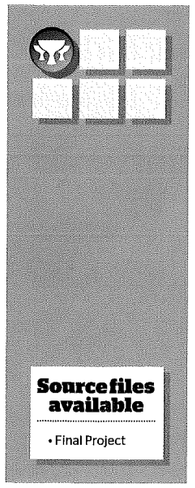
Build web apps with Groovy & Grails, part 1

Build a simple Java web application powered by Groovy and the Grails Model View Controller framework

tools | tech | trends Grails, Groovy/Grails Tool Suite



Grails is an open-source framework that uses the MVC (Model View Controller) approach to create dynamic webapps on a Java platform using Groovy. It's easy to learn and offers many tools and features to assist you. If you are new to Groovy or Grails, there's no need to be afraid. After following this guide you will be well on your way to creating powerful, dynamic Java webapps with minimal fuss.

In this tutorial we will ease into the world of Grails by building a very simple application that will allow users to add, edit, and delete entries in a task list. We will look at creating the domain classes, controllers and view pages; as well as using constraints to enforce data validation on our database model objects. We'll also use Grails' powerful tools to generate the database schema and a portion of the application for us, which helps to streamline how we work and achieve results much faster than with other languages.

01 Install Grails IDE

You could code our complete Grails app using a command line interface and any text editor. However, dedicated tools like the Groovy/Grails Tool Suite IDE, based on the Eclipse framework, offers built-in Grails helper tools as well as server management and debugging. Download the open-source application and install. It will also install the latest version of Grails for you. **www.springsource.org/downloads/sts-ggts**.

02 Create new application

Select File>New>Grails Project from the main menu in the IDE. This will open up the new project wizard. Enter a memorable name for your application. In this example we'll use TaskManager as the name. Accept all defaults you are shown and click Finish to proceed. The project structure will be created for you.

03 Amend config

Open up conf/Config.groovy in the IDE. This is one of a number of configuration files. Here we will change the value of the first property, named grails.project.groupId. By default this will use the project name that already exists, but we'll define a custom package name for use with our objects, domain packages and controllers.

```
001 grails.project.groupId = "org.example.taskmanager"
```

04 Run application

Start up the server and make sure we can reach the Grails implementation. Right-click the project and select Run As>Grails Command (run-app). By default, the application will run in the development environment. Once complete, visit the URL shown in the console panel view to see the default Grails landing page.

```
001 http://localhost:8080/TaskManager/
```

05 Create Controller

Let's get cracking and create our first controller. Right-click on the project in the left-hand pane within the IDE. Select New>Controller from the context menu. Ensure the application name listed is correct, and enter 'Home' as the controller name. This will generate the controller for you with a default action called index.

```
001 package org.example.taskmanager
002 class HomeController {
003
004    def index() { }
005 }
```

06 Create index page

Grails uses naming conventions to define the transactions and processes around the MVC framework. With the default index action, we now need to create a view page with the same name. Right-click the views/home directory and select New>File from the menu. Call the file 'index.gsp' and click Finish to complete the process.

