Chapter 6 Modeling and viewing users, part I

In Chapter 5, we ended with a stub page for creating new users (Section 5.3); over the course of the next three chapters, we'll fulfill the promise implicit in this incipient signup page. The first critical step is to create a *data model* for users of our site, together with a way to store that data. Completing this task is the goal for this chapter and the next (Chapter 7), and we'll give users the ability to sign up (Chapter 8). Once the sample application can create new users, we'll let them sign in and sign out (Chapter 9), and in Chapter 10 (Section 10.2) we'll learn how to protect pages from improper access.

Taken together, the material in Chapter 6 through Chapter 10 develops a full Rails login and authentication system. As you may know, there are various pre-built authentication solutions out there in Rails land; Box 6.1 explains why (at least at first) it's a good idea to roll your own.

Box 6.1.Roll your own authentication system

Virtually all web applications nowadays require a login and authentication system of some sort. Unsurprisingly, most web frameworks have a plethora of options for implementing such systems, and Rails is no exception. Examples of authentication and authorization systems include Clearance, Authlogic, Devise, and CanCan (as well as non-Rails-specific solutions built on top of OpenID or OAuth). It's reasonable to ask why we should reinvent the wheel. Why not just use an off-the-shelf solution instead of rolling our own?

There are several reasons why building our own authentication system is a good idea. First, there is no standard answer to Rails authentication; tying the tutorial to a specific project would leave us open to the risk that the our particular choice would go out of fashion or out of date. Moreover, even if we guessed right, the project's codebase would continue to evolve, rendering any tutorial explanation quickly obsolete. Finally, introducing all the authentication machinery at once would be a pedagogical disaster—to take one example, Clearance contains more than 1,000 lines of code and creates a complicated data model right from the start. Authentication systems are a challenging and rich programming exercise; rolling our own means that we can consider one small piece at a time, leading to a far deeper understanding—of both authentication and of Rails.

I encourage you to study Chapter 6 through Chapter 10 to give yourself a good foundation for future projects. When the time comes, if you decide to use an off-the-shelf authentication system for your own applications, you will be in a good position both to understand it and to tailor it to meet your specific needs.

In parallel with our data modeling, we'll also develop a web page for *showing* users, which will give us a first taste of the REST architecture for structuring web applications. Though we won't get very far in this chapter, our eventual goal for the user profile pages is to show the user's profile image, basic user data, and a list of microposts, as mocked up in Figure 6.1.1 (Figure 6.1 has our first example of *lorem ipsum* text, which has a fascinating story that you should definitely read about some time.) In this chapter, we'll lay the essential foundation for the user show page, and will start filling in the details starting in Chapter 7.

Figure 6.1: A mockup of our best guess at the user show page. (full size)

As usual, if you're following along using Git for version control, now would be a good time to make a topic branch for modeling users:

```
$ git checkout master
$ git checkout -b modeling-users
```

(The first line here is just to make sure that you start on the master branch, so that the `modeling-users` topic branch is based on `master`. You can skip that command if you're already on the master branch.)

6.1 User model

Although the ultimate goal of the next three chapters is to make a signup page for our site, it would do little good to accept signup information now, since we don't currently have any place to put it. Thus, the first step in signing up users is to make a data structure to capture and store their information. In Rails, the default data structure for a data model is called, naturally enough, a *model* (the M in MVC from Section 1.2.4). The default Rails solution to the problem of persistence is to use a *database* for long-term data storage, and the default library for interacting with the database is called *Active Record*.2

Active Record comes with a host of methods for creating, saving, and finding data objects, all without having to use the structured query language (SQL)3 used by relational databases. Moreover, Rails has a feature called *migrations* to allow data definitions to be written in pure Ruby, without having to learn an SQL data definition language (DDL).4 The effect is that Rails insulates you almost entirely from the details of the data store. In this book, by using SQLite for development and Heroku for deployment (Section 1.4), we have developed this theme even further, to the point where we barely ever have to think about how Rails stores data, even for production applications.5

6.1.1 Database migrations

You may recall from Section 4.4.5 that we have already encountered, via a custom-built `User` class, user objects with `name` and `email` attributes. That class served as a useful example, but it lacked the critical property of *persistence*: when we created a User object at the Rails console, it disappeared as soon as we exited. Our goal in this section is to create a model for users that won't disappear quite so easily.

As with the User class in Section 4.4.5, we'll start by modeling a user with two attributes, a `name` and an `email` address, the latter of which we'll use as a unique username.6 (We'll add a password attribute in Section 7.1.) In Listing 4.8, we did this with Ruby's `attr_accessor` keyword:

```
class User
  attr_accessor :name, :email
  .
  .
  .
end
```

In contrast, when using Rails to model users we don't need to identify the attributes explicitly. As noted briefly above, to store data Rails uses a relational database by default, which consists of *tables*

composed of data *rows*, where each row has *columns* of data attributes. For example, to store users with names and email addresses, we'll create a `users` table with `name` and `email` columns (with each row corresponding to one user). By naming the columns in this way, we'll let Active Record figure out the User object attributes for us.

Let's see how this works. (If this discussion gets too abstract for your taste, be patient; the console examples starting in Section 6.1.3 and the database browser screenshots in Figure 6.3 and Figure 6.8 should make things clearer.) In Section 5.3.1, recall that we created a Users controller (along with a `new` action) with the command

```
$ script/generate rspec_controller Users new
```

There is an analogous command for making a User model (Listing 6.1).

Listing 6.1. Generating a User model.

```
$ script/generate rspec_model User name:string email:string
```

(Note that, in contrast to the plural convention for controller names, model names are singular: a Users controller, but a User model.) By passing the optional parameters `name:string` and `email:string`, we tell Rails about the two attributes we want, along with what types those attributes have; compare this with including the action names in Listing 3.2 and Listing 5.22.

One of the results of the `generate` command in Listing 6.1 is a new file called a *migration*. Migrations provide a way to alter the structure of the database incrementally, so that our data model can adapt to changing requirements. In the case of the User model, the migration is created automatically by the model generation script; it creates a `users` table with two columns, `name` and `email`, as shown in Listing 6.2. (We'll see in Section 6.2.4 how to make a migration from scratch.)

Listing 6.2. Migration for the User model (to create a `users` table).
`db/migrate/<timestamp>_create_users.rb`

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.string :name
      t.string :email

      t.timestamps
    end
  end

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

Note that the name of the migration is prefixed by a *timestamp* based on when the migration was generated. In the early days of migrations, the filenames were prefixed with incrementing integers, which caused conflicts for collaborating teams when two programmers had migrations with the same number. Using timestamps conveniently avoids such collisions.

