Extracted from:

Agile Web Development with Rails

Second Edition

This chapter was written by Mike Clark (http://clarkware.com). Mike is an independent consultant, author, and trainer. Most important, he's a programmer. He helps teams build better software faster using agile practices. With an extensive background in J2EE and test-driven development, he's currently putting his experience to work on Rails projects.

Chapter 13
Task T: Testing

In short order we've developed a respectable web-based shopping cart application. Along the way, we got rapid feedback by writing a little code and then punching buttons in a web browser (with our customer by our side) to see whether the application behaved as we expected. This testing strategy works for about the first hour you're developing a Rails application, but soon thereafter you've amassed enough features that manual testing just doesn't scale. Your fingers grow tired and your mind goes numb every time you have to punch all the buttons, so you don't test very often, if ever.

Then one day you make a minor change and it breaks a few features, but you don't realize it until the customer phones up to say she's no longer happy. If that weren't bad enough, it takes you hours to figure out exactly what went wrong. You made an innocent change over here, but it broke stuff way over there. By the time you've unraveled the mystery, the customer has found herself a new best programmer.

It doesn't have to be this way. There's a practical alternative to this madness: write tests!

In this chapter, we'll write automated tests for the application we all know and love—the Depot application.[1] Ideally, we'd write these tests incrementally to get little confidence boosts along the way. Thus, we're calling this Task T, because we should be doing testing all the time. You'll find listings of the code from this chapter starting on page 671.

13.1 Tests Baked Right In

With all the fast and loose coding we've been doing while building Depot, it would be easy to assume that Rails treats testing as an afterthought. Nothing

1. We'll be testing the stock, vanilla version of Depot. If you've made modifications (perhaps by trying some of the playtime exercises at the ends of the chapters), you might have to make adjustments.

could be further from the truth. One of the real joys of the Rails framework is that it has support for testing baked right in from the start of every project. Indeed, from the moment you create a new application using the rails command, Rails starts generating a test infrastructure for you.

We haven't written a lick of test code for the Depot application, but if you look in the top-level directory of that project, you'll notice a subdirectory called test. Inside this directory you'll see five directories and a helper file.

```
depot> ls -p test
fixtures/       integration/    test_helper.rb
functional/     mocks/          unit/
```

So our first decision—where to put tests—has already been made for us. The rails command creates the full test directory structure.

By convention, Rails calls things that test models *unit tests*, things that test a single action in a controller *functional tests*, and things that test the flow through one or more controllers *integration tests*. Let's take a peek inside the unit and functional subdirectories to see what's already there.

```
depot> ls test/unit
line_item_test.rb   order_test.rb    product_test.rb   user_test.rb

depot> ls test/functional
admin_controller_test.rb      login_controller_test.rb
info_controller_test.rb       store_controller_test.rb
```

Look at that! Rails has already created files to hold the unit tests for the models and the functional tests for the controllers we created earlier with the generate script. This is a good start, but Rails can help us only so much. It puts us on the right path, letting us focus on writing good tests. We'll start back where the data lives and then move up closer to where the user lives.

13.2 Unit Testing of Models

The first model we created for the Depot application way back on page 70 was Product. Let's see what kind of test goodies Rails generated inside the file test/unit/product_test.rb when we generated that model.

Download depot_r/test/unit/product_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class ProductTest < Test::Unit::TestCase

  fixtures :products

  def test_truth
    assert true
  end
end
```

OK, our second decision—how to write tests—has already been made for us. The fact that ProductTest is a subclass of the Test::Unit::TestCase class tells us that Rails generates tests based on the Test::Unit framework that comes preinstalled with Ruby. This is good news because it means if we've already been testing our Ruby programs with Test::Unit tests (and why wouldn't we be?), then we can build on that knowledge to test Rails applications. If you're new to Test::Unit, don't worry. We'll take it slow.

Now, what's with the generated code inside of the test case? Rails generated two things for us. The first is the following line of code.

```
fixtures :products
```

There's a lot of magic behind this line of code—it allows us to prepopulate our database with just the right test data—and we'll be talking about it in depth in a minute.

The second thing Rails generated is the method test_truth. If you're familiar with Test::Unit you'll know all about this method. The fact that its name starts with *test* means that it will run as a test by the testing framework. And the assert line in there is an actual test. It isn't much of one, though—all it does is test that true is true. Clearly, this is a placeholder, but it's an important one, because it lets us see that all the testing infrastructure is in place. So, let's try to run this test class.

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
EE
Finished in 0.559942 seconds.

1) Error:
test_truth(ProductTest):
MysqlError: Unknown database 'depot_test'
... a whole bunch of tracing...
1 tests, 0 assertions, 0 failures, 2 errors
```

Guess it wasn't the truth, after all. The test didn't just fail, it exploded! Thankfully, it leaves us a clue—it couldn't find a database called depot_test. Hmph.

A Database Just for Tests

Remember back on page 71 when we created the development database for the Depot application? We called it depot_development. That's because that's the default name Rails gave it in the database.yml file in the config directory. If you look in that configuration file again, you'll notice Rails actually created a configuration for three separate databases.

- depot_development will be our development database. All of our programming work will be done here.

- depot_test is a test database.
- depot_production is the production database. Our application will use this when we put it online.

So far, we've been doing all our work in the development database. Now that we're running tests, though, Rails needs to use the test database, and we haven't created one yet.

Let's remedy that now. As we're using the MySQL database, we'll again use mysqladmin to create the database.

```
depot> mysqladmin -u root create depot_test
```

Now let's run the test again.

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
E
Finished in 0.06429 seconds.

