IBM

developerWorks.

Mastering Grails: Testing your Grails application

Squashing bugs and building executable documentation

Skill Level: Introductory

Scott Davis (scott@aboutgroovy.com)
Editor in Chief
AboutGroovy.com

14 Oct 2008

Grails makes it easy to ensure that your Web applications start out bug free and stay that way. As a bonus, you can leverage your test code to produce a rich set of executable documentation that is always up-to-date. This month, Grails guru Scott Davis shows you the Grails testing ropes.

I'm a huge advocate of test-driven development (TDD). Neal Ford (author of *The Productive Programmer*) says that "it is professionally irresponsible to write code that isn't tested" (see Resources). Michael Feathers (author of *Working Effectively with Legacy Code*) defines "legacy code" as any software that doesn't have a corresponding test associated with it — implying that writing code without tests is an antiquated practice. I often say that a project should have two pounds of test code for every one pound of production code.

Mastering Grails hasn't yet covered TDD because the series has concentrated until now on ways to take advantage of core Grails functionality. There's a modest bit of value in testing infrastructure code (that is, code that you didn't write), but in practice I rarely do it. I trust Grails to render my POGOs to XML correctly, or save my `Trip` to the database when I call `trip.save()`. The real value proposition of testing comes into play when you are validating your own custom code. If you write a complex algorithm, you should also have one or more complementary unit tests to prove that the algorithm does what it says it does. In this article, you'll see how Grails enables and encourages you to test your application.

Writing your first test

© Copyright IBM Corporation 1994, 2008. All rights reserved.

To get started with testing, I'll introduce a new domain class. The class will eventually have some custom functionality that shouldn't be put into production without some tests in place. Type `grails create-domain-class HotelStay`, as shown in Listing 1:

About this series

Grails is a modern Web development framework that mixes familiar Java™ technologies like Spring and Hibernate with contemporary practices like convention over configuration. Written in Groovy, Grails give you seamless integration with your legacy Java code while adding the flexibility and dynamism of a scripting language. After you learn Grails, you'll never look at Web development the same way again.

Listing 1. Creating the HotelStay class

```
$ grails create-domain-class HotelStay

Environment set to development
     [copy] Copying 1 file to
/src/trip-planner2/grails-app/domain
Created Domain Class for HotelStay
     [copy] Copying 1 file to
/src/trip-planner2/test/integration
Created Tests for HotelStay
```

As you can see in Listing 1, Grails creates an empty domain class for you in the grails-app/domain directory. It also creates a `GroovyTestCase` class with an empty `testSomething()` method in the test/integration directory. (I'll talk more about the difference between unit and integration tests in just a moment.) Listing 2 shows the empty `HotelStay` class with its generated test:

Listing 2. An empty class with its generated test

```
class HotelStay {
}

class HotelStayTests extends GroovyTestCase {
    void testSomething() {
    }
}
```

A `GroovyTestCase` is a thin Groovy facade over a JUnit 3.x unit test. If you're familiar with a JUnit `TestCase`, you already know how a `GroovyTestCase` works. In both cases, you test your code by asserting that it does what it claims to do. Various assertion methods are included with JUnit — `assertEquals`, `assertTrue`, and `assertNull`, to name just a few — that allow you to say programmatically, "I assert that this code does what it is supposed to do."

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Why JUnit 3.x instead of 4.x?

A `GroovyTestCase` is a JUnit 3.x `TestCase` for historical reasons. When Groovy 1.0 was released in January 2007, it supported the Java 1.4 language constructs. It ran on Java 1.4, 1.5, and 1.6 JVMs, but the language-level compatibility was specifically with Java 1.4.

The next major release of Groovy was 1.5, released in January 2008. Groovy 1.5 supports all of the Java 1.5 language features such as generics, static imports, the `for/in` loop, and — most important for this discussion — annotations. Groovy 1.5, however, still runs on the Java 1.4 JVM. The Groovy development team pledges that all 1.x versions of Groovy will maintain backwards compatibility with Java 1.4. When the Groovy 2.x branch is released (likely some time in late 2009 or 2010), it will drop Java 1.4 support.

So, what does all of this have to do with the version of JUnit wrapped by a `GroovyTestCase`? JUnit 4.x introduced annotations such as `@test`, `@before`, and `@after`. Although these new features are interesting, JUnit 3.x is still the basis of a `GroovyTestCase` for Java 1.4 backwards-compatibility reasons.

That said, nothing is stopping you from using JUnit 4.x on your own (see Resources for a link to relevant documentation on the Groovy site). And bringing in other testing frameworks that use annotations and Java 5 language features is equally possible (see Resources for an example of using TestNG with Groovy). Groovy is bytecode compatible with Java programming, so you can use any Java testing framework you'd like with Groovy.

Add the code in Listing 3 to grails-app/domain/HotelStay.groovy and test/integration/HotelStayTests.groovy:

Listing 3. A simple test

```
class HotelStay{
  String hotel
}

class HotelStayTests extends GroovyTestCase
{
  void testSomething(){
    HotelStay hs = new
HotelStay(hotel:"Sheraton")
    assertEquals "Sheraton", hs.hotel
  }
}
```

Listing 3, I realize, is exactly the sort of low-value Grails infrastructure testing I mentioned earlier. You should trust Grails to perform this action correctly, so it is a perfect example of the wrong type of test to write. But it helps serve this article's purpose by allowing you to write the simplest possible test and watch it run.

To run all tests, type `grails test-app`. To run this single test, type `grails`

© Copyright IBM Corporation 1994, 2008. All rights reserved.

test-app HotelStay. (Thanks to convention over configuration, the Tests suffix can be left off.) Whichever form of the command you type, you should see the output in Listing 4 at the command prompt. (Note: I've trimmed a lot of noise from the output to highlight the important features.)

Listing 4. Output from running a test

```
$ grails test-app
Environment set to test

No tests found in test/unit to execute ...

-------------------------------------------------------
Running 1 Integration Test...
Running test HotelStayTests...
                    testSomething...SUCCESS
Integration Tests Completed in 253ms
-------------------------------------------------------

Tests passed. View reports in
/src/trip-planner2/test/reports
```

