**1 Introduction - Reference Documentation**

**Authors:** Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, Lari Hotari

**Version:** 3.0.1

**Table of Contents**

[**1.1**What's new in Grails 3.0?](https://grails.github.io/grails-doc/latest/guide/introduction.html#whatsNew)

[**1.1.1**Core Features](https://grails.github.io/grails-doc/latest/guide/introduction.html#coreFeatures)

[**1.1.2**Web Features](https://grails.github.io/grails-doc/latest/guide/introduction.html#webFeatures)

[**1.1.3**Development Environment Features](https://grails.github.io/grails-doc/latest/guide/introduction.html#developmentEnvironmentFeatures)

[**1.1.4**Testing Features](https://grails.github.io/grails-doc/latest/guide/introduction.html#testingFeatures)

**1 Introduction**

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

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

Grails is a full stack framework and attempts to solve as many pieces of the web development puzzle through the core technology and its associated plugins. Included out the box are things like:

* An easy to use Object Relational Mapping (ORM) layer built on [Hibernate](http://www.hibernate.org/)
* An expressive view technology called Groovy Server Pages (GSP)
* A controller layer built on [Spring](http://www.spring.io/) MVC
* An interactive command line environment and build system based on [Gradle](http://gradle.org/)
* An embedded [Tomcat](http://tomcat.apache.org/) 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](http://groovy-lang.org/) language and the extensive use of Domain Specific Languages (DSLs)

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

**1.1 What's new in Grails 3.0?**

This section covers the new features that are present in 3.0 and is broken down into sections covering the build system, core APIs, the web tier, persistence enhancements and improvements in testing. Note there are many more small enhancements and improvements, these sections just cover some of the highlights.

**1.1.1 Core Features**

**Groovy 2.4**

Grails 3.0 comes with Groovy 2.4 which includes many new features and enhancements.

For more information on Groovy 2.4, see the [release notes](http://groovy-lang.org/releasenotes/groovy-2.4.html) for more information.

**Spring 4.1 and Spring Boot 1.2**

Grails 3.0 comes with Spring 4.1 which includes [many new features and enhancements](https://spring.io/blog/2014/09/04/spring-framework-4-1-ga-is-here).

In addition, Grails 3.0 is built on [Spring Boot 1.2](http://projects.spring.io/spring-boot/) which provides the ability to produce runnable JAR files that can embed Tomcat, Jetty or Undertow containers.

**Gradle Build System**

Grails 3.0 deprecates the older Gant-based build system in favour of a new [Gradle-based](http://gradle.org/) build that integrates closely with the [Gradle plugin ecosystem](http://plugins.gradle.org/).

See the new section on the new [Gradle build](https://grails.github.io/grails-doc/latest/guide/single.html#gradleBuild) for more information.

**Application Profiles**

Grails 3.0 supports the notion of application profiles via a new [profile repository](https://github.com/grails/grails-profile-repository). A profile encapsulates an application structure, set of commands, plugins and capabilities. For example the "web" profile allows construction of web applications deployable to a Servlet container. In the future more profiles will be developed targeting different environments.

See the new section on [Profiles](https://grails.github.io/grails-doc/latest/guide/single.html#profiles) for more information.

**Redesigned API based on Traits**

The Grails API has been redesigned so that public API is correctly populated under the grails. package whilst private / internal API that is subject to change can be found in the org.grails.package. The core API has also been rewritten and based around the [Groovy Traits](http://groovy-lang.org/objectorientation.html#_traits).

See the new documentation on Grails 3.0's [core traits](https://grails.github.io/grails-doc/latest/guide/single.html#traits) for more information.

**1.1.2 Web Features**

**New Interceptors API**

In previous versions of Grails, filters were used to define logic that intercepts controller action execution.

As of Grails 3.0, this API is deprecated and has been replaced by the new [Interceptor API](https://grails.github.io/grails-doc/latest/guide/single.html#interceptors). An example interceptor can be seen below:

class MyInterceptor {

boolean before() { **true** }

boolean after() { **true** }

void afterView() {

// no-op

}

}

**1.1.3 Development Environment Features**

**New Shell and Code Generation API**

Replacing Gant, Grails 3.0 features a new interactive command line shell that integrates closely with Gradle and provides APIs for writing scripts that interact with Gradle and perform code generation.

The new shell integrates closely with the concept of application profiles with each profile capable defining [profile specific commands](https://github.com/grails/grails-profile-repository/tree/master/profiles/web/commands). As with previous versions of Grails, plugins can define new shell commands that can invoke Gradle or perform code generation and project automation tasks.

See the new guide on [Creating Custom Scripts](https://grails.github.io/grails-doc/latest/guide/single.html#creatingCustomScripts) for more information.

**Enhanced IDE Integration**

Since Grails 3.0 is built on Gradle, you can now import a Grails project using IntelliJ community edition or GGTS's Gradle tooling support without the need for Grails specific tooling. Grails 3.0 plugins are published as simple JAR files greatly reducing the need for additional IDE support specific to Grails.

**Application Main Class**

Each new Grails 3.0 project features an Application class that has a traditional static void main signature, meaning to run or debug a Grails 3.0 application from an IDE like IntelliJ or GGTS you can simply right-click on the Application class and execute to start your Grails application. All Grails 3.0 tests can also just be run from the IDE directly without needing to resort to the command line (even integration / functional tests!).

**1.1.4 Testing Features**

**Integration and Geb Functional Tests**

Grails 3.0 supports built in support for Spock/Geb functional tests using the [create-functional-test](https://grails.github.io/grails-doc/latest/ref/Command%20Line/create-functional-test.html) command. Functional tests are based on Spring Boot's test running mechanism and load the application just once for an entire suite of tests. The tests can be run from and IDE and don't require the command line.

**Gradle Test Running**

Since Grails 3.0 is built on Gradle the test execution configuration is much more flexible and can easily configured to execute in parallel.

**2 Getting Started - Reference Documentation**

**Authors:** Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, Lari Hotari

**Version:** 3.0.1

**Table of Contents**

[**2.1**Installation Requirements](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#requirements)

[**2.2**Downloading and Installing](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#downloadingAndInstalling)

[**2.3**Creating an Application](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#creatingAnApplication)

[**2.4**A Hello World Example](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#aHelloWorldExample)

[**2.5**Using Interactive Mode](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#usingInteractiveMode)

[**2.6**Getting Set Up in an IDE](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#ide)

[**2.7**Convention over Configuration](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#conventionOverConfiguration)

[**2.8**Running an Application](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#runningAnApplication)

[**2.9**Testing an Application](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#testingAnApplication)

[**2.10**Deploying an Application](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#deployingAnApplication)

[**2.11**Supported Java EE Containers](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#supportedJavaEEContainers)

[**2.12**Creating Artefacts](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#creatingArtefacts)

[**2.13**Generating an Application](https://grails.github.io/grails-doc/latest/guide/gettingStarted.html#generatingAnApplication)

**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 1.7 or above. Download the appropriate JDK for your operating system, run the installer, and then set up an environment variable called JAVA\_HOME pointing to the location of this installation.

To automate the installation of Grails we recommend the [GVM tool](http://gvmtool.net/) which greatly simplifies installing and managing multiple Grails versions.

For manual installation, we recommend the video installation guides from [grailsexample.net](http://www.grailsexample.net/):

* [Windows](http://www.grailsexample.net/installing-a-grails-development-environment-on-windows/)
* [Linux](http://www.grailsexample.net/installing-a-grails-development-environment-on-linux/)
* [Mac OS X](http://www.grailsexample.net/installing-a-grails-development-environment-on-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 cases you will want to manually configure the location of Java. For example:

export JAVA\_HOME=/Library/Java/Home

export PATH="$PATH:$JAVA\_HOME/bin"

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

**2.2 Downloading and Installing**

The first step to getting up and running with Grails is to install the distribution.

The best way to install Grails on \*nix systems is with the [GVM tool](http://gvmtool.net/) which greatly simplifies installing and managing multiple Grails versions.

For manual installation follow these steps:

* [Download](https://github.com/grails/grails-core/releases) a binary distribution of Grails and extract the resulting zip file to a location of your choice
* Set the GRAILS\_HOME environment variable to the location where you extracted the zip
  + On Unix/Linux based systems this is typically a matter of adding something like the following export GRAILS\_HOME=/path/to/grails to your profile
  + On Windows this is typically a matter of setting an environment variable under My Computer/Advanced/Environment Variables
* Then add the bin directory to your PATH variable:
  + On Unix/Linux based systems this can be done by adding export PATH="$PATH:$GRAILS\_HOME/bin" to your profile
  + On Windows this is done by modifying the Path environment variable under My Computer/Advanced/Environment Variables

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

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 command which is used in the following manner:

grails [command name]

Run [create-app](https://grails.github.io/grails-doc/latest/ref/Command%20Line/create-app.html) 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 in your console:

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 "helloworld" directory you just created and start the Grails interactive console:

$ cd helloworld

$ grails

What we want is a simple page that just prints the message "Hello World!" to the browser. In Grails, whenever you want a new page you just create a new controller action for it. Since we don't yet have a controller, let's create one now with the [create-controller](https://grails.github.io/grails-doc/latest/ref/Command%20Line/create-controller.html) command:

grails> create-controller hello

Don't forget that in the interactive console, we have auto-completion on command names. So you can type "cre" and then press <tab> to get a list of all 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](https://grails.github.io/grails-doc/latest/guide/single.html#controllers) in the grails-app/controllers/helloworld directory called HelloController.groovy. Why the extra helloworld directory? Because in Java land, it's strongly recommended that all classes are placed into packages, so Grails defaults to the application name if you don't provide one. The reference page for [create-controller](https://grails.github.io/grails-doc/latest/ref/Command%20Line/create-controller.html) provides more detail on this.

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

**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](https://grails.github.io/grails-doc/latest/ref/Tags/render.html) the page.

Job done. To see your application in action, you just need to start up a server with another command called [run-app](https://grails.github.io/grails-doc/latest/ref/Command%20Line/run-app.html):

grails> run-app

This will start an embedded server on port 8080 that hosts your application. You should now be able to access your application at the URL <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 wish to restore this behavior you can configure a context path in grails-app/conf/application.yml:

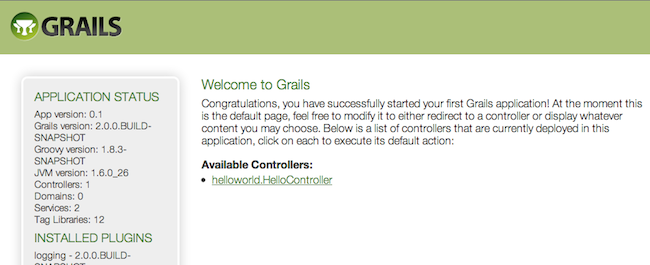
server:

'context-path': '/helloworld'

With the above configuration in place the server will instead startup at the URL <http://localhost:8080/helloworld/>.

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

The result will look something like this:



This is the Grails intro page which is rendered by the grails-app/view/index.gsp file. It detects the presence of your controllers and provides links to them. You can click on the "HelloController" link to see our custom page containing the text "Hello World!". Voila! You have your first working Grails application.

One final thing: a controller can contain many actions, each of which corresponds to a different page (ignoring AJAX at this point). Each page is accessible via a unique URL that is composed from the controller name and the action name: /<appname>/<controller>/<action>. This means you can access the Hello World page via [/helloworld/hello/index](http://localhost:8080/helloworld/hello/index), where 'hello' is the controller name (remove the 'Controller' suffix from the class name and lower-case the first letter) and 'index' is the action name. But you can also access the page via the same URL without the action name: this is because 'index' is the *default action* . See the end of the [controllers and actions](https://grails.github.io/grails-doc/latest/guide/single.html#understandingControllersAndActions) section of the user guide to find out more on default actions.

**2.5 Using Interactive Mode**

Grails 3.0 features an interactive mode which makes command execution faster since the JVM doesn't have to be restarted for each command. To use interactive mode simple type 'grails' from the root of any projects and use TAB completion to get a list of available commands. See the screenshot below for an example:

For more information on the capabilities of interactive mode refer to the section on [Interactive Mode](https://grails.github.io/grails-doc/latest/guide/single.html#interactiveMode) in the user guide.

**2.6 Getting Set Up in an IDE**

**IntelliJ IDEA**

[IntelliJ IDEA](http://www.jetbrains.com/idea) is an excellent IDE for Grails 3.0 development. It comes in 2 editions, the free community edition and the paid-for ultimate edition.

The community edition can be used for most things, although GSP syntax higlighting is only part of the ultimate edition. To get started with Intellij IDEA and Grails 3.0 simply go to File / Import Project and point IDEA at your build.gradle file to import and configure the project.

**Eclipse**

We recommend that users of [Eclipse](http://www.eclipse.org/) looking to develop Grails application take a look at [Groovy/Grails Tool Suite](https://spring.io/tools/ggts), which offers built in support for Grails 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 the ability to run Grails applications in the IDE, code completion and integration with the Glassfish server. For an overview of features see the [NetBeans Integration](http://www.grails.org/NetBeans+Integration) guide on the Grails website which 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](https://github.com/textmate/groovy-grails.tmbundle) exists Groovy / Grails support in [Textmate](http://macromates.com/)
* A [Sublime Text plugin](https://github.com/osoco/sublimetext-grails) can be installed via Sublime Package Control for the [Sublime Text Editor](http://www.sublimetext.com/).
* See [this post](http://www.objectpartners.com/2012/02/21/using-vim-as-your-grails-ide-part-1-navigating-your-project/) for some helpful tips on how to setup VIM as your Grails editor of choise.
* An [Atom Package](https://atom.io/packages/atom-grails) is available for use with the [Atom editor](https://atom.io/)

**2.7 Convention over Configuration**

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

Here is a breakdown and links to the relevant sections:

* grails-app - top level directory for Groovy sources
  + conf - [Configuration sources](https://grails.github.io/grails-doc/latest/guide/single.html#conf).
  + controllers - [Web controllers](https://grails.github.io/grails-doc/latest/guide/single.html#controllers) - The C in MVC.
  + domain - The [application domain](https://grails.github.io/grails-doc/latest/guide/single.html#GORM).
  + i18n - Support for [internationalization (i18n)](https://grails.github.io/grails-doc/latest/guide/single.html#i18n).
  + services - The [service layer](https://grails.github.io/grails-doc/latest/guide/single.html#services).
  + taglib - [Tag libraries](https://grails.github.io/grails-doc/latest/guide/single.html#taglibs).
  + utils - Grails specific utilities.
  + views - [Groovy Server Pages](https://grails.github.io/grails-doc/latest/guide/single.html#gsp) - The V in MVC.
* scripts - [Code generation scripts](https://grails.github.io/grails-doc/latest/guide/single.html#commandLine).
* src/main/groovy - Supporting sources
* src/test/groovy - [Unit and integration tests](https://grails.github.io/grails-doc/latest/guide/single.html#testing).

**2.8 Running an Application**

Grails applications can be run with the built in Tomcat server using the [run-app](https://grails.github.io/grails-doc/latest/ref/Command%20Line/run-app.html) command which will load a server on port 8080 by default:

grails run-app

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

grails -Dserver.port=8090 run-app

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

$ grails

grails> run-app

| Server running. Browse to http://localhost:8080/helloworld

| Application loaded in interactive mode. Type 'stop-app' to shutdown.

| Downloading: plugins-list.xml

grails> stop-app

| Stopping Grails server

grails> run-app

| Server running. Browse to http://localhost:8080/helloworld

| Application loaded in interactive mode. Type 'stop-app' to shutdown.

| Downloading: plugins-list.xml

More information on the [run-app](https://grails.github.io/grails-doc/latest/ref/Command%20Line/run-app.html) 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/test/groovy directory. It is of course up to you to populate these tests with valid test logic, information on which can be found in the section on [Testing](https://grails.github.io/grails-doc/latest/guide/single.html#testing).

To execute tests you run the [test-app](https://grails.github.io/grails-doc/latest/ref/Command%20Line/test-app.html) 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 Archive (WAR file), and Grails includes the [war](https://grails.github.io/grails-doc/latest/ref/Command%20Line/war.html) 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 container's instructions.

Note that by default Grails will include an embeddable version of Tomcat inside the WAR file, this can cause problems if you deploy to a different version of Tomcat. If you don't intend to use the embedded container then you should change the scope of the Tomcat dependencies to provided prior to deploying to your production container in build.gradle:

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

Unlike most scripts which default to the development environment unless overridden, the war command runs in the production environment by 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 regular Java application. Example:

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 sufficient memory allocation. A good set of VM 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 specific container products:

* 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 > Java Virtual Machine > Generic JVM arguments" for older versions of WebSphere. This is no longer needed for WebSphere version 8 or newer.

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

**2.12 Creating Artefacts**

Grails ships with a few convenience targets such as [create-controller](https://grails.github.io/grails-doc/latest/ref/Command%20Line/create-controller.html), [create-domain-class](https://grails.github.io/grails-doc/latest/ref/Command%20Line/create-domain-class.html) and so on that will create [Controllers](https://grails.github.io/grails-doc/latest/guide/single.html#controllers) and different artefact types for you.

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

For example to create the basis of an application you typically need a [domain model](https://grails.github.io/grails-doc/latest/guide/single.html#GORM):

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.groovy such as:

**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](https://grails.github.io/grails-doc/latest/guide/single.html#scaffolding) to generate the skeleton of an application. To do this use one of the generate-\* commands such as [generate-all](https://grails.github.io/grails-doc/latest/ref/Command%20Line/generate-all.html), which will generate a [controller](https://grails.github.io/grails-doc/latest/guide/single.html#controllers) (and its unit test) and the associated [views](https://grails.github.io/grails-doc/latest/guide/single.html#gsp):

grails generate-all helloworld.Book

# 4 Configuration - Reference Documentation

**Authors:** Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, Lari Hotari

**Version:** 3.0.1

## Table of Contents

[**4.1**Basic Configuration](https://grails.github.io/grails-doc/latest/guide/conf.html#config)

[**4.1.1**Built in options](https://grails.github.io/grails-doc/latest/guide/conf.html#builtInOptions)

[**4.1.2**Logging](https://grails.github.io/grails-doc/latest/guide/conf.html#logging)

[**4.1.3**GORM](https://grails.github.io/grails-doc/latest/guide/conf.html#configGORM)

[**4.2**The Application Class](https://grails.github.io/grails-doc/latest/guide/conf.html#applicationClass)

[**4.2.1**Executing the Application Class](https://grails.github.io/grails-doc/latest/guide/conf.html#executing)

[**4.2.2**Customizing the Application Class](https://grails.github.io/grails-doc/latest/guide/conf.html#customizing)

[**4.2.3**The Application LifeCycle](https://grails.github.io/grails-doc/latest/guide/conf.html#applicationLifeCycle)

[**4.3**Environments](https://grails.github.io/grails-doc/latest/guide/conf.html#environments)

[**4.4**The DataSource](https://grails.github.io/grails-doc/latest/guide/conf.html#dataSource)

[**4.4.1**DataSources and Environments](https://grails.github.io/grails-doc/latest/guide/conf.html#dataSourcesAndEnvironments)

[**4.4.2**Automatic Database Migration](https://grails.github.io/grails-doc/latest/guide/conf.html#automaticDatabaseMigration)

[**4.4.3**Transaction-aware DataSource Proxy](https://grails.github.io/grails-doc/latest/guide/conf.html#transactionAwareDataSourceProxy)

[**4.4.4**Database Console](https://grails.github.io/grails-doc/latest/guide/conf.html#databaseConsole)

[**4.4.5**Multiple Datasources](https://grails.github.io/grails-doc/latest/guide/conf.html#multipleDatasources)

[**4.5**Versioning](https://grails.github.io/grails-doc/latest/guide/conf.html#versioning)

[**4.6**Project Documentation](https://grails.github.io/grails-doc/latest/guide/conf.html#docengine)

[**4.7**Dependency Resolution](https://grails.github.io/grails-doc/latest/guide/conf.html#dependencyResolution)

# 4 Configuration

It may seem odd that in a framework that embraces "convention-over-configuration" that we tackle this topic now. With Grails' default settings you can actually develop an application without doing any configuration whatsoever, as the quick start demonstrates, but it's important to learn where and how to override the conventions when you need to. Later sections of the user guide will mention what configuration settings you can use, but not how to set them. 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 configuration is by default specified in YAML in the grails-app/conf/application.yml file.

If you prefer to use Grails 2.0-style Groovy configuration then you can create an additional grails-app/conf/application.groovy file to specify configuration using Groovy's [ConfigSlurper](http://groovy.codehaus.org/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 variable is set, it is used. |
| 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 [grailsApplication](http://grails.org/doc/3.0.x/api/grails/core/GrailsApplication.html) object, which is available 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](http://grails.org/doc/3.0.x/api/grails/config/Config.html) interface and provides a number of useful methods to read the configuration of the application.

Notice that the Config instance is a merged configuration based on Spring's [PropertySource](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/annotation/PropertySource.html) concept and reads configuration from the environment, system properties and the local application configuration merging them into a single object.

and 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](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/beans/factory/annotation/Value.html) annotation to dependency injection 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 Value annotation otherwise it is 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 them.

## 4.1.1 Built in options

Grails has a set of core settings that are worth knowing about. Their defaults are suitable for most projects, but it's important to understand what they do 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 settings:

* grails.enable.native2ascii - Set this to false if you do not require native2ascii conversion of Grails i18n properties files (default: true).
* grails.views.default.codec - Sets the default encoding regime for GSPs - can be one of 'none', 'html', or 'base64' (default: 'none'). To reduce risk of XSS attacks, set this to 'html'.
* grails.views.gsp.encoding - The file encoding used for GSP source files (default: 'utf-8').
* grails.mime.file.extensions - Whether to use the file extension to dictate the mime type in [Content Negotiation](https://grails.github.io/grails-doc/latest/guide/single.html#contentNegotiation) (default: true).
* grails.mime.types - A map of supported mime types used for [Content Negotiation](https://grails.github.io/grails-doc/latest/guide/single.html#contentNegotiation).
* grails.serverURL - A string specifying the server URL portion of absolute links, including server name e.g. grails.serverURL="http://my.yourportal.com". See [createLink](https://grails.github.io/grails-doc/latest/ref/Tags/createLink.html). Also used by redirects.
* grails.views.gsp.sitemesh.preprocess - Determines whether SiteMesh preprocessing happens. Disabling this slows down page rendering, but if you need SiteMesh to parse the generated HTML from a GSP view then disabling it is the right option. Don't worry if you don't understand this advanced property: leave it set to true.
* grails.reload.excludes and grails.reload.includes - Configuring these directives determines the reload behavior for project specific source files. Each directive takes a list of strings that are the class names for project source files that should be excluded from reloading behavior or included accordingly when running the application in development with the run-appcommand. If the grails.reload.includes directive is configured, then only the classes in that list will be reloaded.

## 4.1.2 Logging

By default logging in Grails 3.0 is handled by the [Logback logging framework](http://logback.qos.ch/) and can be configured with the grails-app/conf/logback.groovy file.

If you prefer XML you can replace the logback.groovy file with a logback.xml file instead.

For more information on configuring logging refer to the [Logback documentation](http://logback.qos.ch/manual/groovy.html) on the subject.

## 4.1.3 GORM

Grails provides the following GORM configuration options:

* grails.gorm.failOnError - If set to true, causes the save() method on domain classes to throw a grails.validation.ValidationException if [validation](https://grails.github.io/grails-doc/latest/guide/single.html#validation) fails during a save. This option may also be assigned a list of Strings representing package names. If the value is a list of Strings then the failOnError behavior will only be applied to domain classes in those packages (including sub-packages). See the [save](https://grails.github.io/grails-doc/latest/ref/Domain%20Classes/save.html) 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](https://grails.github.io/grails-doc/latest/ref/Domain%20Classes/merge.html), [save](https://grails.github.io/grails-doc/latest/ref/Domain%20Classes/save.html) and [delete](https://grails.github.io/grails-doc/latest/ref/Domain%20Classes/delete.html) methods to flush the session, replacing the need to explicitly flush using save(flush: true).

## 4.2 The Application Class

Every new Grails application features an Application class witin the the grails-app/init directory.

The Application class subclasses the [GrailsAutoConfiguration](http://grails.org/doc/3.0.x/api/grails/boot/config/GrailsAutoConfiguration.html) class and features a static void main method, meaning it can be run as a regular 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 right click on the class and run it directly from 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 a remote debugger when using the run-app --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 there are packages in other JAR files you wish to scan you can do so by overriding thepackageNames() 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 method annotated with the [Bean](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/annotation/Bean.html) and the returned 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](http://grails.org/doc/3.0.x/api/grails/core/GrailsApplicationLifeCycle.html) interface which all plugins implement.

This means that the Application class can be used to perform the same functions as a plugin. You can override the [regular plugins hooks](https://grails.github.io/grails-doc/latest/guide/single.html#hookingIntoRuntimeConfiguration) such as doWithSpring, doWithApplicationContextand 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 and application.groovy files in the grails-app/conf directory can use per-environment configuration using either YAML or the syntax provided by [ConfigSlurper](http://groovy.codehaus.org/ConfigSlurper). As an example 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=FALSE

test:

dataSource:

dbCreate: update

url: jdbc:h2:mem:testDb;MVCC=TRUE;LOCK\_TIMEOUT=10000;DB\_CLOSE\_ON\_EXIT=FALSE

production:

dataSource:

dbCreate: update

url: jdbc:h2:prodDb;MVCC=TRUE;LOCK\_TIMEOUT=10000;DB\_CLOSE\_ON\_EXIT=FALSE

properties:

jmxEnabled: **true**

initialSize: 5

...

The above can expression 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 block specifies per environment settings for the dbCreate and url properties of the DataSource.

#### Packaging and Running for Different Environments

Grails' [command line](https://grails.github.io/grails-doc/latest/guide/single.html#commandLine) has built in capabilities to execute any command within the context of a specific environment. The format is:

grails [environment] [command name]

In addition, there are 3 preset environments known to Grails: dev, prod, and test for development, production and test. For example to create a WAR for the test environment you 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 [Environment](http://grails.org/doc/3.0.x/api/grails/util/Environment.html) class:

**import** grails.util.Environment

...

**switch** (Environment.current) {

**case** Environment.DEVELOPMENT:

configureForDevelopment()

**break**

**case** Environment.PRODUCTION:

configureForProduction()

**break**

}

#### Per Environment Bootstrapping

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

def init = { ServletContext ctx ->

environments {

production {

ctx.setAttribute("env", "prod")

}

development {

ctx.setAttribute("env", "dev")

}

}

ctx.setAttribute("foo", "bar")

}

#### Generic Per Environment Execution

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

Environment.executeForCurrentEnvironment {

production {

// **do** something in production

}

development {

// **do** something only in development

}

}

## 4.4 The DataSource

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

If you use a database other than H2 you need a JDBC driver. For example for MySQL you would need [Connector/J](http://www.mysql.com/downloads/connector/j/)

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

dependencies {

runtime 'mysql:mysql-connector-java:5.1.29'

}

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

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

* driverClassName - The class name of the JDBC driver
* username - The username used to establish a JDBC connection
* password - The password used to establish a JDBC connection
* url - The JDBC URL of the database
* dbCreate - Whether to auto-generate the database from the domain model - one of 'create-drop', 'create', 'update' or 'validate'
* pooled - Whether to use a pool of connections (defaults to true)
* logSql - Enable SQL logging to stdout
* formatSql - Format logged SQL
* dialect - A String or Class that represents the Hibernate dialect used to communicate with the database. See the [org.hibernate.dialect](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/dialect/package-summary.html) package for available dialects.
* readOnly - If true makes the DataSource read-only, which results in the connection pool calling setReadOnly(true) on each Connection
* transactional - If false leaves the DataSource's transactionManager bean outside the chained BE1PC transaction manager implementation. This only applies to additional datasources.
* persistenceInterceptor - The default datasource is automatically wired up to the persistence interceptor, other datasources are not wired up automatically unless this is set to true
* properties - Extra properties to set on the DataSource bean. See the [Tomcat Pool](http://tomcat.apache.org/tomcat-7.0-doc/jdbc-pool.html#Common_Attributes) documentation. There is also a Javadoc format [documentation of the properties](https://tomcat.apache.org/tomcat-7.0-doc/api/org/apache/tomcat/jdbc/pool/PoolConfiguration.html).
* jmxExport - If false, will disable registration of JMX MBeans for all DataSources. By default JMX MBeans are added for DataSources with jmxEnabled = true in properties.

A typical configuration for MySQL may be something like:

dataSource {

pooled = **true**

dbCreate = "update"

url = "jdbc:mysql://localhost:3306/my\_database"

driverClassName = "com.mysql.jdbc.Driver"

dialect = org.hibernate.dialect.MySQL5InnoDBDialect

username = "username"

password = "password"

properties {

jmxEnabled = **true**

initialSize = 5

maxActive = 50

minIdle = 5

maxIdle = 25

maxWait = 10000

maxAge = 10 \* 60000

timeBetweenEvictionRunsMillis = 5000

minEvictableIdleTimeMillis = 60000

validationQuery = "SELECT 1"

validationQueryTimeout = 3

validationInterval = 15000

testOnBorrow = **true**

testWhileIdle = **true**

testOnReturn = **false**

jdbcInterceptors = "ConnectionState;StatementCache(max=200)"

defaultTransactionIsolation = java.sql.Connection.TRANSACTION\_READ\_COMMITTED

}

}

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

dataSource {

boolean pooled = **true** // type declaration results in ignored local variable

…

}

Example of advanced configuration using extra properties:

dataSource {

pooled = **true**

dbCreate = "update"

url = "jdbc:mysql://localhost: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\_Attributes

// https://tomcat.apache.org/tomcat-7.0-doc/api/org/apache/tomcat/jdbc/pool/PoolConfiguration.html

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\_interceptors

jdbcInterceptors = "ConnectionState;StatementCache(max=200)"

defaultTransactionIsolation = java.sql.Connection.TRANSACTION\_READ\_COMMITTED // safe **default**

// controls **for** leaked connections

abandonWhenPercentageFull = 100 // settings are active only when pool is full

removeAbandonedTimeout = 120

removeAbandoned = **true**

// use JMX console to change **this** setting at runtime

logAbandoned = **false** // causes stacktrace recording overhead, use only **for** debugging

// JDBC driver properties

// Mysql as example

dbProperties {

// Mysql specific driver properties

// http://dev.mysql.com/doc/connector-j/en/connector-j-reference-configuration-properties.html

// 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 driver's cache

cachePrepStmts=**false**

cacheCallableStmts=**false**

// Tomcat JDBC Pool's StatementFinalizer keeps track

dontTrackOpenResources=**true**

// performance optimization: reduce number of SQLExceptions thrown in mysql driver code

holdResultsOpenOverStatementClose=**true**

// enable MySQL query cache - using server prep stmts will disable query caching

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 control over when and how it does this through the dbCreate property, which can take these values:

* **create** - Drops the existing schema and creates the schema on startup, dropping existing tables, indexes, etc. first.
* **create-drop** - Same as **create**, but also drops the tables when the application shuts down cleanly.
* **update** - Creates missing tables and indexes, and updates the current schema without dropping any tables or data. Note that this can't properly handle many schema changes like column renames (you're left with the old column containing the existing data).
* **validate** - Makes no changes to your database. Compares the configuration with the existing database schema and reports warnings.
* any other value - does nothing

You can also remove the dbCreate setting completely, which is recommended once your schema is relatively stable and definitely when your application and database are deployed in production. Database changes are then managed through proper migrations, either with SQL scripts or a migration tool like [Liquibase](http://www.liquibase.org/) (the [Database Migration](http://grails.org/plugin/database-migration) plugin uses Liquibase and is tightly integrated with Grails and GORM).

## 4.4.1 DataSources and Environments

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

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

dataSource {

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 should do at runtime with regards to automatically generating the database tables from [GORM](https://grails.github.io/grails-doc/latest/guide/single.html#GORM) classes. The options are described in the [DataSource](https://grails.github.io/grails-doc/latest/guide/single.html#dataSource) section:

* create
* create-drop
* update
* validate
* no value

In [development](https://grails.github.io/grails-doc/latest/guide/single.html#environments) mode dbCreate is by default set to "create-drop", but at some point in development (and certainly once you go to production) you'll need to stop dropping and re-creating the database every time you start up your server.

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

Grails supports migrations with Flyway or Liquibase using the [same mechanism provided by Spring Boot](http://docs.spring.io/spring-boot/docs/current/reference/html/howto-database-initialization.html).

## 4.4.3 Transaction-aware DataSource Proxy

The actual dataSource bean is wrapped in a transaction-aware proxy so you will be given the connection that's being used by the current transaction or Hibernate Session if one is active.

If this were not the case, then retrieving a connection from the dataSource would be a new connection, and you wouldn't be able to see changes that haven't been committed yet (assuming you have a sensible transaction isolation setting, e.g. READ\_COMMITTED or better).

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

You can access this bean like any other Spring bean, i.e. using dependency injection:

class MyService {

def dataSourceUnproxied

…

}

or by pulling it from the ApplicationContext:

def dataSourceUnproxied = ctx.dataSourceUnproxied

## 4.4.4 Database Console

The [H2 database console](http://h2database.com/html/quickstart.html#h2_console) is a convenient feature of H2 that provides a web-based interface to any database that you have a JDBC driver for, and it's very useful to view the database you're developing against. It's especially useful when running against an in-memory database.