Let's focus on the `self.up` method, which uses a Rails method called `create_table` to create a

table in the database for storing users. (The use of `self` in `self.up` identifies it as a *class method*. This doesn't matter now, but we'll learn about class methods when we make one of our own in Section 7.2.4.) The `create_table` method accepts a block (Section 4.3.2) with one block variable, in this case called `t` (for "table"). Inside the block, the `create_table` method uses the `t` object to create `name` and `email` columns in the database, both of type `string`.[7] Here the table name is plural (`users`) even though the model name is singular (User), which reflects a linguistic convention followed by Rails: a model represents a single user, whereas a database table consists of many users. The final line in the block, `t.timestamps`, is a special command that creates two *magic columns* called `created_at` and `updated_at`, which are timestamps that automatically record when a given user is created and updated. (We'll see concrete examples of the magic columns starting in Section 6.1.3.) The full data model represented by this migration is shown in Figure 6.2.

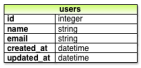

Figure 6.2: The users data model produced by Listing 6.2.

We can run the migration, known as "migrating up", using the `rake` command (Box 1.2) as follows:[8]

```
$ rake db:migrate
```

(You may recall that we have run this command before, in Section 1.2.5 and again in Chapter 2.) The first time `db:migrate` is run, it creates a file called `db/development.sqlite3`, which is an SQLite[9] database. We can see the structure of the database using the excellent SQLite Database Browser to open the `db/development.sqlite3` file (Figure 6.3); compare with the diagram in Figure 6.2. You might note that there's one column in Figure 6.3 not accounted for in the migration: the `id` column. As noted briefly in Section 2.2, this column is created automatically, and is used by Rails to identify each row uniquely.

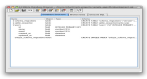

Figure 6.3: The SQLite Database Browser with our new `users` table. (full size)

You've probably inferred that running `db:migrate` executes the `self.up` command in the migration file. What, then, of `self.down`? As you might guess, `down` migrates *down*, reversing the effects of migrating up. In our case, this means *dropping* the `users` table from the database:

```
class CreateUsers < ActiveRecord::Migration
  .
  .
  .
  def self.down
    drop_table :users
  end
end
```

You can execute `down` with `rake` using the argument `db:rollback`:

```
$ rake db:rollback
```

This is often useful if you realize there's another column you want to add but don't want the trouble of making a new migration: you can roll back the migration, add the desired column, and then migrate

back up. (This isn't always convenient, and we'll learn how to add columns to an existing table in Section 7.1.2.)

If you rolled back the database, migrate up again before proceeding:

```
$ rake db:migrate
```

6.1.2 The model file

We've seen how the User model generation in Listing 6.1 generated a migration file (Listing 6.2), and we saw in Figure 6.3 the results of running this migration: it updated a file called `development.sqlite3` by creating a table `users` with columns `id`, `name`, `email`, `created_at`, and `updated_at`. Listing 6.1 also created the model itself; the rest of this section is dedicated to understanding it.

We begin by looking at the code for the User model, which lives in the file `user.rb` inside the `app/models/` directory; it is, to put it mildly, very compact (Listing 6.3).

Listing 6.3. The brand new User model.
`app/models/user.rb`

```
class User < ActiveRecord::Base
end
```

Recall from Section 4.4.2 that the syntax `class User < ActiveRecord::Base` means that the `User` class *inherits* from `ActiveRecord::Base`, so that the User model automatically has all the functionality of the `ActiveRecord::Base` class. Of course, knowledge of this inheritance doesn't do any good unless we know what `ActiveRecord::Base` contains, and we'll get a first taste starting momentarily. Before we move on, though, there are two tasks to complete.

Model annotation

Though it's not strictly necessary, I like to *annotate* my Rails models using the `annotate-models` gem:

```
$ [sudo] gem install annotate-models -v 1.0.4
```

This gem gives us a command called `annotate`, which simply adds the comments containing the data model to the model file:

```
$ annotate
Annotated User
```

The results appear in Listing 6.4.

Listing 6.4. The annotated User model.
`app/models/user.rb`

```
# == Schema Information
```

```
# Schema version: <timestamp>
#
# Table name: users
#
#  id         :integer         not null, primary key
#  name       :string(255)
#  email      :string(255)
#  created_at :datetime
#  updated_at :datetime
#

class User < ActiveRecord::Base
end
```

I find that having the data model visible in the model files is useful for reminding me which attributes the model has, but future code listings will usually omit the annotations for brevity.

Accessible attributes

Another step that isn't strictly necessary but is a really good idea is to tell Rails which attributes of the model are accessible, i.e., which attributes can be modified by outside users (such as users submitting requests with web browsers). We do this with the `attr_accessible` method (Listing 6.5). We'll see in Chapter 10 that using `attr_accessible` is important for preventing a *mass assignment* vulnerability, a distressingly common and often serious security hole in many Rails applications.

Listing 6.5. Making the `name` and `email` attributes accessible.

`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email
end
```

6.1.3 Creating user objects

We've done some good prep work, and now it's time to cash in and learn about Active Record by playing with our newly created User model. As in Chapter 4, our tool of choice is the Rails console. Since we don't (yet) want to make any changes to our database, we'll start the console in a *sandbox*:

```
$ script/console --sandbox
Loading development environment in sandbox (Rails 2.3.8)
Any modifications you make will be rolled back on exit
>>
```

As indicated by the helpful message "Any modifications you make will be rolled back on exit", when started in a sandbox the console will "roll back" (i.e., undo) any database changes introduced during the session.

When working at the console, it's useful to keep an eye on the *development log*, which records the actual low-level SQL statements being issued by Active Record, as shown in Figure 6.4. The way to get

this output at a Unix command line is to `tail` the log:

```
$ tail -f log/development.log
```

The `-f` flag ensures that `tail` will display additional lines as they are written. I recommend keeping a terminal window for tailing the log open whenever working at the console.

Figure 6.4: Tailing the development log. (full size)

In the console session in Section 4.4.5, we created a new user object with `User.new`, which we had access to only after requiring the example user file in Listing 4.8. With models, the situation is different; as you may recall from Section 4.4.4, the Rails console automatically loads the Rails environment, which includes the models. This means that we can make a new user object without any further work:

```
>> User.new
=> #<User id: nil, name: nil, email: nil, created_at: nil, updated_at: nil>
```

We see here the default console representation of a user object, which prints out the same attributes shown in Figure 6.3 and Listing 6.4.

When called with no arguments, `User.new` returns an object with all `nil` attributes. In Section 4.4.5, we designed the example User class to take an *initialization hash* to set the object attributes; that design choice was motivated by Active Record, which allows objects to be initialized in the same way:

```
>> user = User.new(:name => "Michael Hartl", :email => "mhartl@example.com")
#<User id: nil, name: "Michael Hartl", email: "mhartl@example.com",
created_at: nil, updated_at: nil>
```

Here we see that the name and email attributes have been set as expected.

If you've been tailing the development log, you may have noticed that no new lines have shown up yet. This is because calling `User.new` doesn't touch the database; it simply creates a new Ruby object in memory. To save the user object to the database, we call the `save` method on the `user` variable:

