Chapter 7 Modeling and viewing users, part II

In Chapter 6, we created the first iteration of a User model to represent users of our application, but the job is only half-done. Virtually any website with users, including ours, needs *authentication* as well, but currently any user signing up for the site would only have a name and email address, with no way to verify their identity. In this chapter, we'll add the `password` attribute needed for an initial user signup (Chapter 8) and for signing in with an email/password combination (Chapter 9). In the process, we'll re-use several of the ideas from Chapter 6, including migrations and validations, and also introduce some new ideas such as virtual attributes, private methods, and Active Record callbacks.

Once we have a working `password` attribute, we'll make a working action and view for showing user profiles (Section 7.3). By the end of the chapter, our user profiles will display names and profile photos (as indicated by the mockup in Figure 7.1), and they will be nicely tested with user factories and RSpec stubs.

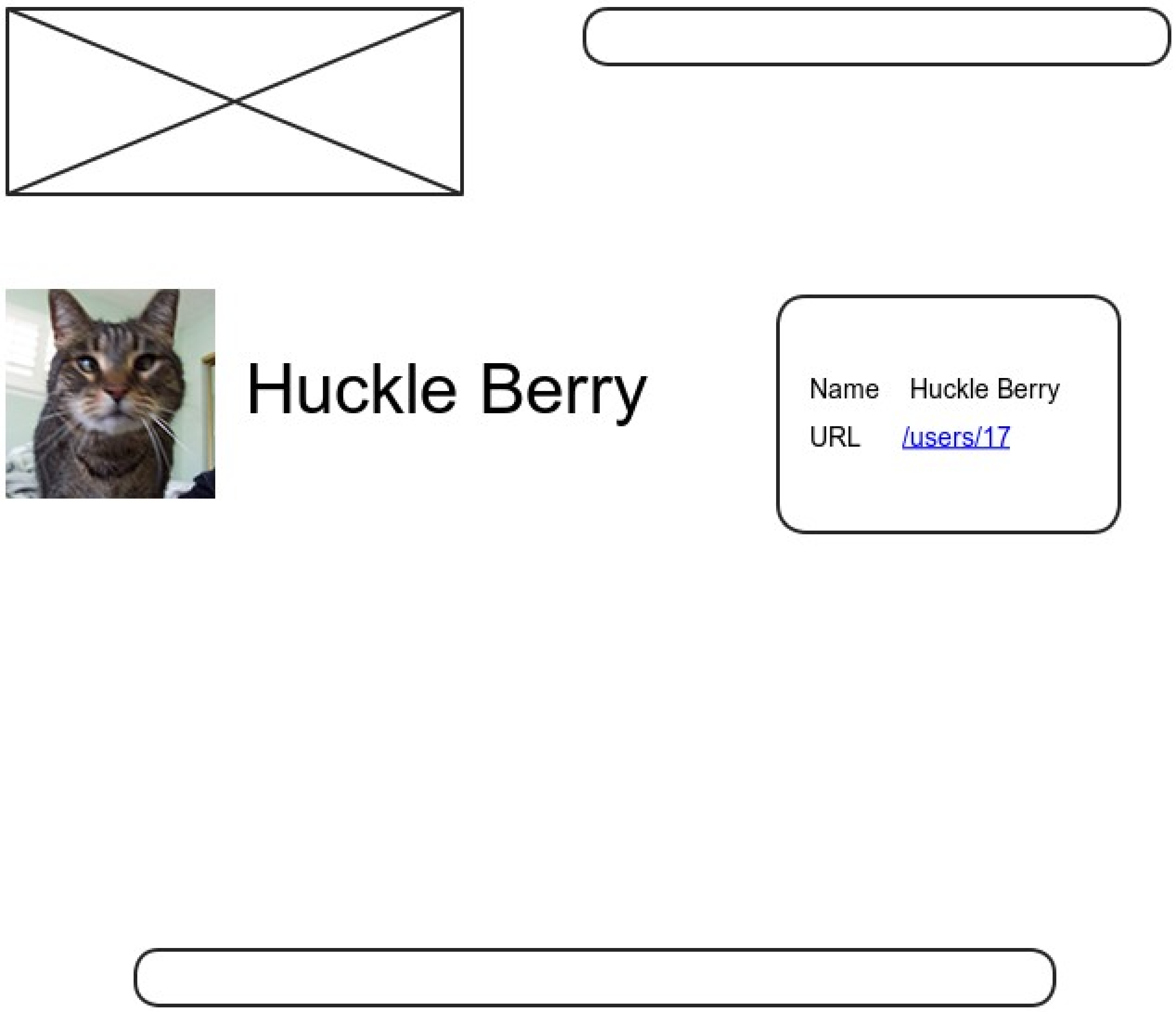


Figure 7.1: A mockup of the user profile made in Section 7.3. (full size)

7.1 Insecure passwords

Making industrial-strength passwords requires a lot of machinery, so we’ll break the process into two main steps. In this section, we’ll make a `password` attribute and add validations. The resulting User model will be functionally complete but badly insecure, with the passwords stored as plain text in the database. In Section 7.2, we’ll fix this problem by encrypting the passwords before saving them, thereby protecting our site against potential attackers.

7.1.1 Password validations

Even though we have yet even to add a column for passwords to our database, we’re already going to start writing tests for them. Our initial plan is to have tests to validate the presence, length, and confirmation of passwords. This is our biggest single block of tests so far, so see if you can read it all in one go. If you get stuck, it might help to review the analogous validations from Section 6.2 or skip ahead to the application code in Listing 7.2.

In order to minimize typos in passwords, when making a user signup page in Chapter 8 we'll adopt the common convention of requiring that users *confirm* their passwords. To get started, let's review the user attributes hash last seen in Listing 6.18:

```
describe User do

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

To write tests for passwords, we'll need to add *two* new attributes to the `@attr` hash, `password` and `password_confirmation`. As you can probably guess, the `password_confirmation` attribute will be used for the password confirmation step.

Let's write tests for the presence of the password and its confirmation, together with tests confirming that the password is a valid length (restricted somewhat arbitrarily to be between 6 and 40 characters long). The results appear in Listing 7.1.

Listing 7.1. Tests for password validations.
`spec/models/user_spec.rb`

