DZone Refcardz

Get More Refcardz! Visit refcardz.com

Getting Started with Grails

By Dave Klein

GETTING STARTED WITH GRAILS

Grails is a full-stack web application framework built on top of such tried and true open source frameworks as Spring, Hibernate, Ant, JUnit and more. By applying principles such as Convention over Configuration and Don't Repeat Yourself, and taking advantage of the dynamic Groovy programming language, Grails makes it incredibly easy to use these powerful tools. Grails doesn't reinvent the wheel; Grails makes a wheel that inflates itself and rolls where you want it to!

In case you are new to Grails, we'll start with a brief introduction, which should be enough to get you hooked and turn you into a Grails developer. That's when this Refcard will come in handy; it is a cheat sheet for Grails developers, a quick source for those things you keep having to go back to the docs to look up. Controllers, Services and Views with a detailed GSP taglib reference

Installing Grails

Download the Grails archive from http://grails.org/download and extract it to a local directory. Set a GRAILS_HOME environment variable to that directory and add GRAILS_HOME/bin to your path. (You also need a valid JAVA_HOME environment variable.) Now you're ready to go!

A Web App in the Blink of an Eye

To create a new Grails application, type:

```
$ grails create-app AutoMart
```

Now change to the AutoMart directory and create a domain class:

```
$ grails create-domain-class Car
```

Open AutoMart/grails-app/domain/Car.groovy and edit it, like so:

```
class Car {
    String make
    String model
    Integer year
}
```

Save this file, and run:

```
$ grails generate-all Car
```

Create-app and create-domain-class are Grails scripts. To see what other scripts are provided by Grails, run grails help from the command line.

You now have a complete working web application, with pages for creating, displaying, editing and listing Car instances. You can launch it with:

```
$ grails run-app
```

Grails runs on port 8080 by default. You can easily run on a different port like this:

```
$ grails -Dserver.port=9090 run-app
```

Figure 1: Create Car

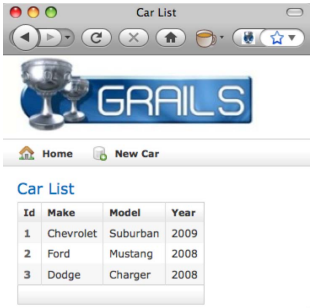


Figure 2: Car List

Navigate to http://localhost:8080/AutoMart and look around. Figure 1 and Figure 2 show a couple of the views Grails gives us "out of the box."

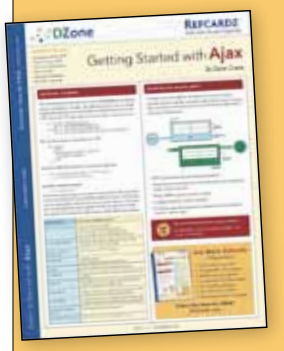
You can also leave it running while you continue to develop. Just save your changes and refresh your browser. This rapid feedback is one of the strengths of Grails.

Grails Conventions

In our example the Car.groovy file that Grails created for us was placed in a directory called grails-app/domain. This is one of the many conventions in Grails. Placing source files in certain directories and naming them in certain ways can make magical things happen in a Grails application.

Domain Classes

Placing a Groovy class file in the grails-app/domain directory will turn it into a persistent domain class. Several properties and methods will be added to the class dynamically, and Grails will create a table based on the name of the class, with fields for each property.



Get More Refcardz
(They're free!)

- Authoritative content
- Designed for developers
- Written by top experts
- Latest tools & technologies
- Hot tips & examples
- Bonus content online
- New issue every 1-2 weeks

Subscribe Now for FREE!
Refcardz.com

Note

Be careful about adding methods beginning with 'get' to a domain class. Grails will consider that a property and try to persist it. You can avoid this by not giving the method a return type (ie. `def getSomething(){...}`)

Controllers

Grails controllers are simple Groovy classes with names ending in 'Controller' and residing in the grails-app/controllers directory. Controllers also receive several methods and properties dynamically. Any closure defined as a property in a Controller will become an action reachable by a URL in the following form: application/controller/action.