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

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

environments {

production {

grails.serverURL = "http://www.changeme.com"

grails.dbconsole.enabled = **true**

grails.dbconsole.urlRoot = '/admin/dbconsole'

}

development {

grails.serverURL = "http://localhost:8080/${appName}"

}

test {

grails.serverURL = "http://localhost:8080/${appName}"

}

}

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

#### Configuration

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

## 4.4.5 Multiple Datasources

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

#### Configuring Additional DataSources

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

---

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=FALSE

test:

dataSource:

dbCreate: update

url: jdbc:h2:mem:testDb;MVCC=TRUE;LOCK\_TIMEOUT=10000;DB\_CLOSE\_ON\_EXIT=FALSE

production:

dataSource:

dbCreate: update

url: jdbc:h2:prodDb;MVCC=TRUE;LOCK\_TIMEOUT=10000;DB\_CLOSE\_ON\_EXIT=FALSE

properties:

jmxEnabled: **true**

initialSize: 5

This configures a single DataSource with the Spring bean named dataSource. To configure extra DataSources, add a dataSources block (at the top level, in an environment block, or both, just like the standard DataSource definition) with a custom name. For example, this configuration adds 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\_EXIT=FALSE

test:

dataSources:

dataSource:

dbCreate: update

url: jdbc:h2:mem:testDb;MVCC=TRUE;LOCK\_TIMEOUT=10000;DB\_CLOSE\_ON\_EXIT=FALSE

production:

dataSources:

dataSource:

dbCreate: update

url: jdbc:h2:prodDb;MVCC=TRUE;LOCK\_TIMEOUT=10000;DB\_CLOSE\_ON\_EXIT=FALSE

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'. Set the datasource property in the mapping block to configure a non-default DataSource. For example, if you want to use the ZipCode domain to use the 'lookup' DataSource, configure it like this;

class ZipCode {

String code

**static** mapping = {

datasource 'lookup'

}

}

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

class ZipCode {

String code

**static** mapping = {

datasources(['lookup', 'auditing'])

}

}

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

class ZipCode {

String code

**static** mapping = {

datasources(['lookup', 'DEFAULT'])

}

}

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

class ZipCode {

String code

**static** mapping = {

datasource 'ALL'

}

}

#### Namespaces and GORM Methods

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

class ZipCode {

String code

**static** mapping = {

datasources(['lookup', 'auditing'])

}

}

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

def zipCode = ZipCode.auditing.get(42)

…

zipCode.auditing.save()

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

#### Hibernate Mapped Domain Classes

You can also partition annotated Java classes into separate datasources. Classes using the default datasource are registered in grails-app/conf/hibernate/hibernate.cfg.xml. To specify that an annotated class uses a non-default datasource, create a hibernate.cfg.xml file 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/conf/hibernate/hibernate.cfg.xml:

<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

'-//Hibernate/Hibernate Configuration DTD 3.0//EN'

'http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd'>

<hibernate-configuration>

<session-factory>

<mapping class='org.example.Book'/>

</session-factory>

</hibernate-configuration>

and if the Library class is in the "ds2" datasource, you would register that in grails-app/conf/hibernate/ds2\_hibernate.cfg.xml:

<?xml version='1.0' encoding='UTF-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

'-//Hibernate/Hibernate Configuration DTD 3.0//EN'

'http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd'>

<hibernate-configuration>

<session-factory>

<mapping class='org.example.Library'/>

</session-factory>

</hibernate-configuration>

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

#### Services

Like Domain classes, by default Services use the default DataSource and PlatformTransactionManager. To configure a Service to use a different DataSource, use the static datasourceproperty, 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 classes whose DataSource is the same as the Service.

Note that the datasource specified in a service has no bearing on which datasources are used for domain classes; that's determined by their declared 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 dataSource2, and WahooService uses dataSource1, a service method that saves a new Foo and a new Bar will only be transactional for Foo since they share the datasource. The transaction won't affect the Bar instance. If you want both to be transactional you'd need to use two services and XA datasources for two-phase commit, e.g. with the Atomikos plugin.

#### Transactions across multiple datasources

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

The [Best Efforts 1PC pattern](http://www.javaworld.com/article/2077963/open-source-tools/distributed-transactions-in-spring--with-and-without-xa.html?page=2) is fairly general but can fail in some circumstances that the developer must be aware of. This is a non-XA pattern that involves a synchronized single-phase commit of a number of resources. Because the [2PC](https://en.wikipedia.org/wiki/Two-phase_commit) is not used, it can never be as safe as an [XA](https://en.wikipedia.org/wiki/X/Open_XA) transaction, but is 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 thing that can go wrong is an infrastructure failure (not a business-processing error). Systems that rely on Best Efforts 1PC reason that infrastructure failures are rare enough that they can afford to take the risk in return for higher throughput. If business-processing services are also designed to be idempotent, then little can go wrong in practice.

The BE1PC implementation was added in Grails 2.3.6. . Before this change additional datasources didn't take part in transactions initiated in Grails. The transactions in additional datasources were basically in auto commit mode. In some cases this might be the wanted behavior. One reason might be performance: on the start of each new transaction, the BE1PC transaction manager creates a new transaction to each datasource. It's possible to leave an additional datasource out of the BE1PC transaction manager by setting transactional = false in the respective configuration block of the additional dataSource. Datasources with readOnly = true will also be left out of the chained transaction manager (since 2.3.7).

By default, the BE1PC implementation will add all beans implementing the Spring [PlatformTransactionManager](http://docs.spring.io/spring/docs/3.2.x/javadoc-api/org/springframework/transaction/PlatformTransactionManager.html) interface to the chained BE1PC transaction manager. For example, a possible[JMSTransactionManager](http://docs.spring.io/spring/docs/3.2.x/javadoc-api/org/springframework/jms/connection/JmsTransactionManager.html) bean in the Grails application context would be added to the Grails BE1PC transaction manager's chain of transaction managers.

You can exclude transaction manager beans from the BE1PC implementation with the this configuration option:

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

The exclude matching is done on the name of the transaction manager bean. The transaction managers of datasources with transactional = false 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 resources (not only datasources), there are several options available for adding XA/2PC support to Grails applications.

The [Spring transactions documentation](http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/transaction.html#transaction-application-server-integration) contains information about integrating the JTA/XA transaction manager of different application servers. In this case, you can configure a bean with the nametransactionManager manually in resources.groovy or resources.xml file.

There is also [Atomikos plugin](http://grails.org/plugin/atomikos) 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 [GrailsApplication](http://grails.org/doc/3.0.x/api/grails/core/GrailsApplication.html) class. For example within [controllers](https://grails.github.io/grails-doc/latest/guide/single.html#controllers) there is an implicit [grailsApplication](https://grails.github.io/grails-doc/latest/ref/Controllers/grailsApplication.html) 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 available for your own Grails projects.

The documentation engine uses a variation on the [Textile](http://textile.sitemonks.com/) syntax to automatically create project documentation with smart linking, formatting etc.

#### Creating project documentation

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

+ src/docs/guide/introduction.gdoc

+ src/docs/guide/introduction/changes.gdoc

+ src/docs/guide/gettingStarted.gdoc

+ src/docs/guide/configuration.gdoc

+ src/docs/guide/configuration/build.gdoc

+ src/docs/guide/configuration/build/controllers.gdoc

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

Once you have your source files, you still need to tell the documentation engine what the structure of your user guide is going to be. To do that, you add a src/docs/guide/toc.yml file that contains the structure and titles for each section. This file is in [YAML](http://www.yaml.org/) format and basically represents the 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 corresponding filename (minus the .gdoc extension) followed by a colon. The next line should containtitle: plus the title of the section as seen by the end user. Every sub-section then has its own line after the title. Leaf nodes, i.e. those without any sub-sections, declare their title on the same line as the section name but after the colon.

That's it. You can easily add, remove, and move sections within the toc.yml to restructure the generated user guide. You should also make sure that all section names, i.e. the gdoc filenames, should be unique since they are used for creating internal links and for the HTML filenames. Don't worry though, 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 belongs to a category and a category is a directory located in the src/docs/ref directory. For example, suppose you have defined a new controller method called renderPDF. That belongs to the 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/Config.groovy file that customize the output of the documentation such as:

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

Other properties such as the version are pulled from your project itself. If a title is not specified, the application name is used.

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

One of the simplest ways to customise the look of the generated guide is to provide a value for grails.doc.css and then put a custom.css file in the corresponding directory. Grails will automatically include this CSS file in the guide. You can also place a custom-pdf.css file in that directory. This allows you to override the styles for the PDF version of the guide.

#### Generating Documentation

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

grails doc

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

#### Documentation Syntax

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

##### Basic Formatting

Monospace: monospace

@monospace@

Italic: italic

\_italic\_

Bold: **bold**

\*bold\*

Image:

!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 doc images, but you can specify one with the grails.doc.images setting in Config.groovy like so:

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

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

##### Linking

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

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

or

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

For links to other sections inside the user guide you can use the guide: prefix with the name of the section you want to link to:

[Intro|guide:introduction]

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

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

[renderPDF|controllers]

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

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

grails.doc.api.org.hibernate=

"http://docs.jboss.org/hibernate/stable/core/javadocs"

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

##### Lists and Headings

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

h3.<space>Heading3

h4.<space>Heading4

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

\* item 1

\*\* subitem 1

\*\* subitem 2

\* item 2

Numbered lists can be defined with the # character:

# item 1

Tables can be created using the table macro:

|  |  |
| --- | --- |
| **Name** | **Number** |
| Albert | 46 |
| Wilma | 1348 |
| James | 12 |

{table}

\*Name\* | \*Number\*

Albert | 46

Wilma | 1348

James | 12

{table}

##### Code and Notes

You can define code blocks with the code macro:

class Book {

String title

}

{code}

class Book {

String title

}

{code}

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

<hello>world</hello>

{code:xml}

<hello>world</hello>

{code}

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

Note:

This is a note!

{note}

This is a note!

{note}

Warning:

This is a warning!

{warning}

This is a warning!

{warning}

## 4.7 Dependency Resolution

Dependency resolution is handled by the [Gradle build tool](http://gradle.org/), all dependencies are defined in the build.gradle file. Refer to the [Gradle user guide](https://www.gradle.org/documentation) for more information.

**5 The Command Line - Reference Documentation**

**Authors:** Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, Lari Hotari

**Version:** 3.0.1

**Table of Contents**

[**5.1**Interactive Mode](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\commandLine.html#interactiveMode)

[**5.2**The Command Line and Profiles](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\commandLine.html#profiles)

[**5.3**Creating Custom Scripts](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\commandLine.html#creatingCustomScripts)

[**5.4**Re-using Grails scripts](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\commandLine.html#reusingGrailsScripts)

[**5.5**Building with Gradle](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\commandLine.html#gradleBuild)

[**5.5.1**Defining Dependencies with Gradle](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\commandLine.html#gradleDependencies)

[**5.5.2**Working with Gradle Tasks](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\commandLine.html#gradleTasks)

[**5.5.3**Grails plugins for Gradle](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\commandLine.html#gradlePlugins)

**5 The Command Line**

Grails 3.0's command line system differs greatly from previous versions of Grails and features APIs for invoking Gradle for build related tasks, as well as performing code generation.

When you type:

grails [command name]

Grails searches the [profile repository](https://github.com/grails/grails-profile-repository) based on the profile of the current application. If the profile is for a web application then commmands a read from the web profile and the base profile which it inherits from.

Since command behavior is profile specific the web profile my provide different behavior for the run-app command then say a profile for running 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 is active)
* PROFILE\_REPOSITORY\_PATH/profiles/web/commands/run-app.yml (for YAML defined commands)

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 sources

compile Compiles a Grails application

...

Refer to the Command Line reference in the Quick Reference menu of the reference guide for more information about individual commands

**non-interactive mode**

When you run a script manually and it prompts you for information, you can answer the questions and continue running the script. But when you run a script as part of an automated process, for example a continuous integration build server, there's no way to "answer" the questions. So you can pass the --non-interactive switch to the script command to tell Grails to accept the default answer for any questions, for example whether to install 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 for quicker execution of commands. To activate interactive mode type 'grails' at the command line and then use TAB completion to get a list of commands:

Grails> create-s 按Tab

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

Grails> open target/test…/index.html

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

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

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

Grails> !ls

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

To exit interactive mode enter the exit command. Note that if the Grails application has been run with run-app normally it will terminate when the interactive mode console exits because the JVM will be terminated. An exception to this would be if the application were running in forked mode which means the application is running in a different JVM. In that case the application will be left running after the interactive mode console terminates. If you want to exit interactive mode and stop an application that is running in forked mode, use the quit command. The quit command will stop the running application and then close interactive mode.

**5.2 The Command Line and Profiles**

When you create a Grails application with the [create-app](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-app.html) 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=web-plugin

Profiles encapsulate the project commands, templates and plugins that are designed to work for a given profile. They are stored in the [Grails Profile Repository](https://github.com/grails/grails-profile-repository) on Github.

This repository is checked out locally and stored in the USER\_HOME/.grails/repository directory.

**Understanding a Profile's Structure**

A profile is a simple directory that contains a profile.yml file and directorys containing the "commands", "skeleton" and "templates" defined by the profile. Example:

web

\* commands

\* create-controller.yml

\* run-app.groovy

…

\* 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 defined as follows:

description: Profile **for** Web applications

**extends**: base

As you can see it contains the description of the profile and a definition of which profiles this profile extends, since one profile can extend from another.

When the create-app command runs it takes the skeleton of the parent profiles and copies the skeletons into a new project structure. Child profiles overwrite files from the parent profile so if the parent defines a build.gradle then the child profile will override the parent.

**Defining Profile Commands**

A profile can define new commands that apply only to that profile using YAML or Groovy scripts. Below is an example of the [create-controller](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-controller.html) command 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.nameController.groovy

- command: render

template: templates/testing/Controller.groovy

destination: src/test/groovy/artifact.**package**.path/artifact.nameControllerSpec.groovy

- 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 available step types are:

* 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 implements the [Command](http://grails.org/doc/3.0.x/api/org/grails/cli/profile/Command.html) interface.
* 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 script commands. Each Command script is extends from the [GroovyScriptCommmand](http://grails.org/doc/3.0.x/api/org/grails/cli/profile/commands/script/GroovyScriptCommmand.html) class and hence has all of the methods of that class available to it.

The following is an example of the [create-script](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-script.html) 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 Groovy commands see the following section on creating custom Grails scripts.

**5.3 Creating Custom Scripts**

You can create your own Command scripts by running the [create-script](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-script.html) command from the root of your project. For example the following command:

grails create-script hello-world

Will create a script called src/main/scripts/hello-world.groovy. Each Command script is extends from the [GroovyScriptCommmand](http://grails.org/doc/3.0.x/api/org/grails/cli/profile/commands/script/GroovyScriptCommmand.html) class and hence has all of the methods of that class available to it.

In general Grails scripts should be used for scripting the Gradle based build system and code generation. Scripts cannot load application classes and in fact should not since Gradle is required to construct the application classpath.

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 the script. The following is a more complete example of providing a description taken from thegenerate-all command:

description( "Generates a controller that performs CRUD operations and the associated views" ) {

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 command to be used as follows:

grails generate-all MyClass --force

**5.4 Re-using Grails scripts**

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

Any script you create an 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 arguments:

testApp('--debug-jvm')

**Invoking Gradle**

Instead of invoking another Grails CLI command you can invoke Gradle directory using the gradle property.

gradle.compileGroovy()

**Invoking Ant**

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

ant.mkdir(dir:"path")

**Template Generation**

Plugins and applications that need to define template generation tasks can do so using scripts. A example of this is the Scaffolding plugin which defines the generate-all and generate-controllerscommands.

Every Grails script implements the [TemplateRenderer](http://grails.org/doc/3.0.x/api/org/grails/cli/profile/commands/templates/TemplateRenderer.html) interface which makes it trivial to render templates to the users project workspace.

The following is an example of the [create-script](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-script.html) 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

**5.5 Building with Gradle**

Grails 3.0 uses the [Gradle Build System](http://gradle.org) for build related tasks such as compilation, runnings tests and producing binary distrubutions of your project. It is recommended to use Gradle 2.2 or above with Grails 3.0.

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

When you invoke the grails command the version of Gradle that ships with Grails 3.0 (currently 2.3) is invoked by the grails process via the [Gradle Tooling API](http://www.gradle.org/docs/current/userguide/embedding.html):

# Equivalent to 'gradle classes'

$ grails compile

You can invoke Gradle directly using the gradle command and use your own local version of Gradle, however you will need Gradle 2.2 or above to work with Grails 3.0:

$ gradle assemble

**5.5.1 Defining Dependencies with Gradle**

Dependencies for your project are defined in the dependencies block. In general you can follow the [Gradle documentation on dependency management](http://www.gradle.org/docs/current/userguide/artifact_dependencies_tutorial.html) 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 dependency management plugin which configures a Maven BOM that defines the default dependency versions for certain commonly used dependencies and plugins:

dependencyManagement {

imports {

mavenBom 'org.grails:grails-bom:' + grailsVersion

}

applyMavenExclusions **false**

}

**5.5.2 Working with Gradle Tasks**

As mentioned previously the grails command uses an embedded version of Gradle and certain Grails commands that existed in previous versions of Grails map onto their Gradle equivalents. The following table shows which Grails command invoke which Gradle task:

|  |  |
| --- | --- |
| **Grails Command** | **Gradle Task** |
| clean | clean |
| compile | classes |
| package | assemble |
| run-app | run |
| test-app | test |
| 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 3.0 (Gradle 2.2 or above). If you wish to invoke a Gradle task using the 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 provides TAB completion for the available Gradle 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 the Gradle tasks task:

gradle tasks

**5.5.3 Grails plugins for Gradle**

When you create a new project with the [create-app](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-app.html) command, a default build.gradle is created. The default build.gradle configures the build with a set of Gradle plugins that allow Gradle to build the Grails project:

plugins {

id "io.spring.dependency-management" version "0.3.1.RELEASE"

}

apply plugin: "spring-boot"

apply plugin: "war"

apply plugin: "asset-pipeline"

apply plugin: "org.grails.grails-web"

apply plugin: "org.grails.grails-gsp"

apply plugin: "maven"

The default plugins are as follows:

* dependency-management - The [dependency management](https://plugins.gradle.org/plugin/io.spring.dependency-management) plugin allows Gradle to read Maven BOM files that define the default dependency versions used by Grails.
* spring-boot - The [Spring Boot](http://docs.spring.io/spring-boot/docs/current/reference/html/build-tool-plugins-gradle-plugin.html) Gradle plugin enhances the default packaging tasks provided by Gradle to allow for the creation of runnable JAR/WAR files.
* war - The [WAR plugin](http://www.gradle.org/docs/current/userguide/war_plugin.html) changes the packaging so that Gradle creates as WAR file from you application. You can comment out this plugin if you wish to create only a runnable JAR file for standalone deployment.
* asset-pipeline - The [asset pipeline](https://github.com/bertramdev/asset-pipeline-core) plugin enables the compilation of static assets (JavaScript, CSS etc.)
* maven - The [maven plugin](http://www.gradle.org/docs/current/userguide/maven_plugin.html) allows installing your application into a local maven repository

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

* org.grails.grails-core - The primary Grails plugin for Gradle, included by all other plugins and designed to operate with all profiles.
* org.grails.grails-plugin - A plugin for Gradle for building Grails plugins.
* org.grails.grails-web - The Grails Web gradle plugin configures Gradle to understand the Grails conventions and directory structure.
* org.grails.grails-gsp - The Grails GSP plugin adds precompilation of GSP files for production deployments.
* org.grails.grails-doc - A plugin for Gradle for using Grails 2.0's documentation engine.

**6 Object Relational Mapping (GORM) - Reference Documentation**

**Authors:** Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, Lari Hotari

**Version:** 3.0.1

**Table of Contents**

[**6.1**Quick Start Guide](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#quickStartGuide)

[**6.1.1**Basic CRUD](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#basicCRUD)

[**6.2**Domain Modelling in GORM](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#domainClasses)

[**6.2.1**Association in GORM](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#gormAssociation)

[**6.2.1.1**Many-to-one and one-to-one](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#manyToOneAndOneToOne)

[**6.2.1.2**One-to-many](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#oneToMany)

[**6.2.1.3**Many-to-many](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#manyToMany)

[**6.2.1.4**Basic Collection Types](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#basicCollectionTypes)

[**6.2.2**Composition in GORM](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#gormComposition)

[**6.2.3**Inheritance in GORM](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#inheritanceInGORM)

[**6.2.4**Sets, Lists and Maps](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#sets,ListsAndMaps)

[**6.3**Persistence Basics](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#persistenceBasics)

[**6.3.1**Saving and Updating](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#savingAndUpdating)

[**6.3.2**Deleting Objects](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#deletingObjects)

[**6.3.3**Understanding Cascading Updates and Deletes](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#cascades)

[**6.3.4**Eager and Lazy Fetching](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#fetching)

[**6.3.5**Pessimistic and Optimistic Locking](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#locking)

[**6.3.6**Modification Checking](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#modificationChecking)

[**6.4**Querying with GORM](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#querying)

[**6.4.1**Dynamic Finders](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#finders)

[**6.4.2**Where Queries](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#whereQueries)

[**6.4.3**Criteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#criteria)

[**6.4.4**Detached Criteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#detachedCriteria)

[**6.4.5**Hibernate Query Language (HQL)](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#hql)

[**6.5**Advanced GORM Features](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#advancedGORMFeatures)

[**6.5.1**Events and Auto Timestamping](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#eventsAutoTimestamping)

[**6.5.2**Custom ORM Mapping](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#ormdsl)

[**6.5.2.1**Table and Column Names](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#tableAndColumnNames)

[**6.5.2.2**Caching Strategy](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#caching)

[**6.5.2.3**Inheritance Strategies](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#inheritanceStrategies)

[**6.5.2.4**Custom Database Identity](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#identity)

[**6.5.2.5**Composite Primary Keys](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#compositePrimaryKeys)

[**6.5.2.6**Database Indices](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#databaseIndices)

[**6.5.2.7**Optimistic Locking and Versioning](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#optimisticLockingAndVersioning)

[**6.5.2.8**Eager and Lazy Fetching](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#fetchingDSL)

[**6.5.2.9**Custom Cascade Behaviour](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#customCascadeBehaviour)

[**6.5.2.10**Custom Hibernate Types](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#customHibernateTypes)

[**6.5.2.11**Derived Properties](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#derivedProperties)

[**6.5.2.12**Custom Naming Strategy](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#customNamingStrategy)

[**6.5.3**Default Sort Order](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#defaultSortOrder)

[**6.6**Programmatic Transactions](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#programmaticTransactions)

[**6.7**GORM and Constraints](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\GORM.html#gormConstraints)

**6 Object Relational Mapping (GORM)**

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

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

You can also write Grails domain classes in Java. See the section on Hibernate Integration for how to write domain classes in Java but still use dynamic persistent methods. Below is a preview of GORM in action:

def book = Book.findByTitle("Groovy in Action")

book

.addToAuthors(name:"Dierk Koenig")

.addToAuthors(name:"Guillaume LaForge")

.save()

**6.1 Quick Start Guide**

A domain class can be created with the [create-domain-class](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-domain-class.html) command:

grails create-domain-class helloworld.Person

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

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

**package** helloworld

class Person {

}

If you have the dbCreate property set to "update", "create" or "create-drop" on your [DataSource](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#dataSource), Grails will automatically generate/modify the database tables for you.

You can customize the class by adding properties:

class Person {

String name

Integer age

Date lastVisit

}

Once you have a domain class try and manipulate it with the [shell](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\shell.html) or [console](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\console.html) by typing:

grails console

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

**6.1.1 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\save.html):

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

p.save()

The [save](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\save.html) 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\get.html) method that expects a database identifier to read the Person object back from the database. You can also load an object in a read-only state by using the [read](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\read.html) method:

def p = Person.read(1)

In this case the underlying Hibernate engine will not do any dirty checking and the object will not be persisted. Note that if you explicitly call the [save](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\save.html) 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\load.html) method:

def p = Person.load(1)

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

**Update**

To update an instance, change some properties and then call [save](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\save.html) again:

def p = Person.get(1)

p.name = "Bob"

p.save()

**Delete**

To delete an instance use the [delete](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\delete.html) method:

def p = Person.get(1)

p.delete()

package helloworld

Person.withNewSession{

// delete all

Person.executeUpdate('delete from Person')

// save

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

p.save()

println "save result:\t"

int id = 0

Person.findAll("from Person").each({println it; id = it.id})

// update

def p\_read = Person.get(id)

p\_read.name = "Bob"

p\_read.save()

println "update result:\t"

Person.findAll("from Person").each({println it})

// delete

Person.get(id).delete(flush: true)

println "after delete:\t"

Person.findAll("from Person").each({println it})

}

**6.2 Domain Modelling in GORM**

When building Grails applications you have to consider the problem domain you are trying to solve. For example if you were building an [Amazon](http://www.amazon.com/)-style bookstore you would be thinking about books, authors, customers and publishers to name a few.

These are modeled in GORM as Groovy classes, so a Book class may have a title, a release date, an ISBN number and so on. The next few sections show how to model the domain in GORM.

To create a domain class you run the [create-domain-class](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-domain-class.html) command as follows:

grails create-domain-class org.bookstore.Book

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

**package** org.bookstore

class Book {

}

This class will map automatically to a table in the database called book (the same name as the class). This behaviour is customizable through the [ORM Domain Specific Language](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#ormdsl)

Now that you have a domain class you can define its properties as Java types. For example:

**package** org.bookstore

class Book {

String title

Date releaseDate

String ISBN

}

Each property is mapped to a column in the database, where the convention for column names is all lower case separated by underscores. For example releaseDate maps onto a columnrelease\_date. The SQL types are auto-detected from the Java types, but can be customized with [Constraints](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#constraints) or the [ORM DSL](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#ormdsl).

**6.2.1 Association in GORM**

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

**6.2.1.1 Many-to-one and one-to-one**

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

**Example A**

class Face {

Nose nose

}

class Nose {

}

In this case we have a unidirectional many-to-one relationship from Face to Nose. To make this relationship bidirectional define the other side as follows (and see the section on controlling the ends of the association just below):

**Example B**

class Face {

Nose nose

}

class Nose {

**static** belongsTo = [face:Face]

}

In this case we use the belongsTo setting to say that Nose "belongs to" Face. The result of this is that we can create a Face, attach a Nose instance to it and when we save or delete the Face instance, GORM will save or delete the Nose. In other words, saves and deletes will cascade from Face to the associated Nose:

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

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

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

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

def f = Face.get(1)

f.delete() // both Face and Nose deleted

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

**Example C**

class Face {

**static** hasOne = [nose:Nose]

}

class Nose {

Face face

}

Note that using this property puts the foreign key on the inverse table to the example A, so in this case the foreign key column is stored in the nose table inside a column called face\_id. Also, hasOneonly works with bidirectional relationships.

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

class Face {

**static** hasOne = [nose:Nose]

**static** constraints = {

nose unique: **true**

}

}

class Nose {

Face face

}

**Controlling the ends of the association**

Occasionally you may find yourself with domain classes that have multiple properties of the same type. They may even be self-referential, i.e. the association property has the same type as the domain class it's in. Such situations can cause problems because Grails may guess incorrectly the type of the association. Consider this simple class:

class Person {

String name

Person parent

**static** belongsTo = [ supervisor: Person ]

**static** constraints = { supervisor nullable: **true** }

}

As far as Grails is concerned, the parent and supervisor properties are two directions of the same association. So when you set the parent property on a Person instance, Grails will automatically set the supervisor property on the other Person instance. This may be what you want, but if you look at the class, what we in fact have are two unidirectional relationships.

To guide Grails to the correct mapping, you can tell it that a particular association is unidirectional through the mappedBy property:

class Person {

String name

Person parent

**static** belongsTo = [ supervisor: Person ]

**static** mappedBy = [ supervisor: "none", parent: "none" ]

**static** constraints = { supervisor nullable: **true** }

}

You can also replace "none" with any property name of the target class. And of course this works for normal domain classes too, not just self-referential ones. Nor is the mappedBy property limited to many-to-one and one-to-one associations: it also works for one-to-many and many-to-many associations as you'll see in the next section.

If you have a property called "none" on your domain class, this approach won't work currently! The "none" property will be treated as the reverse direction of the association (or the "back reference"). Fortunately, "none" is not a common domain class property name.

**6.2.1.2 One-to-many**

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

class Author {

**static** hasMany = [books: Book]

String name

}

class Book {

String title

}

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

The [ORM DSL](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#ormdsl) allows mapping unidirectional relationships using a foreign key association instead

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

def a = Author.get(1)

**for** (book in a.books) {

println book.title

}

The default fetch strategy used by Grails is "lazy", which means that the collection will be lazily initialized on first access. This can lead to the [n+1 problem](http://www.javalobby.org/java/forums/t20533.html) if you are not careful.

If you need "eager" fetching you can use the [ORM DSL](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#ormdsl) or specify eager fetching as part of a [query](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#querying)

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

class Author {

**static** hasMany = [books: Book]

String name

}

class Book {

**static** belongsTo = [author: Author]

String title

}

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

class Airport {

**static** hasMany = [flights: Flight]

**static** mappedBy = [flights: "departureAirport"]

}

class Flight {

Airport departureAirport

Airport destinationAirport

}

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

class Airport {

**static** hasMany = [outboundFlights: Flight, inboundFlights: Flight]

**static** mappedBy = [outboundFlights: "departureAirport",

inboundFlights: "destinationAirport"]

}

class Flight {

Airport departureAirport

Airport destinationAirport

}

**6.2.1.3 Many-to-many**

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

class Book {

**static** belongsTo = Author

**static** hasMany = [authors:Author]

String title

}

class Author {

**static** hasMany = [books:Book]

String name

}

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

For example this will work and cascade saves:

**new** Author(name:"Stephen King")

.addToBooks(**new** Book(title:"The Stand"))

.addToBooks(**new** Book(title:"The Shining"))

.save()

However this will only save the Book and not the authors!

**new** Book(name:"Groovy in Action")

.addToAuthors(**new** Author(name:"Dierk Koenig"))

.addToAuthors(**new** Author(name:"Guillaume Laforge"))

.save()

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

Grails' [Scaffolding](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#scaffolding) feature **does not** currently support many-to-many relationship and hence you must write the code to manage the relationship yourself

**6.2.1.4 Basic Collection Types**

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

class Person {

**static** hasMany = [nicknames: String]

}

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

class Person {

**static** hasMany = [nicknames: String]

**static** mapping = {

hasMany joinTable: [name: 'bunch\_o\_nicknames',

key: 'person\_id',

column: 'nickname',

type: "text"]

}

}

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

**bunch\_o\_nicknames Table**

---------------------------------------------

| person\_id | nickname |

---------------------------------------------

| 1 | Fred |

---------------------------------------------

**6.2.2 Composition in GORM**

As well as [association](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#gormAssociation), Grails supports the notion of composition. In this case instead of mapping classes onto separate tables a class can be "embedded" within the current table. For example:

class Person {

Address homeAddress

Address workAddress

**static** embedded = ['homeAddress', 'workAddress']

}

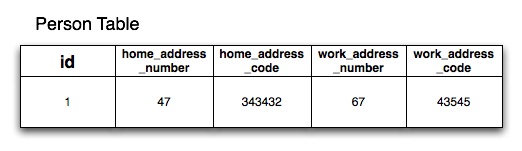
class Address {

String number

String code

}

The resulting mapping would looking like this:



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

**6.2.3 Inheritance in GORM**

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

class Content {

String author

}

class BlogEntry **extends** Content {

URL url

}

class Book **extends** Content {

String ISBN

}

class PodCast **extends** Content {

byte[] audioStream

}

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

**Considerations**

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

Table-per-hierarchy mapping has a down side in that you **cannot** have non-nullable properties with inheritance mapping. An alternative is to use table-per-subclass which can be enabled with the [ORM DSL](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#ormdsl)

However, excessive use of inheritance and table-per-subclass can result in poor query performance due to the use of outer join queries. In general our advice is if you're going to use inheritance, don't abuse it and don't make your inheritance hierarchy too deep.

**Polymorphic Queries**

The upshot of inheritance is that you get the ability to polymorphically query. For example using the [list](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\list.html) method on the Content super class will return all subclasses of Content:

def content = Content.list() // list all blog entries, books and podcasts

content = Content.findAllByAuthor('Joe Bloggs') // find all by author

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

**6.2.4 Sets, Lists and Maps**

**Sets of Objects**

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

class Author {

**static** hasMany = [books: Book]

}

The books property that GORM injects is a java.util.Set. Sets guarantee uniqueness but not order, which may not be what you want. To have custom ordering you configure the Set as a SortedSet:

class Author {

SortedSet books

**static** hasMany = [books: Book]

}

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

class Book **implements** Comparable {

String title

Date releaseDate = **new** Date()

int compareTo(obj) {

releaseDate.compareTo(obj.releaseDate)

}

}

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

**Lists of Objects**

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

class Author {

List books

**static** hasMany = [books: Book]

}

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

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

The way this works at the database level is Hibernate creates a books\_idx column where it saves the index of the elements in the collection to retain this order at the database level.

When using a List, elements must be added to the collection before being saved, otherwise Hibernate will throw an exception (org.hibernate.HibernateException: null index column for collection):

// This won't work!

def book = **new** Book(title: 'The Shining')

book.save()

author.addToBooks(book)

// Do it **this** way instead.

def book = **new** Book(title: 'Misery')

author.addToBooks(book)

author.save()

**Bags of Objects**

If ordering and uniqueness aren't a concern (or if you manage these explicitly) then you can use the Hibernate [Bag](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/collections.html) type to represent mapped collections.

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

class Author {

Collection books

**static** hasMany = [books: Book]

}

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

**Maps of Objects**

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

class Author {

Map books // map of ISBN:book names

}

def a = **new** Author()

a.books = ["1590597583":"Grails Book"]

a.save()

In this case the key and value of the map MUST be strings.

If you want a Map of objects then you can do this:

class Book {

Map authors

**static** hasMany = [authors: Author]

}

def a = **new** Author(name:"Stephen King")

def book = **new** Book()

book.authors = [stephen:a]

book.save()

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

**A Note on Collection Types and Performance**

The Java Set type doesn't allow duplicates. To ensure uniqueness when adding an entry to a Set association Hibernate has to load the entire associations from the database. If you have a large numbers of entries in the association this can be costly in terms of performance.

The same behavior is required for List types, since Hibernate needs to load the entire association to maintain order. Therefore it is recommended that if you anticipate a large numbers of records in the association that you make the association bidirectional so that the link can be created on the inverse side. For example consider the following code:

def book = **new** Book(title:"New Grails Book")

def author = Author.get(1)

book.author = author

book.save()

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

def book = **new** Book(title:"New Grails Book")

def author = Author.get(1)

author.addToBooks(book)

author.save()

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

**6.3 Persistence Basics**

A key thing to remember about Grails is that under the surface Grails is using [Hibernate](http://www.hibernate.org/) for persistence. If you are coming from a background of using [ActiveRecord](http://wiki.rubyonrails.org/rails/pages/ActiveRecord) or [iBatis/MyBatis](http://www.mybatis.org/), Hibernate's "session" model may feel a little strange.

Grails automatically binds a Hibernate session to the currently executing request. This lets you use the [save](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\save.html) and [delete](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\delete.html) methods as well as other GORM methods transparently.

**Transactional Write-Behind**

A useful feature of Hibernate over direct JDBC calls and even other frameworks is that when you call [save](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\save.html) or [delete](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\delete.html) it does not necessarily perform any SQL operations **at that point**. Hibernate batches up SQL statements and executes them as late as possible, often at the end of the request when flushing and closing the session. This is typically done for you automatically by Grails, which manages your Hibernate session.

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

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

**6.3.1 Saving and Updating**

An example of using the [save](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\save.html) method can be seen below:

def p = Person.get(1)

p.save()

This save will be not be pushed to the database immediately - it will be pushed when the next flush occurs. But there are occasions when you want to control when those statements are executed or, in Hibernate terminology, when the session is "flushed". To do so you can use the flush argument to the save method:

def p = Person.get(1)

p.save(flush: **true**)

Note that in this case *all* pending SQL statements including previous saves, deletes, etc. will be synchronized with the database. This also lets you catch any exceptions, which is typically useful in highly concurrent scenarios involving [optimistic locking](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#locking):

def p = Person.get(1)

**try** {

p.save(flush: **true**)

}

**catch** (org.springframework.dao.DataIntegrityViolationException e) {

// deal with exception

}

Another thing to bear in mind is that Grails [validates](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#validation) a domain instance every time you save it. If that validation fails the domain instance will *not* be persisted to the database. By default, save() will simply return null in this case, but if you would prefer it to throw an exception you can use the failOnError argument:

def p = Person.get(1)

**try** {

p.save(failOnError: **true**)

}

**catch** (ValidationException e) {

// deal with exception

}

You can even change the default behaviour with a setting in Config.groovy, as described in the [section on configuration](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#configGORM). Just remember that when you are saving domain instances that have been bound with data provided by the user, the likelihood of validation exceptions is quite high and you won't want those exceptions propagating to the end user.

You can find out more about the subtleties of saving data in [this article](http://blog.springsource.com/2010/06/23/gorm-gotchas-part-1/) - a must read!

**6.3.2 Deleting Objects**

An example of the [delete](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\delete.html) method can be seen below:

def p = Person.get(1)

p.delete()

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

def p = Person.get(1)

p.delete(flush: **true**)

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

def p = Person.get(1)

**try** {

p.delete(flush: **true**)

}

**catch** (org.springframework.dao.DataIntegrityViolationException e) {

flash.message = "Could not delete person ${p.name}"

redirect(action: "show", id: p.id)

}

Note that Grails does not supply a deleteAll method as deleting data is discouraged and can often be avoided through boolean flags/logic.

If you really need to batch delete data you can use the [executeUpdate](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\executeUpdate.html) method to do batch DML statements:

Customer.executeUpdate("delete Customer c where c.name = :oldName",

[oldName: "Fred"])

**6.3.3 Understanding Cascading Updates and Deletes**

It is critical that you understand how cascading updates and deletes work when using GORM. The key part to remember is the belongsTo setting which controls which class "owns" a relationship.

Whether it is a one-to-one, one-to-many or many-to-many, defining belongsTo will result in updates cascading from the owning class to its dependant (the other side of the relationship), and for many-/one-to-one and one-to-many relationships deletes will also cascade.

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

Here is an example:

class Airport {

String name

**static** hasMany = [flights: Flight]

}

class Flight {

String number

**static** belongsTo = [airport: Airport]

}

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

**new** Airport(name: "Gatwick")

.addToFlights(**new** Flight(number: "BA3430"))

.addToFlights(**new** Flight(number: "EZ0938"))

.save()

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

def airport = Airport.findByName("Gatwick")

airport.delete()

However, if I were to remove belongsTo then the above cascading deletion code **would not work**. To understand this better take a look at the summaries below that describe the default behaviour of GORM with regards to specific associations. Also read [part 2](http://blog.springsource.com/2010/07/02/gorm-gotchas-part-2/) of the GORM Gotchas series of articles to get a deeper understanding of relationships and cascading.

**Bidirectional one-to-many with belongsTo**

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

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

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

**Unidirectional one-to-many**

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

class B { }

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

**Bidirectional one-to-many, no belongsTo**

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

class B { A a }

In the case of a bidirectional one-to-many where the many side does not define a belongsTo then the cascade strategy is set to "SAVE-UPDATE" for the one side and "NONE" for the many side.

**Unidirectional one-to-one with belongsTo**

class A { }

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

In the case of a unidirectional one-to-one association that defines a belongsTo then the cascade strategy is set to "ALL" for the owning side of the relationship (A->B) and "NONE" from the side that defines the belongsTo (B->A)

Note that if you need further control over cascading behaviour, you can use the [ORM DSL](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#ormdsl).

**6.3.4 Eager and Lazy Fetching**

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

class Airport {

String name

**static** hasMany = [flights: Flight]

}

class Flight {

String number

Location destination

**static** belongsTo = [airport: Airport]

}

class Location {

String city

String country

}

Given the above domain classes and the following code:

def airport = Airport.findByName("Gatwick")

**for** (flight in airport.flights) {

println flight.destination.city

}

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

**Configuring Eager Fetching**

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

class Airport {

String name

**static** hasMany = [flights: Flight]

**static** mapping = {

flights lazy: **false**

}

}

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

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

Be careful how and where you use eager loading because you could load your entire database into memory with too many eager associations. You can find more information on the mapping options in the [section on the ORM DSL](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#fetchingDSL).

**Using Batch Fetching**

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

class Airport {

String name

**static** hasMany = [flights: Flight]

**static** mapping = {

flights batchSize: 10

}

}

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

class Flight {

…

**static** mapping = {

batchSize 10

}

}

Check out [part 3](http://blog.springsource.com/2010/07/28/gorm-gotchas-part-3/) of the GORM Gotchas series for more in-depth coverage of this tricky topic.

**6.3.5 Pessimistic and Optimistic Locking**

**Optimistic Locking**

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

The version column gets read into a version property that contains the current versioned state of persistent instance which you can access:

def airport = Airport.get(10)

println airport.version

When you perform updates Hibernate will automatically check the version property against the version column in the database and if they differ will throw a [StaleObjectException](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/StaleObjectStateException.html). This will roll back the transaction if one is active.

This is useful as it allows a certain level of atomicity without resorting to pessimistic locking that has an inherit performance penalty. The downside is that you have to deal with this exception if you have highly concurrent writes. This requires flushing the session:

def airport = Airport.get(10)

**try** {

airport.name = "Heathrow"

airport.save(flush: **true**)

}

**catch** (org.springframework.dao.OptimisticLockingFailureException e) {

// deal with exception

}

The way you deal with the exception depends on the application. You could attempt a programmatic merge of the data or go back to the user and ask them to resolve the conflict.

Alternatively, if it becomes a problem you can resort to pessimistic locking.

The version will only be updated after flushing the session.

**Pessimistic Locking**

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

In Grails pessimistic locking is performed on an existing instance with the [lock](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\lock.html) method:

def airport = Airport.get(10)

airport.lock() // lock **for** update

airport.name = "Heathrow"

airport.save()

Grails will automatically deal with releasing the lock for you once the transaction has been committed. However, in the above case what we are doing is "upgrading" from a regular SELECT to a SELECT..FOR UPDATE and another thread could still have updated the record in between the call to get() and the call to lock().

To get around this problem you can use the static [lock](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\lock.html) method that takes an id just like [get](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\get.html):

def airport = Airport.lock(10) // lock **for** update

airport.name = "Heathrow"

airport.save()

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

As well as the [lock](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\lock.html) method you can also obtain a pessimistic locking using queries. For example using a dynamic finder:

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

Or using criteria:

def airport = Airport.createCriteria().get {

eq('name', 'Heathrow')

lock **true**

}

**6.3.6 Modification Checking**

Once you have loaded and possibly modified a persistent domain class instance, it isn't straightforward to retrieve the original values. If you try to reload the instance using [get](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\get.html) Hibernate will return the current modified instance from its Session cache. Reloading using another query would trigger a flush which could cause problems if your data isn't ready to be flushed yet. So GORM provides some methods to retrieve the original values that Hibernate caches when it loads the instance (which it uses for dirty checking).

**isDirty**

You can use the [isDirty](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\isDirty.html) method to check if any field has been modified:

def airport = Airport.get(10)

assert !airport.isDirty()

airport.properties = params

**if** (airport.isDirty()) {

// **do** something based on changed state

}

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

You can also check if individual fields have been modified:

def airport = Airport.get(10)

assert !airport.isDirty()

airport.properties = params

**if** (airport.isDirty('name')) {

// **do** something based on changed name

}

**getDirtyPropertyNames**

You can use the [getDirtyPropertyNames](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\getDirtyPropertyNames.html) method to retrieve the names of modified fields; this may be empty but will not be null:

def airport = Airport.get(10)

assert !airport.isDirty()

airport.properties = params

def modifiedFieldNames = airport.getDirtyPropertyNames()

**for** (fieldName in modifiedFieldNames) {

// **do** something based on changed value

}

**getPersistentValue**

You can use the [getPersistentValue](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\getPersistentValue.html) method to retrieve the value of a modified field:

def airport = Airport.get(10)

assert !airport.isDirty()

airport.properties = params

def modifiedFieldNames = airport.getDirtyPropertyNames()

**for** (fieldName in modifiedFieldNames) {

def currentValue = airport."$fieldName"

def originalValue = airport.getPersistentValue(fieldName)

**if** (currentValue != originalValue) {

// **do** something based on changed value

}

}

**6.4 Querying with GORM**

GORM supports a number of powerful ways to query from dynamic finders, to criteria to Hibernate's object oriented query language HQL. Depending on the complexity of the query you have the following options in order of flexibility and power:

* Dynamic Finders
* Where Queries
* Criteria Queries
* Hibernate Query Language (HQL)

In addition, Groovy's ability to manipulate collections with [GPath](http://groovy.codehaus.org/GPath) and methods like sort, findAll and so on combined with GORM results in a powerful combination.

However, let's start with the basics.

**Listing instances**

Use the [list](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\list.html) method to obtain all instances of a given class:

def books = Book.list()

The [list](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\list.html) method supports arguments to perform pagination:

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

as well as sorting:

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

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

**Retrieval by Database Identifier**

The second basic form of retrieval is by database identifier using the [get](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\get.html) method:

def book = Book.get(23)

You can also obtain a list of instances for a set of identifiers using [getAll](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\getAll.html):

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

**6.4.1 Dynamic Finders**

GORM supports the concept of dynamic finders. A dynamic finder looks like a static method invocation, but the methods themselves don't actually exist in any form at the code level.

Instead, a method is auto-magically generated using code synthesis at runtime, based on the properties of a given class. Take for example the Book class:

class Book {

String title

Date releaseDate

Author author

}

class Author {

String name

}

The Book class has properties such as title, releaseDate and author. These can be used by the [findBy](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\findBy.html) and [findAllBy](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\findAllBy.html) methods in the form of "method expressions":

def book = Book.findByTitle("The Stand")

book = Book.findByTitleLike("Harry Pot%")

book = Book.findByReleaseDateBetween(firstDate, secondDate)

book = Book.findByReleaseDateGreaterThan(someDate)

book = Book.findByTitleLikeOrReleaseDateLessThan("%Something%", someDate)

**Method Expressions**

A method expression in GORM is made up of the prefix such as [findBy](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\findBy.html) followed by an expression that combines one or more properties. The basic form is:

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

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

def book = Book.findByTitle("The Stand")

book = Book.findByTitleLike("Harry Pot%")

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

The possible comparators include:

* InList - In the list of given values
* LessThan - less than a given value
* LessThanEquals - less than or equal a give value
* GreaterThan - greater than a given value
* GreaterThanEquals - greater than or equal a given value
* Like - Equivalent to a SQL like expression
* Ilike - Similar to a Like, except case insensitive
* NotEqual - Negates equality
* InRange - Between the from and to values of a Groovy Range
* Rlike - Performs a Regexp LIKE in MySQL or Oracle otherwise falls back to Like
* Between - Between two values (requires two arguments)
* IsNotNull - Not a null value (doesn't take an argument)
* IsNull - Is a null value (doesn't take an argument)

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

def now = **new** Date()

def lastWeek = now - 7

def book = Book.findByReleaseDateBetween(lastWeek, now)

books = Book.findAllByReleaseDateIsNull()

books = Book.findAllByReleaseDateIsNotNull()

**Boolean logic (AND/OR)**

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

def books = Book.findAllByTitleLikeAndReleaseDateGreaterThan(

"%Java%", **new** Date() - 30)

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

def books = Book.findAllByTitleLikeOrReleaseDateGreaterThan(

"%Java%", **new** Date() - 30)

You can combine as many criteria as you like, but they must all be combined with And or all Or. If you need to combine And and Or or if the number of criteria creates a very long method name, just convert the query to a [Criteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#criteria) or [HQL](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#hql) query.

**Querying Associations**

Associations can also be used within queries:

def author = Author.findByName("Stephen King")

def books = author ? Book.findAllByAuthor(author) : []

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

**Pagination and Sorting**

The same pagination and sorting parameters available on the [list](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\list.html) method can also be used with dynamic finders by supplying a map as the final parameter:

def books = Book.findAllByTitleLike("Harry Pot%",

[max: 3, offset: 2, sort: "title", order: "desc"])

**6.4.2 Where Queries**

The where method, introduced in Grails 2.0, builds on the support for [Detached Criteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#detachedCriteria) by providing an enhanced, compile-time checked query DSL for common queries. The where method is more flexible than dynamic finders, less verbose than criteria and provides a powerful mechanism to compose queries.

**Basic Querying**

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

def query = Person.where {

firstName == "Bart"

}

Person bart = query.find()

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

class Person {

**static** simpsons = where {

lastName == "Simpson"

}

…

}

…

Person.simpsons.each {

println it.firstname

}

Query execution is lazy and only happens upon usage of the [DetachedCriteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#detachedCriteria) instance. If you want to execute a where-style query immediately there are variations of the findAll and find methods to accomplish this:

def results = Person.findAll {

lastName == "Simpson"

}

def results = Person.findAll(sort:"firstName") {

lastName == "Simpson"

}

Person p = Person.find { firstName == "Bart" }

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

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

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

def query = Person.where {

(lastName != "Simpson" && firstName != "Fred") || (firstName == "Bart" && age > 9)

}

def results = query.list(sort:"firstName")

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

def query = Person.where {

firstName ==~ ~/B.+/

}

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

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

def query = Person.where {

age in 18..65

}

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

def query = Person.where {

middleName == **null**

}

**Query Composition**

Since the return value of the where method is a [DetachedCriteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#detachedCriteria) instance you can compose new queries from the original query:

def query = Person.where {

lastName == "Simpson"

}

def bartQuery = query.where {

firstName == "Bart"

}

Person p = bartQuery.find()

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

def callable = {

lastName == "Simpson"

}

def query = Person.where(callable)

The above must be written as follows:

**import** grails.gorm.DetachedCriteria

def callable = {

lastName == "Simpson"

} as DetachedCriteria<Person>

def query = Person.where(callable)

As you can see the closure definition is cast (using the Groovy as keyword) to a [DetachedCriteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#detachedCriteria) instance targeted at the Person class.

**Conjunction, Disjunction and Negation**

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

def query = Person.where {

(lastName != "Simpson" && firstName != "Fred") || (firstName == "Bart" && age > 9)

}

You can also negate a logical comparison using !:

def query = Person.where {

firstName == "Fred" && !(lastName == 'Simpson')

}

**Property Comparison Queries**

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

def query = Person.where {

firstName == lastName

}

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

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

**Querying Associations**

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

def query = Pet.where {

owner.firstName == "Joe" || owner.firstName == "Fred"

}

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

def query = Person.where {

pets { name == "Jack" || name == "Joe" }

}

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

def query = Person.where {

pets { name == "Jack" } || firstName == "Ed"

}

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

def query = Person.where {

pets.size() == 2

}

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

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

**Subqueries**

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

**final** query = Person.where {

age > avg(age)

}

The following table lists the possible subqueries:

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

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

def query = Person.where {

age > avg(age).of { lastName == "Simpson" } && firstName == "Homer"

}

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

Person.where {

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

}

**Other Functions**

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

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

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

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

def query = Pet.where {

year(birthDate) == 2011

}

You can also apply functions to associations:

def query = Person.where {

year(pets.birthDate) == 2009

}

**Batch Updates and Deletes**

Since each where method call returns a [DetachedCriteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#detachedCriteria) instance, you can use where queries to execute batch operations such as batch updates and deletes. For example, the following query will update all people with the surname "Simpson" to have the surname "Bloggs":

def query = Person.where {

lastName == 'Simpson'

}

int total = query.updateAll(lastName:"Bloggs")

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

To batch delete records you can use the deleteAll method:

def query = Person.where {

lastName == 'Simpson'

}

int total = query.deleteAll()

**6.4.3 Criteria**

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

Criteria can be used either with the [createCriteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\createCriteria.html) or [withCriteria](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\withCriteria.html) methods. The builder uses Hibernate's Criteria API. The nodes on this builder map the static methods found in the [Restrictions](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/criterion/Restrictions.html) class of the Hibernate Criteria API. For example:

def c = Account.createCriteria()

def results = c {

between("balance", 500, 1000)

eq("branch", "London")

or {

like("holderFirstName", "Fred%")

like("holderFirstName", "Barney%")

}

maxResults(10)

order("holderLastName", "desc")

}

This criteria will select up to 10 Account objects in a List matching the following criteria:

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

The results will be sorted in descending order by holderLastName.

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

**Conjunctions and Disjunctions**

As demonstrated in the previous example you can group criteria in a logical OR using an or { } block:

or {

between("balance", 500, 1000)

eq("branch", "London")

}

This also works with logical AND:

and {

between("balance", 500, 1000)

eq("branch", "London")

}

And you can also negate using logical NOT:

not {

between("balance", 500, 1000)

eq("branch", "London")

}

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

**Querying Associations**

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

class Account {

…

**static** hasMany = [transactions: Transaction]

…

}

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

def c = Account.createCriteria()

def now = **new** Date()

def results = c.list {

transactions {

between('date', now - 10, now)

}

}

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

def c = Account.createCriteria()

def now = **new** Date()

def results = c.list {

or {

between('created', now - 10, now)

transactions {

between('date', now - 10, now)

}

}

}

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

**Querying with Projections**

Projections may be used to customise the results. Define a "projections" node within the criteria builder tree to use projections. There are equivalent methods within the projections node to the methods found in the Hibernate [Projections](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/criterion/Projections.html) class:

def c = Account.createCriteria()

def numberOfBranches = c.get {

projections {

countDistinct('branch')

}

}

When multiple fields are specified in the projection, a List of values will be returned. A single value will be returned otherwise.

**SQL Projections**

The criteria DSL provides access to Hibernate's SQL projection API.

// Box is a domain class…

class Box {

int width

int height

}

// Use SQL projections to retrieve the perimeter and area of all of the Box instances…

def c = Box.createCriteria()

def results = c.list {

projections {

sqlProjection '(2 \* (width + height)) as perimeter, (width \* height) as area', ['perimeter', 'area'], [INTEGER, INTEGER]

}

}

The first argument to the sqlProjection method is the SQL which defines the projections. The second argument is a list of Strings which represent column aliases corresponding to the projected values expressed in the SQL. The third argument is a list of org.hibernate.type.Type instances which correspond to the projected values expressed in the SQL. The API supports allorg.hibernate.type.Type objects but constants like INTEGER, LONG, FLOAT etc. are provided by the DSL which correspond to all of the types defined inorg.hibernate.type.StandardBasicTypes.

Consider that the following table represents the data in the BOX table.

|  |  |
| --- | --- |
| **width** | **height** |
| 2 | 7 |
| 2 | 8 |
| 2 | 9 |
| 4 | 9 |

The query above would return results like this:

[[18, 14], [20, 16], [22, 18], [26, 36]]

Each of the inner lists contains the 2 projected values for each Box, perimeter and area.

Note that if there are other references in scope wherever your criteria query is expressed that have names that conflict with any of the type constants described above, the code in your criteria will refer to those references, not the type constants provided by the DSL. In the unlikely event of that happening you can disambiguate the conflict by referring to the fully qualified Hibernate type. For example StandardBasicTypes.INTEGER instead of INTEGER.

If only 1 value is being projected, the alias and the type do not need to be included in a list.

def results = c.list {

projections {

sqlProjection 'sum(width \* height) as totalArea', 'totalArea', INTEGER

}

}

That query would return a single result with the value of 84 as the total area of all of the Box instances.

The DSL supports grouped projections with the sqlGroupProjection method.

def results = c.list {

projections {

sqlGroupProjection 'width, sum(height) as combinedHeightsForThisWidth', 'width', ['width', 'combinedHeightsForThisWidth'], [INTEGER, INTEGER]

}

}

The first argument to the sqlGroupProjection method is the SQL which defines the projections. The second argument represents the group by clause that should be part of the query. That string may be single column name or a comma separated list of column names. The third argument is a list of Strings which represent column aliases corresponding to the projected values expressed in the SQL. The fourth argument is a list of org.hibernate.type.Type instances which correspond to the projected values expressed in the SQL.

The query above is projecting the combined heights of boxes grouped by width and would return results that look like this:

[[2, 24], [4, 9]]

Each of the inner lists contains 2 values. The first value is a box width and the second value is the sum of the heights of all of the boxes which have that width.

**Using SQL Restrictions**

You can access Hibernate's SQL Restrictions capabilities.

def c = Person.createCriteria()

def peopleWithShortFirstNames = c.list {

sqlRestriction "char\_length(first\_name) <= 4"

}

SQL Restrictions may be parameterized to deal with SQL injection vulnerabilities related to dynamic restrictions.

def c = Person.createCriteria()

def peopleWithShortFirstNames = c.list {

sqlRestriction "char\_length(first\_name) < ? AND char\_length(first\_name) > ?", [maxValue, minValue]

}

Note that the parameter there is SQL. The first\_name attribute referenced in the example refers to the persistence model, not the object model like in HQL queries. The Person property named firstName is mapped to the first\_name column in the database and you must refer to that in the sqlRestrictionstring.

Also note that the SQL used here is not necessarily portable across databases.

**Using Scrollable Results**

You can use Hibernate's [ScrollableResults](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/ScrollableResults.html) feature by calling the scroll method:

def results = crit.scroll {

maxResults(10)

}

def f = results.first()

def l = results.last()

def n = results.next()

def p = results.previous()

def **future** = results.scroll(10)

def accountNumber = results.getLong('number')

To quote the documentation of Hibernate ScrollableResults:

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

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

**Setting properties in the Criteria instance**

If a node within the builder tree doesn't match a particular criterion it will attempt to set a property on the Criteria object itself. This allows full access to all the properties in this class. This example callssetMaxResults and setFirstResult on the [Criteria](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/Criteria.html) instance:

**import** org.hibernate.FetchMode as FM

…

def results = c.list {

maxResults(10)

firstResult(50)

fetchMode("aRelationship", FM.JOIN)

}

**Querying with Eager Fetching**

In the section on [Eager and Lazy Fetching](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#fetching) we discussed how to declaratively specify fetching to avoid the N+1 SELECT problem. However, this can also be achieved using a criteria query:

def criteria = Task.createCriteria()

def tasks = criteria.list{

eq "assignee.id", task.assignee.id

join 'assignee'

join 'project'

order 'priority', 'asc'

}

Notice the usage of the join method: it tells the criteria API to use a JOIN to fetch the named associations with the Task instances. It's probably best not to use this for one-to-many associations though, because you will most likely end up with duplicate results. Instead, use the 'select' fetch mode:

**import** org.hibernate.FetchMode as FM

…

def results = Airport.withCriteria {

eq "region", "EMEA"

fetchMode "flights", FM.SELECT

}

Although this approach triggers a second query to get the flights association, you will get reliable results - even with the maxResults option.

fetchMode and join are general settings of the query and can only be specified at the top-level, i.e. you cannot use them inside projections or association constraints.

An important point to bear in mind is that if you include associations in the query constraints, those associations will automatically be eagerly loaded. For example, in this query:

def results = Airport.withCriteria {

eq "region", "EMEA"

flights {

like "number", "BA%"

}

}

the flights collection would be loaded eagerly via a join even though the fetch mode has not been explicitly set.

**Method Reference**

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

c { … }

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

c.list { … }

|  |  |
| --- | --- |
| **Method** | **Description** |
| **list** | This is the default method. It returns all matching rows. |
| **get** | Returns a unique result set, i.e. just one row. The criteria has to be formed that way, that it only queries one row. This method is not to be confused with a limit to just the first row. |
| **scroll** | Returns a scrollable result set. |
| **listDistinct** | If subqueries or associations are used, one may end up with the same row multiple times in the result set, this allows listing only distinct entities and is equivalent toDISTINCT\_ROOT\_ENTITY of the [CriteriaSpecification](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/criterion/CriteriaSpecification.html) class. |
| **count** | Returns the number of matching rows. |

**Combining Criteria**

You can combine multiple criteria closures in the following way:

def emeaCriteria = {

eq "region", "EMEA"

}

def results = Airport.withCriteria {

emeaCriteria.delegate = delegate

emeaCriteria()

flights {

like "number", "BA%"

}

}

This technique requires that each criteria must refer to the same domain class (i.e. Airport). A more flexible approach is to use Detached Criteria, as described in the following section.

**6.4.4 Detached Criteria**

Detached Criteria are criteria queries that are not associated with any given database session/connection. Supported since Grails 2.0, Detached Criteria queries have many uses including allowing you to create common reusable criteria queries, execute subqueries and execute batch updates/deletes.

**Building Detached Criteria Queries**

The primary point of entry for using the Detached Criteria is the grails.gorm.DetachedCriteria class which accepts a domain class as the only argument to its constructor:

**import** grails.gorm.\*

…

def criteria = **new** DetachedCriteria(Person)

Once you have obtained a reference to a detached criteria instance you can execute [where](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#whereQueries) queries or criteria queries to build up the appropriate query. To build a normal criteria query you can use thebuild method:

def criteria = **new** DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

Note that methods on the DetachedCriteria instance **do not** mutate the original object but instead return a new query. In other words, you have to use the return value of the build method to obtain the mutated criteria object:

def criteria = **new** DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

def bartQuery = criteria.build {

eq 'firstName', 'Bart'

}

**Executing Detached Criteria Queries**

Unlike regular criteria, Detached Criteria are lazy, in that no query is executed at the point of definition. Once a Detached Criteria query has been constructed then there are a number of useful query methods which are summarized in the table below:

|  |  |
| --- | --- |
| **Method** | **Description** |
| **list** | List all matching entities |
| **get** | Return a single matching result |
| **count** | Count all matching records |
| **exists** | Return true if any matching records exist |
| **deleteAll** | Delete all matching records |
| **updateAll(Map)** | Update all matching records with the given properties |

As an example the following code will list the first 4 matching records sorted by the firstName property:

def criteria = **new** DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

def results = criteria.list(max:4, sort:"firstName")

You can also supply additional criteria to the list method:

def results = criteria.list(max:4, sort:"firstName") {

gt 'age', 30

}

To retrieve a single result you can use the get or find methods (which are synonyms):

Person p = criteria.find() // or criteria.get()

The DetachedCriteria class itself also implements the Iterable interface which means that it can be treated like a list:

def criteria = **new** DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

criteria.each {

println it.firstName

}

In this case the query is only executed when the each method is called. The same applies to all other Groovy collection iteration methods.

You can also execute dynamic finders on DetachedCriteria just like on domain classes. For example:

def criteria = **new** DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

def bart = criteria.findByFirstName("Bart")

**Using Detached Criteria for Subqueries**

Within the context of a regular criteria query you can use DetachedCriteria to execute subquery. For example if you want to find all people who are older than the average age the following query will accomplish that:

def results = Person.withCriteria {

gt "age", **new** DetachedCriteria(Person).build {

projections {

avg "age"

}

}

order "firstName"

}

Notice that in this case the subquery class is the same as the original criteria query class (i.e. Person) and hence the query can be shortened to:

def results = Person.withCriteria {

gt "age", {

projections {

avg "age"

}

}

order "firstName"

}

If the subquery class differs from the original criteria query then you will have to use the original syntax.

In the previous example the projection ensured that only a single result was returned (the average age). If your subquery returns multiple results then there are different criteria methods that need to be used to compare the result. For example to find all the people older than the ages 18 to 65 a gtAll query can be used:

def results = Person.withCriteria {

gtAll "age", {

projections {

property "age"

}

between 'age', 18, 65

}

order "firstName"

}

The following table summarizes criteria methods for operating on subqueries that return multiple results:

|  |  |
| --- | --- |
| **Method** | **Description** |
| **gtAll** | greater than all subquery results |
| **geAll** | greater than or equal to all subquery results |
| **ltAll** | less than all subquery results |
| **leAll** | less than or equal to all subquery results |
| **eqAll** | equal to all subquery results |
| **neAll** | not equal to all subquery results |

**Batch Operations with Detached Criteria**

The DetachedCriteria class can be used to execute batch operations such as batch updates and deletes. For example, the following query will update all people with the surname "Simpson" to have the surname "Bloggs":

def criteria = **new** DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

int total = criteria.updateAll(lastName:"Bloggs")

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

To batch delete records you can use the deleteAll method:

def criteria = **new** DetachedCriteria(Person).build {

eq 'lastName', 'Simpson'

}

int total = criteria.deleteAll()

**6.4.5 Hibernate Query Language (HQL)**

GORM classes also support Hibernate's query language HQL, a very complete reference for which can be found [in the Hibernate documentation](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/queryhql.html) of the Hibernate documentation.

GORM provides a number of methods that work with HQL including [find](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\find.html), [findAll](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\findAll.html) and [executeQuery](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\executeQuery.html). An example of a query can be seen below:

def results =

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

**Positional and Named Parameters**

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

def results =

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

def author = Author.findByName("Stephen King")

def books = Book.findAll("from Book as book where book.author = ?",

[author])

Or even named parameters:

def results =

Book.findAll("from Book as b " +

"where b.title like :search or b.author like :search",

[search: "The Shi%"])

def author = Author.findByName("Stephen King")

def books = Book.findAll("from Book as book where book.author = :author",

[author: author])

**Multiline Queries**

Use the line continuation character to separate the query across multiple lines:

def results = Book.findAll("\

from Book as b, \

Author as a \

where b.author = a and a.surname = ?", ['Smith'])

Triple-quoted Groovy multiline Strings will NOT work with HQL queries.

**Pagination and Sorting**

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

def results =

Book.findAll("from Book as b where " +

"b.title like 'Lord of the%' " +

"order by b.title asc",

[max: 10, offset: 20])

**6.5 Advanced GORM Features**

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

**6.5.1 Events and Auto Timestamping**

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

* beforeInsert - Executed before an object is initially persisted to the database. If you return false, the insert will be cancelled.
* beforeUpdate - Executed before an object is updated. If you return false, the update will be cancelled.
* beforeDelete - Executed before an object is deleted. If you return false, the delete will be cancelled.
* beforeValidate - Executed before an object is validated
* afterInsert - Executed after an object is persisted to the database
* afterUpdate - Executed after an object has been updated
* afterDelete - Executed after an object has been deleted
* onLoad - Executed when an object is loaded from the database

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

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

**Event types**

**The beforeInsert event**

Fired before an object is saved to the database

class Person {

**private** **static** **final** Date NULL\_DATE = **new** Date(0)

String firstName

String lastName

Date signupDate = NULL\_DATE

def beforeInsert() {

**if** (signupDate == NULL\_DATE) {

signupDate = **new** Date()

}

}

}

**The beforeUpdate event**

Fired before an existing object is updated

class Person {

def securityService

String firstName

String lastName

String lastUpdatedBy

**static** constraints = {

lastUpdatedBy nullable: **true**

}

def beforeUpdate() {

lastUpdatedBy = securityService.currentAuthenticatedUsername()

}

}

**The beforeDelete event**

Fired before an object is deleted.

class Person {

String name

def beforeDelete() {

ActivityTrace.withNewSession {

**new** ActivityTrace(eventName: "Person Deleted", data: name).save()

}

}

}

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

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

**The beforeValidate event**

Fired before an object is validated.

class Person {

String name

**static** constraints = {

name size: 5..45

}

def beforeValidate() {

name = name?.trim()

}

}

The beforeValidate method is run before any validators are run.

Validation may run more often than you think. It is triggered by the validate() and save() methods as you'd expect, but it is also typically triggered just before the view is rendered as well. So when writing beforeValidate() implementations, make sure that they can handle being called multiple times with the same property values.

GORM supports an overloaded version of beforeValidate which accepts a List parameter which may include the names of the properties which are about to be validated. This version ofbeforeValidate will be called when the validate method has been invoked and passed a List of property names as an argument.

class Person {

String name

String town

Integer age

**static** constraints = {

name size: 5..45

age range: 4..99

}

def beforeValidate(List propertiesBeingValidated) {

// **do** pre validation work based on propertiesBeingValidated

}

}

def p = **new** Person(name: 'Jacob Brown', age: 10)

p.validate(['age', 'name'])

Note that when validate is triggered indirectly because of a call to the save method that the validate method is being invoked with no arguments, not a Listthat includes all of the property names.

Either or both versions of beforeValidate may be defined in a domain class. GORM will prefer the List version if a List is passed to validate but will fall back on the no-arg version if the Listversion does not exist. Likewise, GORM will prefer the no-arg version if no arguments are passed to validate but will fall back on the List version if the no-arg version does not exist. In that case,null is passed to beforeValidate.

**The onLoad/beforeLoad event**

Fired immediately before an object is loaded from the database:

class Person {

String name

Date dateCreated

Date lastUpdated

def onLoad() {

log.debug "Loading ${id}"

}

}

beforeLoad() is effectively a synonym for onLoad(), so only declare one or the other.

**The afterLoad event**

Fired immediately after an object is loaded from the database:

class Person {

String name

Date dateCreated

Date lastUpdated

def afterLoad() {

name = "I'm loaded"

}

}

**Custom Event Listeners**

As of Grails 2.0 there is a new API for plugins and applications to register and listen for persistence events. This API is not tied to Hibernate and also works for other persistence plugins such as the[MongoDB plugin for GORM](http://grails.org/plugin/mongodb).

To use this API you need to subclass AbstractPersistenceEventListener (in package *org.grails.datastore.mapping.engine.event* ) and implement the methods onPersistenceEvent andsupportsEventType. You also must provide a reference to the datastore to the listener. The simplest possible implementation can be seen below:

**public** MyPersistenceListener(**final** Datastore datastore) {

**super**(datastore)

}

@Override

**protected** void onPersistenceEvent(**final** AbstractPersistenceEvent event) {

**switch**(event.eventType) {

**case** PreInsert:

println "PRE INSERT ${event.entityObject}"

**break**

**case** PostInsert:

println "POST INSERT ${event.entityObject}"

**break**

**case** PreUpdate:

println "PRE UPDATE ${event.entityObject}"

**break**;

**case** PostUpdate:

println "POST UPDATE ${event.entityObject}"

**break**;

**case** PreDelete:

println "PRE DELETE ${event.entityObject}"

**break**;

**case** PostDelete:

println "POST DELETE ${event.entityObject}"

**break**;

**case** PreLoad:

println "PRE LOAD ${event.entityObject}"

**break**;

**case** PostLoad:

println "POST LOAD ${event.entityObject}"

**break**;

}

}

@Override

**public** boolean supportsEventType(Class<? **extends** ApplicationEvent> eventType) {

**return** **true**

}

The AbstractPersistenceEvent class has many subclasses (PreInsertEvent, PostInsertEvent etc.) that provide further information specific to the event. A cancel() method is also provided on the event which allows you to veto an insert, update or delete operation.

Once you have created your event listener you need to register it with the ApplicationContext. This can be done in BootStrap.groovy:

def init = {

application.mainContext.eventTriggeringInterceptor.datastores.each { k, datastore ->

applicationContext.addApplicationListener **new** MyPersistenceListener(datastore)

}

}

or use this in a plugin:

def doWithApplicationContext = { applicationContext ->

application.mainContext.eventTriggeringInterceptor.datastores.each { k, datastore ->

applicationContext.addApplicationListener **new** MyPersistenceListener(datastore)

}

}

**Hibernate Events**

It is generally encouraged to use the non-Hibernate specific API described above, but if you need access to more detailed Hibernate events then you can define custom Hibernate-specific event listeners.

You can also register event handler classes in an application's grails-app/conf/spring/resources.groovy or in the doWithSpring closure in a plugin descriptor by registering a Spring bean named hibernateEventListeners. This bean has one property, listenerMap which specifies the listeners to register for various Hibernate events.

The values of the Map are instances of classes that implement one or more Hibernate listener interfaces. You can use one class that implements all of the required interfaces, or one concrete class per interface, or any combination. The valid Map keys and corresponding interfaces are listed here:

|  |  |
| --- | --- |
| **Name** | **Interface** |
| auto-flush | [AutoFlushEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/AutoFlushEventListener.html) |
| merge | [MergeEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/MergeEventListener.html) |
| create | [PersistEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PersistEventListener.html) |
| create-onflush | [PersistEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PersistEventListener.html) |
| delete | [DeleteEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/DeleteEventListener.html) |
| dirty-check | [DirtyCheckEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/DirtyCheckEventListener.html) |
| evict | [EvictEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/EvictEventListener.html) |
| flush | [FlushEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/FlushEventListener.html) |
| flush-entity | [FlushEntityEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/FlushEntityEventListener.html) |
| load | [LoadEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/LoadEventListener.html) |
| load-collection | [InitializeCollectionEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/InitializeCollectionEventListener.html) |
| lock | [LockEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/LockEventListener.html) |
| refresh | [RefreshEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/RefreshEventListener.html) |
| replicate | [ReplicateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/ReplicateEventListener.html) |
| save-update | [SaveOrUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/SaveOrUpdateEventListener.html) |
| save | [SaveOrUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/SaveOrUpdateEventListener.html) |
| update | [SaveOrUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/SaveOrUpdateEventListener.html) |
| pre-load | [PreLoadEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreLoadEventListener.html) |
| pre-update | [PreUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreUpdateEventListener.html) |
| pre-delete | [PreDeleteEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreDeleteEventListener.html) |
| pre-insert | [PreInsertEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreInsertEventListener.html) |
| pre-collection-recreate | [PreCollectionRecreateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreCollectionRecreateEventListener.html) |
| pre-collection-remove | [PreCollectionRemoveEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreCollectionRemoveEventListener.html) |
| pre-collection-update | [PreCollectionUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PreCollectionUpdateEventListener.html) |
| post-load | [PostLoadEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostLoadEventListener.html) |
| post-update | [PostUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostUpdateEventListener.html) |
| post-delete | [PostDeleteEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostDeleteEventListener.html) |
| post-insert | [PostInsertEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostInsertEventListener.html) |
| post-commit-update | [PostUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostUpdateEventListener.html) |
| post-commit-delete | [PostDeleteEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostDeleteEventListener.html) |
| post-commit-insert | [PostInsertEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostInsertEventListener.html) |
| post-collection-recreate | [PostCollectionRecreateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostCollectionRecreateEventListener.html) |
| post-collection-remove | [PostCollectionRemoveEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostCollectionRemoveEventListener.html) |
| post-collection-update | [PostCollectionUpdateEventListener](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/event/PostCollectionUpdateEventListener.html) |

For example, you could register a class AuditEventListener which implements PostInsertEventListener, PostUpdateEventListener, and PostDeleteEventListener using the following in an application:

beans = {

auditListener(AuditEventListener)

hibernateEventListeners(HibernateEventListeners) {

listenerMap = ['post-insert': auditListener,

'post-update': auditListener,

'post-delete': auditListener]

}

}

or use this in a plugin:

def doWithSpring = {

auditListener(AuditEventListener)

hibernateEventListeners(HibernateEventListeners) {

listenerMap = ['post-insert': auditListener,

'post-update': auditListener,

'post-delete': auditListener]

}

}

**Automatic timestamping**

If you define a dateCreated property it will be set to the current date for you when you create new instances. Likewise, if you define a lastUpdated property it will be automatically be updated for you when you change persistent instances.

If this is not the behaviour you want you can disable this feature with:

class Person {

Date dateCreated

Date lastUpdated

**static** mapping = {

autoTimestamp **false**

}

}

If you have nullable: false constraints on either dateCreated or lastUpdated, your domain instances will fail validation - probably not what you want. Omit constraints from these properties unless you disable automatic timestamping.

**6.5.2 Custom ORM Mapping**

Grails domain classes can be mapped onto many legacy schemas with an Object Relational Mapping DSL (domain specific language). The following sections takes you through what is possible with the ORM DSL.

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

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

class Person {

…

**static** mapping = {

version **false**

autoTimestamp **false**

}

}

You can also configure global mappings in Config.groovy (or an external config file) using this setting:

grails.gorm.**default**.mapping = {

version **false**

autoTimestamp **false**

}

It has the same syntax as the standard mapping block but it applies to all your domain classes! You can then override these defaults within the mapping block of a domain class.

**6.5.2.1 Table and Column Names**

**Table names**

The database table name which the class maps to can be customized using the table method:

class Person {

…

**static** mapping = {

table 'people'

}

}

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

**Column names**

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

class Person {

String firstName

**static** mapping = {

table 'people'

firstName column: 'First\_Name'

}

}

Here firstName is a dynamic method within the mapping Closure that has a single Map parameter. Since its name corresponds to a domain class persistent field, the parameter values (in this case just"column") are used to configure the mapping for that property.

**Column type**

GORM supports configuration of Hibernate types with the DSL using the type attribute. This includes specifying user types that implement the Hibernate [org.hibernate.usertype.UserType](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/usertype/UserType.html) interface, which allows complete customization of how a type is persisted. As an example if you had a PostCodeType you could use it as follows:

class Address {

String number

String postCode

**static** mapping = {

postCode type: PostCodeType

}

}

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

class Address {

String number

String postCode

**static** mapping = {

postCode type: 'text'

}

}

This would make the postCode column map to the default large-text type for the database you're using (for example TEXT or CLOB).

See the Hibernate documentation regarding [Basic Types](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/mapping.html#mapping-types-basictypes) for further information.

**Many-to-One/One-to-One Mappings**

In the case of associations it is also possible to configure the foreign keys used to map associations. In the case of a many-to-one or one-to-one association this is exactly the same as any regular column. For example consider the following:

class Person {

String firstName

Address address

**static** mapping = {

table 'people'

firstName column: 'First\_Name'

address column: 'Person\_Address\_Id'

}

}

By default the address association would map to a foreign key column called address\_id. By using the above mapping we have changed the name of the foreign key column to Person\_Adress\_Id.

**One-to-Many Mapping**

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

class Person {

String firstName

**static** hasMany = [addresses: Address]

**static** mapping = {

table 'people'

firstName column: 'First\_Name'

addresses column: 'Person\_Address\_Id'

}

}

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

class Person {

String firstName

**static** hasMany = [addresses: Address]

**static** mapping = {

table 'people'

firstName column: 'First\_Name'

addresses joinTable: [name: 'Person\_Addresses',

key: 'Person\_Id',

column: 'Address\_Id']

}

}

**Many-to-Many Mapping**

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

class Group {

…

**static** hasMany = [people: Person]

}

class Person {

…

**static** belongsTo = Group

**static** hasMany = [groups: Group]

}

In this case Grails will create a join table called group\_person containing foreign keys called person\_id and group\_id referencing the person and group tables. To change the column names you can specify a column within the mappings for each class.

class Group {

…

**static** mapping = {

people column: 'Group\_Person\_Id'

}

}

class Person {

…

**static** mapping = {

groups column: 'Group\_Group\_Id'

}

}

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

class Group {

…

**static** mapping = {

people column: 'Group\_Person\_Id',

joinTable: 'PERSON\_GROUP\_ASSOCIATIONS'

}

}

class Person {

…

**static** mapping = {

groups column: 'Group\_Group\_Id',

joinTable: 'PERSON\_GROUP\_ASSOCIATIONS'

}

}

**6.5.2.2 Caching Strategy**

**Setting up caching**

[Hibernate](http://www.hibernate.org/) features a second-level cache with a customizable cache provider. This needs to be configured in the grails-app/conf/DataSource.groovy file as follows:

hibernate {

cache.use\_second\_level\_cache=**true**

cache.use\_query\_cache=**true**

cache.provider\_class='org.hibernate.cache.EhCacheProvider'

}

You can customize any of these settings, for example to use a distributed caching mechanism.

For further reading on caching and in particular Hibernate's second-level cache, refer to the [Hibernate documentation](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/performance.html#performance-cache) on the subject.

**Caching instances**

Call the cache method in your mapping block to enable caching with the default settings:

class Person {

…

**static** mapping = {

table 'people'

cache **true**

}

}

This will configure a 'read-write' cache that includes both lazy and non-lazy properties. You can customize this further:

class Person {

…

**static** mapping = {

table 'people'

cache usage: 'read-only', include: 'non-lazy'

}

}

**Caching associations**

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

class Person {

String firstName

**static** hasMany = [addresses: Address]

**static** mapping = {

table 'people'

version **false**

addresses column: 'Address', cache: **true**

}

}

class Address {

String number

String postCode

}

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

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

to further configure the cache usage.

**Caching Queries**

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

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

In order for the results of the query to be cached, you must enable caching in your mapping as discussed in the previous section.

You can also cache criteria queries:

def people = Person.withCriteria {

like('firstName', 'Fr%')

cache **true**

}

**Cache usages**

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

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

**6.5.2.3 Inheritance Strategies**

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

class Payment {

Integer amount

**static** mapping = {

tablePerHierarchy **false**

}

}

class CreditCardPayment **extends** Payment {

String cardNumber

}

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

**6.5.2.4 Custom Database Identity**

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

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

class Person {

…

**static** mapping = {

table 'people'

version **false**

id generator: 'hilo',

params: [table: 'hi\_value',

column: 'next\_value',

max\_lo: 100]

}

}

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

For more information on the different Hibernate generators refer to the [Hibernate reference documentation](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/mapping.html#mapping-declaration-id-generator)

Although you don't typically specify the id field (Grails adds it for you) you can still configure its mapping like the other properties. For example to customise the column for the id property you can do:

class Person {

…

**static** mapping = {

table 'people'

version **false**

id column: 'person\_id'

}

}

**6.5.2.5 Composite Primary Keys**

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

**import** org.apache.commons.lang.builder.HashCodeBuilder

class Person **implements** Serializable {

String firstName

String lastName

boolean equals(other) {

**if** (!(other **instanceof** Person)) {

**return** **false**

}

other.firstName == firstName && other.lastName == lastName

}

int hashCode() {

def builder = **new** HashCodeBuilder()

builder.append firstName

builder.append lastName

builder.toHashCode()

}

**static** mapping = {

id composite: ['firstName', 'lastName']

}

}

The above will create a composite id of the firstName and lastName properties of the Person class. To retrieve an instance by id you use a prototype of the object itself:

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

println p.firstName

Domain classes mapped with composite primary keys must implement the Serializable interface and override the equals and hashCode methods, using the properties in the composite key for the calculations. The example above uses a HashCodeBuilder for convenience but it's fine to implement it yourself.

Another important consideration when using composite primary keys is associations. If for example you have a many-to-one association where the foreign keys are stored in the associated table then 2 columns will be present in the associated table.

For example consider the following domain class:

class Address {

Person person

}

In this case the address table will have an additional two columns called person\_first\_name and person\_last\_name. If you wish the change the mapping of these columns then you can do so using the following technique:

class Address {

Person person

**static** mapping = {

columns {

person {

column name: "FirstName"

column name: "LastName"

}

}

}

}

**6.5.2.6 Database Indices**

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

class Person {

String firstName

String address

**static** mapping = {

table 'people'

version **false**

id column: 'person\_id'

firstName column: 'First\_Name', index: 'Name\_Idx'

address column: 'Address', index: 'Name\_Idx,Address\_Index'

}

}

Note that you cannot have any spaces in the value of the index attribute; in this example index:'Name\_Idx, Address\_Index' will cause an error.

**6.5.2.7 Optimistic Locking and Versioning**

As discussed in the section on [Optimistic and Pessimistic Locking](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#locking), by default GORM uses optimistic locking and automatically injects a version property into every class which is in turn mapped to aversion column at the database level.

If you're mapping to a legacy schema that doesn't have version columns (or there's some other reason why you don't want/need this feature) you can disable this with the version method:

class Person {

…

**static** mapping = {

table 'people'

version **false**

}

}

If you disable optimistic locking you are essentially on your own with regards to concurrent updates and are open to the risk of users losing data (due to data overriding) unless you use [pessimistic locking](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#locking)

**Version columns types**

By default Grails maps the version property as a Long that gets incremented by one each time an instance is updated. But Hibernate also supports using a Timestamp, for example:

**import** java.sql.Timestamp

class Person {

…

Timestamp version

**static** mapping = {

table 'people'

}

}

There's a slight risk that two updates occurring at nearly the same time on a fast server can end up with the same timestamp value but this risk is very low. One benefit of using a Timestamp instead of aLong is that you combine the optimistic locking and last-updated semantics into a single column.

**6.5.2.8 Eager and Lazy Fetching**

**Lazy Collections**

As discussed in the section on [Eager and Lazy fetching](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#fetching), GORM collections are lazily loaded by default but you can change this behaviour with the ORM DSL. There are several options available to you, but the most common ones are:

* lazy: false
* fetch: 'join'

and they're used like this:

class Person {

String firstName

Pet pet

**static** hasMany = [addresses: Address]

**static** mapping = {

addresses lazy: **false**

pet fetch: 'join'

}

}

class Address {

String street

String postCode

}

class Pet {

String name

}

The first option, lazy: false , ensures that when a Person instance is loaded, its addresses collection is loaded at the same time with a second SELECT. The second option is basically the same, except the collection is loaded with a JOIN rather than another SELECT. Typically you want to reduce the number of queries, so fetch: 'join' is the more appropriate option. On the other hand, it could feasibly be the more expensive approach if your domain model and data result in more and larger results than would otherwise be necessary.

For more advanced users, the other settings available are:

1. batchSize: N
2. lazy: false, batchSize: N

where N is an integer. These let you fetch results in batches, with one query per batch. As a simple example, consider this mapping for Person:

class Person {

String firstName

Pet pet

**static** mapping = {

pet batchSize: 5

}

}

If a query returns multiple Person instances, then when we access the first pet property, Hibernate will fetch that Pet plus the four next ones. You can get the same behaviour with eager loading by combining batchSize with the lazy: false option. You can find out more about these options in the [Hibernate user guide](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/performance.html#performance-fetching) and this [primer on fetching strategies](http://community.jboss.org/wiki/AShortPrimerOnFetchingStrategies). Note that ORM DSL does not currently support the "subselect" fetching strategy.

**Lazy Single-Ended Associations**

In GORM, one-to-one and many-to-one associations are by default lazy. Non-lazy single ended associations can be problematic when you load many entities because each non-lazy association will result in an extra SELECT statement. If the associated entities also have non-lazy associations, the number of queries grows significantly!

Use the same technique as for lazy collections to make a one-to-one or many-to-one association non-lazy/eager:

class Person {

String firstName

}

class Address {

String street

String postCode

**static** belongsTo = [person: Person]

**static** mapping = {

person lazy: **false**

}

}

Here we configure GORM to load the associated Person instance (through the person property) whenever an Address is loaded.

**Lazy Single-Ended Associations and Proxies**

Hibernate uses runtime-generated proxies to facilitate single-ended lazy associations; Hibernate dynamically subclasses the entity class to create the proxy.

Consider the previous example but with a lazily-loaded person association: Hibernate will set the person property to a proxy that is a subclass of Person. When you call any of the getters (except for the id property) or setters on that proxy, Hibernate will load the entity from the database.

Unfortunately this technique can produce surprising results. Consider the following example classes:

class Pet {

String name

}

class Dog **extends** Pet {

}

class Person {

String name

Pet pet

}

and assume that we have a single Person instance with a Dog as the pet. The following code will work as you would expect:

def person = Person.get(1)

assert person.pet **instanceof** Dog

assert Pet.get(person.petId) **instanceof** Dog

But this won't:

def person = Person.get(1)

assert person.pet **instanceof** Dog

assert Pet.list()[0] **instanceof** Dog

The second assertion fails, and to add to the confusion, this will work:

assert Pet.list()[0] **instanceof** Dog

What's going on here? It's down to a combination of how proxies work and the guarantees that the Hibernate session makes. When you load the Person instance, Hibernate creates a proxy for its petrelation and attaches it to the session. Once that happens, whenever you retrieve that Pet instance with a query, a get(), or the pet relation *within the same session* , Hibernate gives you the proxy.

Fortunately for us, GORM automatically unwraps the proxy when you use get() and findBy\*(), or when you directly access the relation. That means you don't have to worry at all about proxies in the majority of cases. But GORM doesn't do that for objects returned with a query that returns a list, such as list() and findAllBy\*(). However, if Hibernate hasn't attached the proxy to the session, those queries will return the real instances - hence why the last example works.

You can protect yourself to a degree from this problem by using the instanceOf method by GORM:

def person = Person.get(1)

assert Pet.list()[0].instanceOf(Dog)

However, it won't help here if casting is involved. For example, the following code will throw a ClassCastException because the first pet in the list is a proxy instance with a class that is neither Dog nor a sub-class of Dog:

def person = Person.get(1)

Dog pet = Pet.list()[0]

Of course, it's best not to use static types in this situation. If you use an untyped variable for the pet instead, you can access any Dog properties or methods on the instance without any problems.

These days it's rare that you will come across this issue, but it's best to be aware of it just in case. At least you will know why such an error occurs and be able to work around it.

**6.5.2.9 Custom Cascade Behaviour**

As described in the section on [cascading updates](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#cascades), the primary mechanism to control the way updates and deletes cascade from one association to another is the static [belongsTo](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\belongsTo.html) property.

However, the ORM DSL gives you complete access to Hibernate's [transitive persistence](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/objectstate.html#objectstate-transitive) capabilities using the cascade attribute.

Valid settings for the cascade attribute include:

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

It is advisable to read the section in the Hibernate documentation on [transitive persistence](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/objectstate.html#objectstate-transitive) to obtain a better understanding of the different cascade styles and recommendations for their usage

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

class Person {

String firstName

**static** hasMany = [addresses: Address]

**static** mapping = {

addresses cascade: "all-delete-orphan"

}

}

class Address {

String street

String postCode

}

**6.5.2.10 Custom Hibernate Types**

You saw in an earlier section that you can use composition (with the embedded property) to break a table into multiple objects. You can achieve a similar effect with Hibernate's custom user types. These are not domain classes themselves, but plain Java or Groovy classes. Each of these types also has a corresponding "meta-type" class that implements [org.hibernate.usertype.UserType](http://docs.jboss.org/hibernate/core/3.6/javadocs/org/hibernate/usertype/UserType.html).

The [Hibernate reference manual](http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/mapping.html#mapping-types-custom) has some information on custom types, but here we will focus on how to map them in Grails. Let's start by taking a look at a simple domain class that uses an old-fashioned (pre-Java 1.5) type-safe enum class:

class Book {

String title

String author

Rating rating

**static** mapping = {

rating type: RatingUserType

}

}

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

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

class Book {

String title

Name author

Rating rating

**static** mapping = {

author type: NameUserType, {

column name: "first\_name"

column name: "last\_name"

}

rating type: RatingUserType

}

}

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

column name: "first\_name", index: "my\_idx", unique: **true**

The column definitions do *not* support the following attributes: type, cascade, lazy, cache, and joinTable.

One thing to bear in mind with custom types is that they define the *SQL types* for the corresponding database columns. That helps take the burden of configuring them yourself, but what happens if you have a legacy database that uses a different SQL type for one of the columns? In that case, override the column's SQL type using the sqlType attribute:

class Book {

String title

Name author

Rating rating

**static** mapping = {

author type: NameUserType, {

column name: "first\_name", sqlType: "text"

column name: "last\_name", sqlType: "text"

}

rating type: RatingUserType, sqlType: "text"

}

}

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

**6.5.2.11 Derived Properties**

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

class Product {

Float price

Float taxRate

Float tax

}

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

class Product {

Float price

Float taxRate

Float tax

**static** mapping = {

tax formula: 'PRICE \* TAX\_RATE'

}

}

Note that the formula expressed in the ORM DSL is SQL so references to other properties should relate to the persistence model not the object model, which is why the example refers to PRICE andTAX\_RATE instead of price and taxRate.

With that in place, when a Product is retrieved with something like Product.get(42), the SQL that is generated to support that will look something like this:

select

product0\_.id as id1\_0\_,

product0\_.version as version1\_0\_,

product0\_.price as price1\_0\_,

product0\_.tax\_rate as tax4\_1\_0\_,

product0\_.PRICE \* product0\_.TAX\_RATE as formula1\_0\_

from

product product0\_

where

product0\_.id=?

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

Product.findAllByTaxGreaterThan(21.12)

Derived properties may be referenced in the Criteria API:

Product.withCriteria {

gt 'tax', 21.12f

}

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

select

this\_.id as id1\_0\_,

this\_.version as version1\_0\_,

this\_.price as price1\_0\_,

this\_.tax\_rate as tax4\_1\_0\_,

this\_.PRICE \* this\_.TAX\_RATE as formula1\_0\_

from

product this\_

where

this\_.PRICE \* this\_.TAX\_RATE>?

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

**6.5.2.12 Custom Naming Strategy**

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

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

dataSource {

pooled = **true**

dbCreate = "create-drop"

…

}

hibernate {

cache.use\_second\_level\_cache = **true**

…

naming\_strategy = com.myco.myproj.CustomNamingStrategy

}

You can also specify the name of the class and it will be loaded for you:

hibernate {

…

naming\_strategy = 'com.myco.myproj.CustomNamingStrategy'

}

A third option is to provide an instance if there is some configuration required beyond calling the default constructor:

hibernate {

…

def strategy = **new** com.myco.myproj.CustomNamingStrategy()

// configure as needed

naming\_strategy = strategy

}

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

**package** com.myco.myproj

**import** org.hibernate.cfg.ImprovedNamingStrategy

**import** org.hibernate.util.StringHelper

class CustomNamingStrategy **extends** ImprovedNamingStrategy {

String classToTableName(String className) {

"table\_" + StringHelper.unqualify(className)

}

String propertyToColumnName(String propertyName) {

"col\_" + StringHelper.unqualify(propertyName)

}

}

**6.5.3 Default Sort Order**

You can sort objects using query arguments such as those found in the [list](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\list.html) method:

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

However, you can also declare the default sort order for a collection in the mapping:

class Airport {

…

**static** mapping = {

sort "name"

}

}

The above means that all collections of Airport instances will by default be sorted by the airport name. If you also want to change the sort *order* , use this syntax:

class Airport {

…

**static** mapping = {

sort name: "desc"

}

}

Finally, you can configure sorting at the association level:

class Airport {

…

**static** hasMany = [flights: Flight]

**static** mapping = {

flights sort: 'number', order: 'desc'

}

}

In this case, the flights collection will always be sorted in descending order of flight number.

These mappings will not work for default unidirectional one-to-many or many-to-many relationships because they involve a join table. See [this issue](http://jira.codehaus.org/browse/GRAILS-4089) for more details. Consider using a SortedSet or queries with sort parameters to fetch the data you need.

**6.6 Programmatic Transactions**

Grails is built on Spring and uses Spring's Transaction abstraction for dealing with programmatic transactions. However, GORM classes have been enhanced to make this simpler with the[withTransaction](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\withTransaction.html) method. This method has a single parameter, a Closure, which has a single parameter which is a Spring [TransactionStatus](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/transaction/TransactionStatus.html) instance.

Here's an example of using withTransaction in a controller methods:

def transferFunds() {

Account.withTransaction { status ->

def source = Account.get(params.from)

def dest = Account.get(params.to)

def amount = params.amount.toInteger()

**if** (source.active) {

**if** (dest.active) {

source.balance -= amount

dest.amount += amount

}

**else** {

status.setRollbackOnly()

}

}

}

}

In this example we rollback the transaction if the destination account is not active. Also, if an unchecked Exception or Error (but not a checked Exception, even though Groovy doesn't require that you catch checked exceptions) is thrown during the process the transaction will automatically be rolled back.

You can also use "save points" to rollback a transaction to a particular point in time if you don't want to rollback the entire transaction. This can be achieved through the use of Spring's[SavePointManager](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/transaction/SavepointManager.html) interface.

The withTransaction method deals with the begin/commit/rollback logic for you within the scope of the block.

**6.7 GORM and Constraints**

Although constraints are covered in the [Validation](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#constraints) section, it is important to mention them here as some of the constraints can affect the way in which the database schema is generated.

Where feasible, Grails uses a domain class's constraints to influence the database columns generated for the corresponding domain class properties.

Consider the following example. Suppose we have a domain model with the following properties:

String name

String description

By default, in MySQL, Grails would define these columns as

|  |  |
| --- | --- |
| **Column** | **Data Type** |
| name | varchar(255) |
| description | varchar(255) |

But perhaps the business rules for this domain class state that a description can be up to 1000 characters in length. If that were the case, we would likely define the column as follows *if* we were creating the table with an SQL script.

|  |  |
| --- | --- |
| **Column** | **Data Type** |
| description | TEXT |

Chances are we would also want to have some application-based validation to make sure we don't exceed that 1000 character limit *before* we persist any records. In Grails, we achieve this validation with [constraints](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#constraints). We would add the following constraint declaration to the domain class.

**static** constraints = {

description maxSize: 1000

}

This constraint would provide both the application-based validation we want and it would also cause the schema to be generated as shown above. Below is a description of the other constraints that influence schema generation.

**Constraints Affecting String Properties**

* [inList](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\inList.html)
* [maxSize](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\maxSize.html)
* [size](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\size.html)

If either the maxSize or the size constraint is defined, Grails sets the maximum column length based on the constraint value.

In general, it's not advisable to use both constraints on the same domain class property. However, if both the maxSize constraint and the size constraint are defined, then Grails sets the column length to the minimum of the maxSize constraint and the upper bound of the size constraint. (Grails uses the minimum of the two, because any length that exceeds that minimum will result in a validation error.)

If the inList constraint is defined (and the maxSize and the size constraints are not defined), then Grails sets the maximum column length based on the length of the longest string in the list of valid values. For example, given a list including values "Java", "Groovy", and "C++", Grails would set the column length to 6 (i.e., the number of characters in the string "Groovy").

**Constraints Affecting Numeric Properties**

* [min](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\min.html)
* [max](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\max.html)
* [range](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\range.html)

If the max, min, or range constraint is defined, Grails attempts to set the column precision based on the constraint value. (The success of this attempted influence is largely dependent on how Hibernate interacts with the underlying DBMS.)

In general, it's not advisable to combine the pair min/max and range constraints together on the same domain class property. However, if both of these constraints is defined, then Grails uses the minimum precision value from the constraints. (Grails uses the minimum of the two, because any length that exceeds that minimum precision will result in a validation error.)

* [scale](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\scale.html)

If the scale constraint is defined, then Grails attempts to set the column [scale](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\scale.html) based on the constraint value. This rule only applies to floating point numbers (i.e., java.lang.Float, java.Lang.Double,java.lang.BigDecimal, or subclasses of java.lang.BigDecimal). The success of this attempted influence is largely dependent on how Hibernate interacts with the underlying DBMS.

The constraints define the minimum/maximum numeric values, and Grails derives the maximum number of digits for use in the precision. Keep in mind that specifying only one of min/max constraints will not affect schema generation (since there could be large negative value of property with max:100, for example), unless the specified constraint value requires more digits than default Hibernate column precision is (19 at the moment). For example:

someFloatValue max: 1000000, scale: 3

would yield:

someFloatValue DECIMAL(19, 3) // precision is **default**

but

someFloatValue max: 12345678901234567890, scale: 5

would yield:

someFloatValue DECIMAL(25, 5) // precision = digits in max + scale

and

someFloatValue max: 100, min: -100000

would yield:

someFloatValue DECIMAL(8, 2) // precision = digits in min + **default** scale

**7 The Web Layer - Reference Documentation**

**Authors:** Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, Lari Hotari

**Version:** 3.0.1

**Table of Contents**

[**7.1**Controllers](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#controllers)

[**7.1.1**Understanding Controllers and Actions](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#understandingControllersAndActions)

[**7.1.2**Controllers and Scopes](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#controllersAndScopes)

[**7.1.3**Models and Views](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#modelsAndViews)

[**7.1.4**Redirects and Chaining](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#redirectsAndChaining)

[**7.1.5**Controller Interceptors](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#controllerInterceptors)

[**7.1.6**Data Binding](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#dataBinding)

[**7.1.7**XML and JSON Responses](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#xmlAndJSON)

[**7.1.8**More on JSONBuilder](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#moreOnJSONBuilder)

[**7.1.9**Uploading Files](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#uploadingFiles)

[**7.1.10**Command Objects](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#commandObjects)

[**7.1.11**Handling Duplicate Form Submissions](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#formtokens)

[**7.1.12**Simple Type Converters](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#typeConverters)

[**7.1.13**Declarative Controller Exception Handling](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#controllerExceptionHandling)

[**7.2**Groovy Server Pages](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#gsp)

[**7.2.1**GSP Basics](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#GSPBasics)

[**7.2.1.1**Variables and Scopes](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#variablesAndScopes)

[**7.2.1.2**Logic and Iteration](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#logicAndIteration)

[**7.2.1.3**Page Directives](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#pageDirectives)

[**7.2.1.4**Expressions](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#expressions)

[**7.2.2**GSP Tags](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#tags)

[**7.2.2.1**Variables and Scopes](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#tagVariablesAndScopes)

[**7.2.2.2**Logic and Iteration](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#tagLogicAndIteration)

[**7.2.2.3**Search and Filtering](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#searchAndFiltering)

[**7.2.2.4**Links and Resources](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#linksAndResources)

[**7.2.2.5**Forms and Fields](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#formsAndFields)

[**7.2.2.6**Tags as Method Calls](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#tagsAsMethodCalls)

[**7.2.3**Views and Templates](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#viewsAndTemplates)

[**7.2.4**Layouts with Sitemesh](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#layouts)

[**7.2.5**Static Resources](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#resources)

[**7.2.6**Sitemesh Content Blocks](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#sitemeshContentBlocks)

[**7.2.7**Making Changes to a Deployed Application](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#makingChangesToADeployedApplication)

[**7.2.8**GSP Debugging](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#GSPDebugging)

[**7.3**Tag Libraries](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#taglibs)

[**7.3.1**Variables and Scopes](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#taglibVariablesAndScopes)

[**7.3.2**Simple Tags](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#simpleTags)

[**7.3.3**Logical Tags](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#logicalTags)

[**7.3.4**Iterative Tags](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#iterativeTags)

[**7.3.5**Tag Namespaces](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#namespaces)

[**7.3.6**Using JSP Tag Libraries](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#usingJSPTagLibraries)

[**7.3.7**Tag return value](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#tagReturnValue)

[**7.4**URL Mappings](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#urlmappings)

[**7.4.1**Mapping to Controllers and Actions](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#mappingToControllersAndActions)

[**7.4.2**Mapping to REST resources](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#restfulMappings)

[**7.4.3**Redirects In URL Mappings](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#redirectMappings)

[**7.4.4**Embedded Variables](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#embeddedVariables)

[**7.4.5**Mapping to Views](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#mappingToViews)

[**7.4.6**Mapping to Response Codes](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#mappingToResponseCodes)

[**7.4.7**Mapping to HTTP methods](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#mappingHTTP)

[**7.4.8**Mapping Wildcards](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#mappingWildcards)

[**7.4.9**Automatic Link Re-Writing](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#automaticLinkRewriting)

[**7.4.10**Applying Constraints](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#applyingConstraints)

[**7.4.11**Named URL Mappings](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#namedMappings)

[**7.4.12**Customizing URL Formats](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#customizingUrlFormat)

[**7.4.13**Namespaced Controllers](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#namespacedControllers)

[**7.5**Interceptors](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#interceptors)

[**7.5.1**Defining Interceptors](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#definingInterceptors)

[**7.5.2**Matching Requests with Inteceptors](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#interceptorMatching)

[**7.5.3**Ordering Interceptor Execution](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#interceptorOrdering)

[**7.6**Content Negotiation](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\theWebLayer.html#contentNegotiation)

**7 The Web Layer**

**7.1 Controllers**

A controller handles requests and creates or prepares the response. A controller can generate the response directly or delegate to a view. To create a controller, simply create a class whose name ends with Controller in the grails-app/controllers directory (in a subdirectory if it's in a package).

The default [URL Mapping](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#urlmappings) configuration ensures that the first part of your controller name is mapped to a URI and each action defined within your controller maps to URIs within the controller name URI.

**7.1.1 Understanding Controllers and Actions**

**Creating a controller**

Controllers can be created with the [create-controller](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-controller.html) or [generate-controller](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\generate-controller.html) command. For example try running the following command from the root of a Grails project:

grails create-controller book

The command will create a controller at the location grails-app/controllers/myapp/BookController.groovy:

**package** myapp

class BookController {

def index() { }

}

where "myapp" will be the name of your application, the default package name if one isn't specified.

BookController by default maps to the /book URI (relative to your application root).

The create-controller and generate-controller commands are just for convenience and you can just as easily create controllers using your favorite text editor or IDE

**Creating Actions**

A controller can have multiple public action methods; each one maps to a URI:

class BookController {

def list() {

// **do** controller logic

// create model

**return** model

}

}

This example maps to the /book/list URI by default thanks to the property being named list.

**Public Methods as Actions**

In earlier versions of Grails actions were implemented with Closures. This is still supported, but the preferred approach is to use methods.

Leveraging methods instead of Closure properties has some advantages:

* Memory efficient
* Allow use of stateless controllers (singleton scope)
* You can override actions from subclasses and call the overridden superclass method with super.actionName()
* Methods can be intercepted with standard proxying mechanisms, something that is complicated to do with Closures since they're fields.

If you prefer the Closure syntax or have older controller classes created in earlier versions of Grails and still want the advantages of using methods, you can set thegrails.compile.artefacts.closures.convert property to true in BuildConfig.groovy:

grails.compile.artefacts.closures.convert = **true**

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

If a controller class extends some other class which is not defined under the grails-app/controllers/ directory, methods inherited from that class are not converted to controller actions. If the intent is to expose those inherited methods as controller actions the methods may be overridden in the subclass and the subclass method may invoke the method in the super class.

**The Default Action**

A controller has the concept of a default URI that maps to the root URI of the controller, for example /book for BookController. The action that is 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"

**7.1.2 Controllers and Scopes**

**Available Scopes**

Scopes are hash-like objects where you can store variables. The following scopes are available to controllers:

* [servletContext](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\servletContext.html) - Also known as application scope, this scope lets you share state across the entire web application. The servletContext is an instance of [ServletContext](http://download.oracle.com/javaee/1.4/api/javax/servlet/ServletContext.html)
* [session](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\session.html) - The session allows associating state with a given user and typically uses cookies to associate a session with a client. The session object is an instance of [HttpSession](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpSession.html)
* [request](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\request.html) - The request object allows the storage of objects for the current request only. The request object is an instance of [HttpServletRequest](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpServletRequest.html)
* [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) - Mutable map of incoming request query string or POST parameters
* [flash](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\flash.html) - See below

**Accessing Scopes**

Scopes can be accessed using the variable names above in combination with Groovy's array index operator, even on classes provided by the Servlet API such as the [HttpServletRequest](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpServletRequest.html):

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 clear:

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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\flash.html) scope as a temporary store to make attributes available for this request and the next request only. Afterwards the attributes are cleared. This is useful for setting a message directly before redirecting, for example:

def delete() {

def b = Book.get(params.id)

**if** (!b) {

flash.message = "User not found **for** id ${params.id}"

redirect(action:list)

}

… // remaining code

}

When the list action is requested, the message value will be in scope and can be used to display an information message. It will be removed from the flash scope after this second request.

Note that the attribute name can be anything you want, and the values are often strings used to display messages, but can be any object type.

**Scoped Controllers**

Supported controller scopes are:

* prototype (default) - A new controller will be created for each request (recommended for actions as Closure properties)
* session - One controller is created for the scope of a user session
* singleton - Only one instance of the controller ever exists (recommended for actions as methods)

To enable one of the scopes, add a static scope property to your class with one of the valid scope values listed above, for example

**static** scope = "singleton"

You can define the default strategy under in Config.groovy with the grails.controllers.defaultScope key, for example:

grails.controllers.defaultScope = "singleton"

Newly created applications have the grails.controllers.defaultScope property set in grails-app/conf/Config.groovy with a value of "singleton". You may change this value to any of the supported scopes listed above. If the property is not assigned a value at all, controllers will default to "prototype" scope.

Use scoped controllers wisely. For instance, we don't recommend having any properties in a singleton-scoped controller since they will be shared for *all* requests.

**7.1.3 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 names accessible by the view. There are a couple of ways to return a model. First, you can explicitly return a Map instance:

def show() {

[book: Book.get(params.id)]

}

The above does *not* reflect what you should use with the scaffolding views - see the [scaffolding section](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#scaffolding) for more details.

A more advanced approach is to return an instance of the Spring [ModelAndView](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/web/servlet/ModelAndView.html) class:

**import** org.springframework.web.servlet.ModelAndView

def index() {

// get some books just **for** the index page, perhaps your favorites

def favoriteBooks = ...

// forward to the list view to show them

**return** **new** ModelAndView("/book/list", [ bookList : favoriteBooks ])

}

One thing to bear in mind is that certain variable names can not be used in your model:

* attributes
* application

Currently, no error will be reported if you do use them, but this will hopefully change in a future version of Grails.

**Selecting the View**

In both of the previous two examples there was no code that specified which [view](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#gsp) to render. So how does Grails know which one to pick? The answer lies in the conventions. Grails will look for a view at the location grails-app/views/book/show.gsp for this show action:

class BookController {

def show() {

[book: Book.get(params.id)]

}

}

To render a different view, use the [render](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\render.html) method:

def show() {

def map = [book: Book.get(params.id)]

render(view: "display", model: map)

}

In this case Grails will attempt to render a view at the location grails-app/views/book/display.gsp. Notice that Grails automatically qualifies the view location with the book directory of the grails-app/views directory. This is convenient, but to access shared views you need instead you can use an absolute path instead of a relative one:

def show() {

def map = [book: Book.get(params.id)]

render(view: "/shared/display", model: map)

}

In this case Grails will attempt to render a view at the location grails-app/views/shared/display.gsp.

Grails also supports JSPs as views, so if a GSP isn't found in the expected location but a JSP is, it will be used instead.

**Selecting Views For Namespaced Controllers**

If a controller defines a namespace for itself with the [namespace](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#namespacedControllers) property that will affect the root directory in which Grails will look for views which are specified with a relative path. The default root directory for views rendered by a namespaced controller is grails-app/views/<namespace name>/<controller name>/. If the view is not found in the namespaced directory then Grails will fallback to looking for the view in the non-namespaced directory.

See the example below.

class ReportingController {

**static** namespace = 'business'

def humanResources() {

// This will render grails-app/views/business/reporting/humanResources.gsp

// **if** it exists.

// If grails-app/views/business/reporting/humanResources.gsp does not

// exist the fallback will be grails-app/views/reporting/humanResources.gsp.

// 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 response directly from the controller. For this, the highly flexible render method can be used:

render "Hello World!"

The above code writes the text "Hello World!" to the response. Other examples include:

// write some markup

render {

**for** (b in books) {

div(id: b.id, b.title)

}

}

// render a specific view

render(view: 'show')

// render a template **for** each item in a collection

render(template: 'book\_template', collection: Book.list())

// render some text with encoding and content type

render(text: "<xml>some xml</xml>", contentType: "text/xml", encoding: "UTF-8")

If you plan on using Groovy's MarkupBuilder to generate HTML for use with the render method be careful of naming clashes between HTML elements and Grails tags, for example:

**import** groovy.xml.MarkupBuilder

…

def login() {

def writer = **new** StringWriter()

def builder = **new** MarkupBuilder(writer)

builder.html {

head {

title 'Log in'

}

body {

h1 'Hello'

form {

}

}

}

def html = writer.toString()

render html

}

This will actually [call the form tag](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#tagsAsMethodCalls) (which will return some text that will be ignored by the MarkupBuilder). To correctly output a <form> element, use the following:

def login() {

// …

body {

h1 'Hello'

builder.form {

}

}

// …

}

**7.1.4 Redirects and Chaining**

**Redirects**

Actions can be redirected using the [redirect](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\redirect.html) controller method:

class OverviewController {

def login() {}

def find() {

**if** (!session.user)

redirect(action: 'login')

**return**

}

…

}

}

Internally the [redirect](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\redirect.html) method uses the [HttpServletResponse](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpServletResponse.html) 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 method:

redirect(action: 'myaction', params: [myparam: "myvalue"])

These parameters are made available through the [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) dynamic property that accesses request parameters. If a parameter is specified with the same name as a request parameter, the request parameter is overridden and the controller parameter is used.

Since the params object is a Map, you can use it to pass the current request parameters from one action to the next:

redirect(action: "next", params: params)

Finally, you can also include a fragment in the target URI:

redirect(controller: "test", action: "show", fragment: "profile")

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 example calling the first action in this action:

class ExampleChainController {

def first() {

chain(action: second, model: [one: 1])

}

def second () {

chain(action: third, model: [two: 2])

}

def third() {

[three: 3])

}

}

results in the model:

[one: 1, two: 2, three: 3]

The model can be accessed in subsequent controller actions in the chain using the chainModel map. This dynamic property only exists in actions following the call to the chain method:

class ChainController {

def nextInChain() {

def model = chainModel.myModel

…

}

}

Like the redirect method you can also pass parameters to the chain method:

chain(action: "action1", model: [one: 1], params: [myparam: "param1"])

**7.1.5 Controller Interceptors**

Often it is useful to intercept processing based on either request, session or application state. This can be achieved with action interceptors. There are currently two types of interceptors: before and after.

If your interceptor is likely to apply to more than one controller, you are almost certainly better off writing a standalone [Interceptor](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#interceptors). Standaline Interceptors can be applied to multiple controllers or URIs without the need to change the logic of each controller

**Before Interception**

The beforeInterceptor intercepts processing before the action is executed. If it returns false then the intercepted action will not be executed. The interceptor can be defined for all actions in a controller as follows:

def beforeInterceptor = {

println "Tracing action ${actionUri}"

}

The above is declared inside the body of the controller definition. It will be executed before all actions and does not interfere with processing. A common use case is very simplistic authentication:

def beforeInterceptor = [action: **this**.&auth, except: 'login']

// defined with **private** scope, so it's not considered an action

**private** auth() {

**if** (!session.user) {

redirect(action: 'login')

**return** **false**

}

}

def login() {

// display login page

}

The above code defines a method called auth. A private method is used so that it is not exposed as an action to the outside world. The beforeInterceptor then defines an interceptor that is used on all actions *except* the login action and it executes the auth method. The auth method is referenced using Groovy's method pointer syntax. Within the method it detects whether there is a user in the session, and if not it redirects to the login action and returns false, causing the intercepted action to not be processed.

**After Interception**

Use the afterInterceptor property to define an interceptor that is executed after an action:

def afterInterceptor = { model ->

println "Tracing action ${actionUri}"

}

The after interceptor takes the resulting model as an argument and can hence manipulate the model or response.

An after interceptor may also modify the Spring MVC [ModelAndView](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/web/servlet/ModelAndView.html) object prior to rendering. In this case, the above example becomes:

def afterInterceptor = { model, modelAndView ->

println "Current view is ${modelAndView.viewName}"

**if** (model.someVar) modelAndView.viewName = "/mycontroller/someotherview"

println "View is now ${modelAndView.viewName}"

}

This allows the view to be changed based on the model returned by the current action. Note that the modelAndView may be null if the action being intercepted called redirect or render.

**Interception Conditions**

Rails users will be familiar with the authentication example and how the 'except' condition was used when executing the interceptor (interceptors are called 'filters' in Rails; this terminology conflicts with Servlet filter terminology in Java):

def beforeInterceptor = [action: **this**.&auth, except: 'login']

This executes the interceptor for all actions except the specified action. A list of actions can also be defined as follows:

def beforeInterceptor = [action: **this**.&auth, except: ['login', 'register']]

The other supported condition is 'only', this executes the interceptor for only the specified action(s):

def beforeInterceptor = [action: **this**.&auth, only: ['secure']]

**7.1.6 Data Binding**

Data binding is the act of "binding" incoming request parameters onto the properties of an object or an entire graph of objects. Data binding should deal with all necessary type conversion since request parameters, which are typically delivered by a form submission, are always strings whilst the properties of a Groovy or Java object may well not be.

**Map Based Binding**

The data binder is capable of converting and assigning values in a Map to properties of an object. The binder will associate entries in the Map to properties of the object using the keys in the Map that have values which correspond to property names on the object. The following code demonstrates the basics:

// grails-app/domain/Person.groovy

class Person {

String firstName

String lastName

Integer age

}

def bindingMap = [firstName: 'Peter', lastName: 'Gabriel', age: 63]

def person = **new** Person(bindingMap)

assert person.firstName == 'Peter'

assert person.lastName == 'Gabriel'

assert person.age == 63

To update properties of a domain object you may assign a Map to the properties property of the domain class:

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: [county: 'Surrey', country: 'England'] ]

def person = **new** Person(bindingMap)

assert person.firstName == 'Peter'

assert person.lastName == 'Gabriel'

assert person.age == 63

assert person.homeAddress.county == 'Surrey'

assert person.homeAddress.country == 'England'

**Binding To Collections And Maps**

The data binder can populate and update Collections and Maps. The following code shows a simple example of populating a List of objects in a domain 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 Map being bound to a List but since a Set is unordered, the indexes don't necessarily correspond to the order of elements in the Set. In the code example above, if albums were a Set instead of a List, the bindingMap could look exactly the same but 'Foxtrot' might be the first album in the Set or it might be the second. When updating existing elements in a Set the Map being assigned to the Set must have id elements in it which represent the element in the Set being updated, as in the 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 binding to a List or a Set and the index inside of square brackets corresponds to the key in theMap 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 bound to then a new value will be created and added to the Map with the specified key as in the following example:

def bindingMap = [title: 'The Lamb Lies Down On Broadway',

'players[guitar]': [name: 'Steve Hackett'],

'players[vocals]': [name: 'Peter Gabriel']

'players[keyboards]': [name: 'Tony Banks']]

def album = **new** Album(bindingMap)

assert album.title == 'The Lamb Lies Down On Broadway'

assert album.players.size() == 3

assert album.players.guitar == 'Steve Hackett'

assert album.players.vocals == 'Peter Gabriel'

assert album.players.keyboards == 'Tony Banks'

def updatedBindingMap = ['players[drums]': [name: 'Phil Collins'],

'players[keyboards]': [name: 'Anthony George Banks']]

album.properties = updatedBindingMap

assert album.title == 'The Lamb Lies Down On Broadway'

assert album.players.size() == 4

assert album.players.guitar.name == 'Steve Hackett'

assert album.players.vocals.name == 'Peter Gabriel'

assert album.players.keyboards.name == 'Anthony George Banks'

assert album.players.drums.name == 'Phil Collins'

**Binding Request Data to the Model**

The [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object that is available in a controller has special behavior that helps convert dotted request parameter names into nested Maps that the data binder can work with. For example, if a request includes request parameters named person.homeAddress.country and person.homeAddress.city with values 'USA' and 'St. Louis' respectively, params would include entries like these:

[person: [homeAddress: [country: 'USA', city: 'St. Louis']]]

There are two ways to bind request parameters onto the properties of a domain class. The first involves using a domain classes' Map constructor:

def save() {

def b = **new** Book(params)

b.save()

}

The data binding happens within the code new Book(params). By passing the [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object to the domain class constructor Grails automatically recognizes that you are trying to bind from request parameters. So if we had an incoming request like:

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

Then the title and author request parameters would automatically be set on the domain class. You can use the [properties](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\properties.html) property to perform data binding onto an existing instance:

def save() {

def b = Book.get(params.id)

b.properties = params

b.save()

}

This has the same effect as using the implicit constructor.

When binding an empty String (a String with no characters in it, not even spaces), the data binder will convert the empty String to null. This simplifies the most common case where the intent is to treat an empty form field as having the value null since there isn't a way to actually submit a null as a request 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 binding time. To disable this behavior set the grails.databinding.trimStrings property to false in grails-app/conf/Config.groovy.

// 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 binding time. To disable this behavior set thegrails.databinding.convertEmptyStringsToNull property to false in grials-app/conf/Config.groovy.

// the **default** value is **true**

grails.databinding.convertEmptyStringsToNull = **false**

// ...

The order of events is that the String trimming happens and then null conversion happens so if trimStrings is true and convertEmptyStringsToNull is true, not only will empty Strings be converted to null but also blank Strings. A blank String is any String such that the trim() method returns an empty String.

These forms of data binding in Grails are very convenient, but also indiscriminate. In other words, they will bind *all* non-transient, typed instance properties of the target object, including ones that you may not want bound. Just because the form in your UI doesn't submit all the properties, an attacker can still send malign data via a raw HTTP request. Fortunately, Grails also makes it easy to protect against such attacks - see the section titled "Data Binding and Security concerns" for more information.

**Data binding and Single-ended Associations**

If you have a one-to-one or many-to-one association you can use Grails' data binding capability to update these relationships too. For example if you have an incoming request such as:

/book/save?author.id=20

Grails will automatically detect the .id suffix on the request parameter and look up the Author instance for the given id when doing data binding such as:

def b = **new** Book(params)

An association property can be set to null by passing the literal String "null". For example:

/book/save?author.id=**null**

**Data Binding and Many-ended Associations**

If you have a one-to-many or many-to-many association there are different techniques for data binding depending of the association type.

If you have a Set based association (the default for a hasMany) then the simplest way to populate an association is to send a list of identifiers. For example consider the usage of <g:select> below:

<g:select name="books"

from="${Book.list()}"

size="5" multiple="yes" optionKey="id"

value="${author?.books}" />

This produces a select box that lets you select multiple values. In this case if you submit the form Grails will automatically use the identifiers from the select box to populate the books association.

However, if you have a scenario where you want to update the properties of the associated objects the this technique won't work. Instead you use the subscript operator:

<g:textField name="books[0].title" value="the Stand" />

<g:textField name="books[1].title" value="the Shining" />

However, with Set based association it is critical that you render the mark-up in the same order that you plan to do the update in. This is because a Set has no concept of order, so although we're referring to books0 and books1 it is not guaranteed that the order of the association will be correct on the server side unless you apply some explicit sorting yourself.

This is not a problem if you use List based associations, since a List has a defined order and an index you can refer to. This is also true of Map based associations.

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

<g:textField name="books[0].title" value="the Stand" />

<g:textField name="books[1].title" value="the Shining" />

<g:textField name="books[2].title" value="Red Madder" />

Then Grails will automatically create a new instance for you at the defined position.

You can bind existing instances of the associated type to a List using the same .id syntax as you would use with a single-ended association. For example:

<g:select name="books[0].id" from="${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 by the map key:

<g:select name="images[cover].id"

from="${Image.list()}"

value="${book?.images[cover]?.id}"

noSelection="['null': '']"/>

This would bind the selected image into the Map property images under a key of "cover".

When binding to Maps, Arrays and Collections the data binder will automatically grow the size of the collections as necessary. The default limit to how large the binder will grow a collection is 256. If the data binder encounters an entry that requires the collection be grown beyond that limit, the entry is ignored. The limit may be configured by assigning a value to thegrails.databinding.autoGrowCollectionLimit property in Config.groovy.

// grails-app/conf/Config.groovy

// the **default** value is 256

grails.databinding.autoGrowCollectionLimit = 128

// ...

**Data binding with Multiple domain classes**

It is possible to bind data to multiple domain objects from the [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object.

For example so you have an incoming request to:

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

You'll notice the difference with the above request is that each parameter has a prefix such as author. or book. which is used to isolate which parameters belong to which type. Grails' params object is like a multi-dimensional hash and you can index into it to isolate only a subset of the parameters to bind.

def b = **new** Book(params.book)

Notice how we use the prefix before the first dot of the book.title parameter to isolate only parameters below this level to bind. We could do the same with an Author domain class:

def a = **new** Author(params.author)

**Data Binding and Action Arguments**

Controller action arguments are subject to request parameter data binding. There are 2 categories of controller action arguments. The first category is command objects. Complex types are treated as command objects. See the [Command Objects](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#commandObjects) section of the user guide for details. The other category is basic object types. Supported types are the 8 primitives, their corresponding type wrappers and[java.lang.String](http://docs.oracle.com/javase/6/docs/api/java/lang/String.html). The default behavior is to map request parameters to action arguments by name:

class AccountingController {

// accountNumber will be initialized with the value of params.accountNumber

// accountType will be initialized with params.accountType

def displayInvoice(String accountNumber, int accountType) {

// …

}

}

For primitive arguments and arguments which are instances of any of the primitive type wrapper classes a type conversion has to be carried out before the request parameter value can be bound to the action argument. The type conversion happens automatically. In a case like the example shown above, the params.accountType request parameter has to be converted to an int. If type conversion fails for any reason, the argument will have its default value per normal Java behavior (null for type wrapper references, false for booleans and zero for numbers) and a corresponding error will be added to the 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 controller's errors.hasErrors() will be true, the controller's errors.errorCount will be equal to 1 and the controller's errors.getFieldError('accountType') will contain the corresponding error.

If the argument name does not match the name of the request parameter then the @grails.web.RequestParameter annotation may be applied to 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, int accountType) {

// …

}

}

**Data binding and type conversion errors**

Sometimes when performing data binding it is not possible to convert a particular String into a particular target type. This results in a type conversion error. Grails will retain type conversion errors inside the [errors](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Domain%20Classes\errors.html) property of a Grails domain class. For example:

class Book {

…

URL publisherURL

}

Here we have a domain class Book that uses the java.net.URL class to represent URLs. Given an incoming request such as:

/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 error occurs. You can check for these like this:

def b = **new** Book(params)

**if** (b.hasErrors()) {

println "The value ${b.errors.getFieldError('publisherURL').rejectedValue}" +

" is not a valid URL!"

}

Although we have not yet covered error codes (for more information see the section on [Validation](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#validation)), for type conversion errors you would want a message from the grails-app/i18n/messages.properties file to use for the error. You can use a generic error message handler such as:

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

Or a more specific one:

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

**The BindUsing Annotation**

The [BindUsing](http://grails.org/doc/3.0.x/api/org/grails/databinding/BindUsing.html) annotation may be used to define a custom binding mechanism for a particular field in a class. Any time data binding is being applied to the field the closure value of the annotation will be invoked with 2 arguments. The first argument is the object that data binding is being applied to and the second argument is [DataBindingSource](http://grails.org/doc/3.0.x/api/org/grails/databinding/DataBindingSource.html) which is the data source for the data binding. The value returned from the closure will be bound to the property. The following example would result in the upper case version of the name value in the source being applied to the name field during data binding.

**import** org.grails.databinding.BindUsing

class SomeClass {

@BindUsing({obj, source ->

//source is DataSourceBinding which is similar to a Map

//and defines getAt operation but source.name cannot be used here.

//In order to get name from source use getAt instead as shown below.

source['name']?.toUpperCase()

})

String name

}

Note that data binding is only possible when the name of the request parameter matches with the field name in the class. Here, name from request parameters matches with name from SomeClass.

The [BindUsing](http://grails.org/doc/3.0.x/api/org/grails/databinding/BindUsing.html) annotation may be used to define a custom binding mechanism for all of the fields on a particular class. When the annotation is applied to a class, the value assigned to the annotation should be a class which implements the [BindingHelper](http://grails.org/doc/3.0.x/api/org/grails/databinding/BindingHelper.html) interface. An instance of that class will be used any 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 own mechanism for converting values and a simple way to do this is to write a class which implements[ValueConverter](http://grails.org/doc/3.0.x/api/org/grails/databinding/converters/ValueConverter.html) and register an instance of that class as a bean in the Spring application context.

**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 name is not important. All beans that implemented 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 [BindingFormat](http://grails.org/doc/3.0.x/api/org/grails/databinding/BindingFormat.html) annotation to a Date field.

**import** org.grails.databinding.BindingFormat

class Person {

@BindingFormat('MMddyyyy')

Date birthDate

}

A global setting may be configured in Config.groovy to define date formats which will be used application wide when binding to Date.

// grails-app/conf/Config.groovy

grails.databinding.dateFormats = ['MMddyyyy', 'yyyy-MM-dd HH:mm:ss.S', "yyyy-MM-dd'T'hh:mm:ss'Z'"]

The formats specified in grails.databinding.dateFormats will be attempted in the order in which they are included in the List. If a property is marked with @BindingFormat, the @BindingFormat will take precedence over the values specified in grails.databinding.dateFormats.

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

**Custom Formatted Converters**

You may supply your own handler for the [BindingFormat](http://grails.org/doc/3.0.x/api/org/grails/databinding/BindingFormat.html) annotation by writing a class which implements the [FormattedValueConverter](http://grails.org/doc/3.0.x/api/org/grails/databinding/converters/FormattedValueConverter.html) interface and registering an instance of that class as a bean in the Spring application context. Below is an example of a trivial custom String formatter that might convert 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 name is not important. All beans that implemented 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 binder to take advantage of the custom converter.

**import** org.grails.databinding.BindingFormat

class Person {

@BindingFormat('UPPERCASE')

String someUpperCaseString

@BindingFormat('LOWERCASE')

String someLowerCaseString

String someOtherString

}

**Localized Binding Formats**

The BindingFormat annotation supports localized format strings by using the optional code attribute. If a value is assigned to the code attribute that value will be used as the message code to retrieve the binding format string from the messageSource bean in the Spring application context and that lookup will be localized.

**import** org.grails.databinding.BindingFormat

class Person {

@BindingFormat(code='date.formats.birthdays')

Date birthDate

}

# grails-app/conf/i18n/messages.properties

date.formats.birthdays=MMddyyyy

# grails-app/conf/i18n/messages\_es.properties

date.formats.birthdays=ddMMyyyy

**Structured Data Binding Editors**

A structured data binding editor is a helper class which can bind structured request parameters to a property. The common use case for structured binding is binding to a Date object which might be constructed from several smaller pieces of information contained in several request parameters with names like birthday\_month, birthday\_date and birthday\_year. The structured editor would retrieve all of those individual pieces of information and use them to construct a Date.

The framework provides a structured editor for binding to Date objects. An application may register its own structured editors for whatever types are 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 to accept request parameters like width and height and use those to calculate the area of a Shape at binding time. A structured binding editor is well suited for that.

The way to register a structured editor with the data binding process is to add an instance of the [org.grails.databinding.TypedStructuredBindingEditor](http://grails.org/doc/3.0.x/api/org/grails/databinding/TypedStructuredBindingEditor.html) interface to the Spring application context. The easiest way to implement the TypedStructuredBindingEditor interface is to extend the [org.grails.databinding.converters.AbstractStructuredBindingEditor](http://grails.org/doc/3.0.x/api/org/grails/databinding/converters/AbstractStructuredBindingEditor.html) abstract class and override thegetPropertyValue method as shown below:

// src/groovy/databinding/converters/StructuredShapeEditor.groovy

**package** databinding.converters

**import** databinding.Shape

**import** org.grails.databinding.converters.AbstractStructuredBindingEditor

class StructuredShapeEditor **extends** AbstractStructuredBindingEditor<Shape> {

**public** Shape getPropertyValue(Map values) {

// retrieve the individual values from the Map

def width = values.width as int

def height = values.height as int

// use the values to calculate the area of the Shape

def area = width \* height

// create and **return** a Shape with the appropriate area

**new** Shape(area: area)

}

}

An instance of that class needs to be registered with the Spring application context:

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

beans = {

shapeEditor databinding.converters.StructuredShapeEditor

// …

}

When the data binder binds to an instance of the Gadget class it will check to see if there are request parameters with names compressedShape and expandedShape which have a value of "struct" and if they do exist, that will trigger the use of the StructuredShapeEditor. The individual components of the structure need to have parameter names of the form propertyName\_structuredElementName. In the case of the Gadget class above that would mean that the compressedShape request parameter should have a value of "struct" and the compressedShape\_width andcompressedShape\_height parameters should have values which represent the width and the height of the compressed Shape. Similarly, the expandedShape request parameter should have a value of "struct" and the expandedShape\_width and expandedShape\_height parameters 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\_height=30

&compressedShape=struct&compressedShape\_width=10&compressedShape\_height=3

\*/

// with the request parameters shown above gadget.expandedShape.area would be 2400

// 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](http://grails.org/doc/3.0.x/api/org/grails/databinding/events/DataBindingListener.html) interface provides a mechanism for listeners to be notified of data binding events. The interface looks like this:

**package** org.grails.databinding.events;

**import** org.grails.databinding.errors.BindingError;

**public** **interface** DataBindingListener {

/\*\*

\* @**return** **true** **if** the listener is interested in events **for** the specified type.

\*/

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.BindingResult)

\* @**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.BindingResult)

\* @**return** **true** **if** data binding should **continue**, otherwise **return** **false**

\*/

Boolean beforeBinding(Object target, String propertyName, Object value, Object errors);

/\*\*

\* 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.BindingResult)

\*/

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.BindingResult)

\*/

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.BindingResult)

\* @see BindingError

\*/

void bindingError(BindingError error, Object errors);

}

Any bean in the Spring application context which implements that interface will automatically be registered with the data binder. The [DataBindingListenerAdapter](http://grails.org/doc/3.0.x/api/org/grails/databinding/events/DataBindingListenerAdapter.html) class implements theDataBindingListener interface and provides default implementations for all of the methods in the interface so this class is well suited for subclassing so your listener class only needs to provide implementations for the methods your listener is interested in.

The Grails data binder has limited support for the older [BindEventListener](http://grails.org/doc/3.0.x/api/org/codehaus/groovy/grails/web/binding/BindEventListener.html) style listeners. BindEventListener looks like this:

**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 typeConverter);

}

Support for BindEventListener is disabled by default. To enable support assign a value of true to the grails.databinding.enableSpringEventAdapter property in grails-app/conf/Config.groovy.

// grails-app/conf/Config.groovy

grails.databinding.enableSpringEventAdapter=**true**

...

With enableSpringEventAdapter set to true instances of BindEventListener which are in the Spring application context will automatically be registered with the data binder. Notice that theMutablePropertyValues and TypeConverter arguments to the doBind method in BindEventListener are Spring specific classes and are not relevant to the current data binder. The event adapter will pass null values for those arguments. The only real value passed into the doBind method will be the object being bound to. This limited support is provided for backward compatibility and will be useful for a subset of scenarios. Developers are encouraged to migrate their BindEventListener beans to the newer 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 binding in a Service on some arbitrary object which 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 grailsWebDataBinder. That bean implements the [DataBinder](http://grails.org/doc/3.0.x/api/org/grails/databinding/DataBinder.html) 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](http://grails.org/doc/3.0.x/api/org/grails/databinding/DataBinder.html) 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 bind malicious data to domain classes and be persisted in the database. You can limit what properties are bound to a given domain class using the subscript operator:

def p = Person.get(1)

p.properties['firstName','lastName'] = params

In this case only the firstName and lastName properties will be bound.

Another way to do this is is to use [Command Objects](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#commandObjects) as the target of data binding instead of domain classes. Alternatively there is also the flexible [bindData](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\bindData.html) method.

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

def p = **new** Person()

bindData(p, params)

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

def p = **new** Person()

bindData(p, params, [exclude: 'dateOfBirth'])

Or include only certain properties:

def p = **new** Person()

bindData(p, params, [include: ['firstName', 'lastName']])

Note that if an empty List is provided as a value for the include parameter then all fields will be subject to binding if they are not explicitly excluded.

**7.1.7 XML and JSON Responses**

**Using the render method to output XML**

Grails supports a few different ways to produce XML and JSON responses. The first is the [render](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\render.html) 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 an error:

def list() {

def books = Book.list() // naming conflict here

render(contentType: "text/xml") {

books {

**for** (b in results) {

book(title: b.title)

}

}

}

}

This is because there is local variable books which Groovy attempts to invoke as a method.

**Using the render method to output JSON**

The render method can also be used to output JSON:

def list() {

def results = Book.list()

render(contentType: "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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#GORM) 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#REST)

**Automatic JSON Marshalling**

Grails also supports automatic marshalling to JSON using the same mechanism. Simply substitute XML with JSON:

render Book.list() as JSON

The resulting output would look something like the following:

[

{"id":1,

"class":"Book",

"author":"Stephen King",

"title":"The Stand"},

{"id":2,

"class":"Book",

"author":"Stephen King",

"releaseDate":**new** Date(1194127343161),

"title":"The Shining"}

]

**7.1.8 More on JSONBuilder**

The previous section on on XML and JSON responses covered simplistic examples of rendering XML and JSON responses. Whilst the XML builder used by Grails is the standard [XmlSlurper](http://groovy.codehaus.org/Reading+XML+using+Groovy's+XmlSlurper) found in Groovy, the JSON builder is a custom implementation specific to Grails.

**JSONBuilder and Grails versions**

JSONBuilder behaves different depending on the version of Grails you use. For version below 1.2 the deprecated [grails.web.JSONBuilder](http://grails.org/doc/3.0.x/api/grails/web/JSONBuilder.html) class is used. This section covers the usage of the Grails 1.2 JSONBuilder

For backwards compatibility the old JSONBuilder class is used with the render method for older applications; to use the newer/better JSONBuilder class set the following in Config.groovy:

grails.json.legacy.builder = **false**

**Rendering Simple Objects**

To render a simple JSON object just set properties within the context of the Closure:

render(contentType: "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 directly:

def builder = **new** JSONBuilder()

def result = builder.build {

categories = ['a', 'b', 'c']

title = "Hello JSON"

information = {

pages = 10

}

}

// prints the JSON text

println result.toString()

def sw = **new** StringWriter()

result.render sw

**7.1.9 Uploading Files**

**Programmatic File Uploads**

Grails supports file uploads using Spring's [MultipartHttpServletRequest](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/web/multipart/MultipartHttpServletRequest.html) interface. The first step for file uploading is to create a multipart form like this:

Upload Form: <br />

<g:uploadForm action="upload">

<input type="file" name="myFile" />

<input type="submit" />

</g:uploadForm>

The uploadForm tag conveniently adds the enctype="multipart/form-data" attribute to the standard <g:form> tag.

There are then a number of ways to handle the file upload. One is to work with the Spring [MultipartFile](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/web/multipart/MultipartFile.html) instance directly:

def upload() {

def f = request.getFile('myFile')

**if** (f.empty) {

flash.message = 'file cannot be empty'

render(view: 'uploadForm')

**return**

}

f.transferTo(**new** File('/some/local/dir/myfile.txt'))

response.sendError(200, 'Done')

}

This is convenient for doing transfers to other destinations and manipulating the file directly as you can obtain an InputStream and so on with the [MultipartFile](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/web/multipart/MultipartFile.html) 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 automatically bind the file's contents as a byte to the myFile property:

def img = **new** Image(params)

It's important that you set the [size](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\size.html) or [maxSize](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Constraints\maxSize.html) constraints, otherwise your database may be created with a small column size that can't handle reasonably 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 the image to a String type:

class Image {

String myFile

}

**7.1.10 Command Objects**

Grails controllers support the concept of command objects. A command object is a class that is used in conjunction with [data binding](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#dataBinding), usually to allow validation of data that may not fit into an existing domain class.

Note: A class is only considered to be a command object when it is used as a parameter of an action.

**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 trait allows the definition of [constraints](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#constraints) just like in [domain classes](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#GORM). If the command object is defined in the same source file as the controller that is using it, Grails will automatically make it Validateable. It is not required that command object classes be validateable.

By default, all Validateable object properties are nullable: false which matches the behavior of GORM domain objects. If you want a Validateable that has nullable: true properties by default, you can specify this by defining a defaultNullable method in the class:

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 parameters. The parameter types must be supplied so that Grails knows what objects to create and initialize.

Before the controller action is executed Grails will automatically create an instance of the command object class and populate its properties by binding the request parameters. If the command object class is marked with Validateable then the command object will be validated. For example:

class LoginController {

def login(LoginCommand cmd) {

**if** (cmd.hasErrors()) {

redirect(action: 'loginForm')

**return**

}

// work with the command object data

}

}

If the command object's type is that of a domain class and there is an id request parameter then instead of invoking the domain class constructor to create a new instance a call will be made to the staticget method on the domain class and the value of the id parameter will be passed as an argument. Whatever is returned from that call to get is what will be passed into the controller action. This means that if there is an id request parameter and no corresponding record is found in the database then the value of the command object will be null. If an error occurs retrieving the instance from the database then null will be passed as an argument to the controller action and an error will be added the controller's errors property. If the command object's type is a domain class and there is no idrequest parameter or there is an id request parameter and its value is empty then null will be passed into the controller action unless the HTTP request method is "POST", in which case a new instance of the domain class will be created by invoking the domain class constructor. For all of the cases where the domain class instance is non-null, data binding is only performed if the HTTP request method 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. Nested parameter names may be used to bind down the object graph in an intuitive way. In the example below a request parameter named name will be bound to the name property of the Person instance and a request parameter named address.city will be bound to the city property of theaddress property in the Person.

class StoreController {

def buy(Person buyer) {

// …

}

}

class Person {

String name

Address address

}

class Address {

String city

}

A problem may arise if a controller action accepts multiple command objects which happen to contain the same property name. Consider the following example.

class StoreController {

def buy(Person buyer, Product product) {

// …

}

}

class Person {

String name

Address address

}

class Address {

String city

}

class Product {

String name

}

If there is a request parameter named name it isn't clear if that should represent the name of the Product or the name of the Person. Another version of the problem can come up if a controller action accepts 2 command objects of the same type as shown below.

class StoreController {

def buy(Person buyer, Person seller, Product product) {

// …

}

}

class Person {

String name

Address address

}

class Address {

String city

}

class Product {

String name

}

To help deal with this the framework imposes special rules for mapping parameter names to command object types. The command object data binding will treat all parameters that begin with the controller action parameter name as belonging to the corresponding command object. For example, the product.name request parameter will be bound to the name property in the product argument, the buyer.name request parameter will be bound to the name property in the buyer argument the seller.address.city request parameter will be bound to the city property of the address 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 some custom validation logic which uses a Grails [service](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#services):

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 from the Spring ApplicationContext.

**Binding The Request Body To Command Objects**

When a request is made to a controller action which accepts a command object and the request contains a body, Grails will attempt to parse the body of the request based on the request content type and use the body to do data binding on the command object. See the following example.

// grails-app/controllers/bindingdemo/DemoController.groovy

**package** bindingdemo

class DemoController {

def createWidget(Widget w) {

render "Name: ${w?.name}, Size: ${w?.size}"

}

}

class Widget {

String name

Integer size

}

$ curl -H "Content-Type: application/json" -d '{"name":"Some Widget","size":"42"}' localhost:8080/myapp/demo/createWidget

Name: Some Widget, Size: 42

~ $

$ curl -H "Content-Type: application/xml" -d '<widget><name>Some Other Widget</name><size>2112</size></widget>' localhost:8080/bodybind/demo/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 request after that will fail since the corresponding input stream will be empty. The controller action can either use a command object or it can parse the body of the request on its own (either directly, or by 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

// ...

}

}

**7.1.11 Handling Duplicate Form Submissions**

Grails has built-in support for handling duplicate form submissions using the "Synchronizer Token Pattern". To get started you define a token on the [form](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\form.html) tag:

<g:form useToken="true" ...>

Then in your controller code you can use the [withForm](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\withForm.html) method to handle valid and invalid requests:

withForm {

// good request

}.invalidToken {

// bad request

}

If you only provide the [withForm](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\withForm.html) method and not the chained invalidToken method then by default Grails will store the invalid token in a flash.invalidToken variable and redirect the request back to the original page. This can then be checked in the view:

<g:if test="${flash.invalidToken}">

Don't click the button twice!

</g:if>

The [withForm](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\withForm.html) tag makes use of the [session](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\session.html) and hence requires session affinity or clustered sessions if used in a cluster.

**7.1.12 Simple Type Converters**

**Type Conversion Methods**

If you prefer to avoid the overhead of [Data Binding](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#dataBinding) and simply want to convert incoming parameters (typically Strings) into another more appropriate type the [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) 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, short and so on. Each of these methods is null-safe 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 default value will be returned if a corresponding 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 example you could get a query string such as ?name=Bob&name=Judy.

In this case dealing with one parameter and dealing with many has different semantics since Groovy's iteration mechanics for String iterate over each character. To avoid this problem the [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html)object provides a list method that always returns a list:

**for** (name in params.list('name')) {

println name

}

**7.1.13 Declarative Controller Exception Handling**

Grails controllers support a simple mechanism for declarative exception handling. If a controller declares a method that accepts a single argument and the argument type is java.lang.Exception or some subclass of java.lang.Exception, that method will be invoked any time an action in that 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 method an exception handler, the Exception argument type is the important part.

The exception handler methods can do anything that a controller action can do including invoking render, redirect, returning a model, etc.

One way to share exception handler methods across multiple controllers is to use inheritance. Exception handler methods are inherited into subclasses so an application could define the exception handlers in an abstract class that multiple controllers extend from. Another way to share exception handler 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 methods which are runtime metaprogrammed onto a controller class are not supported.

**7.2 Groovy Server Pages**

Groovy Servers Pages (or GSP for short) is Grails' view technology. It is designed to be familiar for users of technologies such as ASP and JSP, but to be far more flexible and intuitive.

GSPs live in the grails-app/views directory and are typically rendered automatically (by convention) or with the [render](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\render.html) method such as:

render(view: "index")

A GSP is typically a mix of mark-up and GSP tags which aid in view rendering.

Although it is possible to have Groovy logic embedded in your GSP and doing this will be covered in this document, the practice is strongly discouraged. Mixing mark-up and code is a **bad** thing and most GSP pages contain no code and needn't do so.

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

def show() {

[book: Book.get(params.id)]

}

This action will look up a Book instance and create a model that contains a key called book. This key can then be referenced within the GSP view using the name book:

${book.title}

Embedding data received from user input has the risk of making your application vulnerable to an Cross Site Scripting (XSS) attack. Please read the documentation on [XSS prevention](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#xssPrevention) for information on how to prevent XSS attacks.

**7.2.1 GSP Basics**

In the next view sections we'll go through the basics of GSP and what is available to you. First off let's cover some basic syntax that users of JSP and ASP should be familiar with.

GSP supports the usage of <% %> 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 following example demonstrates:

<html>

<body>

<%-- This is my comment --%>

<%="Hello GSP!" %>

</body>

</html>

Embedding data received from user input has the risk of making your application vulnerable to an Cross Site Scripting (XSS) attack. Please read the documentation on [XSS prevention](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#xssPrevention) for information on how to prevent XSS attacks.

**7.2.1.1 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](http://download.oracle.com/javaee/1.4/api/javax/servlet/ServletContext.html) instance
* applicationContext The Spring [ApplicationContext](http://docs.spring.io/spring/docs/4.0.x/javadoc-api/org/springframework/context/ApplicationContext.html) instance
* flash - The [flash](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\flash.html) object
* grailsApplication - The [GrailsApplication](http://grails.org/doc/3.0.x/api/grails/core/GrailsApplication.html) instance
* out - The response writer for writing to the output stream
* params - The [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object for retrieving request parameters
* request - The [HttpServletRequest](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpServletRequest.html) instance
* response - The [HttpServletResponse](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpServletResponse.html) instance
* session - The [HttpSession](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpSession.html) instance
* webRequest - The [GrailsWebRequest](http://grails.org/doc/3.0.x/api/org/grails/web/servlet/mvc/GrailsWebRequest.html) instance

**7.2.1.2 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>

**7.2.1.3 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 default imports and [GSP Tags](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#tags):

<%@ page import="java.awt.\*" %>

GSP also supports the contentType directive:

<%@ page contentType="application/json" %>

The contentType directive allows using GSP to render other formats.

**7.2.1.4 Expressions**

In GSP the <%= %> syntax introduced earlier is rarely used due to the support for GSP expressions. A GSP expression is similar to a JSP EL expression or a Groovy GString and takes the form ${expr}:

<html>

<body>

Hello ${params.name}

</body>

</html>

However, unlike JSP EL you can have any Groovy expression within the ${..} block.

Embedding data received from user input has the risk of making your application vulnerable to an Cross Site Scripting (XSS) attack. Please read the documentation on [XSS prevention](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#xssPrevention) for information on how to prevent XSS attacks.

**7.2.2 GSP Tags**

Now that the less attractive JSP heritage has been set aside, the following sections cover GSP's built-in tags, which are the preferred way to define GSP pages.

The section on [Tag Libraries](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#taglibs) 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 tag starts with g: it is automatically assumed to be a GSP tag. An example GSP tag would look like:

<g:example />

GSP tags can also have a body such as:

<g:example>

Hello world

</g:example>

Expressions can be passed into GSP tag attributes, if an expression is not used it will be assumed to be a String value:

<g:example attr="${new Date()}">

Hello world

</g:example>

Maps can also be passed into GSP tag attributes, which are often used for a named parameter style syntax:

<g:example attr="${new Date()}" attr2="[one:1, two:2, three:3]">

Hello world

</g:example>

Note that within the values of attributes you must use single quotes for Strings:

<g:example attr="${new Date()}" attr2="[one:'one', two:'two']">

Hello world

</g:example>

With the basic syntax out the way, the next sections look at the tags that are built into Grails by default.

**7.2.2.1 Variables and Scopes**

Variables can be defined within a GSP using the [set](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\set.html) tag:

<g:set var="now" value="${new Date()}" />

Here we assign a variable called now to the result of a GSP expression (which simply constructs a new java.util.Date instance). You can also use the body of the <g:set> tag to define a variable:

<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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\flash.html) 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" />

**7.2.2.2 Logic and Iteration**

GSP also supports logical and iterative tags out of the box. For logic there are [if](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\if.html), [else](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\else.html) and [elseif](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\elseif.html) tags for use with branching:

<g:if test="${session.role == 'admin'}">

<%-- show administrative functions --%>

</g:if>

<g:else>

<%-- show basic functions --%>

</g:else>

Use the [each](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\each.html) and [while](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\while.html) 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>

**7.2.2.3 Search and Filtering**

If you have collections of objects you often need to sort and filter them. Use the [findAll](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\findAll.html) and [grep](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\grep.html) tags for these tasks:

Stephen King's Books:

<g:findAll in="${books}" expr="it.author == 'Stephen King'">

<p>Title: ${it.title}</p>

</g:findAll>

The expr attribute contains a Groovy expression that can be used as a filter. The [grep](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\grep.html) tag does a similar job, for example filtering by class:

<g:grep in="${books}" filter="NonFictionBooks.class">

<p>Title: ${it.title}</p>

</g:grep>

Or using a regular expression:

<g:grep in="${books.title}" filter="~/.\*?Groovy.\*?/">

<p>Title: ${it}</p>

</g:grep>

The above example is also interesting due to its usage of GPath. GPath is an XPath-like language in Groovy. The books variable is a collection of Book instances. Since each Book has a title, you can obtain a list of Book titles using the expression books.title. Groovy will auto-magically iterate the collection, obtain each title, and return a new list!

**7.2.2.4 Links and Resources**

GSP also features tags to help you manage linking to controllers and actions. The [link](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\link.html) tag lets you specify controller and action name pairing and it will automatically work out the link based on the [URL Mappings](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#urlmappings), 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>

**7.2.2.5 Forms and Fields**

**Form Basics**

GSP supports many different tags for working with HTML forms and fields, the most basic of which is the [form](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\form.html) tag. This is a controller/action aware version of the regular HTML form tag. The url attribute lets you specify which controller and action to map to:

<g:form name="myForm" url="[controller:'book',action:'list']">...</g:form>

In this case we create a form called myForm that submits to the BookController's list action. Beyond that all of the usual HTML attributes apply.

**Form Fields**

In addition to easy construction of forms, GSP supports custom tags for dealing with different types of fields, including:

* [textField](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\textField.html) - For input fields of type 'text'
* [passwordField](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\passwordField.html) - For input fields of type 'password'
* [checkBox](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\checkBox.html) - For input fields of type 'checkbox'
* [radio](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\radio.html) - For input fields of type 'radio'
* [hiddenField](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\hiddenField.html) - For input fields of type 'hidden'
* [select](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\select.html) - 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\radioGroup.html) (for creating groups of [radio](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\radio.html) tags), [localeSelect](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\localeSelect.html), [currencySelect](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\currencySelect.html) and [timeZoneSelect](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\timeZoneSelect.html) (for selecting locales, currencies and time zones respectively).

**Multiple Submit Buttons**

The age old problem of dealing with multiple submit buttons is also handled elegantly with Grails using the [actionSubmit](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\actionSubmit.html) tag. It is just like a regular submit, but lets you specify an alternative action to submit to:

<g:actionSubmit value="Some update label" action="update" />

**7.2.2.6 Tags as Method Calls**

One major different between GSP tags and other tagging technologies is that GSP tags can be called as either regular tags or as method calls from [controllers](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#controllers), [tag libraries](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#taglibs) 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 methods as String) instead of writing directly to the response when called as methods. For example:

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

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

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

In view technologies that don't support this feature you have to nest tags within tags, which becomes messy quickly and often has an adverse effect of 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: [namespace](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#namespaces) can be invoked without the prefix and a StreamCharBuffer result is returned:

def imageLocation = createLinkTo(dir:"images", file:"logo.jpg").toString()

Prefix the namespace to avoid naming conflicts:

def imageLocation = g.createLinkTo(dir:"images", file:"logo.jpg").toString()

For tags that use a [custom namespace](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#namespaces), use that prefix for the method call. For example (from the [FCK Editor plugin](http://grails.org/plugin/fckeditor)):

def editor = fckeditor.editor(name: "text", width: "100%", height: "400")

**7.2.3 Views and Templates**

Grails also has the concept of templates. These are useful for partitioning your views into maintainable chunks, and combined with [Layouts](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#layouts) provide a highly re-usable mechanism for structured views.

**Template Basics**

Grails uses the convention of placing an underscore before the name of a view to identify it as a template. For example, you might have a template that renders Books located at grails-app/views/book/\_bookTemplate.gsp:

<div class="book" id="${book?.id}">

<div>Title: ${book?.title}</div>

<div>Author: ${book?.author?.name}</div>

</div>

Use the [render](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\render.html) 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 multiple Book instances you can also render the template for each Book using the render tag with acollection 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 grails-app/views/book. However, you may want to share templates across your application.

In this case you can place them in the root views directory at grails-app/views or any subdirectory below that location, and then with the template attribute use an absolute location starting with / instead of a relative location. For example if you had a template called grails-app/views/shared/\_mySharedTemplate.gsp, you would reference it as:

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

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

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

**The Template Namespace**

Since templates are used so frequently there is template namespace, called tmpl, available that makes using templates easier. Consider for example the following usage pattern:

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

This can be expressed with the tmpl namespace as follows:

<tmpl:bookTemplate book="${myBook}" />

**Templates in Controllers and Tag Libraries**

You can also render templates from controllers using the [render](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\render.html) controller method. This is useful for JavaScript heavy applications where you generate 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\render.html) controller method writes directly to the response, which is the most common behaviour. To instead obtain the result of template as a String you can use the [render](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\render.html) 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#tagsAsMethodCalls) instead of the [render](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\render.html) method.

**7.2.4 Layouts with Sitemesh**

**Creating Layouts**

Grails leverages [Sitemesh](http://www.opensymphony.com/sitemesh/), a decorator engine, to support view layouts. Layouts are located in the grails-app/views/layouts directory. A typical layout can be seen below:

<html>

<head>

<title><g:layoutTitle default="An example decorator" /></title>

<g:layoutHead />

</head>

<body onload="${pageProperty(name:'body.onload')}">

<div class="menu"><!--my common menu goes here--></menu>

<div class="body">

<g:layoutBody />

</div>

</div>

</body>

</html>

The key elements are the [layoutHead](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\layoutHead.html), [layoutTitle](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\layoutTitle.html) and [layoutBody](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\layoutBody.html) 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\pageProperty.html) tag which can be used to inspect and return aspects of the target page.

**Triggering Layouts**

There are a few ways to trigger a layout. The simplest is to add a meta tag to the view:

<html>

<head>

<title>An Example Page</title>

<meta name="layout" content="main" />

</head>

<body>This is my content!</body>

</html>

In this case a layout called grails-app/views/layouts/main.gsp will be used to layout the page. If we were to use the layout from the previous section the output would resemble 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" property in a controller. For example, if you have a controller such as:

class BookController {

**static** layout = 'customer'

def list() { … }

}

You can create a layout called grails-app/views/layouts/customer.gsp which will be applied to all views that the BookController delegates to. The value of the "layout" property may contain a directory structure relative to the grails-app/views/layouts/ directory. For example:

class BookController {

**static** layout = 'custom/customer'

def list() { … }

}

Views rendered from that controller would be decorated with the grails-app/views/layouts/custom/customer.gsp template.

**Layout by Convention**

Another way to associate layouts is to use "layout by convention". For example, if you have this controller:

class BookController {

def list() { … }

}

You can create a layout called grails-app/views/layouts/book.gsp, which will be applied to all views that the BookController delegates to.

Alternatively, you can create a layout called grails-app/views/layouts/book/list.gsp which will only be applied to the list action within the BookController.

If you have both the above mentioned layouts in place the layout specific to the action will take precedence when the list action is executed.

If a layout may not be located using any of those conventions, the convention of last resort is to look for the application default layout which is grails-app/views/layouts/application.gsp. The name of the application default layout may be changed by defining a property in grails-app/conf/Config.groovy as follows:

grails.sitemesh.**default**.layout = 'myLayoutName'

With that property in place, the application default layout will be grails-app/views/layouts/myLayoutName.gsp.

**Inline Layouts**

Grails' also supports Sitemesh's concept of inline layouts with the [applyLayout](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\applyLayout.html) tag. This can be used to apply a layout to a template, URL or arbitrary section of content. This lets you even further modularize your view structure by "decorating" your template includes.

Some examples of usage can be seen below:

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

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

<g:applyLayout name="myLayout">

The content to apply a layout to

</g:applyLayout>

**Server-Side Includes**

While the [applyLayout](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\applyLayout.html) tag is useful for applying layouts to external content, if you simply want to include external content in the current page you use the [include](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\include.html) tag:

<g:include controller="book" action="list" />

You can even combine the [include](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\include.html) tag and the [applyLayout](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\applyLayout.html) tag for added flexibility:

<g:applyLayout name="myLayout">

<g:include controller="book" action="list" />

</g:applyLayout>

Finally, you can also call the [include](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\include.html) 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\include.html) tag.

**7.2.5 Static Resources**

Grails 2.0 integrates with the [Asset Pipeline plugin](http://grails.org/plugin/asset-pipeline) to provide sophisticated static asset management. This plugin is installed by default in new Grails applications.

The basic way to include a link to a static asset in your application is to use the [resource](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\resource.html) tag. This simple approach creates a URI pointing to the file.

However modern applications with dependencies on multiple JavaScript and CSS libraries and frameworks (as well as dependencies on multiple Grails plugins) require something more powerful.

The issues that the Asset-Pipeline plugin tackles are:

* Reduced Dependence - The plugin has compression, minification, and cache-digests built in.
* Easy Debugging - Makes for easy debugging by keeping files separate in development mode.
* Asset Bundling using require [directives](http://bertramdev.github.io/asset-pipeline/guide/usage.html#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 they get compiled on War creation.

Take a look at the [documentation](http://bertramdev.github.io/asset-pipeline) for the asset-pipeline to get started.

**7.2.6 Sitemesh Content Blocks**

Although it is useful to decorate an entire page sometimes you may find the need to decorate independent sections of your site. To do this you can use content blocks. To get started, 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>

**7.2.7 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 change to the views requires that you redeploy your whole application. If all you want to do is fix a typo on a page, or change an image link, it can seem like a lot of unnecessary work. For such simple 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 we want to keep them unpacked in a /var/www/grails/my-app directory. We add these two lines to grails-app/conf/Config.groovy :

grails.gsp.enable.reload = **true**

grails.gsp.view.dir = "/**var**/www/grails/my-app/"

The first line tells Grails that modified GSP files should be reloaded at runtime. If you don't have this setting, you can make as many changes as you like but they won't be reflected in the running application until you restart. The second line tells Grails where to load the views and layouts from.

The trailing slash on the grails.gsp.view.dir value is important! Without it, Grails will look for views in the parent directory.

Setting "grails.gsp.view.dir" is optional. If it's not specified, you can update files directly to the application server's deployed war directory. Depending on the application server, these files might get overwritten when the server is restarted. Most application servers support "exploded war deployment" which is recommended in this case.

With those settings in place, all you need to do is copy the views from your web application to the external directory. On a Unix-like system, this would look something like this:

mkdir -p /**var**/www/grails/my-app/grails-app/views

cp -R grails-app/views/\* /**var**/www/grails/my-app/grails-app/views

The key point here is that you must retain the view directory structure, including the grails-app/views bit. So you end up with the path /var/www/grails/my-app/grails-app/views/... .

One thing to bear in mind with this technique is that every time you modify a GSP, it uses up permgen space. So at some point you will eventually hit "out of permgen space" errors unless you restart the server. So this technique is not recommended for frequent or large changes to the views.

There are also some System properties to control GSP reloading:

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Default** |
| grails.gsp.enable.reload | alternative system property for enabling the GSP reload mode without changing Config.groovy |  |
| grails.gsp.reload.interval | interval between checking the lastmodified time of the gsp source file, unit is milliseconds | 5000 |
| grails.gsp.reload.granularity | the number of milliseconds leeway to give before deciding a file is out of date. this is needed because different roundings usually cause a 1000ms difference in lastmodified times | 1000 |

GSP reloading is supported for precompiled GSPs since Grails 1.3.5 .

**7.2.8 GSP Debugging**

**Viewing the generated source code**

* Adding "?showSource=true" or "&showSource=true" to the url shows the generated Groovy source code for the view instead of rendering it. It won't 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.keepgenerateddir" (in Config.groovy) . It must point to a directory that exists and is writable.
* During "grails war" gsp pre-compilation, the generated source code is stored in grails.project.work.dir/gspcompile (usually in ~/.grails/(grails\_version)/projects/(project name)/gspcompile).

**Debugging GSP code with a debugger**

* See [Debugging GSP in STS](http://contraptionsforprogramming.blogspot.com/2010/08/debuggable-gsps-in-springsource-tool.html)

**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 templates can be used to render a single page. It might be hard to find out what GSP template actually renders the html seen in the result. The debug templates -feature adds html comments to the output. The 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 the result in your browser. "debugTemplates" is restricted to development mode. It won't work in production.

Here is an example of comments added by debugTemplates :

<!-- GSP #2 START template: /home/.../views/\_carousel.gsp

precompiled: **false** lastmodified: … -->

.

.

.

<!-- GSP #2 END template: /home/.../views/\_carousel.gsp

rendering time: 115 ms -->

Each comment block has a unique id so that you can find the start & end of each template call.

**7.3 Tag Libraries**

Like [Java Server Pages](http://www.oracle.com/technetwork/java/javaee/jsp/index.html) (JSP), GSP supports the concept of custom tag libraries. Unlike JSP, Grails' tag library mechanism is simple, elegant and completely reloadable at runtime.

Quite simply, to create a tag library create a Groovy class that ends with the convention TagLib and place it within the grails-app/taglib directory:

class SimpleTagLib {

}

Now to create a tag create a Closure property that takes two arguments: the tag attributes and the body content:

class SimpleTagLib {

def simple = { attrs, body ->

}

}

The attrs argument is a Map of the attributes of the tag, whilst the body argument is a Closure that returns the body content when invoked:

class SimpleTagLib {

def emoticon = { attrs, body ->

out << body() << (attrs.happy == '**true**' ? " :-)" : " :-(")

}

}

As demonstrated above there is an implicit out variable that refers to the output Writer which you can use to append content to the response. Then you can reference the tag inside your GSP; no imports are necessary:

<g:emoticon happy="true">Hi John</g:emoticon>

To help IDEs like Spring Tool Suite (STS) and others autocomplete tag attributes, you should add Javadoc comments to your tag closures with @attrdescriptions. Since taglibs use Groovy code it can be difficult to reliably detect all usable 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)

}

}

**7.3.1 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\flash.html) object
* grailsApplication - The [GrailsApplication](http://grails.org/doc/3.0.x/api/grails/core/GrailsApplication.html) instance
* out - The response writer for writing to the output stream
* pageScope - A reference to the [pageScope](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tag%20Libraries\pageScope.html) object used for GSP rendering (i.e. the binding)
* params - The [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object for retrieving request parameters
* pluginContextPath - The context path to the plugin that contains the tag library
* request - The [HttpServletRequest](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpServletRequest.html) instance
* response - The [HttpServletResponse](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpServletResponse.html) instance
* servletContext - The [javax.servlet.ServletContext](http://download.oracle.com/javaee/1.4/api/javax/servlet/ServletContext.html) instance
* session - The [HttpSession](http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpSession.html) instance

**7.3.2 Simple Tags**

As demonstrated in the previous example it is easy to write simple tags that have no body and just output content. Another example is a dateFormat style tag:

def dateFormat = { attrs, body ->

out << **new** java.text.SimpleDateFormat(attrs.format).format(attrs.date)

}

The above uses Java's SimpleDateFormat class to format a date and then write it to the response. The tag can then be used within a GSP as follows:

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

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

def formatBook = { attrs, body ->

out << "<div id="${attrs.book.id}">"

out << "Title : ${attrs.book.title}"

out << "</div>"

}

Although this approach may be tempting it is not very clean. A better approach would be to reuse the [render](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\render.html) tag:

def formatBook = { attrs, body ->

out << render(template: "bookTemplate", model: [book: attrs.book])

}

And then have a separate GSP template that does the actual rendering.

**7.3.3 Logical Tags**

You can also create logical tags where the body of the tag is only output once a set of conditions have been met. An example of this may be a set of security tags:

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 correct set of access privileges:

<g:isAdmin user="${myUser}">

// some restricted content

</g:isAdmin>

**7.3.4 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 times method to iterate the specified number of times:

<g:repeat times="3">

<p>Repeat this 3 times! Current repeat = ${it}</p>

</g:repeat>

Notice how in this example we use the implicit it variable to refer to the current number. This works because when we invoked the body we passed in the current value inside the iteration:

out << body(num)

That value is then passed as the default variable it to the tag. However, if you have nested tags this can lead to conflicts, so you should instead name the variables that the body uses:

def repeat = { attrs, body ->

def **var** = attrs.**var** ?: "num"

attrs.times?.toInteger()?.times { num ->

out << body((**var**):num)

}

}

Here we check if there is a var attribute and if there is use that as the name to pass into the body invocation on this line:

out << body((**var**):num)

Note the usage of the parenthesis around the variable name. If you omit these Groovy assumes you are using a String key and not referring to the variable itself.

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

<g:repeat times="3" var="j">

<p>Repeat this 3 times! Current repeat = ${j}</p>

</g:repeat>

Notice how we use the var attribute to define the name of the variable j and then we are able to reference that variable within the body of the tag.

**7.3.5 Tag Namespaces**

By default, tags are added to the default Grails namespace and are used with the g: prefix in GSP pages. However, you can specify a different namespace by adding a static property to your TagLibclass:

class SimpleTagLib {

**static** namespace = "my"

def example = { attrs ->

…

}

}

Here we have specified a namespace of my and hence the tags in this tag lib must then be referenced from GSP pages like this:

<my:example name="..." />

where the prefix is the same as the value of the static namespace property. Namespaces are particularly useful for plugins.

Tags within namespaces can be invoked as methods using the namespace as a prefix to the method call:

out << my.example(name:"foo")

This works from GSP, controllers or tag libraries

**7.3.6 Using JSP Tag Libraries**

In addition to the simplified tag library mechanism provided by GSP, you can also use JSP tags from GSP. To do so simply declare the JSP to use with the taglib directive:

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

Besides this you have to configure Grails to scan for the JSP tld files. This is configured with the grails.gsp.tldScanPattern setting. It accepts a comma separated String value. Spring's PathMatchingResourcePatternResolver is used to resolve the patterns.

For example you could scan for all available tld files by adding this to Config.groovy:

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

JSTL standard library is no more added as a dependency by default. In case you are using JSTL, you should also add these dependencies to BuildConfig.groovy:

runtime 'javax.servlet:jstl:1.1.2'

runtime 'taglibs:standard:1.1.2'

Then you can use JSP tags like any other tag:

<fmt:formatNumber value="${10}" pattern=".00"/>

With the added bonus that you can invoke JSP tags like methods:

${fmt.formatNumber(value:10, pattern:".00")}

**7.3.7 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 expressions.

Internally Grails intercepts calls to taglib closures. The "out" that is available in a taglib is mapped to a java.io.Writer implementation that writes to a buffer that "captures" the output of the taglib call. This buffer is the return value of a tag library call when it's used as a function.

If the tag is listed in the library's static returnObjectForTags array, then its return value will written to the output when it's used as a normal tag. The return value of the tag lib closure will be returned as-is if it's used as a function in GSP expressions or other taglibs.

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

Example:

class ObjectReturningTagLib {

**static** namespace = "cms"

**static** returnObjectForTags = ['content']

def content = { attrs, body ->

CmsContent.findByCode(attrs.code)?.content

}

}

Given this example cmd.content(code:'something') call in another taglib or GSP expression would return the value "CmsContent.content" directly to the caller without wrapping the return value in a buffer. It might be worth doing so also because of performance optimization reasons. There is no need to wrap the tag return value in an output buffer in such cases.

**7.4 URL Mappings**

Throughout the documentation so far the convention used for URLs has been the default of /controller/action/id. However, this convention is not hard wired into Grails and is in fact controlled by a URL Mappings class located at grails-app/conf/UrlMappings.groovy.

The UrlMappings class contains a single property called mappings that has been assigned a block of code:

class UrlMappings {

**static** mappings = {

}

}

**7.4.1 Mapping to Controllers and Actions**

To create a simple mapping simply use a relative URL as the method name and specify named parameters for the controller and action to map to:

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

In this case we've mapped the URL /product to the list action of the ProductController. Omit the action definition to map to the default action of the controller:

"/product"(controller: "product")

An alternative syntax is to assign the controller and action to use within a block passed to the method:

"/product" {

controller = "product"

action = "list"

}

Which syntax you use is largely dependent on personal preference.

If you have mappings that all fall under a particular path you can group mappings with the group method:

group "/product", {

"/apple"(controller:"product", id:"apple")

"/htc"(controller:"product", id:"htc")

}

To rewrite one URI onto another explicit URI (rather than a controller/action pair) do something like this:

"/hello"(uri: "/hello.dispatch")

Rewriting specific URIs is often useful when integrating with other frameworks.

**7.4.2 Mapping to REST resources**

Since Grails 2.3, it possible to create RESTful URL mappings that map onto controllers by convention. The syntax to do so is as follows:

"/books"(resources:'book')

You define a base URI and the name of the controller to map to using the resources parameter. The above mapping will result in the following URLs:

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URI** | **Grails Action** |
| GET | /books | index |
| GET | /books/create | create |
| POST | /books | save |
| GET | /books/${id} | show |
| GET | /books/${id}/edit | edit |
| PUT | /books/${id} | update |
| DELETE | /books/${id} | delete |

If you wish to include or exclude any of the generated URL mappings you can do so with the includes or excludes parameter, which accepts the name of the Grails action to include or exclude:

"/books"(resources:'book', excludes:['delete', 'update'])

or

"/books"(resources:'book', includes:['index', 'show'])

**Single resources**

A single resource is a resource for which there is only one (possibly per user) in the system. You can create a single resource using the resource parameter (as oppose to resources):

"/book"(resource:'book')

This results in the following URL mappings:

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URI** | **Grails Action** |
| GET | /book/create | create |
| POST | /book | save |
| GET | /book | show |
| GET | /book/edit | edit |
| PUT | /book | update |
| DELETE | /book | delete |

The main difference is that the id is not included in the URL mapping.

**Nested Resources**

You can nest resource mappings to generate child resources. For example:

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

"/authors"(resources:"author")

}

The above will result in the following URL mappings:

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URL** | **Grails Action** |
| GET | /books/${bookId}/authors | index |
| GET | /books/${bookId}/authors/create | create |
| POST | /books/${bookId}/authors | save |
| GET | /books/${bookId}/authors/${id} | show |
| GET | /books/${bookId}/authors/edit/${id} | edit |
| PUT | /books/${bookId}/authors/${id} | update |
| DELETE | /books/${bookId}/authors/${id} | delete |

You can also nest regular URL mappings within a resource mapping:

"/books"(resources: "book") {

"/publisher"(controller:"publisher")

}

This will result in the following URL being available:

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URL** | **Grails Action** |
| GET | /books/1/publisher | index |

**Linking to RESTful Mappings**

You can link to any URL mapping created with the g:link tag provided by Grails simply by referencing the controller and action to link to:

<g:link controller="book" action="index">My Link</g:link>

As a convenience you can also pass a domain instance to the resource attribute of the link tag:

<g:link resource="${book}">My Link</g:link>

This will automatically produce the correct link (in this case "/books/1" for an id of "1").

The case of nested resources is a little different as they typically required two identifiers (the id of the resource and the one it is nested within). For example given the nested resources:

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

"/authors"(resources:"author")

}

If you wished to link to the show action of the author controller, you would write:

// Results in /books/1/authors/2

<g:link controller="author" action="show" method="GET" params="[bookId:1]" id="2">The Author</g:link>

However, to make this more concise there is a resource attribute to the link tag which can be used instead:

// Results in /books/1/authors/2

<g:link resource="book/author" action="show" bookId="1" id="2">My Link</g:link>

The resource attribute accepts a path to the resource separated by a slash (in this case "book/author"). The attributes of the tag can be used to specify the necessary bookId parameter.

**7.4.3 Redirects In URL Mappings**

Since Grails 2.3, it is possible to define URL mappings which specify a redirect. When a URL mapping specifies a redirect, any time that mapping 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 redirect to or must provide a Map representing the target of the redirect. That Map is structured just like the Map that may be passed as an argument to the redirect method in a controller.

"/viewBooks"(redirect: '/books/list')

"/viewAuthors"(redirect: [controller: 'author', action: 'list'])

"/viewPublishers"(redirect: [controller: 'publisher', action: 'list', permanent: **true**])

Request parameters that were part of the original request will be included in the redirect.

**7.4.4 Embedded Variables**

**Simple Variables**

The previous section demonstrated how to map simple URLs with concrete "tokens". In URL mapping speak tokens are the sequence of characters between each slash, '/'. A concrete token is one which is well defined such as as /product. However, in many circumstances you don't know what the value of a particular token will be until runtime. In this case you can use variable placeholders within the URL for example:

**static** mappings = {

"/product/$id"(controller: "product")

}

In this case by embedding a $id variable as the second token Grails will automatically map the second token into a parameter (available via the [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object) called id. For example given the URL/product/MacBook, the following code will render "MacBook" to the response:

class ProductController {

def index() { render params.id }

}

You can of course construct more complex examples of mappings. For example the traditional blog URL format could be mapped as follows:

**static** mappings = {

"/$blog/$year/$month/$day/$id"(controller: "blog", action: "show")

}

The above mapping would 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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object with values available for year, month, day, id and so on.

**Dynamic Controller and Action Names**

Variables can also be used to dynamically construct the controller and action name. In fact the default Grails URL mappings use this technique:

**static** mappings = {

"/$controller/$action?/$id?"()

}

Here the name of the controller, action and id are implicitly obtained from the variables controller, action and id embedded within the URL.

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

**static** mappings = {

"/$controller" {

action = { params.goHere }

}

}

**Optional Variables**

Another characteristic of the default mapping is the ability to append a ? at the end of a variable to make it an optional token. In a further example this technique could be applied to the blog URL mapping to have more flexible linking:

**static** mappings = {

"/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")

}

With this mapping all of these URLs would match with only the relevant parameters being populated in the [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object:

/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 property in a controller:

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 the block passed to the mapping:

"/holiday/win" {

id = "Marrakech"

year = 2007

}

This variables will be available within the [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object passed to the controller.

**Dynamically Resolved Variables**

The hard coded arbitrary variables are useful, but sometimes you need to calculate the name of the variable based on runtime factors. This is also possible by assigning a block to the variable name:

"/holiday/win" {

id = { params.id }

isEligible = { session.user != **null** } // must be logged in

}

In the above case the code within the blocks is resolved when the URL is actually matched and hence can be used in combination with all sorts of logic.

**7.4.5 Mapping to Views**

You can resolve a URL to a view without a controller or action involved. For example to map the root URL / to a GSP at the location grails-app/views/index.gsp you could use:

**static** mappings = {

"/"(view: "/index") // map the root URL

}

Alternatively if you need a view that is specific to a given controller you could use:

**static** mappings = {

"/help"(controller: "site", view: "help") // to a view **for** a controller

}

**7.4.6 Mapping to Response Codes**

Grails also lets you map HTTP response codes to controllers, actions or views. Just use a method name that matches the response code you are interested in:

**static** mappings = {

"403"(controller: "errors", action: "forbidden")

"404"(controller: "errors", action: "notFound")

"500"(controller: "errors", action: "serverError")

}

Or 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 illegalArgument action in ErrorsController, a NullPointerException will be handled by the nullPointer action, and a MyException will be handled by the customException action. Other exceptions will be handled by the catch-all rule and use the /errors/serverError view.

You can access the exception from your custom error handing view or controller action using the request's exception attribute like so:

class ErrorController {

def handleError() {

def exception = request.exception

// perform desired processing to handle the exception

}

}

If your error-handling controller action throws an exception as well, you'll end up with a StackOverflowException.

**7.4.7 Mapping to HTTP methods**

URL mappings can also be configured to map based on the HTTP method (GET, POST, PUT or DELETE). This is very useful for RESTful APIs and for restricting mappings based on HTTP method.

As an example the following mappings provide a RESTful API URL mappings for the ProductController:

**static** mappings = {

"/product/$id"(controller:"product", action: "update", method: "PUT")

}

**7.4.8 Mapping Wildcards**

Grails' URL mappings mechanism also supports wildcard mappings. For example consider the following mapping:

**static** mappings = {

"/images/\*.jpg"(controller: "image")

}

This mapping will match all paths to images such as /image/logo.jpg. Of course you can achieve the same effect with a variable:

**static** mappings = {

"/images/$name.jpg"(controller: "image")

}

However, you can also use double wildcards to match more than one level below:

**static** mappings = {

"/images/\*\*.jpg"(controller: "image")

}

In this cases the mapping will match /image/logo.jpg as well as /image/other/logo.jpg. Even better you can use a double wildcard variable:

**static** mappings = {

// will match /image/logo.jpg and /image/other/logo.jpg

"/images/$name\*\*.jpg"(controller: "image")

}

In this case it will store the path matched by the wildcard inside a name parameter obtainable from the [params](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\params.html) object:

def name = params.name

println name // prints "logo" or "other/logo"

If you use wildcard URL mappings then you may want to exclude certain URIs from Grails' URL mapping process. To do this you can provide an excludes setting inside the UrlMappings.groovy class:

class UrlMappings {

**static** excludes = ["/images/\*", "/css/\*"]

**static** mappings = {

…

}

}

In this case Grails won't attempt to match any URIs that start with /images or /css.

**7.4.9 Automatic Link Re-Writing**

Another great feature of URL mappings is that they automatically customize the behaviour of the [link](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Tags\link.html) tag so that changing the mappings don't require you to go and change all of your links.

This is done through a URL re-writing technique that reverse engineers the links from the URL mappings. So given a mapping such as the blog one from an earlier section:

**static** mappings = {

"/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")

}

If you use the link tag as follows:

<g:link controller="blog" action="show"

params="[blog:'fred', year:2007]">

My Blog

</g:link>

<g:link controller="blog" action="show"

params="[blog:'fred', year:2007, month:10]">

My Blog - 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>

**7.4.10 Applying Constraints**

URL Mappings also support Grails' unified [validation constraints](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#constraints) mechanism, which lets you further "constrain" how a URL is matched. For example, if we revisit the blog sample code from earlier, the mapping currently looks like this:

**static** mappings = {

"/$blog/$year?/$month?/$day?/$id?"(controller:"blog", action:"show")

}

This allows URLs such as:

/graemerocher/2007/01/10/my\_funky\_blog\_entry

However, it would also allow:

/graemerocher/not\_a\_year/not\_a\_month/not\_a\_day/my\_funky\_blog\_entry

This is problematic as it forces you to do some clever parsing in the controller code. Luckily, URL Mappings can be constrained to further validate the URL tokens:

"/$blog/$year?/$month?/$day?/$id?" {

controller = "blog"

action = "show"

constraints {

year(matches:/\d{4}/)

month(matches:/\d{2}/)

day(matches:/\d{2}/)

}

}

In this case the constraints ensure that the year, month and day parameters match a particular valid pattern thus relieving you of that burden later on.

**7.4.11 Named URL Mappings**

URL Mappings also support named mappings, that is mappings which have a name associated with them. The name may be used to refer to a specific mapping when links are generated.

The syntax for defining a named mapping is as follows:

**static** mappings = {

name <mapping name>: <url pattern> {

// …

}

}

For example:

**static** mappings = {

name personList: "/showPeople" {

controller = 'person'

action = 'list'

}

name accountDetails: "/details/$acctNumber" {

controller = 'product'

action = 'accountDetails'

}

}

The mapping may be referenced in a link tag in a GSP.

<g:link mapping="personList">List People</g:link>

That would result in:

<a href="/showPeople">List People</a>

Parameters may be specified using the params attribute.

<g:link mapping="accountDetails" params="[acctNumber:'8675309']">

Show Account

</g:link>

That would result in:

<a href="/details/8675309">Show Account</a>

Alternatively you may reference a named mapping using the link namespace.

<link:personList>List People</link:personList>

That would result in:

<a href="/showPeople">List People</a>

The link namespace approach allows parameters to be specified as attributes.

<link:accountDetails acctNumber="8675309">Show Account</link:accountDetails>

That would result in:

<a href="/details/8675309">Show Account</a>

To specify attributes that should be applied to the generated href, specify a Map value to the attrs attribute. These attributes will be applied directly to the href, not passed through to be used as request parameters.

<link:accountDetails attrs="[class: 'fancy']" acctNumber="8675309">

Show Account

</link:accountDetails>

That would result in:

<a href="/details/8675309" class="fancy">Show Account</a>

**7.4.12 Customizing URL Formats**

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

// grails-app/conf/Config.groovy

grails.web.url.converter = 'hyphenated'

Arbitrary strategies may be plugged in by providing a class which implements the [UrlConverter](http://grails.org/doc/3.0.x/api/grails/web/UrlConverter.html) interface and adding an instance of that class to the Spring application context with the bean name ofgrails.web.UrlConverter.BEAN\_NAME. If Grails finds a bean in the context with that name, it will be used as the default converter and there is no need to assign a value to thegrails.web.url.converter config property.

// src/groovy/com/myapplication/MyUrlConverterImpl.groovy

**package** com.myapplication

class MyUrlConverterImpl **implements** grails.web.UrlConverter {

String toUrlElement(String propertyOrClassName) {

// **return** some representation of a property or class name that should be used in URLs…

}

}

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

beans = {

"${grails.web.UrlConverter.BEAN\_NAME}"(com.myapplication.MyUrlConverterImpl)

}

**7.4.13 Namespaced Controllers**

If an application defines multiple controllers with the same name in different packages, the controllers must be defined in a namespace. The way to define a namespace for a controller is to define a static property named namespace in the controller and assign a String to the property that represents the 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 variable needs to be part of the URL mapping.

// grails-app/conf/UrlMappings.groovy

class UrlMappings {

**static** mappings = {

'/userAdmin' {

controller = 'admin'

namespace = 'users'

}

'/reportAdmin' {

controller = 'admin'

namespace = 'reports'

}

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

}

}

Reverse URL mappings also require that the namespace be specified.

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

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

When resolving a URL mapping (forward or reverse) to a namespaced controller, a mapping will only match if the namespace has been provided. If the application provides several controllers with the same name in different packages, at most 1 of them may be defined without a namespace property. If there are multiple controllers with the same name that do not define a namespace property, the framework will not know how to distinguish between 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 controller provided by the application and for neither of the controllers to define a namespace property as long as the controllers are in separate packages. For example, an application may include a controller named com.accounting.ReportingController and the application may use a plugin which provides a controller named com.humanresources.ReportingController. The only issue with that is the URL mapping for the controller provided by the plugin needs to be explicit in specifying that the mapping applies to the ReportingController which is provided by the plugin.

See the following example.

**static** mappings = {

"/accountingReports" {

controller = "reporting"

}

"/humanResourceReports" {

controller = "reporting"

plugin = "humanResources"

}

}

With that mapping in place, a request to /accountingReports will be handled by the ReportingController which is defined in the application. A request to /humanResourceReports will be handled by the ReportingController which is provided by the humanResources plugin.

There could be any number of ReportingController controllers provided by any number of plugins but no plugin may provide more than one ReportingController even if they are defined in separate packages.

Assigning a value to the plugin variable in the mapping is only required if there are multiple controllers with the same name available at runtime provided by the application and/or plugins. If thehumanResources plugin provides a ReportingController and there is no other 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.

**7.5 Interceptors**

Although Grails [controllers](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#controllers) support fine grained interceptors, these are only really useful when applied to a few controllers and become difficult to manage with larger applications.

To solve this you can create standalone Interceptors using the [create-interceptor](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Command%20Line\create-interceptor.html) command:

$ grails create-interceptor MyInterceptor

The above command will create an Interceptor in the grails-app/controllers directory with the following default contents:

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 for backwards compatibility but are considered deprecated.

The new interceptors concept in Grails 3.0 is superior in a number of ways, most significantly interceptors can using Groovy's CompileStatic annotation to optimize performance (something which is often critical as interceptors can be executed for every request.)

**7.5.1 Defining Interceptors**

By default interceptors will match the controller name they apply to be convention. For example if you have an interceptor called BookInterceptor then all requests the actions of the BookControllerwill trigger the interceptor.

An Interceptor implements the [Interceptor](http://grails.org/doc/3.0.x/api/grails/artefact/Interceptor.html) trait and provides 3 methods that can be used to intercept requests:

/\*\*

\* 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 action by returning false.

The after method is executed after an action executes and can halt view rendering if it returns false. The after method can also modify the view or 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 and if an exception occurs, the exception is available using the throwable property of the [Interceptor](http://grails.org/doc/3.0.x/api/grails/artefact/Interceptor.html) trait.

**7.5.2 Matching Requests with Inteceptors**

As mention in the previous section, by default an interceptor will match only requests to the associated controller by convention. However you can configure the interceptor to match any request using the match or matchAll methods defined in the [Interceptor API](http://grails.org/doc/3.0.x/api/grails/artefact/Interceptor.html).

The matching methods return a [Matcher](http://grails.org/doc/3.0.x/api/grails/interceptors/Matcher.html) instance which can be used to configure how the interceptor matches the request.

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() {

…

}

}

All named arguments accept either a String or a Regex expression. 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 (cannot be used in combination with other arguments)

**7.5.3 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 should should run first or last respectively.

Note that if you write an interceptor that is to be used by others it is better increment or decrement the HIGHEST\_PRECEDENCE and LOWEST\_PRECEDENCE to allow other interceptors to be inserted before or after the interceptor you are authoring:

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 follows:

logger 'grails.artefact.Interceptor', DEBUG, ['STDOUT'], **false**

You can override any interceptors default order by using bean override configuration in grails-app/conf/application.yml:

beans:

authInterceptor:

order: 50

Or in grails-app/conf/application.groovy:

beans {

authInterceptor {

order = 50

}

}

Thus giving you complete control over interceptor execution order.

**7.6 Content Negotiation**

Grails has built in support for [Content negotiation](http://en.wikipedia.org/wiki/Content_negotiation) using either the HTTP Accept header, an explicit format request parameter or the extension of a mapped URI.

**Configuring Mime Types**

Before you can start dealing with content negotiation you need to tell Grails what content types you wish to support. By default Grails comes configured with a number of different content types withingrails-app/conf/Config.groovy using the grails.mime.types setting:

grails.mime.types = [ // the first one is the **default** format

all: '\*/\*', // 'all' maps to '\*' or the first available format in withFormat

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' or 'application/xml' media types as simply 'xml'. You can add your own types by simply adding new entries into the map. The first one is the default format.

**Content Negotiation using the format parameter**

Let's say a controller action can return a resource in a variety of formats: HTML, XML, and JSON. What format will the client get? The easiest and most reliable way for the client to control this is through a format URL parameter.

So if you, as a browser or some other client, want a resource as XML, you can use a URL like this:

http://my.domain.org/books?format=xml

The result of this on the server side is a format property on the response object with the value xml . You could code your controller action to return XML based on this property, but you can also make use of the controller-specific withFormat() 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 content type. So if the preferred format is html then Grails will execute the html() call only. Each 'block' can either be a map model for the corresponding view (as we are doing for 'html' in the above example) or a closure. The closure can contain any standard action code, for example it can return a model or render content directly.

When no format matches explicitly, a **(wildcard) block can be used to handle all other formats.**

**There is a special format, "all", that is handled differently from the explicit formats. If "all" is specified (normally this happens through the Accept header - see below), then the first block ofwithFormat() is executed when there isn't a**(wildcard) block available.

You should not add an explicit "all" block. In this example, a format of "all" will trigger the html handler (html is the first block and there is no \* block).

withFormat {

html bookList: books

json { render books as JSON }

xml { render books as XML }

}

When using [withFormat](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\withFormat.html) make sure it is the last call in your controller action as the return value of the withFormat method is used by the action to dictate what happens next.

**Using the Accept header**

Every incoming HTTP request has a special [Accept](http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html) header that defines what media types (or mime types) a client can "accept". In older browsers this is typically:

\*/\*

which simply means anything. However, newer browsers send more interesting values such as this one sent by Firefox:

text/xml, application/xml, application/xhtml+xml, text/html;q=0.9,

text/plain;q=0.8, image/png, \*/\*;q=0.5

This particular accept header is unhelpful because it indicates that XML is the preferred response format whereas the user is really expecting HTML. That's why Grails ignores the accept header by default for browsers. However, non-browser clients are typically more specific in their requirements and can send accept headers such as

application/json

As mentioned the default configuration in Grails is to ignore the accept header for browsers. This is done by the configuration setting grails.mime.disable.accept.header.userAgents, which is configured to detect the major rendering engines and ignore their ACCEPT headers. This allows Grails' content negotiation to continue to work for non-browser clients:

grails.mime.disable.accept.header.userAgents = ['Gecko', 'WebKit', 'Presto', 'Trident']

For example, if it sees the accept header above ('application/json') it will set format to json as you'd expect. And of course this works with the withFormat() method in just the same way as when theformat URL parameter is set (although the URL parameter takes precedence).

An accept header of '\*/\*' results in a value of all for the format property.

If the accept header is used but contains no registered content types, Grails will assume a broken browser is making the request and will set the HTML format - note that this is different from how the other content negotiation modes work as those would activate the "all" format!

**Request format vs. Response format**

As of Grails 2.0, there is a separate notion of the *request* format and the *response* format. The request format is dictated by the CONTENT\_TYPE header and is typically used to detect if the incoming request can be parsed into XML or JSON, whilst the response format uses the file extension, format parameter or ACCEPT header to attempt to deliver an appropriate response to the client.

The [withFormat](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\withFormat.html) available on controllers deals specifically with the response format. If you wish to add logic that deals with the request format then you can do so using a separate withFormat method available on the request:

request.withFormat {

xml {

// read XML

}

json {

// read JSON

}

}

**Content Negotiation with the format Request Parameter**

If fiddling with request headers if not your favorite activity you can override the format used by specifying a format request parameter:

/book/list?format=xml

You can also define this parameter in the [URL Mappings](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#urlmappings) definition:

"/book/list"(controller:"book", action:"list") {

format = "xml"

}

**Content Negotiation with URI Extensions**

Grails also supports content negotiation using URI extensions. For example given the following URI:

/book/list.xml

This works as a result of the default URL Mapping definition which is:

"/$controller/$action?/$id?(.$format)?"{

Note the inclusion of the format variable in the path. If you do not wish to use content negotiation via the file extension then simply remove this part of the URL mapping:

"/$controller/$action?/$id?"{

**Testing Content Negotiation**

To test content negotiation in a unit or integration test (see the section on [Testing](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#testing)) you can either manipulate the incoming request headers:

void testJavascriptOutput() {

def controller = **new** TestController()

controller.request.addHeader "Accept",

"text/javascript, text/html, application/xml, text/xml, \*/\*"

controller.testAction()

assertEquals "alert('hello')", controller.response.contentAsString

}

Or you can set the format parameter to achieve a similar effect:

void testJavascriptOutput() {

def controller = **new** TestController()

controller.params.format = 'js'

controller.testAction()

assertEquals "alert('hello')", controller.response.contentAsString

}

# 8 Traits - Reference Documentation

**Authors:** Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, Lari Hotari

**Version:** 3.0.1

## Table of Contents

[**8.1**Traits Provided](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\traits.html#traitsprovided)

[**8.1.1**WebAttributes Trait Example](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\traits.html#example)

# 8 Traits

### Overview

Grails provides a number of traits which provide access to properties and behavior that may be accessed from various Grails artefacts as well as arbitrary Groovy classes which are part of a Grails project. Many of these traits are automatically added to Grails artefact classes (like controllers and taglibs, for example) and are easy to add to other classes.

## 8.1 Traits Provided

## Traits Provided By Grails

Grails artefacts are automatically augmented with certain traits at compile time.

#### Domain Class Traits

* [grails.artefact.DomainClass](http://grails.org/doc/3.0.x/api/grails/artefact/DomainClass.html)
* [grails.web.databinding.WebDataBinding](http://grails.org/doc/3.0.x/api/grails/web/databinding/WebDataBinding.html)
* org.grails.datastore.gorm.GormEntity
* org.grails.datastore.gorm.GormValidateable

#### Controller Traits

* [grails.artefact.gsp.TagLibraryInvoker](http://grails.org/doc/3.0.x/api/grails/artefact/gsp/TagLibraryInvoker.html)
* [grails.artefact.AsyncController](http://grails.org/doc/3.0.x/api/grails/artefact/AsyncController.html)
* [grails.artefact.controller.RestResponder](http://grails.org/doc/3.0.x/api/grails/artefact/controller/RestResponder.html)
* [grails.artefact.Controller](http://grails.org/doc/3.0.x/api/grails/artefact/Controller.html)

#### Interceptor Trait

* [grails.artefact.Interceptor](http://grails.org/doc/3.0.x/api/grails/artefact/Interceptor.html)

#### Tag Library Trait

* [grails.artefact.TagLibrary](http://grails.org/doc/3.0.x/api/grails/artefact/TagLibrary.html)

#### Service Trait

* [grails.artefact.Service](http://grails.org/doc/3.0.x/api/grails/artefact/Service.html)

Below is a list of other traits provided by the framework. The javadocs provide more detail about methods and properties related to each trait.

|  |  |
| --- | --- |
| **Trait** | **Brief Description** |
| [grails.web.api.WebAttributes](http://grails.org/doc/3.0.x/api/grails/web/api/WebAttributes.html) | Common Web Attributes |
| [grails.web.api.ServletAttributes](http://grails.org/doc/3.0.x/api/grails/web/api/ServletAttributes.html) | Servlet API Attributes |
| [grails.web.databinding.DataBinder](http://grails.org/doc/3.0.x/api/grails/web/databinding/DataBinder.html) | Data Binding API |
| [grails.artefact.controller.support.RequestForwarder](http://grails.org/doc/3.0.x/api/grails/artefact/controller/support/RequestForwarder.html) | Request Forwarding API |
| [grails.artefact.controller.support.ResponseRedirector](http://grails.org/doc/3.0.x/api/grails/artefact/controller/support/ResponseRedirector.html) | Response Redirecting API |
| [grails.artefact.controller.support.ResponseRenderer](http://grails.org/doc/3.0.x/api/grails/artefact/controller/support/ResponseRenderer.html) | Response Rendering API |
| [grails.validation.Validateable](http://grails.org/doc/3.0.x/api/grails/validation/Validateable.html) | Validation API |

## 8.1.1 WebAttributes Trait Example

[WebAttributes](http://grails.org/doc/3.0.x/api/grails/web/api/WebAttributes.html) is one of the traits provided by the framework. Any Groovy class may implement this trait to inherit all of the properties and behaviors 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

}

}

**9 Web Services - Reference Documentation**

**Authors:** Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, Lari Hotari

**Version:** 3.0.1

**Table of Contents**

[**9.1**REST](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#REST)

[**9.1.1**Domain classes as REST resources](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#domainResources)

[**9.1.2**Mapping to REST resources](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#urlMappings)

[**9.1.3**Linking to REST resources](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#linkingToResources)

[**9.1.4**Versioning REST resources](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#versioningResources)

[**9.1.5**Implementing REST controllers](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#restfulControllers)

[**9.1.5.1**Extending the RestfulController super class](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#extendingRestfulController)

[**9.1.5.2**Implementing REST Controllers Step by Step](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#restControllersStepByStep)

[**9.1.5.3**Generating a REST controller using scaffolding](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#generatingRestControllers)

[**9.1.6**Customizing Response Rendering](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#renderers)

[**9.1.6.1**Customizing the Default Renderers](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#defaultRenderers)

[**9.1.6.2**Registering Custom Objects Marshallers](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#objectMarshallers)

[**9.1.6.3**Using Named Configurations for Object Marshallers](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#namedObjectMarshallers)

[**9.1.6.4**Implementing the ObjectMarshaller Interface](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#objectMarshallerInterface)

[**9.1.6.5**Implementing a Custom Renderer](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#customRenderers)

[**9.1.6.6**Using GSP to Customize Rendering](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#gspRenderers)

[**9.1.7**Hypermedia as the Engine of Application State](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#hypermedia)

[**9.1.7.1**HAL Support](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#hal)

[**9.1.7.2**Atom Support](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#atom)

[**9.1.7.3**Vnd.Error Support](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#vndError)

[**9.1.8**Customizing Binding of Resources](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#binding)

[**9.2**SOAP](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#SOAP)

[**9.3**RSS and Atom](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\webServices.html#RSSAndAtom)

**9 Web Services**

Web Services are all about providing a web API onto your web application and are typically implemented in either [REST](http://en.wikipedia.org/wiki/Representational_State_Transfer) or [SOAP](http://en.wikipedia.org/wiki/SOAP.)

**9.1 REST**

REST is not really a technology in itself, but more an architectural pattern. REST is very simple and just involves using plain XML or JSON as a communication medium, combined with URL patterns that are "representational" of the underlying system, and HTTP methods such as GET, PUT, POST and DELETE.

Each HTTP method maps to an action type. For example GET for retrieving data, POST for creating data, PUT for updating and so on.

Grails includes flexible features that make it easy to create RESTful APIs. Creating a RESTful resource can be as simple as one line of code, as demonstrated in the next section.

**9.1.1 Domain classes as REST resources**

The easiest way to create a RESTful API in Grails is to expose a domain class as a REST resource. This can be done by adding the 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 automatically be available as a REST resource in either XML or JSON formats. The transformation will automatically register the necessary [RESTful URL mapping](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#restfulMappings) and create a controller called 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/myapp/books/1, which will render the response like:

<?xml version="1.0" encoding="UTF-8"?>

<book id="1">

<title>The Stand</title>

</book>

If you change the URL to http://localhost:8080/myapp/books/1.json you will get a JSON response such as:

{"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 attribute of the Resource transformation:

**import** grails.**rest**.\*

@Resource(uri='/books', formats=['json', 'xml'])

class Book {

…

}

With the above example JSON will be prioritized. The list that is passed should contain the names of the formats that the resource should expose. The names of formats are defined in thegrails.mime.types setting of Config.groovy:

grails.mime.types = [

…

json: ['application/json', 'text/json'],

…

xml: ['text/xml', 'application/xml']

]

See the section on [Configuring Mime Types](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#contentNegotiation) 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 header. Here's an example using the Unix curl tool:

$ curl -i -H "Accept: application/json" localhost:8080/myapp/books/1

{"id":1,"title":"The Stand"}

This works thanks to Grails' [Content Negotiation](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#contentNegotiation) 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 Spider"}' localhost:8080/myapp/books

HTTP/1.1 201 Created

Server: Apache-Coyote/1.1

...

Updating can be done with a PUT request:

$ curl -i -X PUT -H "Content-Type: application/json" -d '{"title":"Along Came A Spider"}' localhost:8080/myapp/books/1

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

...

Finally a resource can be deleted with DELETE request:

$ curl -i -X DELETE localhost:8080/myapp/books/1

HTTP/1.1 204 No Content

Server: Apache-Coyote/1.1

...

As you can see, the Resource transformation enables all of the HTTP method verbs on the resource. You can enable only read-only capabilities by setting the readOnly attribute to true:

**import** grails.**rest**.\*

@Resource(uri='/books', readOnly=**true**)

class Book {

…

}

In this case POST, PUT and DELETE requests will be forbidden.

**9.1.2 Mapping to REST resources**

If you prefer to keep the declaration of the URL mapping in your UrlMappings.groovy file then simply removing the uri attribute of the Resource transformation and adding the following line toUrlMappings.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](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#restfulMappings) can be found in the [URL Mappings section](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#urlMappings) of the user guide.

**9.1.3 Linking to REST resources**

The link tag offers an easy way to link to any domain class resource:

<g:link resource="${book}">My Link</g:link>

However, currently you cannot use g:link to link to the DELETE action and most browsers do not support sending the DELETE method directly.

The best way to accomplish this is to use a form submit:

<form action="/book/2" method="post">

<input type="hidden" name="\_method" value="DELETE"/>

</form>

Grails supports overriding the request method via the hidden \_method parameter. This is for browser compatibility purposes. This is useful when using restful resource mappings to create powerful web interfaces. To make a link fire this type of event, perhaps capture all click events for links with a `data-method` attribute and issue a form submit via javascript.

**9.1.4 Versioning REST resources**

A common requirement with a REST API is to expose different versions at the same time. There are a few ways this can be achieved in Grails.

**Versioning using the URI**

A common approach is to use the URI to version APIs (although this approach is discouraged in favour of Hypermedia). For example, you can define the following URL mappings:

"/books/v1"(resources:"book", namespace:'v1')

"/books/v2"(resources:"book", namespace:'v2')

That will match the following controllers:

**package** myapp.v1

class BookController {

**static** namespace = 'v1'

}

**package** myapp.v2

class BookController {

**static** namespace = 'v2'

}

This approach has the disadvantage of requiring two different URI namespaces for your API.

**Versioning with the Accept-Version header**

As an alternative Grails supports the passing of an Accept-Version header from clients. For example you can define the following URL mappings:

"/books"(version:'1.0', resources:"book", namespace:'v1')

"/books"(version:'2.0', resources:"book", namespace:'v2')

Then in the client simply pass which version you need using the Accept-Version header:

$ curl -i -H "Accept-Version: 1.0" -X GET http://localhost:8080/myapp/books

**Versioning using Hypermedia / Mime Types**

Another approach to versioning is to use Mime Type definitions to declare the version of your custom media types (see the section on "Hypermedia as the Engine of Application State" for more information about Hypermedia concepts). For example, in Config.groovy you can declare a custom 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 Type of the request cannot be established then the first entry in the map is used for the response. If you have your new Mime Type at the top then Grails will always try and send back your new Mime Type if the requested Mime Type cannot be established.

Then override the renderer (see the section on "Customizing Response Rendering" for more information on custom renderers) to send back the custom 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.books.org.book+json", [v:"1.0"]))

bookRendererV2(JsonRenderer, myapp.v2.Book, **new** MimeType("application/vnd.books.org.book+json", [v:"2.0"]))

}

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://localhost:8080/myapp/books

**9.1.5 Implementing REST controllers**

The Resource transformation is a quick way to get started, but typically you'll want to customize the controller logic, the rendering of the response or extend the API to include additional actions.

**9.1.5.1 Extending the RestfulController super class**

The easiest way to get started doing so is to create a new controller for your resource that extends the grails.rest.RestfulController super class. For example:

class BookController **extends** RestfulController {

**static** responseFormats = ['json', 'xml']

BookController() {

**super**(Book)

}

}

To customize any logic you can just override the appropriate action. The following table provides the names of the action names and the URIs they map to:

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URI** | **Controller Action** |
| GET | /books | index |
| GET | /books/create | create |
| POST | /books | save |
| GET | /books/${id} | show |
| GET | /books/${id}/edit | edit |
| PUT | /books/${id} | update |
| DELETE | /books/${id} | delete |

Note that the create and edit actions are only needed if the controller exposes an HTML interface.

As an example, if you have a [nested resource](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#restfulMappings) then you would typically want to query both the parent and the child identifiers. For example, given the following URL mapping:

"/authors"(resources:'author') {

"/books"(resources:'book')

}

You could implement the nested controller as follows:

class BookController **extends** RestfulController {

**static** responseFormats = ['json', 'xml']

BookController() {

**super**(Book)

}

@Override

**protected** Book queryForResource(Serializable id) {

Book.where {

id == id && author.id = params.authorId

}.find()

}

}

The example above subclasses RestfulController and overrides the protected queryForResource method to customize the query for the resource to take into account the parent resource.

**Customizing Data Binding In A RestfulController Subclass**

The RestfulController class contains code which does data binding for actions like save and update. The class defines a getObjectToBind() method which returns a value which will be used as the source for data binding. For example, the update action does something like this...

class RestfulController<T> {

def update() {

T instance = // retrieve instance from the database...

instance.properties = getObjectToBind()

// …

}

// …

}

By default the getObjectToBind() method returns the [request](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\request.html) object. When the request object is used as the binding source, if the request has a body then the body will be parsed and its contents will be used to do the data binding, otherwise the request parameters will be used to do the data binding. Subclasses of RestfulController may override the getObjectToBind() method and return anything that is a valid binding source, including a [Map](http://docs.oracle.com/javase/6/docs/api/java/util/Map.html) or a [DataBindingSource](http://grails.org/doc/3.0.x/api/org/grails/databinding/DataBindingSource.html). For most use cases binding the request is appropriate but the getObjectToBind() method allows for changing that 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 is required for supporting Resource annotation with readOnly=true.

This is a template that can be used for subclassed RestfulController classes used in Resource annotations:

class SubclassRestfulController<T> **extends** RestfulController<T> {

SubclassRestfulController(Class<T> domainClass) {

**this**(domainClass, **false**)

}

SubclassRestfulController(Class<T> domainClass, boolean readOnly) {

**super**(domainClass, readOnly)

}

}

You can specify the super class of the controller that backs the Resource annotation with the superClass attribute.

**import** grails.**rest**.\*

@Resource(uri='/books', superClass=SubclassRestfulController)

class Book {

String title

**static** constraints = {

title blank:**false**

}

}

**9.1.5.2 Implementing REST Controllers Step by Step**

If you don't want to take advantage of the features provided by the RestfulController super class, then you can implement each HTTP verb yourself manually. The first step is to create a controller:

$ grails create-controller book

Then add some useful imports and enable readOnly by default:

**import** grails.transaction.\*

**import** **static** org.springframework.http.HttpStatus.\*

**import** **static** org.springframework.http.HttpMethod.\*

@Transactional(readOnly = **true**)

class BookController {

…

}

Recall that each HTTP verb matches a particular Grails action according to the following conventions:

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URI** | **Controller Action** |
| GET | /books | index |
| GET | /books/${id} | show |
| GET | /books/create | create |
| GET | /books/${id}/edit | edit |
| POST | /books | save |
| PUT | /books/${id} | update |
| DELETE | /books/${id} | delete |

The 'create' and 'edit' actions are already required if you plan to implement an HTML interface for the REST resource. They are there in order to render appropriate HTML forms to create and edit a resource. If this is not a requirement they can be discarded.

The key to implementing REST actions is the [respond](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\respond.html) method introduced in Grails 2.3. The respond method tries to produce the most appropriate response for the requested content type (JSON, XML, HTML etc.)

**Implementing the 'index' action**

For example, to implement the index action, simply call the respond method passing the list of objects to respond with:

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 total count. This is only required if you plan to support pagination via some user interface.

The respond method will, using [Content Negotiation](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#contentNegotiation), attempt to reply with the most appropriate response given the content type requested by the client (via the ACCEPT header or file extension).

If the content type is established to be HTML then a model will be produced such that the action above would be the equivalent of writing:

def index(Integer max) {

params.max = Math.min(max ?: 10, 100)

[bookList: Book.list(params), bookCount: Book.count()]

}

By providing an index.gsp file you can render an appropriate view for the given model. If the content type is something other than HTML then the respond method will attempt to lookup an appropriategrails.rest.render.Renderer instance that is capable of rendering the passed object. 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 custom renderer see the section on "Customizing Response Rendering".

**Implementing the 'show' action**

The show action, which is used to display and individual resource by id, can be implemented in one line of Groovy code (excluding the method signature):

def show(Book book) {

respond book

}

By specifying the domain instance as a parameter to the action Grails will automatically attempt to lookup the domain instance using the id parameter of the request. If the domain instance doesn't exist, then null will be passed into the action. The respond method will return a 404 error if null is passed otherwise once again it will attempt to render an appropriate response. If the format is HTML then an appropriate model will produced. The following 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 a resource as the first argument and mark it as Transactional with thegrails.transaction.Transactional transform:

@Transactional

def save(Book book) {

…

}

Then the first thing to do is check whether the resource has any [validation errors](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#validation) and if so respond with the errors:

**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 the case of other formats (JSON, XML etc.), the errors object itself will be rendered in the appropriate format and a status code of 422 (UNPROCESSABLE\_ENTITY) returned.

If there are no errors then the resource can be saved and an appropriate response sent:

book.save flush:**true**

withFormat {

html {

flash.message = message(code: '**default**.created.message', args: [message(code: 'book.label', **default**: 'Book'), book.id])

redirect book

}

'\*' { render status: CREATED }

}

In the case of HTML a redirect is issued to the originating resource and for other formats a status code of 201 (CREATED) is returned.

**Implementing the 'update' action**

The update action updates an existing resource representations and is largely similar to the save action. First define the method signature:

@Transactional

def update(Book book) {

…

}

If the resource exists then Grails will load the resource, otherwise null we passed. In the case of null, you should return a 404:

**if**(book == **null**) {

render status: NOT\_FOUND

}

**else** {

…

}

Then once again check for errors [validation errors](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\single.html#validation) 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 the case of other formats (JSON, XML etc.) the errors object itself will be rendered in the appropriate format and a status code of 422 (UNPROCESSABLE\_ENTITY) returned.

If there are no errors then the resource can be saved and an appropriate response sent:

book.save flush:**true**

withFormat {

html {

flash.message = message(code: '**default**.updated.message', args: [message(code: 'book.label', **default**: 'Book'), book.id])

redirect book

}

'\*' { render status: OK }

}

In the case of HTML a redirect is issued to the originating resource and for other formats a status code of 200 (OK) is returned.

**Implementing the 'delete' action**

The delete action deletes an existing resource. The implementation is largely similar to the update action, expect the delete() method is called instead:

book.delete flush:**true**

withFormat {

html {

flash.message = message(code: '**default**.deleted.message', args: [message(code: 'Book.label', **default**: 'Book'), book.id])

redirect action:"index", method:"GET"

}

'\*'{ render status: NO\_CONTENT }

}

Notice that for an HTML response a redirect is issued back to the index action, whilst for other content types a response code 204 (NO\_CONTENT) is returned.

**9.1.5.3 Generating a REST controller using scaffolding**

To see some of these concepts in action and help you get going the [Scaffolding plugin](http://grails.org/plugin/scaffolding), version 2.0 and above, can generate a REST ready controller for you, simply run the command:

$ grails generate-controller [Domain Class Name]

**9.1.6 Customizing Response Rendering**

There are several ways to customize response rendering in Grails.

**9.1.6.1 Customizing the Default Renderers**

The default renderers for XML and JSON can be found in the grails.rest.render.xml and grails.rest.render.json packages respectively. These use the Grails converters (grails.converters.XML and grails.converters.JSON) by default for response rendering.

You can easily customize response rendering using these default renderers. A common change you may want to make is to include or exclude certain properties from rendering.

**Including or Excluding Properties from Rendering**

As mentioned previously, Grails maintains a registry of grails.rest.render.Renderer instances. There are some default configured renderers and the ability to register or override renderers for a given domain class or even for a collection of domain classes. To include a particular property from rendering you need to register a custom renderer by defining a bean in grails-app/conf/spring/resources.groovy:

**import** grails.**rest**.render.xml.\*

beans = {

bookRenderer(XmlRenderer, Book) {

includes = ['title']

}

}

The bean name is not important (Grails will scan the application context for all registered renderer beans), but for 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. In other words, under the covers they essentially do the following:

**import** grails.converters.\*

…

render book as XML

// or render book as JSON

Why the separation between converters and renderers? Well a renderer has more flexibility to use whatever rendering technology you chose. When implementing a custom renderer you could use[Jackson](http://wiki.fasterxml.com/JacksonHome), [Gson](http://code.google.com/p/google-gson/) or any Java library to implement the renderer. Converters on the other hand are very much tied to Grails' own marshalling implementation.

**9.1.6.2 Registering Custom Objects Marshallers**

Grails' Converters feature the notion of an [ObjectMarshaller](http://grails.org/doc/3.0.x/api/org/grails/web/converters/marshaller/ObjectMarshaller.html) and each type can have a registered ObjectMarshaller. You can register custom ObjectMarshaller instances to completely customize response rendering. For example, you can define the following in BootStrap.init:

XML.registerObjectMarshaller Book, { Book book, XML xml ->

xml.attribute 'id', book.id

xml.build {

title(book.title)

}

}

You can customize the formatting of an individual value this way too. For example the [JodaTime plugin](http://grails.org/plugin/jodatime) does the following to support rendering of JodaTime dates in JSON output:

JSON.registerObjectMarshaller(DateTime) {

**return** it?.toString("yyyy-MM-dd'T'HH:mm:ss'Z'")

}

In the case of JSON it's often simple to use a map to customize output:

JSON.registerObjectMarshaller(Book) {

def map= [:]

map['titl'] = it.title

map['auth'] = it.author

**return** map

}

**Registering Custom Marshallers via Spring**

Note that if you have many custom marshallers it is recommended you split the registration of these into a separate class:

class CustomMarshallerRegistrar {

@javax.annotation.PostConstruct

void registerMarshallers() {

JSON.registerObjectMarshaller(DateTime) {

**return** it?.toString("yyyy-MM-dd'T'HH:mm:ss'Z'")

}

}

}

Then define this class as Spring bean in grails-app/conf/spring/resources.groovy:

beans = {

myCustomMarshallerRegistrar(CustomMarshallerRegistrar)

}

The PostConstruct annotation will get triggered on startup of your application.

**9.1.6.3 Using Named Configurations for Object Marshallers**

It is also possible to register named configurations. For example:

XML.createNamedConfig('publicApi') {

it.registerObjectMarshaller(Book) { Book book, XML xml ->

// **do** **public** API

}

}

XML.createNamedConfig('adminApi') {

it.registerObjectMarshaller(Book) { Book book, XML xml ->

// **do** admin API

}

}

Then when you use either the render or respond methods you can wrap the call in a named configuration if necessary to customize rendering per request:

XML.use( isAdmin ? 'adminApi' : 'publicApi') {

render book as XML

}

or

XML.use( isAdmin ? 'adminApi' : 'publicApi') {

respond book

}

**9.1.6.4 Implementing the ObjectMarshaller Interface**

For more complex marshallers it is recommended you implement the [ObjectMarshaller](http://grails.org/doc/3.0.x/api/org/grails/web/converters/marshaller/ObjectMarshaller.html) interface. For example given a domain class:

class Book {

String title

}

By default the output when using:

render book as XML

Would look like:

<book id="1">

<title>The Stand</title>

</book>

To write a custom marshaller you can do the following:

class BookMarshaller **implements** ObjectMarshaller<XML> {

**public** boolean supports(Object object) {

**return** object **instanceof** Book

}

**public** void marshalObject(Object object, XML converter) {

Book book = (Book)object

converter.chars book.title

}

}

And then register the marshaller with:

XML.registerObjectMarshaller(**new** BookMarshaller())

With the custom ObjectMarshaller in place, the output is now:

<book>The Stand</book>

**Customizing the Name of the Root Element**

If you wish the customize the name of the surrounding element, you can implement [NameAwareMarshaller](http://grails.org/doc/3.0.x/api/org/grails/web/converters/marshaller/NameAwareMarshaller.html):

class BookMarshaller **implements** ObjectMarshaller<XML>,NameAwareMarshaller {

...

String getElementName(Object o) {

**return** 'custom-book'

}

}

With the above change the output would now be:

<custom-book>The Stand</custom-book>

**Outputting Markup Using the Converters API or Builder**

With the passed Converter object you can explicitly code to the Converters API to stream markup to the response:

**public** void marshalObject(Object object, XML converter) {

Book book = (Book)object

converter.attribute 'id', book.id.toString()

converter.attribute 'date-released', book.dateReleased.toString()

converter.startNode 'title'

converter.chars book.title

converter.end()

}

The above code results in:

<book id="1" date-released="...">

<title>The Stand</title>

</book>

You can also use a builder notation to achieve a similar result (although the builder notation does not work for CompileStatic):

**public** void marshalObject(Object object, XML converter) {

Book b = (Book)object

converter.build {

book(id: b.id) {

title b.title

}

}

}

**Using the convertAnother Method to Recursively Convert Objects**

To create more complex responses you can use the convertAnother method to convert associations and other objects:

**public** void marshalObject(Object object, XML converter) {

Book book = (Book)object

converter.startNode 'title'

converter.chars book.title

converter.end()

**if** (book.authors) {

converter.startNode 'authors'

**for**(author in book.authors) {

converter.convertAnother author

}

converter.end()

}

}

**9.1.6.5 Implementing a Custom Renderer**

If you want even more control of the rendering or prefer to use your own marshalling techniques then you can implement your own Renderer instance. 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 MimeType(s) that are accepted (via the ACCEPT header or file extension) for the renderer.

To configure this renderer, simply add it is a bean to grails-app/conf/spring/resources.groovy:

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 you also need to change the binding if you plan to support POST and PUT requests. Grails will not automatically know how to bind data from a custom XML format to a domain class otherwise. See the section on "Customizing Binding of Resources" for further information.

**Container Renderers**

A grails.rest.render.ContainerRenderer is a renderer that renders responses for containers of objects (lists, maps, collections etc.). The interface is largely the same as the Renderer interface except for the addition of the getComponentType() method, which should return the "contained" type. For example:

class BookListRenderer **implements** ContainerRenderer<List, Book> {

Class<List> getTargetType() { List }

Class<Book> getComponentType() { Book }

MimeType[] getMimeTypes() { [ MimeType.XML] as MimeType[] }

void render(List object, RenderContext context) {

....

}

}

**9.1.6.6 Using GSP to Customize Rendering**

You can also customize rendering on a per action basis using Groovy Server Pages (GSP). For example given the show action mentioned previously:

def show(Book book) {

respond book

}

You could supply a show.xml.gsp file to customize the rendering of the XML:

<%@page contentType="application/xml"%>

<book id="${book.id}" title="${book.title}"/>

**9.1.7 Hypermedia as the Engine of Application State**

[HATEOS](http://en.wikipedia.org/wiki/HATEOAS), an abbreviation for Hypermedia as the Engine of Application State, is a common pattern applied to REST architectures that uses hypermedia and linking to define the REST API.

Hypermedia (also called Mime or Media Types) are used to describe the state of a REST resource, and links tell clients how to transition to the next state. The format of the response is typically JSON or XML, although standard formats such as [Atom](http://tools.ietf.org/html/rfc4287) and/or [HAL](http://stateless.co/hal_specification.html) are frequently used.

**9.1.7.1 HAL Support**

[HAL](http://stateless.co/hal_specification.html) is a standard exchange format commonly used when developing REST APIs that follow HATEOAS principals. An example HAL document representing a list of orders can be seen below:

{

"\_links": {

"self": { "href": "/orders" },

"next": { "href": "/orders?page=2" },

"find": {

"href": "/orders{?id}",

"templated": **true**

},

"admin": [{

"href": "/admins/2",

"title": "Fred"

}, {

"href": "/admins/5",

"title": "Kate"

}]

},

"currentlyProcessing": 14,

"shippedToday": 20,

"\_embedded": {

"order": [{

"\_links": {

"self": { "href": "/orders/123" },

"basket": { "href": "/baskets/98712" },

"customer": { "href": "/customers/7809" }

},

"total": 30.00,

"currency": "USD",

"status": "shipped"

}, {

"\_links": {

"self": { "href": "/orders/124" },

"basket": { "href": "/baskets/97213" },

"customer": { "href": "/customers/12369" }

},

"total": 20.00,

"currency": "USD",

"status": "processing"

}]

}

}

**Exposing Resources Using HAL**

To return HAL instead of regular JSON for a resource you can simply override the renderer in grails-app/conf/spring/resources.groovy with an instance ofgrails.rest.render.hal.HalJsonRenderer (or HalXmlRenderer for the XML variation):

**import** grails.**rest**.render.hal.\*

beans = {

halBookRenderer(HalJsonRenderer, **rest**.test.Book)

}

With the bean in place requesting the HAL content type will return HAL:

$ curl -i -H "Accept: application/hal+json" http://localhost:8080/myapp/books/1

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Type: application/hal+json;charset=ISO-8859-1

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books/1",

"hreflang": "en",

"type": "application/hal+json"

}

},

"title": ""The Stand""

}

To use HAL XML format simply change the renderer:

**import** grails.**rest**.render.hal.\*

beans = {

halBookRenderer(HalXmlRenderer, **rest**.test.Book)

}

**Rendering Collections Using HAL**

To return HAL instead of regular JSON for a list of resources you can simply override the renderer in grails-app/conf/spring/resources.groovy with an instance ofgrails.rest.render.hal.HalJsonCollectionRenderer:

**import** grails.**rest**.render.hal.\*

beans = {

halBookCollectionRenderer(HalJsonCollectionRenderer, **rest**.test.Book)

}

With the bean in place requesting the HAL content type will return HAL:

$ curl -i -H "Accept: application/hal+json" http://localhost:8080/myapp/books

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Type: application/hal+json;charset=UTF-8

Transfer-Encoding: chunked

Date: Thu, 17 Oct 2013 02:34:14 GMT

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books",

"hreflang": "en",

"type": "application/hal+json"

}

},

"\_embedded": {

"book": [

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books/1",

"hreflang": "en",

"type": "application/hal+json"

}

},

"title": "The Stand"

},

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books/2",

"hreflang": "en",

"type": "application/hal+json"

}

},

"title": "Infinite Jest"

},

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books/3",

"hreflang": "en",

"type": "application/hal+json"

}

},

"title": "Walden"

}

]

}

}

Notice that the key associated with the list of Book objects in the rendered JSON is book which is derived from the type of objects in the collection, namely Book. In order to customize the value of this key assign a value to the collectionName property on the HalJsonCollectionRenderer bean as shown below:

**import** grails.**rest**.render.hal.\*

beans = {

halBookCollectionRenderer(HalCollectionJsonRenderer, **rest**.test.Book) {

collectionName = 'publications'

}

}

With that in place the rendered HAL will look like the following:

$ curl -i -H "Accept: application/hal+json" http://localhost:8080/myapp/books

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Type: application/hal+json;charset=UTF-8

Transfer-Encoding: chunked

Date: Thu, 17 Oct 2013 02:34:14 GMT

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books",

"hreflang": "en",

"type": "application/hal+json"

}

},

"\_embedded": {

"publications": [

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books/1",

"hreflang": "en",

"type": "application/hal+json"

}

},

"title": "The Stand"

},

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books/2",

"hreflang": "en",

"type": "application/hal+json"

}

},

"title": "Infinite Jest"

},

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books/3",

"hreflang": "en",

"type": "application/hal+json"

}

},

"title": "Walden"

}

]

}

}

**Using Custom Media / Mime Types**

If you wish to use a custom Mime Type then you first need to declare the Mime Types in grails-app/conf/Config.groovy:

grails.mime.types = [

all: "\*/\*",

book: "application/vnd.books.org.book+json",

bookList: "application/vnd.books.org.booklist+json",

…

]

It is critical that place your new mime types after the 'all' Mime Type because if the Content Type of the request cannot be established then the first entry in the map is used for the response. If you have your new Mime Type at the top then Grails will always try and send back your new Mime Type if the requested Mime Type cannot be established.

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/vnd.books.org.book+json", [v:"1.0"]))

halBookListRenderer(HalJsonCollectionRenderer, **rest**.test.Book, **new** MimeType("application/vnd.books.org.booklist+json", [v:"1.0"]))

}

In the above example the first bean defines a HAL renderer for a single book instance that returns a Mime Type of application/vnd.books.org.book+json. The second bean defines the Mime Type used to render a collection of books (in this case application/vnd.books.org.booklist+json).

application/vnd.books.org.booklist+json is an example of a media-range (http://www.w3.org/Protocols/rfc2616/rfc2616.html - Header Field Definitions). This example uses entity (book) and operation (list) to form the media-range values but in reality, it may not be necessary to create a separate Mime type for each operation. Further, it may not be necessary to create Mime types at the entity level. See the section on "Versioning 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/myapp/books/1

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Type: application/vnd.books.org.book+json;charset=ISO-8859-1

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books/1",

"hreflang": "en",

"type": "application/vnd.books.org.book+json"

}

},

"title": ""The Stand""

}

**Customizing Link Rendering**

An important aspect of HATEOAS is the usage of links that describe the transitions the client can use to interact with the REST API. By default the HalJsonRenderer will automatically create links for you for associations and to the resource itself (using the "self" relationship).

However you can customize link rendering using the link method that is added to all domain classes annotated with grails.rest.Resource or any class annotated with grails.rest.Linkable. For example, the show action can be modified as follows to provide a new link in the resulting output:

def show(Book book) {

book.link rel:'publisher', href: g.link(resource:"publisher", params:[bookId: book.id])

respond book

}

Which will result in output such as:

{

"\_links": {

"self": {

"href": "http://localhost:8080/myapp/books/1",

"hreflang": "en",

"type": "application/vnd.books.org.book+json"

}

"publisher": {

"href": "http://localhost:8080/myapp/books/1/publisher",

"hreflang": "en"

}

},

"title": ""The Stand""

}

The link method can be passed named arguments that match the properties of the grails.rest.Link class.

**9.1.7.2 Atom Support**

[Atom](http://tools.ietf.org/html/rfc4287) is another standard interchange format used to implement REST APIs. An example of Atom output can be seen below:

<?xml version="1.0" encoding="utf-8"?>

<feed xmlns="http://www.w3.org/2005/Atom">

<title>Example Feed</title>

<link href="http://example.org/"/>

<updated>2003-12-13T18:30:02Z</updated>

<author>

<name>John Doe</name>

</author>

<id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>

<entry>

<title>Atom-Powered Robots Run Amok</title>

<link href="http://example.org/2003/12/13/atom03"/>

<id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>

<updated>2003-12-13T18:30:02Z</updated>

<summary>Some text.</summary>

</entry>

</feed>

To use Atom rendering again simply define a custom renderer:

**import** grails.**rest**.render.atom.\*

beans = {

halBookRenderer(AtomRenderer, **rest**.test.Book)

halBookListRenderer(AtomCollectionRenderer, **rest**.test.Book)

}

**9.1.7.3 Vnd.Error Support**

[Vnd.Error](https://github.com/blongden/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 will be sent back allow with a 422 respond code:

$ curl -i -H "Accept: application/json" -H "Content-Type: application/json" -X POST -d "" http://localhost:8080/myapp/books

HTTP/1.1 422 Unprocessable Entity

Server: Apache-Coyote/1.1

Content-Type: application/json;charset=ISO-8859-1

{"errors":[{"object":"**rest**.test.Book", "field":"title", "rejected-value":**null**, "message":"Property [title] of class [class **rest**.test.Book] cannot be **null**"}]}

If you wish to change the format to Vnd.Error then simply register grails.rest.render.errors.VndErrorJsonRenderer bean in 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-Type: application/json" -X POST -d "" http://localhost:8080/myapp/books

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Type: application/vnd.error+json;charset=ISO-8859-1

[

{

"logref": ""book.nullable"",

"message": "Property [title] of class [class **rest**.test.Book] cannot be **null**",

"\_links": {

"resource": {

"href": "http://localhost:8080/**rest**-test/books"

}

}

}

]

**9.1.8 Customizing Binding of Resources**

The framework provides a sophisticated but simple mechanism for binding REST requests to domain objects and command objects. One way to take advantage of this is to bind the request property in a controller the properties of a domain class. Given the following XML as the body of the request, the createBook action will create a new Book and assign "The Stand" to the title property and "Stephen King" to the authorName property.

<?xml version="1.0" encoding="UTF-8"?>

<book>

<title>The Stand</title>

<authorName>Stephen King</authorName>

</book>

class BookController {

def createBook() {

def book = **new** Book()

book.properties = request

// …

}

}

If the root element of the XML document contains an id attribute, the id value will be used to retrieve the corresponding persistent instance from the database and then the rest of the document will be bound to the instance. If no corresponding record is found in the database, the command object reference will be null.

<?xml version="1.0" encoding="UTF-8"?>

<book>

<title>The Stand</title>

<authorName>Stephen King</authorName>

</book>

Command objects will automatically be bound with the body of the request:

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 id attribute, the id value will be used to retrieve the corresponding persistent instance from the database and then the rest of the document will be bound to the instance. If no corresponding record is 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](http://grails.org/doc/3.0.x/api/org/grails/databinding/DataBindingSource.html) interface created by an instance of the [DataBindingSourceCreator](http://grails.org/doc/3.0.x/api/org/grails/databinding/bindingsource/DataBindingSourceCreator.html) interface. The specific implementation ofDataBindingSourceCreator will be selected based on the contentType of the request. Several implementations are provided to handle common content types. The default implementations will be fine for most use cases. The following table lists the content types which are supported by the core framework and which DataBindingSourceCreator implementations are used for each. All of the implementation classes 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 types, write a class which implements DataBindingSourceCreator and register an instance of that class in the Spring application context. If you are replacing one of the existing helpers, use the corresponding bean name from above. If you are providing a helper for a content type other than those accounted for by the core framework, the bean name may be anything that you like but you should take care not to conflict with one of the bean names above.

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 bindingTarget

\*

\* @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 bindingTarget, Object bindingSource)

}

[AbstractRequestBodyDataBindingSourceCreator](http://grails.org/doc/3.0.x/api/org/grails/databinding/bindingsource/AbstractRequestbodyDataBindingSourceCreator.html) is an abstract class designed to be extended to simplify writing custom DataBindingSourceCreator classes. Classes which extendAbstractRequestbodyDatabindingSourceCreator need to implement a method named createBindingSource which accepts an InputStream as an argument and returns a DataBindingSource as well as implementing the getMimeTypes method described in the DataBindingSourceCreator interface above. The InputStream argument to 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.AbstractRequestBodyDataBindingSourceCreator

/\*\*

\* 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** AbstractRequestBodyDataBindingSourceCreator {

@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 context.

// 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 DataBindingSourceCreator is needed to deal with a request which has a contentType of "text/custom+demo+csv".

**9.2 SOAP**

Grails does not feature SOAP support out-of-the-box, but there are several plugins that can help for both producing SOAP servers and calling SOAP web services.

**SOAP Clients**

To call SOAP web services there are generally 2 approaches taken, one is to use a tool to generate client stubs, the other is to manually construct the SOAP calls. The former can be easier to use, but the latter provides more flexibility / control.

The [CXF client plugin](http://grails.org/plugin/cxf-client) uses the CXF framework, which includes a wsdl2java tool for generating a client. There is nothing Groovy/Grails specific here in the generated code as it simply provides a Java API which you can invoke to call SOAP web services.

See the documentation on the [CXF client plugin](https://github.com/ctoestreich/cxf-client) for further information.

Alternatively, if you prefer more control over your SOAP calls the [WS-Lite library](https://github.com/jwagenleitner/groovy-wslite) is an excellent choice and features a [Grails plugin](http://grails.org/plugin/wslite). You have more control over the SOAP requests sent, and since Groovy has fantastic support for building and parsing XML it can be very productive approach.

Below is an example of a SOAP call with wslite:

withSoap(serviceURL: 'http://www.holidaywebservice.com/Holidays/US/Dates/USHolidayDates.asmx') {

def response = send {

body {

GetMothersDay(xmlns: 'http://www.27seconds.com/Holidays/US/Dates/') {

year(2011)

}

}

}

println response.GetMothersDayResponse.GetMothersDayResult.text()

}

It is not recommended that you use the [GroovyWS](http://groovy.codehaus.org/GroovyWS) library, it pulls in many dependencies which increases the likelihood of conflicts. The WSlite library provides a far simpler and easier to use solution.

**SOAP Servers**

Again, Grails does not have direct support for exposing SOAP web services, however if you wish to expose a SOAP service from your application then the [CXF plugin](http://grails.org/plugin/cxf) (not to be confused with the cxf-client plugin), provides an easy way to do so.

Typically it involves taking a Grails service and adding 'expose'-style configuration, such as the below:

**static** expose = EndpointType.JAX\_WS\_WSDL

//your path (preferred) or url to wsdl

**static** wsdl = 'org/grails/cxf/test/soap/CustomerService.wsdl'

Please refer to the [documentation of the plugin](https://github.com/thorstadt/grails-cxf#soap) for more information.

**9.3 RSS and Atom**

No direct support is provided for RSS or Atom within Grails. You could construct RSS or ATOM feeds with the [render](file:///H:\pilot\tools\spring\grails-docs-3.0.1\ref\Controllers\render.html) method's XML capability. There is however a [Feeds plugin](http://grails.org/plugin/feeds) available for Grails that provides a RSS and Atom builder using the popular [ROME](https://rome.dev.java.net/) library. An example of its usage can be seen below:

def feed() {

render(feedType: "rss", feedVersion: "2.0") {

title = "My test feed"

link = "http://your.test.server/yourController/feed"

**for** (article in Article.list()) {

entry(article.title) {

link = "http://your.test.server/article/${article.id}"

article.content // **return** the content

}

}

}

}

# 10 Asynchronous Programming - Reference Documentation

**Authors:** Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, Lari Hotari

**Version:** 3.0.1

## Table of Contents

[**10.1**Promises](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\async.html#promises)

[**10.2**Events](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\async.html#events)

[**10.2.1**Consuming Events](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\async.html#consuming)

[**10.2.2**Event Notification](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\async.html#notifying)

[**10.3**Asynchronous GORM](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\async.html#asyncGorm)

[**10.4**Asynchronous Request Handling](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\async.html#asyncRequests)

[**10.5**Servlet 3.0 Async](file:///H:\pilot\tools\spring\grails-docs-3.0.1\guide\async.html#asyncServletApi)

# 10 Asynchronous Programming

With modern hardware featuring multiple cores, many programming languages have been adding asynchronous, parallel programming APIs, Groovy being no exception.

The excellent [GPars](http://gpars.codehaus.org) project features a whole range of different APIs for asynchronous programming techniques including actors, promises, STM and data flow concurrency.

Added Grails 2.3, the Async features of Grails aim to simplify concurrent programming within the framework and include the concept of Promises and a unified event model.

## 10.1 Promises

A Promise is a concept being embraced by many concurrency frameworks. They are similar to java.util.concurrent.Future instances, but include a more user friendly exception handling model, useful features like chaining and the ability to attach listeners.

### Promise Basics

In Grails the grails.async.Promises class provides the entry point to the Promise API:

**import** **static** grails.async.Promises.\*

To create promises you can use the task method, which returns an instance of the grails.async.Promise interface:

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 complete and returns the results.

If you prefer not to block the current thread you can use the onComplete method:

onComplete([p1,p2,p3]) { List results ->

assert [4,16,64] == results

}

The waitAll method will throw an exception if an error occurs executing one of the promises. The originating exception will be thrown. The onComplete method, however, will simply not execute the passed closure if an exception occurs. You can register an onError listener if you wish to handle exceptions without blocking:

onError([p1,p2,p3]) { Throwable t ->

println "An error occured ${t.message}"

}

If you have just a single long running promise then the grails.async.Promise interface provides a similar API on the promise itself. For example:

**import** **static** java.util.concurrent.TimeUnit.\*

**import** **static** grails.async.Promises.\*

Promise p = task {

// Long running task

}

p.onError { Throwable err ->

println "An error occured ${err.message}"

}

p.onComplete { result ->

println "Promise returned $result"

}

// block until result is called

def result = p.get()

// block **for** the specified time

def result = p.get(1,MINUTES)

### Promise Chaining

It is possible to chain several promises and wait for the chain to complete using the then method:

**final** polish = { … }

**final** transform = { … }

**final** save = { … }

**final** notify = { … }

Promise promise = task {

// long running task

}

promise.then polish then transform then save then {

// notify end result

}

If an exception occurs at any point in the chain it will be propagated back to the caller and the next step in the chain will not be called.

### Promise Lists and Maps

Grails' async API also features the concept of a promise lists and maps. These are represented by the grails.async.PromiseList and grails.async.PromiseMap classes respectively.

The easiest way to create a promise list or map is via the tasks method of the Promises class:

**import** **static** grails.async.Promises.\*

def promiseList = tasks([{ 2 \* 2 }, { 4 \* 4}, { 8 \* 8 }])

assert [4,16,64] == promiseList.get()

The tasks method, when passed a list of closures, returns a PromiseList. You can also construct a PromiseList manually:

**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 a java.util.List from the get() method

Working with PromiseMap instances is largely similar. Again you can either use the tasks method:

**import** **static** grails.async.Promises.\*

def promiseList = tasks one:{ 2 \* 2 },

two:{ 4 \* 4},

three:{ 8 \* 8 }

assert [one:4,two:16,three:64] == promiseList.get()

Or construct a PromiseMap manually:

**import** grails.async.\*

def map = **new** PromiseMap()

map['one'] = { 2 \* 2 }

map['two'] = { 4 \* 4 }

map['three'] = { 8 \* 8 }

map.onComplete { Map results ->

assert [one:4,two:16,three:64] == results

}

### Promise Factories

The Promises class uses a grails.async.PromiseFactory instance to create Promise instances.

The default implementation uses the GPars concurrency library and is called org.grails.async.factory.gpars.GparsPromiseFactory, 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 asynchronously during unit tests, as this makes tests harder to write. For this purpose Grails ships with aorg.grails.async.factory.SynchronousPromiseFactory instance that makes it easier to test promises:

**import** org.grails.async.factory.\*

**import** grails.async.\*

Promises.promiseFactory = **new** SynchronousPromiseFactory()

Using the PromiseFactory mechanism is theoretically possible to plug in other concurrency libraries into the Grails framework.

### DelegateAsync Transformation

It is quite common to require both synchronous and asynchronous versions of the same API. Developing both can result in a maintenance problem as typically the asynchronous API would simply delegate to the synchronous version.

The DelegateAsync transformation is designed to mitigate this problem by transforming any synchronous API into an asynchronous one.

For example, consider the following service:

class BookService {

List<Book> findBooks(String title) {

// implementation

}

}

The findBooks method executes synchronously in the same thread as the caller. To make an asynchronous version of this API you can define another class as follows:

**import** grails.async.\*

class AsyncBookService {

@DelegateAsync BookService bookService

}

The DelegateAsync transformation will automatically add a new method that looks like the following to the AsyncBookService class:

Promise<List<Book>> findBooks(String title) {

Promises.task {

bookService.findBooks(title)

}

}

As you see the transform adds equivalent methods that return a Promise and execute asynchronously.

The AsyncBookService can then be injected into other controllers and services and used as follows:

AsyncBookService asyncBookService

def findBooks(String title) {

asyncBookService.findBooks(title)

.onComplete { List results ->

println "Books = ${results}"

}

}

## 10.2 Events

Grails 3.0 introduces a new Events API based on [Reactor](https://github.com/reactor/reactor).

All services and controllers in Grails 3.0 implement the [Events](http://grails.org/doc/3.0.x/api/grails/events/Events.html) 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 Reactor within application.yml, for example:

reactor

dispatchers:

**default**: myExecutor

myExecutor:

type: threadPoolExecutor

size: 5

backlog: 2048

## 10.2.1 Consuming Events

There are several ways to consume an event. As mentioned previously services and controllers implement the [Events](http://grails.org/doc/3.0.x/api/grails/events/Events.html) trait.

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 to implement the Events trait and ensure the class is registered as a Spring bean.

## 10.2.2 Event Notification

The Events trait also provides methods for notifying of events. For example:

notify "myEvent", "myData"

sendAndReceive "myEvent", "myData", {

println "Got response!"

}

## 10.3 Asynchronous GORM

Since Grails 2.3, GORM features an asynchronous programming model that works across all supported datastores (Hibernate, MongoDB etc.).

### Async Namespace

The Asynchronous GORM API is available on every domain class via the async namespace.

For example, the following code listing reads 3 objects from the database asynchronously:

**import** **static** grails.async.Promises.\*

def p1 = Person.async.get(1L)

def p2 = Person.async.get(2L)

def p3 = Person.async.get(3L)

def results = waitAll(p1, p2, p3)

Using the async namespace, all the regular GORM methods are available (even dynamic finders), but instead of executing synchronously, the query is run in the background and a Promise instance is returned.

The following code listing shows a few common examples of GORM queries executed asynchronously:

**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 not concurrency safe, a new session is bound per thread.

This is an important consideration when using GORM async (particularly with Hibernate as the persistence engine). The objects returned from asynchronous queries will be detached entities.

This means you cannot save objects returned from asynchronous queries without first merging them back into session. For example the following will not work:

def promise = Person.async.findByFirstName("Homer")

def person = promise.get()

person.firstName = "Bart"

person.save()

Instead you need to merge the object with the session bound to the calling thread. The above code needs to be written as:

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 would result in the change being lost. In general it is not recommended to read and write objects in different threads and you should avoid this technique unless absolutely necessary.

Finally, another issue with detached objects is that association lazy loading **will not** work and you will encounter LazyInitializationException errors if you do so. If you plan to access the associated objects of those returned from asynchronous queries you should use eager queries (which is 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 merging tends to be inefficient.

However, if you wish to do more complex GORM work asynchronously then the GORM async namespace provides a task method that makes this possible. For example:

def promise = Person.async.task {

withTransaction {

def person = findByFirstName("Homer")

person.firstName = "Bart"

person.save(flush:**true**)

}

}

Person updatedPerson = promise.get()

Note that the GORM task method differs from the static Promises.task method in that it deals with binding a new session to the asynchronous thread for you. If you do not use the GORM version and do asynchronous work with GORM then you need to do this manually. Example:

**import** **static** grails.async.Promises.\*

def promise = task {

Person.withNewSession {

// your logic here

}

}

### Async DetachedCriteria

The DetachedCriteria class also supports the async namespace. For example you can do the following:

DetachedCriteria query = Person.where {

lastName == "Simpson"

}

def promise = query.async.list()

## 10.4 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 responses asynchronously.

In general for controller actions that execute quickly there is little benefit in handling requests asynchronously. However, for long running controller actions it is extremely beneficial.

The reason being that with an asynchronous / non-blocking response, the one thread == one request == one response relationship is broken. The container can keep a client response open and active, and at the same time return the thread back to the container to deal with another request, improving scalability.

For example, if you have 70 available container threads and an action takes a minute to complete, if the actions are not executed in a non-blocking fashion the likelihood of all 70 threads being occupied and the container not being able to respond is quite high and you should consider asynchronous request processing.

Since Grails 2.3, Grails features a simplified API for creating asynchronous responses built on the Promise mechanism discussed previously.

The implementation is based on Servlet 3.0 async so to enable the async features you need to set your servlet target version to 3.0 in BuildConfig.groovy:

grails.servlet.version = "3.0"

#### Async Models

A typical activity in a Grails controller is to produce a model (a map of key/value pairs) that can be rendered by a view.

If the model takes a while to produce then the server could arrive at a blocking state, impacting scalability. You tell Grails to build the model asynchronously by returning a grails.async.PromiseMapvia 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 view. The equivalent synchronous action of the above is:

def index() {

def otherValue = …

[ books: Book.list() ,

totalBooks: Book.count(),

otherValue: otherValue ]

}

You can even render different view by passing the PromiseMap to the model attribute of the render method:

**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=${ticker}&f=nsl1op&e=.csv")

Double price = url.text.split(',')[-1] as Double

render "ticker: $ticker, price: $price"

}

}

}

The above example using Yahoo Finance to query stock prices, executing asynchronously and only rendering the response once the result has been obtained. This is done by returning a Promiseinstance from the controller action.

If the Yahoo URL is unresponsive the original request thread will not be blocked and the container will not become unresponsive.

## 10.5 Servlet 3.0 Async

In addition to the higher level async features discussed earlier in the section, you can access the raw Servlet 3.0 asynchronous API from a Grails application.

#### Servlet 3.0 Asynchronous Rendering

You can render content (templates, binary data etc.) in an asynchronous manner by calling the startAsync method which returns an instance of the Servlet 3.0 AsyncContext. Once you have a reference to the AsyncContext you can use Grails' regular render method to render content:

def index() {

def ctx = startAsync()

ctx.start {

**new** Book(title:"The Stand").save()

render template:"books", model:[books:Book.list()]

ctx.complete()

}

}

Note that you must call the complete() method to terminate the connection.

#### Resuming an Async Request

You resume processing of an async request (for example to delegate to view rendering) by using the dispatch method of the AsyncContext class:

def index() {

def ctx = startAsync()

ctx.start {

// **do** working

…

// render view

ctx.dispatch()

}

}