1) Error:
test_truth(ProductTest):
ActiveRecord::StatementInvalid: MysqlError:
Table 'depot_test.products' doesn't exist: DELETE FROM products
1 tests, 0 assertions, 0 failures, 1 errors
```

Oh, dear! Not much better than last time. But the error is different. Now it's complaining that we don't have a products table in our test database. And indeed we don't: right now all we have is an empty schema. Let's populate the test database schema to match that of our development database. We'll use the db:test:prepare task to copy the schema across.

```
depot> rake db:test:prepare
```

Now we have a database containing a schema. Let's try our unit test one more time.

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
.
Finished in 0.085795 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

OK, that looks better. See how having the stub test wasn't really pointless? It let us get our test environment all set up. Now that it is, let's get on with some real tests.

But, before we do, I have a confession. I wanted to show you how to set up the test database schema manually, and then how to run tests directly. However, there's a short cut. You can use the rake task

```
depot> rake test:units
```

This task does two things: it copies the schema into the test database, and then it runs all the tests in the test/unit directory. Go ahead—try it now.

If I want to run all my tests, I use this rake task. If I want to work on just a particular test, I'll use Ruby to run just that file.

A Real Unit Test

We've added a fair amount of code to the Product model since Rails first generated it. Some of that code handles validation.

Download depot_r/app/models/product.rb

```
validates_presence_of :title, :description, :image_url
validates_numericality_of :price
validates_uniqueness_of :title
validates_format_of :image_url,
                    :with    => %r{\.(gif|jpg|png)$}i,
                    :message => "must be a URL for a GIF, JPG, or PNG image"
protected

def validate
  errors.add(:price, "should be at least 0.01") if price.nil? ||  price < 0.01
end
```

How do we know this validation is working? Let's test it. First, if we create a product with no attributes set, we'll expect it to be invalid and for there to be an error associated with each field. We can use the model's valid? method to see whether it validates, and we can use the invalid? method of the error list to see if there's an error associated with a particular attribute.

Now that we know *what* to test, we need to know *how* to tell the test framework whether our code passes or fails. We do that using *assertions*. An assertion is simply a method call that tells the framework what we expect to be true. The simplest assertion is the method assert, which expects its argument to be true. If it is, nothing special happens. However, if the argument to assert is false, the assertion fails. The framework will output a message and will stop executing the test method containing the failure. In our case, we expect that an empty Product model will not pass validation, so we can express that expectation by asserting that it isn't valid.

```
assert !product.valid?
```

Let's write the full test.

Download depot_r/test/unit/product_test.rb

```
def test_invalid_with_empty_attributes
  product = Product.new
  assert !product.valid?
  assert product.errors.invalid?(:title)
  assert product.errors.invalid?(:description)
  assert product.errors.invalid?(:price)
  assert product.errors.invalid?(:image_url)
end
```

When we run the test case, we'll now see two tests executed (the original test_truth method and our new test method).

```
depot> ruby test/unit/product_test.rb
Loaded suite test/unit/product_test
Started
..
Finished in 0.092314 seconds.
2 tests, 6 assertions, 0 failures, 0 errors
```

Sure enough, the validation kicked in, and all our assertions passed.

Clearly at this point we can dig deeper and exercise individual validations. Let's look at just three of the many possible tests. First, we'll check that the validation of the price works the way we expect.

Download depot_r/test/unit/product_test.rb

```
def test_positive_price
  product = Product.new(:title       => "My Book Title",
                        :description => "yyy",
                        :image_url   => "zzz.jpg")
  product.price = -1
  assert !product.valid?
  assert_equal "should be at least 0.01", product.errors.on(:price)

  product.price = 0
  assert !product.valid?
  assert_equal "should be at least 0.01", product.errors.on(:price)

  product.price = 1
  assert product.valid?
end
```

In this code we create a new product and then try setting its price to -1, 0, and +1, validating the product each time. If our model is working, the first two should be invalid, and we verify the error message associated with the price attribute is what we expect. The last price is acceptable, so we assert that the model is now valid. (Some folks would put these three tests into three separate test methods—that's perfectly reasonable.)

Next, we'll test that we're validating the image URL ends with one of .gif, .jpg, or .png.

Download depot_r/test/unit/product_test.rb

```
def test_image_url
  ok = %w{ fred.gif fred.jpg fred.png FRED.JPG FRED.Jpg
           http://a.b.c/x/y/z/fred.gif }
  bad = %w{ fred.doc fred.gif/more fred.gif.more }

  ok.each do |name|
    product = Product.new(:title       => "My Book Title",
                          :description => "yyy",
                          :price       => 1,
                          :image_url   => name)
    assert product.valid?, product.errors.full_messages
  end

  bad.each do |name|
    product = Product.new(:title => "My Book Title", :description => "yyy", :price => 1,
                          :image_url => name)
    assert !product.valid?, "saving #{name}"
  end
end
```

Here we've mixed things up a bit. Rather than write out the nine separate tests, we've used a couple of loops, one to check the cases we expect to pass validation, the second to try cases we expect to fail. You'll notice that we've also added an extra parameter to our assert method calls. All of the testing assertions accept an optional trailing parameter containing a string. This will be written along with the error message if the assertion fails and can be useful for diagnosing what went wrong.

Finally, our model contains a validation that checks that all the product titles in the database are unique. To test this one, we're going to need to store product data in the database.

One way to do this would be to have a test create a product, save it, then create another product with the same title, and try to save it too. This would clearly work. But there's a more idiomatic way—we can use Rails *fixtures*.

Test Fixtures

In the world of testing, a *fixture* is an environment in which you can run a test. If you're testing a circuit board, for example, you might mount it in a test fixture that provides it with the power and inputs needed to drive the function to be tested.

In the world of Rails, a test fixture is simply a specification of the initial contents of a model (or models) under test. If, for example, we want to ensure that our products table starts off with known data at the start of every unit test, we can specify those contents in a fixture, and Rails will take care of the rest.

You specify fixture data in files in the test/fixtures directory. These files contain test data in either Comma-Separated Value (CSV) or YAML format. For our tests we'll use YAML, the preferred format. Each YAML fixture file contains the data for a single model. The name of the fixture file is significant; the base name of the file must match the name of a database table. Because we need some data for a Product model, which is stored in the products table, we'll add it to the file called products.yml. Rails already created this fixture file when we first created the model.