Views

For each controller in your application there will be a directory under grails-app/views/ named after the controller class (ie. grails-app/views/car). This directory is where your views (.gsp files) go. When a controller action is completed it will automatically attempt to render a view with the name of the action. So, when you call http://localhost:8080/AutoMart/car/list the list action will execute and render the list.gsp page. You can, of course render specific pages but convention can be a huge time-saver.

Services

A Groovy class with a name ending in 'Service' and residing in the grails-app/services directory becomes a Grails service and has built-in transaction handling and more.

TagLibs

Creating custom tags in Grails is so easy it should be illegal. Just create a Groovy class ending with 'TagLib' and place it in the grails-app/TagLib directory. Then define a closure property and write to the OutputStream called 'out' that is already given to you. You can also use the tag's attributes and body by simply declaring them.

```
class YourTagLib{
def saySomething = {attrs, body ->
    if (attrs.tone == 'loud')
        out << body().toUpperCase()
    elseif (attrs.tone == 'quiet')
        out << body().toLowerCase()
    else
        out <<  body()
}

You can use this tag in a .gsp like this:

<g:saySomething tone="loud">
    I'm shouting now!
</g:saySomething>
```

You can use this tag in a .gsp like this:

```
<g:saySomething tone="loud">
    I'm shouting now!
</g:saySomething>
```

I'M SHOUTING NOW!

There are no extra classes to create, no interfaces to implement, no TLDs to create. You could probably take up a new hobby with the time you'll save!

TESTING

Unit Tests

Grails encourages unit testing by automatically creating stubbed out unit tests when creating artifacts (domain classes, controllers, services, and taglibs), and by including a powerful testing framework based on JUnit.

In Grails 1.1 (and earlier via a plugin), there are several classes inheriting from JUnit's TestCase class. Figure 1 shows these classes.

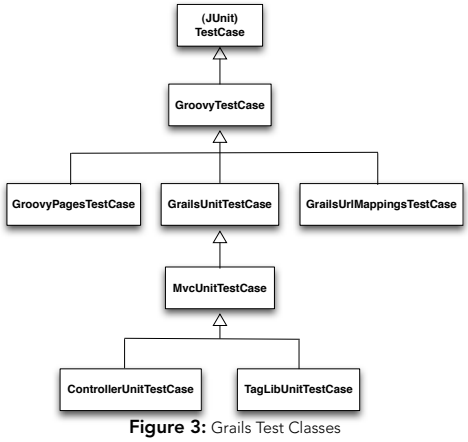


Figure 3: Grails Test Classes

Here are some of the methods that are available to us in the Grails testing framework.

mockDomain(class, list)- Mocks the domain class and stores instances in the list. Provides the dynamic GORM methods to class.
mockFor(class, loose) – Returns a mock of the class. loose determines whether mock has loose expectations or strict.
mockForConstraintsTest(class, list) – Similar to mockDomain but works for domains or command objects and adds a validate method to test constraints.
mockController(controllerClass) – Mocks a controller, adding the usual dynamic properties and methods.
mockTagLib(tagLibClass) – Mocks a TagLib, adding the usual dynamic properties and methods.
mockLogging(class, enableDebug) – Adds a mock logger to the class.