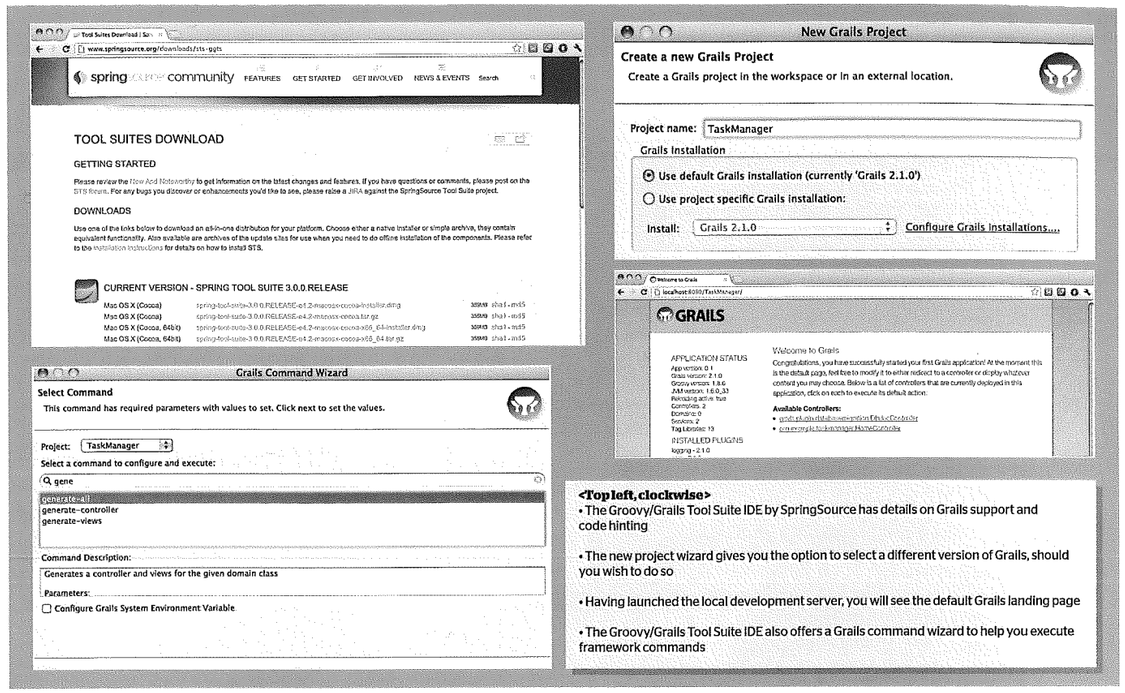
07 Passing data

We'll send some basic data from the controller for display in the view. Open HomeController.groovy, and within the index action define two variables. The first is a String, while the second is a Date object. Send the value of these variables as a mapped object with key/value pairs.

```
001 def index() {
002
003 def String header = 'Welcome to the Grails introduction app'
004 def Date date = new Date()
005
006 ["header": header, "date": date]
007
008 }
```

08 Output variables

Open home/index.gsp, into which we'll add the code to display the variables sent from the controller. Grails variables are wrapped in curly brackets and are prepended with a dollar sign. To format the date object into a string, we'll use one of the many tags included in the Grails library, g:formatDate, and define the format for the date.



<Top left, clockwise>

- The Groovy/Grails Tool Suite IDE by SpringSource has details on Grails support and code hinting
- The new project wizard gives you the option to select a different version of Grails, should you wish to do so
- Having launched the local development server, you will see the default Grails landing page
- The Groovy/Grails Tool Suite IDE also offers a Grails command wizard to help you execute framework commands

```
001 <h1>${header}</h1>
002
003 <p>The current date is <g:formatDate format="dd MMMM, yyyy"
date="${date}"/></p>
```

09 Create domain/model

Grails uses the Model View Controller paradigm. Domain classes are the models in this scenario, and they represent a persistent entity that is mapped onto an underlying database table. Right-click the project and select New>Domain Class from the context menu. Set the name of the new model as 'Task' and click Finish to generate the skeleton object.

```
001 package org.example.taskmanager
002
003 class Task {

004
005    static constraints = {
006    }
007 }
```

10 Set properties

As the domain class we created in the last step represents a database table, we now need to define the columns within that table, which we can do by setting the property names and data types within the class itself. We don't have to worry about the creation of the database tables, as Grails will handle those for us. We'll create two String variables and one Date property object for our tasks.

```
001 import java.util.Date;
002
003 class Task {
004
005 String title
006 String description
007 Date dueDate
```