```
# Read about fixtures at ...
one:
  id: 1
two:
  id: 2
```

The fixture file contains an entry for each row that we want to insert into the database. Each row is given a name. In the case of the Rails-generated fixture, the rows are named *first* and *another*. This name has no significance as far as the database is concerned—it is not inserted into the row data. Instead, as we'll see shortly, the name gives us a convenient way to reference test data inside our test code.

Inside each entry you'll see an indented list of attribute name/value pairs. In the Rails-generated fixture only the id attribute is set. Although it isn't obvious in print, you must use spaces, not tabs, at the start of each of the data lines, and all the lines for a row must have the same indentation. Finally, you need to make sure the names of the columns are correct in each entry; a mismatch with the database column names may cause a hard-to-track-down exception.

Let's replace the dummy data in the fixture file with something we can use to test our product model. We'll start with a single book. (Note that you have to include the id column in test fixtures.)

Download depot_r/test/fixtures/products.yml

```
ruby_book:
  id:          1
  title:       Programming Ruby
  description: Dummy description
  price:       1234
  image_url:   ruby.png
```

Now that we have a fixture file, we want Rails to load up the test data into the products table when we run the unit test. And, in fact, Rails is already doing this, thanks to the following line in ProductTest.

Download depot_r/test/unit/product_test.rb

```
fixtures :products
```

The fixtures directive loads the fixture data corresponding to the given model name into the corresponding database table before each test method in the test case is run. The name of the fixture file determines the table that is loaded, so using :products will cause the products.yml fixture file to be used.

David Says...

Picking Good Fixture Names

Just like the names of variables in general, you want to keep the names of fixtures as self-explanatory as possible. This increases the readability of the tests when you're asserting that product(:valid_order_for_fred) is indeed Fred's valid order. It also makes it a lot easier to remember which fixture you're supposed to test against without having to look up p1 or order4. The more fixtures you get, the more important it is to pick good fixture names. So, starting early keeps you happy later.

But what to do with fixtures that can't easily get a self-explanatory name like valid_order_for_fred? Pick natural names that you have an easier time associating to a role. For example, instead of using order1, use christmas_order. Instead of customer1, use fred. Once you get into the habit of natural names, you'll soon be weaving a nice little story about how fred is paying for his christmas_order with his invalid_credit_card first, then paying his valid_credit_card, and finally choosing to ship it all off to aunt_mary.

Association-based stories are key to remembering large worlds of fixtures with ease.

Let's say that again another way. In the case of our ProductTest class, adding the fixtures directive means that the products table will be emptied out and then populated with the single row for the Ruby book before each test method is run. Each test method gets a freshly initialized table in the test database.

Using Fixture Data

Now we know how to get fixture data into the database, we need to find ways of using it in our tests.

Clearly, one way would be to use the finder methods in the model to read the data. However, Rails makes it easier than that. For each fixture it loads into a test, Rails defines a method with the same name as the fixture. You can use this method to access preloaded model objects containing the fixture data: simply pass it the name of the row as defined in the YAML fixture file, and it'll return a model object containing that row's data. In the case of our product data, calling products(:ruby_book) returns a Product model containing the data we defined in the fixture. Let's use that to test the validation of unique product titles.

Download depot_r/test/unit/product_test.rb

```
def test_unique_title
  product = Product.new(:title       => products(:ruby_book).title,
                        :description => "yyy",
                        :price       => 1,
                        :image_url   => "fred.gif")

  assert !product.save
  assert_equal "has already been taken", product.errors.on(:title)
end
```

The test assumes that the database already includes a row for the Ruby book. It gets the title of that existing row using

```
products(:ruby_book).title
```

It then creates a new Product model, setting its title to that existing title. It asserts that attempting to save this model fails and that the title attribute has the correct error associated with it.

If you want to avoid using a hard-coded string for the Active Record error, you can compare the response against its built-in error message table.

Download depot_r/test/unit/product_test.rb

```
def test_unique_title1
  product = Product.new(:title       => products(:ruby_book).title,
                        :description => "yyy",
                        :price       => 1,
                        :image_url   => "fred.gif")

  assert !product.save
  assert_equal ActiveRecord::Errors.default_error_messages[:taken],
               product.errors.on(:title)
end
```

(To find a list of these built-in error messages, look for the file validations.rb within the Active Record gem. Figure 13.1, on the next page contains a list of the errors at the time this chapter was written, but it may well have changed by the time you're reading it.)

Testing the Cart

Our Cart class contains some business logic. When we add a product to a cart, it checks to see whether that product is already in the cart's list of items. If so, it increments the quantity of that item; if not, it adds a new item for that product. Let's write some tests for this functionality.

The Rails generate command created source files to hold the unit tests for the database-backed models in our application. But what about the cart? We created the Cart class by hand, and we don't have a file in the unit test directory corresponding to it. Nil desperandum! Let's just create one. We'll simply copy

```
@@default_error_messages = {
  :inclusion    => "is not included in the list",
  :exclusion    => "is reserved",
  :invalid      => "is invalid",
  :confirmation => "doesn't match confirmation",
  :accepted     => "must be accepted",
  :empty        => "can't be empty",
  :blank        => "can't be blank",
  :too_long     => "is too long (maximum is %d characters)",
  :too_short    => "is too short (minimum is %d characters)",
  :wrong_length => "is the wrong length (should be %d characters)",
  :taken        => "has already been taken",
  :not_a_number => "is not a number"
}
```

Figure 13.1: Standard Active Record Validation Messages

the boilerplate from another test file into a new cart_test.rb file (remembering to rename the class to CartTest).

Download depot_r/test/unit/cart_test.rb

