Grails in Action
Second Edition

Glen Smith
Peter Ledbrook

Foreword by Dierk König

Manning
**brief contents**

**PART 1  INTRODUCING GRAILS**

1. Grails in a hurry  ...  3
2. The Groovy essentials  33

**PART 2  CORE GRAILS**

3. Modeling the domain  63
4. Creating the initial UI  91
5. Retrieving the data you need  110
6. Controlling application flow  136
7. Services and data binding  155
8. Developing tasty forms, views, and layouts  189

**PART 3  EVERYDAY GRAILS**

9. Building reliable applications  229
10. Using plugins: just add water  261
11. Protecting your application  297
12. Exposing your app to other programs  328
13  ■  Single-page web applications (and other UI stuff)  357
14  ■  Understanding Spring and transactions  380

PART 4  ADVANCED GRAILS ..........................................................399

15  ■  Understanding events, messaging, and scheduling  401
16  ■  NoSQL and Grails  432
17  ■  Beyond compile, test, run  467
18  ■  Grails in the cloud  496
And so we come to the last chapter of the book. We hope you enjoyed the journey and learned plenty of useful things about developing with Grails. To finish off, we dive further into the world of plugins because they’re such a fundamental part of what makes Grails a productive framework.

You’ve seen plenty of examples of using plugins, and we covered that aspect thoroughly in chapter 10. In chapter 17, we briefly introduced you to the modularization of application development through plugins. Now it’s time to make you a Grails expert by showing you how they work and what you can do in your own plugins. Even if you never write your own plugin, you can contribute to existing public plugins, which helps you, your company, and all the other users of those plugins you contribute to!

Given that plugins are so important to Grails, when might you need or want to create your own? We don’t want to limit your thinking on this, but here are three of the most common uses:

**This chapter covers**
- Understanding how plugins work
- Writing your own plugins
- Managing plugin repositories
- Integrating an existing library or tool into Grails
- Providing a specific feature
- Modularizing an application

You’ve already seen examples of the first use, such as Mail (which integrates Java Mail and makes it easier to use), Spring Security, and JMS. These are typically feature-based plugins as well (email, access control, messaging), but you can also have feature plugins that are independent of existing libraries, such as the Resources and Navigation plugins.

The last use case is something that we introduced in chapter 17 and relates to breaking an application into modules. Remember, this doesn’t make sense for a small application such as Hubbub, but large applications can easily become difficult to maintain and understand if they remain monolithic.

Most of this chapter is based on the example of writing a security plugin for your Hubbub application. Let’s say this right from the outset: it’s a stupid idea to write your own security plugin if you don’t have a strong background in security. You’ll inevitably end up with something that has security flaws that leave applications vulnerable. Don’t do it. This will be a throwaway plugin that acts as a useful demonstration of how plugins work. For real applications always use a standard security plugin, such as Spring Security.

### 20.1 Creating the plugin

In chapter 17, you created a plugin that was a module of an application. In this case, you create a feature plugin. The first steps are the same, so you create the plugin project and then add artifacts (domain classes, and so on) that you need.

#### 20.1.1 Are you sure it’s not an application?

Let’s create the skeleton of your new plugin and then take a look at how it differs from an application:

```
grails create-plugin graina-security
```

You can see the directory structure it creates in figure 20.1. It’s no coincidence that the project looks like a Grails application, and you can run it like one using the regular `run-app` command (if you explicitly add either the Tomcat or Jetty plugins to the project’s BuildConfig.groovy). This can be useful for testing.

The most visible difference between a plugin and an application is the presence of a plugin descriptor, the GrainaSecurityGrailsPlugin.groovy file (highlighted in figure 20.1). The descriptor not only contains information about the plugin, such as the author and a description of what it does, but it also contains the code that allows you to hook into Grails and modify the behavior of applications at runtime.

The name of the plugin descriptor class is important, because it determines the official plugin name. This is different from an application, where the name is specified in the application.properties file. Grails works out the name of the plugin by chopping off the GrailsPlugin suffix, converting uppercase letters to lowercase, and separating words with a hyphen. For example, GrainaSecurityGrailsPlugin becomes graina-security.
Now let's take a look at the descriptor you created. The following listing contains a condensed version of its initial contents.

```groovy
class GrainaSecurityGrailsPlugin {
    def version = "0.1"
    def grailsVersion = "2.3 > *"

    def pluginExcludes = [
        "grails-app/views/error.gsp"
    ]

    def title = "Plugin summary/headline"
    def author = "Your name"
    def authorEmail = ""

    def description = '''
    Brief description of the plugin.
    '''
    def documentation = "http://grails.org/plugin/graina-security"
    def license = "APACHE"
    def issueManagement = [
        system: "GitHub",
        url: "https://github.com/GrailsInAction/graina2"
    ]
    def scm = [url: "https://github.com/GrailsInAction/graina2"
    ]

    def doWithSpring = { ... }
    def doWithApplicationContext = { ctx -> ... }
    def doWithWebDescriptor = { xml -> ... }
    def doWithDynamicMethods = { ctx -> ... }
    def onChange = { event -> ... }
    def onConfigChange = { event -> ... }
    def onShutdown = { event -> ... }
}
```

Figure 20.1 The directory structure of a plugin project, with the plugin descriptor highlighted. Note that it's almost identical to the structure of a Grails application.
Most of this is plugin metadata, such as the plugin’s version, documentation URL, license, and so on. The version property should stick to the standard major.minor .patch numbering convention, where each part is an integer and the final patch number is optional. For example, both 1.0.4 and 2.1 follow the convention. You can also append a -SNAPSHOT suffix to indicate that it’s a development version of the plugin.

The grailsVersion field indicates which versions of Grails the plugin works with. This can either be a single version (unusual) or a range of the form

\[ \text{lowerBound} > \text{upperBound} \]

You should read this as “any version from lowerBound up to upperBound, inclusive.” You can also use the wildcard (*) for either bound. The following denotes “any version up to and including 1.1”:

\[ * \geq 1.1 \]

This one denotes “any version from 1.0 up”:

\[ 1.0 \geq * \]

It’s admittedly an unusual syntax, but it doesn’t take long to get used to it. Note that there’s no equivalent range syntax using the less than symbol (<).

We won’t go into any more detail about the metadata right here because we come back to it later when we publish plugins. For now, you can see in figure 20.2 how descriptor properties in the listing match up with the plugin portal on grails.org.

After you get past the plugin metadata, things get more interesting. Those doWith* and on* closures (for example, doWithSpring, onChange) are the hooks that allow your plugin to influence the runtime behavior of the application. These are key to

![Image of Spring Security Core plugin](http://grails.org/plugin/spring-security-core)
providing smooth integration with Grails, such as that provided by GORM. Table 20.1 describes what each of the hooks allows you to do and lists the arguments Grails passes to the associated closure. The first three are shown in the order in which Grails invokes them on application startup.

### Table 20.1 The hooks that allow a plugin to integrate closely with a Grails application

<table>
<thead>
<tr>
<th>Property</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doWithSpring</td>
<td>none</td>
<td>This closure allows you to define and configure Spring beans.</td>
</tr>
<tr>
<td>doWithDynamicMethods</td>
<td>ctx</td>
<td>This closure allows you to add dynamic properties and methods to classes and artifacts. Its sole argument is the fully configured Spring application context.</td>
</tr>
<tr>
<td>doWithApplicationContext</td>
<td>ctx</td>
<td>This closure allows you to work with the fully configured Spring application context. You can manipulate only bean instances, not bean definitions.</td>
</tr>
<tr>
<td>doWithWebDescriptor</td>
<td>webXml</td>
<td>This closure allows you to modify the application's web descriptor. It's passed a GPathResult object (created by XmlSlurper) that you can use to insert elements, such as servlet and filter definitions.</td>
</tr>
<tr>
<td>onChange</td>
<td>event</td>
<td>This closure allows you to react to changes to any artifacts that the plugin is watching.</td>
</tr>
<tr>
<td>onConfigChange</td>
<td>event</td>
<td>This closure allows you to react to changes to the application's Config.groovy file.</td>
</tr>
<tr>
<td>onShutdown</td>
<td>event</td>
<td>This closure allows you to react when the application shuts down. The Spring application context is still available at this point.</td>
</tr>
</tbody>
</table>

Note that all the hooks are closure properties; if you use methods instead, they won’t be executed. We come back to these hooks in the next section because they enable more advanced integration with the Grails platform. First, you need to add the artifacts that the plugin needs to work.

### 20.1.2 Controllers, views, and other artifacts

The easiest way to enhance an application is to provide ready-made artifacts with your plugin, such as domain classes and controllers. Consider your security plugin: you want to assign roles to users and store that information in a persistent way, so domain classes may come in handy. Logging into an application is pretty standard stuff, so you should also include a login page. Finally, you could use tags that show parts of a web page only if the user has a particular role or is logged in. A Remove Post button for Hubbub, for example, should only be visible to administrators.
Any Grails artifact can be included in a plugin. Start with the domain classes your security plugin needs.

**Domain classes**

Because you want to assign users to roles, it makes sense that you need one domain class to represent your users and another one for the roles.

Your `Role` class is going to be simple, consisting of a `name` property that represents the role:

```groovy
package com.grailsinaction.security

class Role {
    String name
    static constraints = {
        name blank: false, unique: true
    }
    static mapping = {
        table "sec_role"
    }
}
```

You create this as you do with a Grails application. The main difference is that you need to consider potential conflicts with the application that the plugin is installed into. That’s why the class package is pretty much mandatory. Remember that table names must be unique, too, hence you specify one here with a prefix.

Perhaps as importantly, you need to be aware that the plugin’s domain classes can be used with any database, which means you can run afoul of reserved names. Unfortunately every database seems to have its own set of reserved names. It’s best to play it safe and avoid any table name that might match a SQL keyword. Examples of risky table (and column) names are `user`, `role`, and `password`. If in doubt, add a custom table (or column) name with a special prefix.

You also need a class to represent a user account, as shown in the following listing, which you’ll assign roles to.

**Listing 20.2 Assigning roles in a user account**

```groovy
package com.grailsinaction.security

class Account {
    String username
    String passwordHash
}
```
static hasMany = [ roles: Role ]
static constraints = {
    username blank: false, unique: true
    passwordHash blank: false, bindable: false
}
static mapping = {
    table "sec_account"
}

You should create a link table between Role and Account to make adding and removing roles much more efficient than with hasMany. This is what the Spring Security plugin does. For simplicity, the example sticks to a standard many-to-many GORM relationship.

Another thing to bear in mind when putting domain classes into a plugin is that they materially affect the domain model of any project that the plugin is installed into. The user has no way of disabling, modifying, or removing those domain classes without editing the plugin files directly.

In your case, the plugin absolutely requires the Account and Role domain classes. If users want to customize them in any way, they can do it through inheritance or associations. An alternative approach is to copy the behavior of the various create-* and generate-* commands and provide scripts to create the domain model. This is the approach favored by the Spring Security plugin with its s2-quickstart command.

**CONTROLLERS**

Your next step is to create a login page that integrates with the plugin’s access control infrastructure. All this page does is accept a username and password and check them against the values stored in the database. If the login is successful, you place the relevant Account instance in the session; otherwise you redirect back to the login page. You need a login view, an action to log users into the system, and an action for logging out. Because you support roles, let’s also include an action that redirects users if they don’t have permission to access a particular page—an access denied page. The following listing shows the outline of your LoginController.

### Listing 20.3 Setting up the LoginController

```groovy
package com.grailsinsaction.security
class LoginController {
    def accessControlService
    def index() {}
    def signIn = {
        if (!accessControlService.login(
            params.username, params.password)) {
            redirect action: "index"
        }
        else {
            redirect uri: "/"
        }
    }
}
```

Property should never take part in automatic data binding, so don’t let apps do it.

Use this injected service to do the work
Displays login page
Redirect to app’s home page after successful login
def signOut() {
    accessControlService.logout()
    redirect uri: "/"
}

def unauthorized() {}

One thing to be aware of is that method-based actions will only work with applications using Grails 2.0 and higher. If you want to target versions of Grails prior to 2.0, use closure properties for your actions. To be honest, there are few benefits to supporting version 1.3.x and earlier unless you have to support a legacy project. Grails 2.0 has been around for a long time now.

Other compatibility points to be aware of include:

- Extending the `RestfulController` that comes with Grails 2.3 obviously limits your plugin to only work in Grails 2.3 and higher applications.
- Referencing the `list` action of scaffolding controllers, such as in a redirect or HTML link, works for Grails 2.2 and earlier but not version 2.3.

After the plugin is installed in an application, the controller and views behave as if they’re part of the application. If this plugin were installed in Hubbub, you could point your browser at `http://localhost:8080/hubbub/login/index` and it would show the login page provided by the plugin. You don’t have to do any configuration whatsoever.

An important property of plugin controllers and views is that you can override them in the application. Let’s say you want to provide your own login page while still using the functionality of the plugin’s `LoginController`. You can place your own `index.gsp` file under the application’s `grails-app/views/login` directory. As its relative path and its name match those of the plugin’s view, it automatically overrides the plugin view. You could even change the login behavior completely by providing your own `LoginController` implementation.

Note that application controllers override those provided by a plugin as long as the name is the same. The packages can and should be different.

**TAG LIBRARIES**

Tag libraries work much like controllers and are common in the plugin world. We strongly recommend that any plugin tag libraries you provide use a namespace that’s likely to be unique for the plugin. For example, use a namespace of `graina` for your security plugin. There’s no guarantee that another plugin won’t use the same namespace, but a careful choice helps minimize the risk.

Why would a plugin provide tags? Custom tags provide an easy way to access a plugin’s functionality from GSP views. Your security plugin would benefit from allowing users to mark blocks of a page that should be visible only if a user is logged in (or not), so create a new taglib with this command:

```
grails create-tag-lib com.grailsinaction.security.AccessControl
```
and set its content to this

```java
package com.grailsinaction.security

class AccessControlTagLib {
    static namespace = "graina"

    def accessControlService

    def isLoggedIn = { attrs, body ->
        if (accessControlService.isAuthenticated()) {
            out << body()
        }
    }
    ...
}
```

The only other concern with tags is how codecs behave. These don’t impact the previous tag because you don’t render anything beyond the body (which is already escaped), but let’s say you have a tag that renders the full name of a user:

```html
<div>Welcome back <graina:fullName user="${currentUser}" /></div>
```

The full name of the current user should be explicitly escaped as HTML before it’s rendered to the page. The user could configure whether or not the output of tags is escaped, but only with Grails 2.3 and above.

If you want to automatically escape the output, though, you need to provide a mechanism for users to override the codec used. That’s where the `encodeAs` attribute comes in:

```html
<graina:fullName user="${currentUser}" encodeAs="none" />
```

This could be implemented as in the following listing.

```
package com.grailsinaction.security

class AccessControlTagLib {
    static namespace = "graina"

    def accessControlService

    /*
     * Prints out the full name of a given user.
     * @attr user REQUIRED The user whose full name should be displayed
     * @attr encodeAs The codec to use when encoding the name for
     * the output. Default is HTML. "none" disables the encoding.
     */
    def fullName = { attrs ->
        def codec = attrs.encodeAs ?: "HTML"
        if (codec.equalsIgnoreCase("none")) {
            out << attrs.user.fullName
        }
    }
```

**Listing 20.4 Overriding the codec**
else {
    out << attrs.user.fullName."encodeAs$codec"();
}
...

Listing 20.4 also demonstrates the javadoc \@attr tag, which provides IDEs with information about what attributes a tag accepts and which ones are required. This is useful for users, so you should add these javadoc tags to all your own custom tags.

**VIEWS**

Views in plugins work like those in applications. And as with controllers, you can also override plugin-provided views in the application. The only requirement is that the application’s views need to have the same path relative to the views directory as the plugin views.

The only compatibility issue is once again with codecs. Plugins’s views are affected by the global default codec settings, so you have to factor in that the default codec could be none or html. Table 20.2 shows you what happens pre-Grails 2.3 both with and without encodeAsHTML() in your GSP expressions, depending on what the default codec is.

<table>
<thead>
<tr>
<th>Default codec</th>
<th>With encodeAsHTML()</th>
<th>Without encodeAsHTML()</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Proper encoding</td>
<td>No encoding (unsafe)</td>
</tr>
<tr>
<td>html</td>
<td>Double encoding</td>
<td>Proper encoding</td>
</tr>
</tbody>
</table>

Grails 2.3 and above doesn’t exhibit the double encoding problem, so it’s always safe to use encodeAsHTML().

The solution to this little problem is to force the use of a specific codec for your view, making the view independent of the global setting. You add a page directive to the view, as shown in the following listing for the plugin’s login page, grails-app/views/login/index.gsp.

```gsp
<%@page defaultCodec="HTML" %>
<html>
<head>
    <title>Application Login</title>
</head>
<body>
    <h1>Log in to the application</h1>
    <g:form controller="login" action="signIn">
        <div>Username: <input type="text" name="username" value="${params.username}"/>
    </div>
</g:form>
</body>
</html>
```
Creating the plugin

It doesn’t matter what you set the default codec to for the page as long as the rest of the page is using the `encodeAsHTML()` method appropriately. Feel free to head back to chapter 11 to refresh your memory on how the codecs work.

**URL mappings**
When you provide controllers and views with your plugin, you need to consider how they’re mapped to URLs in the application. You can’t rely on the default URL mapping because users can (and probably should) remove it.

You have two options: provide default mappings in the plugin or leave it to the application developer to map the actions and views. To add default mappings, add a class to the plugin’s grails-app/conf directory that has a `UrlMappings` suffix. The standard `UrlMappings` class won’t be included in the packaged plugin, but a `GrainaSecurityUrlMappings` class will be. The only reason for the standard `UrlMappings.groovy` file is so you can run the plugin as an application via the `run-app` command. You’re free to delete it if you don’t use `run-app`.

You could create a new `GrainaSecurityUrlMappings.groovy` file in the plugin’s `grails-app/conf/com/grailsinaction/security` directory with content like this:

```groovy
class GrainaSecurityUrlMappings {
    static mappings = {
        "/login"(controller: "login", action: "index")
        "/auth"(controller: "login", action: "signIn")
        "/logout"(controller: "login", action: "signOut")
    }
}
```

Although convenient for getting users started with good URLs, this approach suffers from a significant problem: users can’t disable the mappings. That’s why we think it’s best at the moment not to offer default URL mappings but document suggested ones. If and when Grails allows users to disable mappings, then you should think about providing default mappings.

**Working with GORM**
The last set of considerations we cover here are the plugin’s interactions with GORM. Think about what an application can do:

- Use an alternative GORM implementation to Hibernate
- Globally change the default `failOnError` behavior of `save()`
- Disable the `OpenSessionInViewInterceptor` that guarantees a Hibernate session is open for the whole length of a request
Because these actions have a strong impact on plugins that work with persistent data, they require you to code defensively. If you want to ensure maximum compatibility with applications, then follow this advice:

- Do all data access in transactional services—this ensures that a Hibernate or other type of session is open during data access.
- Don’t ever rely on lazy loading of associations. Always fetch all the data you need within the service using eager fetching.
- Check whether a domain instance is valid before attempting to save it.
- Use dynamic finders, Where queries, or Criteria queries rather than HQL if possible so that the query works with as many types of data store as possible. But if you require a relational database, make that clear in the documentation.

Consider how you’d code your security plugin’s AccessControlService. The following listing adheres to our advice.

```groovy
package com.grailsinaction.security

class AccessControlService {
    static transactional = true
    static scope = "session"

    Long userId

    Account addNewUser(String username, String password) {
        def newUser = new Account(
            username: username,
            password: hashPassword(password))
        if (newUser.validate()) {
            newUser.save()
        }
        return newUser
    }

    void login(String username, String password) { ... }    
    void logout() { this.userId = null }   
    boolean isAuthenticated() { return userId != null } 
    String hashPassword(String password) { password.encodeAsMD5() } 
}
```

Also remember that your services may be invoked from the application as well as from your own plugin’s code, so you should clearly document the behavior of its methods. In the case of addNewUser(), you should make it clear that it always returns a User instance, but it may have errors on it. When that’s the case, the user isn’t persisted.

All the artifacts you need for the plugin are now in place. What you’re missing is a mechanism for controlling access to an application’s pages. That requires access to Grails’s various integration points.
20.2 Deepening the integration with Grails

Packaging and running a Grails application involves work under the hood that you don’t often do as an application developer. As a plugin developer, you need to become familiar with the basic processes if you want your plugins to integrate seamlessly with Grails.

In this section, you learn how to do the following:

- Configure your own Spring beans
- Handle reloading of classes at runtime
- Add entries to the application’s web descriptor
- Implement extra commands

The first three of these are done through the special plugin descriptor hooks (see table 20.1). You investigate these hooks as you add the access control behavior to the plugin.

20.2.1 Setting up Spring beans

To control access to an application’s functionality, there has to be a gatekeeper. Your security plugin fulfills that role, but how it does that depends on what you want to secure. It could be access to particular domain classes or service methods. Access control by URL is also popular. In this case, you create @RoleRequired and @AuthRequired annotations that can be added to controller actions to protect the corresponding pages. The application developer can then do the following:

```groovy
class UserController {
    @RoleRequired("Administrator")
    def delete() {
        ...
    }
    ...
}
```

The logged-in user must have an Administrator role to delete users.

These aren’t special Groovy AST transformation annotations like @TestFor, but plain old Java ones, so you have to iterate through all the controllers and scan for these annotations during application startup. Each time you find an annotation, you have to attach access control logic to the corresponding action.

We’ll concentrate on the @AuthRequired annotation to keep this section brief and leave it as an exercise for you to implement @RoleRequired. Alternatively, you can find the relevant code on GitHub.

You can write annotations in either Groovy or Java. There’s not much difference in the code itself. You’ll do yours in Java, so create the file AuthRequired.java in src/java/com/grailsinaction/security and put this content inside:

```java
package com.grailsinaction.security;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface AuthRequired {
}

By themselves, the annotations don’t do anything without access control logic. The best place for this logic is in a Grails filter, because that can intercept every request. The filter knows what the current controller and action names are, so all it need do is look up whether or not the corresponding action requires authentication. Where should the filter look up this information?

You obviously need to store the state somewhere. The simplest option is a Spring bean, as that can be injected into the filters class and you can also initialize it on application startup. You could use AccessControlService, but that’s currently scoped to the HTTP session. Instead, create a simple security data class, SecurityData, under src/groovy (make sure the file is in the appropriate directory structure for the given package):

```groovy
package com.grailsinaction.security

class SecurityData {
  Map authRequiredActions = [:]
}
```

and set this up as a bean in the doWithSpring hook. The syntax you use is identical to the Bean Builder syntax we showed you in chapter 14:

```groovy
import com.grailsinaction.security.SecurityData
class GrainsGallSecurityGrailsPlugin {
  
  def doWithSpring = {
    grainaSecurityData(SecurityData)
  }
}
```

In effect, the doWithSpring hook is the plugin’s equivalent of the application’s resources.groovy file. You can define as many beans as you like, reference existing ones, and wire them together.

One thing to note is that the Spring application context is a global namespace, so prefix your bean name with graina to help avoid name clashes. You should consider doing something similar in your own plugins. Also bear in mind that users can override the beans that you define in an application’s resources.groovy file—a handy feature!

Now that the bean is defined, you can create the security filters class that uses it. Run

```bash
grails create-filters com.grailsinaction.security.AccessControl
```

and add the content from the following listing to the generated class.

```groovy
package com.grailsinaction.security
class AccessControlFilters {

  def grainaSecurityData
  def accessControlService

  
  
Listing 20.7 Creating AccessControlFilters class
```
```groovy
def filters = {
  all(controller:'*', action:'*') {
    before = {
      def actions =
        grainaSecurityData.authRequiredActions[controllerName]
      if (actionName in actions &&
          !accessControlService.isAuthenticated()) {
        redirect controller: "login", action: "index"
      } else {
        return true
      }
    }
  }
}
```

This filters class controls access to the application based on the information in the `grainaSecurityData` bean. All you need do now is initialize that data bean with the controller actions that require authenticated users. To do that, use Java’s Reflection API to discover which actions are annotated with `@AuthRequired`. How do you determine what controllers are available in the application? Via Grails’s Artefact API.

### 20.2.2 Finding out about application artifacts

Think about what happens when a plugin is installed in a Grails application. When you write it, you know about only the classes provided by the plugin itself, but as soon as you install it into an application, the plugin suddenly finds itself in an environment containing more classes than it’s used to—classes that are provided by other installed plugins as well as the application. There must be a way to find out what’s around without resorting to querying the class loader, right? Thankfully, a way exists.

All of the hooks listed in table 20.1 make a property available called `application`. This is the Grails application instance, and you can use it to find out what artifacts are available. It’s exactly the same object as the `grailsApplication` bean. It provides a set of dynamic methods and properties that form the Artefact API. Say you want to get all the controllers in the application:

```groovy
application.controllerClasses
```

You can also retrieve all the services via `application.serviceClasses`. In fact, you can do something similar for all types of artifacts. Table 20.3 lists the available methods and properties.

<table>
<thead>
<tr>
<th>Method template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;artifact&gt;Classes</code></td>
<td>Returns all the artifact descriptors in the application of type <code>&lt;artifact&gt;</code>. Note that the artifact type should start with a lowercase letter because it’s a property.</td>
</tr>
</tbody>
</table>
Artifact descriptors are objects that provide extra information about their corresponding artifacts (there’s one descriptor per artifact class, for example, for AccessController, UserService, UserController, and so on). Table 20.4 lists the key properties that these descriptors provide.

### Table 20.4 Artifact descriptor properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>clazz</td>
<td>Class</td>
<td>The class of the artifact</td>
<td>com.grailsinaction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>.LoginController</td>
</tr>
<tr>
<td>metaClass</td>
<td>MetaClass</td>
<td>A shortcut to the artifact’s metaclass; same as</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clazz.metaClass</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>The name of the artifact, minus any suffix or</td>
<td>Login</td>
</tr>
<tr>
<td></td>
<td></td>
<td>package</td>
<td></td>
</tr>
<tr>
<td>shortName</td>
<td>String</td>
<td>The name of the artifact minus any package</td>
<td>LoginController</td>
</tr>
<tr>
<td>propertyName</td>
<td>String</td>
<td>The short name of the artifact in property form</td>
<td>loginController</td>
</tr>
<tr>
<td>logicalPropertyName</td>
<td>String</td>
<td>The property name without the artifact type</td>
<td>login</td>
</tr>
<tr>
<td></td>
<td></td>
<td>suffix</td>
<td></td>
</tr>
</tbody>
</table>

### Under the hood

These dynamic artifact methods are implemented by the DefaultGrailsApplication class and map to physical methods such as getArtefacts(). The artifact descriptors are instances of the GrailsClass interface in package org.codehaus.groovy.grails.commons, which you’ll become familiar with when you do more advanced plugin work.
Let’s get back to the security plugin and populate the security data bean based on the controllers in the application. You have a couple of hooks that give direct access to the bean: `doWithApplicationContext` and `doWithDynamicMethods`. Both work, but because you’re not adding any dynamic methods with this plugin, you should probably use the former, as shown in the following listing.

```groovy
package com.grailsinaction.security
import org.codehaus.groovy.grails.commons.GrailsClassUtils as GCU
...

class GrainaSecurityGrailsPlugin {
    ...
    def doWithApplicationContext = { ctx ->
        def securityData = ctx.getBean("grainaSecurityData")
        for (controllerClass in application.controllerClasses) {
            def controllerName = controllerClass.logicalPropertyName
            def actionMap = securityData.authRequiredActions
            if (actionMap[controllerName] == null) {
                actionMap[controllerName] = []
            }
            for (method in controllerClass.clazz.declaredMethods) {
                def ann = field.getAnnotation(AuthRequired)
                if (ann != null) {
                    def actionName = method.name
                    actionMap[controllerName] << actionName
                }
            }
        }
    }
    ...
}
```

The `doWithApplicationContext` is a great hook if you need to do something to beans after they’re initialized. If you want to define new beans or override existing ones, then use `doWithSpring`.

**NOTE** We’re skipping `doWithDynamicMethods` because we don’t recommend you create new dynamic methods unless you have a good reason. They tend to make debugging harder, and it’s difficult for developers to know where these methods come from. It’s often simpler and better to provide a service with the utility methods you want to provide.

You’re almost done adding access control to the application. In fact, it’s already working, so what more could there be? What you’re not taking into account right now is what happens when a user modifies a controller while the application is running. Perhaps they add an annotation to an action or remove one. At the moment those changes won’t be visible to the developer without restarting the server! You need to hook into the class reloading mechanism.
20.2.3 Dealing with class reloading

Grails’s ability to reload changes to application files on the fly is one of its greatest advantages on the productivity front. Unfortunately for plugin writers, users expect plugins to play well with this feature and take advantage of it themselves. This is where the onChange and onConfigChange hooks come in.

Grails maintains lists of what it calls watched resources, which are file path patterns similar to the ones you may have used with the Ant build tool. Whenever a file that matches one of the patterns is modified, Grails first reloads it (if it contains a Groovy class or script) and then notifies interested plugins by calling their onChange closures.

Obviously the first step for your plugin is to register an interest in such changes. You can take two approaches: observe an existing plugin or specify a list of watched resources.

Let’s say you want Grails to notify you when any controllers are modified. Because the core controllers plugin already watches for changes to controller classes, you can observe it and then your plugin is notified of those changes. In fact, your plugin receives exactly the same notifications as the observed plugin. Those may include changes that you don’t expect, so take care—the controllers plugin also watches tag libraries.

Observing a plugin is as simple as adding an observe property to the plugin descriptor, like so:

```groovy
def observe = ["controllers"]
```

Each entry in the list is the name of a plugin you want to observe. If a listed plugin isn’t installed in the current application, it’s quietly ignored. Table 20.5 shows the more useful plugins and the resources that they watch.

<table>
<thead>
<tr>
<th>Plugin name</th>
<th>Resources watched</th>
</tr>
</thead>
<tbody>
<tr>
<td>controllers</td>
<td>Controllers and tag libraries</td>
</tr>
<tr>
<td>filters</td>
<td>Filter classes</td>
</tr>
<tr>
<td>hibernate</td>
<td>Hibernate configuration files</td>
</tr>
<tr>
<td>i18n</td>
<td>i18n bundles (messages.properties, for example)</td>
</tr>
<tr>
<td>services</td>
<td>Services</td>
</tr>
<tr>
<td>urlMappings</td>
<td>URL mappings</td>
</tr>
</tbody>
</table>
The other approach is to specify a `watchedResources` property that contains one or more file patterns that you’re interested in. You could place a watch on controller classes with this:

```groovy
def watchedResources = "file:./grails-app/**/*Controller.groovy"
```

or this

```groovy
def watchedResources = ["file:./grails-app/**/*Controller.groovy"]
```

Before you move on to dealing with the change notifications themselves, let’s look at the example file pattern. The scheme (file: in these examples) is optional and determines how the rest of the path is interpreted. You almost always use the file: scheme.

The second part is the path pattern, which follows Ant’s conventions. A double asterisk wildcard (**) matches any number of nested directories (or none at all), whereas a single asterisk (*) can be used to match a single directory level or any part of a file or directory name.

In the example patterns you saw, matches would include ./grails-app/MyController.groovy and ./grails-app/org/example/MyOtherController.groovy, among others.

**TIP** To learn more about the syntax used for watched resources, see the Resources chapter of the Spring reference manual.1

Now that you know how to register for change notifications, how do you deal with them? Your `onChange` closure should accept a single argument, which will be an event object with the following properties:

- **source**—The file or artifact that changed. If a class changed, the source is the class instance (java.lang.Class); otherwise, it’s a Spring resource (org.springframework.core.io.Resource).
- **application**—The Grails application instance, which is useful for getting information about artifacts. This is redundant because the `onChange` closure has access to an application dynamic property.
- **ctx**—The Spring application context, from which you can access any beans you might need.
- **manager**—The plugin manager.
- **plugin**—The plugin descriptor instance.

Of these, `source` is the most useful. Armed with this information, you can work out what changed, then do whatever is necessary to ensure that the plugin continues to work as it should.

Let’s implement an `onChange` hook so that the security data bean is updated properly any time that a controller class is modified. Because this hook shares implementation with the security data initialization in `doWithApplicationContext`, let’s take this

---

1 The Resources chapter of the Spring reference manual can be found at, http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/resources.html.
opportunity to refactor the class. The following listing shows the necessary updates to your plugin descriptor.

Listing 20.9 Updating the security data bean

```java
class GrainaSecurityPlugin {
    ...[listing content]...
    def onChange = { event ->
        if (application.isControllerClass(event.source)) {
            def clsDesc = application.getControllerClass(event.source?.name)
            if (clsDesc == null) {
                clsDesc = application.addArtefact(
                    ControllerArtefactHandler.TYPE,
                    event.source)
            }
            def securityData = event.ctx.getBean("grainaSecurityData")
            processController(clsDesc, securityData)
        }
    }
    ...[listing content]...
    private processController(clsDesc, securityData) {
        def controllerName = clsDesc.logicalPropertyName
        def actionMap = securityData.authRequiredActions
        if (actionMap[controllerName] == null) {
            actionMap[controllerName] = []
        }
        for (method in controllerClass.clazz.declaredMethods) {
            def ann = field.getAnnotation(AuthRequired)
            if (ann != null) {
                def actionName = method.name
                actionMap[controllerName] << actionName
            }
        }
    }
}
```

Now that your security data updates whenever a controller class changes, you’re done with the access control implementation. But before we finish here, we should take an opportunity to mention the other hook related to reloading: `onConfigChange`.

This is similar to `onChange` except that you don’t need to set up any watched resources (it’s always called when the runtime configuration changes). The event’s `source` property is a reference to the main configuration object with the new settings.
When should you add reloading support to your plugin? Whenever you use information from classes that change, such as annotations, property values, and so on. You should also implement `onChange` if you ever add dynamic methods or properties to artifacts; those methods disappear in a puff of smoke after a reload.

At this point, your security plugin is fully functional. And yet you haven’t implemented all the available hooks. That’s not unusual, and in practice most plugins don’t need to use all the integration points. To explain the last of the hooks that we want to cover, you’ll switch to a public plugin that takes you away from Spring and into the world of servlets and servlet filters—two important aspects of a web application.

### 20.2.4 Playing with filters

Grails is heavily geared toward web application development, and although it’s easy to write a Grails application without thinking about the traditional web descriptor, it should definitely not be forgotten. Remember, this is the only way you can configure servlets and servlet filters.

For this section, we turn to the Shiro plugin to show you how to integrate with the web descriptor. It’s a plugin that also provides access control for Grails applications, and it needs to set up a servlet filter. You can force the user to manually configure this filter in their application web descriptor, but the plugin quickly loses friends if it tries that approach. It’s far better to configure the filter itself.

Reading the XML of the web descriptor, modifying it, and writing it out again is tedious, so you’ll be glad to hear that you don’t have to. Instead, implement the `doWithWebDescriptor` closure, and modify an in-memory representation of the web descriptor. How does it work? Grails starts by parsing a template web descriptor, the result of which is passed to the plugin’s `doWithWebDescriptor` closure as an argument. You can then use that argument to modify the web descriptor.

We know that the argument passed to the `doWithWebDescriptor` closure is an object representation of the parsed template web descriptor. That object representation allows you to use GPath syntax to both extract the information that you may need and add any extra elements or other XML content. The following listing shows how the Shiro plugin does it.

```groovy
def doWithWebDescriptor = { webXml ->
  def contextParam = webXml.'context-param'
  contextParam[contextParam.size() - 1] + {
    'filter' {
      'filter-name'('securityContextFilter')
      'filter-class'('org.jsecurity.spring.SpringJSecurityFilter')
      'init-param' {
        'param-name'('securityManagerBeanName')
        'param-value'('jsecSecurityManager')
      }
    }
  }
}
```

Listing 20.10  Using the Shiro plugin

- Gets all `<context-param>` elements
- Appends new filter definition after last `<context-param>`
def filter = webXml.'filter-mapping'.find {
    it.'filter-name'.text() == 'charEncodingFilter'
}
filter + {
    'filter-mapping' {
        'filter-name'('securityContextFilter')
        'url-pattern'('/**')
    }
}
}

The order of filter definitions doesn’t matter, so Shiro inserts one immediately after the last context-param element using the + { ... } syntax. The content of the closure is self-explanatory if you’re familiar with the web descriptor syntax for filter definitions. Each method name corresponds to the name of an element, and a new closure marks the start of nested elements. In practice, it’s equivalent to this:

<filter>
    <filter-name>securityContextFilter</filter-name>
    <filter-class>org.jsecurity.spring.SpringJSecurityFilter</filter-class>
    <init-param>
        <param-name>securityManagerBeanName</param-name>
        <param-value>jsecSecurityManager</param-value>
    </init-param>
</filter>

Although filter is in quotes in the example code, the quotes aren’t necessary. They’re mainly there for consistency with the other element names, which do require quotes because of the hypen (-).

We said that the order of the filter definitions doesn’t matter, but it’s a different kettle of fish when it comes to the filter mappings. The order in which they’re defined is the order in which they’re executed. For this reason, you need to be more careful about where you insert mappings. In the example, Shiro looks for the filter mapping for the charEncodingFilter, which is typically guaranteed to be there and almost all filters should come after it. You can also search for mappings added by other plugins, but you have to make sure that your plugin is loaded after those plugins. You can do this with a simple property in the plugin descriptor:

class ShiroGrailsPlugin {
    ...
    def loadAfter = [ "controllers", "services" ]
    ...
}

After you’re comfortable with the syntax for generating XML, you’ve effectively mastered the doWithWebDescriptor closure. There isn’t any more to it than that. You can add context parameters, set session timeouts, configure servlets—anything supported by the web descriptor.

You should now have a good idea of what you can achieve with the plugin hooks you explored in this section. Defining Spring beans is a particularly effective way of
providing default behavior that a user can override, so it’s worthwhile trying out the doWithSpring hook even if you don’t use the rest. Now all you need to do is make the plugin available for use in applications by publishing it.

20.3 Publishing your plugin

A plugin in isolation is a sad thing. Without an application to host it, it has no meaning. You saw throughout the book how to incorporate plugins into an application via declared plugin dependencies and now, in this section, you’ll see how to make a plugin available through that mechanism.

Before the plugin is published, though, you need to make sure that it works properly. It’s time to get your testing tools out again.

20.3.1 Testing plugins

A plugin has an almost identical structure to an application, which means that the unit and integration testing you’re already familiar with applies to plugins as well. Both types of testing are useful for checking that the artifacts work correctly, and you write the tests exactly the same way as for an application, putting them in the plugin’s test/unit and test/integration directories. Neither unit nor integration tests make it easy to test the hooks in the plugin descriptor, so functional testing is critical to ensure that a plugin works as expected after it’s installed in an application. It’s the only type of testing in which the plugin is exercised in something close to its final environment. Grails provides two approaches: running the functional tests as if the plugin were an application, or creating separate projects and testing those with the plugin installed.

Testing the plugin as an application

With the first approach, you install a functional testing plugin directly into the project, then create your tests as usual. What do you test? In the case of the security plugin, you need pages that have access control applied to them. The plugin doesn’t have any likely candidates at the moment, so you have to add test controllers and views, and maybe domain classes as well. The trick is to make sure that those extra artifacts don’t get packaged with the plugin.

Imagine that you added Post and User domain classes to the plugin for the tests only. You can exclude them from the official plugin package by using the plugin-Excludes property of the plugin descriptor:

```groovy
pluginExcludes = [  
    "grails-app/domain/Post.groovy",  
    "grails-app/domain/User.groovy"
  ]
```

This simple technique ensures that users won’t unexpectedly find their application creating tables for Post and User when they install your plugin.

Testing the plugin as an application is fine as long as that list of excludes stays short, but thorough testing is likely to see the list balloon in size. It’s also difficult to
test different configuration settings because you can only use one setting value per project. That’s where the second approach we mentioned comes in.

**USING MULTIPLE TEST APPLICATIONS**

Creating test applications for the plugin allows you to create a thorough suite of tests. You can have each application use different configuration settings for the plugin, for example. The problem is that the whole package and install process is too clunky to be useful. Every time you want to test a change to the plugin, you have to package it and install it in an application. Fortunately, there’s another way: you can configure an application to load a plugin from the plugin’s project directory. Add a line like this to the application’s `BuildConfig.groovy`:

```groovy
grails.plugin.location.security = "/path/to/plugin/dir"
```

Any changes you make to the plugin’s source files are seen by the application the next time you start it. If you use this setting, be sure not to add the plugin as a normal dependency!

Figure 20.3 illustrates the typical setup with an application containing two installed plugins and referencing the project directory of your security plugin.

You can now set up as many test projects as you need to cover all the necessary scenarios. We like to keep them in a `test/projects` directory alongside `test/unit`, with each project containing this setting in its `BuildConfig.groovy` file:

```groovy
grails.plugin.location.security = "../../..
```

Anyone can then check out the plugin source and run the functional tests inside each test project.

After you have solid tests behind you, you can think about packaging and releasing the plugin.

### 20.3.2 Releasing the plugin into the wild

To make a plugin available for public use, you must first package it into a zip file. You then publish the plugin to either the Grails Central Plugin Repository or a custom repository so users can declare it as a normal plugin dependency. We’ll look at the publishing mechanism shortly, but first let’s briefly cover how plugins are packaged.

![Diagram](attachment:image.png)

Figure 20.3 A Grails application referencing an in-place plugin (graina-security) with two other plugins installed normally. The security-plugin directory contains the plugin source code.
Packaging the plugin
Packaging is an essential part of software distribution, whether it involves creating an executable JAR file, a WAR, an RPM, or something else. In the case of a Grails plugin, the package is a simple zip file, and there’s a dedicated command for creating it:

```
grails package-plugin
```

This creates a zip file named grails-<name>-<version>.zip, where <name> and <version> are the name and version of the plugin.

Under the hood
When you package a plugin, Grails creates a plugin.xml file that contains the meta-information (such as version, author, and so on) and a list of the artifacts and other resources provided by the plugin. This file is added to the zip and is used by Grails in applications to learn about the plugin.

Why do we mention this? Because the plugin will break Grails (in sometimes mysterious ways) if the plugin.xml file is either corrupted or inconsistent with the resources packaged in the plugin. We don’t expect that you’ll ever have to worry about the file, but we also realize that sometimes stuff happens.

Once upon a time you could install such a package directly using the install-plugin command, but that approach is no longer supported by Grails. If you want to use the plugin in a project on your local machine, the simplest approach is to install the plugin in the Maven cache.

Start by checking whether the following dependency declaration is in your plugin’s BuildConfig.groovy, and if not, add it:

```
plugins {
    build(":release:3.0.1", ":rest-client-builder:1.0.3") {
        export = false
    }
}
```

The Release plugin provides several services, one of which is a command that copies the plugin package into the appropriate directory in your local Maven cache ($HOME/.m2/repository by default). To try this out, first ensure the Release plugin is downloaded and available by running grails compile or grails refresh-dependencies, and then run this command:

```
grails maven-install
```

To then use the plugin in your applications, add this entry to the list of repositories in BuildConfig.groovy

```
repositories {
    ...
    mavenLocal()
}
```
and add your plugin to the list of dependencies:

```groovy
dependencies {
    ...
    compile ":graina-security:0.1"
}
```

The next time you compile or run your application, Grails finds the plugin in the Maven cache and makes it available to the application.

Using the local Maven cache is fine for testing out your plugins in real applications, but no one else can use those plugins. You need to publish your plugins for others to have access to them.

**Publishing to the Central Repository**

The Grails Central Plugin Repository is a write-protected Maven-compatible repository, similar to Maven Central. As it’s write-protected, you need to get permission to publish your plugins to it. The process is straightforward, so do the following:

1. Register at http://grails.org/register
2. Submit the details of your plugin at http://grails.org/plugins/submitPlugin

Note that you can log in to the website via Twitter and Facebook, but you need a username and password when publishing a plugin. You can’t use Twitter or Facebook authentication for that.

Assuming that your request is approved, you can publish your plugin. When you’re ready, run this command, which is provided by the Release plugin:

```
grails publish-plugin
```

It asks for your username and password before packaging the plugin, pushing it to the Grails Central Plugin Repository, and then updating the plugin portal (http://grails.org/plugins).

This is great if you want to make your plugin public so the entire world can use it, but what if the plugin is specific to your own use case or contains private information to your business? You may want to make the plugin available to your team or to other teams in the same company, but not to everyone in the world. That’s where custom repositories come in.

**Publishing to Custom Repositories**

Custom repositories are Maven-compatible ones that aren’t the Grails Central Plugin Repository. You could use Sonatype’s Nexus or JFrog’s Artifactory to store all types of binary artifacts, including Grails plugins. Alternatively, you can use a system such as Bintray (https://bintray.com) to host your plugins publicly in your own account. Regardless of what repository implementation you use, the steps for publishing plugins to it are the same:

1. Configure a named repository.
2. Pass a `--repository=<repoName>` argument to the `publish-plugin` command or declare the default repository in `BuildConfig.groovy`. 
Step 1 involves adding these settings to either the project’s BuildConfig.groovy file or .grails/settings.groovy file in your user home directory:

```groovy
grails.project.repos.myRepo.url = "http://localhost:8081/repos"
grails.project.repos.myRepo.type = "maven"
grails.project.repos.myRepo.username = "admin"
grails.project.repos.myRepo.password = "password"
```

It’s not a good idea to include credentials in BuildConfig.groovy, so we prefer putting this information in the settings.groovy file instead.

In step 2, either run

```
grails publish-plugin --repository=myRepo
```

or add this line to BuildConfig.groovy:

```groovy
grails.project.repos.default = "myRepo"
```

The second option means that you can run publish-plugin without any extra arguments, so it’s generally preferred. The --repository argument is only useful for one-off overrides of the default repository, that is if you want to publish to a different repository for any reason.

Of course, you need to tell Grails applications to search for plugins in this repository, otherwise it won’t find them. But that’s as easy as adding the following entry to the list of repositories in BuildConfig.groovy:

```groovy
repositories {
    ...
    mavenRepo "http://localhost:8081/repos"
}
```

Use custom repositories to keep your own plugins private or to manage customized versions of public plugins. They can even control which versions of public plugins are installed by default by your team. They’re particularly useful when you need to deal with snapshot dependencies and want to control when those get updated because snapshot dependencies have a habit of breaking.

### 20.4 Summary and best practices

Plugins are a complex topic because they can interact with almost every aspect of Grails. Yet they’re such a fundamental part of the Grails platform that you need to understand them if you want to harness its full potential. Even the most trivial of applications uses at least one plugin.

You’ve seen numerous plugins throughout the book, and these form only a fraction of those publicly available. The sheer variety demonstrates how flexible and powerful the plugin system is. The public plugins can also serve as a source of inspiration and ideas.

Whether you use plugins to modularize your application or package self-contained features, they can dramatically simplify development for the plugins’s users and promote
Chapters 20: Developing plugins

This chapter shows you how easy it is to develop plugins, and gives you the tools to take that development further.

You can also use existing plugins as learning material. The Shiro plugin has a significant plugin descriptor with implementations for many of the hooks. It also has a few scripts you can look at. The GWT plugin has a good selection of scripts that use more advanced techniques than Shiro’s.

We finish off with guidelines for designing and developing your own plugins:

- **Keep it simple.** Use only the minimum number of features and hooks that you need to implement your plugins. If it doesn’t make sense to add dynamic properties or methods, then don’t.

  It’s also important to realize that plugins don’t have to be hulking, complex beasts. They can be simple. Remember, a plugin is always an option for code that’s self-contained or shared between projects, no matter how trivial.

- **Implement onChange when necessary.** If your plugin adds dynamic methods or properties, make sure that it also implements the onChange hook, so those methods and properties are retained after a reload. Users get attached to the reloading feature, so plugins that break it become unpopular.

- **Use local plugin repositories for teams.** Local plugin repositories serve several purposes. First, they allow you to control which versions of plugins are installed by your team by default. Second, they allow you to customize public plugins for your own purposes. Third, they can be useful for modularizing applications.

- **Modularize large or complex applications.** The principle of separation of concerns (SoC) is powerful, and plugins allow you to apply it to your Grails applications. You can use either the in-place plugin mechanism (via the grails.plugin.location.* configuration setting) or a local plugin repository to manage the modularization. The latter is a more reliable approach, but requires more work.

- **Write functional tests for your plugins.** For plugins to be reliable, you should write functional tests. The in-place plugin mechanism is particularly useful for this, allowing you to create multiple test projects that load the plugin from source.

Where do you go from here? Mastering Grails requires a good knowledge of the Spring framework and often Hibernate too. Grails simplifies many tasks, but it’s important to realize the amount of work that goes on under the hood. Very little software development is a smooth ride, and when things go wrong, a broad knowledge of the underlying technologies helps a lot in fixing those problems.

We also recommend that you learn as much about Groovy as possible, as it’s one of the biggest strengths of Grails. It’s also useful in writing scripts and in combination with Spock and makes a great language for writing tests for any Java project. We wish you the best of luck with your future endeavors!
It may be time for you to stop reconfiguring, rewriting, and recompiling your Java web apps. Grails, a Groovy-powered web framework, hides all that busy work so you can concentrate on what your applications do, not how they’re built. In addition to its famously intuitive dev environment and seamless integration with Spring and Hibernate, the new Grails 2.3 adds improved REST support, better protection against attacks from the web, and better dependency resolution.

Grails in Action, Second Edition is a comprehensive introduction to Grails 2. In this totally revised edition you’ll master Grails as you apply TDD techniques to a full-scale example (a Twitter clone). Along the way you’ll learn “single-page web app” techniques, work with NoSQL back ends, integrate with enterprise messaging, and implement a RESTful API.

What’s Inside

• Covers Grails 2.3 from the ground up
• Agile delivery and testing using Spock
• How to use and manage plugins
• Tips and tricks from the trenches

No Java or Groovy knowledge is required. Some web development and OOP experience is helpful.

There’s no substitute for experience: Glen Smith and Peter Ledbrook have been fixtures in the Grails community, contributing code, blogging, and speaking at conferences worldwide, since Grails 0.2.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/GrailsinActionSecondEdition

“Provides a solid foundation ... covers best practices and background knowledge.”
—From the Foreword by Dierk König, author of Groovy in Action

“The canonical guide to Grails.”
—Jerry Gaines, 4impact Group

“Packed with practical examples.”
—Pratap Chatterjee
Karolinska Institute

“The best resource to help you get ridiculously productive!”
—Michael A. Angelo
Laird Technologies