts and dramatically reduces the complexity of building web applications on the Java platfo1 does so by building on already established Java technologies like Spring and Hibernate.

Grails is a full stack framework and attempts to solve as many pieces of the web development puzzle 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
- An interactive command line environment and build system based on Gradle
- 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 Dom

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

1.1 What's new in Grails 3.2?

This section covers all the new features introduced in Grails 3.2. Note that Grails 3.2 is at the milestone sta

Grails 3.2 includes the following new features.

1.1.1 GORM 6 Suite

Grails 3.2 comes with GORM 6.0, the biggest release of GORM ever! GORM 6 includes the following nev

- GORM for Neo4j 3.0 / Bolt Driver support
- GORM for MongoDB 3.2
- RxGORM - GORM for RxJava
- RxGORM for REST built on RxNetty
- RxGORM for MongoDB Rx Driver
- Universal Multiple Data Sources Support
- Multi Tenancy Support
- Spring Container Free Bootstrapping
- Improved Unit Testing
- Unified Configuration API
- New Standalone Documentation

There are so many new features and novelties in GORM that we had to write its own independent What's N

1.1.2 AngularJS 1.0 Scaffolding

The `angular` profile has been refined and now also includes a new Angular Scaffolding plugin.

The Angular scaffolding plugin adds an `ng-generate-all` command which will generate the necessa operations in conjunction with a Grails 3 backend.

Not only does this serve as a useful tool to get up and running quickly, but (like previous versions of scaf how to integrate AngularJS and Grails 3.

> ⚠ Support for Angular 2 is planned for the future.

1.1.3 JSON Views 1.1

Version 1.1 of the JSON Views plugin is included with Grails 3.2's "rest-api" profile and includes a n highlights:

Template Inheritance

It is now possible for a child JSON template to specify a parent template, thus allowing better template parent:

```
// grails-app/views/_parent.gson
model {
    Object object
}
json {
    hal.links(object)
    version "1.0"
}
```

A child template can be specified as follows:

```
inherits template:"parent"
model {
    Person person
}
json {
    name person.name
}
```

Global and Default Templates

Global templates can now be created for any GORM custom types. This allows adding support for exter provided by datastores such as MongoDB (example GeoJSON).

A global template is simply another JSON template that is named after the class name. See for example the

In addition it is now possible to provide a fallback template named `/object/_object.gson` that is us

Better HAL Support

The HAL support has been expanded and now includes greater control over `_embedded` and `_links`, fo

```
model {
    Book book
}
json {
    hal.links(self: book )
    hal.embedded(authors: book.authors)
    hal.inline(book) {
        pages 300
    }
}
```

The HAL support has also been improved with support for HAL pagination.

1.1.4 Updated Dependencies

Grails 3.2 ships with the following dependency upgrades:

- Hibernate 5.1.0 (now the default version of Hibernate for new applications)
- Spring Framework 4.3
- Spring Boot 1.4 RC1

1.1.5 Other Novelties

New default date data binding format

Dates formatted like "1970-01-01T00:00:00.000Z" will now be successfully parsed by default. The format

The `run-script` command from Grails 2 is back

The `run-script` command makes a return! It is now possible to run Groovy scripts that are wrapped in a G

```
$ grails run-script my-groovy-script.groovy
```

Refer to the run-script documentation for more information.

REST Profile Refinements

The REST profile has been further refined including more sensible `UrlMappings` and mime type applications.

Ability to skip the Bootstrap process with a system property

When the Grails runtime is started, it will now execute `*Bootstrap.groovy` classe: `grails.bootstrap.skip` is set to `true`, the classes will *not* be executed for that run.

1.2 What's new in Grails 3.1?

Grails 3.1 includes the following new features.

Spring Boot 1.3 and Spring 4.2

Grails 3.1 has been upgraded to Spring Boot 1.3 and Spring 4.2.

1.2.1 Improvements to Grails 3 Profiles

Profile Publishing and Repositories

The following improvements are available in Grails profiles:

- Profiles are now published as regular JAR files to any Maven compatible repository (Artifactory, Nex
- Additional profiles can be created easily with the new create-profile command.
- Profiles can now contribute to the generation of the build
- Profiles can now have one or many features

For more information see the new section on Profiles in the user guide.

1.2.2 REST API and AngularJS Profiles

REST Profile

A new profile is available designed for the creation of pure REST applications without a UI.

To create a REST application use the `rest-api` profile as an argument to create-app:

```
$ grails create-app myapp --profile=rest-api
```

In earlier milestones this profile was named web-api. The profile has been renamed `rest-a` describes its purpose.

Then start interactive mode to see the available commands for the profile:

```
$ cd myapp
$ grails
```

If you hit TAB you will notice code generation commands specific to the profile including:

- `create-domain-resource` - Creates a domain class annotated with the Resource annotation)
- `create-restful-controller` - Creates a controller that extends RestfulController.

JSON and Markup Views

The REST profile includes the ability to define JSON and Markup views and the `build.gradle` i production use.

The REST profile also creates JSON views to render the `index` action and common commands such as g JSON views.

AngularJS Profile

An initial version of the AngularJS profile is now available, making it easier to create and integrate Angula

To create a Grails 3 AngularJS application use the `angular` profile as an argument to create-app:

```
$ grails create-app myapp --profile=angular
```

Then start interactive mode to see the available commands for the profile:

```
$ cd myapp
$ grails
```

You will notice new commands such as `create-ng-component`, `create-ng-controller` etc. application.

The `build.gradle` is also preconfigured with the necessary Gradle plugins to integrate Angular application can be found in `grails-app/assets/javascripts`.

1.2.3 GORM 5 Suite

Grails 3.1 ships with GORM 5 which is a brand new release of GORM supporting the following technolog

- Hibernate 3, 4 and 5 - for SQL databases GORM for Hibernate now supports the latest Hibernate 5.x
- MongoDB 3.x - GORM for MongoDB has been upgraded to the MongoDB 3.x Java driver and suppo
- Neo4j 2.3.x - GORM for Neo4j has been significantly improved and support the latest release of Neo4
- Cassandra - GORM for Cassandra supports the latest 2.0.x drivers

For more information refer to the new GORM 5 website.

1.2.4 Plugin Publishing Plugins

New Gradle plugins are available to simplify publishing of plugins and profiles.

To utilize the plugin apply the `org.grails.grails-plugin-publish` plugin (after any existing G

```
apply plugin: "org.grails.grails-plugin"
apply plugin: "org.grails.grails-plugin-publish"
```

For a profile the `grails-profile-publish` plugin can be used instead:

```
apply plugin: "org.grails.grails-profile"
apply plugin: "org.grails.grails-profile-publish"
```

Then configure the plugin. For example:

```
grailsPublish {
    user = 'user'
    key = 'key'
    githubSlug = 'foo/bar'
    license {
        name = 'Apache-2.0'
    }
    title = "My Plugin Title"
    desc = "My Plugin Description"
    developers = [johndoe:"John Doe"]
}
```

The `user` and `key` are your Bintray credentials. With this done you can continue to use `bintrayUpl` wish to update the Grails plugin portal, you simply need to configure your `grails.org` credentials:

```
grailsPublish {
    …
    portalUser = "..."
    portalPassword = "..."
}
```

Then call `notifyPluginPortal` to update the Grails.org Plugins website:

```
gradle notifyPluginPortal
```

2 Getting Started

2.1 Installation Requirements

Before installing Grails 3.0 you will need as a minimum a Java Development Kit (JDK) installed version 1 your operating system, run the installer, and then set up an environment variable called `JAVA_HOME` point

To automate the installation of Grails we recommend [SDKMAN](#) which greatly simplifies installing and ma

For manual installation, we recommend the video installation guides from [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 many 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.

The best way to install Grails on *nix systems is with [SDKMAN](#) which greatly simplifies installing and ma

For manual installation 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 sc `GRAILS_HOME=/path/to/grails` to your profile
  * On Windows this is typically a matter of setting an environment variable under `My Computer/`
* Then add the `bin` directory to your `PATH` variable:
  * On Unix/Linux based systems this can be done by adding `export PATH="$PATH:$GRAILS`
  * On Windows this is done by modifying the `Path` environment variable under `My Computer/`

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

bc. Grails version: 3.0.0

2.3 Creating an Application

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

```
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

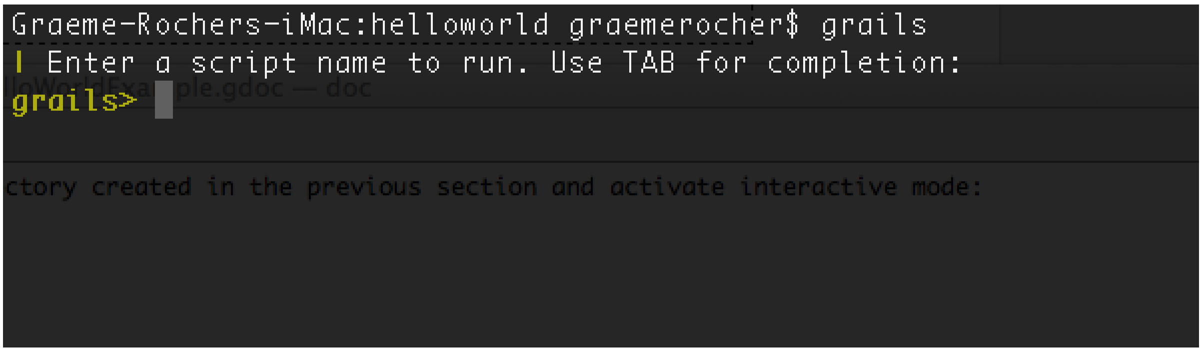
```
cd helloworld
```

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 ' the Grails interactive console:

```
$ cd helloworld
$ grails
```

You should see a prompt that looks like this:



What we want is a simple page that just prints the message "Hello World!" to the browser. In Grails, when
controller action for it. Since we don't yet have a controller, let's create one now with the create-controller c

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

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

The above command will create a new controller in the `grails-app/control`
`HelloController.groovy`. Why the extra `helloworld` directory? Because in Java land, it's strong
packages, so Grails defaults to the application name if you don't provide one. The reference page for create

We now have a controller so let's add an action to generate the "Hello World!" page. The code looks like th

```
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
http://localhost:8080/ - try it!

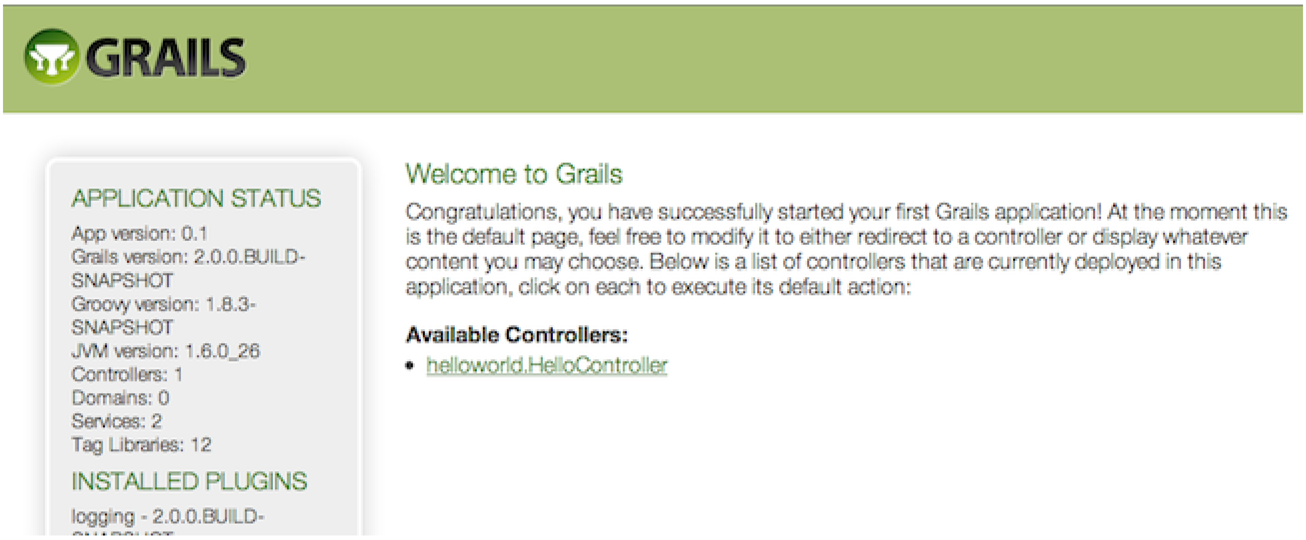
Note that in previous versions of Grails the context path was by default the name of the application. If you
a context path in `grails-app/conf/application.yml`:

```
server:
    'contextPath': '/helloworld'
```

With the above configuration in place the server will instead startup at the URL http://localhost:8080/hello

> If you see the error "Server failed to start for port 8080: Address already in use", then it mean
> that port. You can easily work around this by running your server on a different port usin
> '9090' is just an example: you can pretty much choose anything within the range 1024 to 4915

The result will look something like this:

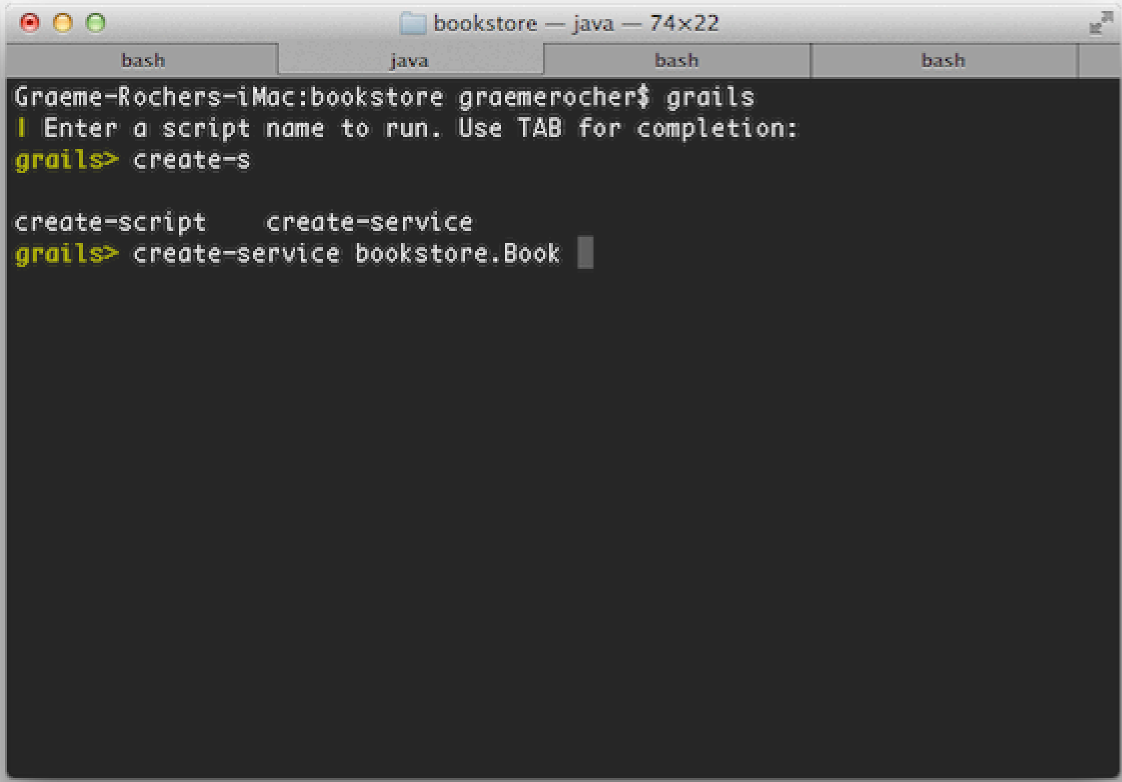


This is the Grails intro page which is rendered by the `grails-app/view/index.gsp` file. It detect
links to them. You can click on the "HelloController" link to see our custom page containing the text "Hel
Grails application.

One final thing: a controller can contain many actions, each of which corresponds to a different pag
accessible via a unique URL that is composed from the controller name and the action name: /<appnan
access the Hello World page via /helloworld/hello/index, where 'hello' is the controller name (remove
lower-case the first letter) and 'index' is the action name. But you can also access the page via the same
'index' is the *default action* . See the end of the controllers and actions section of the user guide to find out

2.5 Using Interactive Mode

Grails 3.0 features an interactive mode which makes command execution faster since the JVM doesn't l
interactive mode simple type 'grails' from the root of any projects and use TAB completion to get a list of
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](#) is an excellent IDE for Grails 3.0 development. It comes in 2 editions, the free community ed

The community edition can be used for most things, although GSP syntax higlighting is only part of the ul and Grails 3.0 simply go to `File / Import Project` and point IDEA at your `build.gradle` file

Eclipse

We recommend that users of [Eclipse](#) looking to develop Grails application take a look at [Groovy/Grails To](#) including automatic classpath management, a GSP editor and quick access to Grails commands.

Like Intellij you can import a Grails 3.0 project using the Gradle project integration.

NetBeans

NetBeans provides a Groovy/Grails plugin that automatically recognizes Grails projects and provides th code completion and integration with the Glassfish server. For an overview of features see the [NetBeans](#) was written by the NetBeans team.

TextMate, Sublime, VIM etc.

There are several excellent text editors that work nicely with Groovy and Grails. See below for references:

- A TextMate bundle exists Groovy / Grails support in Textmate
- A Sublime Text plugin can be installed via Sublime Package Control for the Sublime Text Editor.
- See this post for some helpful tips on how to setup VIM as your Grails editor of choice.
- An Atom Package is available for use with the Atom editor

2.7 Convention over Configuration

Grails uses "convention over configuration" to configure itself. This typically means that the name a configuration, hence you need to familiarize yourself with the directory structure provided by Grails.

Here is a breakdown and links to the relevant sections:

- `grails-app` - top level directory for Groovy sources
  - `conf` - Configuration sources.
  - `controllers` - Web controllers - The C in MVC.
  - `domain` - The application domain.
  - `i18n` - Support for internationalization (i18n).
  - `services` - The service layer.
  - `taglib` - Tag libraries.
  - `utils` - Grails specific utilities.
  - `views` - Groovy Server Pages - The V in MVC.
- `scripts` - Code generation scripts.
- `src/main/groovy` - Supporting sources
- `src/test/groovy` - Unit and integration tests.

2.8 Running and Debugging an Application

Grails applications can be run with the built in Tomcat server using the run-app command which will load

```
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
```

You can debug a grails app by simply right-clicking on the `Application.groovy` class in your IDE a
3).

Alternatively, you can run your app with the following command and then attach a remote debugger to it.

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

More information on the [run-app](run-app) 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 within the `src.`
you to populate these tests with valid test logic, information on which can be found in the section on Testir

To execute tests you run the test-app command as follows:

```
grails test-app
```

2.10 Deploying an Application

Grails applications can be deployed in a number of different ways.

If you are deploying to a traditional container (Tomcat, Jetty etc.) you can create a Web Application Ar command for performing this task:

```
grails war
```

This will produce a WAR file under the `build/libs` directory which can then be deployed as per your c

Note that by default Grails will include an embeddable version of Tomcat inside the WAR file, this ca version of Tomcat. If you don't intend to use the embedded container then you should change the scope o to deploying to your production container in `build.gradle`:

```
provided "org.springframework.boot:spring-boot-starter-tomcat"
```

If you are building a WAR file to deploy on Tomcat 7 then in addition you will need to change the tar against Tomcat 8 APIs by default. To target a Tomcat 7 container, insert a line to `build.gradle` above

```
ext['tomcat.version'] = '7.0.59'
```

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

```
grails dev war
```

If you prefer not to operate a separate Servlet container then you can simply run the Grails WAR file as a r

```
grails war
java -Dgrails.env=prod -jar build/libs/mywar-0.1.war
```

When deploying Grails you should always run your containers JVM with the `-server` option and with s
flags would be:

```
-server -Xmx768M -XX:MaxPermSize=256m
```

2.11 Supported Java EE Containers

Grails runs on any container that supports Servlet 3.0 and above and is known to work on the following spe

- Tomcat 7
- GlassFish 3 or above
- Resin 4 or above
- JBoss 6 or above
- Jetty 8 or above
- Oracle Weblogic 12c or above
- IBM WebSphere 8.0 or above

> ⚠ It's required to set "-Xverify:none" in "Application servers > server > Process Definition > Jav
> JVM arguments" for older versions of WebSphere. This is no longer needed for WebSphere v

Some containers have bugs however, which in most cases can be worked around. A list of known deploym

2.12 Creating Artefacts

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

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

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

```
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 interactive mode.

2.13 Generating an Application

To get started quickly with Grails it is often useful to use a feature called Scaffolding to generate the skele
`generate-*` commands such as generate-all, which will generate a controller (and its unit test) and the a

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

3 Upgrading from previous versions of Grails

Grails 3.0 is a complete ground up rewrite of Grails and introduces new concepts and components for man

When upgrading an application or plugin from Grails 3.0 there are many areas to consider including:

- Removal of dynamic scaffolding from Grails 3.0.0 till 3.0.4 when it was re-introduced
- Removal of before and after interceptors
- Project structure differences
- File location differences
- Configuration differences
- Package name differences
- Legacy Gant Scripts
- Gradle Build System
- Changes to Plugins
- Source vs Binary Plugins

The best approach to take when upgrading a plugin or application (and if your application is using several to create a new Grails 3.0 application of the same name and copy the source files into the correct locations

Removal of before and after interceptors

Before and after interceptors were removed. So all `beforeInterceptor` and `afterInterceptor` r

File Location Differences

The location of certain files have changed or been replaced with other files in Grails 3.0. The following tab new locations:

Old Location	New Location
grails-app/conf/BuildConfig.groovy	build.gradle
grails-app/conf/Config.groovy	grails-app/conf/application.groovy
grails-app/conf/UrlMappings.groovy	grails-app/controllers/UrlMappings.g
grails-app/conf/BootStrap.groovy	grails-app/init/BootStrap.groovy
scripts	src/main/scripts
src/groovy	src/main/groovy
src/java	src/main/groovy (yes, groovy!)
test/unit	src/test/groovy
test/integration	src/integration-test/groovy
web-app	src/main/webapp or src/main/resources/
*GrailsPlugin.groovy	src/main/groovy

`src/main/resources/public` is recommended as `src/main/webapp` only gets included in WA

It is recommended to merge Java source files from `src/java` into `src/main/groovy`. You can crea and it will be used but it is generally better to combine the folders. (The Groovy and Java sources compile

For plugins the plugin descriptor (a Groovy file ending with "GrailsPlugin") which was previously locat moved to the `src/main/groovy` directory under an appropriate package.

New Files Not Present in Grails 2.x

The reason it is best to create a new application and copy your original sources to it is because there ar Grails 2.x by default. These include:

File	Description
build.gradle	The Gradle build descriptor locate
gradle.properties	Properties file defining the Grails
grails-app/conf/logback.groovy	Logging previously defined in Co
grails-app/conf/application.yml	Configuration can now also be def
grails-app/init/PACKAGE_PATH/Application.groovy	The Application class used B

Files Not Present in Grails 3.x

Some files that were previously created by Grails 2.x are no longer created. These have either been rem following table lists files no longer in use:

File	Description
`application.properties`	The application version is now defined in `bui` the directory name, which can be overridden setting the `rootProject.name` property
`grails-app/conf/DataSource.groovy`	Merged together into `application.yml`
`lib`	Dependency resolution should be used to resolv
`web-app/WEB-INF/applicationContext.xml`	Removed, beans can be defined in `grails-ap`
`src/templates/war/web.xml`	Grails 3.0 no longer requires web.xml. Customi
`web-app/WEB-INF/sitemesh.xml`	Removed, sitemesh filter no longer present.
`web-app/WEB-INF/tld`	Removed, can be restored in `src/main/weba`

3.1 Upgrading from Grails 3.1

If you are upgrading from Grails 3.1 there are a few items to take into consideration.

Spring 4.3

Grails 3.2 comes with Spring 4.3 which no longer supports Hibernate 3 and hence Grails 3.2 no longer s upgrade to Hibernate 4 or above.

Spring Boot 1.4 RC1

Spring Boot 1.4 RC1, through its dependency management mechanism, enforces the upgrade for dependencies following the upgrade to ensure the new versions are compatible with your application.

Spring Boot 1.4 RC1 also deprecates many testing annotations (such as `WebIntegrationTest`).

See the [Spring Boot 1.4 release notes] for more information on the changes required at the Boot level.

Hibernate 4 Usage

Related to Spring Boot 1.4, one important change is that Hibernate 5 is now the default version, so if you plugin in Grails such as:

```
compile "org.grails.plugins:hibernate4"
```

This will not be enough to ensure that Hibernate 4 is used. You must instead also directly declare the Hiber

```
dependencies {
    compile "org.grails.plugins:hibernate4"
    compile "org.hibernate:hibernate-core:4.3.10.Final"
    compile "org.hibernate:hibernate-ehcache:4.3.10.Final"
}
```

GORM 6 Configuration Model

In preparation for Hibernate 5.2 support the previous "SessionFactoryBean" notion has been removed. Nov creation you should instead register a custom `org.grails.orm.hibernate.connections.Hi` Spring.

HibernateTestMixin Dependency Changes

The `grails-datastore-test-support` dependency has been removed and the `HibernateTestMixin` class inte a resolve error remove the following dependency from your `build.gradle`:

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

3.2 Upgrading from Grails 3.0

Generally to upgrade an application from Grails 3.0 you can simply modify the version of Grails in `gradl`

There are however some differences to Grails 3.0.x that are documented below.

GORM 5 Upgrade

Grails 3.1 ships with GORM 5, which is a near complete rewrite of GORM ontop of Groovy traits and is of GORM.

If you receive an error such as:

```
Caused by: java.lang.ClassNotFoundException: org.grails.datastore.gorm.GormEntity
    … 8 more
```

You are using a plugin or class that was compiled with a previous version of GORM and these will need compatible.

Hibernate Plugin

For the GORM 5 release the `hibernate` plugin has been renamed to `hibernate4` (and there are h You should change your `build.gradle` to reflect that:

```
compile 'org.grails.plugins:hibernate4'
```

Static Resources Path

The default path for static resources resolved from `src/main/resources/public` has been chan instead of directly under the root of the application. For example a link in GSP pages such as:

```
${g.resource(dir:'files', file:'mydoc.pdf')}
```

Will produce a URI such as `/static/files/mydoc.pdf` instead of `/files/mydoc.pdf`. If you configure this in `application.yml`:

```
grails:
    resources:
        pattern: '/**'
```

Filters Plugin Removed

The Filters plugin was replaced by Interceptors in Grails 3.0.x, although the plugin was still present. In Gra you still wish to use the filters plugin you can add a dependency on the previous version provided by Grail

```
compile 'org.grails:grails-plugin-filters:3.0.12'
```

Spring Transactional Proxies

Because the `grails.transactional.Transactional` transform already provides the ability to c
proxies, traditional support for transactional proxies has been disabled by default.

This means that if you have any services that use the `transactional` property and not the `Transact`
For example the following service:

```
class FooService {
    static transactional = true
}
```

Becomes:

```
import grails.transaction.Transactional

@Transactional
class FooService {

}
```

In addition because in previous versions of a Grails `transactional` defaulted to `true` any services tl
altered too.

If you wish to revert to the previous behavior then transctional proxies can be re-enabled with the followin

```
grails:
    spring:
        transactionManagement:
            proxies: true
```

JSON Converter changes

The default JSON converter no longer includes the `class` property by default. This can be re-enable with

```
grails:
    converters:
        domain:
            include:
                class: true
```

In addition the default JSON converter will no longer render the `id` property if it is `null`.

JSON Builder Groovy Alignment

The class `grails.web.JSONBuilder` has been deprecated and replaced with [groovy.json.Streamir](#)
Groovy. This avoids confusion with the differences between JSON builders and better aligns with Groovy'

This also means that any `render` blocks that rendered JSON will need to be updated to use the [groovy.j](#)
the following code:

```
render(contentType:"application/json") {
    title = "The Stand"
}
```

Should instead be written as:

```
render(contentType:"application/json") {
    title "The Stand"
}
```

If you are upgrading and prefer to continue to use the previous implementation then you can re-enable t
configuration:

```
grails:
    json:
        legacy:
            builder: true
```

JSON Views Replace JSON Converters

With the addition of JSON views the previous API for using JSON converters is largely discouraged in fa future be separated into an external plugin and JSON views phased in to replace it. The JSON converte provide a more fully featured, elegant API that is superior to writing JSON converters and/or marshallers.

Spring Boot 1.3 and Spring 4.2

Grails 3.1 ships with upgraded third party libraries that may require changes. See the [Spring Boot upgrade]

Unlike Spring Boot 1.2, Spring Boot 1.3 no longer uses the [Gradle Application Plugin] so if you relied o plugin will need to be re-applied to your `build.gradle`.

Spring Boot 1.3 also uses Spring Security 4.x by default, so if you project depends on Spring Security 3.x y

```
compile 'org.springframework.security:spring-security-core:3.2.9.RELEASE'
compile 'org.springframework.security:spring-security-web:3.2.9.RELEASE'
```

Gradle `run` task no longer available by default

Because the Gradle `run` task for application startup was provided by the [Gradle Application Plugin] (see al use Gradle to start up your application, use the `bootRun` task instead, or re-apply the Application plugin i

Note: If you don't have need of the Gradle Application plugin's features, but have custom Gradle tasks or I supply your own `run` task that depends on `bootRun` in your `build.gradle`:

```
task run(dependsOn: ['bootRun'])
```

Resource annotation defaults to JSON instead of XML

The [Resource] annotation applied to domain classes defaults to XML in Grails 3.0.x, but in Grails 3.1.x and

If you use this annotation with the expecation of produces XML responses as the default you can modify th

```
import grails.rest.*

@Resource(formats=['xml', 'json'])
class MyDomainClass {}
```

This will restore the Grails 3.0.x behavior.

Geb and HTMLUnit 2.18

If you use Geb with HTMLUnit (something that is not recommended, as a more native driver such as upgrade your dependencies in `build.grade`:

```
testRuntime 'org.seleniumhq.selenium:selenium-htmlunit-driver:2.47.1'
    testRuntime 'net.sourceforge.htmlunit:htmlunit:2.18'
```

Note that there are also some changes in behavior in HTMLUnit 2.18 that may cause issues in existing test

- Expressions that evaluate the title (Example `$('title')`) now return blank and should be replaced
- If you return plain text in a response without surrounding HTML tags, these are no longer regarded a required tags.

3.3 Upgrading from Grails 2.x

This guide takes you through the fundamentals of upgrading a Grails 2.x application or plugins to Grails 3.

3.3.1 Upgrading Plugins

To upgrade a Grails 2.x plugin to Grails 3.x you need to make a number of different changes. This docum upgrade the Quartz plugin to Grails 3, each individual plugin may differ.

Step 1 - Create a new Grails 3 plugin

The first step is to create a new Grails 3 plugin using the command line:

```
$ grails create-plugin quartz
```

This will create a Grails 3 plugin in the `quartz` directory.

Step 2 - Copy sources from the original Grails 2 plugin

The next step is to copy the sources from the original Grails 2 plugin to the Grails 3 plugin:

```
# first the sources
cp -rf ../quartz-2.x/src/groovy/ src/main/groovy
cp -rf ../quartz-2.x/src/java/ src/main/groovy
cp -rf ../quartz-2.x/grails-app/ grails-app
cp -rf ../quartz-2.x/QuartzGrailsPlugin.groovy src/main/groovy/grails/plugins/qua

# then the tests
cp -rf ../quartz-2.x/test/unit/* src/test/groovy
mkdir -p src/integration-test/groovy
cp -rf ../quartz-2.x/test/integration/* src/integration-test/groovy

# then templates / other resources
cp -rf ../quartz-2.x/src/templates/ src/main/templates
```

Step 3 - Alter the plugin descriptor

You will need to add a package declaration to the plugin descriptor. In this case `QuartzGrailsPlugir`

```
// add package declaration
package grails.plugins.quartz
…
class QuartzGrailsPlugin {
 …
}
```

In addition you should remove the `version` property from the descriptor as this is now defined in `builc`

Step 4 - Update the Gradle build with required dependencies

The repositories and dependencies defined in `grails-app/conf/BuildConfig.groovy` of the or
in `build.gradle` of the new Grails 3.x plugin:

```
compile("org.quartz-scheduler:quartz:2.2.1") {
    exclude group: 'slf4j-api', module: 'c3p0'
  }
```

It is recommended to use the latest stable, Grails 3+ compatible version of plugins. (Grails 2.x plugin versi

Step 5 - Modify Package Imports

In Grails 3.x all internal APIs can be found in the `org.grails` package and public fac `org.codehaus.groovy.grails` package no longer exists.

All package declaration in sources should be modified for the new location `org.codehaus.groovy.grails.commons.GrailsApplication` is now `grails.core.Gr`

Step 5 - Migrate Plugin Specific Config to application.yml

Some plugins define a default configuration file. For example the Quart `grails-app/conf/DefaultQuartzConfig.groovy`. In Grails 3.x this default `grails-app/conf/application.yml` and it will automatically be loaded by Grails without requir

Step 6 - Update plugin exclusions

Old plugins may have a `pluginExcludes` property defined that lists the patterns for any files that shou normally used to exclude artifacts such as domain classes that are used in the plugin's integration tests. Yc application.

This property is no longer sufficient in Grails 3, and nor can you use source paths. Instead, you must speci classes. For example, imagine you have some test domain classes in the `grails-app/domain/plug` the `pluginExcludes` value to

```
def pluginExcludes = ["plugin/test/**"]
```

and then add this block to the build file:

```
jar {
    exclude "plugin/test/**"
}
```

The easiest way to ensure these patterns work effectively is to put all your non-packaged class into a separation between the main plugin classes and the rest.

Step 7 - Register ArtefactHandler Definitions

In Grails 3.x ArtefactHandler definitions written in Java need to b `src/main/resources/META-INF/grails.factories` since these need to be known at compile

If the `ArtefactHandler` is written in Groovy this step can be skipped as Grails w
`grails.factories` file during compilation.

The Quartz plugin requires the following definition to register the `ArtrefactHandler`:

```
grails.core.ArtefactHandler=grails.plugins.quartz.JobArtefactHandler
```

Step 8 - Migrate Code Generation Scripts

Many plugins previously defined command line scripts in Gant. In Grails 3.x command line scripts h
generation scripts and Gradle tasks.

If your script is doing simple code generation then for many cases a code generation script can replace an c

The `create-job` script provided by the Quartz plugin in Grails 2.x was defined in `scripts/Create.`

```
includeTargets << grailsScript("_GrailsCreateArtifacts")

target(createJob: "Creates a new Quartz scheduled job") {
    depends(checkVersion, parseArguments)

def type = "Job"
    promptForName(type: type)

for (name in argsMap.params) {
        name = purgeRedundantArtifactSuffix(name, type)
        createArtifact(name: name, suffix: type, type: type, path: "grails-app/jo
        createUnitTest(name: name, suffix: type)
    }
}

setDefaultTarget 'createJob'
```

A replacement Grails 3.x compatible script can be created using the `create-script` command:

```
$ grails create-script create-job
```

Which creates a new script called `src/main/scripts/create-job.groovy`. Using the new code

```
description("Creates a new Quartz scheduled job") {
    usage "grails create-job [JOB NAME]"
    argument name:'Job Name', description:"The name of the job"
}

model = model( args[0] )
render  template:"Job.groovy",
        destination: file( "grails-app/jobs/$model.packagePath/${model.simpleName
        model: model
```

Please refer to the documentation on [Creating Custom Scripts](#) for more information.

Migrating More Complex Scripts Using Gradle Tasks

Using the old Grails 2.x build system it was relatively common to spin up Grails inside the command line application within a code generation script created by the [create-script](#) command.

Instead a new mechanism specific to plugins exists via the [create-command](#) command. The `crea` [ApplicationCommand](#), for example the following command will execute a query:

```
import grails.dev.commands.*
import javax.sql.*
import groovy.sql.*
import org.springframework.beans.factory.annotation.*

class RunQueryCommand implements ApplicationCommand {

@Autowired
  DataSource dataSource

boolean handle(ExecutionContext ctx) {
      def sql = new Sql(dataSource)
      println sql.executeQuery("select * from foo")
      return true
  }
}
```

With this command in place once the plugin is installed into your local Maven cache you can add the plu classpath of the application's `build.gradle` file:

```
buildscript {
  …
  dependencies {
    classpath "org.grails.plugins:myplugin:0.1-SNAPSHOT"
  }
}
…
dependencies {
  runtime "org.grails.plugins:myplugin:0.1-SNAPSHOT"
}
```

Grails will automatically create a Gradle task called `runQuery` and a command named `run-query` s
command:

```
$ grails run-query
$ gradle runQuery
```

Step 8 - Delete Files that were migrated or no longer used

You should now delete and cleanup the project of any files no longer required by Grails 3.x (`Bu`
`DataSource.groovy` etc.)

3.3.2 Upgrading Applications

Upgrading applications to Grails 3.x will require that you upgrade all plugins the application uses first, he
section to first upgrade your plugins.

Step 1 - Create a New Application

Once the plugins are Grails 3.x compatible you can upgrade the application. To upgrade an application it i
using the "web" profile:

```
$ grails create-app myapp
$ cd myapp
```

Step 2 - Migrate Sources

The next step is to copy the sources from the original Grails 2 application to the Grails 3 application:

```
# first the sources
cp -rf ../old_app/src/groovy/ src/main/groovy
cp -rf ../old_app/src/java/ src/main/groovy
cp -rf ../old_app/grails-app/ grails-app

# then the tests
cp -rf ../old_app/test/unit/ src/test/groovy
mkdir -p src/integration-test/groovy
cp -rf ../old_app/test/integration/ src/integration-test/groovy
```

Step 3 - Update the Gradle build with required dependencies

The repositories and dependencies defined in `grails-app/conf/BuildConfig.groovy` of the defined in `build.gradle` of the new Grails 3.x application.

Step 4 - Modify Package Imports

In Grails 3.x all internal APIs can be found in the `org.grails` package and public fac `org.codehaus.groovy.grails` package no longer exists.

All package declaration in sources should be modified for the new location `org.codehaus.groovy.grails.commons.GrailsApplication` is now `grails.core.Gr`

Step 5 - Migrate Configuration

The configuration of the application will need to be migrated, this can normally be done by simply renam to `grails-app/conf/application.groovy` and merging the content of `grails-a` `grails-app/conf/application.groovy`.

Note however that Log4j has been replaced by `grails-app/conf/logback.groovy` for `grails-app/conf/Config.groovy` should be migrated to [logback format].

Step 6 - Migrate web.xml Modifications to Spring

If you have a modified `web.xml` template then you will need to migrate this to Spring as Grails 3.x does r have on in `src/main/webapp/WEB-INF/web.xml`).

New servlets and filters can be registered as Spring beans or with [ServletRegistrationBean] and [FilterRegist

Step 7 - Migrate Static Assets not handled by Asset Pipeline

If you have static assets in your `web-app` directory of your Grails 2.x application such as HTML files, assets such as static HTML pages and so on these should go in `src/main/resources/public`.

TLD descriptors and non public assets should go in `src/main/resources/WEB-INF`.

As noted earlier, `src/main/webapp` folder can also be used for this purpose but it is not recommended.

Step 8 - Migrate Tests

Once the package names are corrected unit tests will continue to run, however any tests that extend the need to be migrated to Spock or JUnit 4.

Integration tests will need to be annotated with the Integration annotation and should not extend GroovyTe

3.3.3 General Changes to be aware of when migrating apps

There are other miscellaneous changes between Grails 2.x and Grails 3.x that it may help to be aware of Minor changes may be required.

Domain classes

The Constraints section of a Domain Class (or other validateable object) looks like this:

```
static constraints = {
  name nullable: true, blank: false
  myField nullable: true
  another unique: true
}
```

In Grails 2.x, fields with no constraints could be declared in the Constraints block, as a method call w syntax is **no longer** supported):

```
static constraints = {
  name nullable: true, blank: false
  mySimpleProperty()                    // <- A field that has no constraints. This
3.
  anotherProperty unique: true
}
```

A different syntax has to be used in Grails 3. Either remove the field declaration from the constraints blocl to keep the field placeholder, pass an empty map argument: `[:]` instead of `()`.

Replacement code for Grails 3.x:

```
static constraints = {
  name nullable: true, blank: false
  mySimpleProperty [:]                  // <- Empty map argument instead of ()
  anotherProperty unique: true
}
```

If such declarations have not yet been changed then a log message like this emits on startup:

```
ORM Mapping Invalid: Specified config option [mySimpleProperty] does not exist fo
```

Multi-project builds (Grails 2.x inline plugins)

If your project had inline plugins in Grails 2.x, contains ASTs, or if your project is composed of severa decide to restructure your project as a Gradle **multi-project build**.

Sample multi-project structure:

```
+ example
    + example-app   <-- Main app
    + example-core  <-- Shared code plugin
    + example-ast   <-- AST transformations plugin
```

How to configure this is documented in the [Plugins](#) section under the heading 'Inline Plugins in Grails 3.0'.

Migrating from Grails 2.x to Grails 3.1+

During the progress of migrating code from Grails 2.4 to Grails 3.1+, your project (and the plugins th GORM 5 (or higher) and other newer library versions. You might also wish to familiarise yourself v [Upgrading from Grails 3.0](#).

AST Transformations

If your application contains AST transformations, please be aware that for these to be applied to your **within a plugin**. (In Grails 2.x it was possible to pre-compile AST transformations then apply them to events in `_Events.groovy`. This is no longer possible. Move your AST Transformation classes an purpose.)

There are two AST patterns on which you can base migration of your AST transformer code:

- Groovy way: Use Groovy AST transformation annotations.
- Grails way: Use Grails AST transformer annotations.

Groovy AST transformations

- Import `org.codehaus.groovy.transform.GroovyASTTransformation`
- Annotate your transformation class with `GroovyASTTransformation(phase=CompilePhas`
- A useful example app can be found here: grails3ast

Grails AST transformations

- Import `grails.compiler.ast.AstTransformer`
- Annotate your transformation class with `AstTransformer`
- Implement applicable interfaces, particularly if you are transforming Artefacts, e.g. `implemen` `AnnotatedClassInjector`
- Your Transformer class must reside in a package under **org.grails.compiler**, otherwise it will not be c
- Examples can be found in the Grails source code
- Example reference: ControllerActionTransformer.java

Deployment to containers

Grails uses Spring Boot to embed a Tomcat or Jetty instance by default. To build a war file for deploymen change to `build.gradle` (so that a container is not embedded).

If you deploy to a Tomcat 7 container then there is an additional step. Grails 3 is built against Tomcat 8 target Tomcat version in the build to 7.

There are standalone deployment options available.

Refer to the Deployment guide for further details.

Multiple datasources

If your application uses multiple datasources, then be aware that the way these are declared in `appl` (previously `DataSources.groovy`) has changed.

If there is more than one DataSource in an application there is now a `dataSources { … }` confi multiple `dataSource` declarations were used, with an underscore and suffix on the additional datasource

Please refer to the user guide section on Multiple Datasources for examples.

Improvements to dependency injection

In your Grails 2.x app you may have used Spring `@Autowired` in a few situations, such as dependency field dependency injection. For example:

```
@Autowired
org.quartz.Scheduler quartzScheduler
```

Grails now has support for dependency injection into typed fields in addition to untyped `def` fields, follo
matching the bean property name. Example:

```
GrailsApplication grailsApplication
```

You may find that `@Autowired` no longer works as it did previously in your code on artefacts or base
for these fields. Changing these to a simple typed Grails dependency following the Grails naming conventi
this.

4 Configuration

It may seem odd that in a framework that embraces "convention-over-configuration" that we tackle this t actually develop an application without doing any configuration whatsoever, as the quick start demonstra override the conventions when you need to. Later sections of the user guide will mention what configuratio The assumption is that you have at least read the first section of this chapter!

4.1 Basic Configuration

Configuration in Grails is generally split across 2 areas: build configuration and runtime configuration.

Build configuration is generally done via Gradle and the `build.gradle` file. Runtime configurat `grails-app/conf/application.yml` file.

If you prefer to use Grails 2.0-style Groovy configuration then you can create an additional `grails-` specify configuration using Groovy's ConfigSlurper syntax.

For Groovy configuration the following variables are available to the configuration script:

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"
```

If you want to read runtime configuration settings, i.e. those defined in `application.yml`, use the `gra` as a variable in controllers and tag libraries:

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

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

The `config` property of the `grailsApplication` object is an instance of the [Config] interface and p
configuration of the application.

In particular, the `getProperty` method (seen above) is useful for efficiently retrieving configuration pr
default type is String) and/or providing a default fallback value.

```
class MyController {

def hello(Recipient recipient) {
        //Retrieve Integer property 'foo.bar.max.hellos', otherwise use value of
        def max = grailsApplication.config.getProperty('foo.bar.max.hellos', Inte

//Retrieve property 'foo.bar.greeting' without specifying type (default is String
        def greeting = grailsApplication.config.getProperty('foo.bar.greeting', "

def message = (recipient.receivedHelloCount >= max) ?
          "Sorry, you've been greeted the max number of times" :  "${greeting}, $
        }

render message
    }
}
```

Notice that the `Config` instance is a merged configuration based on Spring's [PropertySource] concept
system properties and the local application configuration merging them into a single object.

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

```
import grails.core.*

class MyService {
    GrailsApplication grailsApplication

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

Finally, you can also use Spring's [Value] annotation to inject configuration values:

```
import org.springframework.beans.factory.annotation.*

class MyController {
    @Value('${foo.bar.hello}')
    String recipient

def hello() {
        render "Hello ${recipient}"
    }
}
```

In Groovy code you must use single quotes around the string for the value of the Valu interpreted as a GString not a Spring expression.

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

4.1.1 Options for the yml format Config

`application.yml` was introduced in Grails 3.0 for an alternative format for the configuration tasks.

Using system properties / command line arguments

Suppose you are using the `JDBC_CONNECTION_STRING` command line argument and you want to acce the following manner:

```
production:
    dataSource:
        url: '${JDBC_CONNECTION_STRING}'
```

Similarly system arguments can be accessed.

You will need to have this in `build.gradle` to modify the `bootRun` target if `grails run-app` is u

```
bootRun {
    systemProperties = System.properties
}
```

For testing the following will need to change the `test` task as follows

```
test {
    systemProperties = System.properties
}
```

4.1.2 Built in options

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

Runtime settings

On the runtime front, i.e. `grails-app/conf/application.yml`, there are quite a few more core se

- `grails.enable.native2ascii` - Set this to false if you do not require native2ascii conversion
- `grails.views.default.codec` - Sets the default encoding regime for GSPs - can be one o 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 type i
- `grails.mime.types` - A map of supported mime types used for Content Negotiation.
- `grails.serverURL` - A string specifying the server URL portion of absol grails.serverURL="http://my.yourportal.com". See createLink. Also used by redirects.
- `grails.views.gsp.sitemesh.preprocess` - Determines whether SiteMesh preprocessi rendering, but if you need SiteMesh to parse the generated HTML from a GSP view then disablin understand this advanced property: leave it set to true.
- `grails.reload.excludes` and `grails.reload.includes` - Configuring these directiv specific source files. Each directive takes a list of strings that are the class names for project sourc behavior or included accordingly when running the application in development with the `run-app` cc directive is configured, then only the classes in that list will be reloaded.

4.1.3 Logging

By default logging in Grails 3.0 is handled by the Logback logging framework `grails-app/conf/logback.groovy` file.

> ⚠ If you prefer XML you can replace the `logback.groovy` file with a `logback.xml` file ir

For more information on configuring logging refer to the Logback documentation on the subject.

4.1.4 GORM

Grails provides the following GORM configuration options:

- `grails.gorm.failOnError` - If set to `true`, causes the `save()` metho `grails.validation.ValidationException` if validation fails during a save. This o representing package names. If the value is a list of Strings then the failOnError behavior will only (including sub-packages). See the save method docs for more information.

For example, to enable failOnError for all domain classes:

```
grails:
    gorm:
        failOnError: true
```

and to enable failOnError for domain classes by package:

```
grails:
    gorm:
        failOnError:
            - com.companyname.somepackage
            - com.companyname.someotherpackage
```

- `grails.gorm.autoFlush` - If set to `true`, causes the merge, save and delete methods to flush t using `save(flush: true)`.

4.2 The Application Class

Every new Grails application features an `Application` class witin the the `grails-app/init` directo

The `Application` class subclasses the GrailsAutoConfiguration class and features a `static void m` application.

4.2.1 Executing the Application Class

There are several ways to execute the `Application` class, if you are using an IDE then you can simply your IDE which will start your Grails application.

This is also useful for debugging since you can debug directly from the IDE without having to connect `--debug-jvm` command from the command line.

You can also package your application into a runnable WAR file, for example:

```
$ grails package
$ java -jar build/libs/myapp-0.1.war
```

This is useful if you plan to deploy your application using a container-less approach.

4.2.2 Customizing the Application Class

There are several ways in which you can customize the `Application` class.

Customizing Scanning

By default Grails will scan all known source directories for controllers, domain class etc., however if th scan you can do so by overriding the `packageNames()` method of the `Application` class:

```
class Application extends GrailsAutoConfiguration {
    @Override
    Collection<String> packageNames() {
        super.packageNames() + ['my.additional.package']
    }

…
}
```

Registering Additional Beans

The `Application` class can also be used as a source for Spring bean definitions, simply define a me object will become a Spring bean. The name of the method is used as the bean name:

```
class Application extends GrailsAutoConfiguration {
    @Bean
    MyType myBean() {
        return new MyType()
    }

…
}
```

4.2.3 The Application LifeCycle

The `Application` class also implements the GrailsApplicationLifeCycle interface which all plugins im

This means that the `Application` class can be used to perform the same functions as a plugin. You c `doWithSpring`, `doWithApplicationContext` and so on by overriding the appropriate method:

```
class Application extends GrailsAutoConfiguration {
    @Override
    Closure doWithSpring() {
        {->
            mySpringBean(MyType)
        }
    }

…
}
```

4.3 Environments

Per Environment Configuration

Grails supports the concept of per environment configuration. The `application.yml` ar `grails-app/conf` directory can use per-environment configuration using either YAML or the synt consider the following default `application.yml` definition provided by Grails:

```
environments:
    development:
        dataSource:
            dbCreate: create-drop
            url: jdbc:h2:mem:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=
    test:
        dataSource:
            dbCreate: update
            url: jdbc:h2:mem:testDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT
    production:
        dataSource:
            dbCreate: update
            url: jdbc:h2:prodDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=FAL
        properties:
           jmxEnabled: true
           initialSize: 5
        ...
```

The above can be expressed in Groovy syntax in `application.groovy` as follows:

```
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` blc
`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`
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 Er

```
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-environmer `grails-app/init/BootStrap.groovy` file's support for per-environment execution:

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

Generic Per Environment Execution

The previous `BootStrap` example uses the `grails.util.Environment` class internally to execute your own environment specific logic:

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

4.4 The DataSource

Since Grails is built on Java technology setting up a data source requires some knowledge of JDBC 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 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 with how Grails manages its database configu either `grails-app/conf/application.groovy` or `grails-app/conf/application.ym` which includes the following settings:

- `driverClassName` - The class name of the JDBC driver
- `username` - The username used to establish a JDBC connection
- `password` - The password used to establish a JDBC connection
- `url` - The JDBC URL of the database
- `dbCreate` - Whether to auto-generate the database from the domain model - 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 the dat available dialects.
- `readOnly` - If `true` makes the DataSource read-only, which results in the connection po `Connection`
- `transactional` - If `false` leaves the DataSource's transactionManager bean outside the chaine This only applies to additional datasources.
- `persistenceInterceptor` - The default datasource is automatically wired up to the persistenc automatically unless this is set to `true`
- `properties` - Extra properties to set on the DataSource bean. See the [Tomcat Pool] documentatior [of the properties].
- `jmxExport` - If `false`, will disable registration of JMX MBeans for all DataSources. By default `jmxEnabled = true` in properties.

A typical configuration for MySQL in `application.groovy` 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 any of
Groovy will treat these as local variable definitions and they will not be processed. For examp

```
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 {
       // 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/poo
       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-confi
           // 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 the `dbCreate` property, which can take these values:

- **create** - Drops the existing schema and creates the schema on startup, dropping existing tables, index
- **create-drop** - Same as **create**, but also drops the tables when the application shuts down cleanly.
- **update** - Creates missing tables and indexes, and updates the current schema without dropping any ta many schema changes like column renames (you're left with the old column containing the existing d
- **validate** - Makes no changes to your database. Compares the configuration with the existing database
- any other value - does nothing

You can also remove the `dbCreate` setting completely, which is recommended once your schema application and database are deployed in production. Database changes are then managed through pr migration tool like Liquibase (the Database Migration plugin uses Liquibase and is tightly integrated with

4.4.1 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
        }
    }
}
```

4.4.2 Automatic Database Migration

The `dbCreate` property of the `DataSource` definition is important as it dictates what Grails shou generating the database tables from GORM classes. The options are described in the DataSource section:

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

In development mode `dbCreate` is by default set to "create-drop", but at some point in development (
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
conservative. It won't make any changes that could result in data loss, and doesn't detect renamed column
will also have the new one.

Grails supports migrations with Flyway or Liquibase using the same mechanism provided by Spring Boot.

4.4.3 Transaction-aware DataSource Proxy

The actual `dataSource` bean is wrapped in a transaction-aware proxy so you will be given the connecti
Hibernate `Session` if one is active.

If this were not the case, then retrieving a connection from the `dataSource` would be a new connectio
haven't been committed yet (assuming you have a sensible transaction isolation setting, e.g. `READ_COMMI`

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
```

4.4.4 Database Console

The H2 database console is a convenient feature of H2 that provides a web-based interface to any databas
useful to view the database you're developing against. It's especially useful when running against an in-me

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

The console is enabled by default in development mode and can be disabled or enabl `grails.dbconsole.enabled` attribute in `application.groovy`. For example, you could enabl

```
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 fr

Configuration

By default the console is configured for an H2 database which will work with the default settings if you h need to change the JDBC URL to `jdbc:h2:mem:devDB`. If you've configured an external database (Saved Settings dropdown to choose a settings template and fill in the url and username/password informati

4.4.5 Multiple Datasources

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

Configuring Additional DataSources

The default `DataSource` configuration in `grails-app/conf/application.yml` looks somethin

```
---
dataSource:
    pooled: true
    jmxExport: true
    driverClassName: org.h2.Driver
    username: sa
    password:

environments:
    development:
        dataSource:
            dbCreate: create-drop
            url: jdbc:h2:mem:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=
    test:
        dataSource:
            dbCreate: update
            url: jdbc:h2:mem:testDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT
    production:
        dataSource:
            dbCreate: update
            url: jdbc:h2:prodDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=FAL
            properties:
               jmxEnabled: true
               initialSize: 5
```

This configures a single `DataSource` with the Spring bean named `dataSource`. To configure extra `D` the top level, in an environment block, or both, just like the standard `DataSource` definition) with a cus a second `DataSource`, using MySQL in the development environment and Oracle in production:

```
---
dataSources:
    dataSource:
        pooled: true
        jmxExport: true
        driverClassName: org.h2.Driver
        username: sa
        password:
    lookup:
        dialect: org.hibernate.dialect.MySQLInnoDBDialect
        driverClassName: com.mysql.jdbc.Driver
        username: lookup
        password: secret
        url: jdbc:mysql://localhost/lookup
        dbCreate: update

environments:
    development:
        dataSources:
            dataSource:
                dbCreate: create-drop
                url: jdbc:h2:mem:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_E
    test:
        dataSources:
            dataSource:
                dbCreate: update
                url: jdbc:h2:mem:testDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_
    production:
        dataSources:
            dataSource:
                dbCreate: update
                url: jdbc:h2:prodDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT
                properties:
                   jmxEnabled: true
                   initialSize: 5
                   …
            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'`. Se block to configure a non-default `DataSource`. For example, if you want to use the `ZipCode` domain t it like this:

```
class ZipCode {
String code
static mapping = {
      datasource 'lookup'
   }
}
```

A domain class can also use two or more `DataSources`. Use the `datasources` property with a list of

```
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`

```
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 D
particular `DataSource`. For example, consider this class which uses two `DataSource`s:

```
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 v
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 t
`grails-app/conf/hibernate.cfg.xml`. To specify that an annotated class uses a non-default da
for that datasource with the file name prefixed with the datasource name.

For example if the `Book` class is in the default datasource, you would register that in `grails-app/con`

```
<?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 register that in `grails-app/conf/ds`

```
<?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 `PlatformTransacti` 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 domain Service.

Note that the datasource specified in a service has no bearing on which datasources are used for dom datasources in the domain classes themselves. It's used to declare which transaction manager to use.

What you'll see is that if you have a Foo domain class in dataSource1 and a Bar domain class in dataS
service method that saves a new Foo and a new Bar will only be transactional for Foo since they share the
instance. If you want both to be transactional you'd need to use two services and XA datasources for two-p

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 m
involves a synchronized single-phase commit of a number of resources. Because the 2PC is not used, it c
often good enough if the participants are aware of the compromises.

The basic idea is to delay the commit of all resources as late as possible in a transaction so that the only th
(not a business-processing error). Systems that rely on Best Efforts 1PC reason that infrastructure failures
risk in return for higher throughput. If business-processing services are also designed to be idempotent, the

The BE1PC implementation was added in Grails 2.3.6. . Before this change additional datasources didn't
transactions in additional datasources were basically in auto commit mode. In some cases this might
performance: on the start of each new transaction, the BE1PC transaction manager creates a new transact
additional datasource out of the BE1PC transaction manager by setting `transactional = fals`
additional dataSource. Datasources with `readOnly = true` will also be left out of the chained transact

By default, the BE1PC implementation will add all beans implementing the Spring `PlatformTransac`
transaction manager. For example, a possible `JMSTransactionManager` bean in the Grails applicatic
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 managers of c
or `readOnly = true` will be skipped and using this configuration option is not required in that case.

XA and Two-phase Commit

When the Best Efforts 1PC pattern isn't suitable for handling transactions across multiple transactional re
options available for adding XA/2PC support to Grails applications.

The Spring transactions documentation contains information about integrating the JTA/XA transaction r
case, you can configure a bean with the name `transactionManager` manually in `resources.groc`

There is also Atomikos plugin available for XA support in Grails applications.

4.5 Versioning

Detecting Versions at Runtime

You can detect the application version using Grails' support for application metadata using the GrailsA
there is an implicit grailsApplication variable that can be used:

```
def version = grailsApplication.metadata.getApplicationVersion()
```

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

```
def grailsVersion = grailsApplication.metadata.getGrailsVersion()
```

or the `GrailsUtil` class:

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

4.6 Project Documentation

Since Grails 1.2, the documentation engine that powers the creation of this documentation has been availab

The documentation engine uses a variation on the [Textile] syntax to automatically create project documenta

Creating project documentation

To use the engine you need to follow a few conventions. First, you need to create a `src/docs/guide` d
will go. Then, you need to create the source docs themselves. Each chapter should have its own gdoc file
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 section - as the above example shows.

Once you have your source files, you still need to tell the documentation engine what the structure of your `src/docs/guide/toc.yml` file that contains the structure and titles for each section. This file is structure of the user guide in tree form. For example, the above files could be represented as:

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

The format is pretty straightforward. Any section that has sub-sections is represented with the corres followed by a colon. The next line should contain `title:` plus the title of the section as seen by the en after the title. Leaf nodes, i.e. those without any sub-sections, declare their title on the same line as the sect

That's it. You can easily add, remove, and move sections within the `toc.yml` to restructure the generate section names, i.e. the gdoc filenames, should be unique since they are used for creating internal links and the documentation engine will warn you of duplicate section names.

Creating reference items

Reference items appear in the Quick Reference section of the documentation. Each reference item belo located in the `src/docs/ref` directory. For example, suppose you have defined a new controller me `Controllers` category so you would create a gdoc text file at the following location:

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

Configuring Output Properties

There are various properties you can set within your `grails-app/conf/application.gr` 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 applic

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 templates (type `java.io.File`)

One of the simplest ways to customise the look of the generated guide is to provide a value for `grails.` corresponding directory. Grails will automatically include this CSS file in the guide. You can also place a you to override the styles for the PDF version of the guide.

Generating Documentation

Add the plugin in your `build.gradle`:

```
apply plugin: "org.grails.grails-doc"
```

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

```
gradle docs
```

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

Documentation Syntax

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

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 `grails.doc.images` setting in application.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 eith
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

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 additi in `grails-app/conf/application.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}
```

4.7 Dependency Resolution

Dependency resolution is handled by the Gradle build tool, all dependencies are defined in the `build.`
more information.

5 The Command Line

Grails 3.0's command line system differs greatly from previous versions of Grails and features APIs for in
performing code generation.

When you type:

```
grails [command name]
```

Grails searches the [profile repository] based on the profile of the current application. If the profile is for a
the web profile and the base profile which it inherits from.

Since command behavior is profile specific the web profile may provide different behavior for the `run`
batch applications.