```ruby
require File.dirname(__FILE__) + '/../test_helper'

class CartTest < Test::Unit::TestCase
  fixtures :products
end
```

Notice that we've included the existing products fixture into this test. This is common practice: we'll often want to share test data among multiple test cases. In this case the cart tests will need access to product data because we'll be adding products to the cart.

Because we'll need to test adding different products to our cart, we'll need to add at least one more product to our products.yml fixture. The complete file now looks like this.

Download depot_r/test/fixtures/products.yml

```yaml
ruby_book:
  id:          1
  title:       Programming Ruby
  description: Dummy description
  price:       1234
  image_url:   ruby.png

rails_book:
  id:          2
  title:       Agile Web Development with Rails
  description: Dummy description
  price:       2345
  image_url:   rails.png
```

Let's start by seeing what happens when we add a Ruby book and a Rails book to our cart. We'd expect to end up with a cart containing two items. The total price of items in the cart should be the Ruby book's price plus the Rails book's price.

Download depot_r/test/unit/cart_test.rb

```
def test_add_unique_products
  cart = Cart.new
  rails_book = products(:rails_book)
  ruby_book  = products(:ruby_book)
  cart.add_product rails_book
  cart.add_product ruby_book
  assert_equal 2, cart.items.size
  assert_equal rails_book.price + ruby_book.price, cart.total_price
end
```

Let's run the test.

```
depot> ruby test/unit/cart_test.rb
Loaded suite test/unit/cart_test
Started
.
Finished in 0.12138 seconds.

1 tests, 2 assertions, 0 failures, 0 errors
```

So far, so good. Let's write a second test, this time adding two Rails books to the cart. Now we should see just one item in the cart, but with a quantity of 2.

Download depot_r/test/unit/cart_test.rb

```
def test_add_duplicate_product
  cart = Cart.new
  rails_book = products(:rails_book)
  cart.add_product rails_book
  cart.add_product rails_book
  assert_equal 2*rails_book.price, cart.total_price
  assert_equal 1, cart.items.size
  assert_equal 2, cart.items[0].quantity
end
```

We're starting to see a little bit of duplication creeping into these tests. Both create a new cart, and both set up local variables as shortcuts for the fixture data. Luckily, the Ruby unit testing framework gives us a convenient way of setting up a common environment for each test method. If you add a method named setup in a test case, it will be run before each test method—the setup method sets up the environment for each test. We can therefore use it to set up some instance variables to be used by the tests.

Download depot_r/test/unit/cart_test1.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class CartTest < Test::Unit::TestCase
  fixtures :products

  def setup
    @cart  = Cart.new
    @rails = products(:rails_book)
    @ruby  = products(:ruby_book)
  end

  def test_add_unique_products
    @cart.add_product @rails
    @cart.add_product @ruby
    assert_equal 2, @cart.items.size
    assert_equal @rails.price + @ruby.price, @cart.total_price
  end

  def test_add_duplicate_product
    @cart.add_product @rails
    @cart.add_product @rails
    assert_equal 2*@rails.price, @cart.total_price
    assert_equal 1, @cart.items.size
    assert_equal 2, @cart.items[0].quantity
  end
end
```

Is this kind of setup useful for this particular test? It could be argued either way. But, as we'll see when we look at functional testing, the setup method can play a critical role in keeping tests consistent.

Unit Testing Support

As you write your unit tests, you'll probably end up using most of the assertions in the list that follows.

assert(boolean,*message*)

Fails if boolean is false or nil.

```
assert(User.find_by_name("dave"), "user 'dave' is missing")
```

assert_equal(expected, actual,*message*)
assert_not_equal(expected, actual,*message*)

Fails unless expected and actual are/are not equal.

```
assert_equal(3, Product.count)
assert_not_equal(0, User.count, "no users in database")
```

assert_nil(object,*message*)
assert_not_nil(object,*message*)

Fails unless object is/is not nil.

```
assert_nil(User.find_by_name("willard"))
assert_not_nil(User.find_by_name("henry"))
```

assert_in_delta(expected_float, actual_float, delta,*message*)

Fails unless the two floating-point numbers are within delta of each other. Preferred over assert_equal because floats are inexact.

```
assert_in_delta(1.33, line_item.discount, 0.005)
```

assert_raise(Exception, ...,*message*) { block... }
assert_nothing_raised(Exception, ...,*message*) { block... }

Fails unless the block raises/does not raise one of the listed exceptions.

```
assert_raise(ActiveRecord::RecordNotFound) { Product.find(bad_id) }
```

assert_match(pattern, string,*message*)
assert_no_match(pattern, string,*message*)

Fails unless string is matched/not matched by the regular expression in pattern. If pattern is a string, then it is interpreted literally—no regular expression metacharacters are honored.

```
assert_match(/flower/i, user.town)
assert_match("bang*flash", user.company_name)
```

assert_valid(activerecord_object)

Fails unless the supplied Active Record object is valid—that is, it passes its validations. If validation fails, the errors are reported as part of the assertion failure message.

```
user = Account.new(:name => "dave", :email => 'secret@pragprog.com')
assert_valid(user)
```

flunk(*message*)

Fails unconditionally.

```
unless user.valid? || account.valid?
  flunk("One of user or account should be valid")
end
```

Ruby's unit testing framework provides even more assertions, but these tend to be used infrequently when testing Rails applications, so we won't discuss them here. You'll find them in the documentation for Test::Unit.[2] Additionally, Rails provides support for testing an application's routing. We describe that starting on page 424.