```
describe User do

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

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

    it "should require a password" do
      User.new(@attr.merge(:password => "", :password_confirmation => "")).
        should_not be_valid
    end

    it "should require a matching password confirmation" do
      User.new(@attr.merge(:password_confirmation => "invalid")).
        should_not be_valid
    end

    it "should reject short passwords" do
      short = "a" * 5
      hash = @attr.merge(:password => short, :password_confirmation => short)
      User.new(hash).should_not be_valid
    end

    it "should reject long passwords" do
      long = "a" * 41
      hash = @attr.merge(:password => long, :password_confirmation => long)
      User.new(hash).should_not be_valid
    end
  end
end
```

Note in Listing 7.1 how we first collect a set of valid user attributes in `@attr`. If for some reason those attributes aren't valid—as would be the case, for example, if we didn't implement password confirmations properly—then the first test

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

would catch the error. The subsequent tests then check each validation in turn, using the same `@attr.merge` technique first introduced in Listing 6.9.

Now for the application code, which contains a trick. Actually, it contains *two* tricks. First, you might expect at this point that we would run a migration to add a `password` attribute to the User model, as we did with the `name` and `email` attributes in Listing 6.1. But this is not the case: we will store only an *encrypted* password in the database; for the password, we will introduce a *virtual attribute* (that is, an attribute not corresponding to a column in the database) using the `attr_accessor` keyword, much as we did with the original `name` and `email` attributes for the example user in Section 4.4.5. The `password` attribute will not ever be written to the database, but will exist only in memory for use in performing the password confirmation step (implemented below) and the encryption step (implemented in Section 7.1.2 and Section 7.2).

The second trick is that we will *not* introduce a `password_confirmation` attribute, not even a virtual one. Instead, we will use the special validation

```
validates_confirmation_of :password
```

which will *automatically* create a virtual attribute called `password_confirmation`, while confirming that it matches the `password` attribute at the same time.

Thus prepared to understand the implementation, let's take a look at the code itself (Listing 7.2).

Listing 7.2. Validations for the `password` attribute.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessor :password
  attr_accessible :name, :email, :password, :password_confirmation
  .
  .
  .
  # Automatically create the virtual attribute 'password_confirmation'.
  validates_confirmation_of :password

  # Password validations.
  validates_presence_of :password
  validates_length_of   :password, :within => 6..40
end
```

As promised, we use `attr_accessor :password` to create a virtual `password` attribute (as in Section 4.4.5), and include `validates_confirmation_of` to reject users whose password and password confirmations don't match. We also see `validates_presence_of`, which we first saw in Listing 6.6. Finally, we see a second application of `validates_length_of`; previously, we constrained the `name` attribute to be 50 characters or less, using the `:maximum` option:

```
validates_length_of :name, :maximum => 50
```

For the password length validation, instead we've used the `:within` option, passing it the *range*1 `6..40` to enforce the desired length constraints.

Finally, since we'll be accepting passwords and password confirmations as part of the signup process in Chapter 8, we need to add the password and its confirmation to the list of accessible attributes (first mentioned in Section 6.1.2.2), which we've done in the line

```
attr_accessible :name, :email, :password, :password_confirmation
```

7.1.2 A password migration

At this point, you may be concerned that we're not storing user passwords anywhere; since we've elected to use a virtual password, rather than storing it in the database, it exists only in memory. How can we use this password for authentication? The solution is to create a separate attribute dedicated to password storage, and our strategy will be to use the virtual password as raw material for an *encrypted password*, which we *will* store in the database upon user signup (Chapter 8) and retrieve later for use in user authentication (Chapter 9).

Let's plan to store the encrypted password using an `encrypted_password` attribute in our User model. We'll discuss the implementation details in Section 7.2, but we can get started with our encrypted password tests by noting that the encrypted password should at the least *exist*. We can test this using the Ruby method `respond_to?`, which accepts a symbol and returns `true` if the object responds to the given method or attribute and `false` otherwise:

```
$ script/console --sandbox
>> user = User.new
>> user.respond_to?(:password)
=> true
>> user.respond_to?(:encrypted_password)
=> false
```

We can test the existence of an `encrypted_password` attribute with the code in Listing 7.3, which uses RSpec's `respond_to` helper method.

Listing 7.3. Testing for the existence of an `encrypted_password` attribute.
`spec/models/user_spec.rb`

```
describe User do
  .
  .
  .
  describe "password encryption" do

    before(:each) do
      @user = User.create!(@attr)
    end

    it "should have an encrypted password attribute" do
      @user.should respond_to(:encrypted_password)
    end
  end
end
```

Note that in the `before(:each)` block we *create* a user, rather than just calling `User.new`. We could actually get this test to pass using `User.new`, but (as we'll see momentarily) *setting* the encrypted password will require that the user be saved to the database. Using `create!` in this first case does no harm, and putting it in `before(:each)` will allow us to keep all the encrypted password tests in one `describe` block.

To get this test to pass, we'll need a migration to add the `encrypted_password` attribute to the `users` table:

```
$ script/generate migration add_password_to_users encrypted_password:string
```

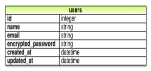
Here the first argument is the migration name, and we've also supplied a second argument with the name and type of attribute we want to create. (Compare this to the original generation of the `users` table in Listing 6.1.) We can choose any migration name we want, but it's convenient to end the name with `_to_users`, since in this case Rails can automatically construct a migration to add columns to the `users` table. Moreover, by including the second argument, we've given Rails enough information to construct the entire migration for us, as seen in Listing 7.4.

Listing 7.4. The migration to add an `encrypted_password` column to the `users` table.
`db/migrate/<timestamp>_add_password_to_users.rb`

```
class AddPasswordToUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :encrypted_password, :string
  end

  def self.down
    remove_column :users, :encrypted_password
  end
end
```

This code uses the `add_column` method to add an `encrypted_password` column to the `users` table (and the complementary `remove_column` method to remove it when migrating down). The result is the data model shown in Figure 7.2.



users	
id	integer
name	string
email	string
encrypted_password	string
created_at	datetime
updated_at	datetime

Figure 7.2: The User model with an added (encrypted) password attribute.

Now if we run the migration and prepare the test database, the test should pass, since the User model will respond to the `encrypted_password` attribute. (Be sure to close any Rails consoles started in a sandbox; the sandbox locks the database and prevents the migration from going through.)

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

We can of course run the full test suite with `spec spec/`, but sometimes it's convenient to run just *one* RSpec example, which we can do with the `-e` ("example") flag:

```
$ spec spec/models/user_spec.rb \
> -e "should have an encrypted password attribute"
.

1 example, 0 failures
```

7.1.3 An Active Record callback

Now that our User model has an attribute for storing the password, we need to arrange to generate and save the encrypted password when Active Record saves the user to the database. We'll do this with a technique called a *callback*, which is a method that gets invoked at a particular point in the lifetime of an Active Record object. In the present case, we'll use a `before_save` callback to create `encrypted_password` just before the user is saved.2

We start with a test for the encrypted password attribute. Since we're deferring the implementation details—and, in particular, the method of encryption—to Section 7.2, in this section we'll just make sure that a saved user's `encrypted_password` attribute is not blank. We do this by combining the `blank?` method on strings (Section 4.4.2) with the RSpec convention for boolean methods (first seen in the context of `valid?`/`be_valid` in Listing 6.9), yielding the test in Listing 7.5.