When you type the following command:

```
grails run-app
```

It results in a search for the following files:

- `PROJECT_HOME/scripts/RunApp.groovy`
- `PROFILE_REPOSITORY_PATH/profiles/web/commands/run-app.groovy` (if the web
- `PROFILE_REPOSITORY_PATH/profiles/web/commands/run-app.yml` (for YAML def

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:

```
grails [environment]* [target] [arguments]*'

| Examples:
$ grails dev run-app
$ grails create-app books

| Available Commands (type grails help 'command-name' for more info):
| Command Name                          Command Description
----------------------------------------------------------------------------------
clean                                   Cleans a Grails application's compiled so
compile                                 Compiles a Grails application
...
```

> Refer to the Command Line reference in the Quick Reference menu of the reference guid
> individual commands

non-interactive mode

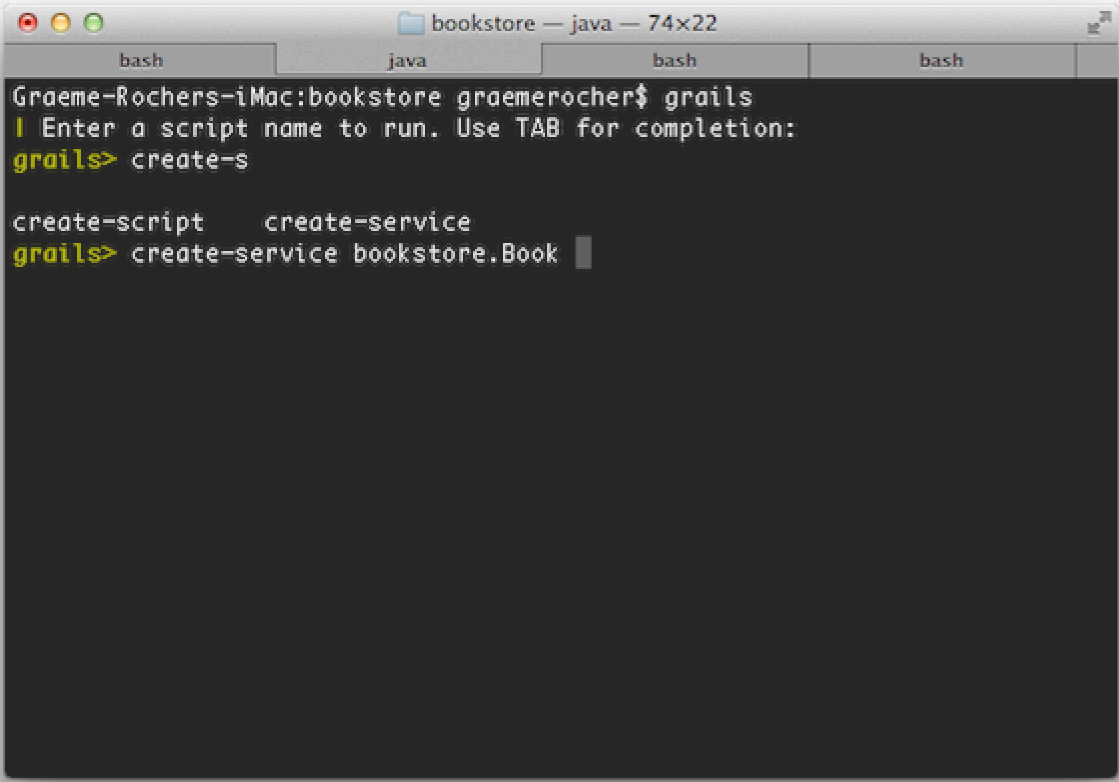
When you run a script manually and it prompts you for information, you can answer the questions and c script as part of an automated process, for example a continuous integration build server, there's no way `--non-interactive` switch to the script command to tell Grails to accept the default answer for a missing plugin.

For example:

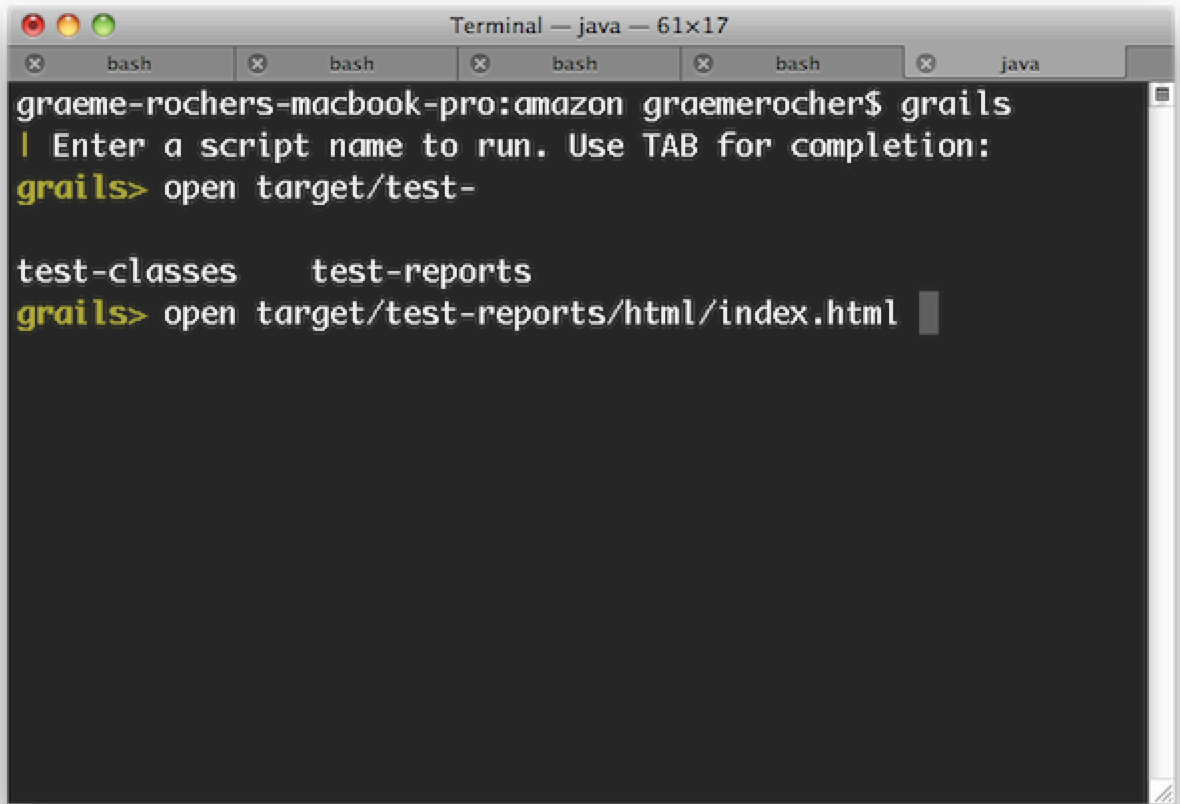
```
grails war --non-interactive
```

5.1 Interactive Mode

Interactive mode is the a feature of the Grails command line which keeps the JVM running and allows f interactive mode type 'grails' at the command line and then use TAB completion to get a list of commands:

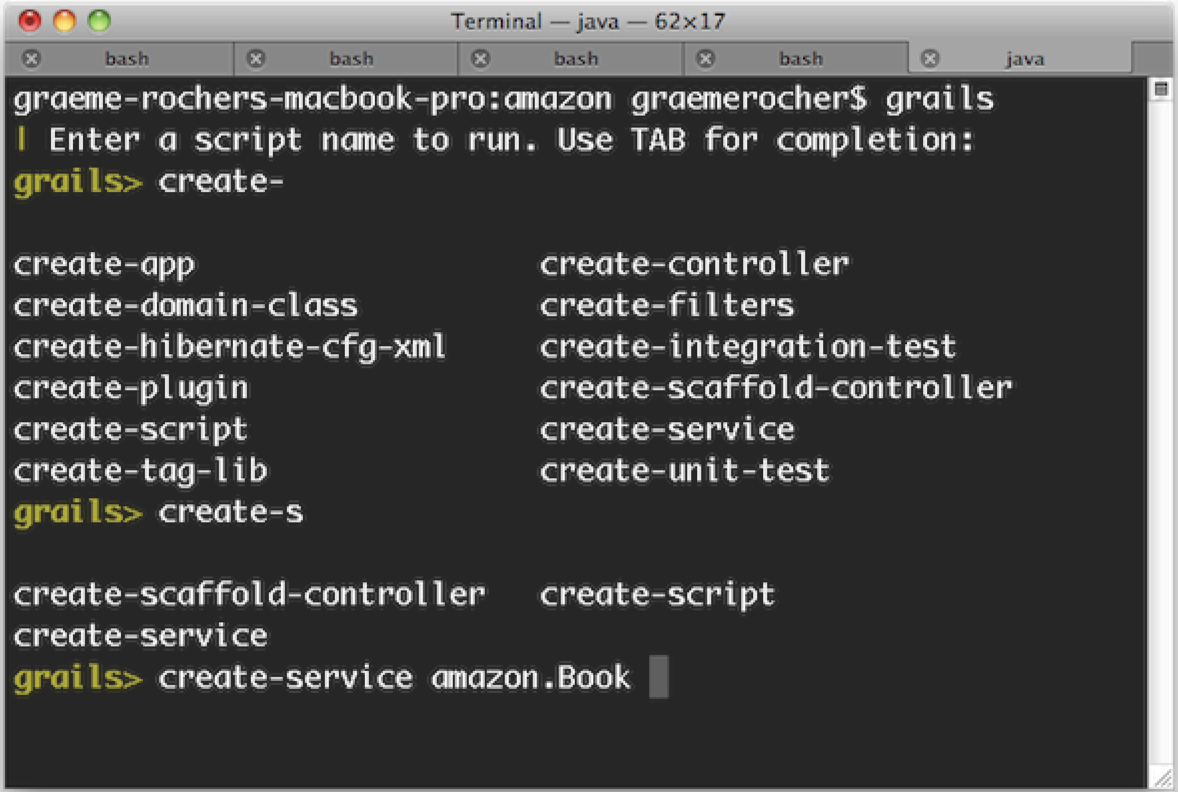


If you need to open a file whilst within interactive mode you can use the `open` command which will TAB

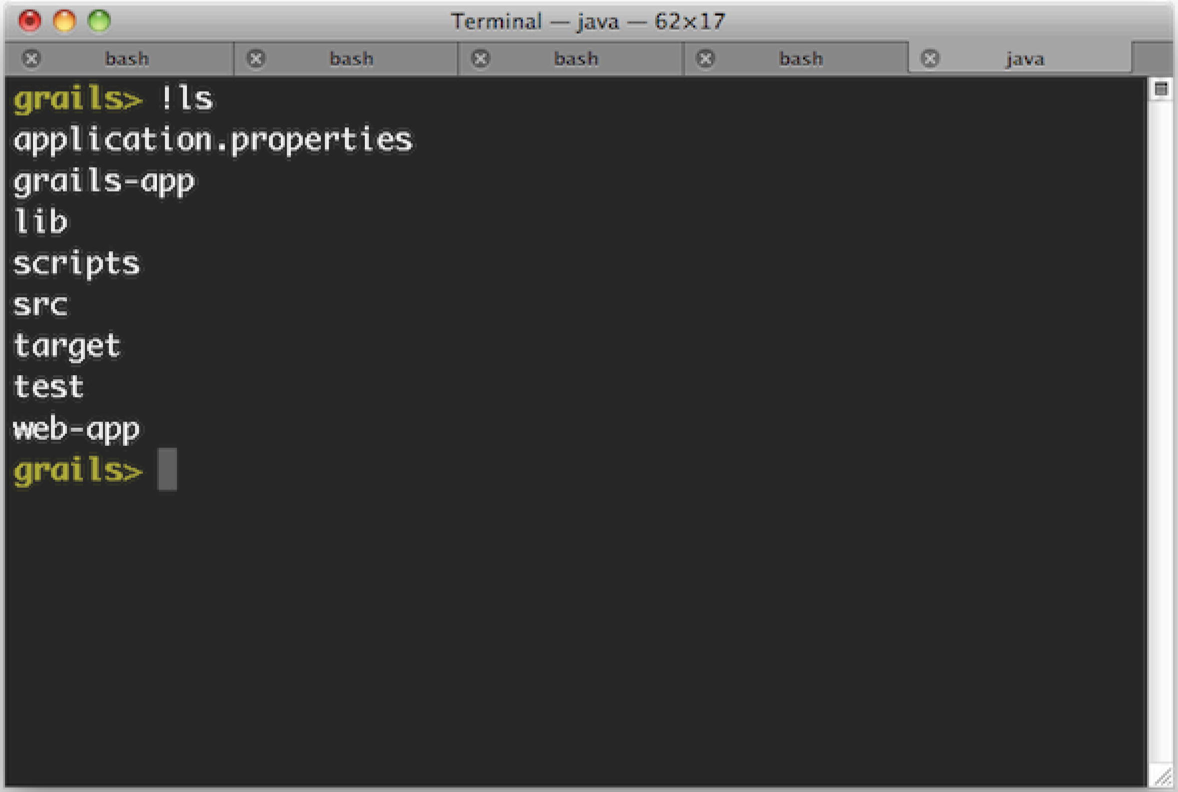


Even better, the `open` command understands the logical aliases 'test-report' and 'dep-report', which will o
respectively. In other words, to open the test report in a browser simply execute `open test-report`.
`test-report test/unit/MyTests.groovy` will open the HTML test report in your browser a
text 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
etc.

To exit interactive mode enter the `exit` command. Note that if the Grails application has been run with interactive mode console exits because the JVM will be terminated. An exception to this would be if the means the application is running in a different JVM. In that case the application will be left running after want to exit interactive mode and stop an application that is running in forked mode, use the `quit` comm application and then close interactive mode.

5.2 Creating Custom Scripts

You can create your own Command scripts by running the [create-script] command from the root of your pr create a script called `src/main/scripts/hello-world.groovy`:

```
grails create-script hello-world
```

> ⚠ In general Grails scripts should be used for scripting the Gradle based build system and co load application classes and in fact should not since Gradle is required to construct the applica

See below for an example script that prints 'Hello World':

```
description "Example description", "grails hello-world"
println "Hello World"
```

The `description` method is used to define the output seen by `grails help` and to aid users of
example of providing a description taken from the `generate-all` command:

```
description( "Generates a controller that performs CRUD operations and the associ
  usage "grails generate-all [DOMAIN CLASS]"
  flag name:'force', description:"Whether to overwrite existing files"
  argument name:'Domain Class', description:'The name of the domain class'
}
```

As you can see this description profiles usage instructions, a flag and an argument. This allows the comma

```
grails generate-all MyClass --force
```

5.3 Re-using Grails scripts

Grails ships with a lot of command line functionality out of the box that you may find useful in your own
reference guide for info on all the commands).

Any script you create can invoke another Grails script simply by invoking a method:

```
testApp()
```

The above will invoke the `test-app` command. You can also pass arguments using the method argumen

```
testApp('--debug-jvm')
```

Invoking Gradle

Instead of invoking another Grails CLI command you can invoke Gradle directory using the `gradle` prop

```
gradle.compileGroovy()
```

Invoking Ant

You can also invoke Ant tasks from scripts which can help if you need to writing code generation and auto

```
ant.mkdir(dir:"path")
```

Template Generation

Plugins and applications that need to define template generation tasks can do so using scripts. A example the `generate-all` and `generate-controllers` commands.

Every Grails script implements the TemplateRenderer interface which makes it trivial to render templates t

The following is an example of the create-script command written in Groovy:

```
description( "Creates a Grails script" ) {
  usage "grails create-script [SCRIPT NAME]"
  argument name:'Script Name', description:"The name of the script to create"
  flag name:'force', description:"Whether to overwrite existing files"
}

def scriptName = args[0]
def model = model(scriptName)
def overwrite = flag('force') ? true : false

render  template: template('artifacts/Script.groovy'),
        destination: file("src/main/scripts/${model.lowerCaseName}.groovy"),
        model: model,
        overwrite: overwrite
```

5.4 Building with Gradle

Grails 3.1 uses the [Gradle Build System](#) for build related tasks such as compilation, runnings tests and pro recommended to use Gradle 2.2 or above with Grails 3.1.

The build is defined by the `build.gradle` file which specifies the version of your project, the depende find those dependencies (amongst other things).

When you invoke the `grails` command the version of Gradle that ships with Grails 3.1 (currently 2.9) is [Tooling API](#):

```
# Equivalent to 'gradle classes'
$ grails compile
```

You can invoke Gradle directly using the `gradle` command and use your own local version of Gradle, work with Grails 3.0:

```
$ gradle assemble
```

5.4.1 Defining Dependencies with Gradle

Dependencies for your project are defined in the `dependencies` block. In general you can foll [management](#) to understand how to configure additional dependencies.

The default dependencies for the "web" profile can be seen below:

```
dependencies {
  compile 'org.springframework.boot:spring-boot-starter-logging'
  compile('org.springframework.boot:spring-boot-starter-actuator')
  compile 'org.springframework.boot:spring-boot-autoconfigure'
  compile 'org.springframework.boot:spring-boot-starter-tomcat'
  compile 'org.grails:grails-dependencies'
  compile 'org.grails:grails-web-boot'

compile 'org.grails.plugins:hibernate'
  compile 'org.grails.plugins:cache'
  compile 'org.hibernate:hibernate-ehcache'

runtime 'org.grails.plugins:asset-pipeline'
  runtime 'org.grails.plugins:scaffolding'

testCompile 'org.grails:grails-plugin-testing'
  testCompile 'org.grails.plugins:geb'

// Note: It is recommended to update to a more robust driver (Chrome, Firefox etc
  testRuntime 'org.seleniumhq.selenium:selenium-htmlunit-driver:2.44.0'

console 'org.grails:grails-console'
}
```

Note that version numbers are not present in the majority of the dependencies. This is thanks to the depe
Maven BOM that defines the default dependency versions for certain commonly used dependencies and pl

```
dependencyManagement {
    imports {
        mavenBom 'org.grails:grails-bom:' + grailsVersion
    }
    applyMavenExclusions false
}
```

5.4.2 Working with Gradle Tasks

As mentioned previously the `grails` command uses an embedded version of Gradle and certain Grails
Grails map onto their Gradle equivalents. The following table shows which Grails command invoke which

Grails Command	Gradle Task
clean	clean
compile	classes
package	assemble
run-app	bootRun
test-app	test
test-app --integration	integrationTest
war	assemble

You can invoke any of these Grails commands using their Gradle equivalents if you prefer:

```
$ gradle test
```

Note however that you will need to use a version of Gradle compatible with Grails 3.1 (Gradle 2.2 or abov
version of Gradle used by Grails you can do so with the `grails` command:

```
$ grails gradle compileGroovy
```

However, it is recommended you do this via interactive mode, as it greatly speeds up execution and pro
tasks:

```
$ grails
| Enter a command name to run. Use TAB for completion:
 grails> gradle compileGroovy
 ...
```

To find out what Gradle tasks are available without using interactive mode TAB completion you can use tl

```
gradle tasks
```

5.4.3 Grails plugins for Gradle

When you create a new project with the [create-app](#) command, a default `build.gradle` is created. The
with a set of Gradle plugins that allow Gradle to build the Grails project:

```
apply plugin:"war"
apply plugin:"org.grails.grails-web"
apply plugin:"org.grails.grails-gsp"
apply plugin:"asset-pipeline"
```

The default plugins are as follows:

- `war` - The [WAR plugin](#) changes the packaging so that Gradle creates as WAR file from you applic
  wish to create only a runnable JAR file for standalone deployment.
- `asset-pipeline` - The [asset pipeline](#) plugin enables the compilation of static assets (JavaScript, C

Many of these are built in plugins provided by Gradle or third party plugins. The Gradle plugins that Grails

- `org.grails.grails-core` - The primary Grails plugin for Gradle, included by all other plugins
- `org.grails.grails-gsp` - The Grails GSP plugin adds precompilation of GSP files for product
- `org.grails.grails-doc` - A plugin for Gradle for using Grails 2.0's documentation engine.
- `org.grails.grails-plugin` - A plugin for Gradle for building Grails plugins.
- `org.grails.grails-plugin-publish` - A plugin for publishing Grails plugins to the central
- `org.grails.grails-profile` - A plugin for use when creating Grails [Profiles](#).
- `org.grails.grails-profile-publish` - A plugin for publishing Grails profiles to the centr
- `org.grails.grails-web` - The Grails Web gradle plugin configures Gradle to understand the C

6 Application Profiles

When you create a Grails application with the create-app command by default the "web" profile is used:

```
grails create-app myapp
```

You can specify a different profile with the profile argument:

```
grails create-app myapp --profile=rest-api
```

Profiles encapsulate the project commands, templates and plugins that are designed to work for a given p
on Github, whilst the profiles themselves are published as JAR files to the Grails central repository.

To find out what profiles are available use the list-profiles command:

```
$ grails list-profiles
```

For more information on a particular profile use the profile-info command:

```
$ grails profile-info rest-api
```

Profile Repositories

By default Grails will resolve profiles from the Grails central repository. However, you can override w
repositories in the `USER_HOME/grails/settings.groovy` file.

If you want profiles to be resolved with a custom repository in addition to the Grails central repository, you

```
grails {
  profiles {
    repositories {
      myRepo {
        url = "http://foo.com/repo"
        snapshotsEnabled = true
      }
      grailsCentral {
        url = "https://repo.grails.org/grails/core"
        snapshotsEnabled = true
      }
    }
  }
}
```

> Note that Grails uses Aether to resolve profiles, as a Gradle instance is not yet available when is executed. This means that you can also define repositories and more advanced configur etc.) in your `USER_HOME/.m2/settings.xml` file if you wish.

Profile Defaults

To create an application that uses a custom profile, you must specify the full artifact.

```
$ grails create-app myapp --profile=com.mycompany.grails.profiles:myprofile:1.0.0
```

To make this process easier, you can define defaults for a given profile in the `USER_HOME/grails/se`

```
grails {
  profiles {
    myprofile {
      groupId = "com.mycompany.grails.profiles"
      version = "1.0.0"
    }
    repositories {
      …
    }
  }
}
```

With the default values specified, the command to create an application using that profile becomes:

```
$ grails create-app myapp --profile=myprofile
```

6.1 Creating Profiles

The idea behind creating a new profile is that you can setup a default set of commands and plugins organisation.

To create a new profile you can use the [create-profile] command which will create a new empty profile that

```
$ grails create-profile mycompany
```

The above command will create a new profile in the "mycompany" directory where the command is ex directory you will get a set of commands for creating profiles:

```
$ cd mycompany
$ grails
| Enter a command name to run. Use TAB for completion:
grails>

create-command      create-creator-command      create-feature      create-genera
create-gradle-command   create-template
```

The commands are as follows:

- `create-command` - creates a new command that will be available from the Grails CLI when the pr
- `create-creator-command` - creates a command available to the CLI that renders a template (E
- `create-generator-command` - creates a command available to the CLI that renders a t generate-controller)
- `create-feature` - creates a feature that can be used with this profile
- `create-gradle-command` - creates a CLI command that can invoke gradle
- `create-template` - creates a template that can be rendered by a command

To customize the dependencies for your profile you can specify additional dependencies in `profile.ym`

Below is an example `profile.yml` file:

```
features:
    defaults:
        - hibernate
        - asset-pipeline
build:
    plugins:
        - org.grails.grails-web
    excludes:
        - org.grails.grails-core
dependencies:
    compile:
        - "org.mycompany:myplugin:1.0.1"
```

With the above configuration in place you can publish the profile to your local repository with `gradle i`

```
$ gradle install
```

Your profile is now usable with the `create-app` command:

```
$ grails create-app myapp --profile mycompany
```

With the above command the application will be created with the "mycompany" profile which includes an and also includes the "hibernate" and "asset-pipeline" features (more on features later).

Note that if you customize the dependency coordinates of the profile (group, version etc.) then you ma create an application:

```
$ grails create-app myapp --profile com.mycompany:mycompany:1.0.1
```

6.2 Profile Inheritance

One profile can extend one or many different parent profiles. To define profile inheritance you can modi the profile dependences. For example typically you want to extend the `base` profile:

```
dependencies {
    runtime project(':base')
}
```

By inheriting from a parent profile you get the following benefits:

- When the create-app command is executed the parent profile's skeleton is copied first
- Dependencies and `build.gradle` is merged from the parent(s)
- The `application.yml` file is merged from the parent(s)
- CLI commands from the parent profile are inherited
- Features from the parent profile are inherited

To define the order of inheritance ensure that your dependencies are declared in the correct order. For exar

```
dependencies {
    runtime project(':plugin')
    runtime project(':web')
}
```

In the above snippet the skeleton from the "plugin" profile is copied first, followed by the "web" pr commands from the "plugin" profile, whilst if the dependency order was reversed the "plugin" profile wou

6.3 Publishing Profiles

Publishing Profiles to the Grails Central Repository

Any profile created with the create-profile command already comes configured with a `grail` `build.gradle`:

```
apply plugin: "org.grails.grails-profile-publish"
```

To publish a profile using this plugin to the Grails central repository first upload the source to Github (cl register for an account on Bintray and configure your keys as follows in the profile's `build.gradle` file

```
grailsPublish {
  user = 'YOUR USERNAME'
  key = 'YOUR KEY'
  githubSlug = 'your-repo/your-profile'
  license = 'Apache-2.0'
}
```

> The `githubSlug` argument should point to the path to your Github repository. For exampl
> at `https://github.com/foo/bar` then your `githubSlug` is `foo/bar`

With this in place you can run `gradle publishProfile` to publish your profile:

```
$ gradle publishProfile
```

The profile will be uploaded to Bintray. You can then go the the Grails profiles repository and request to
My Package" button on Bintray's interface (you must be logged in to see this).

Publishing Profiles to an Internal Repository

The aforementioned `grails-profile-publish` plugin configures Gradle's Maven Publish plugin.
you need to do is define the repository in `build.gradle`. For example:

```
publishing {
    repositories {
        maven {
            credentials {
                username "foo"
                password "bar"
            }

url "http://foo.com/repo"
        }
    }
}
```

Once configured you can publish your plugin with `gradle publish`:

```
$ gradle publish
```

6.4 Understanding Profiles

A profile is a simple directory that contains a `profile.yml` file and directories containing the "comma
profile. Example:

```
web
    * commands
        * create-controller.yml
        * run-app.groovy
        …
    * features
        * asset-pipeline
            * skeleton
            * feature.yml
    * skeleton
        * grails-app
            * controllers
            …
        * build.gradle
    * templates
        * artifacts
            * Controller.groovy
    * profile.yml
```

The above example is a snippet of structure of the 'web' profile. The `profile.yml` file is used to des
configured.

Understanding the profile.yml descriptor

The `profile.yml` can contain the following child elements.

1) repositories

A list of Maven repositories to include in the generated build. Example:

```
repositories:
    - "https://repo.grails.org/grails/core"
```

2) build.repositories

A list of Maven repositories to include in the buildscript section of the generated build. Example:

```
build:
    repositories:
        - "https://repo.grails.org/grails/core"
```

3) build.plugins

A list of Gradle plugins to configure in the generated build. Example:

```
build:
    plugins:
        - eclipse
        - idea
        - org.grails.grails-core
```

4) build.excludes

A list of Gradle plugins to exclude from being inherited from the parent profile:

```
build:
    excludes:
        - org.grails.grails-core
```

5) dependencies

A map of scopes and dependencies to configure. The `excludes` scope can be used to exclude from the pa

```
dependencies:
    excludes:
        - "org.grails:hibernate"
    build:
        - "org.grails:grails-gradle-plugin:$grailsVersion"
    compile:
        - "org.springframework.boot:spring-boot-starter-logging"
        - "org.springframework.boot:spring-boot-autoconfigure"
```

6) features.defaults

A default list of features to use if no explicit features are specified.

```
features:
    defaults:
        - hibernate
        - asset-pipeline
```

What happens when a profile is used?

When the `create-app` command runs it takes the skeleton of the parent profiles and copies the skeleton

The `build.gradle` file is generated as is result of obtaining all of the dependency information define required dependencies.

The command will also merge any `build.gradle` files defined within a profile and its parent profiles.

The `grails-app/conf/application.yml` file is also merged into a a single YAML file takin profiles.

6.5 Creating Profile Commands

A profile can define new commands that apply only to that profile using YAML or Groovy scripts. Below defined in YAML:

```
description:
    - Creates a controller
    - usage: 'create-controller [controller name]'
    - completer: org.grails.cli.interactive.completers.DomainClassCompleter
    - argument: "Controller Name"
      description: "The name of the controller"
steps:
 - command: render
   template: templates/artifacts/Controller.groovy
   destination: grails-app/controllers/artifact.package.path/artifact.nameControl
 - command: render
   template: templates/testing/Controller.groovy
   destination: src/test/groovy/artifact.package.path/artifact.nameControllerSpec
 - command: mkdir
   location: grails-app/views/artifact.propertyName
```

Commands defined in YAML must define one or many steps. Each step is a command in itself. The availa

- `render` - To render a template to a given destination (as seen in the previous example)
- `mkdir` - To make a directory specified by the `location` parameter
- `execute` - To execute a command specified by the `class` parameter. Must be a class that impleme
- `gradle` - To execute one or many Gradle tasks specified by the `tasks` parameter.

For example to invoke a Gradle task, you can define the following YAML:

```
description: Creates a WAR file for deployment to a container (like Tomcat)
minArguments: 0
usage: |
 war
steps:
 - command: gradle
   tasks:
     - war
```

If you need more flexiblity than what the declarative YAML approach provides you can create Groovy scr
from the GroovyScriptCommmand class and hence has all of the methods of that class available to it.

The following is an example of the create-script command written in Groovy:

```
description( "Creates a Grails script" ) {
  usage "grails create-script [SCRIPT NAME]"
  argument name:'Script Name', description:"The name of the script to create"
  flag name:'force', description:"Whether to overwrite existing files"
}

def scriptName = args[0]
def model = model(scriptName)
def overwrite = flag('force') ? true : false

render  template: template('artifacts/Script.groovy'),
        destination: file("src/main/scripts/${model.lowerCaseName}.groovy"),
        model: model,
        overwrite: overwrite
```

For more information on creating CLI commands see the section on [Creating custom scripts] in the Comma

6.6 Creating Profile Features

A Profile feature is a shareable set of templates and dependencies that may span multiple profiles. Typic features and child profiles that inherit from the parent and hence can use the features available from the pa

To create a feature use the `create-feature` command from the root directory of your profile:

```
$ grails create-feature myfeature
```

This will create a `myfeature/feature.yml` file that looks like the following:

```
description: Description of the feature
# customize versions here
# dependencies:
#   compile:
#     - "org.grails.plugins:myplugin2:1.0"
#
```

As a more concrete example. The following is the `feature.yml` file from the "asset-pipeline" feature:

```
description: Adds Asset Pipeline to a Grails project
build:
    plugins:
        - asset-pipeline
dependencies:
    build:
        - 'com.bertramlabs.plugins:asset-pipeline-gradle:2.5.0'
    runtime:
        - "org.grails.plugins:asset-pipeline"
```

The structure of a feature is as follows:

```
FEATURE_DIR
    * feature.yml
    * skeleton
        * grails-app
            * conf
                * application.yml
        * build.gradle
```

The contents of the skeleton get copied into the application tree, whilst the `application.yml` and `bu` counterparts in the profile by used.