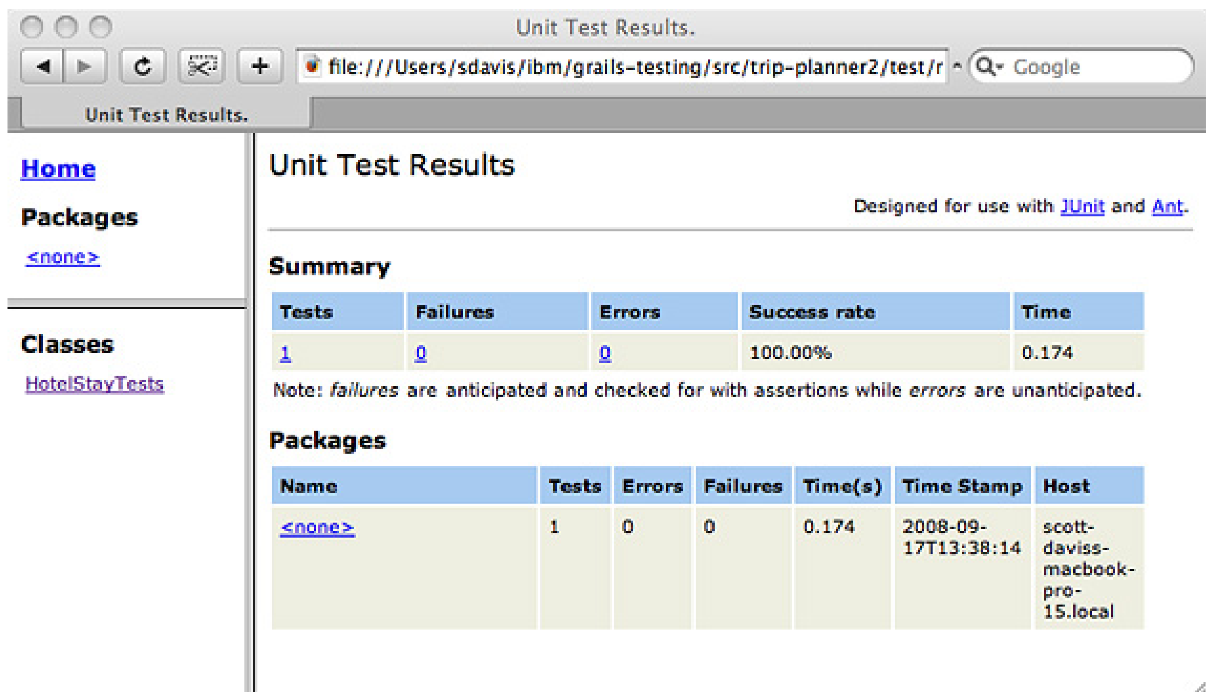
Four important things occurred:

1. As you can see, the environment is set to test. This means that the database settings in the test block in the conf/DataSource.groovy file are in play.
2. The scripts in test/unit are run. You haven't written any unit tests yet, so it should be no surprise that no unit tests were found.
3. The scripts in test/integration are run. You can see the output from the HotelStayTests.groovy script — with a big SUCCESS next to it.
4. The script points you to a set of reports.

If you open /src/trip-planner2/test/reports/html/index.html in a Web browser, you should see a report of all tests run, as shown in Figure 1:

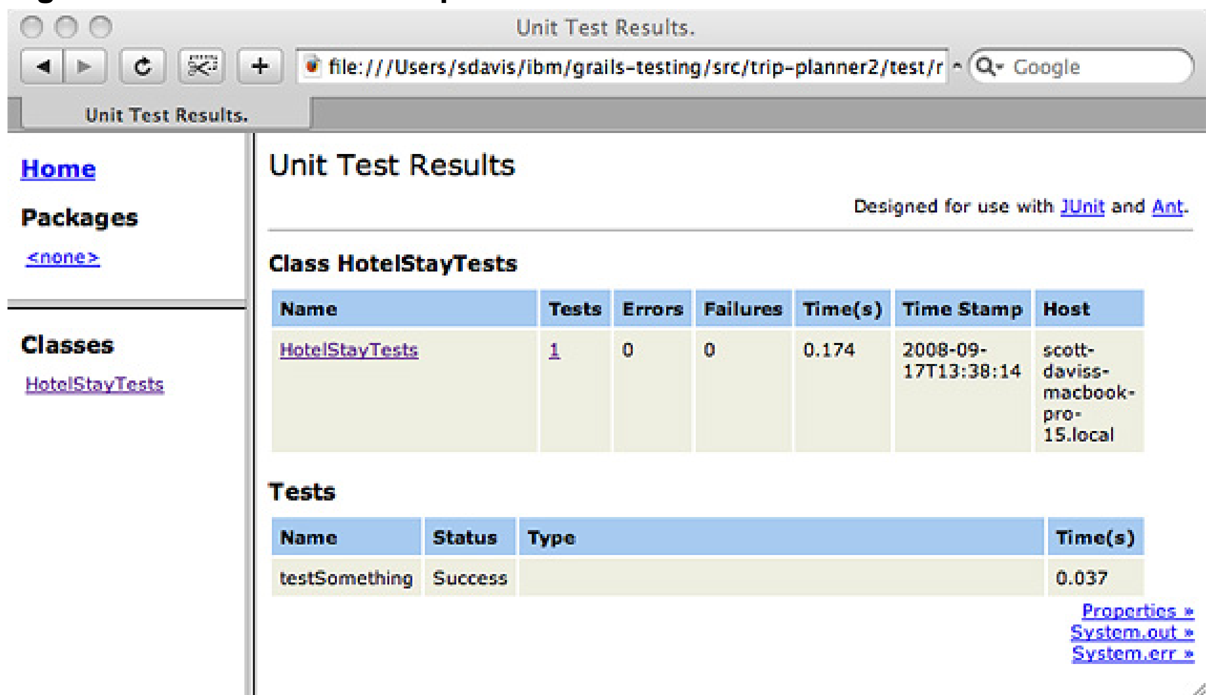
Figure 1. JUnit top-level summary report

© Copyright IBM Corporation 1994, 2008. All rights reserved.



If you click on the HotelStayTests link, you should see the `doSomething()` test, shown in Figure 2:

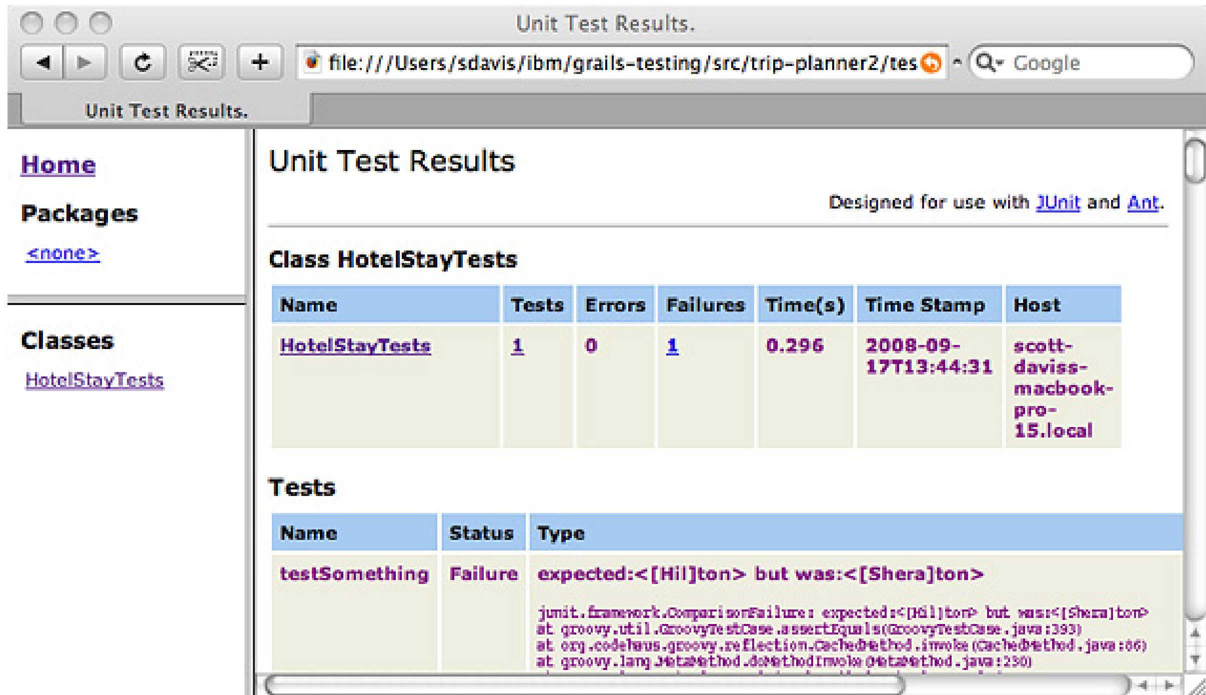
Figure 2. JUnit class-level report



If the test fails for some reason, both the command-prompt output and the HTML report (shown in Figure 3) will tell you so:

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Figure 3. A JUnit test that failed



Writing your first test worth writing

Now that the first simple test is working, here's a more realistic example of writing a test. Suppose that your `HotelStay` class has two fields: `Date checkIn` and `Date checkOut`. A user story says the output of the `toString` method should look like this: `Hilton (Wednesday to Sunday)`. Getting the dates in the right format is easy enough, thanks to the `java.text.SimpleDateFormat` class. You should write a test for this, but not to verify that `SimpleDateFormat` works correctly. Your test does two things: it validates that your `toString` method does what you claim it does, and it gives you proof that you satisfied the user story.

Unit tests are executable documentation

User requirements usually come across your desk as a document of some sort. It's your job as a developer to translate those requirements into working software.

The problem with the requirements document is that it is almost immediately out of date the moment actual software development begins. It isn't a "living document" that evolves as the software evolves. The word *artifact* is perfectly descriptive in this case — the document describes the initial, historical idea of what the software should do, not what the current implementation does.

Having a good, comprehensive set of tests in place does more than just keep your code bug free. The collateral benefit is that you also have a form of "executable documentation": a living, changing representation of the project requirements in code. If you map your

© Copyright IBM Corporation 1994, 2008. All rights reserved.

tests to requirements, you have something that you can share with your users. You have proof that both the code is sound and that the user requirement has been met. Couple this executable documentation with a continuous integration server such as CruiseControl (a server that continuously runs the tests over and over), and you now have a safeguard mechanism in place that ensures that new features don't introduce regressions into an otherwise good piece of software.

Behavior-Driven Development (BDD) fully embraces this idea of executable documentation. easyb is a BDD framework written in Groovy that allows you to write your tests as user requirements in a way that is readable to both users and developers (see Resources). If you have forward-thinking users who don't mind abandoning Microsoft® Word (for example), easyb can eliminate the need for the antiquated requirements document altogether. The requirements for the project can be executable from the very beginning.

Type the code in Listing 5 into HotelStay.groovy and HotelStayTests.groovy:

Listing 5. Using assertToString

```
import java.text.SimpleDateFormat
class HotelStay {
  String hotel
  Date checkIn
  Date checkOut

  String toString(){
    def sdf = new SimpleDateFormat("EEEE")
    "${hotel} (${sdf.format(checkIn)} to
${sdf.format(checkOut)})"
  }
}


import java.text.SimpleDateFormat
class HotelStayTests extends GroovyTestCase
{

    void testSomething(){...}

    void testToString() {
      def h = new HotelStay(hotel:"Hilton")
      def df = new
SimpleDateFormat("MM/dd/yyyy")
      h.checkIn = df.parse("10/1/2008")
      h.checkOut = df.parse("10/5/2008")
      println h
      assertToString h, "Hilton (Wednesday
to Sunday)"
    }
}
```

Type `grails test-app` to verify that this second test passes.

The `testToString` method uses one of the new assert methods — `assertToString` — that `GroovyTestCase` brings to the party. You certainly

© Copyright IBM Corporation 1994, 2008. All rights reserved.

could have achieved the same result by using the JUnit `assertEquals` method, but `assertToString` is a bit more expressive. The name of the test method and the final assertion leave little doubt what this test is trying to accomplish. (See Resources for a link to the full list of assertions that `GroovyTestCase` supports, including `assertArrayEquals`, `assertContains`, and `assertLength`.)

Adding a controller and views

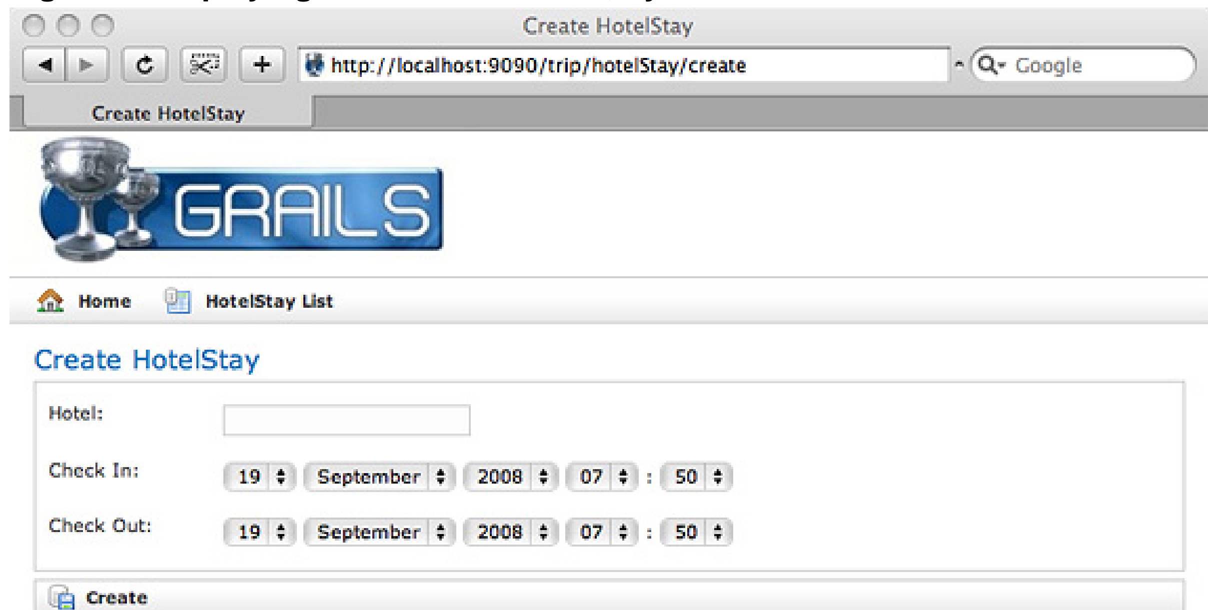
Up to this point, you have been interacting with the `HotelStay` domain class programmatically. Add a `HotelStayController`, shown in Listing 6, so that you can play around with that class in a Web browser too:

Listing 6. HotelStayController source

```
class HotelStayController {
  def scaffold = HotelStay
}
```

You should make a subtle UI tweak to the `create` form. By default, date fields include day, month, year, hours, and minutes, as shown in Figure 4:

Figure 4. Displaying both date and time, by default

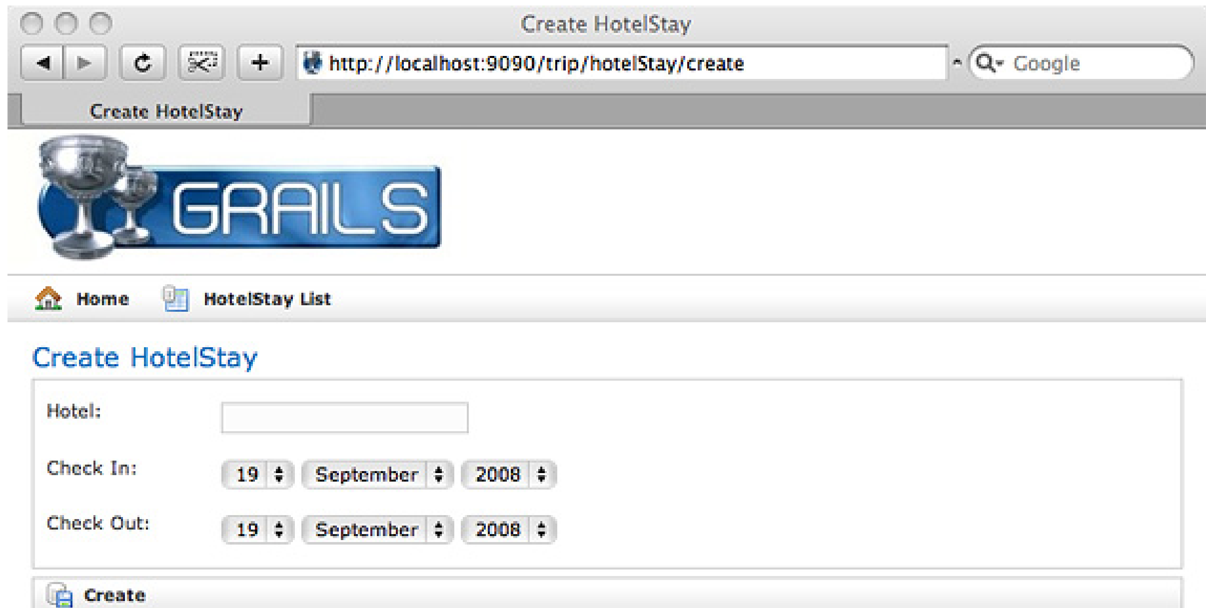


In this case, the date field's timestamp portion can safely be ignored. Type `grails generate-views HotelStay`. To create the modified UI shown in Figure 5, add `precision="day"` to the `<g:datePicker>` elements in both

© Copyright IBM Corporation 1994, 2008. All rights reserved.

views/hotelStay/create.gsp and views/hotelStay/edit.gsp:

Figure 5. Displaying only the date



Having a live, functional `HotelStay` running in the servlet container leads perfectly into the next discussion about testing: unit or integration?

Unit vs. integration tests

As I mentioned earlier, Grails supports two basic types of tests: unit and integration. There's no syntactical difference between the two — both are written as a `GroovyTestCase` using the same assertions. The difference is the semantics. A unit test is meant to test the class in isolation, whereas the integration test allows you to test the class in a full, running environment.

Quite frankly, if you want to write all of your Grails tests as integration tests, that's just fine with me. All of the Grails `create-*` commands generate corresponding integration tests, so most folks simply use what is already there. As you'll see in just a moment, most of the things you want to test require the full environment to be up and running anyway, so integration tests are a pretty good default.

If you have noncore Grails classes that you'd like to test, unit tests are perfectly fine. To create a unit test, type `grails create-unit-test MyTestUnit`. Because test scripts aren't created in different packages, you should give unique names to your unit and integration tests. If you don't, you'll be greeted by the nasty error message shown in Listing 7:

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Listing 7. Error message if you have an integration and unit test with the same name

```
The sources
/src/trip-planner2/test/integration/HotelStayTests.groovy and
   /src/trip-planner2/test/unit/HotelStayTests.groovy are
   containing both a class of the name HotelStayTests.
 @ line 3, column 1.
   class HotelStayTests extends GroovyTestCase {
   ^

1 error
```

Because the integration tests use the `Tests` suffix by default, I give all of my unit tests a `UnitTests` suffix to keep them unique.

Writing tests for simple validation error messages

Your next user story states that the hotel field cannot be left blank. This is simple enough to accomplish using the built-in Grails validation framework. Add a `static constraints` block to `HotelStay`, as shown in Listing 8:

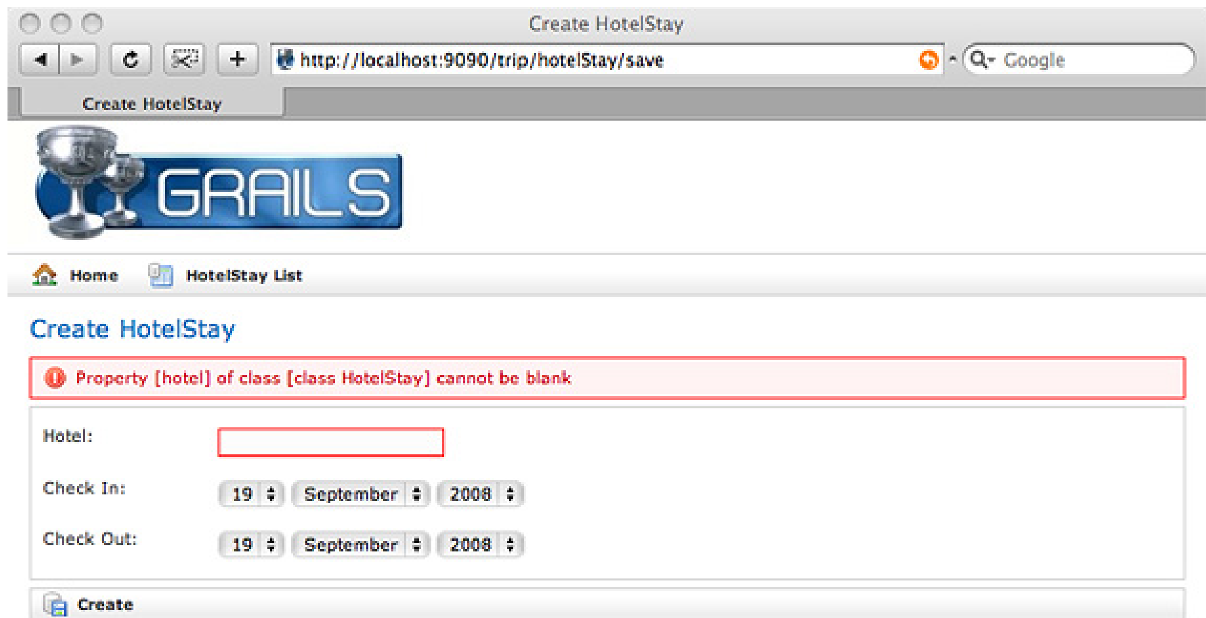
Listing 8. Adding a static constraints block to HotelStay

```
class HotelStay {
  static constraints = {
    hotel(blank:false)
    checkIn()
    checkOut()
  }

  String hotel
  Date checkIn
  Date checkOut

  //the rest of the class remains the same
}
```

Type `grails run-app`. If you try to create a `HotelStay` without filling in the hotel field, you should get the error message shown in Figure 6:

Figure 6. The default error message for blank fields

© Copyright IBM Corporation 1994, 2008. All rights reserved.