Listing 7.5. Testing that the `encrypted_password` attribute is nonempty.
`spec/models/user_spec.rb`

```
describe User do
  .
  .
  .
  describe "password encryption" do

    before(:each) do
      @user = User.create!(@attr)
    end
    .
    .
    .
    it "should set the encrypted password" do
```

```
      @user.encrypted_password.should_not be_blank
    end
  end
end
```

This code verifies that `encrypted_password.blank?` is not true using the construction `should_not be_blank`.

To get this test to pass, we *register* a callback called `encrypt_password` by passing a symbol of that name to the `before_save` method, and then define an `encrypt_password` method to perform the encryption. With the `before_save` in place, Active Record will automatically call the corresponding method before saving the record. The result appears in Listing 7.6.

Listing 7.6. A `before_save` callback to create the `encrypted_password` attribute.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  before_save :encrypt_password

  private

    def encrypt_password
      self.encrypted_password = encrypt(password)
    end

    def encrypt(string)
      string # Only a temporary implementation!
    end
end
```

Here the `encrypt_password` callback delegates the actual encryption to an `encrypt` method; as noted in the comment, this is only a temporary implementation—as currently constructed, Listing 7.6 simply sets the encrypted to the *unencrypted* password, which kind of defeats the purpose. But it's enough to get our test to pass, and we'll make the `encrypt` method do some actual encryption in Section 7.2.

Before trying to understand the implementation, first note that the encryption methods appear after the `private` keyword; inside a Ruby class, all methods defined *after* `private` are used internally by the object and are not intended for public use.[3] For an example, we can look at a User object in the console:

```
>> user = User.new
>> user.encrypt_password
NoMethodError: Attempt to call private method
```

Here Ruby raises a `NoMethodError` exception and issues a warning that the method is private.

In the present context, making the `encrypt_password` and `encrypt` methods private isn't strictly necessary, but it's a good practice to make them private unless they are needed for the public interface.[4]

Now that we understand the `private` keyword, let's take another look at the `encrypt_password` method:

```
def encrypt_password
  self.encrypted_password = encrypt(password)
end
```

This is a one-line method (the best kind!), but it contains not one but *two* subtleties. First, the left-hand side of the statement explicitly assigns the `encrypted_password` attribute using the `self` keyword. (Recall from Section 4.4.2 that inside the class `self` refers to the object itself, which for the User model is just the user.) The use of `self` is *required* in this context; if we omitted `self` and wrote

```
def encrypt_password
  encrypted_password = encrypt(password)
end
```

Ruby would create a *local variable* called `encrypted_password`, which isn't what we want at all.

Second, the right hand side of the assignment calls `encrypt` on `password`, but there is no `password` in sight. In the console, we would access the password attribute through a user object

```
>> user = User.new(:password => "foobar")
>> user.password
=> "foobar"
```

Inside the User class, the user object is just `self`, and we could write

```
def encrypt_password
  self.encrypted_password = encrypt(self.password)
end
```

in analogy with the console example, just replacing `user` with `self`. But the `self` is optional, so for brevity we can write simply

```
def encrypt_password
  self.encrypted_password = encrypt(password)
end
```

as in Listing 7.6 above. (Of course, as we've noted, the `self` is *not* optional when assigning to an attribute, so we have to write `self.encrypted_password` in this case.)

7.2 Secure passwords

With the code from Section 7.1, in principle we are done: although the "encrypted" password is the same as the unencrypted password, as long as we are willing to store unencrypted passwords in the database we have the necessary foundation for user login and authentication.[5] Our standards in *Rails Tutorial* are much loftier, though: any web developer worth his salt should know how to implement a password system with *secure one-way hashing*. In this section we will build on the material from Section 7.1 to implement just such an industrial-strength password system.

7.2.1 A secure password test

As hinted at in Section 7.1.3, all of the machinery for password encryption will be tucked away in the `private` regions of the User model, which presents a challenge for testing it. What we need is some sort of *public interface* that we can expose to the rest of the application. One useful aspect of test-driven development is that, by acting as a client for our application code, the tests motivate us to design a useful interface right from the start.

Authenticating users involves comparing the encrypted version of a submitted password to the (encrypted) password of a given user. This means we need to define some method to perform the comparison, which we'll call `has_password?`; this will be our public interface to the encryption machinery.6 The `has_password?` method will test whether a user has the same password as one submitted on a sign-in form (to be written in Chapter 9); a skeleton method for `has_password?` appears in Listing 7.7.

Listing 7.7. A `has_password?` method for users.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  before_save :encrypt_password

  # Return true if the user's password matches the submitted password.
  def has_password?(submitted_password)
    # Compare encrypted_password with the encrypted version of
    # submitted_password.
  end

  private
  .
  .
  .
end
```

With this method, we can write tests as in Listing 7.8, which uses the RSpec methods `be_true` and `be_false` to test that `has_password?` returns `true` or `false` in the proper cases.

Listing 7.8. Tests for the `has_password?` method.
`spec/models/user_spec.rb`

```
describe User do
  .
  .
  .
  describe "password encryption" do

    before(:each) do
      @user = User.create!(@attr)
    end
    .
    .
    .
    describe "has_password? method" do

      it "should be true if the passwords match" do
        @user.has_password?(@attr[:password]).should be_true
      end

      it "should be false if the passwords don't match" do
        @user.has_password?("invalid").should be_false
      end
    end
  end
end
```

In Section 7.2.3, we’ll complete the implementation of `has_password?` (and get the test to pass in the process). But first we need to learn a little more about secure passwords.

7.2.2 Some secure password theory

The basic idea of encrypted passwords is simple: rather than storing a raw password in the database (known as “cleartext”), we store a string generated using a cryptographic hash function, which is essentially irreversible, so that even an attacker in possession of the hashed password will be unable to infer the original. To verify that a submitted password matches the user’s password, we first encrypt the submitted string and then compare the hashes. Let’s drop into a console session to see how this works:

```
$ script/console
>> require 'digest'
>> def secure_hash(string)
>>   Digest::SHA2.hexdigest(string)
>> end
=> nil
>> password = "secret"
=> "secret"
>> encrypted_password = secure_hash(password)
=> "2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b"
>> submitted_password = "secret"
=> "secret"
>> encrypted_password == secure_hash(submitted_password)
=> true
```

Here we’ve defined a function called `secure_hash` that uses a cryptographic hash function called SHA2, part of the SHA family of hash functions, which we include into Ruby through the `digest` library.7 It’s not important to know exactly how these hash functions work; for our purposes what’s important is that they are one-way: there is no computationally tractable way to discover that

```
2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b
```

is the SHA2 hash of the string `"secret"`.

If you think about it, though, we still have a problem: if an attacker ever got hold of the hashed passwords, he would still have a chance at discovering the originals. For example, he could guess that we used SHA2, and so write a program to compare a given hash to the hashed values of potential passwords:

```
>> hash = "2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b"
>> secure_hash("secede") == hash
=> false
>> secure_hash("second") == hash
=> false
>> secure_hash("secret") == hash
=> true
```

So our attacker has a match—bad news for any users with password `"secret"`. This technique is known as a *rainbow attack*.

To foil a potential rainbow attack, we can use a *salt*, which is a different unique string for each user.8 One common way to (nearly) ensure uniqueness is to hash the current time (in UTC to be time-zone-independent) along with the password, so that two users will have the same salt only if they are created at exactly the same time *and* have the same password. Let's see how this works using the `secure_hash` function defined in the console above:

```
>> Time.now.utc
=> Fri Jan 29 18:11:27 UTC 2010
>> password = "secret"
=> "secret"
>> salt = secure_hash("#{Time.now.utc}#{password}")
=> "d1c2f8e1a4f5caed8818fcaf72117e65631e3168af4adcfda5bcfa2c5a36d311"
>> encrypted_password = secure_hash("#{salt}#{password}")
=> "918574661e80aacd3d81a380f5d22489eeb1e2f0bd33bb6575e8a64ac5a5aecc"
```

In the last line, we've hashed the salt with the password, yielding an encrypted password that is virtually impossible to crack.

7.2.3 Implementing `has_password?`

Having finished with the theory, we're now ready for the implementation. Let's look ahead a little to see where we're going. Each user object knows its own encrypted password, so to check for a match with a submitted password we can define `has_password?` as follows:

```
def has_password?(submitted_password)
  encrypted_password == encrypt(submitted_password)
end
```

As long as we encrypt the submitted password using the same salt used to encrypt the original password, this function will be true if and only if the submitted password matches.

Since comparing a user password with a submitted password will involve encrypting the submitted password with the salt, we need to store the salt somewhere, so the first step is to add a `salt` column to the `users` table:

```
$ script/generate migration add_salt_to_users salt:string
```

As with the `encrypted_password` migration (Section 7.1.2), this migration has a name that ends in `_to_users` and passes a second argument containing the attribute name and type, so Rails automatically constructs the right migration (Listing 7.9).

Listing 7.9. The migration to add a `salt` column to the `users` table.
`db/migrate/<timestamp>_add_salt_to_users.rb`

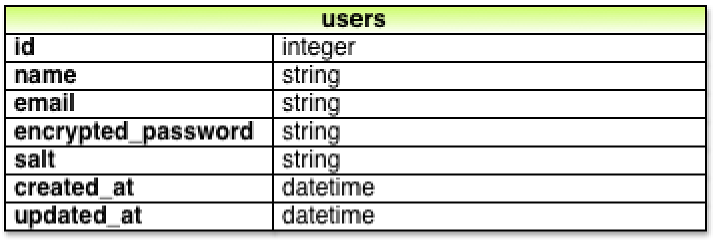
```
class AddSaltToUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :salt, :string
  end

  def self.down
    remove_column :users, :salt
  end
end
```

Then we migrate the database and prepare the test database as usual:

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

The result is a database with the data model shown in Figure 7.3.



users	
id	integer
name	string
email	string
encrypted_password	string
salt	string
created_at	datetime
updated_at	datetime

Figure 7.3: The User model with an added salt.

Finally, we're ready for the full implementation. When last we saw the `encrypt` function (Listing 7.6), it did nothing, simply returning the string in its argument. With the ideas from Section 7.2.2, we're now in a position to use a secure hash instead (Listing 7.10).[9]

Listing 7.10. The `has_password?` method with secure encryption.
`app/models/user.rb`

```
require 'digest'
class User < ActiveRecord::Base
  .
  .
  .
  before_save :encrypt_password

  def has_password?(submitted_password)
    encrypted_password == encrypt(submitted_password)
  end

  private

    def encrypt_password
      self.salt = make_salt
      self.encrypted_password = encrypt(password)
    end
    def encrypt(string)
      secure_hash("#{salt}#{string}")
    end

    def make_salt
      secure_hash("#{Time.now.utc}#{password}")
    end

    def secure_hash(string)
      Digest::SHA2.hexdigest(string)
    end
end
```

This code contains the same two subtleties mentioned in Section 7.1.3, namely, the assignment to an Active Record attribute in the line

```
self.salt = make_salt
```

and the omission of the `self` keyword in the `encrypt` method:

```
def encrypt(string)
  secure_hash("#{salt}#{string}")
end
```

Since we're inside the User class, Ruby knows that `salt` refers to the user's `salt` attribute.

At this point, the tests from Listing 7.8 should pass:10

```
$ spec spec/models/user_spec.rb -e "should be true if the passwords match"
.

1 example, 0 failures

$ spec spec/models/user_spec.rb -e "should be false if the passwords don't match"
.

1 example, 0 failures
```

7.2.4 An authenticate method

Having a `has_password?` method for each user is nice, but by itself it isn't very useful. We'll end our discussion of passwords by using `has_password?` to write a method to authenticate a user based on an email/password combination. In Chapter 9, we'll use this `authenticate` method when signing users in to our site.

We can get a hint of how this will work by using the console. First, we'll create a user, and then retrieve that user by email address to verify that it has a given password:11

```
$ script/console --sandbox
>> User.create(:name => "Michael Hartl", :email => "mhartl@example.com",
?>             :password => "foobar", :password_confirmation => "foobar")
>> user = User.find_by_email("mhartl@example.com")
>> user.has_password?("foobar")
=> true
```

Using these ideas, let's write a method that will return an authenticated user on password match, and `nil` otherwise. We should be able to use the resulting `authenticate` class method as follows:

```
User.authenticate(email, submitted_password)
```

We start with the tests, which we'll use to specify the behavior we expect from `User.authenticate`. There are three cases to check: `authenticate` (1) should return `nil` when the email/password combination is invalid or (2) when no user exists with the given email address, and (3) should return the user object itself on success. With this information, we can write the tests for `authenticate` as in Listing 7.11.

Listing 7.11. Tests for the `User.authenticate` method.
`spec/models/user_spec.rb`

```
describe User do
  .
  .
  .
  describe "password encryption" do
    .
    .
    .
    describe "authenticate method" do

      it "should return nil on email/password mismatch" do
        wrong_password_user = User.authenticate(@attr[:email], "wrongpass")
        wrong_password_user.should be_nil
      end

      it "should return nil for an email address with no user" do
        nonexistent_user = User.authenticate("bar@foo.com", @attr[:password])
        nonexistent_user.should be_nil
      end

      it "should return the user on email/password match" do
        matching_user = User.authenticate(@attr[:email], @attr[:password])
        matching_user.should == @user
      end
    end
  end
end
```

Now we're ready for the implementation, which will get our tests to pass and show how to define a *class method* as a bonus. We've mentioned class methods several times before, most recently in Section 6.1.1; a class method is simply a method attached to a class, rather than an instance of that class. For example, `new`, `find`, and `find_by_email` are all class methods on the User class. Outside of the class, they are invoked using the class name, as in `User.find`, but inside the class we can omit the class name.

Box 7.1.What is `self`?

We've talked about how `self` is "the object itself", but exactly what that means depends on context. Inside of an ordinary method, `self` refers to an *instance* of the class, that is, the object itself. For example, in Listing 7.10, `self` is a *user*:

```
def encrypt_password
  self.salt = make_salt
  self.encrypted_password = encrypt(password)
end
```

Inside the `encrypt_password` method, `self` is a user object, so `self.salt` is the same as `user.salt` outside the method:

```
$ script/console
>> user = User.first
>> user.salt
=> "d3b9af261c502947fbf32f78cb8179b16e62eabacf059451efee404328b2f537"
```

On the other hand, Listing 7.12 shows the definition of `authenticate`, which uses `self` to define a *class method*; here, `self` is the `User` class itself:

```
def self.authenticate(email, submitted_password)
  .
  .
  .
end
```

Because it is defined on the `User` class, `authenticate` gets invoked directly on `User`:

```
>> user = User.authenticate('example@railstutorial.org', 'foobar')
>> user.name
=> "Example User"
```

It's worth noting two alternative ways of defining an `authenticate` class method equivalent to the one shown in Listing 7.12. First, we could indicate the `User` class explicitly by name:

```
def User.authenticate(email, submitted_password)
  .
  .
  .
end
```

(Some people might find this syntax clearer, but it's not as idiomatically correct.) Second, we could use the following code, which quite frankly melts my brain:

```
class << self
  def authenticate(email, submitted_password)
    .
    .
    .
  end
end
```

The weird `class << self` starts a block in which all new methods are automatically class methods. I find this syntax rather confusing, but it's possible you'll encounter it in others' code, so it's worth knowing what it does. (I recommend *The Well-Grounded Rubyist* by David A. Black if you want to dig into Ruby details like this one.)

The way to define a class method is to use the `self` keyword in the method definition. (This `self` is not the same as the `self` shown in Listing 7.10; see Box 7.1.) Listing 7.12 shows this construction in the context of the `authenticate` method. Note the call to `find_by_email`, in which we omit the explicit `User` class name since this method is already inside the `User` class.

Listing 7.12. The `User.authenticate` method.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  def has_password?(submitted_password)
    encrypted_password == encrypt(submitted_password)
  end

  def self.authenticate(email, submitted_password)
    user = find_by_email(email)
    return nil  if user.nil?
    return user if user.has_password?(submitted_password)
  end

  private
  .
  .
  .
end
```

There are several equivalent ways to write the `authenticate` method, but I find the implementation above the clearest. It handles two cases (invalid email and a successful match) with explicit `return` keywords, and handles the third case (password mismatch) implicitly, since in that case we reach the end of the method, which automatically returns `nil`. See Section 7.5 for some of the other possible ways to implement this method.

7.3 Better user views

Now that User model is effectively complete,12 we are in a position to add a sample user to the development database and make a `show` page to show some of that user's information. Along the way, we'll add some tests to the Users controller spec started in Section 5.3.1.

Before continuing, it's helpful to see where we left off by recalling what the Users controller spec looks like right now (Listing 7.13). Our tests for the user show page will follow this example, but we'll find that, unlike the tests for the `new` action, the tests for the `show` action will require the use of an instance of the User model. We'll meet this challenge using *factories* and *stubs*.

Listing 7.13. The current Users controller spec.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe UsersController do
  integrate_views
  describe "GET 'new'" do

    it "should be successful" do
      get 'new'
      response.should be_success
    end

    it "should have the right title" do
      get 'new'
      response.should have_tag("title", /Sign up/)
    end
  end
end
```

7.3.1 Testing the user show page (with factories)

One goal of automated testing is to test each component separately if possible, so that, for example, model tests query the database (since that's one of the things models do) but controller/view tests do not.[13] As seen in Listing 7.14, the Users controller `show` action needs an instance of the User model, so the tests for this action will require that we create an `@user` variable somehow. We'll accomplish this with a user *factory*, which is a convenient way to insert a user object into our test database.[14]

Listing 7.14. The user `show` action from Listing 6.23.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController

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

We'll be using the factories generated by Factory Girl,[15] a Ruby gem produced by the good people at thoughtbot. As with other Ruby gems, we can install it using the `gem` command:[16]

```
$ [sudo] gem install factory_girl -v 1.2.3
```

Next, we need to tell Rails about Factory Girl, which we can do by using a special gem configuration command in the environment file for our tests, as seen in Listing 7.15.[17]

Listing 7.15. Including the Factory Girl gem into the test environment file.
`config/environments/test.rb`

```
# Settings specified here will take precedence over those in
# config/environment.rb

config.gem 'factory_girl'
```

Now we’re ready to create the file `spec/factories.rb` and define a User factory, as shown in Listing 7.16. By putting the `factories.rb` file in the `spec/` directory, we arrange for RSpec to load our factories automatically whenever the tests run.

Listing 7.16. A factory to simulate User model objects.
`spec/factories.rb`

```
# By using the symbol ':user', we get Factory Girl to simulate the User model.
Factory.define :user do |user|
  user.name                  "Michael Hartl"
  user.email                 "mhartl@example.com"
  user.password              "foobar"
  user.password_confirmation "foobar"
end
```

With the definition in Listing 7.16, we can create a User factory in the tests like this:

```
@user = Factory(:user)
```

As noted in the comment in the first line of Listing 7.16, by using the symbol `:user` we ensure that Factory Girl will guess that we want to use the User model, so in this case `@user` will simulate an instance of `User`.

To use our new User factory in the Users controller spec, we’ll create an `@user` variable in a `before(:each)` block, stub it out (details below), and then `get` the show page and verify success (just as we did with the `new` page in Listing 7.13). The result appears in Listing 7.17.

Listing 7.17. A test for `get`ting the user `show` page, with a factory and a stub.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe UsersController do
  integrate_views

  describe "GET 'show'" do

    before(:each) do
      @user = Factory(:user)
      # Arrange for User.find(params[:id]) to find the right user.
      User.stub!(:find, @user.id).and_return(@user)
    end

    it "should be successful" do
      get :show, :id => @user
      response.should be_success
    end
  end
  .
  .
  .
end
```

Apart from the first use of a factory, the real novelty here is the use of a *stub*, which is a facility provided by RSpec to avoid hits to the database. The code

```
User.stub!(:find, @user.id).and_return(@user)
```

ensures that any call to `User.find` with the given `id` will return `@user`. Since this is just what we have in the application code (Listing 7.14), the stub will cause RSpec to intercept the call to `User.find` and, instead of hitting the database, returning `@user` instead.

There are two other details worth noting. First, in the call to `get`, the test uses the *symbol* `:show` instead of the string `'show'`, which is different from the convention in the other tests (for example, in Listing 3.7 we wrote `get 'home'`). Both

```
get :show
```

and

```
get 'show'
```

do the same thing, but when testing the canonical REST actions (Table 6.2) I prefer to use symbols, which for some reason feel more natural in this context.18 Second, note that the value of the hash key `:id`, instead of being the user's `id` attribute `@user.id`, is the user object itself:

```
get :show, :id => @user
```

We could use the code

```
get :show, :id => @user.id
```

to accomplish the same thing, but in this context Rails automatically converts the user object to the corresponding id.19 Using the more succinct construction

```
get :show, :id => @user
```

is a very common Rails idiom.

Because of the code we added in Listing 6.23, the test in this section already passes. If you're feeling paranoid, you can comment out the line

```
@user = User.find(params[:id])
```

and verify that the test fails, then uncomment it to get it to pass. (We went through this same process once before, in Section 6.2.1.)

7.3.2 A name and a Gravatar

In this section, we'll improve the look of the user show page by adding a heading with the user's name and profile image. This is one of those situations where I can go either way on test-driven development, and often when making views I'll experiment with the HTML before bothering with tests. Let's stick with the TDD theme for now, and test for a second-level heading (`h2` tag) containing the user's name and an `img` tag of class `gravatar`. (We'll talk momentarily about what this second part means.)

To view a working user show page in a browser, we'll need to create a sample user in the development database. To do this, first reset the database with `rake db:reset`, which will clear out any old sample users from previous sessions, and then start the console (*not* in a sandbox this time) and create the user:

```
$ rake db:reset
$ script/console
>> User.create(:name => "Example User", :email => "user@example.com",
?>             :password => "foobar", :password_confirmation => "foobar")
```

The tests in this section are similar to the tests for the `new` page seen in Listing 5.25. In particular, we use the `have_tag` method to check the title and the content of the `h2` tag, as seen in Listing 7.18.

Listing 7.18. Tests for the user show page.
`spec/controllers/users_controller_spec.rb`

```
require 'spec_helper'

describe UsersController do
  integrate_views

  describe "GET 'show'" do
    .
    .
    .
    it "should have the right title" do
      get :show, :id => @user
      response.should have_tag("title", /#{@user.name}/)
    end

    it "should include the user's name" do
      get :show, :id => @user
      response.should have_tag("h2", /#{@user.name}/)
    end

    it "should have a profile image" do
      get :show, :id => @user
      response.should have_tag("h2>img", :class => "gravatar")
    end
  end
  .
  .
  .
end
```

The first new construction is

```
/#{@user.name}/
```

to interpolate the user name into a regular expression. This works much like

```
"#{@name} <#{@email}>"
```

from Listing 4.8, which interpolates the name and email into a string. RSpec's `have_tag` method uses the resulting regular expression to test for a match, in this case verifying the presence of an `h2` tag containing the user's name. The other new element is the code `h2>img`, which makes sure that the `img` tag is *inside* the `h2` tag.20

We can get the first test to pass by setting the `@title` variable for use in the `title` helper

(Section 4.1.1), in this case setting it to the user's name (Listing 7.19).

Listing 7.19. A title for the user show page.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController

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

This code introduces a potential problem: a user could enter a name with malicious code—called a cross-site scripting attack—which would be injected into our application by the `title` helper defined in Listing 4.2. The solution is to *escape* potentially problematic code using the `h` method (short for `html_escape`), as seen in Listing 7.20.21

Listing 7.20. The title helper using `h` to escape the HTML.
`app/helpers/application_helper.rb`

```
module ApplicationHelper

  # Return a title on a per-page basis.
  def title
    base_title = "Ruby on Rails Tutorial Sample App"
    if @title.nil?
      base_title
    else
      "#{base_title} | #{h(@title)}"
    end
  end
end
```

The `h` method converts, e.g., `<script>` to `&lt;script&gt;`, rendering any malicious scripts completely harmless.

Now for the other tests. Creating an `h2` with the (escaped) user name is easy (Listing 7.21).

Listing 7.21. The user show view with the user's name.
`app/views/users/show.html.erb`

```
<h2>
  <%= h @user.name %>
</h2>
```

Getting the `img` test to pass is trickier. The first step is to install a plugin to handle each user's Gravatar,22 which is a "globally recognized avatar":23

```
$ script/plugin install \
> svn://rubyforge.org/var/svn/gravatarplugin/plugins/gravatar
```

(If you get an error complaining that the `svn` command isn't found, you may need to install Subversion for your system.)

Gravatars are a convenient way to include user profile images without going through the trouble of managing image upload, cropping, and storage.24 The Gravatar plugin comes with a helper method called `gravatar_for`, which we'll use in our user show view, as seen in Listing 7.22. The result (after stopping and restarting the development server to load the plugin) appears in Figure 7.4, which shows our example user with the default Gravatar image. And since the `gravatar_for` helper automatically assigns its `img` tag the class `gravatar`, the tests from Listing 7.18 should now pass.

Listing 7.22. The user show view with name and Gravatar.
`app/views/users/show.html.erb`

```
<h2>
  <%= gravatar_for @user %>
  <%= h @user.name %>
</h2>
```

Figure 7.4: The initial user show page `/users/1` with the default Gravatar. (full size)

If this Gravatar business seems like magic, worry not. Let's fire up the console to get a little more insight into what's going on:

```
>> user = User.first
>> user.update_attributes(:email => "example@railstutorial.org",
?>                        :password => "foobar",
?>                        :password_confirmation => "foobar")
=> true
```

Note that we can pull out the first (and, at this point, only) user in the database with the handy `User.first` method. In the `update_attributes` step we've reassigned the user's email address, changing it to `example@railstutorial.org`. As you can see from Figure 7.5, this change results in a new Gravatar being displayed: the Rails Tutorial logo. What's going on is that Gravatar works by associating images with email addresses; since `user@example.com` is an invalid email address (the example.com domain is reserved for examples), there is no Gravatar for that email address. But at my Gravatar account I've associated the address `example@railstutorial.org` with the Rails Tutorial logo, so when updating the example user with that email address the Gravatar changes automatically.

Figure 7.5: The user show page `/users/1` with the Rails Tutorial Gravatar. (full size)

If you're wondering how the `gravatar_for` helper works, you can get a hint from Listing 7.23, which is an excerpt from the Gravatar plugin. You can see that `gravatar_for` calls the `gravatar` method (not shown) with the argument `user.email`. In other words, since our User model already *has* an `email` attribute, the code `gravatar_for(@user)` Just Works.™ (What would you do if there were no `email` attribute on the User model? That's a good question, and it's left as an exercise (Section 7.5).)

Listing 7.23. The definition of `gravatar_for` from the Gravatar plugin source code.
`vendor/plugins/gravatar/lib/gravatar.rb`

```
    .
    .
    .
    # Return the HTML img tag for the given user's gravatar. Presumes that
    # the given user object will respond_to "email", and return the user's
    # email address.
    def gravatar_for(user, options={})
      gravatar(user.email, options)
    end
    .
    .
    .
```

7.3.3 A user sidebar

Even though our tests are passing, and the user show page is much-improved, it's still nice to polish it up just a bit more. In Listing 7.24, we have a `table` tag with one table row (`tr`) and two table data cells (`td`).25

Listing 7.24. The user show page `/users/1` with a sidebar.
`app/views/users/show.html.erb`

```
<table class="profile">
  <tr>
    <td class="main">
      <h2>
        <%= gravatar_for @user %>
        <%= h @user.name %>
      </h2>
    </td>
    <td class="sidebar round">
      <strong>Name</strong> <%= h @user.name %><br />
      <strong>URL</strong>  <%= link_to user_path(@user), @user %>
    </td>
  </tr>
</table>
```

Here we've used an HTML break tag `<br />` to put a break between the user's name and URL. Also note the use of `user_path` to make a clickable link so that users can easily share their profile URLs. This is only the first of many named routes (Section 5.2.2) associated with the User resource (Listing 6.24); we'll see many more in the next few chapters. The code

```
user_path(@user)
```

returns the path to the user, in this case `/users/1`. The related code

```
user_url(@user)
```

just returns the entire URL, `http://localhost:3000/users/1`. (Compare to the routes created in Section 5.2.2.) Both are examples of the named routes created by the users resource in Listing 6.24; a list of all the named routes appears in Table 7.1.

Named route	Path
`users_path`	`/users`
`user_path(@user)`	`/users/1`
`new_user_path`	`/users/new`
`edit_user_path(@user)`	`/users/1/edit`
`users_url`	`http://localhost:3000/users`
`user_url(@user)`	`http://localhost:3000/users/1`
`new_user_url`	`http://localhost:3000/users/new`
`edit_user_url(@user)`	`http://localhost:3000/users/1/edit`

Table 7.1: Named routes provided by the users resource in Listing 6.24.

Note that in

```
<%= link_to user_path(@user), @user %>
```

`user_path(@user)` is the link *text*, while the address is just `@user`. In the context of a `link_to`, Rails converts `@user` to the appropriate URL; in other words, the code above is equivalent to the code

```
<%= link_to user_path(@user), user_path(@user) %>
```

Either way works fine, but, as in the `:id => @user` idiom from Listing 7.17, using just `@user` is a common Rails convention. In both cases, the Embedded Ruby produces the HTML

```
<a href="/users/1">/users/1</a>
```

With the HTML elements and CSS classes in place, we can style the show page with the CSS shown in Listing 7.25. The resulting page is shown in Figure 7.6.

Listing 7.25. CSS for styling the user show page, including the sidebar.
`public/stylesheets/custom.css`

```
.
.
.
/* User show page */

table.profile {
  width: 100%;
  margin-bottom: 0;
}

td.main {
  width: 70%;
  padding: 1em;
}

td.sidebar {
  width: 30%;
  padding: 1em;
  vertical-align: top;
  background: #ffc;
}

.profile img.gravatar {
  border: 1px solid #999;
  margin-bottom: -15px;
}
```

Figure 7.6: The user show page `/users/1` with a sidebar and CSS. (full size)

7.4 Conclusion

In this chapter, we've effectively finished the User model, so we're now fully prepared to sign up new users and to let them sign in securely with an email/password combination. Moreover, we have a nice first cut of the user profile page, so after signing in users will have a place to go.

7.4.1 Git commit

Before moving on, we should close the Git loop opened in the introduction to Chapter 6 by making a final commit to the `modeling-users` branch and then merging into `master`.[26] First, verify that we are on the `modeling-users` branch:

```
$ git branch
  master
* modeling-users
```

As noted in Section 1.3.5.1, the asterisk here identifies the present branch, so we are indeed ready to commit and merge:[27]

```
$ git add .
$ git commit -am "User model with passwords, and user show page"
$ git checkout master
$ git merge modeling-users
```

7.4.2 Heroku deploy

If you've deployed your sample application to Heroku, you can push it up at this point:

```
$ git push heroku
```

Then migrate the database on the remote server using the `heroku` command:

```
$ heroku rake db:migrate
```

Now if you want to create a sample user on Heroku, you can use the Heroku console:

```
$ heroku console
>> User.create(:name => "Example User", :email => "user@example.com",
?>             :password => "foobar", :password_confirmation => "foobar")
```

7.5 Exercises

1. Copy each of the variants of the `authenticate` method from Listing 7.26 through Listing 7.30 into your User model, and verify that they are correct by running your test suite.
2. The final `authenticate` example (Listing 7.30) is particularly challenging. Experiment with the console to see if you can understand how it works.
3. How could you get the Gravatar helper `gravatar_for` to work if our User model used `email_address` instead of `email` to represent email addresses? *Hint:* The best answer does *not* involve editing the Gravatar plugin.

Listing 7.26. The `authenticate` method with `User` in place of `self`.

```
  def User.authenticate(email, submitted_password)
    user = find_by_email(email)
    return nil  if user.nil?
    return user if user.has_password?(submitted_password)
  end
```

Listing 7.27. The `authenticate` method with an explicit third `return`.

```
  def self.authenticate(email, submitted_password)
    user = find_by_email(email)
    return nil  if user.nil?
    return user if user.has_password?(submitted_password)
    return nil
  end
```

Listing 7.28. The `authenticate` method using an `if` statement.

```
  def self.authenticate(email, submitted_password)
    user = find_by_email(email)
    if user.nil?
      nil
    elsif user.has_password?(submitted_password)
      user
    else
      nil
    end
  end
```

Listing 7.29. The `authenticate` method using an `if` statement and an implicit return.

```
def self.authenticate(email, submitted_password)
  user = find_by_email(email)
  if user.nil?
    nil
  elsif user.has_password?(submitted_password)
    user
  end
end
```

Listing 7.30. The `authenticate` method using the ternary operator.

```
def self.authenticate(email, submitted_password)
  user = find_by_email(email)
  user && user.has_password?(submitted_password) ? user : nil
end
```

« Chapter 6 Modeling and viewing users, part I Chapter 8 Sign up »

1. We saw ranges before in Section 4.3.1. ↑
2. For more details on the kind of callbacks supported by Active Record, see the discussion of callbacks at the Rails Guides. ↑
3. The extra level of indentation is a typographical reminder that we’re in a private section; otherwise, it’s easy to miss the `private` keyword and be confused when trying to access a private method that you think is public. I thought the extra indentation was a stupid convention until I burned an hour on just this problem a couple years back. Now I add the extra indentation. ↑
4. Ruby has a closely related keyword called `protected` that differs subtly from `private`. As far as I can tell, the only reason to learn the difference is so that you can ace a job interview that asks you “In Ruby, what is the difference between `private` and `protected`?” But do you really want to work at a company that asks you such a lame interview question? At his keynote at RubyConf in 2008, Dave Thomas (author of *Programming Ruby*) suggested eliminating `protected` from future versions of Ruby, and I agree with the sentiment. Just use `private` and you’ll be fine. ↑
5. I am ashamed to admit that this is how we implemented passwords in *RailsSpace*. Consider this section my penance. ↑
6. The alert reader may notice that none of what we do in this section *requires* encryption, but, once we develop some of the theory of secure passwords and write a basic implementation (Section 7.2.2), the only way for the `has_password?` method to work properly is for all the encryption machinery to work properly as well. ↑
7. In my setup, the `require ’digest’` line is unnecessary, but several readers have reported getting a `NameError` exception if they don’t include it explicitly. It does no harm in any case, so I’ve included the explicit `require` just to be safe. ↑
8. Technically, rainbow attacks could still succeed, but using a salted hash makes them computationally unfeasible. ↑
9. As noted in Section 7.2.2, the explicit `require ’digest’` line is unnecessary on some systems, but it does no harm to include it. ↑
10. It would be nice to be able to run all the examples in a particular `describe` block (in this case, `"has_password? method"`), but the `spec` script doesn’t have this option. You can,

however, pass `spec` the *line number* of the `describe` block using the `-l` option. Though hard-coding the line number into the tutorial would be brittle and error-prone, feel free to try using `-l` with the appropriate line number from your current version of `user_spec.rb` to see how it works. ↑

11. Recall from Box 6.2 that the *index* on the `email` column ensures that this retrieval is efficient. ↑
12. We'll plan to add a couple more attributes (one to identify administrative users and one to allow a "remember me" feature), but they are not strictly necessary. All the *essential* user attributes have now been defined. ↑
13. Some programmers take this to the next level, and attempt to test views separately from controllers. I think this goes too far, since views and controllers are necessarily coupled. In fact, if you take the default tests generated by RSpec and change the name of an instance variable in a controller but not in the view, all the tests still pass, even though the application is completely broken. In *Rails Tutorial*, we combine controller and views specs using `integrate_views` to avoid this problem. ↑
14. Many experienced Rails programmers find that this factory approach is much more flexible than *fixtures*, which Rails uses by default but can be quite brittle and difficult to maintain. ↑
15. Presumably "Factory Girl" is a reference to the movie of the same name. ↑
16. As noted briefly in Section 1.2.2, when installing gems I prefer to use the version flag `-v` to install a gem version known to work with the tutorial—in this case, version 1.2.3 of Factory Girl. Feel free omit this flag to use the latest version, but you're on your own if it breaks. ↑
17. You can also put the `config.gem` line in `config/environment.rb`, but we only need factories in the test environment, so there's no need to include it in development or production. Also, when a gem is loaded with `config.gem` in the `test.rb` file, you can have Rails *automatically* install it using `rake gems:install RAILS_ENV=test`. That's a bit complicated at this point, though, and gem dependencies are completely different in Rails 3, so I'm not bothering with this detail now. ↑
18. I used `get 'new'` in Listing 5.23 and subsequent tests for the `new` action because at that point we had yet to encounter the idea of standard REST actions. I'll switch to `get :new` in future tests. ↑
19. It does this by calling the `to_param` method on the `@user` variable. ↑
20. It's not necessarily always a good idea to make HTML tests this specific, since we don't always want to constrain the HTML layout this tightly. Feel free to experiment and find the right level of detail for your projects and tastes. ↑
21. I'm not covering the details on HTML escaping because in Rails 3 all HTML inserted by ERb will be escaped by default, so application developers won't have to worry about it any more. ↑
22. Gravatar was originally created by Tom Preston-Werner, cofounder of GitHub, and was acquired and scaled by Automattic (best known as the makers of WordPress). ↑
23. In Hinduism, an avatar is the manifestation of a deity in human or animal form. By extension, the term *avatar* is commonly used to mean some kind of personal representation, especially in a virtual environment. But you've seen the movie by now, so you already knew this. ↑
24. If your application does need to handle images or other file uploads, Paperclip is the way to go. Like Factory Girl, Paperclip is brought to you by thoughtbot. (Though I do know several people there, I have no vested interest in promoting thoughtbot; they just make good software.) ↑
25. If anyone gives you grief for using, horror of horrors, *tables for layout*, have them point their Firebug inspector at Twitter's profile sidebar and tell you what they see. In fact, you'll find that, while "semantic markup" using `div`s and `span`s is increasingly common, virtually all sites resort to `table`s for layout on occasion. In the present case, getting the vertical alignment just

right is *much* easier with tables. ↑

26. Ordinarily, I recommend making more frequent, smaller commits, but frequent Git commits throughout the tutorial would be hard to maintain and would break up the flow of the discussion. ↑
27. If you’re *not* on the right branch, run `git checkout modeling-users` before proceeding. ↑