With the `feature.yml` you can define additional dependencies. This allows users to create applications

```
$ grails create-app myapp --profile myprofile --features myfeature,hibernate
```

The above example will create a new application using your new feature and the "hibernate" feature.

7 Object Relational Mapping (GORM)

Domain classes are core to any business application. They hold state about business processes and hope 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 solution) and thanks to the dynamic nature of Groovy with its static and dynamic typing, along w 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 wr 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 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 y
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 datal
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 pe
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
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 Further Reading on GORM

For more information on using GORM see the dedicated documentation for the GORM project.

8 The Web Layer

8.1 Controllers

A controller handles requests and creates or prepares the response. A controller can generate the respo controller, simply create a class whose name ends with `Controller` in the `grails-app/contro` package).

The default URL Mapping configuration ensures that the first part of your controller name is mapped 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 try rur 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 con 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 prefer

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 do

If you prefer the Closure syntax or have older controller classes created in earlier versions of Grails and can set the `grails.compile.artefacts.closures.convert` property to true in `applicatio`

```
grails:
    compile:
        artefacts:
            closures:
                convert: true
```

and a compile-time AST transformation will convert your Closures to methods in the generated bytecode.

> If a controller class extends some other class which is not defined under the `grails-app` methods inherited from that class are not converted to controller actions. If the intent is to ex as controller actions the methods may be overridden in the subclass and the subclass method super class.

The Default Action

A controller has the concept of a default URI that maps to the root URI of the controller, for example /b
called when the default URI is requested is dictated by the following rules:

- If there is only one action, it's the default
- If you have an action named `index`, it's the default
- Alternatively you can set it explicitly with the `defaultAction` property:

```
static defaultAction = "list"
```

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 entire we of ServletContext
- session - The session allows associating state with a given user and typically uses cookies to associate instance of HttpSession
- request - The request object allows the storage of objects for the current request only. The request obj
- 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 operat 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 attributes are cleared. This is useful for setting a message directly before redirecting, for example:

```
def delete() {
    def b = Book.get(params.id)
    if (!b) {
        flash.message = "User not found for id ${params.id}"
        redirect(action:list)
    }
    … // remaining code
}
```

When the `list` action is requested, the `message` value will be in scope and can be used to display an inf `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 mes

Scoped Controllers

Newly created applications have the `grails.controllers.defaultScope` property set to a valu may change this value to any of the supported scopes listed below. If the property is not assigned a val scope.

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 l

```
static scope = "singleton"
```

You can define the default strategy in `application.yml` with the `grails.controllers.defau`

```
grails:
    controllers:
        defaultScope: singleton
```

> Use scoped controllers wisely. For instance, we don't recommend having any properties in 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 na 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 scaffoldin

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 o

Selecting the View

In both of the previous two examples there was no code that specified which view to render. So how does in the conventions. Grails will look for a view at the location `grails-app/views/book/show.gsp`

```
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` the view location with the `book` directory of the `grails-app/views` directory. This is convenient, b 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/displ`

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 directo specified with a relative path. The default root directory for views rendered by a namespaced contro `name>/<controller name>/`. If the view is not found in the namespaced directory then Grail 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 res 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` methoc
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 `MarkupBui`
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 param name as a request parameter, the request parameter is overridden and the controller parameter is used.

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 exa

```
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 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 Data Binding

Data binding is the act of "binding" incoming request parameters onto the properties of an object or an e
with all necessary type conversion since request parameters, which are typically delivered by a form sub
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. T
properties of the object using the keys in the Map that have values which correspond to property names
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:
'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
class:

```
class Band {
    String name
    static hasMany = [albums: Album]
    List albums
}

class Album {
    String title
    Integer numberOfTracks
}
```

```
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 a unordered, the indexes don't necessarily correspond to the order of elements in the `Set`. In the code exam `List`, the `bindingMap` could look exactly the same but 'Foxtrot' might be the first album in the `Set` or elements in a `Set` the `Map` being assigned to the `Set` must have `id` elements in it which represent th following example:

```
/*
 * 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 square brackets corresponds to the key in the `Map` being bound to. See the following code:

```
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'
```

When updating an existing `Map`, if the key specified in the binding `Map` does not exist in the `Map` being 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 request binder can work with. For example, if a request includes request parameters named `person.homeAddress.city` with values 'USA' and 'St. Louis' respectively, `params` would include

```
[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 usi

```
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 recognizes that you are trying to bind from request parameters. So if we had an incoming request like:

```
/book/save?title=The%20Stand&author=Stephen%20King
```

Then the `title` and `author` request parameters would automatically be set on the domain class. You 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 con most common case where the intent is to treat an empty form field as having the value null since there is parameter. When this behavior is not desirable the application may assign the value directly.

The mass property binding mechanism will by default automatically trim all Strings at bindi `grails.databinding.trimStrings` property to false in `grails-app/conf/application.`

```
// 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 `grails.databinding.convertEmptyStringsToNull` property to false in `grails-app/cor`

```
// the default value is true
grails.databinding.convertEmptyStringsToNull = false

// ...
```

The order of events is that the String trimming happens and then null conversion happe `convertEmptyStringsToNull` is `true`, not only will empty Strings be converted to null but also l that the `trim()` method returns an empty String.

> These forms of data binding in Grails are very convenient, but also indiscriminate. In otl non-transient, typed instance properties of the target object, including ones that you may not form in your UI doesn't submit all the properties, an attacker can still send malign da Fortunately, Grails also makes it easy to protect against such attacks - see the section titled 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 capability t 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` instanc 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

If you have a `Set` based association (the default for a `hasMany`) then the simplest way to populate ar 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 Grail 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 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 has no concept of order, so although we're referring to `books0` and `books1` it is not guaranteed that th 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 associations.

Note also that if the association you are binding to has a size of two and you refer to an element that is outs

```
<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.

You can bind existing instances of the associated type to a `List` using the same `.id` syntax as you 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 colle

> ⚠ The default limit to how large the binder will grow a collection is 256. If the data binder enc the collection be grown beyond that limit, the entry is ignored. The limit may be configured `grails.databinding.autoGrowCollectionLimit` property in `application.g`

```
// grails-app/conf/application.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 `autho` parameters belong to which type. Grails' `params` object is like a multi-dimensional hash and you ca 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 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 co command objects. Complex types are treated as command objects. See the Command Objects section of basic object types. Supported types are the 8 primitives, their corresponding type wrappers and java.lang parameters to action arguments by name:

```
class AccountingController {

// accountNumber will be initialized with the value of params.accountNumber
   // accountType will be initialized with params.accountType
   def displayInvoice(String accountNumber, int accountType) {
       // …
   }
}
```

For primitive arguments and arguments which are instances of any of the primitive type wrapper classes a request parameter value can be bound to the action argument. The type conversion happens automaticall `params.accountType` request parameter has to be converted to an `int`. If type conversion fails fo value per normal Java behavior (null for type wrapper references, false for booleans and zero for numbers) `errors` property of the defining controller.

```
/accounting/displayInvoice?accountNumber=B59786&accountType=bogusValue
```

Since "bogusValue" cannot be converted to type int, the value of accountType will be zero, the controll controller's `errors.errorCount` will be equal to 1 and the controller's `errors.getFieldE` corresponding error.

If the argument name does not match the name of the request parameter then the `@grails.web.Requ` an argument to express the name of the request parameter which should be bound to that argument:

```
import grails.web.RequestParameter

class AccountingController {

// mainAccountNumber will be initialized with the value of params.accountNumber
   // accountType will be initialized with params.accountType
   def displayInvoice(@RequestParameter('accountNumber') String mainAccountNumber
       // …
   }
}
```

Data binding and type conversion errors

Sometimes when performing data binding it is not possible to convert a particular String into a particul
error. Grails will retain type conversion errors inside the errors property of a Grails domain class. For exan

```
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 erro

```
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
from the `grails-app/i18n/messages.properties` file to use for the error. You can use a generi

```
typeMismatch.java.net.URL=The field {0} is not a valid URL
```

Or a more specific one:

```
typeMismatch.Book.publisherURL=The publisher URL you specified is not a valid URL
```

The BindUsing Annotation

The BindUsing annotation may be used to define a custom binding mechanism for a particular field in a c the field the closure value of the annotation will be invoked with 2 arguments. The first argument is the ob second argument is DataBindingSource which is the data source for the data binding. The value returned The following example would result in the upper case version of the `name` value in the source being applie

```
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 wit Here, `name` from request parameters matches with `name` from `SomeClass`.

The BindUsing annotation may be used to define a custom binding mechanism for all of the fields on a pa a class, the value assigned to the annotation should be a class which implements the BindingHelper inter time a value is bound to a property in the class that this annotation has been 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 their owi
way to do this is to write a class which implements [ValueConverter] and register an instance of that class as

```
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 nan
ValueConverter will be automatically plugged in to the data binding process.

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

beans = {

addressConverter com.myapp.converters.AddressValueConverter

// ...

}
```

```
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 B

```
import org.grails.databinding.BindingFormat

class Person {
    @BindingFormat('MMddyyyy')
    Date birthDate
}
```

A global setting may be configured in `application.groovy` to define date formats which will be usec

```
// grails-app/conf/application.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 wh marked with @BindingFormat, the @BindingFormat will take precedence over the values specified in `gra`

The default formats that are used are "yyyy-MM-dd HH:mm:ss.S", "yyyy-MM-dd'T'hh:mm:ss'Z'" and "yyy

Custom Formatted Converters

You may supply your own handler for the BindingFormat annotation by writing a class which impleme registering an instance of that class as a bean in the Spring application context. Below is an example of a tı the case of a String based on the value assigned to the BindingFormat annotation.

```
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 nan FormattedValueConverter will be automatically plugged in to the data binding process.

```
// 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 bin

```
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 value will be used as the message code to retrieve the binding format string from the `messageSource` 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
is binding to a Date object which might be constructed from several smaller pieces of information cont
like birthday_month, birthday_date and birthday_year. The structured editor would retrie
and use them to construct a Date.

The framework provides a structured editor for binding to Date objects. An application may register it
appropriate. Consider the following classes:

```
// src/groovy/databinding/Gadget.groovy
package databinding

class Gadget {
    Shape expandedShape
    Shape compressedShape
}
```

```
// 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
height and use those to calculate the area of a Shape at binding time. A structured binding editor is w

The way to register a structured editor with the data binding process is to add an instance of the org.gr
interface to the Spring application context. The easiest way to implement the TypedStructured
org.grails.databinding.converters.AbstractStructuredBindingEditor abstract class and override the getPro

```
// 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 par `expandedShape` which have a value of "struct" and if they do exist, that will trigger the use of the components of the structure need to have parameter names of the form propertyName_structuredElemen that would mean that the `compressedShape` request parameter should have a value of "struct" `compressedShape_height` parameters should have values which represent the width and the hei `expandedShape` request parameter should have a value of "struct" and the `expandedShape_widt` should have values which represent the width and the height of the expanded `Shape`.

```
// 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

```
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 automaticall [DataBindingListenerAdapter](#) class implements the `DataBindingListener` interface and provides de the interface so this class is well suited for subclassing so your listener class only needs to provide im 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 `grails.databinding.enableSpringEventAdapter` property in `grails-app/conf/appl

```
// grails-app/conf/application.groovy
grails.databinding.enableSpringEventAdapter=true

...
```

With `enableSpringEventAdapter` set to `true` instances of `BindEventListener` whicl automatically be registered with the data binder. Notice that the `MutablePropertyValues` and ' method in `BindEventListener` are Spring specific classes and are not relevant to the current data b for those arguments. The only real value passed into the `doBind` method will be the object being bound t compatibility and will be useful for a subset of scenarios. Developers are encouraged to migrate their `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 binc is not a domain class. The following will not work because the `properties` property is read only.

```
// 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 `grailsW`
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 to
persisted in the database. You can limit what properties are bound to a given domain class using the subscr

```
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 c
bindData method.

The `bindData` method allows the same data binding capability, but to arbitrary objects:

```
def p = new Person()
bindData(p, params)
```

The `bindData` method also lets you exclude certain parameters that you don't want updated:

```
def p = new Person()
bindData(p, params, [exclude: 'dateOfBirth'])
```

Or include only certain properties:

```
def p = new Person()
bindData(p, params, [include: ['firstName', 'lastName']])
```

> ⚠ Note that if an empty List is provided as a value for the `include` parameter then all fields
> they are not explicitly excluded.

The bindable constraint can be used to globally prevent data binding for certain properties.

8.1.6 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` wi

```
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"}
 ]
```

8.1.7 More on JSONBuilder

The previous section on on XML and JSON responses covered simplistic examples of rendering XML and
by Grails is the standard [XmlSlurper] found in Groovy, the JSON builder is a custom implementation speci

JSONBuilder and Grails versions

JSONBuilder behaves different depending on the version of Grails you use. For version below 1.2 the de
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
`JSONBuilder` class set the following in `application.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 direc

```
def builder = new JSONBuilder()

def result = builder.build {
    categories = ['a', 'b', 'c']
    title = "Hello JSON"
    information = {
        pages = 10
    }
}

// prints the JSON text
println result.toString()

def sw = new StringWriter()
result.render sw
```

8.1.8 Uploading Files

Programmatic File Uploads

Grails supports file uploads using Spring's [MultipartHttpServletRequest] interface. The first step for file up

```
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
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
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 a
sized files. For example, both H2 and MySQL default to a blob size of 255 bytes for `byte` properties.

It is also possible to set the contents of the file as a string by changing the type of the `myFile` property on

```
class Image {
   String myFile
}
```

8.1.9 Command Objects

Grails controllers support the concept of command objects. A command object is a class that is used in
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 a

Declaring Command Objects

Command object classes are defined just like any other class.

```
class LoginCommand implements grails.validation.Validateable {
    String username
    String password

static constraints = {
        username(blank: false, minSize: 6)
        password(blank: false, minSize: 6)
    }
}
```

In this example, the command object class implements the `Validateable` trait. The `Validateable` in [domain classes](). If the command object is defined in the same source file as the controller that i `Validateable`. It is not required that command object classes be validateable.

By default, all `Validateable` object properties are `nullable: false` which matches the behav `Validateable` that has `nullable: true` properties by default, you can specify this by defining a d

```
class AuthorSearchCommand implements grails.validation.Validateable {
    String  name
    Integer age

static boolean defaultNullable() {
        true
    }
}
```

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 paran that Grails knows what objects to create and initialize.

Before the controller action is executed Grails will automatically create an instance of the command object request parameters. If the command object class is marked with `Validateable` then the command obje

```
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 of a new instance a call will be made to the static `get` method on the domain class and the value of the Whatever is returned from that call to `get` is what will be passed into the controller action. This means corresponding record is found in the database then the value of the command object will be `null`. If a database then `null` will be passed as an argument to the controller action and an error will be added the object's type is a domain class and there is no `id` request parameter or there is an `id` request parameter a into the controller action unless the HTTP request method is "POST", in which case a new instance of tl domain class constructor. For all of the cases where the domain class instance is non-null, data binding is "POST", "PUT" or "PATCH".

Command Objects And Request Parameter Names

Normally request parameter names will be mapped directly to property names in the command object. Nes the object graph in an intuitive way. In the example below a request parameter named `name` will be boun and a request parameter named `address.city` will be bound to the `city` property of the `address` pr

```
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 tl 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 `Produc` of the problem can come up if a controller action accepts 2 command objects of the same type as shown be

```
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 will treat all parameters that begin with the controller action parameter name as belonging to the corr `product.name` request parameter will be bound to the `name` property in the `product` argument, the to the `name` property in the `buyer` argument the `seller.address.city` request parameter will be property of the `seller` argument, etc...

Command Objects and Dependency Injection

Command objects can participate in dependency injection. This is useful if your command object has sor [service](#):

```
class LoginCommand implements grails.validation.Validateable {

def loginService

String username
    String password

static constraints = {
        username validator: { val, obj ->
            obj.loginService.canLogin(obj.username, obj.password)
        }
    }
}
```

In this example the command object interacts with the `loginService` bean which is injected by name fı

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 ‹
the request based on the request content type and use the body to do data binding on the command object. !

```
// 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"}
 Name: Some Widget, Size: 42
~ $
$ curl -H "Content-Type: application/xml" -d '<widget><name>Some Other Widget</na
localhost:8080/bodybind/demo/createWidget
 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 re input stream will be empty. The controller action can either use a command object or it can parse the body referring to something like request.JSON), but cannot do both.

```
// 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

// ...

}
}
```

8.1.10 Handling Duplicate Form Submissions

Grails has built-in support for handling duplicate form submissions using the "Synchronizer Token Patterr 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 de `flash.invalidToken` variable and redirect the request back to the original page. This can then be che

```
<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 ses

8.1.11 Simple Type Converters

Type Conversion Methods

If you prefer to avoid the overhead of Data Binding and simply want to convert incoming parameters (t 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`, `shor` and safe from any parsing errors, so you don't have to perform any additional checks on the parameters.

Each of the conversion methods allows a default value to be passed as an optional second argument. The entry cannot be found in the map or if an error occurs during the conversion. Example:

```
def total = params.int('total', 42)
```

These same type conversion methods are also available on the `attrs` parameter of GSP tags.

Handling Multi Parameters

A common use case is dealing with multiple request parameters of the same name. For exam `?name=Bob&name=Judy`.

In this case dealing with one parameter and dealing with many has different semantics since Groovy's ite character. To avoid this problem the params object provides a `list` method that always returns a list:

```
for (name in params.list('name')) {
    println name
}
```

8.1.12 Declarative Controller Exception Handling

Grails controllers support a simple mechanism for declarative exception handling. If a controller declares argument type is `java.lang.Exception` or some subclass of `java.lang.Exception`, that met controller throws an exception of that type. See the following example.

```
// 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...

```
// 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 the argument type is the important part.

The exception handler methods can do anything that a controller action can do including invoking `rende`

One way to share exception handler methods across multiple controllers is to use inheritance. Exception an application could define the exception handlers in an abstract class that multiple controllers extend methods across multiple controllers is to use a trait, as shown below...

```
// src/groovy/com/demo/DatabaseExceptionHandler.groovy
package com.demo

trait DatabaseExceptionHandler {
    def handleSQLException(SQLException e) {
        // handle SQLException
    }

def handleBatchUpdateException(BatchUpdateException e) {
        // handle BatchUpdateException
    }
}
```

```
// 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 method 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 users far more flexible and intuitive.

GSPs live in the `grails-app/views` directory and are typically rendered automatically (by convention

```
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 practice is strongly discouraged. Mixing mark-up and code is a **bad** thing and most GSP page do so.

A GSP typically has a "model" which is a set of variables that are used for view rendering. The model is example consider the following controller action:

```
def show() {
    [book: Book.get(params.id)]
}
```

This action will look up a `Book` instance and create a model that contains a key called `book`. This key ca the name `book`:

```
${book.title}
```

Embedding data received from user input has the risk of making your application vulnerab (XSS) attack. Please read the documentation on [XSS prevention] for information on how to pre

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 should be familiar with.

GSP supports the usage of `<% %>` scriptlet blocks to embed Groovy code (again this is discouraged):

```
<html>
   <body>
     <% out << "Hello GSP!" %>
   </body>
</html>
```

You can also use the `<%= %>` syntax to output values:

```
<html>
   <body>
     <%="Hello GSP!" %>
   </body>
</html>
```

GSP also supports JSP-style server-side comments (which are not rendered in the HTML response) as the

```
<html>
   <body>
     <%-- This is my comment --%>
     <%="Hello GSP!" %>
   </body>
</html>
```

Embedding data received from user input has the risk of making your application vulnerab (XSS) attack. Please read the documentation on [XSS prevention] for information on how to pre

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.

The import directive lets you import classes into the page. However, it is rarely needed due to Groovy's de

```
<%@ 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 G 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 vulnerab (XSS) attack. Please read the documentation on XSS prevention for information on how to pre

8.2.2 GSP Tags

Now that the less attractive JSP heritage has been set aside, the following sections cover GSP's built-in t pages.

> The section on Tag Libraries covers how to add your own custom tag libraries.

All built-in GSP tags start with the prefix `g:`. Unlike JSP, you don't specify any tag library imports. If a t
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 new `j` 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 us

```
<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 jo

```
<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 Groov instances. Since each `Book` has a `title`, you can obtain a list of Book titles using the expression `bool` 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 specif automatically work out the link based on the URL Mappings, even if you change them! For example:

```
<g:link action="show" id="1">Book 1</g:link>

<g:link action="show" id="${currentBook.id}">${currentBook.name}</g:link>

<g:link controller="book">Book Home</g:link>

<g:link controller="book" action="list">Book List</g:link>

<g:link url="[action: 'list', controller: 'book']">Book List</g:link>

<g:link params="[sort: 'title', order: 'asc', author: currentBook.author]"
        action="list">Book List</g:link>
```

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 is version of the regular HTML form tag. The `url` attribute lets you specify which controller and action to m

```
<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. Beyo

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 c
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 usin
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 a
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 meth
response when called as methods. For example:

```
Static Resource: ${createLinkTo(dir: "images", file: "logo.jpg")}
```

This is particularly useful for using a tag within an attribute:

```
<img src="${createLinkTo(dir: 'images', file: 'logo.jpg')}" />
```

In view technologies that don't support this feature you have to nest tags within tags, which becomes m
WYSIWYG tools such as Dreamweaver that attempt to render the mark-up as it is not well-formed:

```
<img src="<g:createLinkTo dir="images" file="logo.jpg" />" />
```

Tags as method calls from Controllers and Tag Libraries

You can also invoke tags from controllers and tag libraries. Tags within the default `g:` namespac
`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 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 templat 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 mu 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 `BookController` and its views at
may want to share templates across your application.

In this case you can place them in the root views directory at grails-app/views or any subdirectory below tl
use an absolute location starting with `/` instead of a relative location. For exa
`grails-app/views/shared/_mySharedTemplate.gsp`, you would reference it as:

```
<g:render template="/shared/mySharedTemplate" />
```

You can also use this technique to reference templates in any directory from any view or controller:

```
<g:render template="/book/bookTemplate" model="[book: myBook]" />
```

The Template Namespace

Since templates are used so frequently there is template namespace, called `tmpl`, available that makes u
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 Jav
small HTML or data responses to partially update the current page instead of performing new request:

```
def bookData() {
    def b = Book.get(params.id)
    render(template:"bookTemplate", model:[book:b])
}
```

The [render](#) controller method writes directly to the response, which is the most common behaviour. To inst
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 are located in the `grails`
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--></div>
        <div class="body">
            <g:layoutBody />
        </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 th
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" pr
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 ap
delegates to. The value of the "layout" property may contain a directory structure relative to the `gra`
example:

```
class BookController {
    static layout = 'custom/customer'

def list() { … }
}
```

Views rendered from that controller would be decorated with the `grails-app/views/layouts/cus`

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 appl
delegates to.

Alternatively, you can create a layout called `grails-app/views/layouts/book/list.gsp` v
within the `BookController`.

If you have both the above mentioned layouts in place the layout specific to the action will take precedenc

If a layout may not be located using any of those conventions, the convention of last resort is to loc
`grails-app/views/layouts/application.gsp`. The name of the application default layou
`grails-app/conf/application.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
section of content. This lets you even further modularize your view structure by "decorating" your templat

Some examples of usage can be seen below:

```
<g:applyLayout name="myLayout" template="bookTemplate" collection="${books}" />

<g:applyLayout name="myLayout" url="http://www.google.com" />

<g:applyLayout name="myLayout">
The content to apply a layout to
</g:applyLayout>
```

Server-Side Includes

While the applyLayout tag is useful for applying layouts to external content, if you simply want to include
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 applications.

The basic way to include a link to a static asset in your application is to use the resource tag. This simple a

However modern applications with dependencies on multiple JavaScript and CSS libraries and framewo 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.

The asset-pipeline allows you to define your javascript or css requirements right at the top of the file and th

Take a look at the documentation for the asset-pipeline to get started.

If you do not want to use the Asset-Pipeline plugin, you can serve the static assets from directories `src/m
the latter one only gets included in WAR packaging but not in JAR packaging.

8.2.6 Sitemesh Content Blocks

Although it is useful to decorate an entire page sometimes you may find the need to decorate independe
content blocks. To get started, partition the page to be decorated using the `<content>` tag:

```
<content tag="navbar">
… draw the navbar here…
</content>

<content tag="header">
… draw the header here…
</content>

<content tag="footer">
… draw the footer here…
</content>

<content tag="body">
… draw the body here…
</content>
```

Then within the layout you can reference these components and apply individual layouts to each:

```
<html>
    <body>
        <div id="header">
            <g:applyLayout name="headerLayout">
                <g:pageProperty name="page.header" />
            </g:applyLayout>
        </div>
        <div id="nav">
            <g:applyLayout name="navLayout">
                <g:pageProperty name="page.navbar" />
            </g:applyLayout>
        </div>
        <div id="body">
            <g:applyLayout name="bodyLayout">
                <g:pageProperty name="page.body" />
            </g:applyLayout>
        </div>
        <div id="footer">
            <g:applyLayout name="footerLayout">
                <g:pageProperty name="page.footer" />
            </g:applyLayout>
        </div>
    </body>
</html>
```

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 any your whole application. If all you want to do is fix a typo on a page, or change an image link, it can seem requirements, Grails does have a solution: the `grails.gsp.view.dir` configuration setting.

How does this work? The first step is to decide where the GSP files should go. Let's say `/var/www/grails/my-app` directory. We add these two lines to `grails-app/conf/applicat`

```
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 sett but they won't be reflected in the running application until you restart. The second line tells Grails where to

> The trailing slash on the `grails.gsp.view.dir` value is important! Without it, Grails wi directory.

Setting "grails.gsp.view.dir" is optional. If it's not specified, you can update files directly to the applicatio the application server, these files might get overwritten when the server is restarted. Most application serve recommended in this case.

With those settings in place, all you need to do is copy the views from your web application to the extern 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-ap` `/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 of permgen space" errors unless you restart the server. So this technique is not recommended for frequent

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 without chang
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 of roundings usually cause a 1000ms difference in lastmodified 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 c show the source code of included templates. This only works in development mode
- The saving of all generated source code can be activated by setting the property "grails.views.gsp.kee It must point to a directory that exists and is writable.
- During "grails war" gsp pre-compilation, the generated source code is stored in gr ~/.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 templa be hard to find out what GSP template actually renders the html seen in the result. The debug templates comments contain debug information about gsp templates used to render the page.

Usage is simple: append "?debugTemplates" or "&debugTemplates" to the url and view the source of tl 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' completely reloadable at runtime.

Quite simply, to create a tag library create a Groovy class that ends with the convention `TagLib` an 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 retu

```
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
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 shou your tag closures with `@attr` descriptions. Since taglibs use Groovy code it can be difficu attributes.

For example:

```
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.

```
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 outpu style tag:

```
def dateFormat = { attrs, body ->
    out << new java.text.SimpleDateFormat(attrs.format).format(attrs.date)
}
```

The above uses Java's `SimpleDateFormat` class to format a date and then write it to the response. The

```
<g:dateFormat format="dd-MM-yyyy" date="${new Date()}" />
```

With simple tags sometimes you need to write HTML mark-up to the response. One approach would be to

```
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 rend

```
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 security tags:

```
def isAdmin = { attrs, body ->
    def user = attrs.user
    if (user && checkUserPrivs(user)) {
        out << body()
    }
}
```

The tag above checks if the user is an administrator and only outputs the body content if he/she has the cor

```
<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 `t
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 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 l
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 assum
and not referring to the variable itself.

Now we can change the usage of the tag as follows:

```
<g:repeat times="3" var="j">
<p>Repeat this 3 times! Current repeat = ${j}</p>
</g:repeat>
```

Notice how we use the `var` attribute to define the name of the variable `j` and then we are able to reference

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
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 u

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. `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 configured with the `grails` comma separated String value. Spring's PathMatchingResourcePatternResolver is used to resolve the patte

For example you could scan for all available tld files by adding this to `application.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, `build.gradle`:

```
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 to a `ja` a buffer that "captures" the output of the taglib call. This buffer is the return value of a tag library call whe

If the tag is listed in the library's static `returnObjectForTags` array, then its return value will writt The return value of the tag lib closure will be returned as-is if it's used as a function in GSP expressions or

If the tag is not included in the returnObjectForTags array, then its return value will be discarded. Using not supported.

Example:

```
class ObjectReturningTagLib {
    static namespace = "cms"
    static returnObjectForTags = ['content']

def content = { attrs, body ->
        CmsContent.findByCode(attrs.code)?.content
    }
}
```

Given this example cms.content(code:'something') call in another taglib or GSP expression would return caller without wrapping the return value in a buffer. It might be worth doing so also because of perform wrap the tag return value in an output buffer in such cases.

8.4 URL Mappings

Throughout the documentation so far the convention used for URLs has been the default of `/controll` not hard wired into Grails and is in fact controlled by a URL Mappings class located at `grails-app/cc`

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 parameters

```
"/product"(controller: "product", action: "list")
```

In this case we've mapped the URL `/product` to the `list` action of the `ProductController`. C
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")
}
```

You can also create nested `group` url mappings:

```
group "/store", {
    group "/product", {
        "/$id"(controller:"product")
    }
}
```

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. Th

```
"/books"(resources:'book')
```

You define a base URI and the name of the controller to map to using the `resources` parameter. The abo

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 are not sure which mapping will be generated for your case just run the command `url-mapping`
you a really neat report for all the url mappings.

If you wish to include or exclude any of the generated URL mappings you can do so with the `include`
name of the Grails action to include or exclude:

```
"/books"(resources:'book', excludes:['delete', 'update'])
or
"/books"(resources:'book', includes:['index', 'show'])
```

Explicit REST Mappings

As of Grails 3.1, if you prefer not to rely on the a `resources` mapping to define your mappings then yo
method name (in lower case) to indicate the HTTP method it applies to. The following URL mapping:

```
"/books"(resources:'book')
```

Is equivalent to:

```
get "/books"(controller:"book", action:"index")
get "/books/create"(controller:"book", action:"create")
post "/books"(controller:"book", action:"save")
get "/books/$id"(controller:"book", action:"show")
get "/books/$id/edit"(controller:"book", action:"edit")
put "/books/$id"(controller:"book", action:"update")
delete "/books/$id"(controller:"book", action:"delete")
```

Notice how the HTTP method name is prefixed prior to each URL mapping definition.