```
>> user.save
=> true
```

The `save` method returns `true` if it succeeds and `false` otherwise. (Currently, all saves should succeed; we'll see cases in Section 6.2 when some will fail.) As soon as you save, you should see a line in the development log with the SQL command to `INSERT INTO "users"`. Because of the many methods supplied by Active Record, we won't ever need raw SQL in this book, and I'll omit discussion of the SQL commands from now on. But you can learn a lot by watching the log.

You may have noticed that the new user object had `nil` values for the `id` and the magic columns `created_at` and `updated_at` attributes. Let's see if our `save` changed anything:

```
>> user
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2010-01-05 00:57:46", updated_at: "2010-01-05 00:57:46">
```

We see that the `id` has been assigned a value of `1`, while the magic columns have been assigned the

current time and date.[10] Currently the created and updated timestamps are identical; we'll see them differ in Section 6.1.5.

As with the User class in Section 4.4.5, instances of the User model allow access to their attributes using a dot notation:[11]

```
>> user.name
=> "Michael Hartl"
>> user.email
=> "mhartl@example.com"
>> user.updated_at
=> Tue, 05 Jan 2010 00:57:46 UTC +00:00
```

As we'll see in Chapter 8, it's often convenient to make and save a model in two steps as we have above, but Active Record also lets you combine them into one step with `User.create`:

```
>> User.create(:name => "A Nother", :email => "another@example.org")
=> #<User id: 2, name: "A Nother", email: "another@example.org", created_at:
"2010-01-05 01:05:24", updated_at: "2010-01-05 01:05:24">
>> foo = User.create(:name => "Foo", :email => "foo@bar.com")
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2010-01-05
01:05:42", updated_at: "2010-01-05 01:05:42">
```

Note that `User.create`, rather than returning `true` or `false`, returns the User object itself, which we can optionally assign to a variable (such as `foo` in the second command above).

The inverse of `create` is `destroy`:

```
>> foo.destroy
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2010-01-05
01:05:42", updated_at: "2010-01-05 01:05:42">
```

Oddly, `destroy`, like `create`, returns the object in question, though I can't recall ever having used the return value of `destroy`. Even odder, perhaps, is that the `destroy`ed object still exists in memory:

```
>> foo
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2010-01-05
01:05:42", updated_at: "2010-01-05 01:05:42">
```

How do we know if we really destroyed an object? And for saved and non-destroyed objects, how can we retrieve users from the database? It's time to learn how to use Active Record to find user objects.

6.1.4 Finding user objects

Active Record provides several options for finding objects. Let's use them to find the first user we created while verifying that the third user (`foo`) has been destroyed. We'll start with the existing user:

```
>> User.find(1)
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2010-01-05 00:57:46", updated_at: "2010-01-05 00:57:46">
```

Here we've passed the id of the user to `User.find`; Active Record returns the user with that `id` attribute.

Let's see if the user with an `id` of `3` still exists in the database:

```
>> User.find(3)
ActiveRecord::RecordNotFound: Couldn't find User with ID=3
```

Since we destroyed our third user in Section 6.1.3, Active Record can't find it in the database. Instead, `find` raises an *exception*, which is a way of indicating an exceptional event in the execution of a program—in this case, a nonexistent Active Record id, which causes `find` to raise an `ActiveRecord::RecordNotFound` exception.12

In addition to the generic `find`, Active Record also allows us to find users by specific attributes:

```
>> User.find_by_email("mhartl@example.com")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2010-01-05 00:57:46", updated_at: "2010-01-05 00:57:46">
```

Since we will be using email addresses as usernames, this sort of find will be useful when we learn how to let users sign in to our site (Chapter 8).13

We'll end with a couple of more general ways of finding users. First, there's `first`:

```
>> User.first
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2010-01-05 00:57:46", updated_at: "2010-01-05 00:57:46">
```

Naturally, `first` just returns the first user in the database. There's also `all`:

```
>> User.all
=> [#<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2010-01-05 00:57:46", updated_at: "2010-01-05 00:57:46">,
#<User id: 2, name: "A Nother", email: "another@example.org", created_at:
"2010-01-05 01:05:24", updated_at: "2010-01-05 01:05:24">]
```

No prizes for inferring that `all` returns an array (Section 4.3.1) of all users in the database.

6.1.5 Updating user objects

Once we've created objects, we often want to update them. There are two basic ways to do this. First, we can assign attributes individually, as we did in Section 4.4.5:

```
>> user          # Just a reminder about our user's attributes
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2010-01-05 00:57:46", updated_at: "2010-01-05 00:57:46">
>> user.email = "mhartl@example.net"
=> "mhartl@example.net"
>> user.save
=> true
```

Note that the final step is necessary to write the changes to the database. We can see what happens without a save by using `reload`, which reloads the object based on the database information:

```
>> user.email
=> "mhartl@example.net"
>> user.email = "foo@bar.com"
=> "foo@bar.com"
```

```
>> user.reload.email
=> "mhartl@example.net"
```

Now that we've updated the user, the magic columns differ, as promised in Section 6.1.3:

```
>> user.created_at
=> "2010-01-05 00:57:46"
>> user.updated_at
=> "2010-01-05 01:37:32"
```

The second way to update attributes is to use `update_attributes`:

```
>> user.update_attributes(:name => "The Dude", :email => "dude@abides.org")
=> true
>> user.name
=> "The Dude"
>> user.email
=> "dude@abides.org"
```

The `update_attributes` method accepts a hash of attributes, and on success performs both the update and the save in one step (returning `true` to indicate that the save went through). It's worth noting that, once you have defined some attributes as accessible using `attr_accessible` (Section 6.1.2.2), *only* those attributes can be modified using `update_attributes`. If you ever find that your models mysteriously start refusing to update certain columns, check to make sure that those columns are included in the call to `attr_accessible`.

6.2 User validations

The User model we created in Section 6.1 now has working `name` and `email` attributes, but they are completely generic: any string (including an empty one) is currently valid in either case. And yet, names and email addresses are more specific than this. For example, `name` should be non-blank, and `email` should match the specific format characteristic of email addresses. Moreover, since we'll be using email addresses as unique usernames when users sign in, we shouldn't allow email duplicates in the database.

In short, we shouldn't allow `name` and `email` to be just any strings; we should enforce certain constraints on their values. Active Record allows us to impose such constraints using *validations*. In this section we'll cover several of the most common cases, validating *presence*, *length*, *format* and *uniqueness*. In Section 7.1.1 we'll add a final common validation, *confirmation*. And we'll see in Section 8.2 how validations give us convenient error messages when users make submissions that violate them.

As with the other features of our sample app, we'll add User model validations using test-driven development. Since we've changed the data model, it's a good idea to prepare the test database before proceeding:

```
$ rake db:test:prepare
```

This just ensures that the data model from the development database, `db/development.sqlite3`, is reflected in the test database, `db/test.sqlite3`.

6.2.1 Validating presence

We'll start with a test for the presence of a `name` attribute. Although the first step in TDD is to write a *failing* test (Section 3.2.2), in this case we don't yet know enough about validations to write the proper test, so we'll write the validation first, using the console to understand it. Then we'll comment out the validation, write a failing test, and verify that uncommenting the validation gets the test to pass. This procedure may seem pedantic for such a simple test, but I have seen[14] *many* "simple" tests that test the wrong thing; being meticulous about TDD is simply the *only* way to be confident that we're testing the right thing. (This comment-out technique is also useful when rescuing an application whose application code is already written but—*quelle horreur!*—has no tests.)

The way to validate the presence of particular attribute is to use `validates_presence_of`, whose arguments are symbols indicating which attributes to validate—in this case, `name` (Listing 6.6).

Listing 6.6. Validating the presence of a `name` attribute.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  validates_presence_of :name
end
```

This may look like magic, but `validates_presence_of` is just a method, as indeed is `attr_accessible`; an equivalent formulation of Listing 6.6 is as follows:

```
class User < ActiveRecord::Base
  attr_accessible(:name, :email)

  validates_presence_of(:name)
end
```

Let's drop into the console to see the effects of adding a validation to our User model:[15]

```
$ script/console --sandbox
>> user = User.new(:name => "", :email => "mhartl@example.com")
>> user.save
=> false
>> user.valid?
=> false
```

Here `user.save` returns `false`, indicating a failed save. In the final command, we use the `valid?` method, which returns `false` when the object fails one or more validations, and `true` when all validations pass. (Recall from Section 4.2.3 that Ruby uses a question mark to indicate such true/false *boolean* methods.) In this case, we only have one validation, so we know which one failed, but it can still be helpful to check using the `errors` object generated on failure:

```
>> user.errors.full_messages
=> ["Name can't be blank"]
```

(The error message is a hint that Rails validates the presence of an attribute using the `blank?` method, which we saw at the end of Section 4.4.2.)

Now for the failing test. To ensure that our incipient test will fail, let's comment out the validation at

this point (Listing 6.7).

Listing 6.7. Commenting out a validation to ensure a failing test.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  # validates_presence_of :name
end
```

As in the case of controller generation (e.g., Listing 5.22), the model generate command in Listing 6.1 produces an initial spec for testing users. I've modified the default a bit to produce the initial tests shown in Listing 6.8.

Listing 6.8. The initial user spec (modified slightly from the default).
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before(:each) do
    @attr = { :name => "Example User", :email => "user@example.com" }
  end

  it "should create a new instance given valid attributes" do
    User.create!(@attr)
  end

  it "should require a name"
end
```

We've seen `require` and `describe` before, most recently in Listing 5.27. The next line is a `before(:each)` block; this was covered briefly in an exercise (Listing 3.25), and all it does is run the block before each test, in this case setting the `@attr` instance variable to an initialization hash.

The first test is just a sanity check, verifying that the User model is basically working. It uses `User.create!` (read "create bang"), which works just like the `create` method we saw in Section 6.1.3 except that it raises an `ActiveRecord::RecordInvalid` exception if the creation fails (similar to the `ActiveRecord::RecordNotFound` exception we saw in Section 6.1.4). As long as the attributes are valid, it won't raise any exceptions, and the test will pass.

The final line is the test for the presence of the `name` attribute—or rather, it *would* be the actual test, if it had anything in it. Instead, the test is just a stub, but a useful stub it is: it's a *pending spec*, which is a way to write a description of the application's behavior without worrying yet about the implementation. Pending specs are handled well by programs for running specs, as seen for Autotest in Figure 6.5. (The output of `spec spec/` is similarly useful.)

Figure 6.5: Autotest (via `autospec`) with a pending User spec. (full size)

In order to fill in the pending spec, we need a way to make an attributes hash with an invalid name. (The `@attr` hash is valid by construction, with a non-blank `name` attribute.) The `Hash` method `merge` does the trick, as we can see with `script/console`:

```
>> @attr = { :name => "Example User", :email => "user@example.com" }
=> {:name => "Example User", :email => "user@example.com"}
>> @attr.merge(:name => "")
=> {:name => "", :email => "user@example.com"}
```

With `merge` in hand, we're ready to make the new spec (using a trick I'll explain momentarily), as seen in Listing 6.9.

Listing 6.9. A failing test for validation of the `name` attribute.
`spec/models/user_spec.rb`

```
describe User do

  before(:each) do
    @attr = { :name => "Example User", :email => "user@example.com" }
  end
  .
  .
  .
  it "should require a name" do
    no_name_user = User.new(@attr.merge(:name => ""))
    no_name_user.should_not be_valid
  end
end
```

Here we use `merge` to make a new user called `no_name_user` with a blank name. The second line then uses the RSpec `should_not` method to verify that the resulting user is *not* valid. The trick I alluded to above is related to `be_valid`: we know from earlier in this section that a User object responds to the `valid?` boolean method. RSpec adopts the useful convention of allowing us to test *any* boolean method by dropping the question mark and prepending `be_`. In other words,

```
no_name_user.should_not be_valid
```

is equivalent to

```
no_name_user.valid?.should_not == true
```

Since it sounds more like natural language, writing `should_not be_valid` is definitely more idiomatically correct RSpec.

With that, our new test should fail, which we can verify with Autotest or by running the `user_spec.rb` file using the `spec` script:

```
$ spec spec/models/user_spec.rb
.F

1)
'User should require a name' FAILED
expected valid? to return false, got true
```

```
./spec/models/user_spec.rb:14:

2 examples, 1 failure
```

Now uncomment the validation (i.e., revert Listing 6.7 back to Listing 6.6) to get the test to pass:

```
$ spec spec/models/user_spec.rb
..

2 examples, 0 failures
```

Of course, we also want to validate the presence of email addresses. The test (Listing 6.10) is analogous to the one for the `name` attribute.

Listing 6.10. A test for presence of the `email` attribute.
`spec/models/user_spec.rb`

```
describe User do

  before(:each) do
    @attr = { :name => "Example User", :email => "user@example.com" }
  end
  .
  .
  .
  it "should require an email address" do
    no_email_user = User.new(@attr.merge(:email => ""))
    no_email_user.should_not be_valid
  end
end
```

The only trick to the implementation is knowing that `validates_presence_of` (like `attr_accessible`) can take multiple arguments, as seen in Listing 6.11.

Listing 6.11. Validating the presence of the `name` and `email` attributes.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  validates_presence_of :name, :email
end
```

Now all the tests should pass, and the “presence of” validations are complete.

6.2.2 Length validation

We’ve constrained our User model to require a name for each user, but we should go further: the user’s names will be displayed on the sample site, so we should enforce some limit on their length. With all the work we did in Section 6.2.1, this step is easy.

We start with a test. There’s no science to picking a maximum length; we’ll just pull `50` out of thin air

as a reasonable upper bound, which means verifying that names of 51 characters are too long (Listing 6.12).

Listing 6.12. A test for `name` length validation.
`spec/models/user_spec.rb`

```
describe User do

  before(:each) do
    @attr = { :name => "Example User", :email => "user@example.com" }
  end
  .
  .
  .
  it "should reject names that are too long" do
    long_name = "a" * 51
    long_name_user = User.new(@attr.merge(:name => long_name))
    long_name_user.should_not be_valid
  end
end
```

For convenience, we've used "string multiplication" in Listing 6.12 to make a string 51 characters long. We can see how this works using the console:

```
>> s = "a" * 51
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
>> s.length
=> 51
```

The test in Listing 6.12 should fail. To get it to pass, we need to know about the validation to constrain length, `validates_length_of`; in this case, we use a `:maximum` parameter to enforce the upper bound (Listing 6.13). (This is a one-element *options hash*; recall from Section 4.3.4 that curly braces are optional when passing hashes as the final argument in a method. As noted in Section 5.1.1, the use of options hashes is a recurring theme in Rails.)

Listing 6.13. Adding length validation for the `name` attribute.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  validates_presence_of :name, :email
  validates_length_of   :name, :maximum => 50
end
```

With our test suite passing again, we can move on to a more challenging validation: email format.

6.2.3 Format validation

Our validations for the `name` attribute enforce only minimal constraints—any non-blank name under 51 characters will do—but of course the `email` attribute must satisfy more stringent requirements. So

far we've only rejected blank email addresses; in this section, we'll require email addresses to conform to the familiar pattern `user@example.com`.

Neither the tests nor the validation will be exhaustive, just good enough to accept most valid email addresses and reject most invalid ones. We'll start with a couple tests involving collections of valid and invalid addresses. To make these collections, it's worth knowing about a useful method for making arrays of strings, as seen in this console session:

```
>> %w[foo bar baz]
=> ["foo", "bar", "baz"]
>> addresses = %w[user@foo.com THE_USER@foo.bar.org first.last@foo.jp]
=> ["user@foo.com", "THE_USER@foo.bar.org", "first.last@foo.jp"]
>> addresses.each do |address|
?>   puts address
>> end
user@foo.com
THE_USER@foo.bar.org
first.last@foo.jp
```

Here we've iterated over the elements of the `addresses` array using the `each` method (Section 4.3.2). With this technique in hand, we're ready to write some basic email format validation tests (Listing 6.14).

Listing 6.14. Tests for email format validation.
`spec/models/user_spec.rb`

```
describe User do

  before(:each) do
    @attr = { :name => "Example User", :email => "user@example.com" }
  end
  .
  .
  .
  it "should accept valid email addresses" do
    addresses = %w[user@foo.com THE_USER@foo.bar.org first.last@foo.jp]
    addresses.each do |address|
      valid_email_user = User.new(@attr.merge(:email => address))
      valid_email_user.should be_valid
    end
  end

  it "should reject invalid email addresses" do
    addresses = %w[user@foo,com user_at_foo.org example.user@foo.]
    addresses.each do |address|
      invalid_email_user = User.new(@attr.merge(:email => address))
      invalid_email_user.should_not be_valid
    end
  end
end
```

As noted above, these are far from exhaustive, but we do check the common valid email forms `user@foo.com`, `THE_USER@foo.bar.org` (uppercase, underscores, and compound domains), and `first.last@foo.jp` (the standard corporate username `first.last`, with a two-letter top-level domain `jp`), along with several invalid forms.

The application code for email format validation uses a *regular expression* (covered briefly in Section 5.3.1) to define the format, along with the Rails `validates_format_of` method (Listing 6.15).

Listing 6.15. Validating the email format with a regular expression.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  EmailRegex = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i

  validates_presence_of :name, :email
  validates_length_of   :name, :maximum => 50
  validates_format_of   :email, :with => EmailRegex
end
```

Here `EmailRegex` is a Ruby *constant*, as indicated by the leading capital letter.16 (Using a constant is simply conventional in this context; we could have used a variable, too.) The value of `EmailRegex` is a *regular expression*, also known as a *regex*. The code

```
  EmailRegex = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  .
  .
  .
  validates_format_of :email, :with => EmailRegex
```

ensures that only email addresses that match the pattern will be valid.

So, what of the pattern? Regular expressions consist of a terse (some would say unreadable) language for matching text patterns; learning to construct regexes is an art, and to get you started I've broken `EmailRegex` into bite-sized pieces (Table 6.1).17 To really learn about regular expressions, though, I consider the amazing Rubular regular expression editor (Figure 6.6) to be simply essential.18 The Rubular website has a beautiful interactive interface for making regular expressions, along with a handy regex quick reference. I encourage you to study Table 6.1 with a browser window open to Rubular—no amount of reading about regular expressions can replace a couple of hours playing with Rubular. (Video is a better format than print for understanding regexes, and I plan to cover them in more depth in the Rails Tutorial screencasts.)

Expression	Meaning
`/\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i`	full regex
`/`	start of regex
`\A`	match start of a string
`[\w+\-.]+`	at least one word character, plus, hyphen, or dot
`@`	literal "at sign"
`[a-z\d\-.]+`	at least one letter, digit, hyphen, or dot
`\.`	literal dot
`[a-z]+`	at least one letter
`\z`	match end of a string

`/`	end of regex
`i`	case insensitive

Table 6.1: Breaking down the email regex from Listing 6.15.

By the way, there actually exists a full regex for matching email addresses according to the official standard, but it's really not worth the trouble. The one in Listing 6.15 is fine, maybe even better than the official one.19

Figure 6.6: The awesome Rubular regular expression editor. (full size)

The tests should all be passing now. (In fact, the tests for valid email addresses should have been passing all along; since regexes are notoriously error-prone, the valid email tests are there mainly as a sanity check on `EmailRegex`.) This means that there's only one constraint left: enforcing the email addresses to be unique.

6.2.4 Uniqueness validation

To enforce uniqueness of email addresses (so that we can use them as usernames), we'll be using the method `validates_uniqueness_of`. But be warned: there's a *major* caveat, so don't just skim this section—read it carefully.

We'll start, as usual, with our tests. In our previous model tests, we've mainly used `User.new`, which just creates a Ruby object in memory, but for uniqueness tests we actually need to put a record into the database.20 The (first) duplicate email test appears in Listing 6.16.

Listing 6.16. A test for the rejection of duplicate email addresses.
`spec/models/user_spec.rb`

```
describe User do

  before(:each) do
    @attr = { :name => "Example User", :email => "user@example.com" }
  end
  .
  .
  .
  it "should reject duplicate email addresses" do
    # Put a user with given email address into the database.
    User.create!(@attr)
    user_with_duplicate_email = User.new(@attr)
    user_with_duplicate_email.should_not be_valid
  end
end
```

The method here is to create a user and then try to make another one with the same email address. (We use the noisy method `create!`, first seen in Listing 6.8, so that it will raise an exception if anything goes wrong. Using `create`, without the bang `!`, risks having a silent error in our test, a potential source of elusive bugs.) We can get this test to pass with the code in Listing 6.17.21

Listing 6.17. Validating the uniqueness of email addresses.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  validates_format_of     :email, :with => EmailRegex
  validates_uniqueness_of :email
end
```

We're not quite done, though. Email addresses are case-insensitive—`foo@bar.com` goes to the same place as `FOO@BAR.COM` or `FoO@BAr.coM`—so our validation should cover this case as well. We test for this with the code in Listing 6.18.

Listing 6.18. A test for the rejection of duplicate email addresses, insensitive to case.
`spec/models/user_spec.rb`

```
describe User do

  before(:each) do
    @attr = { :name => "Example User", :email => "user@example.com" }
  end
  .
  .
  .
  it "should reject email addresses identical up to case" do
    upcased_email = @attr[:email].upcase
    User.create!(@attr.merge(:email => upcased_email))
    user_with_duplicate_email = User.new(@attr)
    user_with_duplicate_email.should_not be_valid
  end
end
```

Here we are using the `upcase` method on strings (seen briefly in Section 4.3.2). This test does the same thing as the first duplicate email test, but with an upper-case email address instead. If this test feels a little abstract, go ahead and fire up the console:

```
$ script/console --sandbox
>> @attr = { :name => "Example User", :email => "user@example.com" }
=> {:name => "Example User", :email => "user@example.com"}
>> upcased_email = @attr[:email].upcase
=> "USER@EXAMPLE.COM"
>> User.create!(@attr.merge(:email => upcased_email))
>> user_with_duplicate_email = User.new(@attr)
>> user_with_duplicate_email.valid?
=> true
```

Of course, currently `user_with_duplicate_email.valid?` is `true`, since this is a failing test, but we want it to be `false`. Fortunately, `validates_uniqueness_of` accepts an option, `:case_sensitive`, for just this purpose (Listing 6.19).

Listing 6.19. Validating the uniqueness of email addresses, ignoring case.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  validates_format_of     :email, :with => EmailRegex
  validates_uniqueness_of :email, :case_sensitive => false
end
```

With that, our tests pass, and email uniqueness is validated.

The uniqueness caveat

There's just one small problem, the caveat alluded to above:

Using `validates_uniqueness_of` does not guarantee uniqueness.

D'oh! But what can go wrong? Here's what:

1. Alice signs up for the sample app, with address `alice@wonderland.com`.
2. Alice accidentally clicks on "Submit" *twice*, sending two requests in quick succession.
3. The following sequence occurs: request 1 creates a user in memory that passes validation, request 2 does the same, request 1's user gets saved, request 2's user gets saved.
4. Result: two user records with the exact same email address, despite the uniqueness validation.

If the above sequence seems implausible, believe me, it isn't: it happens on any Rails website with significant traffic.22 Luckily, the solution is straightforward to implement; we just need to enforce uniqueness at the database level as well. Our method is to create a database *index* on the email column, and then require that the index be unique.

The email index represents an update to our data modeling requirements, which (as discussed in Section 6.1.1) is handled in Rails using migrations. We saw in Section 6.1.1 that generating the User model automatically created a new migration (Listing 6.2); in the present case, we are adding structure to an existing model, so we need to create a migration directly using the `migration` generator:

```
$ script/generate migration add_email_uniqueness_index
```

Unlike the migration for users, the email uniqueness migration is not pre-defined, so we need to fill in its contents with Listing 6.20.23

Listing 6.20. The migration for enforcing email uniqueness.
`db/migrate/<timestamp>_add_email_uniqueness_index.rb`

```
class AddEmailUniquenessIndex < ActiveRecord::Migration
  def self.up
    add_index :users, :email, :unique => true
  end

  def self.down
    remove_index :users, :email
  end
end
```

This uses a Rails method called `add_index` to add an index on the `email` column of the `users` table. The index by itself doesn't enforce uniqueness, but the option `:unique => true` does.

The final step is to migrate the database:

```
$ rake db:migrate
```

Now the Alice scenario above will work fine: the database will save a user record based on the first request, and will reject the second save for violating the uniqueness constraint. (An error will appear in the Rails log, but that doesn't do any harm. You can actually catch the `ActiveRecord::StatementInvalid` exception that gets raised—see here for an example—but in this tutorial we won't bother with this step.) Adding this index on the email attribute accomplishes a second goal, alluded to briefly in Section 6.1.4: it fixes an efficiency problem in `find_by_email` (Box 6.2).

Box 6.2.Database indices

When creating a column in a database, it is important to consider if we will need to *find* records by that column. Consider, for example, the `email` attribute created by the migration in Listing 6.2. When we allow users to sign in to the sample app starting in Chapter 8, we will need to find the user record corresponding to the submitted email address; unfortunately, based on the naïve data model, the only way to find a user by email address is to look through *each* user row in the database and compare its email attribute to the given email. This is known in the database business as a *full-table scan*, and for a real site with thousands of users it is a Bad Thing.

Putting an index on the email column fixes the problem. To understand a database index, it's helpful to consider the analogy of a book index. In a book, to find all the occurrences of a given string, say "foobar", you would have to scan each page for "foobar". With a book index, on the other hand, you can just look up "foobar" in the index to see all the pages containing "foobar". (Of course, with *this* book you can always just search it electronically.) A database index works essentially the same way.

6.3 Viewing users

We're not quite done with the basic user model—we need to add passwords, a task for Chapter 7—but we do have enough in place to make a minimalist page for showing user information. This will allow a gentle introduction to the REST style of organizing the actions for our site's users. Since this is just a rough demonstration for now, there are no tests in this section; we'll add tests when we flesh out the user view in Section 7.3.

6.3.1 Debug and Rails environments

As preparation for adding dynamic pages to our sample application, now is a good time to add some debug information to our site layout (Listing 6.21). This displays some useful information about each page using the built-in `debug` method and `params` variable (which we'll learn more about in Section 6.3.2), as seen in Figure 6.7.

Listing 6.21. Adding some debug information to the site layout.
`app/views/layouts/application.html.erb`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  .
  .
  .
  <body>
    <div class="container">
      .
      .
      .
      <%= render 'layouts/footer' %>
      <%= debug(params) if Rails.env.development? %>
    </div>
  </body>
</html>
```

Since we don't want to display debug information to users of a deployed application, we use

```
if Rails.env.development?
```

to restrict the debug information to the *development environment*. Though we've seen evidence of Rails environments before (most recently in Section 6.1.3), this is the first time it has mattered to us.

Figure 6.7: The sample application Home page (/) with debug information at the bottom. (full size)

Rails comes equipped with three environments: `test`, `development`, and `production`.[24] The default environment for the Rails console is `development`:

```
$ script/console
Loading development environment (Rails 2.3.8)
>> Rails.env
=> "development"
>> Rails.env.development?
=> true
>> Rails.env.test?
=> false
```

As you can see, Rails provides a `Rails` object with an `env` attribute and associated environment boolean methods. In particular, `Rails.env.development?` is `true` only in a development environment, so the Embedded Ruby

```
<%= debug(params) if Rails.env.development? %>
```

won't be inserted into production applications or tests. (Inserting the debug information into tests probably doesn't do any harm, but it probably doesn't do any good, either, so it's best to restrict the debug display to development only.)

If you ever need to run a console in a different environment (to debug a test, for example), you can pass the environment as a parameter to the `console` script:

```
$ script/console test
Loading test environment (Rails 2.3.8)
>> Rails.env
=> "test"
```

As with the console, `development` is the default environment for the local Rails server, but you can also run it in a different environment:

```
$ script/server --environment production
```

If you view your app running in production, it won't work without a production database, which we can create by running `rake db:migrate` in production:

```
$ rake db:migrate RAILS_ENV=production
```

(I find it confusing that the console, server, and migrate commands specify non-default environments in three mutually incompatible ways, which is why I bothered showing all three.)

By the way, if you have deployed your sample app to Heroku, you can see its environment using the `heroku` gem, which provides its own (remote) console:

```
$ heroku console
Ruby console for yourapp.heroku.com
>> Rails.env
=> "production"
>> Rails.env.production?
=> true
```

Naturally, since Heroku is a platform for production sites, it runs each application in a production environment.

6.3.2 User model, view, controller

In order to make a page to view a user, we'll use the User *model* to put a user into the database, make a *view* to display some user information, and then add an action to the Users *controller* to handle the browser request. In other words, for the first time in this tutorial, we'll see in one place all three elements of the model-view-controller architecture first discussed in Section 1.2.4.

Our first step is to create a user using the console, which we'll take care *not* to start in a sandbox since this time the whole point is to save a record to the database:

```
$ script/console
Loading development environment (Rails 2.3.8)
>> User.create!(:name => "Michael Hartl", :email => "mhartl@example.com")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2010-01-07 23:05:14", updated_at: "2010-01-07 23:05:14">
```

To double-check that this worked, let's look at the row in the development database using the SQLite Database Browser (Figure 6.8). Note that the columns correspond to the attributes of the data model defined in Section 6.1.

Figure 6.8: A user row in the SQLite database `db/development.sqlite3`. (full size)

Next comes the view, which is minimalist to emphasize that this is just a demonstration (Listing 6.22). We use the standard Rails location for showing a user, `app/views/users/show.html.erb`; unlike the `new.html.erb` view, which we created with the generator in Listing 5.22, the `show.html.erb` file doesn't currently exist, so you'll have to create it by hand.

Listing 6.22. A stub view for showing user information.
`app/views/users/show.html.erb`

```
<%= @user.name %>, <%= @user.email %>
```

This view uses Embedded Ruby to display the user's name and email address, assuming the existence of an instance variable called `@user`. Of course, eventually the real user show page will look very different, and won't display the email address publicly.

Finally, we'll add the `show` action to the Users controller (corresponding to the `show.html.erb` view) with the code in Listing 6.23, which defines the `@user` instance variable needed by the view.

Listing 6.23. The Users controller with a `show` action.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController

  def show
    @user = User.find(params[:id])
  end

  def new
    @title = "Sign up"
  end
end
```

Here we've gotten a little ahead of ourselves by using the standard Rails `params` object to retrieve the user id. When we make the appropriate request to the Users controller, `params[:id]` will be the user id `1`, so the effect is the same as the `find` command

```
User.find(1)
```

we saw in Section 6.1.4.

With the `show` action defined as in Listing 6.23, we can view the user show page at the url `/users/show/1` (Figure 6.9), which adds `id` to the traditional `/controller/action/id` pattern we first encountered in Section 3.1.2.

Figure 6.9: The user show page at `/users/show/1`. (full size)

If you look closely at the debug information in Figure 6.9, you'll see the following:[25]

```
--- !map:HashWithIndifferentAccess
action: show
id: "1"
controller: users
```

This is a representation of `params` (which is basically a hash), and we see that its `id` attribute is `"1"`. This is why

```
User.find(params[:id])
```

finds the user with id `1`. (The `find` method knows how to convert the string `"1"` into the integer `1`.)

6.3.3 A Users resource

Although the user show URL in Section 6.3.2 works fine for now, it doesn't fit in well with the REST architecture favored in Rails applications. This style is based on the ideas of *representational state transfer* identified and named by computer scientist Roy Fielding in his doctoral dissertation *Architectural Styles and the Design of Network-based Software Architectures*.[26] The REST design style emphasizes representing data as *resources* that can be created, shown, updated, or destroyed—four actions corresponding to the four fundamental operations `POST`, `GET`, `PUT`, and `DELETE` defined by the HTTP standard (Box 3.1).

When following REST principles, resources are typically referenced using the resource name and a unique identifier. What this means in the context of users—which we're now thinking of as a Users *resource*—is that we should view the user with id `1` by issuing a `GET` request to the URL `/users/1`. Here the `show` action is *implicit* in the type of request—when Rails' REST features are activated, `GET` requests are *automatically* handled by the `show` action.

Unfortunately, the URL `/users/1` doesn't work quite yet (Figure 6.10). By default, Rails only recognizes the `/controller/action/id` pattern, but recall that Rails URLs are determined by a routes file (Section 5.2.2). We can get the REST-style Users URL to work by adding users as a resource to `config/routes.rb` (Listing 6.24).

Figure 6.10: The initial effect of hitting `/users/1`. (full size)

Listing 6.24. Adding a Users resource to the routes file.
`config/routes.rb`

```
ActionController::Routing::Routes.draw do |map|
  map.resources :users
  .
  .
  .
end
```

Now that we've added the Users resource, the URL `/users/1` works perfectly (Figure 6.11).

Figure 6.11: The user show page at `/users/1` after adding a Users resource. (full size)

The one additional resource line in Listing 6.24 doesn't just add a working `/users/1` URL; it endows our sample application with all the actions needed for a RESTful Users resource,27 along with a large number of named routes (Section 5.2.3) for generating user URLs. The resulting correspondence of URLs, actions, and named routes is shown in Table 6.2. (Compare to Table 2.2.) Over the course of the next three chapters, we'll cover all of the other entries in Table 6.2 as we fill in all the actions necessary to make users a fully RESTful resource.28

HTTP request	URL	Action	Named route	Purpose
`GET`	`/users`	`index`	`users_path`	page to list all users
`GET`	`/users/1`	`show`	`user_path(1)`	page to show user with id `1`
`GET`	`/users/new`	`new`	`new_user_path`	page to make a new user (signup)
`POST`	`/users`	`create`	`users_path`	create a new user
`GET`	`/users/1/edit`	`edit`	`edit_user_path(1)`	page to edit user with id `1`
`PUT`	`/users/1`	`update`	`user_path(1)`	update user with id `1`
`DELETE`	`/users/1`	`destroy`	`user_path(1)`	delete user with id `1`

Table 6.2: RESTful routes provided by the Users resource in Listing 6.24.

6.4 Conclusion

This chapter is the first half of the two-step process of creating a working User model. Our users now have `name` and `email` attributes, together with validations enforcing several important constraints on their values. We've also taken a first small step toward a working user show page and a Users resource based on the principles of representational state transfer (REST). In Chapter 7, we'll complete the process by adding user passwords and a more useful user view.

If you're using Git, now would be a good time to commit if you haven't done so in a while:

```
$ git add .
$ git commit -am "Finished first cut of the User model"
```

6.5 Exercises

1. Read through the Rails API entry for `ActiveRecord::Base` to get a sense of its capabilities.
2. Study the entries in the Rails API for each of the `validates_` methods in this chapter to learn more about their capabilities and options.
3. Spend a couple hours playing with Rubular.

« Chapter 5 Filling in the layout Chapter 7 Modeling and viewing users, part II »

1. Mockingbird doesn't support custom images like the profile photo in Figure 6.1; I put that in by hand using Adobe Fireworks. The hippo here is from http://www.flickr.com/photos/43803060@N00/24308857/. ↑
2. The name comes from the "active record pattern", identified and named in *Patterns of Enterprise Application Architecture* by Martin Fowler. ↑
3. Pronounced "ess-cue-ell", though the alternate pronunciation "sequel" is also common. ↑
4. In its earliest incarnations, Rails did require knowledge of an SQL DDL. Even after Rails added migrations, setting up the old default database (MySQL) was quite involved. Happily, as noted in Section 1.2.5, Rails now uses SQLite by default, which stores its data as a simple file—no setup required. ↑
5. Occasionally, it is necessary to pierce this abstraction layer, but one design goal of this tutorial is to make all the code database-independent. (Indeed, this is a worthy goal in general.) In case you ever do need to write database-specific code to deploy on Heroku, you should know that they use the excellent PostgreSQL ("post-gres-cue-ell") database. PostgreSQL is free, open-source, and cross-platform; if you develop PostgreSQL-specific applications, you can install it locally, and configure Rails to use it in development by editing the `config/database.yml` file. Such configuration is beyond the scope of this tutorial, but there are lots of resources on the web; use a search engine to find the most up-to-date information for your platform. ↑
6. By using an email address as the username, we open the theoretical possibility of communicating with our users at a future date. ↑
7. Don't worry about exactly how the `t` object manages to do this; the beauty of *abstraction layers* is that we don't have to know. We can just trust the `t` object to do its job. ↑
8. We'll see how to migrate up on a remote Heroku server in Section 7.4.2. ↑
9. Officially pronounced "ess-cue-ell-ite", although the (mis)pronunciation "sequel-ite" is also common. ↑
10. In case you're curious about `"2010-01-05 00:57:46"`, I'm not writing this after midnight; the timestamps are recorded in Coordinated Universal Time (UTC), which for most practical purposes is the same as Greenwich Mean Time. From the NIST Time and Frequency FAQ: **Q:** Why is UTC used as the acronym for Coordinated Universal Time instead of CUT? **A:** In 1970 the Coordinated Universal Time system was devised by an international advisory group of technical experts within the International Telecommunication Union (ITU). The ITU felt it was best to designate a single abbreviation for use in all languages in order to minimize confusion. Since unanimous agreement could not be achieved on using either the English word order, CUT, or the French word order, TUC, the acronym UTC was chosen as a compromise. ↑
11. Note the value of `user.updated_at`. Told you the timestamp was in UTC. ↑
12. Exceptions and exception handling are somewhat advanced Ruby subjects, and we won't need them much in this book. They are important, though, and I suggest learning about them using one of the Ruby books recommended in Section 1.1.1. ↑
13. To those worried that `find_by_email` will be inefficient if there are a large number of users, you're ahead of the game. We'll cover this issue, and its solution via database indices, in Section 6.2.4. ↑
14. and written ↑
15. I'll omit the output of console commands when they are not particularly instructive—for example, the results of `User.new`. ↑
16. It's not a coincidence that it looks like a `ClassName`—class names are constants, too. ↑
17. Note that, in Table 6.1, "letter" really means "lower-case letter", but the `i` at the end of the regex enforces case-insensitive matching. ↑
18. If you find it as useful as I do, I encourage you to donate to Rubular to reward developer Michael Lovitt for his wonderful work. ↑

19. Did you know that `"Michael Hartl"@example.com`, with quotation marks and a space in the middle, is a valid email address according to the standard? Incredibly, it is—but it's absurd. If you don't have an email address that contains only letters, numbers, underscores, and dots, then get one. N.B. The regex in Listing 6.15 allows plus signs, too, because Gmail (and possibly other email services) does something useful with them: for example, to filter orders from Amazon, you can use `username+amazon@gmail.com`, which will go to the Gmail address `username@gmail.com`, allowing you to filter on the string `amazon`. ↑
20. As noted briefly in the introduction to this section, there is a dedicated test database, `db/test.sqlite3`, for this purpose. ↑
21. If you're wondering why the `create!` line in Listing 6.8 doesn't cause this to fail by creating a duplicate user, it's because Rails tests are *transactional*: each test is wrapped in a transaction, which *rolls back* the database after the test executes. This way, each test runs against a fresh database. ↑
22. Yes, it happened to me. How do you think I found out about this issue? ↑
23. Of course, we could just edit the migration file for the `users` table in Listing 6.2 but that would require rolling back and then migrating back up. The Rails Way is to use migrations every time we discover that our data model needs to change. ↑
24. You can define your own custom environments as well; see the Railscast on adding an environment for details. ↑
25. The Rails `debug` information is shown as YAML (a recursive acronym standing for "YAML Ain't Markup Language"), which is a friendly data format designed to be both machine- *and* human-readable. ↑
26. Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. ↑
27. This means that the *routing* works, but the corresponding pages don't necessarily work at this point. For example, `/users/1/edit` gets routed properly to the `edit` action of the Users controller, but since the `edit` action doesn't exist yet actually hitting that URL will return an error. ↑
28. Note that Table 6.2 includes the URL `/users/new`, which actually worked fine in Section 5.3.1. This is because `/users/new` matches both the `/controller/action` pattern and the RESTful routes pattern. ↑