SAMPLE CHAPTER

Grails IN ACTION

Glen Smith
Peter Ledbrook

FOREWORD BY Dierk König

MANNING

Grails in Action

by Glen Smith

and Peter Ledbrook

Chapter 7

Copyright 2009 Manning Publications

brief contents

PART 1 INTRODUCING GRAILS 1

1 ■ Grails in a hurry... 3
2 ■ The Groovy essentials 31

PART 2 CORE GRAILS 63

3 ■ Modeling the domain 65
4 ■ Putting the model to work 92
5 ■ Controlling application flow 121
6 ■ Developing tasty views, forms, and layouts 155
7 ■ Building reliable applications 188

PART 3 EVERYDAY GRAILS 219

8 ■ Using plugins: adding Web 2.0 in 60 minutes 221
9 ■ Wizards and workflow with webflows 255
10 ■ Don't let strangers in—security 280
11 ■ Remote access 310
12 ■ Understanding messaging and scheduling 340

PART 4 ADVANCED GRAILS 363

13 ■ Advanced GORM kung fu 365
14 ■ Spring and transactions 395
15 ■ Beyond compile, test, and run 415
16 ■ Plugin development 442

7 Building reliable applications

In this chapter

- Why you should do testing
- How to test different parts of your application
- Choosing between different testing strategies

We've introduced all the main components of a Grails application, and you have learned more than enough to start developing some useful and usable applications. You're probably highly motivated to get coding and experimenting! And while your motivation is high, it makes sense to take a step back and look at how you can improve the quality and reliability of your applications through testing.

7.1 Why should we test software?

There are two approaches to writing software that works as it should: *correct by design* and *correct through testing*. Those in the former camp believe that you can design software to be correct from the outset. This philosophy has yet to penetrate the

commercial sector to any great extent, but its counterpart, demonstrating correctness through testing, has become immensely popular in recent years.

The principle is quite simple: for every line of code that we write, we should have a test that makes sure the code works as expected. When all the tests are passing, we know that the software is working correctly, and that gives us confidence when we release the software. The key is to automate as many of the tests as possible, so that they can be run frequently and reliably. Testing also allows you to refactor software with a far higher degree of confidence than would otherwise be possible. One of the many problems in software development is that changes can have unintended side effects, so anything that mitigates that is a significant benefit.

Tests take on even greater significance within Grails. As you know, the Groovy compiler can't pick up the same sorts of type errors that the Java compiler does because Groovy is a dynamic language. That means tests are the first line of defense against class cast exceptions and missing properties or methods.

Some would argue that this means you gain productivity in writing application code at the cost of a greater testing burden. We disagree. If you follow a strategy of light (or no) testing in Java, you'll certainly find the level of testing required for Grails applications to be higher. But this is a poor comparison; you should be aiming for full test coverage in all your applications, whether they be written in Java or Groovy. Doing otherwise runs the risk of introducing far more bugs into the software than can possibly be justified.

There are several benefits to testing, but in the end it comes down to a simple question: Do you want to write software that runs reliably? If the answer is yes (and we sincerely hope it is), there is no avoiding testing. But before you start going weak at the knees thinking about the extra burden of writing tests, we'd like to allay your fears. Even if you're not used to writing tests, you'll find that the process becomes easier the more you do it. Not only that, you'll almost certainly see your confidence in the code grow, and the tests will become an indispensable safety net.

We'll take a bottom-up approach to testing that starts with the unit tests and then moves on to functional testing. Another perfectly valid approach is top-down, in which you start with the feature-based testing and work down to the unit-testing level as you implement each feature. Which approach suits you or your team best is worth investigating, but whichever process you follow, the techniques we describe here are equally applicable. If you're unfamiliar with the terms "unit testing" and "functional testing," don't worry; we'll explain them as we go.

TEST-DRIVEN DEVELOPMENT

In recent years, the status of testing has changed. Once it was something that you did after the code was written, if at all. Now there is a movement that says, "Tests are important. In fact, they're so important that they should be written first!" The process championed by this movement is known as test-driven development (TDD). The principle is simple: write a test before you write any corresponding implementation code.

TDD offers several benefits. First and foremost, this approach guarantees that you have tests for all your classes. Second, you're able to develop code faster than otherwise because the test gives you early feedback on whether that code works or not—the "code a bit, test a bit" cycle becomes more efficient. Third, if you write the tests first, you're far more likely to design your classes and systems so that they're easily tested. Bolting tests on after the code has been written can be difficult without redesigning the code. Finally, TDD forces you to think about what behavior you want from your class at the outset. Focusing on the *what* rather than the *how* can improve the design of your classes and speed up code writing. Knowing where you're going helps you get there sooner.

Fixing bugs

We're sure that you're well versed in the art of fixing bugs, but we'd like to encourage you to follow a particular process when doing so. The first stage of fixing a bug should be to write a test for it. This test should be written in such a way that it fails until the bug is fixed.

You could fix the bug and then write a test afterwards, but how could you be sure that the test fails with the bug in place? You would have to revert the changes and run the test again. Far better to write the test first and retain the benefits of TDD.

Because of these benefits, we think that TDD should be actively encouraged. We could even take TDD further by following the top-down testing approach, which starts with writing functional tests, then writing unit tests, and only then writing the code. If you choose to take this route, make sure you build up the functional tests gradually, only testing the parts of the feature (or features) that relate to the specific classes you're working on. Attempting to write a functional test that covers the whole feature up front is something of a fool's errand.

7.2 Unit testing

Testing a Grails application can initially seem an intimidating prospect. Grails itself provides much of the plumbing and also seems to sprinkle many of your classes with magic, so it may seem next to impossible to test those classes in isolation. Never fear, though! Grails provides a testing framework to help you navigate these waters and make unit testing easy.

Why are we starting with unit tests? One reason is that we're taking a bottom-up approach, so it's the logical place to begin. But as you'll see later, you can also use Grails integration tests for "unit testing" if you want. There are two primary advantages of unit tests over integration tests:

- They're quicker to run.
- You can run them from within your IDE.

Reducing the length of your "code a bit, test a bit" cycles is crucial to maintaining focus and momentum, so unit tests are ideal at this stage.

That's enough of the background—let's take a look at some real code. Just as we started with the domain model back in chapter 3, we'll start with testing domain classes.

7.2.1 Testing domain classes

It might strike you as a bit strange to test domain classes—what is there to test? This is a valid question if your domain classes have no behavior (methods). But all domain classes contain logic that qualifies them for testing: their constraints. Because of this, Grails provides explicit support for testing constraints via `GrailsUnitTestCase`.

WRITING THE UNIT TEST

Let's take a look at how this works by writing a test for Hubbub's `User` class, which has some interesting constraints. All unit tests are stored under the test/unit directory, so create the file test/unit/com/grailsinaction/UserUnitTests.groovy if it doesn't already exist. Listing 7.1 shows the test case implementation.

Listing 7.1 Our first domain class unit test

```
package com.grailsinaction

class UserUnitTests extends grails.test.GrailsUnitTestCase {
   void testConstraints() {
      def will = new User(userId: "william")
      mockForConstraintsTests(User, [ will ])        // 1 Mocks the domain class

      def testUser = new User()
      assertFalse testUser.validate()                 // 2 Validates the test object
      assertEquals "nullable",
               testUser.errors["userId"]
      assertEquals "nullable",
               testUser.errors["password"]            // 3 Checks validation errors

      testUser = new User(userId: "william", password: "william")
      assertFalse testUser.validate()
      assertEquals "unique", testUser.errors["userId"]
      assertEquals "validator", testUser.errors["password"]
      ...
      testUser = new User(userId: "glen", password: "passwd")
      assertTrue testUser.validate()
   }
}
```

One of the first things you might notice is how much code there is in the test. You'll find that thorough tests often contain more code than what is being tested, but the benefits far outweigh the cost. You can see from the example that the test code itself is quite simple, albeit rather repetitive. Also note that we've named the class `UserUnitTests`: the `UnitTests` suffix isn't a requirement of Grails but it's a useful convention for easily distinguishing between unit and integration tests in your IDE.

The first thing that happens in listing 7.1 is that the test case extends a helper class provided by Grails: `GrailsUnitTestCase`. This is the granddaddy in the family, and you'll probably use it either directly or indirectly in all your unit tests. Not only does it make your life easier, but it's essential for creating mock objects—it prevents certain side-effects that make tests fail for no apparent reason.

> **Who are you mocking?**
> The process of *mocking* involves providing fake versions of classes or methods that can be used in place of the real implementations. The cool thing is that you can control the behavior of the mock object or method, so you can pass known data to the object under test.
>
> Mocking can be used at any level of testing, but it's most commonly associated with unit tests. Many of the methods provided by `GrailsUnitTestCase` are used to mock the dynamic properties and methods that would normally be injected into classes by Grails.

Next we use a special method provided by `GrailsUnitTestCase` to mock our domain class so that we can test its constraints ❶. Remember that the domain class has no knowledge of the constraints—it's Grails itself that interprets and processes them.

Once `mockForConstraintsTests()` has been called for a domain class, you can start testing the validation. All you have to do is create instances of the mocked domain class and call `validate()` on them ❷. If all the field values of the domain instance pass validation, the return value is `true`, otherwise `false`.

> **Under the hood**
> All of the `mock*()` methods provided by `GrailsUnitTestCase` use metaclass programming to add mock properties, methods, and objects. That means you can add your own or override the existing ones to fine-tune the behavior.
>
> In addition, the mock validation performs real Spring data-binding, so the `errors` property is a real instance of Spring's `Errors` interface. This allows you to check for multiple constraint violations on a single field, for example.

At this point, you'll find that your domain instance also has an `errors` property, which you can use to find out what fields failed and why. This property, created by the testing framework, behaves almost identically to the one automatically created by Grails, so you won't get confused when switching between tests and application code. The one difference is that you can't treat the `errors` property as a map in your application code. The big advantage of this extra behavior in tests is that you can easily check whether a constraint has been violated by passing the name of a field as the map key and testing the return value ❸. That value will be the name of the first constraint

violated by the corresponding field, or `null` if the field passed validation. If the field values in our `User` instance are `null`, we can check for the string `"nullable"` (that is, the name of the constraint).

RUNNING THE TEST

Let's try our test now. One way to do this is to run the test directly from within your IDE, if that's supported, but for now we'll use the Grails command:

```
grails test-app -unit
```

This will run all the application's unit tests (those under the test/unit directory) and generate reports for them. In our case, there is only one unit test, so the output looks like this:

```
...
Running 1 Unit Test...
Running test UserUnitTests...
SUCCESS
Unit Tests Completed in 1514ms ...
...
```

Not only do you get a summary of results printed to the console, but Grails also generates an HTML report in the test/reports/html directory. Fire up your browser and open the index.html file in that directory to see a breakdown of the tests by package and class.

How does Grails know which methods to execute for the tests? For example, you might have a `checkErrors()` method that's called from several of your test methods, but you wouldn't want it run as a test itself. Not a problem: Grails will only execute public methods that are explicitly declared to return `void`, whose name starts with "test", and that have no arguments. The `testConstraints()` method in listing 7.1 is an example.

It's straightforward so far. Let's now take a closer look at ❶, because we have yet to explain a rather important feature of `mockForConstraintsTests()`. You'll see that we pass a list of `User` instances as its second argument, albeit a single-item list in this case. To test the `unique` constraint on the `userId` field, we need some existing domain instances to compare the new value against, and the list we pass to `mockForConstraintsTests()` counts as the set of "existing" records. As you can see from the test case, `userId` fails validation when its value is `"william"` because the list already contains a domain instance with that value.

To see this in action, change the fourth line of the unit test to this:

```
mockForConstraintsTests(User)
```

Now run the tests again. This time you'll see the following line in the console output:

```
testConstraints...FAILURE
```

If you look at the HTML report, you'll see that the test failed because it expected `unique` but found `null`. The `unique` constraint was *not* violated.

At this point, you may be wondering whether the effort of testing constraints is worth it. If you aren't yet convinced, consider this: a typo in the name of the `constraints` field introduces a subtle and difficult to diagnose bug that people often spend hours trying to resolve. Surely the 5 to 10 minutes it takes to write a test is worth it, just for that?

Only one question remains: where do those `assert*()` methods come from? JUnit users will find them familiar, and for good reason.

BEHIND THE SCENES: JUNIT

JUnit is the de facto standard unit-testing framework for Java, so not only does Grails use it as the basis of its own testing framework, but Groovy itself comes with direct support for it. Figure 7.1 shows the relevant class hierarchy from JUnit, through Groovy, to Grails. Don't worry about the top three classes in the diagram—we'll get to them in due course.

Every Grails unit-test class you write will be one of these JUnit test cases, so you can use exactly the same techniques you would with plain JUnit. It's not only those classes that Grails takes advantage of either: the `test-app` command uses Ant's JUnit tasks to generate the reports.

Most of the `assert*()` methods are provided by JUnit's `TestCase`, but `GroovyTestCase` also contains some extra ones. Of particular note are the set of `shouldFail()` methods that allow you to check that a particular block of code throws an exception. For example, this code checks that a method throws an exception when it's passed an empty string:

```
shouldFail(IllegalArgumentException) {
   myService.testMethod("")
}
```

In this case, we specified the type of exception that we expect to be thrown, but you can also leave that argument out if you don't care what the type of the exception is.

Table 7.1 lists some of the more common and useful assertions that you have access to when your tests extend `GroovyTestCase`. Note that all the methods shown also have versions that take a string as the first argument. This allows you to specify a message

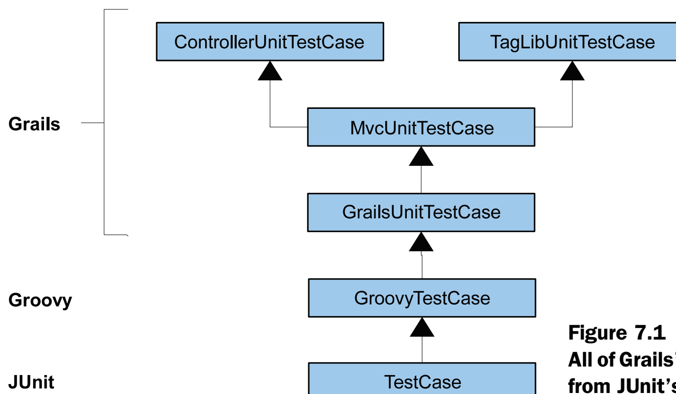


Figure 7.1 The Grails test class hierarchy. All of Grails' test cases are ultimately derived from JUnit's `TestCase`.

that will be reported when the assertion fails, allowing you to easily determine from the reports why a particular test failed. For example, if this assertion fails,

```
assertEquals "Unique constraint should have triggered, but didn't",
      "unique",
      testUser.errors["userId"]
```

the message "Unique constraint should have…" will appear in the test report.

Table 7.1 Useful assertions available to your test cases

Method	Description
`assertEquals(expected, actual)`	Compares two values or objects for equality using the `equals()` method
`assertTrue(value)`	Checks that the given value is `true` after being coerced to a Boolean
`assertFalse(value)`	Checks that the given value is `false` after being coerced to a Boolean
`assertNull(value)`	Checks that the given value is `null`
`assertNotNull(value)`	Checks that the given value is *not* `null`
`assertSame(expected, actual)`	Checks that two objects are the same instance
`assertArrayEquals(expected, actual)`	Compares two arrays for equality, checking that they have exactly the same elements
`shouldFail(type, closure)`	Checks that the given closure throws an exception of the given type, or any exception if `type` is omitted

That was a quick look at the JUnit framework, but you can find more information in books and online articles. One good place to start is http://www.junit.org. Most of that knowledge applies equally to Grails tests—that's one of the great things about reusing other frameworks!

Many domain classes include more than fields and constraints—people often enrich their domain model with methods, which obviously need testing. Methods on domain classes aren't much different from methods on any other class, so they can be tested in the normal way. We'll show you the techniques available by looking at testing a service, but remember that the approach taken applies to any class, including domain classes.

7.2.2 Testing services

It's a common (and recommended) practice to place the business logic of an application in services. We'll see more of services in later chapters, but from a testing perspective they're interesting because they're so ordinary. Grails doesn't inject any dynamic properties or methods into them (except for the `log` property), so the techniques we'll use to test them can be applied to any class.

Let's test a simple service. The following class is a rather contrived example (entirely unrelated to Hubbub), but it allows us to demonstrate a variety of Grails' unit-testing features:

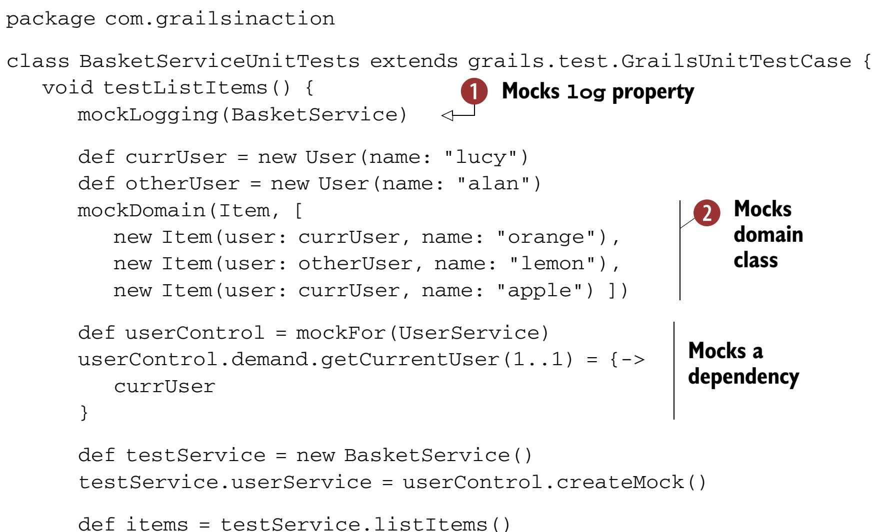
```
class BasketService {
   def userService

   def listItems() {
      def user = userService.getCurrentUser()
      if (!user) { throw new NoUserException() }
      else {
         log.info "Retrieving basket items for ${user.name}"
         return Item.findAllByUser(user)
      }
   }
}
```

The idea is that the `listItems()` method returns all the items in the current user's shopping basket, or throws an exception if there is no user.

There are some nasty obstacles in the way of testing this class, such as the query, the `log` property, and the field. At runtime, Grails provides all three itself, so you don't have to worry about setting them up. This isn't the case during a unit test, though, so we have to mock them ourselves either by using metaprogramming tricks (for the dynamic properties and methods) or by creating mock objects (for the service dependency). That's quite a bit of work to do for a single test, and it's not easy either.

Fortunately, Grails' testing support provides you with the tools you need to solve these problems easily and concisely. Listing 7.2 contains a unit test for the example service. Notice how the test case extends `GrailsUnitTestCase`? It provides all the mocking methods we're about to discuss.

Listing 7.2 Testing a Grails service



```
package com.grailsinaction

class BasketServiceUnitTests extends grails.test.GrailsUnitTestCase {
   void testListItems() {
      mockLogging(BasketService)

      def currUser = new User(name: "lucy")
      def otherUser = new User(name: "alan")
      mockDomain(Item, [
         new Item(user: currUser, name: "orange"),
         new Item(user: otherUser, name: "lemon"),
         new Item(user: currUser, name: "apple") ])

      def userControl = mockFor(UserService)
      userControl.demand.getCurrentUser(1..1) = {->
         currUser
      }

      def testService = new BasketService()
      testService.userService = userControl.createMock()

      def items = testService.listItems()
```

```
        assertEquals 2, items.size()
        assertEquals "orange", items[0].name
        assertEquals "apple", items[1].name

        userControl.verify()
    }
}
```

Checks mock methods were called

Let's take the `log` property first: by calling the `mockLogging(Class)` method ❶, any subsequently created instance of the given class will magically gain a working `log` property that echoes all log messages to the console. By default, debug and trace messages aren't included, to avoid excess output, but you can enable the debug messages by passing `true` as an optional second argument to `mockLogging()`.

A similar scheme exists for domain classes ❷, although it's more powerful and consequently more complicated. First of all, `mockDomain(Class, List)` adds dummy implementations of almost all the dynamic domain class methods. The most notable exceptions are criteria and Hibernate Query Language (HQL) queries. But dynamic finders are fully supported and behave as the real methods do.

Most of the `static` methods that Grails adds dynamically only make sense if there are existing database records to find and work with, but this is a unit test, so there isn't even a database. Fortunately, this is taken care of by the second argument to `mockDomain()`. Just as with `mockForConstraintsTests()`, that argument is a (potentially empty) list of domain instances representing existing database records that acts as the backing store for the dynamic finders and other related methods. For example, `Item.list()` will return all the items in the list passed to `mockDomain()`. What's more, the search methods will always return domain instances in the same order as they're defined in the list, which means guaranteed and consistent ordering for your tests. There is almost nothing worse than a test failing because the ordering of a collection has changed.

While we're talking about that second argument, what about `id` values? You can give each of the domain instances in the list an explicit `id` value if you want, but you don't have to. By default, each domain instance will be given an ID matching its position in the list, starting from 1.

id and version fields, and IDEs

As you'll have noticed by now, domain classes don't have explicit `id` and `version` fields, but you can still set their values from your code. This is because the Grails compiler adds those two fields to the generated class files.

Unfortunately, not all IDEs use the Grails compiler when compiling your application's domain classes, so the fields may not be added. If that's the case and you try to give either field a value in your unit test, the IDE will complain of a missing property when it attempts to run the test. Not surprising, because neither of them exist.

You can work around this problem by explicitly declaring `Long id` and `version` fields in the domain classes. It might also be a good idea to suggest to the IDE developers that they use the Grails compiler or inject the fields themselves.

Even if the method that you're testing doesn't use the search or object retrieval methods, it can still be worth passing a list into `mockDomain()`. The `save()` implementation will add the domain instance to the list, whereas `delete()` will remove it. This allows you to check whether those methods have been called correctly or not.

We've dealt with the `log` property and the domain classes, but we still need to work out a way of mocking those dependencies. None of the methods we've seen so far help us because they're far too specialized. What we need is a more general mocking system, and that's where `mockFor()` comes in.

7.2.3 General mocking in Grails

A mock object should look and behave like the class it's mocking. For example, if the class implements an interface, you could create a new class that also implements the interface but has mock implementations of the methods. But if you have lots of classes to mock and not all of them implement interfaces, this involves a lot of overhead. You can also run into problems with properties and methods that would normally be dynamically added to the target class.

What we need is a lightweight approach that enables us to add a mock implementation of any method and would allow us to mock only those methods that are called by the code under test, keeping the burden of testing low. The `mockFor()` method does exactly what we need.

The first step in mocking a dependency (or *collaborator*, in testing parlance) is to call `mockFor()` with the class you want to mock. You saw this in listing 7.2, but here's the relevant line again:

```
def mockControl = mockFor(UserService)
```

Why do we assign its return value to a variable called `mockControl`? Because the object returned by the method isn't a mock instance of `UserService` itself, but one that allows you to set up the mock methods and create the mock instance. Its behavior is similar to the `MockControl` class from the EasyMock 1.x library; hence the name of the variable.

Once we have the control object, we can start specifying the methods we expect to be called and providing mock implementations for them. In listing 7.2, we expect the service we're testing to call the `getCurrentUser()` method on the `userService` collaborator, so we declare that expectation using the `demand` property on the control object. Figure 7.2 breaks the call down into its component parts.

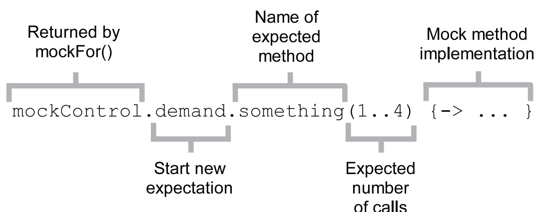


Figure 7.2 Setting an expectation on the object returned by `mockFor()`

This syntax will be familiar if you have ever used Groovy's `MockFor` or `StubFor` classes. The expected method appears as a call on the `demand` property with a range as its argument. This range specifies how many times you expect the method to be called. For example, `2..2` states that the method should be called exactly twice, whereas `1..4` states that the call should be called at least once, but no more than four times. If no range is given, an implicit value of `1..1` is assumed, which means the method should be called exactly once.

Why not use Groovy's MockFor class?

There are several reasons for not using `MockFor`, but the main one is that it doesn't fit as nicely with the metaprogramming system used by Grails as it could. Also, its syntax gets ugly if you need to mock more than one collaborator in any given test.

That doesn't mean you can't use `MockFor` in Grails unit tests, but we think you'll find the `mockFor()` method easier to use and a little more flexible.

The closure is the mock implementation. You can use it to check the values of the arguments and return a particular value or object to the code under test. The number and types of the closure's arguments should match those of the method being mocked, or your mock implementation won't be called.

What if the method under test accesses a property on one of its collaborators? Easy. As you know, property access in Groovy translates to calls on the corresponding getter and setter methods, so all you have to do is mock those methods. For example, say your collaborator has a property called `length`—mocking access to it is as simple as this:

```
mockControl.demand.getLength(1..2) {-> ... }
```

You can take the same approach with the setter method too.

We've now set up the expectations on the control object, so what's next? That depends. If the class under test creates the collaborator itself, there is nothing more to be done at this point. The collaborator will automatically have the mock method implementations that we defined. But in listing 7.2 we have to provide the collaborator ourselves, so we use the mock control's `createMock()` method:

```
testService.userService = userControl.createMock()
```

This has all been preparatory work. We still need to call the method that we're testing and check that it behaves as we expect. The main idea is to verify that its behavior corresponds to the values returned from the mocked methods. In many cases, it also makes sense to check that the method under test passes the appropriate argument values to those mocked methods.

After the method has been executed and we've tested its behavior, we're still not quite finished. How do we know whether the mocked methods have been called the correct number of times, or at all? It's quite possible that you don't care, and in some

scenarios that's appropriate. But what if the real implementation of one of the mocked methods persists some data somewhere, or sends an email? You'll want to make sure that the method is called.

Behind the scenes, the mock control counts the number of times a particular method has been called. As soon as one is called more times than expected, an assertion error will be thrown. That doesn't help us if the method is never called, but that problem is easily solved. Once you have executed the method under test, you can call `mockControl.verify()`. This method will throw an assertion error if any of the expected methods have been called fewer than the expected number of times.

Figure 7.3 illustrates the basic sequence of steps followed by a typical unit test. This is a common pattern that's effectively fixed by the way the mocking system works, so almost all of your own unit tests will look similar.

Before we move on, there are a couple of extra features of `mockFor()` that you might be interested in. The first involves mocking static methods, which can be achieved by inserting `static` after the `demand` property:

```
mockControl.demand.static.list() { Map args -> ... }
```

That's it. Nothing else needs to be done to get the mock static method working.

The second feature relates to the order in which the mocked methods are called. It might be important in certain scenarios for the method under test to call the various methods in a fixed order, and for any deviation to result in a test failure. This is known as *strict* mode, and it's the default behavior for `mockFor()`. On the other hand, the order may not be important at all, in which case you don't want tests failing because the methods are called in a different order. Fortunately, `mockFor()` has a *loose* mode for such cases, and you can enable it by passing `true` as the second argument:

```
def mockControl = mockFor(OtherService, true)
```

As you can see, `mockFor()` is a powerful and yet easy-to-use testing tool that can be used to mock almost anything. When it comes to controllers, though, you want

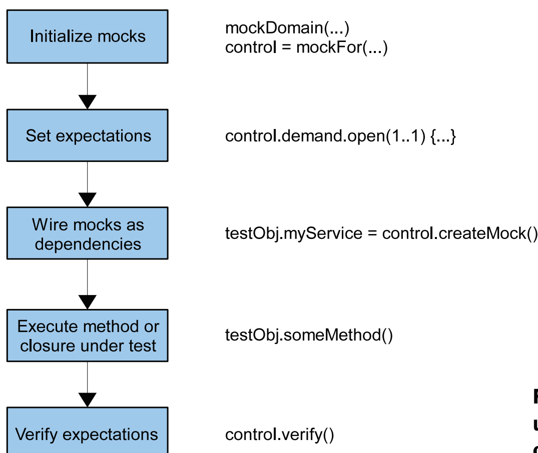


Figure 7.3 The structure of a unit test that uses Grails' mocking support, with examples of each step

something that will handle all the standard dynamic properties and methods. Manually doing it with mockFor() would be verbose and tedious in the extreme.

7.2.4 Testing controllers

Controllers are tricky beasts to test because they have so many dynamic properties and methods, as well as several ways they can render a page. They can return a model, render a view, render arbitrary text, or redirect to another URL. That's why Grails comes with a support class for controllers: ControllerUnitTestCase.

Let's take it for a spin by testing one of Hubbub's controllers. We'll start with one of the actions that returns a model. All we want to know is whether the model contains the data that it should, given the parameters passed in to the action. To do this, we need to set up some test parameters, execute the action, and inspect the map that it returns. Listing 7.3 shows how we go about that.

Listing 7.3 Testing a controller action that returns a model

```
package com.grailsinaction

class PostControllerUnitTests extends grails.test.ControllerUnitTestCase {
   void testShow() {
      mockDomain(User, [
            new User(userId: "glen"),
            new User(userId: "peter") ]
      this.controller.params.id = "peter"        <-- 1 Sets request parameter

      def model = this.controller.show()

      assertEquals "peter", model["viewUser"]?.userId      <-- 2 Tests action's model
   }
}
```

The ControllerUnitTestCase class instantiates a test controller at the start of each test, so we can start using it straight away (1). How does it know which controller class to instantiate? By convention. It looks at the name of the unit-test class and uses the string up to and including "Controller", so PostControllerUnitTests becomes PostController. If this convention doesn't work for you, you can pass the controller class directly to the super constructor:

```
PostControllerUnitTests() {
   super(PostController)
}
```

Often your controller actions work with parameters, so you want to pass those in somehow. That's easy—add them to the params map on the controller (1). This is one of several special properties that you have access to from your tests, all of which are listed in table 7.2.

We then invoke the action and assign the return value to a variable so that we can check the entries it contains (2).

Table 7.2 Mock controller properties available in unit tests

Property name	Description
`request`	The mock request (`GrailsMockHttpServletRequest`)
`response`	The mock response (`GrailsMockHttpServletResponse`)
`session`	The mock HTTP session (Spring's `MockHttpSession`)
`flash`	A map representing flash scope
`params`	A map representing and containing the action parameters
`redirectArgs`	A map of the arguments passed to the `redirect()` method by the action under test
`renderArgs`	A map of the arguments passed to the `render()` method by the action under test
`template`	A map of template arguments if `render()` has been called for a template; contains the name of the template under the map key `name` and an entry with a map key of `bean`, `collection`, or `model` (depending on which named argument was passed to `render()`)
`modelAndView`	A Spring `ModelAndView` instance that can be queried for both the name of the target view and the view's model

That's it—nice and painless. But controller code is rarely this simple, and actions that return a model are the easiest to test. What if the controller performs a redirect or uses the `render()` method?

How you handle these other cases depends on what mechanism the controller uses. Consider a redirect, for example:

```
redirect(controller: "post", action: "list")
```

To check whether the method is called correctly, look at the `redirectArgs` map on the controller:

```
assertEquals "post", this.controller.redirectArgs["controller"]
assertEquals "list", this.controller.redirectArgs["action"]
```

A similar approach can be used for the `render()` method, although you'll want to check the `renderArgs` property rather than `redirectArgs`. But not all forms of `render()` can be tested like this—some of them write to the response directly, including the following:

- `render ... as XML/JSON`
- `render { ... }`
- `render "..."`
- `render text: "..."`

Rather than check the `renderArgs` map, you can examine the content generated by these methods via the `contentAsString` property on the `response` object. Here we make sure that the start of the XML response begins with the `post` element:

```
assertTrue response.contentAsString.startsWith("<post>")
```

The main thing to be careful of when checking the response content directly is unexpected whitespace and ordering of text elements. For example, when you're rendering a Hubbub post as XML, how you do it affects what text is generated. Remember, most whitespace can be safely ignored by XML parsers, and the order of attributes isn't significant. So you may initially use an approach that generates nicely formatted and indented XML and later replace that implementation with one that uses no whitespace at all. Any pure string comparisons you make in your tests would then fail.

> **Under the hood**
> As you know, controllers have several dynamic methods and properties available to them at runtime, which makes them tricky to test. `ControllerUnitTestCase` makes life easier for you by using the `mockController()` method on `GrailsUnitTestCase` to add those methods and properties during test setup. These mock implementations are the ones that populate those `...Args` maps and render content to the mock response so that you can check it via the `contentAsString` property.

Controllers have quite a few unique features, and we could probably fill a whole chapter on the various techniques for testing them. Fortunately, you can read about the full gamut in the Grails user guide. Two items, though, deserve special mention: command objects and tag libraries.

COMMAND OBJECTS

Testing controllers that use the `params` object directly is easy enough, but using the same technique for actions that take command objects won't work. The problem is that the magic that binds parameters to the command objects at runtime isn't available to unit tests. But don't worry, the solution is surprisingly simple.

Say you have a `login` action that accepts a command object containing a username and password:

```
def login = { LoginCommand cmd ->
    ...
}
```

All you need to do in the test is create an instance of that command object and pass it to the action:

```
def testLogin() {
    def testCmd = new LoginCommand(username: "peter", password: "pass")
    this.controller.login(cmd)
    ...
}
```

It's that easy. The only fly in the ointment is with validation errors. Because you're creating the command object yourself, no validation is being performed on the data. That makes it difficult to test that your action is handling validation errors correctly.

Never fear—the Grails developers are one step ahead, and in this case you can call `validate()` directly on your command object:

```
def testLogin() {
   mockCommandObject(LoginCommand)

   def testCmd = new LoginCommand(username: "peter", password: "pass")
   testCmd.validate()
   this.controller.login(cmd)
   ...
}
```

If any of the fields in the command object contain invalid values, the `validate()` method will populate the associated `errors` property appropriately. The action under test won't be able to tell the difference: it will see a command object with validation errors.

TAG LIBRARIES

Testing with tag libraries is a bit less straightforward than with command objects. Unlike command objects, you can't pass tag libraries into a controller action, which means we need a different approach.

Let's take a look at some arbitrary controller code that uses the `link` tag:

```
def show = {
   ...
   render link(controller: "post", action: "list") { "Home" }
}
```

Notice how the tag invocation looks like a method call? That's our way in! We can mock the tag "method" using `mockFor()` on the controller, like so:

```
def mock = mockFor(MyController)
mock.demand.link(1..1) { Map args, Closure c ->
   ...
}
```

When the test runs, the controller will invoke this mock version rather than the tag. This approach will work regardless of whether the controller uses `link(...)` or `g.link(...)`. The notable missing feature is support for other tag namespaces, so if your controller calls tags that are in a custom namespace, you'll need to use integration tests.

It's all very well mocking tags when testing controllers, but it might be a good idea to test the tags themselves. They do contain code, after all.

7.2.5 Testing tag libraries

At first, the prospect of testing tag libraries might fill you with dread. You usually see them in Groovy Server Pages (GSP) files, so how do you test them from a simple class? Easy. A tag library is a class with some closures, and you can treat those closures like methods. In fact, tag libraries share a lot of aspects with controllers, from Dependency Injection to the dynamic properties that are available.

It won't come as a surprise to you that Grails provides a unit-test support class for tag libraries: `TagLibUnitTestCase`. This class behaves in almost exactly the same way as its controller counterpart, using convention to determine which tag library to instantiate. It also provides access to mock request, response, and session objects.

To demonstrate how to write a unit test for tag libraries, let's start with a simple tag that only writes its content out to the response if a user is logged into Hubbub:

```
class SecurityTagLib {
   def isLoggedIn = { attrs, body ->
      if (session.user) {
         out << body()
      }
   }
}
```

The tag would be used like this:

```
<g:isLoggedIn>You are logged in!</g:isLoggedIn>
```

It's simple, but it will help us demonstrate an important technique in testing tags.

Listing 7.4 contains the implementation of our unit test for this tag library. It tests both conditions: if a user is logged in and if no user is logged in.

Listing 7.4 A sample unit test for a tag library

```
package com.grailsinaction

class SecurityTagLibUnitTests extends grails.test.TagLibUnitTestCase {
   void testIsLoggedIn() {
      String testContent = "You are logged in!"
      mockSession["user"] = "me"
      tagLib.isLoggedIn([:]) {-> testContent }        ❶ Calls tag as a method

      assertEquals testContent, tagLib.out.toString()   ❷ Checks tag output
   }

   void testIsLoggedInNoUser() {
      String testContent = "You are not logged in!"
      tagLib.isLoggedIn([:]) {-> testContent }

      assertEquals "", tagLib.out.toString()
   }
}
```

There isn't anything too earth-shattering here. In one case, we add a `user` entry to the mock session, which the tag library can access via the dynamic `session` property. The more interesting aspects concern how we invoke the tag and then check the output it generates.

First of all, the unit-test support class automatically instantiates the tag library and assigns it to the `tagLib` property ❷. We then call the tag as if it were a method. One oddity is the empty map (`[:]`) as an argument, which is required because the tag expects an attribute map as the first argument. If we were passing in at least one attribute value, we could use the implicit syntax instead:

```
tagLib.isLoggedIn(attr1: "value") { ... }
```

Much nicer, but not an option if you have no attributes.

The real key to this unit test is the next bit on line ❶, where we pass a simple, zero-argument closure to the tag. As you saw in the previous chapter, the content inside a GSP tag is passed to the implementation as a closure, so here we provide a mock one that returns a simple string. If the tag writes out its content, which in this case happens if the session contains a `user` object, that string is streamed to the magic `out` property.

In order to check whether the appropriate text has been generated by the tag, we then perform a `toString()` on the `out` property and compare it to the expected value ❷. That's all there is to it, although it might be worth noting that the `out` property is an instance of `StringWriter`.

Not all tags are as simple as `isLoggedIn`, but all the features we've used so far in this chapter also apply to testing tag libraries. You can use `mockFor()` to mock dependencies and calls to other tags, and you can use `mockLogging()` if your tag uses the `log` property.

That pretty much covers everything you need to know about unit testing. It was a lot to digest in one go, so well done for making it this far! As you have seen, there aren't many things that you can't unit test with the classes provided by Grails. Sometimes, though, you can find yourself writing terribly complex mocks to test a simple method. This may be a sign that the method itself needs refactoring, but in many cases it's the nature of the beast. At that point, it's worth contemplating an alternative approach.

7.3 Integration testing

Where unit tests are about testing individual, atomic *units* of functionality (like classes and their methods), integration tests are designed to test larger parts of an application, such as a controller *and* its dependencies together. The main focus is on ensuring that the units work together correctly—that they *integrate* properly.

In Grails, integration tests appear on the surface to be unit tests. In fact, if you create a new integration test with this command,

```
grails create-integration-test Book
```

the only apparent difference is that the test is created in the test/integration directory rather than in test/unit. But as we all know, appearances can be deceptive.

The fundamental difference is that integration tests run inside a full Grails environment, with a running database and all those dynamic methods and properties that you're used to. You don't have to mock any of them. For all intents and purposes, the tests run against a real instance of your application. The main thing missing is the servlet container, the lack of which has certain repercussions.

Here is a summary of what works and what doesn't:

- All plugins that don't exclude the test environment are loaded.
- `BootStrap` is executed. Spring application context is fully populated.
- Domain classes work against a live database (an in-memory HSQLDB, by default).

- All dynamic methods and properties are available.
- Mock versions of the servlet request, response, and session are used.
- No Grails or servlet filters are executed.
- URL mappings have no effect.
- Views can't be tested.

Some of these may not mean much to you at this point, but we suggest you refer back to this list as you become more familiar with the trappings of Grails, such as the services and the Spring application context. You'll also get a better feel for integration tests as we work through some examples.

> **Under the hood**
> Each integration test is run inside a transaction that's rolled back at the end of the test. This means that no changes are committed to the database. Also, the default Grails environment for tests is "test".

You're probably now wondering when you should use integration tests and how to write them. It would be great if we could draw on some solid best practices from the Grails community, but few guidelines have emerged so far. Nonetheless, we'll endeavor to provide you with a useful starting point.

There are three main situations in which you'll need to write integration tests:

- When you can't test something with unit tests.
- When you want to test the interactions between objects.
- When it's easier to test with real data than to mock everything.

We'll look at some of the situations in which unit tests won't work in section 7.3.2, but first we'll look at the second and third points.

7.3.1 Filling the gaps

You have probably seen the "mind the gap" sign many a time, a helpful hint that you risk tripping or falling headfirst into a hole. It may surprise you, but that warning has a special resonance when it comes to testing applications. Unit tests are tremendously useful and help ensure that classes and methods are doing what they should *in isolation*, but there are plenty of things that can go wrong when they start interacting with each other. Those interactions constitute the "gaps" in the unit testing coverage that you need to be mindful of.

Let's look at a concrete example. Say you have a simple controller that uses a service to get a list of items associated with a user. It then extracts the names of the items into a new list and passes them to the associated view. The code would look something like this:

```
class MyController {
   def myService
```

```
   def show = {
      def items = myService.fetchItemsFor(params.userId)
      [ itemNames: items.collect { it.name } ]
   }
}
```

Let's also assume that the unit tests for the controller and service are both passing, so there's nothing obviously wrong with the controller code. Does that mean the code is fine?

Whether there's a problem or not depends on the behavior of the service method: if it returns an empty list when there are no items, everything works fine. But what if the service method returns `null` instead? There's no reason it can't do so, but now you can see that the controller will throw a `NullPointerException` because of the code `items.collect {...}`. This bug has materialized because of a faulty assumption on the part of the controller: that the service method will return an empty list rather than `null` (which is why methods should return empty collections rather than `null`).

TIP Bugs often manifest themselves in the boundary conditions of applications, when collections are empty, values are missing, or numbers are zero or negative. That's why it's crucial to test as many of these conditions as possible, particularly because they tend to be ignored during manual testing.

This example demonstrates the kind of issues that can crop up when classes that pass their unit tests start working together. When you add in dynamic methods and properties, you'll find that there are plenty of things that can go wrong. A simple integration test would have picked the problem up quickly, and you can see an example test in listing 7.5. This test not only covers the case where there are no items, but also the one where there are a few.

Listing 7.5 Your first integration test—including domain objects and a service

```
import grails.test.GroovyPagesTestCase

class MyControllerTests extends GroovyTestCase {
   def myService                                         // 1 Injects MyService instance

   void testShowNoItems() {
      def myController = new MyController()              // 2 Creates object to test
      myController.myService = myService

      myController.params.userId = "glen"                // 3 Sets action parameters
      def model = myController.show()

      assertEquals 0, model["itemNames"].size()
   }

   void testShow() {
      new Item(userId: "glen", name: "chair").save()     // 4 Initializes data
      new Item(userId: "peter", name: "desk").save()
      new Item(userId: "glen", name: "table").save(
            flush: true)
```

```
        def myController = ...
        ...
        def names = model["itemNames"]
        assertEquals 2, names.size()
        assertNotNull names.find { it == "chair" }
        assertNotNull names.find { it == "table" }
    }
}
```

5 **Checks results**

On the surface, the test looks like the unit tests we looked at earlier. The differences are profound, though, so let's take a closer look. The first "oddity" you'll see is the `myService` property **1**, like the one in `MyController`. When you run the test, Grails will set this property to a fully configured instance of `MyService`. It might seem strange to inject dependencies into a test like this, but the reason will be made clear as we look at the test method itself.

As with unit tests, if you want to test a particular class, be it a controller, a service, or whatever, you instantiate it with `new` **2**. The problem with this approach is that any dependencies that would normally be injected at runtime (`myService`, in this case) are left empty. As you'll see in chapter 14, the dependencies are only populated if the object is managed by Spring. (Integration tests and `BootStrap` are exceptions to this rule, but they're instantiated by Grails.) To work around this problem, we manually set the `myService` property on the controller to the value of the test's `myService` field. This may look like a hack, but it's the 100 percent certified way to handle such dependencies in integration tests.

Fed up with mocking?

Sometimes it can be tiresome to mock class after class in your unit tests, at which point integration tests become appealing. Beware, though: the effort required to set up the data in such a way as to test all boundary conditions in integration testing can be more than that required to mock collaborators. On the other hand, sometimes it can be far easier.

Unfortunately, there are no hard and fast rules, so it's worth experimenting. If an integration test proves easier to write, then go for it! The harder the tests are to write, the less likely they are to be written.

Once the test controller is created and initialized, it's time to provide some test parameters **3**. Note that we used exactly the same syntax in our unit tests, so you shouldn't get too confused when switching between unit and integration tests. One notable difference is with command objects: in integration tests you populate the `params` object as we have done in this example. Grails automatically instantiates any command objects declared by the controller action and binds the parameters to them. This means you can add a command object to an action without impacting its test.

Finally, we come to the issue of test data. With unit tests, the data is either passed into the object or method under test, or it's provided by mock collaborators. The

approach for integration tests is different: the tests are run against a live database, so you create data via the domain classes (4) as you do in the live application. But because you are working against a database, there are no guarantees that data will be returned in the order you expect, unless it's explicitly sorted. Bear that in mind when you assert results, and try to come up with order-independent ways of testing the data (5).

Hopefully you now have a reasonable idea of what an integration test is and why they come in handy. You now know enough to write integration tests for any controller or service. For further information, check out the Grails user guide, which has more specific information on testing controllers, tag libraries, and other types of Grails artifacts.

For services and controllers, you have a choice of which road to go down, but there are some situations in which unit tests don't cut it.

7.3.2 When only an integration test will do

Grails' unit test support will get you a long way in your testing endeavors, but there *are* gaps in its coverage you need to be aware of. In particular, it doesn't give you any help with either criteria or Hibernate Query Language (HQL) queries, nor does it have any support for testing URL mappings. You'll also see in chapter 9 that web flows require integration tests.

When it comes to criteria and HQL queries, you can use the integration-testing approach you saw in listing 7.5. Grails will take care of everything for you. You could even switch between dynamic finders, criteria, and HQL queries without needing to touch a line of the test case.

Testing URL mappings is a simple matter of leveraging the `GrailsUrlMappingsTestCase` support class. Let's say, for the sake of argument, that we have these mappings:

```
class UrlMappings {
   static mappings = {
      "/basket/$username"(controller: "basket", action: "show")
      "/$controller/$action?/$id?"()
      "500"(view: "error")
   }
}
```

Our test case would look something like the code in listing 7.6, which makes sure that the correct mapping is used for particular URLs. It will also test the reverse mapping—from controller and action to URL.

Listing 7.6 Testing URL mappings

```
class UrlMappingsTestCase extends grails.test.GrailsUrlMappingsTestCase {
   void testMappings() {
      assertUrlMapping("/item/edit/123",
                  controller: "item",
                  action: "edit") {
         id = 123
      }
```

(1) Tests URL mapping

```
        assertUrlMapping("/basket/fred",
                    controller: "basket",
                    action: "show") {
            username = "fred"           <-- Checks that parameter
        }                                   is mapped

        assertForwardUrlMapping(500, view: "error")    <-- Tests status code
    }                                               (2) mapping
}
```

As you can see, setting up the test is quite straightforward. The superclass automatically loads all the application's URL mappings and does all the necessary preparation in `setUp()`, so we can go straight into the tests themselves. The foot soldier of URL mapping tests is the `assertUrlMapping()` method (1), which accepts a URL as the first argument, and a controller and action as named ones. In the first test, we make sure that the URL `"/item/edit/123"` maps to the `item` controller and `edit` action, with the value `123` mapped to the `id` parameter. This method also checks that the reverse mapping works.

The second test is similar to the first, but we're making sure that the more specific URL mapping (`"/basket/$username"`) overrides the more general one.

Finally, we test that the `500` status code mapping works (2). We use the `assertForwardUrlMapping()` method here because the support class doesn't support reverse mappings for status codes.

There is also a corresponding `assertReverseUrlMapping()` method that accepts the same arguments as the other assertion methods.

There isn't any more to integration tests than that. Writing them is surprisingly simple. The trick is determining when to use them. Hopefully we've given you enough information that you can make an informed decision.

The next stage of testing involves making sure that the application works as a whole, particularly the user interface. To do that, we need to use a system of testing that runs the application in a servlet container and interacts with it. We need functional tests.

7.4 Functional testing

A combination of unit and integration tests can give you a great deal of confidence that your application is working, and they're fairly easy to develop and run. Yet there are gaping holes in their coverage: views, filters, and controller interceptors. How do we go about plugging those gaps?

The answer is to test the whole application, including the user interface. That's what end users see and interact with, so it's important to make sure it's working as expected. We call this "functional testing," and there are several different ways to do it. You could, for example, sit a person in front of a computer screen and get him to exercise the application by following some written instructions: click on this link, enter this data, click on that button, and so on. This is a perfectly valid approach and one that's probably still widely used.

The reason companies often do manual testing is because functional tests can be difficult to write, and user interfaces are particularly troublesome. But with manual testing you lose the benefits of automation: regular feedback, reproducibility, and so on. We don't want that, so it's good to know that there are several Grails plugins that can help us out. We'll start by looking at a way to exercise our UI without a browser.

7.4.1 Introducing the Functional Test plugin

The natural environment for a Grails application is a servlet container. The only way to test the application in that environment is by communicating with the server via HTTP. Several Grails plugins allow you to do this, and the most mature one is the WebTest plugin from Canoo. But we're going to give a relatively new kid a chance to strut its stuff: the Functional Test plugin.

We begin by installing the plugin:

```
grails install-plugin functional-test
```

Although it doesn't have a particularly catchy name, the Functional Test plugin more than makes up for it in use.

Before we can write a test using our freshly installed plugin, we first need something to test. For this example, we're going to look at the Hubbub page containing the user's timeline, because it has plenty of content we can test, and it also includes a form. Creating a skeleton test for this page is as simple as this:

```
grails create-functional-test PostList
```

This command creates the test/functional/PostListTests.groovy file, which you'll find is pretty spartan.

Our next step is to flesh it out with some test code. Listing 7.7 contains two tests: the first checks that the application generates a server error if the client attempts to access the timeline page without first logging in; the second tests that the page is displayed correctly when a user *is* logged in.

Listing 7.7 Functional test for Hubbub's timeline page

```
package com.grailsinaction

class PostListTests extends functionaltestplugin.FunctionalTestCase {
   void testTimelineNotLoggedIn() {
      get("/post/timeline")                    // 1 Sends GET request to URL

      assertStatus 500                         // Checks HTTP status
   }

   void testTimeline() {
      post("/login/index") {                   // 2 Sends POST with data
         userId = "peter"
         password = "password"
      }

      assertTitle "Hubbub » New Post"          // 3 Checks page content
      assertContentContains(
         "What are you hacking on right now?")
```

```
    }
}
```

The test may not look like much, but it does a surprising amount. We start off by sending a GET request to the timeline page's URL ❶. In this case, the URL is relative to the servlet context because it starts with a slash (/) but you can also specify either of the following:

- absolute URLs (of the form "http://...")
- relative URLs (such as "post/timeline")

URLs of the second form are interpreted as relative to the *last URL loaded* or, if this is the first request, a default value (specified by the config option `grails.functional.test.baseURL`).

Under the hood

The Functional Test plugin uses the HtmlUnit library under the covers, so if you're familiar with it, you can access the HtmlUnit `Page` object via the `page` property:

```
assertTrue page.forms["userDetails"]
```

You might also notice that `get()`, `post()`, `put()`, and so on, don't return anything. That's because the plugin (or more specifically, HtmlUnit) saves the response in memory—all subsequent actions and assertions apply to that response until the next request is sent.

We know that a user can't access this page directly without logging in, so we expect that the response will have a status of 500, which we check. And that's it for the first test.

The second one starts by logging a user into the application with a POST to the login controller ❷. This is similar to our earlier GET, but we can populate the POST data by passing a closure to the `post()` method. Each property assignment with that closure corresponds to a parameter value in the POST data. It's simple and easy to read—just how we like it. And you aren't limited to GET and POST—the plugin supports `put()` and `delete()` methods too.

Getting back to the test, the login action redirects to the timeline page for the user, so we check that the page title and contents ❸ match that page. You can also check that the page is a result of a redirect by using this approach:

```
redirectEnabled = false
post("/login/index") { ... }

assertRedirectUrl("/post/timeline")
followRedirect()
redirectEnabled = true
```

The `redirectEnabled` property allows you to control whether the plugin "follows" redirects. By setting it to `false`, you can assert the URL of the redirect and get the

plugin to load the target URL via the `followRedirect()` method. In our case, though, we're only interested in whether the timeline page is displayed, not whether it happened via a redirect.

Now that the test has been written, we can run it:

```
grails functional-tests
```

The tests take a while to run because the command first has to start your application in a servlet container. Once it has finished, you'll find that it generates the same types of reports in exactly the same place (test/reports) as the `test-app` command.

In this brief example, we've demonstrated only a few of the assertions you can use, but the plugin supports quite a collection, and we've listed many of them in table 7.3. You can find the full selection documented at http://www.grails.org/Grails+Functional+Testing.

Many of the assertions have variations with "Contains" and "Strict" in the name, so there is some flexibility there.

Table 7.3 Common assertions in the Functional Test plugin

Assertion	Description
`assertStatus(int)`	Checks the status code of the HTTP response
`assertContent(String)`	Checks the content of the page (case-insensitive comparison and whitespace is ignored)
`assertTitle(String)`	Checks the page title
`assertHeader(String, String)`	Checks the value of the named HTTP header
`assertContentType(String)`	Checks the content type of the response

By this point, you may be thinking that it all looks like too much hard work and that the test code is too low-level. True, the feature set may be small, but it gives you a great deal of power and flexibility. You can easily group fragments of tests into your own methods and reuse them. For example, say we need to check the timeline page in several tests. We don't want to repeat the code over and over again, so instead we can factor out the assertions into a method, like so:

```
def checkTimelinePage() {
   assertTitle "Hubbub » New Post"
   assertContentContains "What are you hacking on right now?"
   ...
}
```

We can call this method from anywhere in our tests to perform a complete check on the page. By building intermediate blocks of code like this, we begin to create a suite of robust and thorough tests.

This is all good stuff, but a word of warning: UI tests are inherently fragile because the user interface tends to change frequently, if only in small ways. The key to

overcoming this fragility is to strike a balance between checking enough aspects of the UI to ensure it's working correctly and not being so specific that the tests regularly break. Experience counts for a lot here, but we suggest you err on the side of being too strict with the tests, because you can easily loosen them if they break too often. Erring the other way means that bugs get through.

And that (almost) completes our stack of tests covering (almost) every aspect of the application. Functional tests provide that last element of confidence that things are working. You can use them for system integration testing too, so if and when you deploy the application to staging or production, you can run the functional test suite to check that all is well.

We've "almost" completed our testing coverage, so what's left? That would be JavaScript and other client-side technologies.

7.4.2 Other testing tools

Both the WebTest and Functional Test plugins use the Rhino JavaScript engine, and although it's improving, the engine still struggles with some JavaScript libraries. That leaves a gap for some other tools.

SELENIUM

Where the Functional Test plugin uses a Java library to send HTTP requests and handle the responses, Selenium piggybacks on the browser. This has two distinct advantages over the other approach:

- You can check that your application works with the browsers your users prefer.
- You can get the browser's JavaScript handling for free!

Differences between browsers are the bane of a web developer's life, so the ability to run a full suite of automated tests on multiple browsers isn't to be sniffed at. And if the other functional testing plugins choke on your beautiful, dynamic user interface because they can't handle the JavaScript, using the browser to do the tests becomes a no-brainer.

A nice feature of this tool is the Selenium IDE—a plugin for the Firefox browser. This gives you a point-and-click interface for writing your tests. You'll almost certainly want to customize the generated tests, but it makes getting started that much easier. You can find out more on the plugin's web page: http://www.grails.org/Selenium+plugin.

This raises an interesting question: should you use Functional Test (or WebTest) or Selenium? Maybe even both? This is a difficult question, and it pretty much comes down to personal preference. You may be more comfortable with the user interface presented by Selenium IDE, but the Functional Test and WebTest plugins generally run quite a bit faster.

Being code junkies, we like the expressiveness of the Functional Test plugin and the control it gives us in writing the tests. We prefer to start with that and only consider Selenium if we need cross-browser testing or if the JavaScript is causing problems.

JSUNIT

Speaking of JavaScript, isn't that code? Shouldn't we be testing it too? Yes, and maybe. Although it's code, you can effectively test it via Selenium if you want. But then you're talking about functional tests for your JavaScript rather than unit tests.

If you have non-trivial JavaScript in your application, it makes sense to unit test it. That's where JsUnit comes in and, as you can probably guess, it's a member of the xUnit family of testing frameworks. Install the jsunit plugin for Grails and away you go!

You can check out the plugin documentation on the Grails website and the JsUnit documentation at http://jsunit.berlios.de/.

FIXTURES

As you know, Grails already has a mechanism for setting up data on startup: the `BootStrap` class. You may think this is more than enough for your needs, but if that's the case, we'd like you to reconsider.

The main problem relates to functional tests: although your `BootStrap` class is called when the server starts up, there's no way to reset the data in the middle of the tests. That means your database starts filling up with records, and your later tests have to account for all the extra data added by earlier ones. Nightmare!

As an example, imagine we have a test that expects the timeline page to display five posts. Then another developer comes along and creates a test that adds a post to that timeline. Suddenly our test breaks because the timeline page is now displaying *six* posts.

The solution to this problem lies in another of Grails' wonderful plugins: fixtures. It allows you to configure a set of data (using domain classes rather than SQL) that you can load at any time. You can see an example of its use in integration tests on its web page: http://www.grails.org/Fixtures+Plugin.

Using fixtures in functional tests is trickier, because your tests are running outside of the server. One option is to set up a dedicated controller that drops the existing data from the database and then loads the fixture data. Just be careful that you don't make the controller available in the production system.

That concludes our quick look at alternative testing tools and also our coverage of testing in Grails. You should now have a solid foundation in unit and functional testing and the knowledge to build on that foundation. You'll find that there are many options to make testing easier and more fun, including tools like easyb for behavior-driven development (http://www.easyb.org/) and the Code Coverage plugin for seeing how much of your code is exercised by tests (http://www.piragua.com/ or http://www.grails.org/Test+Code+Coverage+Plugin). So, keep on testing!

7.5 Summary and best practices

Testing is an essential part of ensuring product quality. Without it, how can you be confident that your application works? It also has a massive impact on the process of refactoring, because without testing you have no idea whether you have broken something.

The first aim of this chapter was to instill a testing habit. Once testing becomes a habit, you'll find it much easier to keep it going. The second was to show you how to write tests at all levels: unit, integration, and functional. You need to utilize testing at all these levels to ensure maximum reliability. The tools are there, and they will only get better, so why not make use of them?

The following guidelines will help you get the most out of your testing:

- *Test first, test often* By writing your tests first, you not only ensure that your code is covered, but you also have to think clearly about what you want the code to do. If you run the tests often, you get quicker feedback when something breaks, which makes it easier to fix. This helps speed up the development cycle.
- *Make testing easy* You might think making testing easy is the responsibility of the testing framework, and you would be partly right. But making your code easy to test can go a long way to ensuring that it does have tests. Such things as allowing test cases to inject collaborators, using interfaces, and avoiding static variables and methods can be a great help.

  In addition, if you do find something difficult to test, look for a solution that can be reused for similar code. Otherwise, none of it will have associated tests.
- *Maintain test coverage* It can be easy to skip tests or postpone writing them, but that's the start of a slippery slope. There is a theory that neighborhoods with visible broken windows are more susceptible to crime, because people think that no one cares so they don't either. Something similar happens with tests: if there are gaps in the testing coverage, developers form the impression that tests don't matter much. Try to avoid gaps in test coverage as much as possible.
- *Use continuous integration* When you're working with a team, you'll quickly find that code committed by one person breaks when it's combined with changes from someone else. Use a continuous integration (CI) system to ensure that these breakages are picked up early and often. In fact, CI can even be useful for a one-person team, by detecting files that haven't been committed, for example.
- *Test at all levels* We can't stress this enough: make sure that you're testing at both the unit and functional levels. Even if there is some overlap and redundancy, that's better than having gaps in your test coverage.

And so the lecture finishes. As a reward for persevering, you now get the opportunity to have some fun with cool plugins.

JAVA

Grails IN ACTION

Glen Smith and Peter Ledbrook, FOREWORD BY Dierk König

Free ebook
SEE INSERT

Web apps shouldn't be hard to build, right? The developers of Grails agree. This hyper-productive open-source web framework lets you "code by convention," leaving you to focus on what makes your app special. Through its use of Groovy, it gives you a powerful, Java-like language and full access to all Java libraries. And you can adapt your app's behavior at runtime without a server restart.

Grails in Action is a comprehensive guide to the Grails framework. First, the basics: the domain model, controllers, views, and services. Then, the fun! Dive into a Twitter-style app with features like AJAX/JSON, animation, search, wizards—even messaging and Jabber integration. Along the way, you'll discover loads of great plugins that'll make your app shine. Learn to integrate with existing Java systems using Spring and Hibernate. You'll need basic familiarity with Java and the web. Prior experience with Groovy is not necessary.

What's Inside

- A concise Groovy primer
- Advanced UI development
- Enterprise integration
- Plugin development
- Tips and tricks from the trenches

About the Authors

A frequent speaker and the co-host of the Grails podcast, **Glen Smith** launched the first public-facing Grails app (an SMS Gateway) on Grails 0.2. **Peter Ledbrook** is a core Grails developer and author of several popular plugins, who has worked as an engineer for both G2One and SpringSource.

For online access to the authors, code samples, and a free ebook for owners of this book, go to www.manning.com/GrailsinAction

"... rock-solid solutions delivered with ease."
—From the Foreword by Dierk König

"Bulging with serious content and tips. Bursting with fun."
—Paul King, ASERT

"Excellent resource, both as a guide and a reference."
—John Guthrie, Sybase

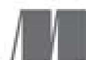
"Grails is the PHP of Web 2.0 and this book is pure gold."
—John G. Ledo
Ledo Works Inc.

"... an approachable and entertaining book ... sure to simplify your Grails development."
—Joe McTee, JEKLsoft

"Great framework + Great authors = Awesome book!"
—Dave Klein, Contegix

"Read this book if you want to raise your Grails IQ by 100+ points!"
—Zan Thrash
Thrash Consulting

ISBN-13: 978-1933988931
ISBN-10: 1933988932



MANNING US $44.99/CAN $56.99 [INCLUDES EBOOK]