Single resources

A single resource is a resource for which there is only one (possibly per user) in the system. You can 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

To map a URI directly below a resource then use a collection:

```
"/books"(resources: "book") {
    collection {
        "/publisher"(controller:"publisher")
    }
}
```

This will result in the following URL being available (without the ID):

HTTP Method	URL	Grails Action
GET	/books/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

```
<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 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"
```

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"). Th
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 mappi
matches an incoming request, a redirect is initiated with information provided by the mapping.

When a URL mapping specifies a redirect the mapping must either supply a String representing a URI to
the target of the redirect. That Map is structured just like the Map that may be passed as an argument to the

```
"/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 mappin
between each slash, '/'. A concrete token is one which is well defined such as as `/product`. However, i
value of a particular token will be until runtime. In this case you can use variable placeholders within the U

```
static mappings = {
  "/product/$id"(controller: "product")
}
```

In this case by embedding a $id variable as the second token Grails will automatically map the second t
object) called `id`. For example given the URL `/product/MacBook`, the following code will render "M

```
class ProductController {
     def index() { render params.id }
}
```

You can of course construct more complex examples of mappings. For example the traditional blog URL f

```
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

Dynamic Controller and Action Names

Variables can also be used to dynamically construct the controller and action name. In fact the default Grai

```
static mappings = {
    "/$controller/$action?/$id?"()
}
```

Here the name of the controller, action and id are implicitly obtained from the variables `controller`, ac

You can also resolve the controller name and action name to execute dynamically using a closure:

```
static mappings = {
    "/$controller" {
        action = { params.goHere }
    }
}
```

Optional Variables

Another characteristic of the default mapping is the ability to append a ? at the end of a variable to mak
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 th

```
/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 in th

```
"/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 variabl by assigning a block to the variable name:

```
"/holiday/win" {
     id = { params.id }
     isEligible = { session.user != null } // must be logged in
}
```

In the above case the code within the blocks is resolved when the URL is actually matched and hence can l

8.4.5 Mapping to Views

You can resolve a URL to a view without a controller or action involved. For example to map `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 th
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 `illegalAr` `NullPointerException` will be handled by the `nullPointer` action, and a `MyException` will l Other exceptions will be handled by the catch-all rule and use the `/errors/serverError` view.

You can access the exception from your custom error handing view or controller action using the request's

```
class ErrorController {
    def handleError() {
        def exception = request.exception
        // perform desired processing to handle the exception
    }
}
```

If your error-handling controller action throws an exception as well, you'll end up with a `Sta`

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 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")
}
```

Note that if you specify a HTTP method other than GET in your URL mapping, you also have to spec passing the `method` argument to `g:link` or `g:createLink` to get a link of the desired format.

8.4.8 Mapping Wildcards

Grails' URL mappings mechanism also supports wildcard mappings. For example consider the following n

```
static mappings = {
    "/images/*.jpg"(controller: "image")
}
```

This mapping will match all paths to images such as `/image/logo.jpg`. Of course you can achieve the

```
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.jp` 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 ma `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 tag 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 an earlier section:

```
static mappings = {
   "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

If you use the link tag as follows:

```
<g:link controller="blog" action="show"
        params="[blog:'fred', year:2007]">
    My Blog
</g:link>

<g:link controller="blog" action="show"
        params="[blog:'fred', year:2007, month:10]">
    My Blog - October 2007 Posts
</g:link>
```

Grails will automatically re-write the URL in the correct format:

```
<a href="/fred/2007">My Blog</a>
<a href="/fred/2007/10">My Blog - October 2007 Posts</a>
```

8.4.10 Applying Constraints

URL Mappings also support Grails' unified validation constraints mechanism, which lets you further "cor
we revisit the blog sample code from earlier, the mapping currently looks like this:

```
static mappings = {
   "/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")
}
```

This allows URLs such as:

```
/graemerocher/2007/01/10/my_funky_blog_entry
```

However, it would also allow:

```
/graemerocher/not_a_year/not_a_month/not_a_day/my_funky_blog_entry
```

This is problematic as it forces you to do some clever parsing in the controller code. Luckily, URL Map 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 patt

8.4.11 Named URL Mappings

URL Mappings also support named mappings, that is mappings which have a name associated with the mapping when links are generated.

The syntax for defining a named mapping is as follows:

```
static mappings = {
   name <mapping name>: <url pattern> {
      // …
   }
}
```

For example:

```
static mappings = {
    name personList: "/showPeople" {
        controller = 'person'
        action = 'list'
    }
    name accountDetails: "/details/$acctNumber" {
        controller = 'product'
        action = 'accountDetails'
    }
}
```

The mapping may be referenced in a link tag in a GSP.

```
<g:link mapping="personList">List People</g:link>
```

That would result in:

```
<a href="/showPeople">List People</a>
```

Parameters may be specified using the params attribute.

```
<g:link mapping="accountDetails" params="[acctNumber:'8675309']">
    Show Account
</g:link>
```

That would result in:

```
<a href="/details/8675309">Show Account</a>
```

Alternatively you may reference a named mapping using the link namespace.

```
<link:personList>List People</link:personList>
```

That would result in:

```
<a href="/showPeople">List People</a>
```

The link namespace approach allows parameters to be specified as attributes.

```
<link:accountDetails acctNumber="8675309">Show Account</link:accountDetails>
```

That would result in:

```
<a href="/details/8675309">Show Account</a>
```

To specify attributes that should be applied to the generated `href`, specify a `Map` value to the `attrs` attri
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:

```
<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 a controller named `MathHelperController` would be something like `/mathHelper/addNumbe` pattern and provides an implementation which replaces the camel case convention with a hyphenate `/math-helper/add-numbers`. To enable hyphenated URLs assign a value of "hyphenated" to the `grails-app/conf/application.groovy`.

```
// grails-app/conf/application.groovy

grails.web.url.converter = 'hyphenated'
```

Arbitrary strategies may be plugged in by providing a class which implements the UrlConverter interface a application context with the bean name of `grails.web.UrlConverter.BEAN_NAME`. If Grails find used as the default converter and there is no need to assign a value to the `grails.web.url.convert`

```
// 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 controllers mu
a namespace for a controller is to define a static property named `namespace` in the controller and ass
namespace.

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

class AdminController {

static namespace = 'reports'

// …
}
```

```
// 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` v

```
// grails-app/controllers/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 mat
application provides several controllers with the same name in different packages, at most 1 of them may
there are multiple controllers with the same name that do not define a `namespace` property, the framev
them for forward or reverse mapping resolutions.

It is allowed for an application to use a plugin which provides a controller with the same name as a contr
of the controllers to define a `namespace` property as long as the controllers are in separate package
controller named `com.accounting.ReportingController` and the application may use a
`com.humanresources.ReportingController`. The only issue with that is the URL mapping fc
be explicit in specifying that the mapping applies to the `ReportingController` which is provided by

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 `Repor`
application. A request to `/humanResourceReports` will be handled by the `Reporting`
`humanResources` plugin.

There could be any number of `ReportingController` controllers provided by any number of plug
`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 contr
provided by the application and/or plugins. If the `humanResources` plugin provides a `Repor`
`ReportingController` available at runtime, the following mapping would 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 Interceptors

Grails provides standalone Interceptors using the create-interceptor command:

```
$ grails create-interceptor MyInterceptor
```

The above command will create an Interceptor in the `grails-app/controllers` directory with the f

```
class MyInterceptor {

boolean before() { true }

boolean after() { true }

void afterView() {
    // no-op
  }

}
```

Interceptors vs Filters

In versions of Grails prior to Grails 3.0, Grails supported the notion of filters. These are still supported deprecated.

The new interceptors concept in Grails 3.0 is superior in a number of ways, most significantly inte annotation to optimize performance (something which is often critical as interceptors can be executed for e

8.5.1 Defining Interceptors

By default interceptors will match the controllers with the same name. For example if you have an in requests to the actions of the `BookController` will trigger the interceptor.

An `Interceptor` implements the Interceptor trait and provides 3 methods that can be used to intercept r

```
/**
 * Executed before a matched action
 *
 * @return Whether the action should continue and execute
 */
boolean before() { true }

/**
 * Executed after the action executes but prior to view rendering
 *
 * @return True if view rendering should continue, false otherwise
 */
boolean after() { true }

/**
 * Executed after view rendering completes
 */
void afterView() {}
```

As described above the `before` method is executed prior to an action and can cancel the execution of the

The `after` method is executed after an action executes and can halt view rendering if it returns false. T model using the `view` and `model` properties respectively:

```
boolean after() {
  model.foo = "bar" // add a new model attribute called 'foo'
  view = 'alternate' // render a different view called 'alternate'
  true
}
```

The `afterView` method is executed after view rendering completes. If an exception occurs, the excepti of the Interceptor trait.

8.5.2 Matching Requests with Inteceptors

As mention in the previous section, by default an interceptor will match only requests to the associate configure the interceptor to match any request using the `match` or `matchAll` methods defined in the Inte

The matching methods return a Matcher instance which can be used to configure how the interceptor matcl

For example the following interceptor will match all requests except those to the `login` controller:

```
class AuthInterceptor {
  AuthInterceptor() {
    matchAll()
    .excludes(controller:"login")
  }

boolean before() {
    // perform authentication
  }
}
```

You can also perform matching using named argument:

```
class LoggingInterceptor {
  LoggingInterceptor() {
    match(controller:"book", action:"show") // using strings
    match(controller: ~/(author|publisher)/) // using regex
  }

boolean before() {
    …
  }
}
```

You can use any number of matchers defined in your interceptor. They will be executed in the order in w
above interceptor will match for all of the following:

- when the `show` action of `BookController` is called
- when `AuthorController` or `PublisherController` is called

All named arguments except for `uri` accept either a String or a Regex expression. The `uri` argument
Spring's AntPathMatcher. The possible named arguments are:

- `namespace` - The namespace of the controller
- `controller` - The name of the controller
- `action` - The name of the action
- `method` - The HTTP method
- `uri` - The URI of the request. If this argument is used then all other arguments will be ignored and on

8.5.3 Ordering Interceptor Execution

Interceptors can be ordered by defining an `order` property that defines a priority.

For example:

```
class AuthInterceptor {
int order = HIGHEST_PRECEDENCE
…
}
```

The default value of the `order` property is 0.

The values `HIGHEST_PRECEDENCE` and `LOWEST_PRECEDENCE` can be used to define filters that shou

Note that if you write an interceptor that is to be used by others it is better increment or de `LOWEST_PRECEDENCE` to allow other interceptors to be inserted before or after the interceptor you are a

```
int order = HIGHEST_PRECEDENCE + 50
// or
int order = LOWEST_PRECEDENCE - 50
```

To find out the computed order of interceptors you can add a debug logger to `logback.groovy` as follo

```
logger 'grails.artefact.Interceptor', DEBUG, ['STDOUT'], false
```

You can override any interceptors default order by using bean override configuration in `grails-app/c`

```
beans:
  authInterceptor:
    order: 50
```

Or in `grails-app/conf/application.groovy`:

```
beans {
  authInterceptor {
    order = 50
  }
}
```

Thus giving you complete control over interceptor execution order.

8.6 Content Negotiation

Grails has built in support for Content negotiation using either the HTTP `Accept` header, an explicit mapped URI.

Configuring Mime Types

Before you can start dealing with content negotiation you need to tell Grails what content types you wish with a number of different content types within `grails-app/conf/application.yml` using the `g`

```
grails:
    mime:
        types:
            all: '*/*'
            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 setting can also be done in `grails-app/conf/application.groovy` as shown below:

```
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/: 'xml'. You can add your own types by simply adding new entries into the map. The first one is the default f

Content Negotiation using the format Request Parameter

Let's say a controller action can return a resource in a variety of formats: HTML, XML, and JSON. What 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 `xml` .

You can also define this parameter in the URL Mappings definition:

```
"/book/list"(controller:"book", action:"list") {
    format = "xml"
}
```

You could code your controller action to return XML based on this property, but you can also make u 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 requested c
then Grails will execute the `html()` call only. Each 'block' can either be a map model for the correspondi
example) or a closure. The closure can contain any standard action code, for example it can return a model

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 s Accept header - see below), then the first block of `withFormat()` is executed when there isn't a (wi

You should not add an explicit "all" block. In this example, a format of "all" will trigger the `html` han 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 returr 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 typically:

```
*/*
```

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 form That's why Grails ignores the accept header by default for browsers. However, non-browser clients are ty can send accept headers such as

```
application/json
```

As mentioned the default configuration in Grails is to ignore the accept header for browsers. `grails.mime.disable.accept.header.userAgents`, which is configured to detect the majo headers. This allows Grails' content negotiation to continue to work for non-browser clients:

```
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 `withFormat()` method in just the same way as when the `format` URL parameter is set (although the U

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 b request and will set the HTML format - note that this is different from how the other conten 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 for and is typically used to detect if the incoming request can be parsed into XML or JSON, whilst the r parameter or ACCEPT header to attempt to deliver an appropriate response to the client.

The withFormat available on controllers deals specifically with the response format. If you wish to add l can do so using a separate `withFormat` method available on the request:

```
request.withFormat {
    xml {
        // read XML
    }
    json {
        // read JSON
    }
}
```

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 via th 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 manipula

```
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 Traits

Overview

Grails provides a number of traits which provide access to properties and behavior that may be accessed f
Groovy classes which are part of a Grails project. Many of these traits are automatically added to Grails
example) and are easy to add to other classes.

9.1 Traits Provided by Grails

Grails artefacts are automatically augmented with certain traits at compile time.

Domain Class Traits

- grails.artefact.DomainClass
- grails.web.databinding.WebDataBinding
- org.grails.datastore.gorm.GormEntity
- org.grails.datastore.gorm.GormValidateable

Controller Traits

- grails.artefact.gsp.TagLibraryInvoker
- grails.artefact.AsyncController
- grails.artefact.controller.RestResponder
- grails.artefact.Controller

Interceptor Trait

- grails.artefact.Interceptor

Tag Library Trait

- grails.artefact.TagLibrary

Service Trait

- grails.artefact.Service

Below is a list of other traits provided by the framework. The javadocs provide more detail about methods

Trait	Brief Description
grails.web.api.WebAttributes	Common Web Attributes
grails.web.api.ServletAttributes	Servlet API Attributes
grails.web.databinding.DataBinder	Data Binding API
grails.artefact.controller.support.RequestForwarder	Request Forwarding API
grails.artefact.controller.support.ResponseRedirector	Response Redirecting API
grails.artefact.controller.support.ResponseRenderer	Response Rendering API
grails.validation.Validateable	Validation API

9.1.1 WebAttributes Trait Example

WebAttributes is one of the traits provided by the framework. Any Groovy class may implement this tr provided by the trait.

```
// src/main/groovy/demo/Helper.groovy
package demo

import grails.web.api.WebAttributes

class Helper implements WebAttributes {

List<String> getControllerNames() {
        // There is no need to pass grailsApplication as an argument
        // or otherwise inject the grailsApplication property.  The
        // WebAttributes trait provides access to grailsApplication.
        grailsApplication.getArtefacts('Controller')*.name
    }
}
```

The traits are compatible with static compilation...

```
// src/main/groovy/demo/Helper.groovy
package demo

import grails.web.api.WebAttributes
import groovy.transform.CompileStatic

@CompileStatic
class Helper implements WebAttributes {

List<String> getControllerNames() {
        // There is no need to pass grailsApplication as an argument
        // or otherwise inject the grailsApplication property.  The
        // WebAttributes trait provides access to grailsApplication.
        grailsApplication.getArtefacts('Controller')*.name
    }
}
```

10 Web Services

Web Services are all about providing a web API onto your web application and are typically implemented

10.1 REST

REST is not really a technology in itself, but more an architectural pattern. REST is very simple and communication medium, combined with URL patterns that are "representational" of the underlying system and DELETE.

Each HTTP method maps to an action type. For example GET for retrieving data, POST for creating data,

Grails includes flexible features that make it easy to create RESTful APIs. Creating a RESTful reso demonstrated in the next section.

10.1.1 Domain classes as REST resources

The easiest way to create a RESTful API in Grails is to expose a domain class as a REST r `grails.rest.Resource` transformation to any domain class:

```
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 automati XML or JSON formats. The transformation will automatically register the necessary RESTful U `BookController`.

You can try it out by adding some test data to `BootStrap.groovy`:

```
def init = { servletContext ->

new Book(title:"The Stand").save()
        new Book(title:"The Shining").save()
    }
```

And then hitting the URL http://localhost:8080/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/books/1.json` you will get a JSON respons

```
{"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 `formats`

```
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 th names of formats are defined in the `grails.mime.types` setting of `application.groovy`:

```
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 tool:

```
$ curl -i -H "Accept: application/json" localhost:8080/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
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
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/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. 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.

10.1.2 Mapping to REST resources

If you prefer to keep the declaration of the URL mapping in your `UrlMappings.groovy` file th `Resource` transformation and adding the following line to `UrlMappings.groovy` will suffice:

```
"/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

10.1.3 Linking to REST resources from GSP pages

The `link` tag offers an easy way to link to any domain class resource:

```
<g:link resource="${book}">My Link</g:link>
```

However, currently you cannot use g:link to link to the DELETE action and most browsers do not support

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 co restful resource mappings to create powerful web interfaces. To make a link fire this type of event, pe `data-method` attribute and issue a form submit via javascript.

10.1.4 Versioning REST resources

A common requirement with a REST API is to expose different versions at the same time. There are a few

Versioning using the URI

A common approach is to use the URI to version APIs (although this approach is discouraged in favour of 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 example y

```
"/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/books
```

Versioning using Hypermedia / Mime Types

Another approach to versioning is to use Mime Type definitions to declare the version of your custom med Engine of Application State" for more information about Hypermedia concepts). For example, in `appl` Mime Type for your resource that includes a version parameter (the 'v' parameter):

```
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 ' established then the first entry in the map is used for the response. If you have your new Mim will always try and send back your new Mime Type if the requested Mime Type cannot be est

Then override the renderer (see the section on "Customizing Response Rendering" for more information Mime Type in `grails-app/conf/spring/resourses.groovy`:

```
import grails.rest.render.json.*
import grails.web.mime.*

beans = {
    bookRendererV1(JsonRenderer, myapp.v1.Book, new MimeType("application/vnd.boo
    bookRendererV2(JsonRenderer, myapp.v2.Book, new MimeType("application/vnd.boo
}
```

Then update the list of acceptable response formats in your controller:

```
class BookController extends RestfulController {
    static responseFormats = ['json', 'xml', 'book', 'bookv2']

// …
}
```

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://lo
```

10.1.5 Implementing REST controllers

The `Resource` transformation is a quick way to get started, but typically you'll want to customize the c extend the API to include additional actions.

10.1.5.1 Extending the RestfulController super class

The easiest way to get started doing so is to create a new controller for your resource that extends the `g` 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 nar
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 a
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 `queryForResc` 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` method which returns a value which will be used as the source for data binding. For example, the update ac

```
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 is body then the body will be parsed and its contents will be used to do the data binding, otherwise the binding. Subclasses of RestfulController may override the `getObjectToBind()` method and return an [Map](#) or a [DataBindingSource](#). For most use cases binding the request is appropriate but the `getObjec` behavior where desired.

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 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
    }
}
```

10.1.5.2 Implementing REST Controllers Step by Step

If you don't want to take advantage of the features provided by the `RestfulController` super cl
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 inte
> They are there in order to render appropriate HTML forms to create and edit a resource. If
> can be discarded.

The key to implementing REST actions is the [respond] method introduced in Grails 2.3. The `responc` 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 support pagination via some user interface.

The `respond` method will, using [Content Negotiation], attempt to reply with the most appropriate respon (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 wo

```
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 `respond` method will attempt to lookup an appropriate `grails.rest.render.Renderer` instanc This is done by inspecting the `grails.rest.render.RendererRegistry`.

By default there are already renderers configured for JSON and XML, to find out how to register a cu Response Rendering".

Implementing the 'show' action

The `show` action, which is used to display and individual resource by id, can be implemented in on signature):

```
def show(Book book) {
    respond book
}
```

By specifying the domain instance as a parameter to the action Grails will automatically attempt to lookup the request. If the domain instance doesn't exist, then `null` will be passed into the action. The `respond` otherwise once again it will attempt to render an appropriate response. If the format is HTML then an a action is functionally equivalent to the above action:

```
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 accepts `Transactional` with the `grails.transaction.Transactional` transform:

```
@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 t
errors object itself will be rendered in the appropriate format and a status code of 422 (UNPROCESSABLI

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'), 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 code of 2

Implementing the 'update' action

The `update` action updates an existing resource representations and is largely similar to the `save` action.

```
@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 t
errors object itself will be rendered in the appropriate format and a status code of 422 (UNPROCESSABL

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
'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 `update`
instead:

```
book.delete flush:true
withFormat {
    html {
        flash.message = message(code: 'default.deleted.message', args: [message(c
'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
returned.

10.1.5.3 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 a
you, simply run the command:

```
$ grails generate-controller [Domain Class Name]
```

10.1.6 The REST Profile

Since Grails 3.1, Grails supports a tailored profile for creating REST applications that provides a more focu

To get started with the REST profile create an application with by specifying `rest-api` as the name of tl

```
$ grails create-app my-api --profile rest-api
```

This will create a new REST application that provides the following features:

- Default set of commands for creating and generating REST endpoints
- Defaults to using JSON views for rendering responses (see the next section)
- Few plugins than the default Grails plugin (no GSP, no Asset Pipeline, Nothing HTML related)

You will notice for example in the `grails-app/views` directory that there are `*.gson` files for renc
404 and 500 errors.

If you issue the following set of commands:

```
$ grails create-domain-class book
$ grails generate-all my.api.Book
```

Instead of CRUD HTML interface a REST endpoint is generated that produces JSON responses. In add
default test the REST endpoint.

10.1.7 The Angular Profile

Since Grails 3.1, Grails supports a profile for creating applications with AngularJS that provides a more f
angular profile inherits from the REST profile and therefore has all of the commands and properties that th

To get started with the Angular profile create an application with by specifying `angular` as the name of t

```
$ grails create-app my-api --profile angular
```

This will create a new Grails application that provides the following features:

- Default set of commands for creating Angular artefacts
- Gradle plugin to manage client side dependencies
- Gradle plugin to execute client side unit tests
- Asset Pipeline plugins to ease development

By default the Angular profile includes GSP support in order to render the index page. This is necessa
pipeline.

The new commands are:

- `create-ng-component`
- `create-ng-controller`
- `create-ng-directive`
- `create-ng-domain`
- `create-ng-module`
- `create-ng-service`

Project structure

The Angular profile is designed around a specific project structure. The `create-ng` commands will a exist.

Example:

```
$ grails create-ng-controller foo
```

This will produce a `fooController.js` file in `grails-app/assets/javascripts/${defau`

> ⚠ By default the angular profile will create files in the `javascripts` directory. You can configuration with the key `grails.codegen.angular.assetDir`.

```
$ grails create-ng-domain foo.bar
```

This will produce a `Bar.js` file in `grails-app/assets/javascripts/foo/domains`. It w already exist.

```
$ grails create-ng-module foo.bar
```

This will produce a `foo.bar.js` file in `grails-app/assets/javascripts/foo/bar`. Note than other artefacts.

```
$ grails create-ng-service foo.bar --type constant
```

This will produce a `bar.js` file in `grails-app/assets/javascripts/foo/services`. It v already exist. The `create-ng-service` command accepts a flag `-type`. The types that can be used ar

- service
- factory *default*
- value
- provider
- constant

Along with the artefacts themselves, the profile will also produce a skeleton unit test file under `src/test`

Client side dependencies

The [Gradle Bower Plugin] is used to manage dependencies with bower. Visit the plugin documentation to l

Unit Testing

The [Gradle Karma Plugin] is used to execute client side unit tests. All generated tests are written with Ja how to use the plugin.

Asset Pipeline

The Angular profile includes several asset pipeline plugins to make development easier.

- [JS Closure Wrap Asset Pipeline] will wrap your Angular code in immediately invoked function expres
- [Annotate Asset Pipeline] will annotate your dependencies to be safe for minification.
- [Template Asset Pipeline] will put your templates into the `$templateCache` to prevent http requests

10.1.8 JSON Views

As mentioned in the previous section the REST profile by default uses JSON views to render JSON re instead are optimized for outputing JSON responses instead of HTML.

You can continue to separate your application in terms of MVC, with the logic of your application residin matters are handled by JSON views.

JSON views also provide the flexibility to easily customize the JSON presented to clients without hav libraries like Jackson or Grails' marshaller API.

> ⚠ Since Grails 3.1, JSON views are considered by the Grails team the best way to present JSON that reason the section on writing custom marshallers has been removed from the user g information on that topic, see [the Grails 3.0.x guide].

10.1.8.1 Getting Started

If you are using the REST or AngularJS profiles then the JSON views plugin will already be included Otherwise you will need to modify your `build.gradle` to include the necessary plugin to activate JSON

```
compile 'org.grails.plugins:views-json:1.0.0' // or whatever is the latest versio
```

Tip: The source code repository for JSON views can be found on Github if you are looking contributions

In order to compile JSON views for production deployment you should also activate the Gradle plugin by f

```
buildscript {
    …
    dependencies {
        …
        classpath "org.grails.plugins:views-gradle:1.0.0"
    }
}
```

Then apply the `org.grails.plugins.views-json` Gradle plugin after any Grails core gradle plug

```
…
apply plugin: "org.grails.grails-web"
apply plugin: "org.grails.plugins.views-json"
```

This will add a `compileGsonViews` task to Gradle, which is invoked prior to creating the production J/

10.1.8.2 Creating JSON Views

JSON views go into the `grails-app/views` directory and end with the `.gson` suffix. They are re; Groovy editor.

Example JSON view:

```
json.person {
    name "bob"
}
```

> Tip: To open them in the Groovy editor in Intellij double click on the file and when asked \
choose "Groovy"

The above JSON view produces:

```
{"person":{"name":"bob"}}
```

There is an implicit `json` variable which is an instance of [StreamingJsonBuilder](#).

Example usages:

```
json(1,2,3) == "[1,2,3]"
json { name "Bob" } == '{"name":"Bob"}'
json([1,2,3]) { n it } == '[{"n":1},{"n":2},{"n":3}]'
```

Refer to the API documentation on [StreamingJsonBuilder](#) for more information about what is possible.

10.1.8.3 JSON View Templates

You can define templates starting with underscore _. For example given the following template called _pe

```
model {
    Person person
}
json {
    name person.name
    age person.age
}
```

You can render it with a view as follows:

```
model {
    Family family
}
json {
    name family.father.name
    age family.father.age
    oldestChild g.render(template:"person", model:[person: family.children.max {
    children g.render(template:"person", collection: family.children, var:'person
}
```

Alternatively for a more concise way to invoke templates, using the tmpl variable:

```
model {
    Family family
}
json {
    name family.father.name
    age family.father.age
    oldestChild tmpl.person( family.children.max { Person p -> p.age } ] )
    children tmpl.person( family.children )
}
```

10.1.8.4 Rendering Domain Classes with JSON Views

Typically your model may involve one or many domain instances. JSON views provide a render method fc

For example given the following domain class:

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

And the following template:

```
model {
    Book book
}

json g.render(book)
```

The resulting output is:

```
{id:1,"title":"The Stand"}
```

You can customize the rendering by including or excluding properties:

```
json g.render(book, [includes:['title']])
```

Or by providing a closure to add additional JSON output:

```
json g.render(book) {
    pages 1000
}
```

10.1.8.5 JSON Views by Convention

There are a few useful conventions you can follow when creating JSON views. For example if you hav template located at `grails-app/views/book/_book.gson` and using the respond method will resu

```
def show(Long id) {
    respond Book.get(id)
}
```

In addition if an error occurs during validation by default Grails will try to render a template called `gra` otherwise it will try ty render `grails-app/views/errors/_errors.gson` if the former doesn't e

This is useful because when persisting objects you can `respond` with validation errors to render these afo

```
@Transactional
def save(Book book) {
    if (book.hasErrors()) {
        transactionStatus.setRollbackOnly()
        respond book.errors
    }
    else {
        // valid object
    }
}
```

If a validation error occurs in the above example the `grails-app/views/book/_errors.gson` te

For more information on JSON views (and Markup views), see the README and documentation included

10.1.9 Customizing Response Rendering

If you are looking for a more low-level API and JSON or Markup views don't suite your needs then you renderer.

10.1.9.1 Customizing the Default Renderers

The default renderers for XML and JSON can be found in the `grails.rest.render.xml` an respectively. These use the Grails converters (`grails.converters.XML` and `grails.converter`

You can easily customize response rendering using these default renderers. A common change you may properties from rendering.

Including or Excluding Properties from Rendering

As mentioned previously, Grails maintains a registry of `grails.rest.render.Renderer` instance and the ability to register or override renderers for a given domain class or even for a collection of domai rendering you need to register a custom renderer by defining a bean in `grails-app/conf/spring/r`

```
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 registe
> organizational and readability purposes it is recommended you name it something 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. I
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 wh
implementing a custom renderer you could use [Jackson](), [Gson]() or any Java library to implement the rende
tied to Grails' own marshalling implementation.

10.1.9.2 Implementing a Custom Renderer

If you want even more control of the rendering or prefer to use your own marshalling techniques then you
For example below is a simple implementation that customizes the rendering of the `Book` class:

```
package myapp
import grails.rest.render.*
import 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 `Mi
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
> binding if you plan to support POST and PUT requests. Grails will not automatically know hc
> XML format to a domain class otherwise. See the section on "Customizing Binding of Resour

Container Renderers

A `grails.rest.render.ContainerRenderer` is a renderer that renders responses for containe
interface is largely the same as the `Renderer` interface except for the addition of the `getCompon
"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) {
        ....
    }
}
```

10.1.9.3 Using GSP to Customize Rendering

You can also customize rendering on a per action basis using Groovy Server Pages (GSP). For example giv

```
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}"/>
```

10.1.10 Hypermedia as the Engine of Application State

HATEOAS, an abbreviation for Hypermedia as the Engine of Application State, is a common pattern appl 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 lin The format of the response is typically JSON or XML, although standard formats such as Atom and/or HA

10.1.10.1 HAL Support

HAL is a standard exchange format commonly used when developing REST APIs that follow HATE 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 override the renderer in `grails-` with an instance of `grails.rest.render.hal.HalJsonRenderer` (or `HalXmlRenderer` for t

```
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/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/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 `grails-app/conf/spring/resources.groovy` with an instance of `grails.rest.render.`

```
import grails.rest.render.hal.*
beans = {
    halBookCollectionRenderer(HalJsonCollectionRenderer, 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/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/books",
      "hreflang": "en",
      "type": "application/hal+json"
    }
  },
  "_embedded": {
    "book": [
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/books/1",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "The Stand"
      },
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/books/2",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "Infinite Jest"
      },
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/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 dei namely `Book`. In order to customize the value of this key assign a value to the `collectionName` prope 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/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/books",
      "hreflang": "en",
      "type": "application/hal+json"
    }
  },
  "_embedded": {
    "publications": [
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/books/1",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "The Stand"
      },
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/books/2",
            "hreflang": "en",
            "type": "application/hal+json"
          }
        },
        "title": "Infinite Jest"
      },
      {
        "_links": {
          "self": {
            "href": "http://localhost:8080/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 Mime Types in `grails-app/`

```
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 ' established then the first entry in the map is used for the response. If you have your new Min will always try and send back your new Mime Type if the requested Mime Type cannot be est

Then override the renderer to return HAL using the custom Mime Types:

```
import grails.rest.render.hal.*
import grails.web.mime.*

beans = {
    halBookRenderer(HalJsonRenderer, rest.test.Book, new MimeType("application/vn
    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 in `application/vnd.books.org.book+json`. The second bean defines the Mime Type used t `application/vnd.books.org.booklist+json`).