DOMAIN CLASS MAGIC

Dynamic methods

Grails adds several methods to our domain classes at runtime. Here are some of the most commonly used domain class methods:

Method	Description
`get(id)`	Retrieves an instance by id
`getAll([id,id,id...])`	Retrieves multiple instances by ids
`list()`	Retrieves all instances
`listOrderBy*()`	Retrieves list of instances sorted by expression
`findBy*()`	Returns first instance matching expression
`findAllBy*()`	Returns all instances matching expression
`count()`	Returns total number of instances
`countBy*()`	Returns count of instances matching expression
`save()`	Attempts to persist instance
`validate()`	Validates an instance based on constraints
`delete()`	Attempts to permanently remove instance
`withTransaction`	Executes a closure within a transaction
`withCriteria`	Executes a hibernate criteria using the CriteriaBuilder
`hasErrors`	Returns true if instance has validation errors

Methods such as countBy*() and findBy*() are synthesized methods, made up of the root method name and a combination of domain class properties and the following comparators:

- Equal (Implied default comparator)
- NotEqual
- LessThan
- LessThanEquals
- GreaterThan
- GreaterThanEquals
- Like
- like (case insensitive Like)
- InList
- Between
- IsNull
- IsNotNull

Up to two properties with optional comparators can be combined with a logical operator for example:

```
Car.countByMakeAndAgeLessThan('Dodge', 4)
Car.findAllByMakeAndModelInList('Ford', ['Mustang', 'Explorer'])
```

Pagination and Sorting Parameters

The domain method list() takes four parameters which are used for sorting and pagination. These same parameters can be passed in a Map as the last parameter to the findAllBy* methods.

Parameter	Description
`max`	The maximum number of instances to return
`offset`	The position to begin retrieving from
`sort`	Domain property to sort by
`order`	Whether to sort asceding or decending

Example: `Car.list(max:10, sort:'year', order:'desc')`

Constraint Validation

Either the save() or validate() methods shown above will trigger Grails' powerful data validation. Grails provides 17 built-in constraints plus the validation constraint for custom validation.

Many of these constraints also influence database schema generation; those are in shown in bold.

blank	(true/false) allow an empty string value
nullable	(true/false) allow nulls
max	Maximum value of any type that implements java.lang.Comparable
min	Minimum value of any type that implements java.lang.Comparable
size	Uses a range to determine the upper and lower limits of a collection or a String
maxSize	The maximum size of a collection or String
minSize	The minimum size of a collection or String
range	Uses a range to determine the limits of a numeric value
scale	Rounds value to specified number of decimal places - Does not generate an Error
notEqual	No comment
inList	Value must be contained in supplied list
Matches	Value must match supplied regular expression
Unique	(true/false) Verifies uniqueness in database
url	(true/false) Must be valid URL
email	(true/false) Must be valid email address
creditCard	(true/false) Must be valid credit card number
password	(true/false) Must be valid password
Validator	Takes a closure for custom validation. First parameter is value, second (if supplied) is the instance being validated

Here's an example of how constraints are declared in a domain class:

```
static constraints = {
    make()
    model()
    year()
    desription()
}
```

If validation fails during a call to save(), no exception is thrown. The failure is quietly recorded and stored in the instances errors property. To see if a domain class instance has validation errors, use the `hasErrors()` method in conjuction with the `errors` property, like so:

```
if (carInsance.hasErrors()){
    carInstance.errors.allErrors.each{ println it }
}
```

Relationships

GORM also makes relationships between different domain classes easier. Here we'll show how to implement the basic domain relationships.

One-to-one

Figure 4: One-to-one

Unidirectional one-to-one is the simplest type of relationship. One class has a Reference to another. This is declared by a property of the type of another domain class:

```
class Car {
    Engine engine
}
```

A **bi-directional one-to-one** is the same thing but with each class in the relationship having a reference to the other.

```
class Driver {
    Car car
}
class Car {
    Driver driver
}
```

Usually in a situation like this you want to show ownership and have cascading updates based on that ownership. With Grails that just takes single line of code:

```
class Car{
    Driver driver
    static belongsTo = Driver
}
```

Now a Car belongs to a Driver and when the Driver is saved the Car will be saved too. If the Driver is deleted then the Car goes with him. Makes sense. `belongsTo` also works with uni-directional relationships.