13.3 Functional Testing of Controllers

Controllers direct the show. They receive incoming web requests (typically user input), interact with models to gather application state, and then respond by causing the appropriate view to display something to the user. So when we're testing controllers, we're making sure that a given request is answered with an

2. At http://ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit/Assertions.html, for example

appropriate response. We still need models, but we already have them covered with unit tests.

Rails calls something that tests a single controller a *functional test*. The Depot application has four controllers, each with a number of actions. There's a lot here that we could test, but we'll work our way through some of the high points. Let's start where the user starts—logging in.

Login

It wouldn't be good if anybody could come along and administer the Depot. Although we may not have a sophisticated security system, we'd like to make sure that the login controller at least keeps out the riffraff.

Because the LoginController was created with the generate controller script, Rails has a test stub waiting for us in the test/functional directory.

Download depot_r/test/functional/login_controller_test.rb

```
require File.dirname(__FILE__) + '/../test_helper'
require 'login_controller'

# Re-raise errors caught by the controller.
class LoginController; def rescue_action(e) raise e end; end

class LoginControllerTest < Test::Unit::TestCase
  def setup
    @controller = LoginController.new
    @request    = ActionController::TestRequest.new
    @response   = ActionController::TestResponse.new
  end

  # Replace this with your real tests.
  def test_truth
    assert true
  end
end
```

The key to functional tests is the setup method. It initializes three instance variables needed by every functional test.

- @controller contains an instance of the controller under test.
- @request contains a request object. In a running, live application, the request object contains all the details and data from an incoming request. It contains the HTTP header information, POST or GET data, and so on. In a test environment, we use a special test version of the request object that can be initialized without needing a real, incoming HTTP request.
- @response contains a response object. Although we haven't seen response objects as we've been writing our application, we've been using them. Every time we send a request back to a browser, Rails is populating

a response object behind the scenes. Templates render their data into a response object, the status codes we want to return are recorded in response objects, and so on. After our application finishes processing a request, Rails takes the information in the response object and uses it to send a response back to the client.

The request and response objects are crucial to the operation of our functional tests—using them means we don't have to fire up a real web server to run controller tests. That is, functional tests don't necessarily need a web server, a network, or a client.

Index: For Admins Only

Great, now let's write our first controller test—a test that simply "hits" the index page.

Download depot_r/test/functional/login_controller_test.rb

```
def test_index
  get :index
  assert_response :success
end
```

The get method, a convenience method loaded by the test helper, simulates a web request (think HTTP GET) to the index action of the LoginController and captures the response. The assert_response method then checks whether the response was successful.

OK, let's see what happens when we run the test. We'll use the -n option to specify the name of a particular test method that we want to run.

```
depot> ruby test/functional/login_controller_test.rb -n test_index
Loaded suite test/functional/login_controller_test
Started
F
Finished in 0.239281 seconds.

1) Failure:
test_index(LoginControllerTest) [test/functional/login_controller_test.rb:23]:
Expected response to be a <:success>, but was <302>
```

That seemed simple enough, so what happened? A response code of 302 means the request was redirected, so it's not considered a success. But why did it redirect? Well, because that's the way we designed the LoginController. It uses a before filter to intercept calls to actions that aren't available to users without an administrative login.

Download depot_r/app/controllers/login_controller.rb

```
before_filter :authorize, :except => :login
```

The before filter makes sure that the authorize method is run before the index action is run.

Download depot_r/app/controllers/application.rb

```
class ApplicationController < ActionController::Base

  # Pick a unique cookie name to distinguish our session data from others'
  session :session_key => '_depot_session_id'

  private

  def authorize
    unless User.find_by_id(session[:user_id])
      flash[:notice] = "Please log in"
      redirect_to(:controller => "login", :action => "login")
    end
  end

end
```

Since we haven't logged in, a valid user isn't in the session, so the request gets redirected to the login action. According to authorize, the resulting page should include a *flash* notice telling us that we need to log in. OK, so let's rewrite the functional test to capture that flow.

Download depot_r/test/functional/login_controller_test.rb

```
def test_index_without_user
  get :index
  assert_redirected_to :action => "login"
  assert_equal "Please log in", flash[:notice]
end
```

This time when we request the index action, we expect to get redirected to the login action and see a flash notice generated by the view.

```
depot> ruby test/functional/login_controller_test.rb
Loaded suite test/functional/login_controller_test
Started
.
Finished in 0.0604571 seconds.
1 tests, 3 assertions, 0 failures, 0 errors
```

Indeed, we get what we expect.[3] Now we know the administrator-only actions are off limits until a user has logged in (the *before filter* is working). Let's try looking at the index page if we have a valid user.

Recall that the application stores the id of the currently logged in user into the session, indexed by the :user_id key. So, to fake out a logged in user, we just

3. With one small exception. Our test method contains two assertions, but the console log shows three assertions passed. That's because the assert_redirected_to method uses two low-level assertions internally.

need to set a user id into the session before issuing the index request. Our only problem now is knowing what to use for a user id.

We can't just stick a random number in there, because the application controller's authorize method fetches the user row from the database based on its value. It looks as if we'll need to populate the users table with something valid. And that gives us an excuse to look at dynamic fixtures.

Dynamic Fixtures

We'll create a users.yml test fixture to add a row to the users table. We'll call the user "dave."

```
dave:
  id:   1
  name: dave
  salt: NaCl
  hashed_password: ???
```

All goes well until the hashed_password line. What should we use as a value? In the real table, it is calculated using the encrypted_password method in the user class. This takes a clear-text password and a salt value and creates an SHA1 hash value.

Now, one approach would be to crank up script/console and invoke that method manually. We could then copy the value returned by the method, pasting it into the fixture file. That'd work, but it's a bit obscure, and our tests might break if we change the password generation mechanism. Wouldn't it be nice if we could use our application's code to generate the hashed password as data is loaded into the database? Well, have a look at the following.

Download depot_r/test/fixtures/users.yml

```
<% SALT = "NaCl" unless defined?(SALT) %>

dave:
  id:   1
  name: dave
  salt: <%= SALT %>
  hashed_password: <%= User.encrypted_password('secret', SALT) %>
```

The syntax on the hashed_password line should look familiar: the <%=...%> directive is the same one we use to substitute values into templates. It turns out that Rails supports these substitutions in test fixtures. That's why we call them dynamic.

Now we're ready to test the index action again. We have to remember to add the fixtures directive to the login controller test class.

```
fixtures :users
```

And then we write the test method.

Download depot_r/test/functional/login_controller_test.rb

```ruby
def test_index_with_user
  get :index, {}, { :user_id => users(:dave).id }
  assert_response :success
  assert_template "index"
end
```

The key concept here is the call to the get method. Notice that we added a couple of new parameters after the action name. Parameter two is an empty hash—this represents the HTTP parameters to be passed to the action. Parameter three is a hash that's used to populate the session data. This is where we use our user fixture, setting the session entry :user_id to be our test user's id. Our test then asserts that we had a successful response (not a redirection) and that the action rendered the index template. (We'll look at all these assertions in more depth shortly.)

Logging In

Now that we have a user in the test database, let's see whether we can log in as that user. If we were using a browser, we'd navigate to the login form, enter our user name and password, and then submit the fields to the login action of the login controller. We'd expect to get redirected to the index listing and to have the session contain the id of our test user neatly tucked inside. Here's how we do this in a functional test.

Download depot_r/test/functional/login_controller_test.rb

```ruby
def test_login
  dave = users(:dave)
  post :login, :name => dave.name, :password => 'secret'
  assert_redirected_to :action => "index"
  assert_equal dave.id, session[:user_id]
end
```

Here we used a post method to simulate entering form data and passed the name and password form field values as parameters.

What happens if we try to log in with an invalid password?

Download depot_r/test/functional/login_controller_test.rb

```ruby
def test_bad_password
  dave = users(:dave)
  post :login, :name => dave.name, :password => 'wrong'
  assert_template "login"
end
```

As expected, rather than getting redirected to the index listing, our test user sees the login form again.

Functional Testing Conveniences

That was a brisk tour through how to write a functional test for a controller. Along the way, we used a number of support methods and assertions included with Rails that make your testing life easier. Before we go much further, let's look at some of the Rails-specific conveniences for testing controllers.

HTTP Request Methods

The methods get, post, put, delete, and head are used to simulate an incoming HTTP request method of the same name. They invoke the given action and make the response available to the test code.

Each of these methods takes the same four parameters. Let's take a look at get, as an example.

get(action, parameters = nil, session = nil, flash = nil)

Executes an HTTP GET request for the given action. The @response object will be set on return. The parameters are as follows.

- action: The action of the controller being requested
- parameters: An optional hash of request parameters
- session: An optional hash of session variables
- flash: An optional hash of flash messages

Examples:

```
get :index
get :add_to_cart, :id => products(:ruby_book).id
get :add_to_cart, { :id => products(:ruby_book).id },
                  { :session_key => 'session_value'}, { :message => "Success!" }
```

You'll often want to post form data within a function test. To do this, you'll need to know that the data is returned as a hash nested inside the params hash. The key for this subhash is the name given when you created the form. Inside the subhash are key/value pairs corresponding to the fields in the form. So, to post a form to the edit action containing User model data, where the data contains a name and an age, you could use

```
post :edit, :user => { :name => "dave", :age => "24" }
```

You can simulate an xml_http_request using

xhr(method, action, parameters, session, flash)
xml_http_request(method, action, parameters, session, flash)

Simulates an xml_http_request from a JavaScript client to the server. The first parameter will be :post or :get. The remaining parameters are identical to those passed to the get method described previously.

```
xhr(:post, :add_to_cart, :id => 11)
```

Assertions

In addition to the standard assertions we listed back on page 198, additional functional test assertions are available after executing a request.

assert_dom_equal(expected_html, actual_html,*message*)
assert_dom_not_equal(expected_html, actual_html,*message*)

Compare two strings containing HTML, succeeding if the two are represented/not represented by the same document object model. Because the assertion compares a normalized version of both strings, it is fragile in the face of application changes. Consider using assert_select instead.

```
expected = "<html><body><h1>User Unknown</h1></body></html>"
assert_dom_equal(expected, @response.body)
```

assert_response(type,*message*)

Asserts that the response is a numeric HTTP status or one of the following symbols. These symbols can cover a range of response codes (so :redirect means a status of 300–399).

- :success
- :redirect
- :missing
- :error

Examples:

```
assert_response :success
assert_response 200
```

assert_redirected_to(options,*message*)

Asserts that the redirection options passed in match those of the redirect called in the last action. You can also pass a simple string, which is compared to the URL generated by the redirection.

Examples:

```
assert_redirected_to :controller => 'login'
assert_redirected_to :controller => 'login', :action => 'index'
assert_redirected_to "http://my.host/index.html"
```

assert_template(expected,*message*)

Asserts that the request was rendered with the specified template file.

Examples:

```
assert_template 'store/index'
```

assert_select(...)

See Section 13.3, *Testing Response Content*, on page 208.

assert_tag(...)

Deprecated in favor of assert_select.

Rails has some additional assertions to test the routing component of your controllers. We discuss these in Section 20.2, *Testing Routing*, on page 424.

Variables

After a request has been executed, functional tests can make assertions using the values in the following variables.

assigns(key=nil)

Instance variables that were assigned in the last action.

```
assert_not_nil assigns["items"]
```

The assigns hash must be given strings as index references. For example, assigns[:items] will not work because the key is a symbol. To use symbols as keys, use a method call instead of an index reference.

```
assert_not_nil assigns(:items)
```

We can test that a controller action found three orders with

```
assert_equal 3, assigns(:orders).size
```

session

A hash of objects in the session.

```
assert_equal 2, session[:cart].items.size
```

flash

A hash of flash objects currently in the session.

```
assert_equal "Danger!", flash[:notice]
```

cookies

A hash of cookies being sent to the user.

```
assert_equal "Fred", cookies[:name]
```

redirect_to_url

The full URL that the previous action redirected to.

```
assert_equal "http://test.host/login", redirect_to_url
```

Functional Testing Helpers

Rails provides the following helper methods in functional tests.

find_tag(conditions)

Finds a tag in the response, using the same conditions as assert_tag.

```
get :index
tag = find_tag :tag => "form",
               :attributes => { :action => "/store/add_to_cart/993" }
assert_equal "post", tag.attributes["method"]
```

This is probably better written using assert_select.

find_all_tag(conditions)

Returns an array of tags meeting the given conditions.

follow_redirect

If the preceding action generated a redirect, this method follows it by issuing a get request. Functional tests can follow redirects only to their own controller.

```
post :add_to_cart, :id => 123
assert_redirect :action => :index
follow_redirect
assert_response :success
```

fixture_file_upload(path, mime_type)

Create the MIME-encoded content that would normally be uploaded by a browser <input type="file"...> field. Use this to set the corresponding form parameter in a post request.

```
post :report_bug,
     :screenshot => fixture_file_upload("screen.png", "image/png")
```

Testing Response Content

Rails 1.2 introduced a new assertion, assert_select, which allows you to dig into the structure and content of the responses returned by your application. (It replaces assert_tag, which is now deprecated.) For example, a functional test could verify that the response contained a title element containing the text "Pragprog Books Online Store" with the assertion

```
assert_select "title", "Pragprog Books Online Store"
```

For the more adventurous, the following tests that the response contains a <*div*> with the id cart. Within that <*div*> there must be a table containing three rows. The last <*td*> in the row with the class total-line must have the content $57.70.

```
assert_select "div#cart" do
  assert_select "table" do
    assert_select "tr", :count => 3
    assert_select "tr.total-line td:last-of-type", "$57.70"
  end
end
```

This is clearly powerful stuff. Let's spend some time looking at it.

assert_select is built around Assaf Arkin's HTML::Selector library. This library allows you to navigate a well-formed HTML document using a syntax drawn heavily from Cascading Style Sheets selectors. On top of the selectors, Rails layers the ability to perform a set of tests on the resulting nodesets. Let's start by looking at the selector syntax.

Selectors

Selector syntax is complex—probably more complex than regular expressions. However, its similarity to CSS selector syntax means that you should be able to find many examples on the Web if the brief summary that follows is too condensed. In the description that follows, we'll borrow the W3C terminology for describing selectors.[4]

A full selector is called a *selector chain*. A selector chain is a combination of one or more *simple selectors*. Let's start by looking at the simple selectors.

Simple Selectors

A simple selector consists of an optional *type selector*, followed by any number of *class selectors*, *id selectors*, *attribute selectors*, or *pseudoclasses*.

A type selector is simply the name of a tag in your document. For example, the type selector

```
p
```

matches all *<p>* tags in your document. (It's worth emphasizing the word *all*—selectors work with sets of document nodes.)

If you omit the type selector, all nodes in the document are selected.

A type selector may be qualified with *class selectors*, *id selectors*, *attribute selectors*, or *pseudoclasses*. Each qualifier whittles down the set of nodes that are selected. Class and ID selectors are easy.

```
p#some-id       # selects the paragraph with id="some-id"

p.some-class    # selects paragraph(s) with class="some-class"
```

Attribute selectors appear between square brackets. The syntax is

```
p[name]           # paragraphs with an attribute name=
p[name=value]     # paragraphs with an attribute name=value
p[name^=string]   # ... name=value, value starts with 'string'
p[name$=string]   # ... name=value, value ends with 'string'
p[name*=string]   # ... name=value, value must contain 'string'
p[name~=string]   # ... name=value, value must contain 'string'
                  # as a space-separated word
p[name|=string]   # ... name=value, value starts 'string'
                  # followed by a space
```

Let's look at some examples.

```
p[class=warning]     # all paragraphs with class="warning"

tr[id=total]         # the table row with id="total"
```

4. http://www.w3.org/TR/REC-CSS2/selector.html

```
table[cellpadding]   # all table tags with a cellpadding attribute

div[class*=error]    # all div tags with a class attribute
                     # containing the text error

p[secret][class=shh] # all p tags with both a secret attribute
                     # and a class="shh" attribute

[class=error]        # all tags with class="error"
```

The class and id selectors are shortcuts for class= and id=.

```
p#some-id           # same as p[id=some-id]
p.some-class        # same as p[class=some-class]
```

Chained Selectors
You can combine multiple simple selectors to create chained selectors. These allow you to describe the relationship between elements. In the descriptions that follow, *sel_1*, *sel_2*, and so on, represent simple selectors.

sel_1 ␣ *sel_2s*
: All *sel_2*s that have a *sel_1* as an ancestor. (The selectors are separated by one or more spaces.)

sel_1 > *sel_2s*
: All *sel_2*s that have *sel_1* as a parent. Thus:

```
table  td       # will match all td tags inside table tags
table > td      # won't match in well-formed HTML,
                # as td tags have tr tags as parents
```

sel_1 + *sel_2s*
: Selects all *sel_2*s that immediately follow *sel_1*s. Note that “follow” means that the two selectors describe peer nodes, not parent/child nodes.

```
td.price + td.total     # select all td nodes with class="total"
                        # that follow a <td class="price">
```

sel_1 ~ *sel_2s*
: Selects all *sel_2*s that follow *sel_1*s.

```
div#title ~ p   # all the p tags that follow a
                # <div id="title">
```

sel_1, *sel_2s*
: Selects all elements that are selected by *sel_1* or *sel_2*.

```
p.warn, p.error     # all paragraphs with a class of
                    # warn or error
```

Pseudoclasses

Pseudo-classes typically allow you to select elements based on their position (although there are some exceptions). They are all prefixed with a colon.