`application/vnd.books.org.booklist+json` is an example (http://www.w3.org/Protocols/rfc2616/rfc2616.html - Header Field Definitions). This exa operation (list) to form the media-range values but in reality, it may not be necessary to cre each operation. Further, it may not be necessary to create Mime types at the entity level. Se REST resources" for further information about how to define your own Mime types.

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/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 `HalJsonRenderer` will automatically create links for you for associations and to the resource itself (usi

However you can customize link rendering using the `link` method that is added to all domain classes a any class annotated with `grails.rest.Linkable`. For example, the `show` action can be modified a output:

```
def show(Book book) {
    book.link rel:'publisher', href: g.createLink(absolute: true, resource:"publi
    respond book
}
```

Which will result in output such as:

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/books/1",
      "hreflang": "en",
      "type": "application/vnd.books.org.book+json"
    }
    "publisher": {
        "href": "http://localhost:8080/books/1/publisher",
        "hreflang": "en"
    }
  },
  "title": ""The Stand""
}
```

The `link` method can be passed named arguments that match the properties of the `grails.rest.Lin`

10.1.10.2 Atom Support

[Atom](#) is another standard interchange format used to implement REST APIs. An example of Atom output

```
<?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)
}
```

10.1.10.3 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 object w

```
$ curl -i -H "Accept: application/json"  -H "Content-Type: application/json" -X P
http://localhost:8080/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
[class rest.test.Book] cannot be null"}]}
```

If you wish to change the format to Vnd.Error then simply register `grails.rest.render.er` `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
http://localhost:8080/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"
            }
        }
    }
]
```

10.1.11 Customizing Binding of Resources

The framework provides a sophisticated but simple mechanism for binding REST requests to domain c advantage of this is to bind the `request` property in a controller the `properties` of a domain class request, the `createBook` action will create a new `Book` and assign "The Stand" to the `title` prop 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

// …
    }
}
```

Command objects will automatically be bound with the body of the request:

```
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 `ic` the corresponding persistent instance from the database and then the rest of the document will be boun found in the database, the command object reference will be null.

```
<?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 an instance o specific implementation of `DataBindingSourceCreator` will be selected based on the `contentTy` provided to handle common content types. The default implementations will be fine for most use cases. T are supported by the core framework and which `DataBindingSourceCreator` implementations are are in the `org.grails.databinding.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 `DataBindingSourceCreator` and register an instance of that class in the Spring application context. use the corresponding bean name from above. If you are providing a helper for a content type other than bean name may be anything that you like but you should take care not to conflict with one of the bean nam

The `DataBindingSourceCreator` interface defines just 2 methods:

```
package org.grails.databinding.bindingsource

import grails.web.mime.MimeType
import 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
}
```

AbstractRequestBodyDataBindingSourceCreator is an abstract class designed to be e `DataBindingSourceCreator` classes. Classes which extend `AbstractRequestbodyDatabir` method named `createBindingSource` which accepts an `InputStream` as an argument and implementing the `getMimeTypes` method described in the `DataBindingSourceCreator` interf `createBindingSource` provides 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 grails.web.mime.MimeType
import 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

// …
}
```

With that in place the framework will use the `myCustomCreator` bean any time a `DataBindingSou`
which has a `contentType` of "text/custom+demo+csv".

10.2 RSS and Atom

No direct support is provided for RSS or Atom within Grails. You could construct RSS or ATOM feeds v
is however a Feeds plugin available for Grails that provides a RSS and Atom builder using the popular
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
            }
        }
    }
}
```

11 Asynchronous Programming

With modern hardware featuring multiple cores, many programming languages have been adding asyn
being no exception.

The excellent [GPars] project features a whole range of different APIs for asynchronous programming techr
flow concurrency.

Added Grails 2.3, the Async features of Grails aim to simplify concurrent programming within the framew
unified event model.

11.1 Promises

A Promise is a concept being embraced by many concurrency frameworks. They are similar to `java.`
include a more user friendly exception handling model, useful features like chaining and the ability to attac

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.Pr`

```
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 c

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. Tl `onComplete` method, however, will simply not execute the passed closure if an exception occurs. You c 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 provides a s

```
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 in t

Promise Lists and Maps

Grails' async API also features the concept of a promise lists and maps. These are represented by `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 Pı

```
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
}
```

> ⚠ The `PromiseList` class does not implement the java.util.List interface, but instead returns 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` instanc

The default implementation uses the GPars concurrency library and is called `org.grails.async.fa` however it is possible to swap implementations by setting the `Promises.promiseFactory` variable.

One common use case for this is unit testing, typically you do not want promises to execute asynchronous write. For this purpose Grails ships with a `org.grails.async.factory.SynchronousPromis` promises:

```
import org.grails.async.factory.*
import grails.async.*

Promises.promiseFactory = new SynchronousPromiseFactory()
```

Grails also has a `org.grails.async.factory.reactor.ReactorPromiseFactory` which a settings `Promises.promiseFactory` as given above.

Using the `PromiseFactory` mechanism it is theoretically possible to plug in other concurrency librari to override the two interfaces `grails.async.Promise` and `grails.async.PromiseFactory`.

DelegateAsync Transformation

It is quite common to require both synchronous and asynchronous versions of the same API. Developir typically the asynchronous API would simply delegate to the synchronous version.

The `DelegateAsync` transformation is designed to mitigate this problem by transforming any synchron

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 class as follows:

```
import grails.async.*

class AsyncBookService {
   @DelegateAsync BookService bookService
}
```

The `DelegateAsync` transformation will automatically add a new method that looks like the following

```
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}"
       }
}
```

11.2 Events

Grails 3.0 introduces a new Events API based on Reactor.

All services and controllers in Grails 3.0 implement the Events trait.

The `Events` trait allows the ability to consume and publish events that are handled by Reactor.

The default Reactor configuration utilises a thread pool backed event bus. You can however configure Rea

```
reactor:
    dispatchers:
        default: myExecutor
        myExecutor:
            type: threadPoolExecutor
            size: 5
            backlog: 2048
```

11.2.1 Consuming Events

There are several ways to consume an event. As mentioned previously services and controllers implement

The `Events` trait provides several methods to register event consumers. For example:

```
on("myEvent") {
    println "Event fired!"
}
```

Note that if you wish a class (other than a controller or service) to be an event consumer you simply have class is registered as a Spring bean.

For example given the following class:

```
import grails.events.*
import javax.annotation.*

class MyClass implements Events {

@PostConstruct
    void init() {
        on("myEvent") {
            println "Event fired!"
        }
    }
}
```

You can override `doWithSpring` in your `Application` class to register it as a Spring bean (or annota

```
Closure doWithSpring() {
        {->
            myClass(MyClass)
        }
    }
```

11.2.2 Event Notification

The `Events` trait also provides methods for notifying of events. For example:

```
notify "myEvent", "myData"
sendAndReceive "myEvent", "myData", {
    println "Got response!"
}
```

11.2.3 Reactor Spring Annotations

Reactor provides a few useful annotations that can be used for declaratively consuming events in a Grails s

To declare an event consumer use the `Consumer` annotation:

```
import reactor.spring.context.annotation.*

@Consumer
class MyService {

}
```

Then to register to listen for an event use the `Selector` annotation:

```
import reactor.spring.context.annotation.*

@Consumer
class MyService {
    @Selector('my.event')
    void myEventListener(Object data) {
        println "GOT EVENT $data"
    }
}
```

11.2.4 Events from GORM

GORM defines a [number of useful events](#) that you can listen for.

Each event is translated into a key that starts with `gorm:`. For example:

```
import org.grails.datastore.mapping.engine.event.*
...
on("gorm:preInsert") { PreInsertEvent event ->
    println "GOT EVENT $event"
}
```

> These events are triggered asynchronously, and so cannot cancel or manipulate the persisten
> do that see the section on [Events & Auto Timestamping](#) in the GORM docs

11.2.5 Events from Spring

Spring also fires a number of useful events. All events in the `org.springframework` package are pref

For example:

```
import org.springframework.web.context.support.*
import org.springframework.boot.context.event.*
...

on("spring:applicationStarted") { ApplicationStartedEvent event ->
    // fired when the application starts
}

on("spring:servletRequestHandled") { RequestHandledEvent event ->
    // fired each time a request is handled
}
```

11.3 Asynchronous GORM

Since Grails 2.3, GORM features an asynchronous programming model that works across all supported dat

> Although GORM executes persistence operations asynchronously, these operations still bloc drivers are not asynchronous. Asynchornous GORM is designed to allow you to isolate thes separate thread you can scale and control allowing your controller layer to remain non-blockir

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), but i run in the background and a `Promise` instance is returned.

The following code listing shows a few common examples of GORM queries executed asynchronously:

```
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 n thread.

This is an important consideration when using GORM async (particularly with Hibernate as the pe asynchronous queries will be detached entities.

This means you cannot save objects returned from asynchronous queries without first merging them back 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 wo is not recommended to read and write objects in different threads and you should avoid this technique unle

Finally, another issue with detached objects is that association lazy loading **will not** work and you will enc
errors if you do so. If you plan to access the associated objects of those returned from asynchronous c
recommended anyway to avoid N+1 problems).

Multiple Asynchronous GORM calls

As discussed in the previous section you should avoid reading and writing objects in different threads as m

However, if you wish to do more complex GORM work asynchronously then the GORM async names
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 w
thread for you. If you do not use the GORM version and do asynchronous work with GORM then you need

```
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()
```

11.4 Asynchronous Request Handling

If you are deploying to a Servlet 3.0 container such as Tomcat 7 and above then it is possible to deal with r

In general for controller actions that execute quickly there is little benefit in handling requests asynchr actions it is extremely beneficial.

The reason being that with an asynchronous / non-blocking response, the one thread == one request == on can keep a client response open and active, and at the same time return the thread back to the container to c

For example, if you have 70 available container threads and an action takes a minute to complete, if the ac the likelihood of all 70 threads being occupied and the container not being able to respond is quite high processing.

Since Grails 2.3, Grails features a simplified API for creating asynchronous responses built on the `Promis`

The implementation is based on Servlet 3.0 async. So, to enable the async features you need to set your ser

```
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 render

If the model takes a while to produce then the server could arrive at a blocking state, impacting s asynchronously by returning a `grails.async.PromiseMap` via the `Promises.tasks` method:

```
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 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` m

```
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:

```
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=${
           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 re 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

11.5 Servlet 3.0 Async

In addition to the higher level async features discussed earlier in the section, you can access the raw application.

Servlet 3.0 Asynchronous Rendering

You can render content (templates, binary data etc.) in an asynchronous manner by calling the `startA` Servlet 3.0 `AsyncContext`. Once you have a reference to the `AsyncContext` you can use Grails' regu

```
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 `disp`

```
def index() {
    def ctx = startAsync()
    ctx.start {
        // do working
        …
        // render view
        ctx.dispatch()
    }
}
```

12 Validation

Grails validation capability is built on [Spring's Validator API] and data binding capabilities. However Gra
to define validation "constraints" with its constraints mechanism.

Constraints in Grails are a way to declaratively specify validation rules. Most commonly they are applied t
[Command Objects] also support constraints.

12.1 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 w

```
class User {
    ...

static constraints = {
        login size: 5..15, blank: false, unique: true
        password size: 5..15, blank: false
        email email: true, blank: false
        age min: 18
    }
}
```

In this example we've declared that the `login` property must be between 5 and 15 characters long, it ca
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`

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 value l

```
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 legal `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 return a boolean to indicate whether the new value for the `answer` property, `val`, is valid.

12.2 Validating Constraints

Validation Basics

Call the validate method to validate a domain class instance:

```
def user = new User(params)

if (user.validate()) {
    // do something with user
}
else {
    user.errors.allErrors.each {
        println it
    }
}
```

The `errors` property on domain classes is an instance of the Spring Errors interface. The `Errors` interf
errors and also retrieve the original values.

Validation Phases

Within Grails there are two phases of validation, the first one being data binding which occurs when you b

```
def user = new User(params)
```

At this point you may already have errors in the `errors` property due to type conversion (such as conver
obtain the original input value using the `Errors` API:

```
if (user.hasErrors()) {
    if (user.errors.hasFieldErrors("login")) {
        println user.errors.getFieldError("login").rejectedValue
    }
}
```

The second phase of validation happens when you call validate or save. This is when Grails will valida
defined. For example, by default the save method calls `validate` before executing, allowing you to writ

```
if (user.save()) {
    return user
}
else {
    user.errors.allErrors.each {
        println it
    }
}
```

12.3 Sharing Constraints Between Classes

A common pattern in Grails is to use [command objects] for validating user-submitted data and then coן relevant domain classes. This often means that your command objects and domain classes share properties and paste the constraints between the two, but that's a very error-prone approach. Instead, make use of Gra

Global Constraints

In addition to defining constraints in domain classes, command objects and [other validateab] `grails-app/conf/application.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 constraints defined in Despite the name of the configuration setting, you can reference these shared constraints from any validate

The '*' constraint is a special case: it means that the associated constraints ('nullable' and 'size' in the above validateable classes. These defaults can be overridden by the constraints declared in a validateable class.

Importing Constraints

Grails 2 introduced an alternative approach to sharing constraints that allows you to import a set of constra

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 the c
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`. The
class (`User`) that don't have corresponding properties in the importing class (`UserCommand`). In
'lastName' constraints will be imported into `UserCommand` because those are the only properties shared b

If you want more control over which constraints are imported, use the `include` and `exclude` argument
expression strings that are matched against the property names in the source constraints. So for exampl
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.

12.4 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 s rich set of tags for dealing with errors. To render the errors as a list you can use renderErrors:

```
<g:renderErrors bean="${user}" />
```

If you need more control you can use hasErrors and eachError:

```
<g:hasErrors bean="${user}">
  <ul>
   <g:eachError var="err" bean="${user}">
       <li>${err}</li>
   </g:eachError>
  </ul>
</g:hasErrors>
```

Highlighting Errors

It is often useful to highlight using a red box or some indicator when a field has been incorrectly inpu 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 cl highlight the `div`.

Retrieving Input Values

Each error is actually an instance of the FieldError class in Spring, which retains the original input value 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

12.5 Validation and Internationalization

Another important thing to note about errors in Grails is that error messages are not hard coded anyv 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 `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 ver com.mycompany.myapp.User.login.blank will be used before user.login.blank. This allows for cases wher 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 yourself:

```
<g:hasErrors bean="${user}">
  <ul>
   <g:eachError var="err" bean="${user}">
       <li><g:message error="${err}" /></li>
   </g:eachError>
  </ul>
</g:hasErrors>
```

In this example within the body of the eachError tag we use the message tag in combination with its `err` error.

12.6 Applying Validation to Other Classes

Domain classes and command objects support validation by default. Other classes may be made valid property in the class (as described above) and then telling the framework about them. It is important that with the framework. Simply defining the `constraints` property is not sufficient.

The Validateable Trait

Classes which define the static `constraints` property and implement the Validateable trait will be valid

```
// src/groovy/com/mycompany/myapp/User.groovy
package com.mycompany.myapp

import grails.validation.Validateable

class User implements Validateable {
    ...

static constraints = {
        login size: 5..15, blank: false, unique: true
        password size: 5..15, blank: false
        email email: true, blank: false
        age min: 18
    }
}
```

13 The Service Layer

Grails defines the notion of a service layer. The Grails team discourages the embedding of core applicatio
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 resp
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 te

```
grails create-service helloworld.simple
```

> ⚠ If no package is specified with the create-service script, Grails automatically uses the appl
> name.

The above example will create a service at the location `grails-app/services/helloworld/S`
ends with the convention `Service`, other than that a service is a plain Groovy class:

```
package helloworld

class SimpleService {
}
```

13.1 Declarative Transactions

Declarative Transactions

Services are typically involved with coordinating logic between domain classes, and hence often involv
Given the nature of services, they frequently require transactional behaviour. You can use programmatic
however this is repetitive and doesn't fully leverage the power of Spring's underlying transaction abstractio

Services enable transaction demarcation, which is a declarative way of defining which methods are to be
service use the `Transactional` transform:

```
import grails.transaction.*

@Transactional
class CountryService {

}
```

The result is that all methods are wrapped in a transaction and automatic rollback occurs if a method thı RuntimeException) or an Error. The propagation level of the transaction is by default set to PROPA

> Checked exceptions do **not** roll back transactions. Even though Groovy blurs the distir unchecked exceptions, Spring isn't aware of this and its default behaviour is used, so it's distinction between checked and unchecked exceptions.

> Warning: dependency injection is the **only** way that declarative transactions work. You will ı if you use the `new` operator such as `new BookService()`

The Transactional annotation vs the transactional property

In previous versions of Grails prior to Grails 3.1, Grails created Spring proxies and used the `transac` creation. These proxies are disabled by default in Grails 3.1 and above in favour of the `@Transactiona`

If you wish to renable this feature (not recommended) then you must set `grails.spring` `grails-app/conf/application.yml` or `grails-app/conf/application.groovy`

> In addition, prior to Grails 3.1 services were transactional by default, as of Grails 3.1 they `@Transactional` transformation is applied.

Custom Transaction Configuration

Grails also provides `@Transactional` and `@NotTransactional` annotations for cases where you r at a per-method level or need to specify an alternative propagation level. For example, the `@NotTran:` particular method to be skipped when a class is annotated with `@Transactional`.

> The `grails.transaction.Transactional` annotation was first introduced in Gra @Transactional annotation was used.

> ⚠ Annotating a service method with `Transactional` disables the default Grails transactiona
> the same way that adding `transactional=false` does) so if you use any annotations y
> that require transactions.

In this example `listBooks` uses a read-only transaction, `updateBook` uses a default read-write tran
(probably not a good idea given its name).

```
import grails.transaction.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 then
this service is equivalent to one that has no annotations (since the default is implicitly `transactional=`

```
import grails.transaction.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
read-only transaction:

```
import grails.transaction.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 configuratio

For more information refer to the section of the Spring user guide on [Using @Transactional](#).

Unlike Spring you do not need any prior configuration to use `Transactional`; just specify the annot automatically.

13.1.1 Transactions Rollback and the Session

Understanding Transactions and the Hibernate Session

When using transactions there are important considerations you must take into account with regards to ho by Hibernate. When a transaction is rolled back the Hibernate session used by GORM is cleared. This detached and accessing uninitialized lazy-loaded collections will lead to `LazyInitializationExcep`

To understand why it is important that the Hibernate session is cleared. Consider the following example:

```
class Author {
    String name
    Integer age

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

If you were to save two authors using consecutive transactions as follows:

```
Author.withTransaction { status ->
    new Author(name: "Stephen King", age: 40).save()
    status.setRollbackOnly()
}

Author.withTransaction { status ->
    new Author(name: "Stephen King", age: 40).save()
}
```

Only the second author would be saved since the first transaction rolls back the author `save()` by clearin
were not cleared then both author instances would be persisted and it would lead to very unexpected result

It can, however, be frustrating to get `LazyInitializationException`s 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 maximum
throwing an `AuthorException`. The `AuthorException` references the author but whe
`LazyInitializationException` will be thrown because the underlying Hibernate session has been

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 r
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 refre

```
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]
        }
    }
}
```

13.2 Scoped Services

By default, access to service methods is not synchronised, so nothing prevents concurrent execution of singleton and may be used concurrently, you should be very careful about storing state in a service. Or tak 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 implement `java.i` 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"
```

Upgrade note: Starting with Grails 2.3, new applications are generated with configuratio
controllers to `singleton`. If `singleton` controllers interact with `prototype` scoped se
behave as per-controller singletons. If non-singleton services are required, controller scope sh

See Controllers and Scopes in the user guide for more information.

13.3 Dependency Injection and Services

Dependency Injection Basics

A key aspect of Grails services is the ability to use Spring Framework's dependency injection featu
convention". In other words, you can use the property name representation of the class name of a service t
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 configu
name. You can also specify the type as follows:

```
class AuthorService {
    BookService bookService
}
```

NOTE: Normally the property name is generated by lower casing the first letter of the type. F
`BookService` class would map to a property named `bookService`.

To be consistent with standard JavaBean conventions, if the first 2 letters of the class name
name is the same as the class name. For example, the property name of the `JDBCHelpe`
`JDBCHelperService`, not `jDBCHelperService` or `jdbcHelperService`.

See section 8.8 of the JavaBean specification for more information on de-capitalization rules.

Only the top level object is subjected to injection as traversing all nested objects to p
performance issue.

Be careful when injecting the non-default datasources. For example, using this config:

```
dataSources:
    dataSource:
        pooled: true
        jmxExport: true
    .....
    secondary:
        pooled: true
        jmxExport: true
    .....
```

You can inject the primary `dataSource` like you would expect:

```
class BookSqlService {

def dataSource
}
```

But to inject the `secondary` datasource, you have to use Spring's `Autowired` injection or `resources`

```
class BookSqlSecondaryService {

@Autowired
  @Qualifier('dataSource_secondary')
  def dataSource2
}
```

Dependency Injection and Services

You can inject services in other services with the same technique. If you had an `AuthorService` that n
`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 ricl

```
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 ser packages. For example consider the situation where an application defines a service class named `com.de` uses a plugin named `ReportingUtilities` and that plugin provides a service class named `com.re|` default bean name for each of those would be `reportingService` so they would conflict with each otl bean name for services provided by plugins by prefixing the bean name with the plugin name. In the scena bean would be an instance of the `com.demo.ReportingService` class de `reportingUtilitiesReportingService` bean would be an instance of the `com.reporting` by the `ReportingUtilities` plugin. For all service beans provided by plugins, if there are no application or other plugins in the application then a bean alias will be created which does not include tl referred to by the name that does include the plugin name prefix. For example, if the `ReportingU` `com.reporting.util.AuthorService` and there is no other `AuthorService` in the applicatio using then there will be a bean named `reportingUtilitiesAuthorServic` `com.reporting.util.AuthorService` class and there will be a bean alias defined in the context same bean.

14 Static Type Checking And Compilation

Groovy is a dynamic language and by default Groovy uses a dynamic dispatch mechanism to carry out r dispatch mechanism provides a lot of flexibility and power to the language. For example, it is possible to and it is possible to dynamically replace existing methods at runtime. Features like these are importa However, there are times when you may want to disable this dynamic dispatch in favor of a more static d to do that. The way to tell the Groovy compiler that a particular class should compiled groovy.transform.CompileStatic annotation 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 ar example, in Grails you would not be able to invoke a GORM dynamic finder from a class that is marked cannot verify that the dynamic finder method exists, because it doesn't exist at compile time. It may be tha compilation benefits without giving up access to dynamic dispatch for Grails specific things grails.compiler.GrailsCompileStatic comes in. `GrailsCompileStatic` behaves just like `CompileS` and allows access to those specific features to be accessed dynamically.

14.1 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 by marki specifying that the type checking should be skipped for that particular method as shown below.

```
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 Grail compiled but that `GrailsCompileStatic` can identify as permissible for dynamic dispatch. These ir DSL code in configuration blocks like constraints and mapping closures in domain classes.

Care must be taken when deciding to statically compile code. There are benefits associated with static cor benefits you are giving up the power and flexibility of dynamic dispatch. For example if code is staticall metaprogramming enhancements which may be provided by plugins.

14.2 The GrailsTypeChecked Annotation

GrailsTypeChecked

The grails.compiler.GrailsTypeChecked annotation works a lot like the `GrailsCompileStatic` an checking, not static compilation. This affords compile time feedback for expressions which cannot be 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() {
        // …
    }
}
```

15 Testing

Automated testing is a key part of Grails. Hence, Grails provides many ways to making testing easier fro
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
> Grails 2.0.x and above deprecates these test harnesses in favour of mixins that can be applie
> 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 `un`
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/SimpleCont`
`test/unit/com/acme/app/SimpleControllerTests.groovy`. What Grails won't do howe
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 v
are generally the best ones to look at.

Using Grails' interactive mode confers some distinct advantages when executing tests. First, the tests wil
subsequent runs. Second, a shortcut is available to open the HTML reports in your browser:

```
open test-report
```

You can also run your unit tests from within most IDEs.

Targeting Tests

You can selectively target the test(s) to be run in different ways. To run all tests for a controller named `Si`

```
grails test-app SimpleController
```

This will run any tests for the class named `SimpleController`. Wildcards can be used...

```
grails test-app *Controller
```

This will test all classes ending in `Controller`. Package names can optionally be specified...

```
grails test-app some.org.*Controller
```

or to run all tests in a package...

```
grails test-app some.org.*
```

or to run all tests in a package including subpackages...

```
grails test-app some.org.**.*
```

You can also target particular test methods...

```
grails test-app SimpleController.testLogin
```

This will run the `testLogin` test in the `SimpleController` tests. You can specify as many patterns i

```
grails test-app some.org.* SimpleController.testLogin BookController
```

> In Grails 2.x, adding `-rerun` as an argument would only run those tests which failed in th
> argument is no longer supported.

Debugging

In order to debug your tests via a remote debugger, you can add `--debug-jvm` after `grails` in any con

```
grails --debug-jvm test-app
```

This will open the default Java remote debugging port, 5005, for you to attach a remote debugger from you

> This differs from Grails 2.3 and previous, where the `grails-debug` command existed.

Targeting Test Phases

In addition to targeting certain tests, you can also target test *phases*. By default Grails has two testing phas

> Grails 2.x uses `phase:type` syntax. In Grails 3.0 it was removed, because it made no sense

To execute `unit` tests you can run:

```
grails test-app -unit
```

To run `integration` tests you would run...

```
grails test-app -integration
```

Targeting Tests When Using Phases

Test and phase targeting can be applied at the same time:

```
grails test-app some.org.**.* -unit
```

This would run all tests in the `unit` phase that are in the package `some.org` or a subpackage.

15.1 Unit Testing

Unit testing are tests at the "unit" level. In other words you are testing individual methods or blocks infrastructure. Unit tests are typically run without the presence of physical resources that involve I/O such ensure they run as quick as possible since quick feedback is important.

The Test Mixins

Since Grails 2.0, a collection of unit testing mixins is provided by Grails that lets you enhance the behavio following sections cover the usage of these mixins.

> The previous JUnit 3-style `GrailsUnitTestCase` class hierarchy is still present in Grails but is now deprecated. The previous documentation on the subject can be found in the [Grails 1

You won't normally have to import any of the testing classes because Grails does that for you. But if yo 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 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 of c "controller" field will be present, however if `TestFor` was defined for a service a "service" field would b

The `Mock` annotation creates mock version of any collaborators. There is an in-memory implementation with the GORM API.

doWithSpring and doWithConfig callback methods, FreshRuntime annotation

The `doWithSpring` callback method can be used to add beans with the BeanBuilder DSL. There is the the grailsApplication.config values before the grailsApplication instance of the test runtime gets 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.Fre`]() 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.grails.spring.beans.factory.InstanceFactoryBean` together with do mock beans in tests.

```
import grails.test.mixin.support.GrailsUnitTestMixin
import grails.test.runtime.FreshRuntime

import org.grails.spring.beans.factory.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 to in which might be problematic for other tests and as such the runtime should be refreshed after this test metho

```
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 feature is This annotation takes an optional class parameter implements `SharedRuntimeConfigurer` int SharedRuntimeConfigurer implementation class will share the same runtime during a single test run. annotation can also implement `TestEventInterceptor` . In this case the instance of the class will be runtime.

Loading application beans in unit tests

Adding `static loadExternalBeans = true` field definition to a unit test class makes the Grails `grails-app/conf/spring/resources.groovy` and `grails-app/conf/spring/resour`

```
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!'
    }
}
```

15.1.1 Unit Testing Controllers

The Basics

You use the `grails.test.mixin.TestFor` annotation to unit test controllers. Using `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 crea

The `TestFor` annotation will also automatically annotate any public methods starting wit annotation. If any of your test method don't start with "test" just add this manually

To test the simplest "Hello World"-style example you can do the following:

```
// Test class
class SimpleController {
    def hello() {
        render "hello"
    }
}
```

```
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 `GrailsMockHttpServletR` `org.codehaus.groovy.grails.plugins.testing`) which extends Spring's `MockHttpSer` 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', 'm
Providing these in the test is as simple as adding appropriate values to a special `params` variable:

```
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. Final

```
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 `modelA` `org.springframework.web.servlet.ModelAndView`) or you can use the `view` and `model` p

```
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 controller.

Testing Template Rendering

Unlike view rendering, template rendering will actually attempt to write the template directly to the res 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.`

```
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 case you c

```
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 it c

```
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
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](http://groovy.codehaus.org/Reading+XML+using+Groovy's+XmlSlurper) class which is very convenient for parsing

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.JSONElemen` parsing JSON responses.

Testing XML and JSON Requests

Grails provides various convenient ways to automatically parse incoming XML and JSON packets. For e 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` proper by specifying a String containing the XML:

```groovy
import grails.test.mixin.TestFor
import grails.test.mixin.Mock
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

```groovy
import grails.test.mixin.TestFor
import grails.test.mixin.Mock
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:

```
import grails.test.mixin.TestFor
import grails.test.mixin.Mock
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 controller action below:

```
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
    }
}
```

```
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="
    }

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="
    }

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 execut 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 followir

```
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 ha 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. For

```
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 controller action method. Grails will create an instance of the command object, perform data binding on object just like it does in when the application is running. See the test below.

```
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'
    }
}
```

Testing allowedMethods

The unit testing environment respects the allowedMethods property in controllers. If a controller actior 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 t example to test a call to the `message` tag, add a message to the `messageSource`. Consider the followir

```
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.

15.1.2 Unit Testing Tag Libraries

The Basics

Tag libraries and GSP pages can be tested with the `grails.test.mixin.web.GroovyPageUni` 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 field can be used to test calling tags as function calls. The return value of a function call is either a Stream the tag closure when returnObjectForTags feature is used.

Note that if you are testing invocation of a custom tag from a controller you can combine the `GroovyPageUnitTestMixin` using the `Mock` annotation:

```
import grails.test.mixin.TestFor
import grails.test.mixin.Mock
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 libraries do. The `Gro` `mockTagLib()` method that you can use to mock a custom tag library. For example consider the followi

```
class SimpleTagLib {

static namespace = 's'

def hello = { attrs, body ->
        out << "Hello ${attrs.name ?: 'World'}"
    }

def bye = { attrs, body ->
        out << "Bye ${attrs.author.name ?: 'World'}"
    }
}
```

You can test this tag library by using `TestFor` and supplying the name of the tag library:

```
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: '
    }

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`

```
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 outpu

```
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)` method
:

```
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/_hello`
custom tag libraries you need to call `mockTagLib` as described in the previous section.

Some core tags use the active controller and action as input. In GroovyPageUnitTestMixin tests, you ca
name by setting controllerName and actionName properties on the webRequest object:

```
webRequest.controllerName = 'simple'
    webRequest.actionName = 'hello'
```

15.1.3 Unit Testing Domains

Overview

Domain class interaction can be tested without involving a real database connection using `Domai` `HibernateTestMixin`.

The GORM implementation in DomainClassUnitTestMixin is using a simple in-memory `Concurren` 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

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 i given domain classes for use in a unit test. The @Domain annotation is used to tell which domain classes s

DomainClassUnitTestMixin Basics

`DomainClassUnitTestMixin` is typically used in combination with testing either a controller, ser collaborator defined by the `Mock` annotation:

```
import grails.test.mixin.TestFor
import grails.test.mixin.Mock
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` doma 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 grails.test.mixin.Mock
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:

```
import grails.test.mixin.TestFor
import grails.test.mixin.Mock
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 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 data:

```
mockDomain(Book, [
            [title: "The Stand", pages: 1000],
            [title: "The Shining", pages: 400],
            [title: "Along Came a Spider", pages: 300] ])
```

Testing Constraints

There are 3 types of validateable classes:

1. Domain classes
2. Classes which implement the `Validateable` trait
3. Command Objects which have been made validateable automatically

These are all easily testable in a unit test with no special configuration necessary as long as the test method
the `GrailsUnitTestMixin` using `TestMixin`. See the examples below.

```
// src/groovy/com/demo/MyValidateable.groovy
package com.demo

class MyValidateable implements grails.validation.Validateable {
    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].*/
    }
}
```

```
// 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
    }
}
```

```
// 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()
    }
}
```

```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
    }
}
```

```
// 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()
    }
}
```

That's it for testing constraints. One final thing we would like to say is that testing the constraints in t "constraints" property name which is a mistake that is easy to make and equally easy to overlook. A u problem straight away.

HibernateTestMixin Basics

`HibernateTestMixin` allows Hibernate 4 to be used in Grails unit tests. It uses a H2 in-memory datab

```
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
            hibernateSession != null
    }
}
```

This library dependency is required in build.gradle for adding support for `HibernateTestMixin`.

```
dependencies {
        testCompile 'org.grails:grails-datastore-test-support:4.0.4.RELEASE'
    }
```

HibernateTestMixin is only supported with hibernate4 plugin versions >= 4.3.8.1 .

```
dependencies {
        compile "org.grails.plugins:hibernate:4.3.8.1"
    }
```

Configuring domain classes for HibernateTestMixin tests

The `grails.test.mixin.gorm.Domain` annotation is used to configure the list of domain clas 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 c
- 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/pack `SharedRuntime` class which has been added for the test.

It's not possible to use DomainClassUnitTestMixin's `Mock` annotation in HibernateTestMixin tests. Use t HibernateTestMixin tests.

15.1.4 Unit Testing Filters

Unit testing filters is typically a matter of testing a controller where a filter is a mock collaborator. For exa

```
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. targets the `SimpleController` class and add the `CancellingFilters` as a mock collaborator:

```
import grails.test.mixin.TestFor
import grails.test.mixin.Mock
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 grails.test.mixin.Mock
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 tl
parameter is optional and taken from the controller under test. If it is another controller you are testing ther

```
withFilters(controller:"book",action:"list") {
    controller.list()
}
```

15.1.5 Unit Testing URL Mappings

The Basics

Testing URL mappings can be done with the `TestFor` annotation testing a particular URL mappin
mappings you can do the following:

```
import com.demo.SimpleController
import grails.test.mixin.TestFor
import grails.test.mixin.Mock
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 also

With that done there are a number of useful methods that are defined by the `grails.test.mixir` testing URL mappings. These include:

- `assertForwardUrlMapping` - Asserts a URL mapping is forwarded for the given controller clas a mock collaborate for this to work)
- `assertReverseUrlMapping` - Asserts that the given URL is produced when reverse mapping a
- `assertUrlMapping` - Asserts a URL mapping is valid for the given URL. This combine `assertReverseUrlMapping` assertions

Asserting Forward URL Mappings

You use `assertForwardUrlMapping` to assert that a given URL maps to a given controller. For exan

```
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 grails.test.mixin.Mock
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 example test is largely identical to the previous listing except you use `assertReverseUrlMappin` Note that you can combine these 2 assertions with `assertUrlMapping`.

15.1.6 Mocking Collaborators

The Spock Framework manual has a chapter on [Interaction Based Testing] which also explains mocking co

15.1.7 Mocking Codecs

The `GrailsUnitTestMixin` provides a `mockCodec` method for mocking [custom codecs] which may

```
mockCodec(MyCustomCodec)
```

Failing to mock a codec which is invoked while a unit test is running may result in a MissingMethodExcep

15.1.8 Unit Test Metaprogramming

If runtime metaprogramming needs to be done in a unit test it needs to be done early in the process before This should be done when the unit test class is being initialized. For a Spock based test this should be dc test this should be done in a method marked with `@BeforeClass`.

```
package myapp

import grails.test.mixin.*
import spock.lang.Specification

@TestFor(SomeController)
class SomeControllerSpec extends Specification {

    def setupSpec() {
        SomeClass.metaClass.someMethod = { ->
            // do something here
        }
    }

// …
}
```

```
package myapp

import grails.test.mixin.*
import org.junit.*

@TestFor(SomeController)
class SomeControllerTests {

@BeforeClass
    static void metaProgramController() {
        SomeClass.metaClass.someMethod = { ->
            // do something here
        }
    }

// ...

}
```

15.2 Integration Testing

Integration tests differ from unit tests in that you have full access to the Grails environment within the te
create-integration-test command:

```
$ grails create-integration-test Example
```

The above command will create a new integration test at the location `src/integration-test/groc`

Grails uses the test environment for integration tests and loads the application prior to the first test run. All

Transactions

Integration tests run inside a database transaction by default, which is rolled back at the end of the each tes
persisted to the database (which is shared across all tests). The default generated integration test template i

```
import grails.test.mixin.integration.Integration
import grails.transaction.*
import spock.lang.*

@Integration
@Rollback
class artifact.nameSpec extends Specification {

...

void "test something"() {
        expect:"fix me"
            true == false
    }
}
```

The `Rollback` annotation ensures that each test methods runs in a transaction that is rolled back. Gene
your tests depending on order or application state.

Using Spring's Rollback annotation

In Grails 3.0 tests rely on `grails.transaction.Rollback` annotation to bind the session in integra
and `setupSpec()` method in the test is run prior to the transaction starting hence you would see `No`
running integration test if `setup()` sets up data and persists them as shown in the below sample:

```
import grails.test.mixin.integration.Integration
import grails.transaction.*
import spock.lang.*

@Integration
@Rollback
class artifact.nameSpec extends Specification {

void setup() {
        // Below line would throw a Hibernate session not found error
        new Book(name: 'Grails in Action').save(flush: true)
    }

void "test something"() {
        expect:
        Book.count() == 1
    }
}
```

To make sure the setup logic runs within the transaction you have to move it to be called from the met
method shown below:

```
import grails.test.mixin.integration.Integration
import grails.transaction.*
import spock.lang.*

@Integration
@Rollback
class artifact.nameSpec extends Specification {

void setupData() {
        new Book(name: 'Grails in Action').save(flush: true)
    }

void "test something"() {
        given:
        setupData()

expect:
        Book.count() == 1
    }
}
```

Another approach could be to use Spring's [@Rollback](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/annotation/Rollback.html) instead.

```
import grails.test.mixin.integration.Integration
import org.springframework.test.annotation.Rollback
import spock.lang.*

@Integration
@Rollback
class artifact.nameSpec extends Specification {

void setup() {
        new Book(name: 'Grails in Action').save(flush: true)
    }

void "test something"() {
        expect:
        Book.count() == 1
    }
}
```

> It isn't possible to make `grails.transaction.Rollback` behave the same way as because `grails.transaction.Rollback` transforms the byte code of the class, elim (which Spring's version requires). This has the downside that you cannot implement it diff Spring does for testing).

DirtiesContext

If you do have a series of tests that will share state you can remove the `Rollback` and the last test annotation which will shutdown the environment and restart it fresh (note that this will have an impact on t

Autowiring

To obtain a reference to a bean you can use the Autowired annotation. For example:

```
…
import org.springframework.beans.factory.annotation.*

@Integration
@Rollback
class artifact.nameSpec extends Specification {

@Autowired
    ExampleService exampleService
    ...

void "Test example service"() {
        expect:
            exampleService.countExamples() == 0
    }
}
```

Testing Controllers

To integration test controllers it is recommended you use create-functional-test command to create a G functional testing for more information.

15.3 Functional Testing

Functional tests involve making HTTP requests against the running application and verifying the resultan from the integration phase in that the Grails application is now listening and responding to actual HTTI scenarios, such as making REST calls against a JSON API.

Grails by default ships with support for writing functional tests using the Geb framework. To `create-functional-test` command which will create a new functional test:

```
$ grails create-functional-test MyFunctional
```

The above command will create a new Spock spec called `MyFunctionalSpec.groovy` in the `src/i` test is annotated with the Integration annotation to indicate it is a integration test and extends the `GebSpe`

```
@Integration
class HomeSpec extends GebSpec {

def setup() {
    }

def cleanup() {
    }

void "Test the home page renders correctly"() {
        when:"The home page is visited"
            go '/'

then:"The title is correct"
            $('title').text() == "Welcome to Grails"
    }
}
```

When the test is run the application container will be loaded up in the background and you can send reques

Note that the application is only loaded once for the entire test run, so functional tests share the state of the

In addition the application is loaded in the JVM as the test, this means that the test has full access to the data services such as GORM to setup and cleanup test data.

The `Integration` annotation supports an optional `applicationClass` attribute which may be use functional test. The class must extend GrailsAutoConfiguration.

```
@Integration(applicationClass=com.demo.Application)
class HomeSpec extends GebSpec {

// ...

}
```

If the `applicationClass` is not specified then the test runtime environment will attempt to locate t problematic in multiproject builds where multiple application classes may be present.

16 Internationalization

Grails supports Internationalization (i18n) out of the box by leveraging the underlying Spring MVC intern to customize the text that appears in a view based on the user's Locale. To quote the javadoc for the Locale

> *A Locale object represents a specific geographical, political, or cultural region. An operation tha called locale-sensitive and uses the Locale to tailor information for the user. For example, di operation--the number should be formatted according to the customs/conventions of the user's native*

A Locale is made up of a language code and a country code. For example "en_US" is the code for US English.

16.1 Understanding Message Bundles

Now that you have an idea of locales, to use them in Grails you create message bundle file containing t Message bundles in Grails are located inside the `grails-app/i18n` directory and are simple Java prop

Each bundle starts with the name `messages` by convention and ends with the locale. Grails ships with 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 local simply creating a new properties file that ends with the locale you are interested. For example `messages_`

16.2 Changing Locales

By default the user locale is detected from the incoming `Accept-Language` header. However, you can 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

16.3 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 message:

```
my.localized.content=Hola, Me llamo John. Hoy es domingo.
```

Messages can also include arguments, for example:

```
<g:message code="my.localized.content" args="${ ['Juan', 'lunes'] }" />
```

The message declaration specifies positional parameters which are dynamically specified:

```
my.localized.content=Hola, Me llamo {0}. Hoy es {1}.
```

Reading Messages in Controllers and Tag Libraries

It's simple to read messages in a controller since you can invoke tags as methods:

```
def show() {
    def msg = message(code: "my.localized.content", args: ['Juan', 'lunes'])
}
```

The same technique can be used in tag libraries, but if your tag library uses a custom namespace then you r

```
def myTag = { attrs, body ->
    def msg = g.message(code: "my.localized.content", args: ['Juan', 'lunes'])
}
```

16.4 Scaffolding and i18n

Grails scaffolding templates for controllers and views are fully i18n-aware. The GSPs use the message ta
messages use i18n to resolve locale-specific messages.

The scaffolding includes locale specific labels for domain classes and domain fields. For example, if you h

```
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
key other than it's the convention used by the scaffolding.

17 Security

Grails is no more or less secure than Java Servlets. However, Java servlets (and hence Grails) are extremel overrun and malformed URL exploits due to the nature of the Java Virtual Machine underpinning the code

Web security problems typically occur due to developer naivety or mistakes, and there is a little Grail 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 escaping
4. Grails provides codecs to let you trivially escape data when rendered as HTML, JavaScript and URLs

17.1 Securing Against Attacks

SQL injection

Hibernate, which is the technology underlying GORM domain classes, automatically escapes data when However it is still possible to write bad dynamic HQL code that uses unchecked request parameters. Fo 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
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 fro
is also important to ensure that all data values rendered into views are escaped correctly. For example w
ensure that people cannot maliciously inject JavaScript or other HTML into data or tags viewed by others.

Grails 2.3 and above include special support for automatically encoded data placed into GSP pages. See th
prevention for further information.

You must also avoid the use of request parameters or data fields for determining the next URL to re
parameter for example to determine where to redirect a user to after a successful login, attackers can imitat
then redirect the user back to their own site once logged in, potentially allowing JavaScript code to then ex

Cross-site request forgery

CSRF involves unauthorized commands being transmitted from a user that a website trusts. A typical exan
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 your for
for more information on how to use it. An additional measure would be to not use remember-me cookies.

HTML/URL injection

This is where bad data is supplied such that when it is later used to create a link in a page, clicking it w
redirect to another site or alter request parameters.

HTML/URL injection is easily handled with the codecs supplied by Grails, and the tag libraries su
appropriate. If you create your own tags that generate URLs you will need to be mindful of doing this too.

Denial of service

Load balancers and other appliances are more likely to be useful here, but there are also issues relating t
created by an attacker to set the maximum value of a result set so that a query could exceed the memory li
solution here is to always sanitize request parameters before passing them to dynamic finders or other GOI

```
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 elsew
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 befor

Not doing this is "security through obscurity" which is inevitably breached, just like having a default passv

You must assume that every unprotected URL is publicly accessible one way or another.

17.2 Cross Site Scripting (XSS) Prevention

Cross Site Scripting (XSS) attacks are a common attack vector for web applications. They typically inv
form such that when that code is displayed, the browser does something nasty. It could be as simple as pop
The solution is to escape all untrusted user input when it is displayed in a page. For 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
relevant attribute values.

So what happens when you want to stop Grails from escaping some content? There are valid use cases for
it as-is, as long as that content is **trusted**. In such cases, you can tell Grails that the content is safe as should

```
<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 attack. Such an attack is less likely to succeed than would otherwise be the case, but developer of potential attack vectors and attempt to uncover vulnerabilities in the application during test an unsafe default, thereby increasing the risk of a vulnerability being introduced.

There are more details about the XSS in [OWASP - XSS prevention rules] and [OWASP - Types of Cross-
Reflected XSS] and [DOM based XSS]. [DOM based XSS prevention] is coming more important because of th
and Single Page Apps.

Grails codecs are mainly for preventing stored and reflected XSS type of attacks. Grails 2.4 includes HTM
based XSS attacks.

It's difficult to make a solution that works for everyone, and so Grails provides a lot of flexibility with rega
you to keep most of your application safe while switching off default escaping or changing the codec used

Configuration

It is recommended that you review the configuration of a newly created Grails application to garner an und

GSP features the ability to automatically HTML encode GSP expressions, and as of Grails 2.3 this is the
(found in `application.yml`) for a newly created Grails application can be seen below:

```
grails:
    views:
        gsp:
            encoding: UTF-8
            htmlcodec: xml # use xml escaping instead of HTML4 escaping
            codecs:
                expression: html # escapes values inside ${}
                scriptlets: html # escapes output from scriptlets in GSPs
                taglib: none # escapes output from taglibs
                staticparts: none # escapes output from static template parts
```

GSP features several codecs that it uses when writing the page to the response. The codecs are configured i

- `expression` - The expression codec is used to encode any code found within ${..} expressions. Th encoding.
- `scriptlet` - Used for output from GSP scriplets (<% %>, <%= %> blocks). The default for newly
- `taglib` - Used to encode output from GSP tag libraries. The default is `none` for new application author to define the encoding of a given tag and by specifying `none` Grails remains backwards comp
- `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 enabling t using existing plugins due to encoding being applied twice (once by the `html` codec and then again if the

Grails 2.3 includes double encoding prevention so that when an expression is evaluated, it will not encode `${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 inpu 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 have following configuration in your `application.groovy` will disable encoding for only the `foo` 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 the tag l

```
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 p method. Consider for example the "<g:javascript>"" tag which allows you to embed JavaScript code in t not HTML coding for the execution of the body of the tag (but not for the markup that is 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 f 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 can also make your application even more secure by configuring Grails to always encode all output at `filteringCodecForContentType` configuration in `application.groovy`:

```
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 r

```
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
    }
```

17.3 Encoding and Decoding Objects

Grails supports the concept of dynamic encode/decode methods. A set of standard codecs are bundl mechanism for developers to contribute their own codecs that will be recognized at runtime.

Codec Classes

A Grails codec class is one that may contain an encode closure, a decode closure or both. When a Gr dynamically loads codecs from the `grails-app/utils/` directory.

The framework looks under `grails-app/utils/` for class names that end with the convention `Code` ships with Grails is `HTMLCodec`.

If a codec contains an `encode` closure Grails will create a dynamic `encode` method and add that methoc the codec that defined the encode closure. For example, the `HTMLCodec` class defines an `encode` `encodeAsHTML`.

The `HTMLCodec` and `URLCodec` classes also define a `decode` closure, so Grails attaches those wit
respectively. Dynamic codec methods may be invoked from anywhere in a Grails application. For exan
property called 'description' which may contain special characters that must be escaped to be presented in
in a GSP is to encode the description property using the dynamic encode method as shown below:

```
${report.description.encodeAsHTML()}
```

Decoding is performed using `value.decodeHTML()` syntax.

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.grails.encoder;

public interface CodecLookup {
    public Encoder lookupEncoder(String codecName);
    public Decoder lookupDecoder(String codecName);
}
```

example of using `CodecLookup` and `Encoder` interface

```
import org.grails.encoder.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 HT
damaging the page layout. For example, given a value "Don't you know that 2 > 1?" you wouldn't be a
because the > will look like it closes a tag, which is especially bad if you render this data within an attribut

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 us values to avoid text with apostrophes affecting your page.

HTMLCodec defaults to HTML4 style escaping (legacy HTMLCodec implementation in Grails versions b

You can use plain XML escaping instead of HTML4 escaping by setting this config property in `applica`

```
grails.views.gsp.htmlcodec = 'xml'
```

XMLCodec

This codec performs XML escaping and unescaping. It escapes & , < , > , " , ' , \ , @ , ` , non breaki paragraph separator (\u2029).

HTMLJSCodec

This codec performs HTML and JS encoding. It is used for preventing some DOM-XSS vulnerabilities. Se [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 cr getting into the URL and changing its meaning, for example "Apple & Blackberry" is not going to wo 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 stri

```
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 syster
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 system

```
byte[] passwordHash = params.password.encodeAsMD5Bytes()
```

SHA1Codec

Uses the SHA1 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default syster
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 systen

```
byte[] passwordHash = params.password.encodeAsSHA1Bytes()
```

SHA256Codec

Uses the SHA256 algorithm to digest byte arrays or lists of integers, or the bytes of a string (in default
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
usage:

```
byte[] passwordHash = params.password.encodeAsSHA256Bytes()
```

Custom Codecs

Applications may define their own codecs and Grails will load them along with the standard codecs. `grails-app/utils/` directory and the class name must end with `Codec`. The codec may contain a s closure or both. The closure must accept a single argument which will be the object that the dynamic meth

```
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()}
```

17.4 Authentication

Grails has no default mechanism for authentication as it is possible to implement authentication in many c a simple authentication mechanism using interceptors. This is sufficient for simple use cases but it's h framework, for example by using the Spring Security or the Shiro plugin.

Interceptors let you apply authentication across all controllers or across a URI space. For example you c `grails-app/controllers/SecurityInterceptor.groovy` by running:

```
grails create-interceptor security
```

and implement your interception logic there:

```
class SecurityInterceptor {

SecurityInterceptor() {
        matchAll()
        .except(controller:'user', action:'login')
    }

boolean before() {
        if (!session.user && actionName != "login") {
            redirect(controller: "user", action: "login")
            return false
        }
        return true
    }

}
```

Here the interceptor intercepts execution *before* all actions except `login` are executed, and if there is nc 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"])
    }
}
```

17.5 Security Plugins

If you need more advanced functionality beyond simple authentication such as authorization, roles etc. the security plugins.

17.5.1 Spring Security

The Spring Security plugins are built on the [Spring Security](#) project which provides a flexible, e authentication and authorization schemes. The plugins are modular so you can install just the functionality Security plugins are the official security plugins for Grails and are actively maintained and supported.

There is a Core plugin which supports form-based authentication, encrypted/salted passwords, HTTP Bas plugins provide alternate functionality such as OpenID authentication, ACL support, single sign-on wi authentication, and a plugin providing user interface extensions and security workflows.

See the Core plugin page for basic information and the user guide for detailed information.

17.5.2 Shiro

Shiro is a Java POJO-oriented security framework that provides a default domain model that models realn extend a controller base class called `JsecAuthBase` in each controller you want secured and then pr 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.

18 Plugins

Grails is first and foremost a web application framework, but it is also a platform. By exposing a number from the command line interface to the runtime configuration engine, Grails can be customised to suit al you need to do is create a plugin.

Extending the platform may sound complicated, but plugins can range from trivially simple to incredibl application, you'll know how to create a plugin for sharing a data model or some static resources.

18.1 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-p` project called `example`.

In Grails 3.0 you should consider whether the plugin you create requires a web environment or whether th plugin does not require a web environment then use the "plugin" profile instead of the "web-plugin" profile

```
grails create-plugin [PLUGIN NAME] --profile=plugin
```

Make sure the plugin name does not contain more than one capital letter in a row, or it won't work. Camel

The structure of a Grails plugin is very nearly the same as a Grails application project's except that in the s package structure you will find a plugin descriptor class (a class that ends in "GrailsPlugin").

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, running in a browser right after creating it, you will receive a page not found erro 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 {
   …
}
```

All plugins must have this class under the src/main/groovy directory, otherwise they are not regarde about the plugin, and optionally various hooks into plugin extension points (covered shortly).

You can also provide additional information about your plugin using several special properties:

- title - short one-sentence description of your plugin
- grailsVersion - The version range of Grails that the plugin supports. eg. "1.2 > *" (indicating 1.2
- 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
- license - License of the plugin
- issueManagement - Issue Tracker of the plugin
- scm - Source code management location of the plugin

Here is an example from the Quartz Grails plugin:

```
class QuartzGrailsPlugin {
    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 `install` command:

```
grails install
```

This will install the plugin into your local Maven cache. Then to use the plugin within an applicatior `build.gradle` file:

```
compile "org.grails.plugins:quartz:0.1"
```

> In Grails 2.x plugins were packaged as ZIP files, however in Grails 3.x plugins are simple J the classpath of the IDE.

Plugins and Multi-Project Builds

If you wish to setup a plugin as part of a multi project build then follow these steps.

Step 1: Create the application and the plugin

Using the `grails` command create an application and a plugin:

```
$ grails create-app myapp
$ grails create-plugin myplugin
```

Step 2: Create a settings.gradle file

In the same directory create a `settings.gradle` file with the following contents:

```
include "myapp", "myplugin"
```

The directory structure should be as follows:

```
PROJECT_DIR
  - settings.gradle
  - myapp
    - build.gradle
  - myplugin
    - build.gradle
```

Step 3: Declare a project dependency on the plugin

Within the `build.gradle` of the application declare a dependency on the plugin within the `plugins` b

```
grails {
    plugins {
        compile project(':myplugin')
    }
}
```

> You can also declare the dependency within the `dependencies` block, however you will r
> you do this!

Step 4: Run the application

Now run the application using the `grails run-app` command from the root of the application direc
Gradle output:

```
$ cd myapp
$ grails run-app -verbose
```

You will notice from the Gradle output that plugins sources are built and placed on the classpath of your ap

```
:myplugin:compileAstJava UP-TO-DATE
:myplugin:compileAstGroovy UP-TO-DATE
:myplugin:processAstResources UP-TO-DATE
:myplugin:astClasses UP-TO-DATE
:myplugin:compileJava UP-TO-DATE
:myplugin:configScript UP-TO-DATE
:myplugin:compileGroovy
:myplugin:copyAssets UP-TO-DATE
:myplugin:copyCommands UP-TO-DATE
:myplugin:copyTemplates UP-TO-DATE
:myplugin:processResources
:myapp:compileJava UP-TO-DATE
:myapp:compileGroovy
:myapp:processResources UP-TO-DATE
:myapp:classes
:myapp:findMainClass
:myapp:bootRun
Grails application running at http://localhost:8080 in environment: development
```

Notes on excluded Artefacts

Although the create-plugin command creates certain files for you so that the plugin can be run as a Grai
when packaging a plugin. The following is a list of artefacts created, but not included by package-plugin:

- `grails-app/build.gradle` (although it is used to generate `dependencies.groovy`)
- `grails-app/conf/application.yml` (renamed to plugin.yml)
- `grails-app/conf/spring/resources.groovy`
- `grails-app/conf/logback.groovy`
- Everything within `/src/test/**`
- SCM management files within `**/.svn/**` and `**/CVS/**`

Customizing the plugin contents

When developing a plugin you may create test classes and sources that are used during the developme
exported to the application.

To exclude test sources you need to modify the `pluginExcludes` property of the plugin descri `build.gradle` file. For example say you have some classes under the `com.demo` package that ar packaged in the application. In your plugin descriptor you should exclude these:

```
// resources that should be loaded by the plugin once installed in the applicatio
  def pluginExcludes = [
    '**/com/demo/**'
  ]
```

And in your `build.gradle` you should exclude the compiled classes from the JAR file:

```
jar {
  exclude "com/demo/**/**"
}
```

Inline Plugins in Grails 3.0

In Grails 2.x it was possible to specify inline plugins in `BuildConfig`, in Grails 3.x this functionality h feature.

To set up a multi project build create an appliation and a plugin in a parent directory:

```
$ grails create-app myapp
$ grails create-plugin myplugin
```

Then create a `settings.gradle` file in the parent directory specifying the location of your application

```
include 'myapp', 'myplugin'
```

Finally add a dependency in your application's `build.gradle` on the plugin:

```
compile project(':myplugin')
```

Using this technique you have achieved the equivalent of inline plugins from Grails 2.x.

18.2 Plugin Repositories

Distributing Plugins in the Grails Central Plugin Repository

The preferred way to distribute plugin is to publish to the official Grails Central Plugin Repository. This 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, you'll f account on the plugin portal website.

18.3 Providing Basic Artefacts

Add Command Line Commands

A plugin can add new commands to the Grails 3.0 interactive shell in one of two ways. First, using the script which will become available to the application. The `create-script` command will create the scr

```
+ src/main/scripts     <-- additional scripts here
 + grails-app
      + controllers
      + services
      + etc.
```

Code generation scripts can be used to create artefacts within the project tree and automate interactions wit

If you want to create a new shell command that interacts with a loaded Grails application instance command:

```
$ grails create-command MyExampleCommand
```

This will create a file called `grails-app/commands/PACKAGE_PATH/MyExampleCommand.gr`

```
import grails.dev.commands.*

class MyExampleCommand implements ApplicationCommand {

boolean handle(ExecutionContext ctx) {
      println "Hello World"
      return true
  }
}
```

An `ApplicationCommand` has access to the `GrailsApplication` instance and is subject to autowi

For each `ApplicationCommand` present Grails will create a shell command and a Gradle task to invo example you can invoke the `MyExampleCommand` class using either:

```
$ grails my-example
```

Or

```
$ gradle myExample
```

The Grails version is all lower case hyphen separated and excludes the "Command" suffix.

The main difference between code generation scripts and `ApplicationCommand` instances is that th state and hence can be used to perform tasks that interactive with the database, call into GORM etc.

In Grails 2.x Gant scripts could be used to perform both these tasks, in Grails 3.x code generation and int cleanly separated.

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.

```
+ grails-app
      + controllers  <-- additional controllers here
      + services <-- additional services here
      + etc.  <-- additional XXX here
```

Providing Views, Templates and View resolution

When a plugin provides a controller it may also provide default views to be rendered. This is an excelle plugins. Grails' view resolution mechanism will first look for the view in the application it is installed into within the plugin. This means that you can override views provided by a plugin by creating `grails-app/views` directory.

For example, consider a controller called `BookController` that's provided by an 'amazon' plugin. If the look for a view called `grails-app/views/book/list.gsp` then if that fails it will look for the sam

However if the view uses templates that are also provided by the plugin then the following syntax may be r

```
<g:render template="fooTemplate" plugin="amazon"/>
```

Note the usage of the `plugin` attribute, which contains the name of the plugin where the template resides the template relative to the application.

Excluded Artefacts

By default Grails excludes the following files during the packaging process:

- `grails-app/conf/logback.groovy`
- `grails-app/conf/application.yml` (renamed to `plugin.yml`)
- `grails-app/conf/spring/resources.groovy`
- Everything within `/src/test/**`
- SCM management files within `**/.svn/**` and `**/CVS/**`

In addition, the default `UrlMappings.groovy` file is excluded to avoid naming conflicts, however you a different name which **will** be included. For example a file called `grails-app/controllers/Blog`

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

18.4 Evaluating Conventions

Before looking at providing runtime configuration based on conventions you first need to understand ho Every plugin has an implicit `application` variable which is an instance of the [GrailsApplication](#) interfa

The `GrailsApplication` interface provides methods to evaluate the conventions within the projec classes within your application.

Artifacts implement the [GrailsClass](#) interface, which represents a Grails resource such as a control `GrailsClass` instances you can do:

```
for (grailsClass in application.allClasses) {
    println grailsClass.name
}
```

`GrailsApplication` has a few "magic" properties to narrow the type of artefact you are interested in.

```
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 particul `application.getControllerClass("PersonController")`
- `is*Class` - Returns `true` if the given class is of the give `application.isControllerClass(PersonController)`

The `GrailsClass` interface has a number of useful methods that let you further evaluate and work with

- `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 pa
- `getPropertyName` - Returns the name of the class as a property name
- `getLogicalPropertyName` - Returns the logical property name of the class in the application wi
- `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.

18.5 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 overriding the `doWithSpring` method from the P additional beans. For example the following snippet is from one of the core Grails plugins that provides i1

```
import org.springframework.web.servlet.i18n.CookieLocaleResolver
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor
import org.springframework.context.support.ReloadableResourceBundleMessageSource
import grails.plugins.*

class I18nGrailsPlugin extends Plugin {

def version = "0.1"

Closure 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 Loca
Bean Builder syntax to do so.

Customizing the Servlet Environment

In previous versions of Grails it was possible to dynamically modify the generated `web.xml`. In Grails 3.
to programmatically modify the `web.xml` file anymore.

However, it is possible to perform the most commons tasks of modifying the Servlet environment in Grails

Adding New Servlets

If you want to add a new Servlet instance the simplest way is simply to define a new Spring bean in the `dc`

```
Closure doWithSpring() {{->
  myServlet(MyServlet)
}}
```

If you need to customize the servlet you can use Spring Boot's ServletRegistrationBean:

```
Closure doWithSpring() {{->
  myServlet(ServletRegistrationBean, new MyServlet(), "/myServlet/*") {
    loadOnStartup = 2
  }
}}
```

Adding New Servlet Filters

Just like Servlets, the simplest way to configure a new filter is to simply define a Spring bean:

```
Closure doWithSpring() {{->
  myFilter(MyFilter)
}}
```

However, if you want to control the order of filter registrations you will need to use Spring Boot's FilterRe

```
myFilter(FilterRegistrationBean) {
    filter = bean(MyFilter)
    urlPatterns = ['/*']
    order = Ordered.HIGHEST_PRECEDENCE
}
```

> Grails' internal registered filters (`GrailsWebRequestFilter`, `HiddenHttpMethod` incrementing `HIGHEST_PRECEDENCE` by 10 thus allowing several filters to be inserted befo

Doing Post Initialisation Configuration

Sometimes it is useful to be able do some runtime configuration after the Spring ApplicationContext `doWithApplicationContext` closure property.

```
class SimplePlugin extends Plugin{

def name = "simple"
    def version = "1.1"

@Override
    void doWithApplicationContext() {
        def sessionFactory = applicationContext.sessionFactory
        // do something here with session factory
    }
}
```

18.6 Adding Methods at Compile Time

Grails 3.0 makes it easy to add new traits to existing artefact types from a plugin. For example say you w
controllers. This can be done by defining a trait in `src/main/groovy`:

```
package myplugin

@Enhances("Controller")
trait DateTrait {
  Date currentDate() {
    return new Date()
  }
}
```

The `@Enhances` annotation defines the types of artefacts that the trait should be applied to.

As an alternative to using the `@Enhances` annotation above, you can implement a [TraitInjector](http://docs.grails.org/3.3.8/api/grails/compiler/traits/TraitInjector.html) to tell G
into at compile time:

```
package myplugin

@CompileStatic
class ControllerTraitInjector implements TraitInjector {

@Override
    Class getTrait() {
        DateTrait
    }

@Override
    String[] getArtefactTypes() {
        ['Controller'] as String[]
    }
}
```

The above `TraitInjector` will add the `DateTrait` to all controllers. The `getArtefactTypes` m should be applied to.

18.7 Adding Dynamic Methods at Runtime

The Basics

Grails plugins let you register dynamic methods with any Grails-managed or other class at runtime. This method.

> Note that Grails 3.x features newer features such as traits that are usable from code compiled recommended that dynamic behavior is only added for cases that are not possible with traits.

```
class ExamplePlugin extends Plugin {
    void doWithDynamicMethods() {
        for (controllerClass in grailsApplication.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' Met `myNewMethod` to each controller. If you know beforehand the class you wish the add a method to you ca

For example we can add a new method `swapCase` to `java.lang.String`:

```
class ExamplePlugin extends Plugin  {

@Override
    void doWithDynamicMethods() {
        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. T within it. For example if you were implementing a method to interact with Hibernate you could use the Se a HibernateTemplate:

```
import org.springframework.orm.hibernate3.HibernateTemplate
class ExampleHibernatePlugin extends Plugin{
void doWithDynamicMethods() {
for (domainClass in grailsApplication.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 impl use the application context to wire dependencies into your object at runtime:

```
class MyConstructorPlugin {
void doWithDynamicMethods()
         for (domainClass in grailsApplication.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!

18.8 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 application state at runtime. For example, consider this simplified snippet from the Grails ServicesPlu

```
class ServicesGrailsPlugin extends Plugin {
    …
    def watchedResources = "file:./grails-app/services/*Service.groovy"

…
    void onChange( Map<String, Object> event) {
        if (event.source) {
            def serviceClass = grailsApplication.addServiceClass(event.source)
            def serviceName = "${serviceClass.propertyName}"
            beans {
                "$serviceName"(serviceClass.getClazz()) { bean ->
                    bean.autowire =  true
                }
            }
        }
    }
}
```

First it defines `watchedResources` as either a String or a List of strings that contain either the referer
watched resources specify a Groovy file, when it is changed it will automatically be reloaded and pass
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 "S
re-registered with the `ApplicationContext` when one of the service classes changes.

Influencing Other Plugins

In addition to reacting to changes, sometimes a plugin needs to "influence" another.

Take for example the Services and Controllers plugins. When a service is reloaded, unless you reload the
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
then reload its influenced plugins. For example consider this snippet from the `ServicesGrailsPlugi`

```
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 reso
property:

```
def observe = ["controllers"]
```

In this case when a controller is changed you will also receive the event chained from the controllers plugi

It is also possible for a plugin to observe all loaded plugins by using a wildcard:

```
def observe = ["*"]
```

The Logging plugin does exactly this so that it can add the `log` property back to *any* artefact that changes

18.9 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
first is called `dependsOn`. For example, take a look at this snippet from the Hibernate plugin:

```
class HibernateGrailsPlugin {
def version = "1.0"
def dependsOn = [dataSource: "1.0",
                     domainClass: "1.0",
                     i18n: "1.0",
                     core: "1.0"]
}
```

The Hibernate plugin is dependent on the presence of four plugins: the `dataSource`, `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

```
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 an example the expression "1.0 > 1.1" would match any of the following versions:

- 1.1
- 1.0
- 1.0.1
- 1.0.3-SNAPSHOT
- 1.1-BETA2

Controlling Load Order

Using `dependsOn` establishes a "hard" dependency in that if the dependency is not resolved, the plugin v 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 loade the other plugin, for example the Hibernate plugin has this code in its `doWithSpring` closure:

```
if (manager?.hasGrailsPlugin("controllers")) {
    openSessionInViewInterceptor(OpenSessionInViewInterceptor) {
        flushMode = HibernateAccessor.FLUSH_MANUAL
        sessionFactory = sessionFactory
    }
    grailsUrlHandlerMapping.interceptors << openSessionInViewInterceptor
}
```

Here the Hibernate plugin will only register an `OpenSessionInViewInterceptor` if the `contro` variable is an instance of the GrailsPluginManager interface and it provides methods to interact with other

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 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 pac from the 'war' phase. This allows `development-only` plugins to not be packaged for production use.

The full list of available scopes are defined by the enum BuildScope, but here's a summary:

- `test` - when running tests
- `functional-test` - when running functional tests
- `run` - for run-app and run-war
- `war` - when packaging the application as a WAR file
- `all` - plugin applies to all scopes (default)

Both properties can be one of:

- a string - a sole inclusion
- a list - a list of environments or scopes to include
- a map - for full control, with 'includes' and/or 'excludes' keys that can have string or list values

For example,

```
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.

18.10 The Artefact API

You should by now understand that Grails has the concept of artefacts: special types of classes that it knc
Groovy and Java classes, for example by enhancing them with extra properties and methods. Examples of
What you may not be aware of is that Grails allows application and plugin developers access to the unde
you can find out what artefacts are available and even enhance them yourself. You can even provide your

18.10.1 Asking About Available Artefacts

As a plugin developer, it can be important for you to find out about what domain classes, controllers
application. For example, the [Searchable plugin](#) needs to know what domain classes exist so it can che
index the appropriate ones. So how does it do it? The answer lies with the `grailsApplication`
available automatically in controllers and GSPs and can be [injected](#) everywhere else.

The `grailsApplication` object has several important properties and methods for querying artefacts.
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, t particularly useful properties and methods, including one for the underlying `Class`:

- `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 `MyGrea`
- `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 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. 'MyGreatControlle particular type of artefact. For example, you can use `grailsApplication.isControllerClass(` check whether `MyGreatController` is in fact a controller.

18.10.2 Adding Your Own Artefact Types

Plugins can easily provide their own artefacts so that they can easily find out what implementations are av to do is create an `ArtefactHandler` implementation and register it in your main plugin class:

```
class MyGrailsPlugin {
    def artefacts = [ org.somewhere.MyArtefactHandler ]
    …
}
```

The `artefacts` list can contain either handler classes (as above) or instances of handlers.

So, what does an artefact handler look like? Well, put simply it is an implementation of the ArtefactHand skeleton implementation that can readily be extended: ArtefactHandlerAdapter.

In addition to the handler itself, every new artefact needs a corresponding wrapper class that implements G available such as AbstractInjectableGrailsClass, which is particularly useful as it turns your artefact i 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.

19 Grails and Spring

This section is for advanced users and those who are interested in how Grails integrates with and builds
plugin developers considering doing runtime configuration Grails.

19.1 The Underpinnings of Grails

Grails is actually a Spring MVC application in disguise. Spring MVC is the Spring framework's built-i
Spring MVC suffers from some of the same difficulties as frameworks like Struts in terms of its ease of
was, for Grails, the perfect framework to build another framework on top of.

Grails leverages Spring MVC in the following areas:

- Basic controller logic - Grails subclasses Spring's DispatcherServlet and uses it to delegate to Grails c
- Data Binding and Validation - Grails' validation and data binding capabilities are built on those provic
- 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 const

- Grails constructs a parent `ApplicationContext` from the `web-app/WEB-INF/a`
`ApplicationContext` configures the GrailsApplication instance and the GrailsPluginManager.
- Using this `ApplicationContext` as a parent Grails' analyses the conventions with the `Grails/`
`ApplicationContext` that is used as the root `ApplicationContext` of the web application

Configured Spring Beans

Most of Grails' configuration happens at runtime. Each plugin may configure Spring beans that are reg
reference as to which beans are configured, refer to the reference guide which describes each of the Grails

19.2 Configuring Additional Beans

Using the Spring Bean DSL

You can easily register new (or override existing) beans by configuring them in `grails-app/conf/s`
Grails Spring DSL. Beans are defined inside a `beans` property (a Closure):

```
beans = {
    // beans here
}
```

As a simple example you can configure a bean with the following syntax:

```
import my.company.MyBeanImpl

beans = {
    myBean(MyBeanImpl) {
        someProperty = 42
        otherProperty = "blue"
    }
}
```

Once configured, the bean can be auto-wired into Grails artifacts and other classes that su
`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 used t
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 regist
> plugin, your bean will replace the previous registration. This is a convenient way to customi
> to editing plugin code or other approaches that would affect maintainability.

Using XML

Beans can also be configured using a `grails-app/conf/spring/resources.xml`. In earlier
generated for you by the `run-app` script, but the DSL in `resources.groovy` is the preferred approa
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

```
<bean id="myBean" class="my.company.MyBeanImpl">
    <property name="someProperty" value="42" />
    <property name="otherProperty" value="blue" />
</bean>
```

Like the other bean it can be auto-wired into any class that supports dependency injection:

```
class ExampleController {

def myBean
}
```

Referencing Existing Beans

Beans declared in `resources.groovy` or `resources.xml` can reference other beans by conventi
class its Spring bean name would be `bookService`, so your bean would reference it like this in the DSL

```
beans = {
    myBean(MyBeanImpl) {
        someProperty = 42
        otherProperty = "blue"
        bookService = ref("bookService")
    }
}
```

or like this in 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 Groovy

```
package my.company

class MyBeanImpl {
    Integer someProperty
    String otherProperty
    BookService bookService // or just "def bookService"
}
```

or in Java like this:

```
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 referenc in place when the final application context configuration occurs, everything will be resolved correctly.

For a full reference of the available beans see the plugin reference in the reference guide.

19.3 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 t programmatic XML creation which is both error prone and verbose. Grails' BeanBuilder changes all that together components at runtime, allowing you to adapt the logic based on system properties or environmen

This enables the code to adapt to its environment and avoids unnecessary duplication of code (having dif production environments)

The BeanBuilder class

Grails provides a grails.spring.BeanBuilder class that uses dynamic Groovy to construct bean definitions. T

```
import org.apache.commons.dbcp.BasicDataSource
import org.grails.orm.hibernate.ConfigurableLocalSessionFactoryBean
import org.springframework.context.ApplicationContext
import grails.spring.BeanBuilder

def bb = new BeanBuilder()

bb.beans {

dataSource(BasicDataSource) {
        driverClassName = "org.h2.Driver"
        url = "jdbc:h2:mem:grailsDB"
        username = "sa"
        password = ""
    }

sessionFactory(ConfigurableLocalSessionFactoryBean) {
        dataSource = ref('dataSource')
        hibernateProperties = ["hibernate.hbm2ddl.auto": "create-drop",
                               "hibernate.show_sql":     "true"]
    }
}

ApplicationContext appContext = bb.createApplicationContext()
```

> ⚠ Within plugins and the grails-app/conf/spring/resources.groovy file you don't need to `BeanBuilder`. Instead the DSL is implicitly available inside the `doWithSpring` and `bea`

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 bea the bean's class, whilst the last argument is a block. Within the body of the block you can set properties on

Bean references are resolved automatically using the name of the bean. This can be seen in the example ab 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 re 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>
      grails.web.servlet.context.GrailsWebApplicationContext
    </param-value>
</context-param>
```

Then create a `/WEB-INF/applicationContext.groovy` file that does the rest:

```
import org.apache.commons.dbcp.BasicDataSource

beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.h2.Driver"
        url = "jdbc:h2:mem:grailsDB"
        username = "sa"
        password = ""
    }
}
```

Loading Bean Definitions from the File System

You can use the `BeanBuilder` class to load external Groovy scripts that define beans using the same pat

```
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.groo` example script can be seen below:

```
import org.apache.commons.dbcp.BasicDataSource
import org.grails.orm.hibernate.ConfigurableLocalSessionFactoryBean

beans {

dataSource(BasicDataSource) {
        driverClassName = "org.h2.Driver"
        url = "jdbc:h2:mem:grailsDB"
        username = "sa"
        password = ""
    }

sessionFactory(ConfigurableLocalSessionFactoryBean) {
        dataSource = dataSource
        hibernateProperties = ["hibernate.hbm2ddl.auto": "create-drop",
                               "hibernate.show_sql":     "true"]
    }
}
```

Adding Variables to the Binding (Context)

If you're loading beans from a script you can set the binding to use by creating a Groovy `Binding`:

```
def binding = new Binding()
binding.maxSize = 10000
binding.productGroup = 'finance'

def bb = new BeanBuilder()
bb.binding = binding
bb.loadBeans("classpath:*SpringBeans.groovy")

def ctx = bb.createApplicationContext()
```

Then you can access the `maxSize` and `productGroup` properties in your DSL files.

19.4 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 co 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
case the bean has no Class argument and instead you must pass the name of the factory bean to the bean de

```
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
syntax:

```
bb.beans {

myFactory(ExampleFactoryBean) {
        someProperty = [1, 2, 3]
    }

myBean(myFactory: "getInstance") {
        name = "blah"
    }
}
```

Here the `getInstance` method on the `ExampleFactoryBean` bean will be called to create the `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
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
just as well with a name that is generated programmatically based on configuration, system properties, etc.

Furthermore, because sometimes bean names are not known until runtime you may need to reference ther
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
from a parent `ApplicationContext` that is provided in the constructor of the `BeanBuilder`:

```
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 ref you can omit the type and just use the specified bean definition instead to setup the factory:

```
bb.beans {

personFactory(PersonFactory)

marge(Person) {
        name = "Marge"
        husband = { bean ->
            bean.factoryBean = "personFactory"
            bean.factoryMethod = "newInstance"
            name = "Homer"
            age = 45
            props = [overweight: true, height: "1.8m"]
        }
        children = [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 tl
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

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 ar 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 c 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 on you also get full access to all of the powerful AOP support in Spring from BeanBuilder. For example giver

```
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)"
        }
    }
}
```

19.5 Property Placeholder Configuration

Grails supports the notion of property placeholder configuration through an extended version of Spring's P

Settings defined in either ConfigSlurper scripts or Java properties files can be used as place `grails-app/conf/spring/resources.xml` and `grails-app/conf/spring/resource` entries in `grails-app/conf/application.groovy` (or an externalized config):

```
database.driver="com.mysql.jdbc.Driver"
database.dbname="mysql:mydb"
```

You can then specify placeholders in `resources.xml` as follows using the familiar ${..} syntax:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
        <value>${database.driver}</value>
    </property>
    <property name="url">
        <value>jdbc:${database.dbname}</value>
    </property>
 </bean>
```

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.

19.6 Property Override Configuration

Grails supports setting of bean properties via 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
```

20 Grails and Hibernate

If GORM (Grails Object Relational Mapping) is not flexible enough for your liking you can alternatively
with XML mapping files or JPA annotations. You will be able to map Grails domain classes onto a
flexibility in the creation of your database schema. Best of all, you will still be able to call all of the dyn
GORM!

20.1 Using Hibernate XML Mapping Files

Mapping your domain classes with XML is pretty straightforward. Simply create a `hibernate.cfg.x`
directory, either manually or with the create-hibernate-cfg-xml command, that contains the following:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Example mapping file inclusion -->
        <mapping resource="org.example.Book.hbm.xml"/>
        …
    </session-factory>
</hibernate-configuration>
```

The individual mapping files, like 'org.example.Book.hbm.xml' in the above example, also go into the `gra`
map domain classes with XML, check out the Hibernate manual.

If the default location of the `hibernate.cfg.xml` file doesn't suit you, you can change i
`grails-app/conf/application.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 Hibernate into `grails-app/conf` and either put the Java files in `src/java` or the classes in the project's `lib` JAR. You still need the `hibernate.cfg.xml` though!

20.2 Mapping with Hibernate Annotations

To map a domain class with annotations, create a new class in `src/java` and use the annotations defin this see the [Hibernate Annotations Docs]():

```
package com.books;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Book {
    private Long id;
    private String title;
    private String description;
    private Date date;

@Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

public void setId(Long id) {
        this.id = id;
    }

public String getTitle() {
        return title;
    }

public void setTitle(String title) {
        this.title = title;
    }

public String getDescription() {
        return description;
    }

public void setDescription(String description) {
        this.description = description;
    }
}
```

Then register the class with the Hibernate `sessionFactory` by adding relevant entries to the `grails` follows:

```
<!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
section on [Scaffolding](#).

20.3 Adding Constraints

You can still use GORM validation even if you use a Java domain model. Grails lets you define constra
directory. The script must be in a directory that matches the package of the corresponding domain class a
example, if you had a domain class `org.example.Book`, then y
`src/java/org/example/BookConstraints.groovy`.

Add a standard GORM `constraints` block to the script:

```
constraints = {
    title blank: false
    author blank: false
}
```

Once this is in place you can validate instances of your domain class!

21 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

The way for an application to express a dependency on the scaffolding plugin is by including the following

```
dependencies {
// ...
compile "org.grails.plugins:scaffolding"
// ...
}
```

Dynamic Scaffolding

The simplest way to get started with scaffolding is to enable it by setting the `scaffold` property in the co

```
class BookController {
    static scaffold = Book  // Or any other domain class such as "Author", "Publi
}
```

With this configured, when you start your application the actions and views will be autogenerated at r 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/book` in a

Note: The old alternative of defining `scaffold` property:

```
class BookController {
    static scaffold = true
}
```

is no longer supported above Grails 3.0.

If you prefer to keep your domain model in Java and [mapped with Hibernate] you can still use scaffolding, 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 'rows' no effect.
>
> Also, the standard scaffold views expect model variables of the form `<propertyName>In` and `<propertyName>Instance` for single instances. It's tempting to use properties like won't work.

Static Scaffolding

Grails lets you generate a controller and the views used to create the above interface from the command lir

```
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 incl

```
grails generate-all com.bookstore.Book
```

Customizing the Generated Views

The views adapt to Validation constraints. For example you can change the order that fields appear in the
the builder:

```
def constraints = {
    title()
    releaseDate()
}
```

You can also get the generator to generate lists instead of text inputs if you use the `inList` constraint:

```
def constraints = {
    title()
    category(inList: ["Fiction", "Non-fiction", "Biography"])
    releaseDate()
}
```

Or if you use the `range` constraint on a number:

```
def constraints = {
    age(range:18..65)
}
```

Restricting the size with a constraint also effects how many characters can be entered in the d view:

```
def constraints = {
    name(size:0..30)
}
```

Customizing the Scaffolding templates

The templates used by Grails to generate the controller and views can be customized by installing the temp

22 Deployment

Grails applications can be deployed in a number of ways, each of which has its pros and cons.

22.1 Standalone

"grails run-app"

You should be very familiar with this approach by now, since it is the most common method of running a embedded Tomcat server is launched that loads the web application from the development sources, thus a files.

You can also deploy to production this way using:

```
grails prod run-app
```

Runnable WAR or JAR file

Another way to deploy in Grails 3.0 or above is to use the new support for runnable JAR or WAR fi `package`:

```
grails package
```

You can then run either the WAR file or the JAR using your Java installation:

```
java -Dgrails.env=prod -jar build/libs/mywar-0.1.war    (or .jar)
```

22.2 Container Deployment (e.g. Tomcat)

Grails apps can be deployed to a Servlet Container or Application Server.

WAR file

A common approach to Grails application deployment in production is to deploy to an existing Servl multiple applications to be deployed on the same port with different paths.

Creating a WAR file is as simple as executing the [war](war) command:

```
grails war
```

This will produce a WAR file that can be deployed to a container, in the `build/libs` directory.

Note that by default Grails will include an embeddable version of Tomcat inside the WAR file so that it cause problems if you deploy to a different version of Tomcat. If you don't intend to use the embedded co Tomcat dependencies to `provided` prior to deploying to your production container in `build.gradle`:

```
provided "org.springframework.boot:spring-boot-starter-tomcat"
```

Application servers

Ideally you should be able to simply drop a WAR file created by Grails into any application server and it rarely ever this simple. The Grails website contains a list of application servers that Grails has been tested get a Grails WAR file working.

22.3 Deployment Configuration Tasks

Setting up HTTPS and SSL certificates for standalone deployment

To configure an SSL certificate and to listen on an HTTPS port instead of HTTP, add properties like these

```
server:
    port: 8443                                          # The port to listen o
    ssl:
        enabled: true                                   # Activate HTTPS mode
        key-store: <the-location-of-your-keystore>      # e.g. /etc/tomcat7/ke
        key-store-password: <your-key-store-password>   # e.g. changeit
        key-alias: <your-key-alias>                     # e.g. tomcat
        key-password: <usually-the-same-as-your-key-store-password>
```

These settings control the embedded Tomcat container for a production deployment. Alternatively, the pr Example: `-Dserver.ssl.enabled=true -Dserver.ssl.key-store=/path/to/keysto`

> ⚠ Configuration of both an HTTP and HTTPS connector via application properties is not suppor
> then you'll need to configure one of them programmatically. (More information on how to
> how-to guide below.)

There are other relevant settings. Further reference:

- Spring Boot: How to configure SSL on embedded servlet containers
- Spring Boot: Common Application Properties

23 Contributing to Grails

Grails is an open source project with an active community and we rely heavily on that community to hel ways in which people can contribute to Grails. One of these is by writing useful plugins and making them some of the other options.

23.1 Report Issues in Github's issue tracker

Grails uses Github to track issues in the core framework. Similarly for its documentation there is a separat particular feature added, these are the places to start. You'll need to create a (free) github account in ord existing one in either of these.

When submitting issues, please provide as much information as possible and in the case of bugs, make sur and various plugins you are using. Other environment details - OS version, JDK, Gradle etc. should also be be dealt with if you upload a reproducible sample application on a github repository and provide a link in t

Reviewing issues

There are quite a few old issues in github, some of which may no longer be valid. The core team c contribution that you can make is to verify one or two issues occasionally.

Which issues need verification? Going to the issue tracker will display all issues that haven't been resolved

Once you've verified an issue, simply add a short comment explaining what you found. Be sure to metion y

23.2 Build From Source and Run Tests

If you're interested in contributing fixes and features to any part of grails, you will have to learn how to ge with your own applications. Before you start, make sure you have:

- A JDK (7 or above)
- A git client

Once you have all the pre-requisite packages installed, the next step is to download the Grails sourc repositories owned by the "grails" GitHub user. This is a simple case of cloning the repository you're intere 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 installation from the source.

Creating a Grails installation

If you look at the project structure, you'll see that it doesn't look much like a standard `GRAILS_HOME` ins 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` instal
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
path. When you next type `grails` command to run, you'll be using the version you just built.

If you are using SDKMAN then that can also be used to work with this local installation via the following:

```
sdk install grails dev /path/to/grails-core
```

Now you will have a dev version in your local which you can use to test your features.

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 comman
`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" target

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 fe

- Add the springloaded-core JAR file in $GRAILS_HOME/lib/org.springsource.springloaded/springloa
- 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 ge cleanEclipse eclipse" in that directory and edit the .classpath file again (remove the line "<classpath possible empty "scripts" directory under grails-scripts if you are not able to add the linked folder.
- Do a clean build for the whole workspace.
- To use Eclipse GIT scm team provider: Select all projects (except "Servers") in the navigation and r projects"). Choose "Git". Then check "Use or create repository in parent folder of project" and click ".
- Get the recommended code style settings from the mailing list thread (final style not decided yet, cu file to STS in Window->Preferences->Java->Code Style->Formatter->Import . Grails code uses space

Debugging Grails or a Grails application

To enable debugging, run:

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

By default Grails forks a JVM to run the application in. The `-debug-jvm` argument causes the debugge to instead attach the debugger to the build system which is going to fork the JVM use the `-debug` option:

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

23.3 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 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 reposito]()

What follows are some guidelines to help ensure that your pull requests are speedily dealt with and provid your life easier!

Make sure your fork is up to date

Making changes to outdated sources is not a good idea. Someone else may have already made the change.

```
git pull upstream master
```

Create a local branch for your changes

Your life will be greatly simplified if you create a local branch to make your changes on. For example, as locally, execute

```
git checkout -b issue_123
```

This will create a new local branch called "issue_123" based off the "master" branch. Of course, you can idea would be to reference the GitHub issue number that the change is relevant to. Each Pull Request shou

Create Github issues for non-trivial changes

For any non-trivial changes, raise an issue on github if one doesn't already exist. That helps us keep trac
Grails.

Include github issue ID in commit messages

This may not seem particularly important, but having a github issue ID in a commit message means that v
made. Include the ID in any and all commits that relate to that issue. If a commit isn't related to an issue, th

Make sure your fork is up to date again and rebase

Since the core developers must merge your commits into the main repository, it makes life much easier
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 pull
the local "issue_123" branch but not on "master". The first step involves pulling any changes from the mai
fetched and merged:

```
git checkout master
git pull upstream master
```

This should complete without any problems or conflicts. Next, rebase your local branch against the now up

```
git checkout issue_123
git rebase master
```

What this does is rearrange the commits such that all of your changes come after the most recent one in
deck rather than shuffling them into the pack.

Push your branch to GitHub and send Pull Request

Finally, you must push your changes to your fork on GitHub, otherwise the core developers won't be able t

```
git push origin issue_123
```

> ⚠ You should not merge your branch to your forks master. If the Pull Request is not accepted, y
> sync with upstream forever.

You're now ready to send the pull request from the GitHub user interface.

Say what your pull request is for

A pull request can contain any number of commits and it may be related to any number of issues. In the pu issues that the request relates to. Also give a brief description of the work you have done, such as: "I re custom number editors. Fixes #xxxx".

23.4 Submit Patches to Grails Documentation

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 r 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

```
./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 `grails.home` property, then the build will fetch the Grails source - a download of 10s of megabytes. It take a while too.

Additionally you can create a ~/.gradle/gradle.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 c containing the reference material. To view the user guide, simply open `build/docs/index.html`.

Publishing

The publishing system for the user guide is the same as [the one for Grails projects](). You write your chapte then converted to HTML for the final guide. Each chapter is a top-level gdoc file in the `src/<lang>/g` go into directories with the same name as the chapter gdoc but without the suffix.

The structure of the user guide is defined in the `src/<lang>/guide/toc.yml` file, which (language-specific) section titles. If you add or remove a gdoc file, you must update the TOC as well!

The `src/<lang>/ref` directory contains the source for the reference sidebar. Each directory is the nam Hence the directories need different names for the different languages. Inside the directories go the gd methods, commands, properties or whatever that the files describe.

Translations

This project can host multiple translations of the user guide, with `src/en` being the main one. To a directory under `src` and copy into it all the files under `src/en`. The build will take care of the rest.

Once you have a copy of the original guide, you can use the `{hidden}` macro to wrap the English tex This makes it easier to compare changes to the English guide against your translation. For example:

```
{hidden}
When you create a Grails application with the [create-app|commandLine] command,
Grails doesn't automatically create an Ant build.xml file but you can generate
one with the [integrate-with|commandLine] command:
{hidden}

Quando crias uma 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 bits of your translation need updating. On top of that, the `{hidden}` macro ensures that the text inside can display it by adding this URL as a bookmark: `javascript:toggleHidden();` (requires you later).

Even better, you can use the `left_to_do.groovy` script in the root of the project to see what still need

```
./left_to_do.groovy es
```

This will then print out a recursive diff of the given translation against the reference English user gui changed since being translated will *not* appear in the diff output. In other words, all you will see is conten has changed since it was translated. Note that `{code}` blocks are ignored, so you *don't* need to include the

To provide translations for the headers, such as the user guide title and subtitle, just add language specific so:

```
es.title=El Grails Framework
es.subtitle=...
```

For each language translation, properties beginning `<lang>`. will override the standard ones. In the abov Framework for the Spanish translation. Also, translators can be credited by adding a '<lang>.translators' pr

```
fr.translators=Stphane Maldini
```

This should be a comma-separated list of names (or the native language equivalent) and it will be display
itself.

You can build specific translations very easily using the `publishGuide_*` and `publishPdf_*` task
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 `buil`
guide by opening `build/docs/<lang>/index.html`.

All translations are created as part of the [Hudson CI build for the grails-doc](#) project, so you can easily see
the docs yourself.