One-to-many

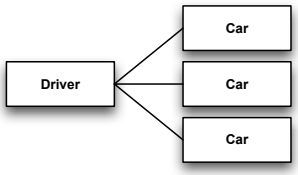


Figure 5: One-to-many

To declare a **uni-directional one-to-many** relationship just include a static **`hasMany`** property in the owning class.

```
class Driver {
    static hasMany = [cars : Car]
}
```

Now a Driver will have a Collection of Car instances called cars.

To make this **bi-directional** add a belongsTo property to the many class.

```
class Car{
    Driver driver
    static belongsTo = Driver
}
```

Many-to-many

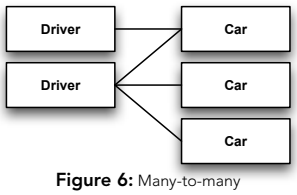


Figure 6: Many-to-many

A many-to-many relationship can be declared by adding a hasMany to both classes involved and a belongsTo property to one of them.

```
class Driver {
    static hasMany = [cars : Car]
}
class Car {
    static hasMany = [drivers : Driver]
    static belongsTo = Driver
}
```

THE THREE R'S OF CONTROLLERS

There are three possible conclusions to a controller action in Grails. You can `return` map of data or nothing and Grails will attempt to display the .gsp view with the same name as the action. You can also redirect to another url or action. The third way is to render something. This can be a view, a template, JSON, XML or just about anything that can be written to the response.

Redirect Method Parameters	
controller	Controller to redirect to. If action is not present the default action of the controller will be used.
action	Action to redirect to. If controller is not present current controller will be used.
id	Id to be passed, as params.id, to the redirect.
url	URL to redirect to ('http://grails.org').
uri	URI to redirect to ('/car/edit/1').
params	A map of parameters to be passed to the redirect.
Render Method Parameters	
text	Text to rendered to the resonse
view	A GSP view. Can include path info or the view folder for the current controller will be assumed
template	A template (partial GSP view). Often used with AJAX actions.
model	A map containing data to be used by the view or template.
bean	A single bean to used as the model for the view or template.
collection	A collection of objects to be used as the model for the view or template.
builder	Builder object to be used to render markup.
contentType	Sets the content type of the response.
encoding	Sets the encoding of the response

SERVICES

Grails service classes are powerful and easy to use. Here's a few important things to remember when using services.

Transactions

Grails service classes are transactional by default. If you don't want that behavior, for example, if you are going to handle transactions with the dynamic domain class method withTransaction, you can turn it off with a single line:

```
class CarService {
    static transactional = false
}
```

Service Injection

Services can be injected into controllers, domain classes, or other services simply by declaring a property with the same name as the service class type but with the first letter lowercase:

```
class CarController{
    def carService
}
```

You can also inject Services into plain old groovy objects, but it takes a couple more lines of code. First declare the service in your POGO.

```
class MyPogo {
    def carService
}
```

Then when you create an instance of your POGO do this:

```
import org.codehaus.groovy.grails.commons.ApplicationHolder
def myPogo = new MyPogo()
def ctx = ApplicationHolder.application.mainContext
ctx.autowireCapableBeanFactory.autowireBeanProperties(
      myPogo,
      AutowireCapableBeanFactory.AUTOWIRE_BY_NAME,
      false)
```

This will tell Spring to inject any Spring beans that are declared in the MyPogo class into the myPogo instance. Since Grails service classes are Spring beans, the carService will be injected.

Services and Scope

Grails services are singletons by default. To change that you just need to declare a static scope variable.

```
static scope =  'request'
```

Here are the possible values for scope:

Scope	Description
`request`	A new instance is created for each request
`session`	One instance is created for an entire session
`flash`	An instance is created that will exist for this request and the next request
`prototype`	A new instance is created for each objec that it's injected into
`flow`	instance exists for the life of a flow (webflow only)
`conversation`	Instance exists for the life of a flow and all sub-flows (webflow only)

GSP

Along with the ability to easily create custom GSP tags, Grails provides over 50 built-in tags. So many that it can be easy to miss one that could be just what you need. The following table shows some of the tags you don't want to miss:

Logical Tags

if- Conditionaly render GSP portions

Attribute	Description
`test*`	Expression to evaluate
`env*`	Name of a Grails environment

```
<g:if env="development" test="${car.year > 2008}">
     <p>This car is new.</p>
</g:if>
```

Else- The logical else tag

No Attributes

```
<g:if test="car.make == 'Honda'">
...
</g:if>
<g:else>
     <p>Sir, this car is not a Honda.</p>
</g:else>
```

Looping Tags

each- iterate over each element of the specified object

Attribute	Description
in	The object to iterate over
status	Variable to stor the iteration index in
var	The name of the item

```
<g:each var="car" in="${cars}">
     <p>Make: ${car.make}</p>
     <p>Year: ${car.year}</p>
</g:each>
```

findAll- condtionally iterate over objects in a collection

Attribute	Description
in	The collection to iterate over
expr	A Gpath expression

```
<h1>2003 Vehicles</h1>
<ul>
<g:findAll in="${cars}" expr="it.year == '2003'">
     <li>${it.make} ${it.model}</li>
</g:findAll>
</ul>
```

Form Tags

form- HTML Form with an action attribute based on controller/action/id

Attribute	Description
action*	Action to use in the Form Action
controller*	Controller to use in the Form Action
id*	ID to use in the Form Action
url*	A map containing the action/controller/id

```
<g:form name="myForm" action="show" id="1">
  ...
</g:form>

<g:form name="myOtherForm" url="[action:'list',controller:'car']">
  ...
</g:form>
```

**if not specified, the current controller/action will be used*

datePicker- Creates HTML selects for day/month/year/hour/second

Attribute	Description
name	Name of the date picker field set
value	Current value of the date picker
default	Default date. if "=none", the default is blank
Precision	Date granularity: year, month, day, hour, minute
noSelection	Map detailing the key and value to use for the "no selection made" choice in the selected box
Years	List/range of displayed years, in specific order

```
<g:datePicker name="myDate" value="${new Date()}"
noSelection="['':'-Choose-']"/>

<g:datePicker name="myDate" value="${new Date()}" precision="day"
years="${1930..1970}"/>
```

checkBox- Creates an HTML checkbox form field

Attribute	Description
name	Name of the checkBox
value	Expression; if evaluates to true, "checked = true"

```
<g:checkBox name="Used" value="${true}" />
```

radio- Creates an HTML radio button

Attribute	Description
value	value represented by radio button. Displayed as label
name	Name of radio button
checked	Boolean true/false for checked status

```
<g:radio name="myGroup" value="2" />
```

Select- Creates an HTML select

from	a list or range to select from
value	the current value of the property
optionKey	property of the bean to use as the key
optionValue	property of the bean to use as the value
noSelection	single-entry Map with a default key to return and a default value to display.
valueMessage-Prefix	Will be prepended to the option value with a '.' to create a key to lookup the value in the i18n message bundle.

```
<g:select name="car.year" from="${1903..2009}" value="${year}"
noSelection="['':'-Choose car year-']"/>
```

hiddenField- Creates an HTML input of type 'hidden'

name	the name of the input field
value	the value of the text field

```
<g:hiddenField name="year" value="${carInstance.year}"/>
```

actionSubmit- Creates a submit button with the specified value

name	Required; the title of the button. if 'action' is not specified, this will be the default action.
action	the action to execute

```
<g:actionSubmit value="Button Label" action="Update"/>
```

Link Tags

link- Creates an HTML anchor tag based on parameters

action	name of action to link to - if not specified, the default action will be used
controller	name of the controller to link to – if not specified, the current controller will be used
id	the id to use in the link
params	a Map containing request parameters
url	a Map containing the controller, action, etc. to use in the link

```
<g:link controller="car" action="list">List of Cars</g:link>
```

createLink- Creates a URL which can be used in anchor tags, etc.