11 Constraints

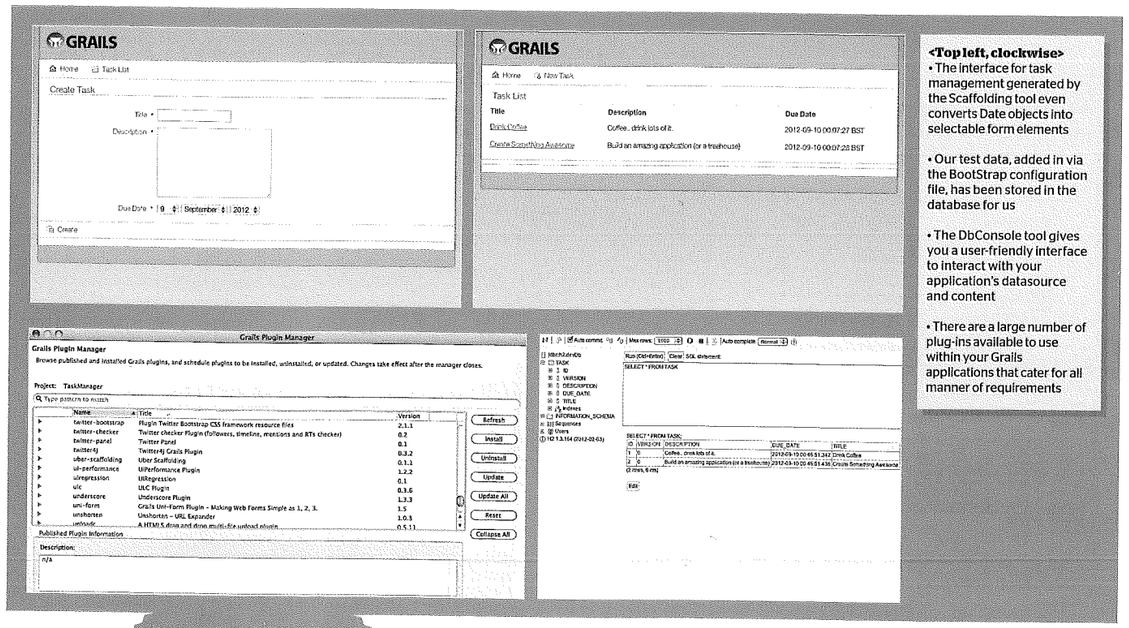
With the properties defined, we can add constraints to the domain class to help validate the values being passed into the class and stored in the database, as well as to help Grails define the correct column properties when generating the database schema. Here we want to ensure both string values are provided, and that the title will have a minimum of three characters.

```
001 static constraints = {
002 title size: 3..100, blank: false
003 description maxSize: 1000, blank: false
004 }
```

12 Generate all

Grails can really enhance the development process with some fantastic built-in tools. We'll use a feature known as Scaffolding to automatically create the controller, associated unit tests and the view pages that will interact with our Task domain class. To do so, right-click and select Grails Tools>Open Grails Command Prompt and enter the following command.

```
001 grails> generate-all org.example.taskmanager.Task
```



<Top left, clockwise>

- The interface for task management generated by the Scaffolding tool even converts Date objects into selectable form elements
- Our test data, added in via the BootStrap configuration file, has been stored in the database for us
- The DbConsole tool gives you a user-friendly interface to interact with your application's datasource and content
- There are a large number of plug-ins available to use within your Grails applications that cater for all manner of requirements

Use Scaffolding

The Scaffolding tool is a great way to create features to manage the full database interaction. The generated code also acts as a good resource on how to create pages manually.

13 Generated pages

Following the generate-all command, visit to the root of your application in the browser (localhost:8080/TaskManager) and click through to the TaskController. You will see that Grails has generated all of the views required to list, add, edit and delete items from the database. It has done a lot of work for you in seconds.

14 Homeward bound

We can change routes and URL mappings in Grails to point towards specific controllers and actions. We want to view the home/index.gsp page whenever we view the root URL instead of the default Grails page. To do so, open up config/UrlMappings.groovy and change the root mapping to point to the home controller and the index action.

```
001 class UrlMappings {
002
003 static mappings = {
004   "/$controller/$action?/$id?"{
005     constraints {
006       // apply constraints here
007     }
008   }
009
010   "/"(controller: "home", action: "index")
011   "500"(view:'/error')
012 }
013 }
```

15 Test data

To save entering test data every time we run the application, we can set default data that will automatically populate the database whenever we start the server. Open conf/BootStrap.groovy and set the values for the default records for testing purposes within the init() block. We first check to make sure the test data does not already exist using the count() method.

```
001 def init = { servletContext ->
002 // Check whether the test data already exists.
003 if (!Task.count()) {
004   new Task(title: "Drink Coffee", description: "Coffee..
drink lots of it.", dueDate: new Date()).save(failOnError: true)
005   new Task(title: "Create Something Awesome", description:
"Build an amazing application (or a treehouse)", dueDate: new
Date()).save(failOnError: true)
006 }
007 }
```

16 Home controller

We want to be able to view a list of all Task entities from the homepage. While we could set another URL Mapping to point the root of the application to the Task controller index action, we'll instead revise controllers/HomeController.groovy and set the relevant variables to read from the database and pass them through to the view.


```
001 def index(Integer max) {
002
003 params.max = Math.min(max ?: 10, 100)
004
005 def String header = 'Welcome to the Grails introduction app'
006 def Date date = new Date()
007
008 [taskInstanceList: Task.list(params), taskInstanceTotal: Task.
count(), "header": header, "date": date]
009
010 }
```

17 Loop over data

Open views/home/index.gsp. Beneath the existing code we'll add a conditional statement to check for the existence of records returned to us. If we have data, we'll use another built-in tag, g:each, to loop over the collection and create a list item for each. We'll also define a link to allow the user to edit the task and generate pagination links. The code for this step can be found on the resource disc.

18 Create link

From the homepage, we also want to give our users a link to allow them to add new tasks to the database. We'll use the g:link tag once more to generate the full HTML anchor tag. This will send them to the create action in the task controller. Place this at the bottom of the index.gsp file outside of the closing g:if tag.

```
001 <p>
002 <g:link controller="task" action="create">Create new task</
g:link>
003 </p>
```

19 Persistent database

By default, the Grails development environment uses a database that is persisted in memory and is recreated with every server restart. We can alter this behavior by changing a configuration property. Open conf/DataSource.groovy and change the development datasource url, removing the mem: string from the value and changing the dbCreate property to 'update'.

```
001 environments {
002   development {
003     dataSource {
004       dbCreate = "update" // one of 'create', 'create-
drop', 'update', 'validate', ''
005       url = "jdbc:h2:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000"
006     }
007   }
```

20 DbConsole tool

Grails provides you with an incredibly easy way to interact with the databases. Log in to the DbConsole tool, making sure the JDBC URL value matches that set in the DataSource.groovy configuration file. The console is available via the URL on the local server, and you can query, view, amend and manage the database content directly from here. localhost:8080/TaskManager/DbConsole.

```
001 initialize: function() {
002 this.model.on('remove', this.unrender, this);
003 // ...
004 },
005 // ...
006 unrender: function() {
007 this.$el.remove();
008 }
```

21 Change port

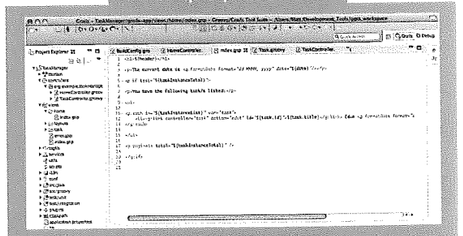
By default, the development environment server will run on port 8080. If for any reason you have something else running on that port or want to change it to something more memorable, you can set a permanent property in conf/BuildConfig.groovy, specifying the preferred port number to use. Add this line to the top of that configuration file. The code for this step can be found on the resource disc.

22 Highly extensible

Grails can be extended and enhanced using plug-ins. You can create your own, but have a look at the list of those available to use via the Grails Plugin Manager, accessible from the Grails Tools context menu. URL Shortening, Social Networks and framework tools are just some of those available to download and implement into your applications.

ORM and Hibernate

Grails has an Object Relational Mapping layer built on top of Hibernate, which makes the tasks of managing database access and persistent objects an incredibly simple process.



A powerful IDE to assist your workflow

An IDE can be an incredibly personal tool, as we spend a lot of time using it. We want it to do everything we need with minimal fuss and as quickly, cleanly and easily as possible. The more streamlined we can make our development process, the easier it makes building an application and the simpler our working lives can be.

The GGTS IDE provides us with a clean, simple layout packed full of features and functions to interact with the Grails framework at a command level. For anyone not fully comfortable with command line development, GGTS will handle this communication for you, as everything you need is available from a menu option. The tool also includes detailed (and very useful) step debugging tools to watch and debug the application as it is running.

Build web apps with Groovy & Grails, part 2

Add authentication and security roles to your Grails application using a helpful plug-in

tools | tech | trends Grails, Groovy/Grails Tool Suite

Source files available
- Starting Project
- Final Project

Thanks to the command-line interface and tooling, a good portion of creating web apps with Grails is automated. However, all URLs and controller actions are openly accessible to visitors. When developing systems that allow for content management, it's important to add security features to stop unwanted access.

In this tutorial we will use authentication rules and processes to add security restrictions to editable content using the Spring Security Core plug-in, which will help us to define user objects, assign roles and handle database input and encoding.

We will also create a new layout template to add a styled user interface to our default homepage, and use tag library features to read content from calling pages.

Thanks to Grails' infrastructure, we can keep our code modular and reusable as possible, which will assist in code maintenance and ease workflow development.

01 Install security plug-in

Let's start by installing the Spring Security Core plug-in into our existing application. Open the command prompt interface by right-clicking on the TaskManager project folder and selecting Grails Tools>Open Grails Command Prompt. Enter the following command to install the plug-in:

```
001 grails> install-plugin spring-security-core
```

02 Create classes

Following a successful installation of the plug-in, you will be notified in the command output to run a new command, which will generate the minimum required domain classes to use with the security authentication processes. Enter the following command into the prompt to create two classes, User and Role, which will be generated for you.

```
001 grails> s2-quickstart org.example.taskmanager User Role
```

03 Check config

The s2-quickstart command has created the domain classes, controllers and GSP view pages for you to handle the user interface and layout for the login form page. It has also added some extra parameters to the conf/Config.groovy file to configure the paths to the domain classes. Open up the file and at the bottom you should see code similar to the following:

Added by the Spring Security Core plug-in:

```
001 grails.plugins.springsecurity.userLookup.userDomainClassName =
'org.example.taskmanager.User'
002 grails.plugins.springsecurity.userLookup.
authorityJoinClassName = 'org.example.taskmanager.UserRole'
003 grails.plugins.springsecurity.authority.className = 'org.
example.taskmanager.Role'
```

04 Alter encryption

The Spring Security plug-in manages a lot of the daunting tasks of the authentication process for you, including the automatic encryption of user passwords before saving them into the database. By default this uses the SHA-256 algorithm, but we can change this to a number of alternatives. Add the following line at the bottom of the Config.groovy file to use bcrypt.

```
001 grails.plugins.springsecurity.password.algorithm = 'bcrypt'
```

05 BootStrap users

Open conf/BootStrap.groovy within the IDE. Here, we will set a default user with the correct administrative roles so that we can log into the application's restricted pages. First, we need to import the new domain classes we created using the Spring Security plug-in. Add the following imports to the top of the file before the opening class definition.

```
001 import org.example.taskmanager.Task
002 import org.example.taskmanager.User
003 import org.example.taskmanager.Role
004 import org.example.taskmanager.UserRole
```

06 Define Role

Now we can create our default roles – one role for an administrator, and the other role for a standard user. We will also create a dummy user account for testing purposes and grant that user with administrative rights by assigning the correct admin role to them. Place the following code within the init definition.

```
001 if (!User.count()) {
002  def adminRole = new Role(authority: 'ROLE_ADMIN').
save(failOnError: true)
003  def testUser = new User(username: 'coldfumonkeh',
enabled: true, password: 'mypassword')
004  testUser.save(failOnError: true)
005  UserRole.create testUser, adminRole, true
006 }
```

07 Annotation class

With the user defined, we can now start to apply the access restrictions. Open TaskController.groovy in the IDE. First, we need to import the required classes and references from the Spring Security framework. Add the following import definition before the opening class definition within the task controller file.

```
001 import grails.plugins.springsecurity.Secured
```

08 Secure everything

We want to restrict access to all of the views offered through the TaskController.groovy file. We can do this on a per-definition basis, setting the security restriction for each individual action, or by setting one rule to cover the entire controller, which we'll do in this case. Add the following code before the opening class definition.

```
001 @Secured(['ROLE_ADMIN'])
002 class TaskController {
```

09 Run application

Let's run the application to generate the new user access definitions and roles set in the Bootstrap.groovy file. Open the Grails Command Prompt window and enter the required command to run the application. Once complete, the new user details should be stored in the database for us.

```
001 grails> run-app
```

10 Check database

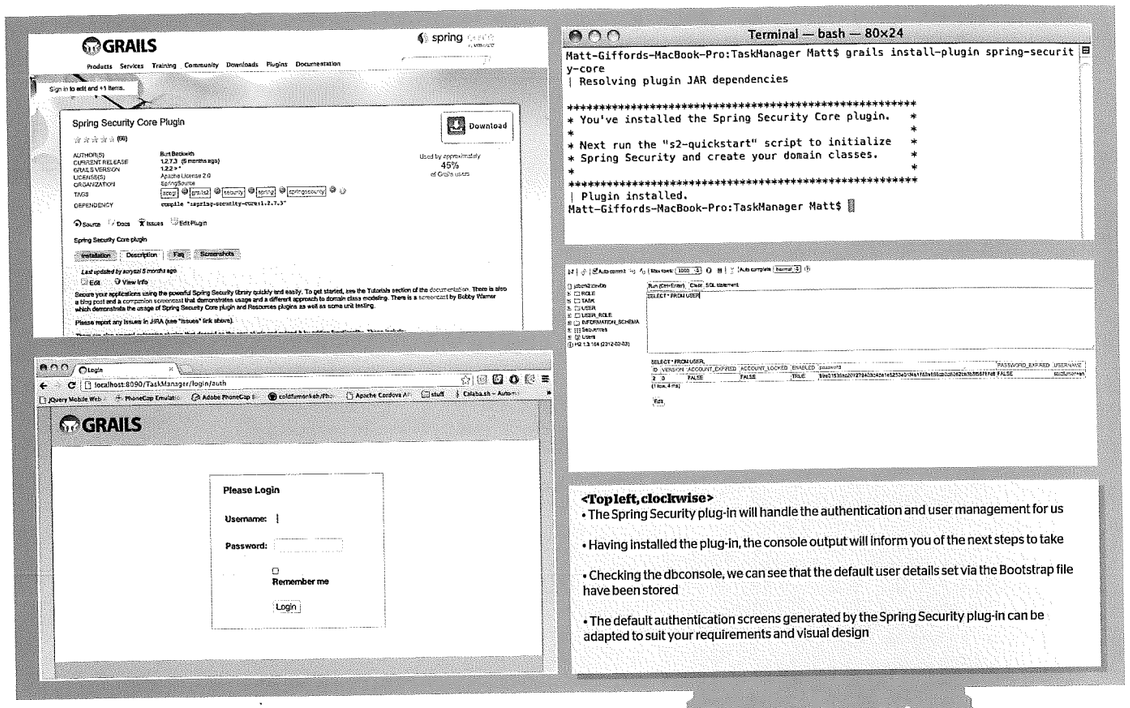
Open up the dbconsole tool in the browser by visiting **localhost:8090/TaskManager/dbconsole** and login with the database security credentials as defined in the conf/DataSource.groovy file. Run a select statement against the USER, ROLE and USER_ROLES tables to ensure the data has been successfully written to them.

11 Attempting access

Browse to the root of the site (**localhost:8090:TaskManager**). This will display our home screen, complete with the task list. If we click on a task item link to edit its details, we should now be presented with a login screen, complete with remember me functionality. Very handily, the user interface for the login pages has also been created for us by the plug-in, negating the need for us to do so.

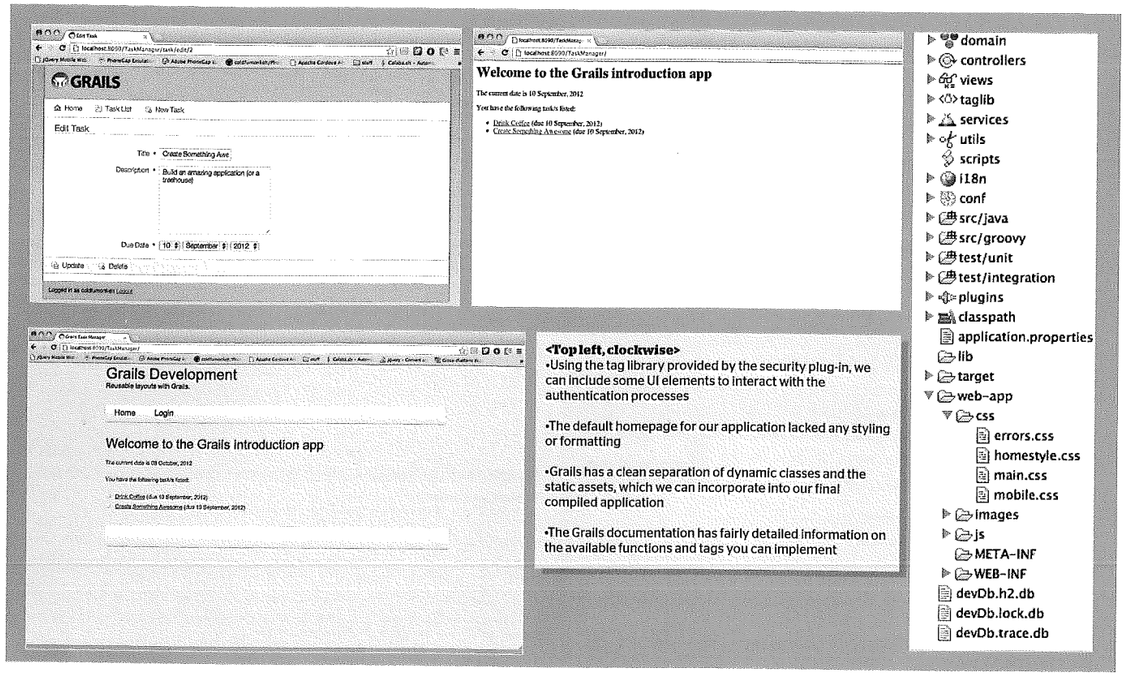
12 Logging out

At the moment we don't have a clear way of logging an authenticated user out of our application. The Spring Security plug-in provides us with a number of GSP tags that we can use within our views, which use the sec namespace. Open Views>Layouts>main.gsp and add the following within the footer div element.



<Top left, clockwise>
- The Spring Security plug-in will handle the authentication and user management for us
- Having installed the plug-in, the console output will inform you of the next steps to take
- Checking the dbconsole, we can see that the default user details set via the Bootstrap file have been stored
- The default authentication screens generated by the Spring Security plug-in can be adapted to suit your requirements and visual design

Documentation
There are a lot of features and functions in Grails to help create dynamic, scalable applications. The online documentation is a great resource to reference.




```
001 <sec:ifLoggedIn>
002  Logged in as <sec:username />
003  <g:link controller="logout" action="index">Logout</g:link>
004 </sec:ifLoggedIn>
```

13 Revised homepage

Open Views>Home>index.gsp. At the moment, this page does not contain any core HTML structure. To use this content with our forthcoming layout template, let's update this to include the following code. Place the existing content in this file between the body tags. By creating this definition, we will be able to reference elements for use within the layout template.

```
001 <!doctype html>
002 <html>
003 <head>
004    <title>Grails Task Manager</title>
005    <meta http-equiv="content-type" content="text/
html; charset=UTF-8" />
006 </head>
007 <body>
008
009 </body>
010 </html>
```

14 Reference assets

To apply a new stylesheet to our layout, we can call a dynamic reference to the homelayout.css file. Any static files, including images, are kept within the web app directory of the project. Add the following to the head of the view page. This will call the correct resource from the CSS directory.

```
001 <link rel="stylesheet" href="${resource(dir: 'css', file:
'homestyle.css')}" type="text/css">
```

15 Layout template

Let's create a new layout for use with our default home page. Create a new file called 'homelayout.gsp' within the Views>Layouts directory. This will hold the main HTML template to render our view page, with the exception of the actual body content, which will come from the view itself. Full code for this layout is included in the project files.

```
001 <div id="main">
002    <header>
003      <div id="logo">
004        <div id="logo_text">
005          <h1><a href="${createLink(uri: '/')}">Grails<span
class="logo_colour"> Development</span></a></h1>
006          <h2>Reusable layouts with Grails.</h2>
007        </div>
008      </div>
009      <nav>
010        <ul class="sf-menu" id="nav">
011
012        </ul>
013      </nav>
014    </header>
```