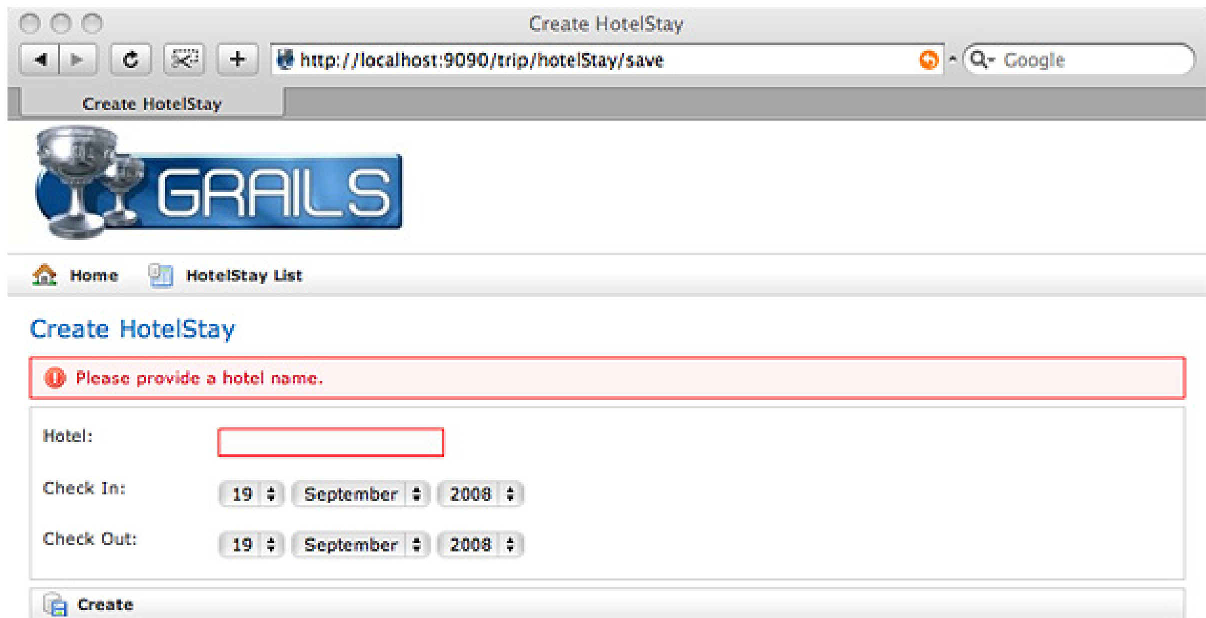
I'm sure that your users will be happy with the feature but less than impressed with the default error message. Assume that they modify the user story a bit: The hotel field cannot be left blank; if it is, the error message should say "Please provide a hotel name."

Now that you are adding some custom code — even if it is as simple as a custom `String` — it's time to add a test. (Of course, writing a test to prove that the user story is complete — even if it doesn't involve custom code — is also perfectly acceptable.)

Open grails-app/i18n/messages.properties and add `hotelStay.hotel.blank=Please provide a hotel name`. Try to submit a blank hotel in your browser. You should now see your custom message, as in Figure 7:

Figure 7. Displaying a custom validation error message

© Copyright IBM Corporation 1994, 2008. All rights reserved.



Add a new test to HotelStayTests.groovy to verify that the blank validation works, as shown in Listing 9:

Listing 9. Testing for validation errors

```
class HotelStayTests extends GroovyTestCase {
  void testBlankHotel(){
    def h = new HotelStay(hotel:"")
    assertFalse "there should be errors", h.validate()
    assertTrue "another way to check for errors after you call validate()", h.hasErrors()
  }

  //the rest of the tests remain unchanged
}
```

You've seen the `save()` method that gets added to each domain class in the generated controllers. I could have called `save()` here as well, but I don't really want to save the new class to the database. All I care about is whether or not the validation kicks in. The `validate()` method does the trick. If the validation fails, it returns `false`. If validation succeeds, it returns `true`.

The `hasErrors()` method is another valuable method for testing. After you call `save()` or `validate()`, `hasErrors()` allows you to check for validation errors after the fact.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Listing 10 is an expanded version of `testBlankHotel()` that introduces a couple of other valuable validation methods:

Listing 10. An advanced test for validation errors

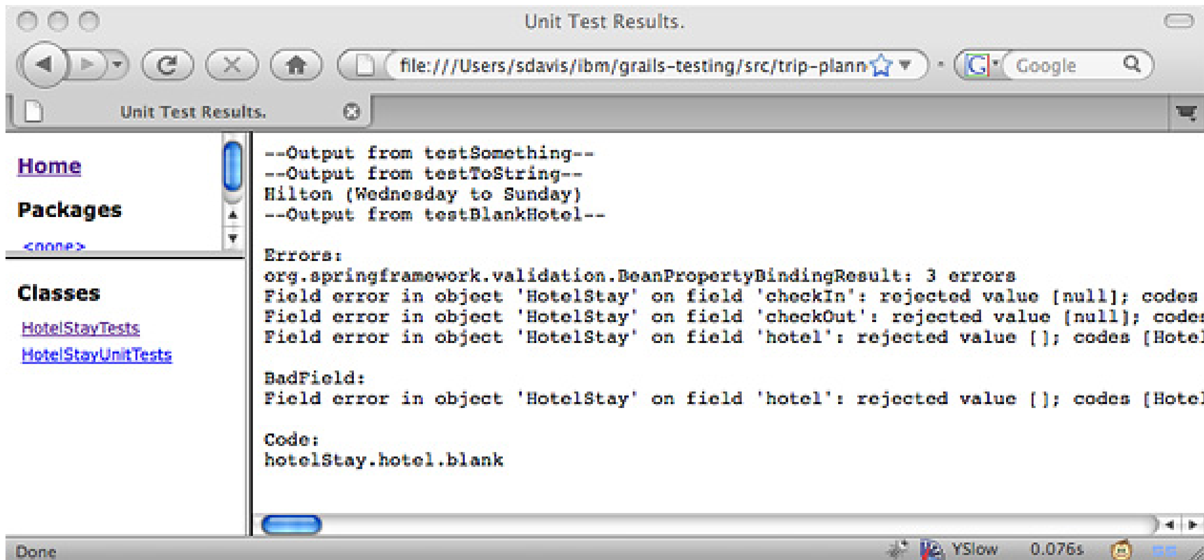
```
class HotelStayTests extends GroovyTestCase {
  void testBlankHotel(){
   def h = new HotelStay(hotel:"")
   assertFalse "there should be errors", h.validate()
   assertTrue "another way to check for errors after you call validate()", h.hasErrors()

   println "\nErrors:"
   println h.errors ?: "no errors found"

   def badField = h.errors.getFieldError('hotel')
   println "\nBadField:"
   println badField ?: "hotel wasn't a bad field"
   assertNotNull "I'm expecting to find an error on the hotel field", badField


   def code = badField?.codes.find {it == 'hotelStay.hotel.blank'}
   println "\nCode:"
   println code ?: "the blank hotel code wasn't found"
   assertNotNull "the blank hotel field should be the culprit", code
  }
}
```

Once you are sure that your class failed validation, you can call the `getErrors()` method (shortened to just `errors` here thanks to Groovy's shortcut getter syntax) to return an `org.springframework.validation.BeanPropertyBindingResult`. Just as GORM is a thin Groovy facade over Hibernate, Grails validation is simply Spring validation under the hood.

The results of the `println` calls won't show up at the command line, but they appear in the HTML report, shown in Figure 8:

Figure 8. Viewing println output from a test

© Copyright IBM Corporation 1994, 2008. All rights reserved.

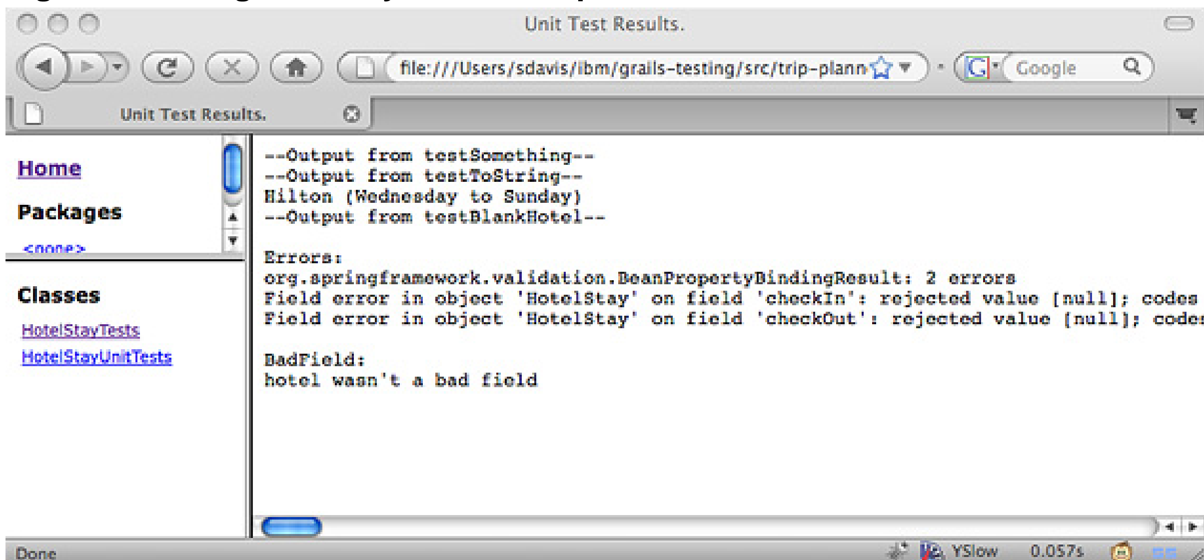

```
--Output from testSomething--
--Output from testToString--
Hilton (Wednesday to Sunday)
--Output from testBlankHotel--

Errors:
org.springframework.validation.BeanPropertyBindingResult: 3 errors
Field error in object 'HotelStay' on field 'checkIn': rejected value [null]; codes
Field error in object 'HotelStay' on field 'checkOut': rejected value [null]; codes
Field error in object 'HotelStay' on field 'hotel': rejected value []; codes [Hotel

BadField:
Field error in object 'HotelStay' on field 'hotel': rejected value []; codes [Hotel

Code:
hotelStay.hotel.blank
```

Click on the System.out link in the lower right corner of the `HotelStayTests` report.

The affectionately named Elvis operator (turn your head to one side — can you see his pompadour and two eyes?) in Listing 10 is a Groovy shortcut ternary operator. If the object to the left of the ?: is null, the value to the right is used instead.

Change the hotel field to `"Holiday Inn"` and rerun the tests. You should see the alternate Elvis output in the HTML report, as shown in Figure 9:

Figure 9. Seeing Elvis in your test output


```
--Output from testSomething--
--Output from testToString--
Hilton (Wednesday to Sunday)
--Output from testBlankHotel--

Errors:
org.springframework.validation.BeanPropertyBindingResult: 2 errors
Field error in object 'HotelStay' on field 'checkIn': rejected value [null]; codes
Field error in object 'HotelStay' on field 'checkOut': rejected value [null]; codes

BadField:
hotel wasn't a bad field
```

Don't forget to change the hotel field back to blank after you've seen Elvis — you don't want to leave broken tests in place.

You needn't be concerned that other validation errors are still showing up for

© Copyright IBM Corporation 1994, 2008. All rights reserved.

checkIn and checkOut. As far as this test is concerned, you can safely ignore them. This does, however, illustrate the point that you shouldn't simply test for the presence of errors — you should make sure that your *specific* error is being thrown.

Notice that I'm not asserting the exact text of the custom error message. Why did I care about matching the string last time (when testing the output of toString) and not this time? The custom output of the toString method was the whole point of the previous test. This time, I'm less concerned with Grails rendering the message properly than simply verifying that the validation code is being exercised. This just goes to show that testing is more of an art than a science. (If I wanted to validate the exact message output, I'd probably use a Web-tier testing tool such as Canoo WebTest or ThoughtWorks Selenium.)

Creating and testing custom validation

It's time to tackle the next user story. You need to ensure that checkOut dates don't happen before checkIn dates. Solving this requires you to write a custom validation. If you write it, you should test it.

Add the custom validation code in Listing 11 to the static constraints block:

Listing 11. A custom validation

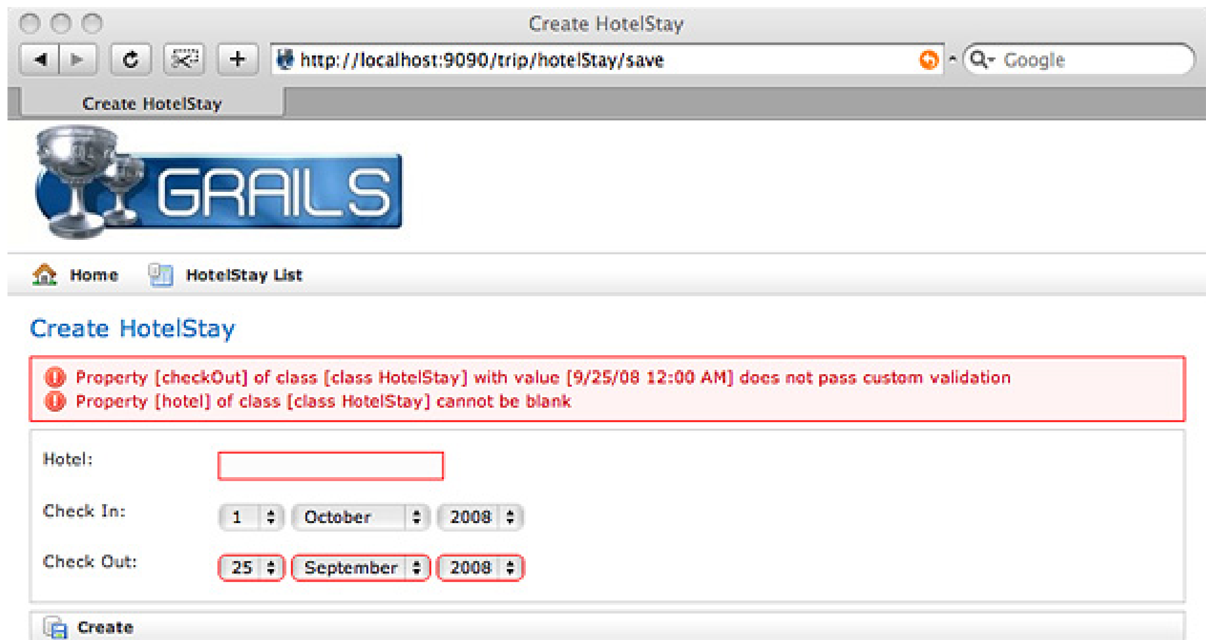
```
class HotelStay {
  static constraints = {
    hotel(blank:false)
    checkIn()
    checkOut(validator:{val, obj->
      return val.after(obj.checkIn)
    })
  }

  //the rest of the class remains the same
}
```

The val variable is the current field. The obj variable represents the current HotelStay instance. Groovy adds before() and after() methods to all Date objects, so this validation simply returns the results of the after() method call. If checkOut is after checkIn, the validation returns true. If not, it returns false and triggers an error.

Now type grails run-app. Verify that you can't create a new HotelStay with a checkOut date earlier than the checkIn date, as shown in Figure 10:

Figure 10. The default custom validation error message

© Copyright IBM Corporation 1994, 2008. All rights reserved.

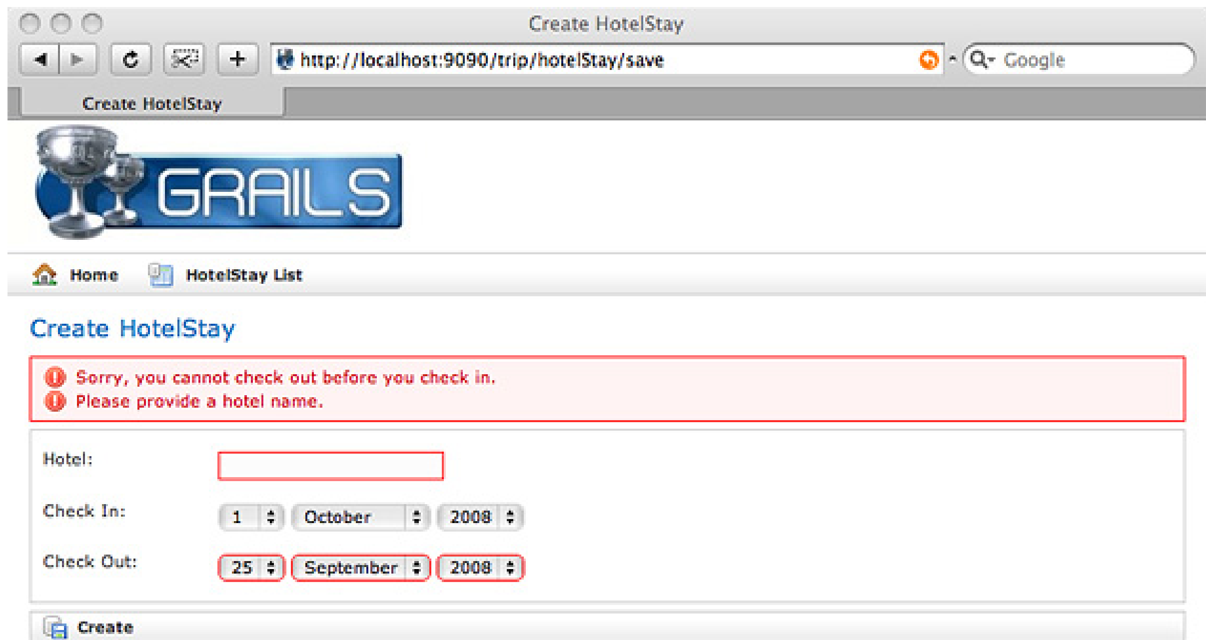


Open grails-app/i18n/messages.properties and add a custom validation message for the `checkOut` field: `hotelStay.checkOut.validator.invalid=Sorry, you cannot check out before you check in`.

Save the messages.properties file and try to save the malformed `HotelStay` again. You should see the error messages shown in Figure 11:

Figure 11. The custom validation error message

© Copyright IBM Corporation 1994, 2008. All rights reserved.



Now it's time to write the test, shown in Listing 12:

Listing 12. Testing your custom validation

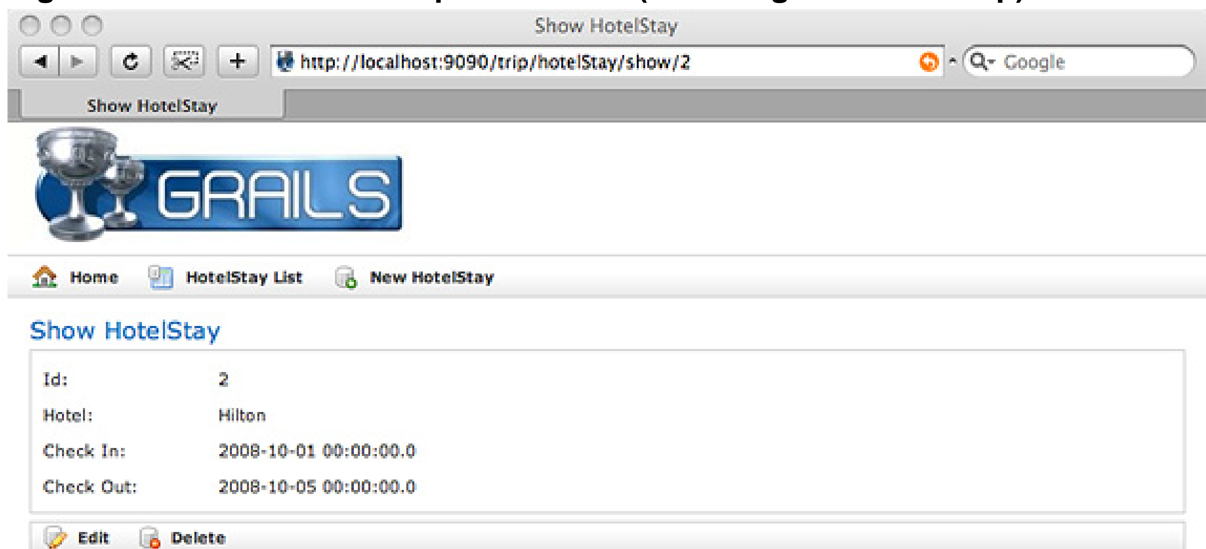
```
import java.text.SimpleDateFormat
class HotelStayTests extends GroovyTestCase {
  void testCheckOutIsNotBeforeCheckIn(){
    def h = new HotelStay(hotel:"Radisson")
    def df = new SimpleDateFormat("MM/dd/yyyy")
    h.checkIn = df.parse("10/15/2008")
    h.checkOut = df.parse("10/10/2008")

    assertFalse "there should be errors", h.validate()
    def badField = h.errors.getFieldError('checkOut')
    assertNotNull "I'm expecting to find an error on the checkOut field", badField
    def code = badField?.codes.find {it == 'hotelStay.checkOut.validator.invalid'}
    assertNotNull "the checkOut field should be the culprit", code
  }
}
```

Testing custom TagLibs

There's one last user story to tackle. You successfully got rid of the timestamp portion of `checkIn` and `checkOut` in the `create` and `edit` views, but it is still incorrect in the `list` and `show` views, as you can see in Figure 12:

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Figure 12. Default Grails output for dates (including the timestamp)



The easiest course of action is to define a new `TagLib`. Grails has a `<g:formatDate>` tag already defined that you could take advantage of, but creating a custom tag of your own is pretty easy in its own right. I want to end up with a `<g:customDateFormat>` tag that can be used in two ways.

One form of the `<g:customDateFormat>` tag wraps a `Date` and accepts a custom format attribute that accepts any valid `SimpleDateFormat` pattern:

```
<g:customDateFormat format="EEEE">${new Date()}</g:customDateFormat>
```

Because the most common use case is to return dates in the American "MM/dd/yyyy" format, I'd like that format to be the default if I don't specify anything else:

```
<g:customDateFormat>${new Date()}</g:customDateFormat>
```

Now that you understand the requirements of the user story, type `grails create-tag-lib Date`, as shown in Listing 13, to create a brand new DateTagLib.groovy file and a corresponding DateTagLibTests.groovy file:

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Listing 13. Creating a new TagLib

```
$ grails create-tag-lib Date
[copy] Copying 1 file to /src/trip-planner2/grails-app/taglib
Created TagLib for Date
[copy] Copying 1 file to /src/trip-planner2/test/integration
Created TagLibTests for Date
```

Add the code in Listing 14 to DateTagLib.groovy:

Listing 14. Creating your custom TagLib

```
import java.text.SimpleDateFormat

class DateTagLib {
  def customDateFormat = {attrs, body ->
    def b = attrs.body ?: body()
    def d = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss").parse(b)

    //if no format attribute is supplied, use this
    def pattern = attrs["format"] ?: "MM/dd/yyyy"
    out << new SimpleDateFormat(pattern).format(d)
  }
}
```

A `TagLib` takes in simple `String` values in the form of attributes and the body of the tag, and sends a `String` to the output stream. Since you're going to use this custom tag to wrap an unformatted `Date` field, you need two `SimpleDateFormat` objects. The input object reads in a `String` that matches the default format of the `Date.toString()` call. Once you've parsed that into a proper `Date` object, you can create a second `SimpleDateFormat` object to pass it back as a `String` in another format.

Wrap the `checkIn` and `checkOut` fields in list.gsp and show.gsp with your new `TagLib`, as shown in Listing 15:

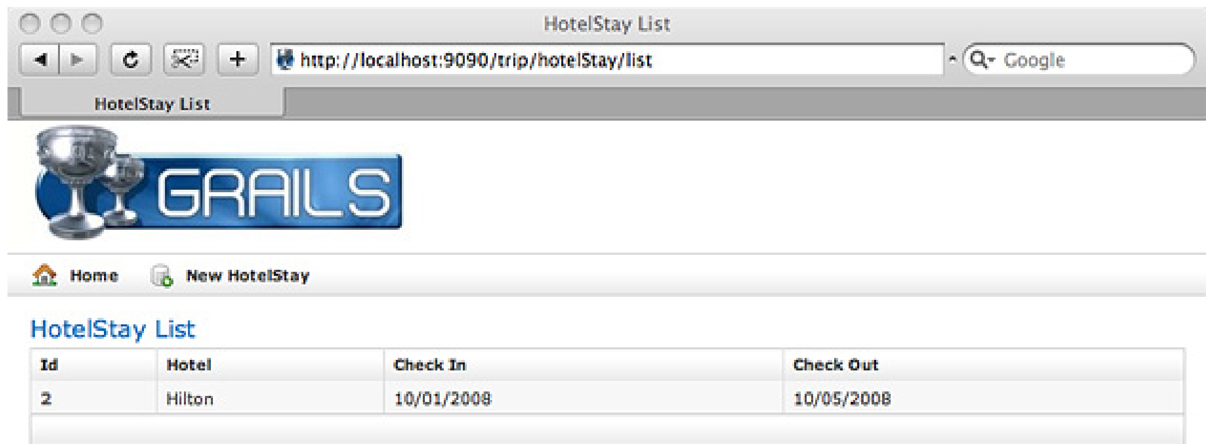
Listing 15. Using your custom TagLib

```
<g:customDateFormat>${fieldValue(bean:hotelStay, field:'checkIn')}</g:customDateFormat>
```

Type `grails run-app` and visit http://localhost:9090/trip/hotelStay/list to verify visually that your custom `TagLib` is in use, as shown in Figure 13:

Figure 13. Date output using a custom TagLib

© Copyright IBM Corporation 1994, 2008. All rights reserved.



Now, write the couple of tests in Listing 16 to verify that the `TagLib` works as expected:

Listing 16. Testing your custom TagLib

```
import java.text.SimpleDateFormat

class DateTagLibTests extends GroovyTestCase {
    void testNoFormat() {
      def output =
         new DateTagLib().customDateFormat(format:null, body:"2008-10-01 00:00:00.0")
      println "\ncustomDateFormat using the default format:"
      println output

      assertEquals "was the default format used?", "10/01/2008", output
    }

    void testCustomFormat() {
      def output =
         new DateTagLib().customDateFormat(format:"EEEE", body:"2008-10-01 00:00:00.0")
      assertEquals "was the custom format used?", "Wednesday", output
    }
}
```

Conclusion

Now that you have a few tests in place and see how easy it is to test your Grails

© Copyright IBM Corporation 1994, 2008. All rights reserved.

components, you can continue to be test-driven going forward. You'll feel more confident about the work you do, and if you match your tests to your user stories, you'll reap the benefits of having a rich set of executable documentation that is always up-to-date.

In the next article, I'll focus on JavaScript Object Notation (JSON). Grails has outstanding JSON support out of the box. You'll see how to generate JSON from a controller and how to consume it from a GSP. In the meantime, have fun mastering Grails.

© Copyright IBM Corporation 1994, 2008. All rights reserved.

Resources

Learn

- *Mastering Grails*: Read more in this series to gain a further understanding of Grails and all you can do with it.
- Grails: Visit the Grails Web site.
- Grails Framework Reference Documentation: The Grails bible.
- "Static Typing & Bureaucracy Redux" (Neil Ford, Meme Agora, May 2007): In this blog entry, Neil Ford says that dynamic languages let you do more testing in less time.
- Using JUnit 4 with Groovy: You can use JUnit 4 with Grails long as you are using Groovy 1.5+ and Java 5+. You'll find some examples here.
- Test'N'Groove: Test'N'Groove integrates Groovy and TestNG.
- Unit Testing: Read how Groovy facilitates JUnit testing. You'll find the full list of supported assertions here.
- "*In pursuit of code quality*: Adventures in behavior-driven development" (Andrew Glover, developerWorks, September 2007): Learn more about focusing on program behaviors rather than outcomes.
- *Groovy Recipes*: Learn more about Groovy and Grails in Scott Davis' latest book.
- *Practically Groovy*: This developerWorks series is dedicated to exploring the practical uses of Groovy and teaching you when and how to apply them successfully.
- Groovy: Learn more about Groovy at the project Web site.
- AboutGroovy.com: Keep up with the latest Groovy news and article links.
- Technology bookstore: Browse for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Grails: Download the latest Grails release.
- easyb: Download easyb, a behavior-driven development framework for the Java platform.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks

© Copyright IBM Corporation 1994, 2008. All rights reserved.

community.

About the author

Scott Davis
Scott Davis is an internationally recognized author, speaker, and software developer. His books include *Groovy Recipes: Greasing the Wheels of Java*, *GIS for Web Developers: Adding Where to Your Application*, *The Google Maps API*, and *JBoss At Work*.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

© Copyright IBM Corporation 1994, 2008. All rights reserved.