:root

Selects only the root element. Sometimes useful when testing an XML response.

```
order:root         # only returns a selection if the
                   # root of the response is <order>
```

sel:empty

Selects only if *sel* has neither children nor text content.

```
div#error:empty    # selects the node <div id="error">
                   # only if it is empty
```

sel_1 sel_2:only-child

Selects the nodes that are the only children of *sel_1* nodes.

```
div :only-child   # select the child nodes of divs that
                  # have only one child
```

sel_1 sel_2:first-child

Selects all *sel_2* nodes that are the first children of *sel_1* nodes.

```
table tr:first-child  # the first row from each table
```

sel_1 sel_2:last-child

Selects all *sel_2* nodes that are the last children of *sel_1* nodes.

```
table tr:last-child  # the last row from each table
```

sel_1 sel_2:nth-child(*n*)

Selects all *sel_2* nodes that are the n^{th} child of *sel_1* nodes, where *n* counts from 1. Contrast this with nth-of-type, described later.

```
table tr:nth-child(2)  # the second row of every table

div p:nth-child(2)     # the second element of each div
                       # if that element is a <p>
```

sel_1 sel_2:nth-last-child(*n*)

Selects all *sel_2* nodes that are the n^{th} child of *sel_1* nodes, counting from the end.

```
table tr:nth-last-child(2)  # the second to last row in every table
```

sel_1 sel_2:only-of-type

Selects all *sel_2* nodes that are the only children of *sel_1* nodes. (That is, the *sel_1* node may have multiple children but only one of type *sel_2*.)

```
div p:only-of-type  # all the paragraphs in divs that
                    # contain just one paragraph
```

sel_1 sel_2:first-of-type
: Selects the first node of type *sel_2* whose parents are *sel_1* nodes.

```
div.warn p:first-of-type  # the first paragraph in <div class="warn">
```

sel_1 sel_2:last-of-type
: Selects the last node of type *sel_2* whose parents are *sel_1* nodes.

```
div.warn p:last-of-type  # the last paragraph in <div class="warn">
```

sel_1 sel_2:nth-of-type(*n*)
: Selects all *sel_2* nodes that are the n^{th} child of *sel_1* nodes, but only counting nodes whose type matches *sel_2*.

```
div p:nth-of-type(2)  # the second paragraph of each div
```

sel_1 sel_2:nth-last-of-type(*n*)
: Selects all *sel_2* nodes that are the n^{th} child of *sel_1* nodes, counting from the end, but only counting nodes whose type matches *sel_2*.

```
div p:nth-last-of-type(2)  # the second to last paragraph of each div
```

The numeric parameter to the nth-*xxx* selectors can be of the form:

d (a number)
: Count *d* nodes.

*a*n+*d* (nodes from groups)
: Divide the child nodes into groups of *a*, and then select the d^{th} node from each group.

```
div#story p:nth-child(3n+1)     # every third paragraph of
                                # the div with id="story"
```

-*a*n+*d* (nodes from groups)
: Divide the child nodes into groups of *a*, and then select the first node of up to *d* groups. (Yes, this is a strange syntax.)

```
div#story p:nth-child(-n+2)     # The first two paragraphs
```

odd (odd-numbered nodes)
even (even-numbered nodes)
: Alternating child nodes.

```
div#story p:nth-child(odd)      # paragraphs 1, 3, 5, ...
div#story p:nth-child(even)     # paragraphs 2, 4, 6, ...
```

Finally, you can invert the sense of any selector.

:not(*sel*)
: Selects all nodes that are not selected by *sel*.

```
div :not(p)          # all the non-paragraph nodes of all divs
```

Now we know how to select nodes in the response, let's see how to write assertions to test the response's content.

Response-Oriented Assertions

The assert_select assertion can be used within functional and integration tests. At its simplest it takes a selector. The assertion passes if at least one node in the response matches, and it fails if no nodes match.

```
assert_select "title"      # does our response contain a <title> tag

                           # and a <div class="cart"> with a
                           # child <div id="cart-title">
assert_select "div.cart > div#cart-title"
```

As well as simply testing for the presence of selected nodes, you can compare their content with a string or regular expression. The assertion passes only if all selected nodes equal the string or match the regular expression.

```
assert_select "title",  "Pragprog Online Book Store"
assert_select "title",  /Online/
```

If instead you pass a number or a Ruby range, the assert passes if the number of nodes is equal to the number or falls within the range.

```
assert_select "title", 1                  # must be just one title element
assert_select "div#main div.entry", 1..10  # one to 10 entries on a page
```

Passing false as the second parameter is equivalent to passing zero: the assertion succeeds if no nodes are selected.

You can also pass a hash after the selector, allowing you to test multiple conditions. For example, to test that there is exactly one title node and that node matches the regular expression /pragprog/, you could use

```
assert_select "title", :count => 1, :text => /pragprog/
```

The hash may contain the following keys:

:text =>$S \mid R$	Either a string or a regular expression, which must match the contents of the node.
:count =>n	Exactly n nodes must have been selected.
:minimum =>n	At least n nodes must have been selected.
:maximum =>n	At most n nodes must have been selected.

Nesting Select Assertions

Once assert_select has chosen a set of nodes and passed any tests associated with those nodes, you may want to perform additional tests within that nodeset. For example, we started this section with a test that checked that the page contained a <*div*> with an id of cart. This <*div*> should contain a table which