16 Navigation

We now need to add the navigation elements into the template to handle the display of the links to log in and log out for users. Once again, we'll make use of the Spring Security tag library to handle the display of the links based on the user's current authentication. Place this between the nav tag within the homelayout.gsp file.

```
001 <ul class="sf-menu" id="nav">
002 <li class="selected"><a href="${createLink(uri: '/')}">Home</
a></li>
003 <sec:ifLoggedIn>
004 <li><g:link controller="logout" action="index">Logout</
g:link></li>
005 </sec:ifLoggedIn>
006 <sec:ifNotLoggedIn>
007 <li><g:link controller="login" action="index">Login</g:link></
li>
008 </sec:ifNotLoggedIn>
009 </ul>
```

17 Layout title

When using reusable layouts, we want to be able to use as much of the existing information from the decorated view page as much as possible, which includes the title, if available. We can use the grails layoutTitle tag within the layout file to read this value, or use a default value if one is not present.

```
001 <title><g:layoutTitle default="Task Manager"/></title>
```

18 Layout head

We set the reference to the CSS file for the new layout within the view page. This means we can use individual stylesheets for each page, but we still need the layout to call these into play. For this, we can use yet another Grails tag called layoutHead, which will use any content located within the calling page's head tag.

```
001 <g:layoutHead />
```

19 Getting body content

With the layout template more or less complete, we now need to define the body content to display from the decorated page. Once more, Grails makes this easy with the use of another tag called layoutBody. Place this within the content div block in the layout file. This will display all content from between the calling page's body tags.

```
001 <div id="site_content">
002     <div class="content">
003       <g:layoutBody/>
004     </div>
005 </div>
```

20 Include JavaScript

As we reference and include stylesheets and other assets dynamically, we can do the same when including JavaScript libraries or files into our application. Before the closing body tag in our homelayout.gsp file, add the following code to load the jQuery library directly into the application.

```
001 <!-- javascript at the bottom for fast page loading -->
002 <g:javascript library="jquery" />
003 <r:layoutResources />
```

21 Apply layout

With the layout complete, we now need to ensure our view page uses the new template for rendering. There are a few ways to achieve this, but we'll use the applyLayout tag. Wrap the content of the view page with this tag, specifying the name of the template to use. All calls to this page will now use the new template.

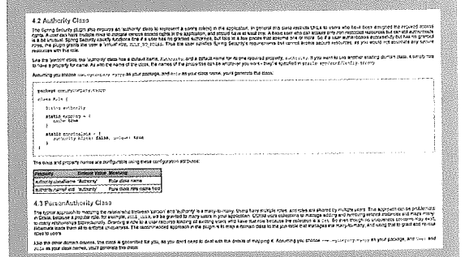
```
001 <g:applyLayout name="homelayout">
002
003 </g:applyLayout>
```

22 Run app

Let's run the application to ensure the new template is called for the home page view. Open the Grails Command Prompt window and enter run-app to compile the files and start the development server. Once complete, navigate to the URL where you should see the revised layout, including links to log in as an authenticated user.

SOCIAL SECURITY

There are a number of Grails plug-ins built on top of the Spring Security Core authentication access plug-in that give you the ability to authenticate a user using a number of popular social networking applications.



Defining security access, roles and restrictions within your application

Running the s2-quickstart command, we asked it to create two domain classes for us to represent and manage the User and Role features respectively. The first relates to a new user object whereas the second deals with the specific security authority. The plug-in created a third domain class for us, UserRole, which handles the many-to-many relationship mapping between these two individual classes and stores the relationship into the database for us.

One user can have many roles (or authorities). Similarly, a specific authority can have many users with the same rights and privileges. Although the default properties with the classes deal with login information only, you can also extend the class with your own to enhance the User object, and provide more detailed information such as first name, last name and email address, for example.

For more details on the plug-in, check out the official documentation available here: **monkeh.me/2kli7**