action	the action to be used in the link- if not specified the default action will be used
controller	the controller to be used in the link – if not specified the current controller will be used
id	the id to be used in the link
url	a Map containing the controller, action, id, etc.

```
<g:createLink controller="car" action="show" id="1"/>
will create
AutoMart/car/show/1

As a method call in a GSP:
<a href="${g.createLink(controller:car, action:show, id:1)}">View
Car #1</a>
will create
<a href="/AutoMart/car/show/1">View Car #1</a>
```

createLinkTo- Creates a link to a static resource

`dir`	the directory in the application to link to
`file`	the name of the file in the application to link to

```
<g:createLinkTo dir="css" file="main.css" />
creates a link to
/shop/css/main.css
```

Ajax Tags

Note

In order for the Ajax tags to work properly, you need to include the javascript tag, specifying the library to use, in the <head> section of your page. For example:

```
<head>
    <javascript library="yui" />
</head>
```

javascript- For inclusion of JavaScript libraries and scripts; also a shorthand for inline JavaScript

`library`	the library to include
`src`	the name of the JavaScript file to import - will look for a file in /web-app/js/
`base`	the full URL to prepend to the library name

```
<g:javascript src="thisisascript.js" />
will import
/web-app/js/thisisascript.js

<g:javascript library="scriptaculous" />
will import necessary JavaScript for the Scriptaculous library

Inline JavaScript:
<g:javascript>alert('hello')</g:javascript>
```

ABOUT THE AUTHOR

Dave Klein is a developer with Contegix, a company specializing in delivering managed internet infrastructure based upon Linux, Mac OS X, JEE, and Grails. Dave has worked as a developer, architect, project manager, mentor, and trainer for the past 15 years, and has presented at user groups and national conferences. Dave's Groovy and Grails-related thoughts can be found at http://dave-klein.blogspot.com.

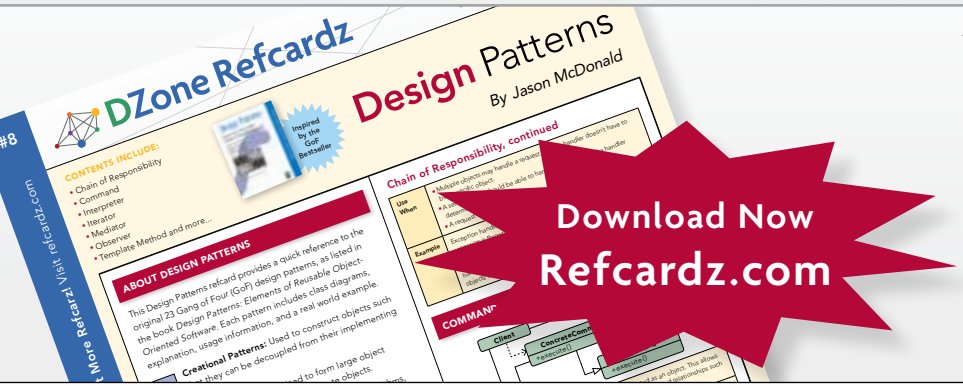
RECOMMENDED CLASS

In *Grails: A Quick-Start Guide*, you'll see how to use Grails by iteratively building an unique, working application. By the time we're done, you'll have built and deployed a real, functioning website.

BUY NOW

books.dzone.com/books/grails-quick-start

Professional Cheat Sheets You Can Trust



"Exactly what busy developers need: simple, short, and to the point."

James Ward, Adobe Systems

Upcoming Titles

RichFaces
Agile Software Development
BIRT
JSF 2.0
Adobe AIR
BPM&BPMN
Flex 3 Components

Most Popular

Spring Configuration
jQuery Selectors
Windows Powershell
Dependency Injection with EJB 3
Netbeans IDE JavaEditor
Getting Started with Eclipse
Very First Steps in Flex

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-53-0
ISBN-10: 1-934238-53-8
50795
9 781934 238530
$7.95

Copyright © 2009 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher. Reference: