881-4 CH04.qxd 8/19/07 3:11 PM Page 37

CHAPTER 4

Store Administration

This chapter will walk you through the first part of the Shoplet application. It will be a fairly detailed look at most of the code necessary, and as such will give you a good overview of what's required to create a Rails application from scratch.

The Shoplet application is a basic web shop. It lets customers look through different products in an inventory, partitioned by type and categorized. Customers can add products to their shopping cart and later check out, and order the products in question. Payment is by billing address, because implementing the handling for a payment gateway is out of this chapter's scope.

The second part of the Shoplet application is the administrative user interface. This part is protected by username and password, and it's possible for currently authenticated users to add new ones. The administrative tasks that we'll create include adding, removing, and editing products, and handling orders.

Because most of the administrative user interface is easy to create from the basis of scaffolding, that's what I'll show you first. The customer part of the Shoplet system is covered in the first part of the next chapter.

As mentioned in the last chapter, I'll show some test code, but not nearly as much as would be necessary in a real application. The same thing is true with regards to validation in models. I'll add enough of these to show how it works, but not all parts of the model will be validated in a way that would be necessary in a deployed application.

Also note that the instructions from here on require that you have set up your environment in the way described in Chapter 2. The commands will be shown in a Linux environment, but should be trivial to translate into the Windows or Mac OS counterparts.

Creating a New Rails Application

The first step to create a new Rails application is to go to the directory where the new application will live, and then execute the `rails` command with the name of the application to create as a single argument. In this case, you'll do it like this:

```
jruby -S rails shoplet
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 38

This command generates 20 to 30 lines of output, telling you what files and directories it created. On my system the first few lines look like this:

```
create
create  app/controllers
create  app/helpers
create  app/models
create  app/views/layouts
create  config/environments
create  components
create  db
```

What has happened is that the `rails` script has created a new subdirectory called shoplet in the current directory, which contains the standard structure of a Rails application (described further in Chapter 3). With just one small adjustment, this will be a functional (but not very useful) Rails application that can be run with a web server and executed by visiting the correct address with a browser.

The next step requires that you have set up a database, as described in Chapter 2. For our purposes, I'll assume it's a MySQL database on `localhost`. If this isn't the way you've done it, you'll have to translate the instructions into your circumstances.

First of all, you need to create some databases in your MySQL installation. You also need to add users with the correct privileges to access and create tables in the database. The script in Listing 4-1 works fine on any MySQL installation.

Listing 4-1. *create_shoplet_db.sql*

```
CREATE DATABASE shoplet_dev;
CREATE DATABASE shoplet_test;
CREATE DATABASE shoplet_prod;
GRANT ALL PRIVILEGES ON shoplet_dev.* TO shoplet_dev@'%'
          IDENTIFIED BY 'shoplet';
GRANT ALL PRIVILEGES ON shoplet_test.* TO shoplet_test@'%'
          IDENTIFIED BY 'shoplet';
GRANT ALL PRIVILEGES ON shoplet_prod.* TO shoplet_prod@'%'
          IDENTIFIED BY 'shoplet';
FLUSH PRIVILEGES;
```

This creates three new databases, with three corresponding users who all have the same password. Of course, you should modify this script for your own needs, but the three databases named `dev`, `test`, and `prod` should exist in some form, because this is one of the things that makes Rails a joy to work with.

After you've created the databases, all you need to do is to configure the file `config/database.yml`. There are three top-level entries in this file, called `development`, `test`, and `production`. When you write in this file, you need to make sure that you don't accidentally get a tab character in. Try to indent using spaces instead. With the preceding settings, and the comments removed (lines that begin with # are comments), the new Shoplet `database.yml` should look like this:

Listing 4-2. *config/database.yml*

```
development:
  adapter: jdbc
  driver: com.mysql.jdbc.Driver
  url: jdbc:mysql://localhost/shoplet_dev
  username: shoplet_dev
  password: shoplet

test:
  adapter: jdbc
  driver: com.mysql.jdbc.Driver
  url: jdbc:mysql://localhost/shoplet_test
  username: shoplet_test
  password: shoplet

production:
  adapter: jdbc
  driver: com.mysql.jdbc.Driver
  url: jdbc:mysql://localhost/shoplet_prod
  username: shoplet_prod
  password: shoplet
```

Now you've configured the database, which means you're just about set to test your Rails application for the first time. The only thing missing is a small change to the file `config/environment.rb`. Just before the line that begins with `Rails::Initializer`, you need to add this line:

```
require 'jdbc_adapter'
```

This makes sure that Rails will be able to use AR-JDBC as the database provider.

RAILS ENVIRONMENTS

The Rails framework consistently uses the concept of environments to control various configuration options. These decide which runtime parameters are set, how they interact, and a few other things. There are three different runtime environments in standard Rails. These are called development, test, and production. It's important to keep in mind that the three environments have separate databases and don't share session information.

development

The development runtime environment is the default environment where you run migrations and start a web server. This is where you'll most likely spend your time. This environment tries to make the development process as easy as possible. It does this in a number of different ways. The most important one is that it reloads most of your code on each request. This means you can change the validators in your model, and the next time you hit the browser, the changes will be there, without having to restart the web server or Rails. In the same way, you can change controllers and views and never have to restart.

881-4 CH04.qxd 8/19/07 3:11 PM Page 40

Another difference from what you might expect is that caching isn't turned on. The error messages are also informative, and tell things that you should never let a user see. A third thing is that the development environment enables the breakpoint server automatically. See Chapter 3 for more information about the breakpointer.

test

The most important thing to remember about the test environment is that you never use it explicitly. Whenever you run any tests in Rails, the test environment is used, but you shouldn't specify it explicitly when starting a Rails console or web server. To make sure everything is as Rails expects it to be when running tests, the database in question is always re-created before each test run. So, don't store any important data here.

Information is cached when testing, but error messages are still on the more talkative side.

production

The production environment is where your application should run in production, simply. Caching happens, the production database is used, and error messages are specifically written for users, not developers.

staging

The staging runtime environment doesn't come with Rails. It's something that many in the community wish were there, though. Simply enough, in a real customer-developer relationship, you often find yourself in the situation that you want to show the customer what the web site will look like, but you don't want it to run against the production database. There might also be other things you don't want to have it using in this situation. So, what you do is simply add a new environment, copied from the production one. This is simple to do; you just copy the file `config/environments/production.rb` to `config/environments/staging.rb`, and add a new entry in `config/databases.yml` called `staging`, which describes the database settings for this environment.

The staging environment should be as close as possible to the real production environment, without using the same resources. In some situations there is no production database to clobber, and in that case you won't need the staging environment. However, in most real deployment scenarios there are always things going wrong, and this solution works well.

Running with Mongrel

When the time comes to test your application, you'll need to start it up in some way. For now, we won't look at the more advanced deployment options available, but instead I'll show you the simplest way possible to get something started. As you might have guessed, the support for this is already available within Rails. In Chapter 3 I mentioned the server script, and you can use that to start a number of different web servers. By default, Rails starts WEBrick, but if you have installed Mongrel, Rails will use it instead. To just run everything, execute this command:

```
jruby script/server
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 41

If you have configured the database correctly, and there are no exceptional errors in your code that would make everything crash during startup, you'll see some startup messages from Mongrel, telling you that it has started and is listening on port 3000. So, just point your web browser at `http://localhost:3000/` and you should see be able to see the Rails status page.

The `server` script takes several parameters. The two most interesting are `-e` and `-p`. With `-p` you can set which port Mongrel should listen on, and with `-e` you can specify that Mongrel should start with another runtime environment than the default (which is development).

From now on you should just leave Mongrel running in the background so you can check your results easily, whenever you have changed anything.

Of course, if you'd like to try running with WEBrick, or any other web server, you can do that by naming it on the command line like this:

```
jruby script/server webrick
```

If Rails can find the web server, it will start using that instead.

A First Model

After you've generated the new application and tested that your database configuration is sound by starting up the web server, it's time to create the first parts of your model. You'll create three different model classes here, define their relationship, and then create the migrations for them. You'll generate the models associated with products first, because that's the first part of the administrative user interface you'll create. I've decided to name the models `Product`, `ProductType`, and `ProductCategory`. These names are important to remember, because Rails has some guidelines regarding how to translate a model name into a database table.

Product Type

First of all, you'll create the model called `ProductType`, because it doesn't have any dependencies on other models. Each `Product` will have one type, and each `ProductCategory` will also have one type, so these depend on the type being there.

To create a new model, you use the model generator, like this (in the root directory of the Shoplet application):

```
jruby script/generate model ProductType
```

This creates several files and also prints some output. It should look like this:

```
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/product_type.rb
      create  test/unit/product_type_test.rb
      create  test/fixtures/product_types.yml
      create  db/migrate
      create  db/migrate/001_create_product_types.rb
```

Now, as mentioned before, all model code resides in the directory `app/models`. A file called `product_type.rb` now exists in that directory. However, because the `ProductType` is a simple

881-4 CH04.qxd 8/19/07 3:11 PM Page 42

model object, you won't add anything to it right now. It's fine as it is. You do need to define the migration for `ProductType`, and decide which fields it should contain. The way to do that is to open the file called `db/migrate/001_create_product_types.rb` in a text editor. Right now that file is quite sparse, because you haven't added any content to it. So, right now it will create a table with one column called `id` in it, and that's it. A `ProductType` should probably contain some information, at least. Right now, you only need a name, and that's easy to add because Rails has added a name definition in a comment. So, remove the pound sign and you have all the table definitions you need for a `ProductType`.

But that's not all; you also need some initial data. There will never be many types in the system, and you know already what types those should be. Further, this is part of the definition of the product types, so you should also add some code that adds three new rows to the `product_types` table. The easiest way to do this is to use ActiveRecord. The final code for the `file 001_create_product_types.rb` should look like this:

```
class CreateProductTypes < ActiveRecord::Migration
  class ProductType < ActiveRecord::Base; end

  def self.up
    create_table :product_types do |t|
      t.column :name, :string
    end

    data
  end

  def self.data
    ProductType.create :name => 'Book'
    ProductType.create :name => 'Music'
    ProductType.create :name => 'Movie'
  end

  def self.down
    drop_table :product_types
  end
end
```

There are three differences from the code that Rails generated. First, the line defining the `name` column isn't commented out any more. Second, an inner class called `ProductType` is defined that inherits from `ActiveRecord::Base`. The class definition is empty, but it still allows us to access the ActiveRecord helpers later on. Third, you've defined a method called `data`, and you call that method last in the `up` method. The `data` method creates three `ProductType` objects with different names. The `create` method saves these directly to the database. That's all there is to it right now. Before showing how to make the migrations go into the database, we'll go through the `Product` and `ProductCategory` models and define these.

Product

The next step is to create the `Product` model. As with the `ProductType`, you'll do it with the `generate` script:

```
jruby script/generate model Product
```

The output from this command looks almost exactly like the one for `ProductType`, so I won't repeat it. The migration file for products is called `db/migrate/002_create_products.rb`, and you need to add some more changes to this one. A `Product` should have a name, a description, a product type, and a price. It also has zero or more categories associated with it, but we'll take care of those next. For the moment you'll save the price in cents, because Rails 1.1.6 doesn't handle decimal types in databases that well. (Also, you should never store a price as a `float` or `double`, due to the problems with representing real numbers exactly in binary.) So, open up the file `db/migrate/002_create_products.rb` and uncomment the `name` entry. You should also add some new columns. The final version of the file should look like this:

```
class CreateProducts < ActiveRecord::Migration
  def self.up
    create_table :products do |t|
      t.column :name, :string
      t.column :description, :text
      t.column :product_type_id, :integer
      # This price will be in cents to make it easier
      # It might be better to implement this using Decimal
      t.column :price, :integer
    end
  end

  def self.down
    drop_table :products
  end
end
```

As you can see, you just added entries for `description`, `product_type_id`, and `price`. A comment is also in place to make it obvious why the price is an integer. After adding the database information, you need to change some parts in the model definition. That file is called `app/models/product.rb`. After the current changes it should look like this:

```
class Product < ActiveRecord::Base
  has_and_belongs_to_many :product_categories
  belongs_to :product_type
end
```

The directives simply say what it looks like they say. Having `belongs_to` means that for every `ProductType` there are zero or more `Products` for it. Using `has_and_belongs_to_many` requires us to have a join table for `product_categories`, and that's what we'll look at next.

Product Categories

As for the other model objects, you generate the base with the generator script:

```
jruby script/generate model ProductCategory
```

The migration code for product categories is slightly more involved than the other two, because you need to do a fair number of things in it. First of all, a product category belongs to a product type, and has a name. That means the part that defines the `product_categories` table looks like this (in `db/migrate/003_create_product_categories.rb`):

```
create_table :product_categories do |t|
  t.column :product_type_id, :integer
  t.column :name, :string
end
```

You also need a `data` method, like the one you used for `ProductType`. Because `ProductCategory` references `ProductType`, you first need to fetch those. Second, you create some categories for each type. The `data` method looks like this:

```
def self.data
  book = ProductType.find_by_name 'Book'
  music = ProductType.find_by_name 'Music'
  movie = ProductType.find_by_name 'Movie'

  %w(Computers Mysteries
     Science\ Fiction Crime).each do |v|
    ProductCategory.create :product_type => book, :name => v
  end

  %w(Jazz World\ Music Electronic
    Rock Indie Country).each do |v|
    ProductCategory.create :product_type => music, :name => v
  end

  %w(Action Science\ Fiction Drama
     Comedy Thriller).each do |v|
    ProductCategory.create :product_type => movie, :name => v
  end
end
```

However, this won't work unless you define `ProductType` and `ProductCategory`, so that's what you'll do first, before the `up` method:

```
class ProductType < ActiveRecord::Base; end
class ProductCategory < ActiveRecord::Base
  belongs_to :product_type
end
```

Last, you also need to create the join table that connects `Product` and `ProductCategory`. Rails has a specific naming scheme for such tables. The two tables to join should be in the

881-4 CH04.qxd 8/19/07 3:11 PM Page 45

name, ordered alphabetically, and joined with an underscore. In our case, the join table should be named `product_categories_products`, and the definition looks like this:

```
#join table for products and product_categories
create_table :product_categories_products, :id => false do |t|
  t.column :product_id, :integer
  t.column :product_category_id, :integer
end
```

Because a join table should only contain the IDs for the two entries to join, you need to tell Rails that no `id` column should be generated for the table. The final file for defining `ProductCategories` should look like this:

```
class CreateProductCategories < ActiveRecord::Migration
 class ProductType < ActiveRecord::Base; end
 class ProductCategory < ActiveRecord::Base
   belongs_to :product_type
 end

 def self.up
   create_table :product_categories do |t|
     t.column :product_type_id, :integer
     t.column :name, :string
   end

   data

   #join table for products and product_categories
   create_table :product_categories_products, :id => false do |t|
     t.column :product_id, :integer
     t.column :product_category_id, :integer
   end
 end

 def self.data
   book = ProductType.find_by_name 'Book'
   music = ProductType.find_by_name 'Music'
   movie = ProductType.find_by_name 'Movie'

   %w(Computers Mysteries
      Science\ Fiction Crime).each do |v|
     ProductCategory.create :product_type => book, :name => v
   end

   %w(Jazz World\ Music Electronic
      Rock Indie Country).each do |v|
     ProductCategory.create :product_type => music, :name => v
   end
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 46

```
   %w(Action Science\ Fiction Drama
      Comedy Thriller).each do |v|
     ProductCategory.create :product_type => movie, :name => v
   end
 end

 def self.down
   drop_table :product_categories_products rescue nil
   drop_table :product_categories
 end
end
```

When this migration runs, you'll have a few different categories to choose from in every type. The model file for `ProductCategory` is comparatively simple. You just define the same relationship to `ProductType` and `Product` as you've already done in the corresponding models:

```
class ProductCategory < ActiveRecord::Base
  has_and_belongs_to_many :products
  belongs_to :product_type
end
```

Notice that you could have put a `has_many :product_categories` inside of `product_type.rb`, if you ever were interested in going from `ProductType` to all product categories for that type.

Running the Migrations

You've defined three different migration files, mostly automatically generated by the Rails scripts. You might have noticed that the file naming follows a simple pattern. That's because a database used with migrations always has a version number associated with it. From scratch, that version number is `0`. For each migration file run, the version is incremented by one. Because migration files are run alphabetically from the `db/migrate` directory, it's a good custom to name each file with the version number it will result in, even though it isn't necessary.

So, how to go about creating the database tables for your model? It's simple. You run `rake` on the target `db:migrate`, and Rails takes care of the rest:

```
jruby -S rake db:migrate
```

By default, this migrates your database to the latest version present in your files. If you want a specific version, you can specify that:

```
jruby -S rake rake db:migrate VERSION=2
```

The environment this runs in is `development`. At some point you should run your migrations for `test` and `production` too, and that's equally easy:

```
jruby -S rake db:migrate RAILS_ENV=production
```

That's about it for migrations at the moment. There's more to it, but right now this will get you started.

Validations

Normally the values that can be accepted on an attribute are constrained in one or several ways that won't show up in the database schema. For example, a price for a `Product` shouldn't be negative. These model constraints exist in the model class, and you'll add a few to show how Rails validations typically look. Validations are important, because they stop invalid data from entering the database.

The first model is `ProductType`. There's only one invariant, and that is that there should always be a name for it. (Rails automatically caters for the `id` field, so you won't have to ensure that it exists.) Validating the presence of an attribute, where *presence* means it should be there, and not be empty, is easy. You just add the `validates_presence_of` method to the model class. After adding that, the file `app/models/product_type.rb` looks like this:

```
class ProductType < ActiveRecord::Base
  has_many :products

  validates_presence_of :name
end
```

You'll also add a few validations to the `Product` model. First, there are a few required attributes. These are `price`, `name`, and `product_type`. A `Product` isn't valid without this information. On the other hand, a `Product` doesn't need a description. So, you add the validation to the `app/models/product.rb` file:

```
validates_presence_of :price, :name, :product_type
```

The next part is price. Because you represent price as cents, it should be an integer, and nothing else. You ensure that with the `validates_numericality_of` validator:

```
validates_numericality_of :price, :only_integer => true
```

Note the `only_integer` attribute. If you didn't write that part, `numericality` includes real numbers too, which you don't want.

The next two validations are slightly more involved. You first want to check that the price isn't `0` and that the price isn't negative. You should use `validate_each` to achieve that:

```
validates_each :price do |m,attr,value|
  if value == 0
    m.errors.add(attr,"Price can't be 0")
  elsif !value.nil? && value < 0
    m.errors.add(attr, "Price can't be negative")
  end
end
```

In this validator you need to check the attribute value specifically, and also add the error conditions by hand. The `m` parameter is the model object in question, so you can validate by comparing different attributes if you want. That is also useful in the next validation, which is

881-4 CH04.qxd 8/19/07 3:11 PM Page 48

a bit more complex. You want to make sure all the `product_categories` have the same type as the current model object:

```
validates_each :product_categories do |m,attr,value|
  if !value.nil? && value.any? {|v| m.product_type != v.product_type }
    m.errors.add(attr, "Category can't be of another type")
  end
end
```

You do this by using the `Enumerable` method `any?`. If any objects match the condition, you add an error condition.

These kinds of validations are useful, and can include many important preconditions and postconditions. Whenever you create models, you should think long and hard about what the invariants and contracts for that object should be, and add validations that take care of the exceptions from these contracts. As the next section talks about, you should also test that the validations you've written say what you think they say.

Unit Testing Products

As mentioned in the section about testing in Chapter 3, unit tests are used to test models. I'll show a few tests for `ProductType` and `Product` here, but as mentioned earlier there won't be space enough to test everything as it should be tested. First of all, there is only one interesting fact to test about `ProductType`: that `name` must be provided. So, open up the file `test/unit/product_type_test.rb`, remove the `test_truth` method, and add this method instead:

```
def test_invalid_name
  p = ProductType.new
  assert !p.valid?
  assert p.errors.invalid?(:name)
end
```

This method first creates a new `ProductType`, and because you don't specify a name, it shouldn't be valid. The method `assert`, and all methods beginning with `assert_`, are used to check a certain invariant. In this method, you just check that the model object is not valid, and that the errors provided include at least one for `name`.

There are more things to test for `Product`, though. First of all, go ahead and open the file `test/unit/product_test.rb` and remove the `test_truth` method. The next step demands a slight deviation into the territory of fixtures. A fixture is a YAML file that contains data for test fixtures, which means your tests will always use the same data, instead of relying on whatever can be found in the database at the moment. Because `Products` needs product types to work, you'll first change those fixtures. Open the file `test/fixtures/product_types.yml` and replace the contents with this:

```
book:
  id: 1
  name: Book
music:
  id: 2
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 49

```
  name: Music
movie:
  id: 3
  name: Movie
```

■ **Caution** The indentation here must be done with spaces, not tabs. You'll see errors when running the tests otherwise.

As you might notice, this is the same type as the migrations added, but adding them here means you can get at the objects much more easily later on. Next, you add two fixtures for `Product` by replacing the contents of the file `test/fixtures/products.yml` with this:

```
first:
  id: 1
  product_type_id: 1
  name: Abc
  price: 1440
another:
  id: 2
  product_type_id: 2
  name: Cde
  price: 2990
```

Now you're almost ready to add some `Product` testing. You just need to tell Rails that it should use the fixtures for product types, in addition to the fixtures for `Product`. You can achieve this by adding this line after the corresponding line for products, in the file `product_test.rb`:

```
fixtures :product_types
```

The first test is the same as for product type, just checking that an invalid name can't get in:

```
def test_invalid_name
  p = Product.new
  assert !p.valid?
  assert p.errors.invalid?(:name)
end
```

You do the exact same thing for `type`:

```
def test_invalid_type
  p = Product.new
  assert !p.valid?
  assert p.errors.invalid?(:product_type)
end
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 50

The next step is to check that invalid categories will be noticed, and that your validation for that works:

```
def test_invalid_category
  p = Product.new
  p.product_type = product_types('book')
  c = ProductCategory.create :name => 'ABC',
            :product_type => product_types('music')
  p.product_categories << c
  assert !p.valid?
  assert p.errors.invalid?(:product_categories)
end
```

Here you use the product type fixtures, by calling the method `product_types` with the name of the fixture to fetch. In this way you conveniently create a new `Product` with a specific type and add a newly created category from another type.

You need also to make absolutely sure that invalid prices can't be set:

```
def test_invalid_price
  p = Product.new
  p.price = nil
  assert !p.valid?
  assert p.errors.invalid?(:price)
  p = Product.new
  p.price = 1.0
  assert !p.valid?
  assert p.errors.invalid?(:price)
  p = Product.new
  p.price = 0
  assert !p.valid?
  assert p.errors.invalid?(:price)
  p = Product.new
  p.price = -17
  assert !p.valid?
  assert p.errors.invalid?(:price)
end
```

This test is slightly longer because you want to try a few different invalid prices, to see that all corner cases are covered.

These tests are all well and good, but they only test the negative side. For what it's worth, each of these tests would pass if something was really wrong with the system. So, what you do is add a positive test too, where everything works as it should. If that test fails, you know something is iffy:

```
def test_valid_product
  p = Product.new
  p.price = 122
  p.name = "Hello Goodbye"
  p.product_type = product_types('book')
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 51

```
  assert p.valid?
end
```

That concludes the testing of the product family of model objects. Do add more tests if you come up with something suitable. You can never have too many tests.

To run these tests, just execute this command:

```
jruby -S rake
```

The standard `rake` task runs all unit and functional tests by default. Make sure that you've migrated everything in the test environment before doing this, though.

Creating a Scaffold for Products

Now that we've created the basis for our model, added some validations, and also made sure that those validations work, it's finally time to create a web interface to handle products. The first step in such an endeavor is to create a scaffold. In Rails, a scaffold means generated code to support the basic CRUD operations for a specific model object. We'll begin from such a scaffold and then change it to fit our needs. This will be the most important part of our administrative user interface.

To create the scaffolds, you use the `script/generate` script, like this:

```
jruby script/generate scaffold Product
```

As usual when generating something, you get some output that tells you exactly which files have been created or modified. Now start up the web server, and visit the address `http://localhost:3000/products`. If everything has gone right, you'll see an empty listing of products and a link to add a new one. Go ahead and explore the interface and see what you can do. You probably won't be able to add a new product yet, because there's no way to choose among the product types available yet. So, your first step is to change the addition of products so you can get data in there.

A Rails scaffold is a controller (that can be found in `app/controllers/products_controller.rb`) and a set of views, which almost always are RHTML files. To make it possible to work with product types within the `Product` scaffold, you first need to open up `app/controllers/products_controller.rb`. Find the method called `new`. It looks like this:

```
def new
  @product = Product.new
end
```

You simply need to provide the `ProductTypes` available here. You'll later need to add `ProductCategories` too, so just chuck that in while you're at it. The method should look like this when you're finished:

```
def new
  @product = Product.new
  @product_types = ProductType.find(:all)
  @product_categories = ProductCategory.find(:all)
end
```

That is all that's needed to provide the necessary information to the view. The scaffolds are smart, though; Rails uses the same code that views the RHTML for adding a new product to edit an existing product, so you'll want to make the same change to the edit method, which should look like this now:

```
def edit
  @product = Product.find(params[:id])
  @product_types = ProductType.find(:all)
  @product_categories = ProductCategory.find(:all)
end
```

The next step is to alter the file called app/views/products/_form.rhtml. The underscore means it's a partial, and it is used from the views edit and new. You want to change this file a little bit, so just go ahead and change the contents into this:

```
<%= error_messages_for 'product' %>

<!--[form:product]-->
<p><label for="product_name">Name</label><br/>
<%= text_field 'product', 'name'  %></p>

<p><label for="product_description">Description</label><br/>
<%= text_area 'product', 'description'  %></p>

<p><label for="product_product_type">Product Type</label><br/>
<%= select 'product', 'product_type_id', @product_types.collect {|p|
      [ p.name, p.id ] },{}%></p>

<p><label for="product_price">Price</label><br/>
$<%= text_field_tag 'product[price]', price(@product)[1..-1]  %></p>
<!--[eoform:product]-->
```

You've changed just the entries for ProductType and Price, but those are important changes. First, the ProductType change means you can switch among the available product types, and choose the one you want. The select helper method makes a select box with all types in it. The price is a little trickier, though. Remember that you represent it as cents? Well, it should be formatted in the regular format when viewed by the end user. So, you'll add a helper method that accomplishes this for you. The place to add this helper is in the file app/helpers/application_helper.rb because you want all of your application to have access to this helper. You just add these two methods to the module within:

```
def price(product)
  money product.price
end

def money(pr)
  pr ? "$%d.%02d" % pr.divmod(100) : "$0.00"
end
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 53

Because you most often want to display the price with a dollar sign in front, you have to strip out that sign when displaying the price in the box. The next step is making sure that saving these objects works too. To do this, you need to change the `create` and `update` methods. First of all, though, you need a small helper method in the controller that allows you to handle price more easily. So, in the end of the class, add this method declaration:

```
private
def intern_price
  if params[:product] && params[:product][:price]
    v = params[:product][:price].split('.').map(&:to_i)
    params[:product][:price] = v[0]*100 + v[1]
  end
end
```

This helper would reformat the price into cents if a `price` parameter was submitted. The `private` in the beginning says that this method should not be available as an action on the controller.

Next you need to change the `create` method to handle the new price:

```
def create
  intern_price
  @product = Product.new(params[:product])
  if @product.save
    flash[:notice] = 'Product was successfully created.'
    redirect_to :action => 'list'
  else
    @product_types = ProductType.find(:all)
    @product_categories = ProductCategory.find(:all)
    render :action => 'new'
  end
end
```

The other thing you added was `@product_types` and `@product_categories`, in case something goes wrong. This lets you see the original page again, with error messages attached. You need to do the same thing with the `update` method:

```
def update
  @product = Product.find(params[:id])
  intern_price
  if @product.update_attributes(params[:product])
    flash[:notice] = 'Product was successfully updated.'
    redirect_to :action => 'show', :id => @product
  else
    @product_types = ProductType.find(:all)
    @product_categories = ProductCategory.find(:all)
    render :action => 'edit'
  end
end
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 54

Now you can go ahead and create a product or two. You don't need to restart the web server either. If everything works correctly, you should also be able to see your new products in the list. You can edit and destroy products without trouble, too. You might note that the listing of products isn't that good right now. It shows a lengthy description, but not product type. That should probably be changed, so we'll take a look at that next. Now, you don't need to change anything in the controller to change those parts of the listing, because you have all the data you need already. You can find the listing at `app/views/products/list.rhtml`, and it contains a generic mechanism that walks you through the available attributes and shows these. We'll change it a bit:

```
<h1>Products</h1>

<table width="400">
  <tr>
    <th align="left">Name</th>
    <th>Type</th>
    <th>Price</th>
    <th>&nbsp;</th>
    <th>&nbsp;</th>
  </tr>
  <% for product in @products %>
    <tr>
      <td align="left" valign="top"><%= link_to h(product.name),
         {:action => 'show', :id => product},
                    :class=>'productLink' %></td>
      <td align="right" valign="top"><%=h product.product_type.name%></td>
      <td align="right" valign="top"><%=price product %></td>
      <td>&nbsp;</td>
      <td><%= link_to 'Remove', {:action => 'destroy', :id => product},
                          :confirm => 'Are you sure?', :post => true %></td>
    </tr>
  <% end %>
</table>

<%= link_to 'Previous page',
   { :page => @product_pages.current.previous } if
                 @product_pages.current.previous %>
<%= link_to 'Next page',
   { :page => @product_pages.current.next } if
                 @product_pages.current.next %>

<br />
<%= link_to 'New product', :action => 'new' %>
```

As you can see, you hard code the columns, and you only display the name, type, and price. By using the `link_to` helper, you make it possible to show a `Product` by clicking the product name. In that way you can remove the separate `edit` and `show` links. The `h` helper takes a string and returns a string where all HTML-specific characters have been encoded;

it's a good habit to always use this when displaying data. Also note that you use the `price` helper here again, and that you've added a CSS class called `productLink` to the name display. This makes it easy to add some good looks later on. While we're working on the layout of displaying products, let's also fix the show page. You can find it in `app/views/products/show.rhtml`, and you should turn it into something like this:

```
<p><b>Name:</b> <%= @product.name %></p>
<p><b>Description:</b><br/>
  <%= @product.description %>
</p>
<p><b>Price:</b> <%= price(@product) %></p>
<p>
  <b>Product Type:</b> <%=h @product.product_type.name %>
</p>

<%= link_to 'Edit', :action => 'edit', :id => @product %> |
<%= link_to 'Back', :action => 'list' %>
```

Once again, there's nothing unexpected. You just use the `price` helper and show the name of the product type.

Ajax

Our administrative user interface for products is starting to look good. Except for some CSS styling, only one thing is missing: the product categories. These are slightly more tricky. You want it to be possible to choose zero or more products while editing or adding a product, but the only choices available should be the ones that match the currently selected product type. That means you'd need to reload the page each time the product type changed. However, there's a better way. Ajax is the new hype word in web development. I won't talk too much about it here, but suffice it to say, it's a perfect technology for the current problem. With Ajax you can create a listener that updates just part of the page when product types have been changed. This is ideal right now. So, how do you go about it? Well, the first step is to create a partial. This partial will be used both by the forms on the full request, and by the Ajax call that will update the page later on. First, create a file called `app/views/products/_categories.rhtml` with this content:

```
<%= select_tag 'product_categories[]',
      options_for_select(@product_categories.select {|p|
            p.product_type.id ==  ((@product &&
         @product.product_type) || @product_types.first).id
        }.collect {|p| [ p.name, p.id ] },
   (@product && @product.product_categories.collect(&:id)) || nil),
   :multiple=>true, :size=>5 %>
```

It isn't much code, but it's slightly messy. What you do is, based on the current `@product`, walk through all product categories in the system, only using those that match the product's product type. Add these to the select box. The next step is to collect all categories that should be selected, which is the next parameter to the `select_tag` call. The next step is to use this

881-4 CH04.qxd 8/19/07 3:11 PM Page 56

partial from within the forms. So, open up `app/views/products/_form.rhtml` again and add this code after the part that displays the product type:

```
<p><label for="product_categories[]">Product Categories
<span id="waitOut"></span></label><br/>
<div id="categoriesSelector">
<%= render :partial => 'categories' %></div></p>
```

Notice that you've added a span that is empty, and a `div` tag that contains the output generated from the partial. You'll need these two elements later on for the Ajax-y parts of the system. But right now, you need to alter the `products_controller.rb` again, to make sure everything will be saved when you save a product. This is a little bit outside the box, because Rails doesn't handle these multiple selections out of the box. In the `create` method, you need to add this code directly after the call to `Product.new`:

```
if params[:product_categories]
  @product.product_categories <<
           ProductCategory.find(
        params[:product_categories].collect(&:to_i))
end
```

This code adds all `product_categories` to the product, if there are any. You need to change the `update` method a little bit more, because you now have to make sure validation takes place. The new `update` method should look like this:

```
def update
  @product = Product.find(params[:id])
  intern_price
  if @product.update_attributes(params[:product])
    if params[:product_categories]
      @product.product_categories = ProductCategory.find(
           params[:product_categories].collect(&:to_i))
    end
    if @product.save
      flash[:notice] = 'Product was successfully updated.'
      redirect_to :action => 'show', :id => @product
    end
  end
  if !@product.valid?
    @product_types = ProductType.find(:all)
    @product_categories = ProductCategory.find(:all)
    render :action => 'edit'
  end
end
```

The big difference here is that you check if the product is valid explicitly, because it can become invalid in two different places (in `update_attributes`, or `save`). You could do this in other ways, but that would mean you'd have to duplicate code in the method.

881-4 CH04.qxd 8/19/07 3:11 PM Page 57

While you're still in the controller, you'll add another action that the Ajax call is supposed to use to update the partial. The method is called `categories_partial` and looks like this:

```
def categories_partial
  @product = Product.find_by_id(params[:id]) || Product.new
  @product.product_type = ProductType.find(params[:tp])
  @product_types = ProductType.find(:all)
  @product_categories = ProductCategory.find(:all)
  render :partial => 'categories'
end
```

You need to create a dummy product for the partial to work and set the `product_type` correctly. After you've done that, you fetch the needed product types and categories, and render the partial. Nothing strange here, really. However, now you're slowly moving in to the part that makes Ajax so practical. Ajax uses JavaScript to asynchronously update parts of the view. So, what you have to do is provide a listener on the product type select box that updates the `categories` partial. Once again, open up `app/views/products/_form.rhtml` and change the product type parts to look like this:

```
<p><label for="product_product_type">Product Type</label><br/>
<%= select 'product', 'product_type_id',
          @product_types.collect {|p| [ p.name, p.id ] },{},
           :onChange => "\$(waitOut).innerHTML=
           '<i>(Updating categories, please wait...)</i>';
          new Ajax.Updater('categoriesSelector','#{url_for(
   :controller=>'products',:action=>'categories_partial',
   :id=>@product)}?tp=' + this[this.selectedIndex].value,
             {onComplete:function(){\$(waitOut).innerHTML='';
          new Effect.Highlight('categoriesSelector');},
 asynchronous:true});"%></p>
```

What you do here is a little involved. The first two rows are exactly the same. On the fourth row, you add an `onChange` handler that first sets the `innerHTML` attribute of the span with the ID `waitOut` to a text that lets the user know something is happening. The code then creates a new `Ajax.Updater` that asynchronously does an HTTP request to the `categories_partial` action. When completed it replaces the content of the `div` with the ID `categoriesSelector` with what it got from the HTTP request. It also removes the "please wait" text and highlights the new contents when they arrive.

To get all this working, you need to make a slight detour to layouts. If you remember from Chapter 3, you can define a layout that defines the look for many pages. Because the Ajax parts need some JavaScript included, you need to create a layout that manages this for you. So, open the file called `app/views/layouts/products.rhtml` and change it to look like this:

```
<html>
<head>
  <title>Products: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
  <%= javascript_include_tag :defaults %>
</head>
```

```
<body>

<p style="color: green"><%= flash[:notice] %></p>

<%= yield  %>

</body>
</html>
```

The only thing you change right now is to make the view call `javascript_include_tag` with an argument of `:defaults`. This includes all the commonly used JavaScript files needed for Ajax to work. After you've made this change, go create a new product and you'll notice that the product categories now work. Try to change `ProductType`; you'll notice that you get a message to wait patiently, and then the product categories are updated.

This is just a small taste of what can be achieved with Ajax. We'll look more at Web 2.0 techniques in Chapter 8. For now, you also need to update the `show` view to display the product categories. This should be done in the file `app/views/products/show.rhtml` and is simple. You just add this after the entry for `ProductType`:

```
<p>
  <b>Product Categories:</b><br/>
  <% @product.product_categories.each do |cat| %>
  <%=h cat.name %><br/>
  <% end %>
</p>
```

Adding Some Good Looks

Finally, you need to make everything look slightly more presentable. The first step is to create a new layout called `admin.rhtml` that the entire administrative user interface will use. So, create the file `app/views/layouts/admin.rhtml` and fill it with this code:

```
<html>
<head>
  <title>Shoplet Online Store Administration:
            <%=controller.action_name%>
            <%=h params[:controller]%></title>
  <%= stylesheet_link_tag 'shoplet' %>
  <%= javascript_include_tag :defaults %>
</head>
<body>
  <table width="100%" height="100%">
    <tr>
      <td width="250" class="leftMenu" align="center" valign="top">
        <h2><%= link_to 'Shoplet', :controller => 'store',
                            :action=>'index'%></h2>
        <h3><%= link_to 'Online Shopping', :controller => 'store',
                            :action=>'index'%></h3>
        <h3><%= link_to 'Administration', :controller => 'store',
                        :action=>'index'%></h3>
        <br/>
        <ul style="text-align: left;">
          <li><%= link_to 'Administrate products', {:controller =>
                    'products'},:class => 'adminLink' %></li>
          <li><%= link_to 'Handle orders', {:controller =>
                    'orders'},:class => 'adminLink' %></li>
          <li><%= link_to 'Authenticated users', {:controller =>
                    'users'},:class => 'adminLink' %></li>
        </ul>
        <br/>
      </td>
      <td class="main" valign="top">
        <p style="color: green"><%= flash[:notice] %></p>
        <p style="color: red"><b><%= flash[:error] %></b></p>

        <%= yield  %>
      </td>
    </tr>
  </table>
</body>
</html>
```

Note that I've added some links here to controllers that you haven't created yet. That's where the rest of the administrative user interface will be found. After you've created this file, you need to edit `products_controller.rb` to use it, too. That's easy enough. Just add this method call on the second row:

```
layout "admin"
```

Before you try it out, you should add a new style sheet too. Create a file called `public/stylesheets/shoplet.css` and fill it with this code:

```
body {
  margin: 0px;
  padding: 0px;
}

h3 {
  font-size: 1em;
  font-weight: bold;
}

h2 a {
  text-decoration: none;
  color: black;
}
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 60

```
h3 a {
  text-decoration: none;
  color: black;
}

a {
  text-decoration: none;
  font-weight: bold;
}

thead td {
  font-weight: bold;
}

.productLink {
  color: black;
  font-weight: normal;
}

.adminLink {
  color: black;
  font-weight: normal;
}

.leftMenu {
  padding-top: 20px;
  border: 1px solid black;
  border-left: none;
  font-family: arial, sans-serif;
  background-color: #CCCCEE;
}

.main {
  padding: 30px;
  color: dark-grey;
}
```

If you now update the web browser, you'll note that everything looks many magnitudes better. If you open up the file `test/functional/products_controller_test.rb`, you'll also see that lots of test cases have already been created for you, entirely for free. You need to modify these slightly in some cases to accommodate the structure you've adopted, with price and categories handled specially. But I'll wait and show you how to do that when we talk about testing the login engine we'll write in the sections "Adding Some Authentication" and "Functional Tests," because that changes the functional tests a bit.

881-4 CH04.qxd 8/19/07 3:11 PM Page 61

More Models

Because you have more or less finished the administration side of `Products`, it's time to think about the other parts the user interface should sport. If you remember the links we added to the layout in the end of the last section, the two parts needed will be one for order handling, and one for user administration. I've modeled the order system like this: An `Order` has zero or more `OrderLines`, and each `Order` is associated with a `Customer`. Regarding users, there will be a `User` model object. You'll begin by generating these models:

```
jruby script/generate model Customer
jruby script/generate model Order
jruby script/generate model OrderLine
jruby script/generate model User
```

After you've generated these models, you should open up the file `db/migrate/004_create_customers.rb and` change it to look like this:

```
class CreateCustomers < ActiveRecord::Migration
  def self.up
    create_table :customers do |t|
      t.column :given_name, :string
      t.column :sur_name, :string

      t.column :shipping_address_street, :string
      t.column :shipping_address_postal, :string
      t.column :shipping_address_zip, :string
      t.column :shipping_address_country, :string

      t.column :billing_address_street, :string
      t.column :billing_address_postal, :string
      t.column :billing_address_zip, :string
      t.column :billing_address_country, :string
    end
  end

  def self.down
    drop_table :customers
  end
end
```

This is arguably not such a good database design, but extending it more would take too much focus from the core you need for the current project. The `Order` model that you find in `db/migrate/005_create_orders.rb` should look like this:

```
class CreateOrders < ActiveRecord::Migration
  def self.up
    create_table :orders do |t|
      t.column :customer_id, :integer
      t.column :time, :timestamp
      t.column :status, :string
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 62

```
    end
  end

  def self.down
    drop_table :orders
  end
end
```

This is all you need to specify an order. You want to know the customer information, the time the order happened, and if it has been handled or not. The file `db/migrate/006_create_order_lines.rb` contains the database definitions for `OrderLine`:

```
class CreateOrderLines < ActiveRecord::Migration
  def self.up
    create_table :order_lines do |t|
      t.column :order_id, :integer
      t.column :product_id, :integer
      t.column :amount, :integer
    end
  end

  def self.down
    drop_table :order_lines
  end
end
```

An `OrderLine` is associated with an `Order` and a `Product`, and can contain more than one of the same `Product`. Last, here's `db/migrate/007_create_users.rb`:

```
class CreateUsers < ActiveRecord::Migration
  class User < ActiveRecord::Base; end

  def self.up
    create_table :users do |t|
      t.column :username, :string
      t.column :password, :string
    end

    User.create :username => 'admin',
                :password => 'admin'
  end

  def self.down
    drop_table :users
  end
end
```

The only thing that's different about this model is that you need to create at least one user from scratch, which you can use to add further users. Before you go any further, it's important that you migrate the database, so all these new tables are available:

```
jruby -S rake db:migrate
jruby -S rake db:migrate RAILS_ENV=test
```

When this is done, you can edit the model files and add all relationships that until now you only had in the database. You begin with `Customer`, in the file `app/models/customer.rb`. It should look like this:

```
class Customer < ActiveRecord::Base
  has_many :orders

  def to_s
    "#{given_name} #{sur_name}"
  end
end
```

The only thing you would possibly want from a `Customer` is to know which orders he or she is associated with. In some circumstances printing a `Customer` is interesting, so you add a custom `to_s` method to cater for this. Next, open the file `app/models/order.rb` and change it into this:

```
class Order < ActiveRecord::Base
  has_many :order_lines
  belongs_to :customer
end
```

An `Order` has many `OrderLines` and has one `Customer`. You can find the definitions for `OrderLine` in `app/models/order_line.rb` and you should change them into this:

```
class OrderLine < ActiveRecord::Base
  belongs_to :product
  belongs_to :order
end
```

All this is obvious. Finally, the `User` model is good as it is.

User Administration

Now it's time to add a new controller. The purpose of this one is to allow us to add or remove users, because you'll need this as soon as you switch on the authentication system. You'll begin with a scaffold for this:

```
jruby script/generate scaffold User
```

Then you'll tear a whole lot of stuff out of it, because you just want to be able to do three things: list the users, add a user, and remove a user. (You can implement password changing if you want, but at this stage it doesn't matter if you change passwords or just re-create the user.) So, open up `app/controllers/users_controller.rb` and remove the `show`, `edit`, and `update`

881-4 CH04.qxd 8/19/07 3:11 PM Page 64

methods. Also remove the reference to `update` in the `verify` call at the top. Next, add a directive to use the `admin` layout on the second line:

```
layout "admin"
```

That's all that's needed for the controller. Next, on to the views. You can safely remove the files `app/views/users/show.rhtml` and `app/views/users/edit.rhtml`. Next, change the `app/views/users/list.rhtml` file to read like this:

```
<h1>Authenticated users</h1>

<table width="300">
  <tr>
    <th align="left">Username</th>
    <th align="right">Password</th>
  </tr>

<% for user in @users %>
  <tr>
    <td align="left"><%= h user.username %></td>
    <td align="right"><%= h user.password.gsub(/./,'*') %></td>
    <td>&nbsp;</td>
    <td><%= link_to 'Remove', { :action => 'destroy',
           :id => user }, :confirm => 'Are you sure?', :post => true %></td>
  </tr>
<% end %>
</table>

<%= link_to 'Previous page', { :page =>
            @user_pages.current.previous } if
        @user_pages.current.previous %>
<%= link_to 'Next page', { :page =>
           @user_pages.current.next } if
       @user_pages.current.next %>

<br />
<%= link_to 'New user', :action => 'new' %>
```

Try it out by going to `http://localhost:3000/users`. If all is well, you should see one entry for the "admin" user, and nothing else. This concludes the user administrative interface. Go ahead and add a user or two. Later on you'll protect these pages from entry if you don't have a valid username and password.

Order Handling

For order handling, the flow is that you get a list of unhandled orders. Then you choose one order, see all information about it, and from that step you can mark the order as handled, or remove the order completely. You'll once again begin with a scaffold to make your work easier:

```
jruby script/generate scaffold Order
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 65

Next, open up the controller just generated (app/controllers/orders_controller.rb) and remove the show, new, create, edit, update, and destroy methods. The new complete file should look like this:

```
class OrdersController < ApplicationController
  layout "admin"

  def index
    list
    render :action => 'list'
  end

  # GETs should be safe
  # (see http://www.w3.org/2001/tag/doc/whenToUseGet.html)
  verify :method => :post, :only => [ :remove, :handled ],
         :redirect_to => { :action => :list }

  def list
    @orders = Order.find(:all,:conditions => "status = 'placed'")
  end

  def handle
    @order = Order.find(params[:id])
    @price = @order.order_lines.inject(0) do |sum,l|
      sum + l.amount * l.product.price
    end
  end

  def remove
    Order.find(params[:id]).destroy
    redirect_to :action => 'list'
  end

  def handled
    @order = Order.find(params[:id])
    @order.status = "handled"
    if @order.save
      flash[:notice] = 'Order has been handled.'
      redirect_to :action => 'list'
    else
      @price = @order.order_lines.inject(0) do |sum,l|
        sum + l.amount * l.product.price
      end
      render :action => 'handle'
    end
  end
end
```

There's lots of new information here. First of all, you've added the `admin` layout so you get a unified layout. Second, the parameters to the `verify` method have been changed, so the only parameter includes `remove` and `handled`. The `list` method has been changed, so it only shows orders where the status is `'placed'`. This is so you can retain the orders in the database, but you don't have to see them when they've been handled.

There are also three new methods. Rails will call the `handle` method when a specific order should be shown and handled. It finds the order in question, and then sums the total price together.

The `remove` method removes the order in question from the database.

The `handled` method sets the status to `"handled"` on the order in question, and redirects to the listing. Open up the `app/views/orders/list.rhtml` file and change it so it looks like this:

```
<h1>Orders to handle</h1>

<table width="500">
  <tr>
    <th>Customer</th>
    <th>Time</th>
    <th>Amount</th>
    <th>Items</th>
  </tr>

<% for order in @orders %>
  <tr>
    <td><%= h order.customer %></td>
    <td><%= order.time.strftime("%F %H:%M") %></td>
    <td align="right"><%= money(order.order_lines.inject(0){
               |sum,ol| sum + ol.amount*ol.product.price}) %></td>
    <td align="right"><%= order.order_lines.inject(0){
               |sum,ol| sum + ol.amount} %></td>
    <td align="right" width="150"><%=
          link_to "Handle order", :action => 'handle', :id => order %></td>
  </tr>
<% end %>
</table>
```

This shows a pleasing list of orders, showing the time each order was placed, how much money it amounts to, and how many items there are. The next step is to create a new file called `app/views/orders/handle.rhtml`. This will be a big file, because it's the main place for watching data about an order. Here it is:

```
<h2>Handle order</h2>

<p><b>Customer:</b> <%= h @order.customer %></p>
<p><b>Shipping address:</b><br/>
<%= h @order.customer.shipping_address_street %><br/>
<%= h "#{@order.customer.shipping_address_postal}
       #{@order.customer.shipping_address_zip}"%><br/>
<%= h @order.customer.shipping_address_country %></p>
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 67

```
<p><b>Billing address:</b><br/>
<%= h @order.customer.billing_address_street %><br/>
<%= h "#{@order.customer.billing_address_postal}
       #{@order.customer.billing_address_zip}"%><br/>
<%= h @order.customer.billing_address_country %></p>

<br/>

<table width="480">
  <thead>
    <td width="300" align="left">Product Name</td>
    <td width="20" align="right">Quantity</td>
    <td width="80" align="right">Each</td>
    <td width="80" align="right">Price</td>
  </thead>

  <% @order.order_lines.each do |ol| %>
    <tr>
      <td width="300" align="left"><%= h ol.product.name %></td>
      <td width="20" align="right"><%= ol.amount %></td>
      <td width="80" align="right"><%= price ol.product %></td>
      <td width="80" align="right"><%= money(
           ol.amount * ol.product.price) %></td>
    </tr>
  <% end %>
  <tr height="60">
    <td colspan="4">&nbsp;</td>
  </tr>
  <tr>
    <td colspan="3" align="right"><b>Total:</b></td>
    <td align="right"><%= money @price %></td>
  </tr>
</table>

<%= button_to 'Handled', :action=>'handled',:id=>@order %>
<%= button_to 'Remove', :action=>'remove',:id=>@order %>
```

As you can see, you first display the shipping address and billing address, then list all the items with quantity, price, and combined price. Finally, two buttons let the handler either remove or mark the order as handled.

Adding Some Authentication

You now have almost all functionality finished for the administration part of the Shoplet application. There's just a small piece missing. At the moment, anybody who knew the address could do anything they wanted with the shop, and because the addresses are easy to guess, this is no way to leave it. You've already prepared for adding authentication by creating the User model, and the scaffolds for handling these. Now you need to secure your actions. When you try to go to the admin parts of the application, you should be redirected to a login page, submit your username and password, and if it is correct you should be redirected back to the page you tried to access first. You'll accomplish this through controller filters.

Rails provides filters to let you perform some task before or after an action runs. This has profound implications and makes many tasks easy, not just authentication and security.

The first step you'll take is to create a new controller. This controller will be the base for all your protected controllers, and won't have any actions itself. Open up the file app/controllers/admin_controller.rb and write this into it:

```
class AdminController < ApplicationController
  before_filter :authentication

  private
  def authentication
    unless session[:user_id] && User.find_by_id(session[:user_id])
      flash[:notice] = "Please log in"
      redirect_to(:controller => 'auth', :action =>
                        'login', :into => url_for(params))
    else
      @loggedin = true
    end
  end
end
```

You first declare that the method called authentication should be called as a before_filter, which means it should execute before an action. You then define the method itself, and mark it as private. You first check if the parameter called :user_id in the session object is set, and if it is you also make sure there is an existing user with that ID. Otherwise you place a message in the flash and redirect to the login action on the auth controller (which you'll create soon). If the person is logged in, you just set the instance variable @loggedin to true.

Next, create a new controller by using the generate script:

```
jruby script/generate controller auth login logout
```

This creates a new controller called AuthController, with two actions called login and logout available to it. In this way, you get some things for free, including tests and default views. Open up the file app/controllers/auth_controller.rb and change it so it looks like this:

```
class AuthController < ApplicationController
  layout "admin"

  def login
    if request.post?
      if user = User.find_by_username_and_password(
                   params[:username],params[:password])
        session[:user_id] = user.id
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 69

```
        redirect_to params[:into] || {:controller => 'products'}
        return
      else
        flash[:error] = "Wrong username or password"
      end
    end
    @into = params[:into]
  end

  def logout
    session[:user_id] = nil
    redirect_to "/"
  end
end
```

Several interesting things are going on here. First of all, you use the standard `admin` layout, but you'll modify it so it only shows the links on the left if someone is logged in. Next, the `login` method will do two different things depending on if it's called using an HTTP `POST` or not. In this way you can let the view post information back to itself, and the `login` method will handle it differently. So, if there was a `POST`, you check the username and password provided. If they match, you set the session information and redirect either to the `into` parameter, or if there is no such parameter you redirect to the products controller instead. If the username or password doesn't match, you fall through, setting a `flash`. Then you do the same thing as if it was a `GET`, which is that you set the `@into` instance variable and display the view.

The `logout` method just wipes the session and redirects to the starting URL.

Next, let's take a look at the `login` view that can be found in `app/views/auth/login.rhtml`. It should look like this:

```
<h2>Please login with your username and password</h2>

<%= start_form_tag %>
<%= hidden_field_tag 'into', @into %>
<table>
  <tr>
    <td>Username:</td><td><%= text_field_tag 'username' %></td>
  </tr>
  <tr>
    <td>Password:</td><td><%= password_field_tag 'password' %></td>
  </tr>
  <tr>
    <td colspan="2" align="right"><%= submit_tag 'Login' %></td>
  </tr>
</table>
<%= end_form_tag %>
```

Here you start a new form, but use all the default parameters, which means the browser will `POST` it back to the same address. You set a hidden field with the `'into'` parameter and then ask for a username and password, display a login button, and end the form.

Now that you can make sure people can log in, you also need to modify all your controllers so they won't let anyone in if they haven't been authenticated. So, open up app/controllers/products_controller.rb, app/controllers/orders_controller.rb, and app/controllers/users_controller.rb, and change the first line by replacing the word ApplicationController with AdminController. The first line in the file for the ProductsController should look like this:

```
class ProductsController < AdminController
```

Because all three of our controllers inherit from the admin controller, the before_filter you applied earlier will act on all actions written in any of the controllers. The only thing left is to open the layout file called app/views/layouts/admin.rhtml and change it by replacing the part that looks like this:

```
<ul style="text-align: left;">
 <li><%= link_to 'Administrate products',
            {:controller => 'products'},:class => 'adminLink' %></li>
 <li><%= link_to 'Handle orders',
            {:controller => 'orders'},:class => 'adminLink' %></li>
 <li><%= link_to 'Authenticated users',
            {:controller => 'users'},:class => 'adminLink' %></li>
</ul>
```

Replace this part with an if statement that only shows this when the person is logged in:

```
<% if @loggedin %>
<ul style="text-align: left;">
 <li><%= link_to 'Administrate products',
             {:controller => 'products'},:class => 'adminLink' %></li>
 <li><%= link_to 'Handle orders',
             {:controller => 'orders'},:class => 'adminLink' %></li>
 <li><%= link_to 'Authenticated users',
             {:controller => 'users'},:class => 'adminLink' %></li>
</ul>
<br/>
<p><%= link_to 'Log out', :controller=>'auth',:action=>'logout'%></p>
<% end %>
```

You also add a small 'Log out' link here, if the person is logged in.

Now it's time to try it out. Remember that you added an "admin" user with password "admin" before? You should use this now, if you haven't already created an extra user. Try to visit http://localhost:3000/products and see what happens. Also try to log out and log in and add products. Everything should work fine as soon as you're logged in, but should stop working otherwise.

Functional Tests

Now the time has come to test a controller. You'll base the tests on the test code automatically generated for the products_controller, but it won't work in the current state, because you added authentication in the last section. So, first test that you can just run the tests and that they blow up in various interesting ways. However, before you do that, make sure the database migration is at the latest version. Do that by running this:

```
jruby -S rake db:migrate RAILS_ENV=test
```

Next, run the functional tests for `products_controller` by running this command:

```
jruby test/functional/products_controller_test.rb
```

Now, open up `test/functional/products_controller_test.rb` and begin the testing by adding two fixtures you're going to need. Add these lines:

```
fixtures :users
fixtures :product_types
```

Then open up the file `test/fixtures/users.yml` and change it so it looks like this:

```
admin:
  id: 1
  username: admin
  password: admin
other:
  id: 2
  username: other
  password: other
```

Next, change the `test_index` test case to make sure it redirects correctly to the `auth` controller, and also rename it to `test_index_without_login`. Do that by writing a method that looks like this:

```
def test_index_without_login
  get :index
  assert_redirected_to :controller=> 'auth', :action => 'login'
  assert_equal 'Please log in',flash[:notice]
end
```

As you can see, you first issue a `get` request for the index of the controller in question. You then make sure you've been redirected to the right place, and that the flash is also set.

To test a method that's protected by authentication, you'll use a trick. The `get` method in functional tests can take several optional parameters. One describes the parameters to set, and another describes what the session should look like. By adding this parameter, you can make it look like a user is logged in. Now create a new `test_index` method that looks like this:

```
def test_index
  get :index, {}, {:user_id => users(:admin).id}
  assert_response :success
  assert_template 'list'
end
```

The second parameter to get is the query string or post parameters, and because you don't want to add anything like these, you just provide an empty hash. The next parameter sets

881-4 CH04.qxd 8/19/07 3:11 PM Page 72

the variable :user_id in the session to a valid user ID, which means the action should succeed as expected.

You do the same transformation with test_list, test_show, and test_new, adding the empty hash if needed, and otherwise enclosing the parameters in an explicit hash.

Next you'll change the test_create method. Because a new product isn't valid without a few parameters, you'll change the line that posts information, and this results in a test_create method that looks like this:

```
def test_create
  num_products = Product.count

  post :create,{:product => {:product_type_id=>1,
        :name => 'abc', :price => '10.00'}},
        {:user_id => users(:admin).id}

  assert_response :redirect
  assert_redirected_to :action => 'list'

  assert_equal num_products + 1, Product.count
end
```

Here you have to provide a product_type_id, a name and a valid price, and the user_id for the session. You also make sure that the new product has been added by counting all available products before and after the action has been performed. You should also fix the tests test_edit, test_update, and test_destroy by adding the session parameter and enclosing the rest in an explicit hash. The end result of these methods should look like this:

```
def test_edit
  get :edit, {:id => 1}, {:user_id => users(:admin).id}

  assert_response :success
  assert_template 'edit'

  assert_not_nil assigns(:product)
  assert assigns(:product).valid?
end

def test_update
  post :update, {:id => 1}, {:user_id => users(:admin).id}
  assert_response :redirect
  assert_redirected_to :action => 'show', :id => 1
end

def test_destroy
  assert_not_nil Product.find(1)
```

881-4 CH04.qxd 8/19/07 3:11 PM Page 73

```
  post :destroy, {:id => 1}, {:user_id => users(:admin).id}
  assert_response :redirect
  assert_redirected_to :action => 'list'

  assert_raise(ActiveRecord::RecordNotFound) {
    Product.find(1)
  }
end
```

Now you can run the test again, and ideally it will result in much better output. To be able to run all tests, you need to modify the `orders_controller_test.rb` and `users_controller_test.rb` too, so the test cases handle authentication, and also so that the methods you removed aren't used in the tests. However, I leave that as an exercise for you. When that is done, you can always run your tests by just invoking `rake` in the directory of your application:

```
jruby -S rake
```

Summary

We've walked through the administration parts of a completely new application that could well be the backbone for an online shop application. Of course, some things are missing, such as further validations and more testing, but the core is there. I hope this has also served as a fast, practical introduction to many of the day-to-day tasks in creating a Rails application. The next chapter will talk about how the user interacts with this application; we'll create the code for that, and then I'll take a few pages to talk about different databases for a JRuby on Rails application.

881-4 CH04.qxd 8/19/07 3:11 PM Page 74