```
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
end
```

6.3 Adding a secure password

Now that we've defined validations for the name and email fields, we're ready to add the last of the basic User attributes: a secure password. The method is to require each user to have a password (with a password confirmation), and then store a *hashed* version of the password in the database. (There is some potential for confusion here. In the present context, a *hash* refers not to the Ruby data structure from Section 4.3.3 but rather to the result of applying an irreversible hash function to input data.) We'll also add a way to *authenticate* a user based on a given password, a method we'll use in Chapter 8 to allow users to log in to the site.

The method for authenticating users will be to take a submitted password, hash it, and compare the result to the hashed value stored in the database. If the two match, then the submitted password is correct and the user is authenticated. By comparing hashed values instead of raw passwords, we will be able to authenticate users without storing the passwords themselves. This means that, even if our database is compromised, our users' passwords will still be secure.

6.3.1 A hashed password

Most of the secure password machinery will be implemented using a single Rails method called **`has_secure_password`**, which we'll include in the User model as follows:

```
class User < ApplicationRecord
  .
  .
  .
  has_secure_password
end
```

When included in a model as above, this one method adds the following functionality:

- The ability to save a securely hashed **password_digest** attribute to the database
- A pair of virtual attributes[19] (**password** and **password_confirmation**), including presence validations upon object creation and a validation requiring that they match
- An **authenticate** method that returns the user when the password is correct (and **false** otherwise)

The only requirement for **has_secure_password** to work its magic is for the corresponding model to have an attribute called **password_digest**. (The name *digest* comes from the terminology of cryptographic hash functions. In this context, *hashed password* and *password digest* are synonyms.)[20] In the case of the User model, this leads to the data model shown in Figure 6.9.

To implement the data model in Figure 6.9 , we first generate an appropriate migration for the **password_digest** column. We can choose any migration name we want, but it's convenient to end the name with **to_users**, since in this case Rails automatically constructs a migration to add columns to the **users** table. The result, with migration name **add_password_digest_to_users**, appears as follows:

```
$ rails generate migration add_password_digest_to_users password_digest:string
```

[19] In this context, *virtual* means that the attributes exist on the model object but do not correspond to columns in the database.

[20] Hashed password digests are often erroneously referred to as *encrypted passwords*. For example, the source code of **has_secure_password** makes this mistake, as did the first two editions of this tutorial. This terminology is wrong because by design encryption is *reversible*—the ability to encrypt implies the ability to *decrypt* as well. In contrast, the whole point of calculating a password's hash digest is to be *irreversible*, so that it is computationally intractable to infer the original password from the digest. (Thanks to reader Andy Philips for pointing out this issue and for encouraging me to fix the broken terminology.)

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

Figure 6.9: The User data model with an added `password_digest` attribute.

Here we've also supplied the argument `password_digest:string` 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, which included the arguments `name:string` and `email:string`.) By including `password_digest:string`, we've given Rails enough information to construct the entire migration for us, as seen in Listing 6.36.

Listing 6.36: The migration to add a `password_digest` column.
db/migrate/[timestamp]_add_password_digest_to_users.rb

```
class AddPasswordDigestToUsers < ActiveRecord::Migration[6.0]
  def change
    add_column :users, :password_digest, :string
  end
end
```

Listing 6.36 uses the `add_column` method to add a `password_digest` column to the `users` table. To apply it, we just migrate the database:

```
$ rails db:migrate
```

To make the password digest, **has_secure_password** uses a state-of-the-art hash function called bcrypt. By hashing the password with bcrypt, we ensure that an attacker won't be able to log in to the site even if they manage to obtain a copy of the database. To use bcrypt in the sample application, we need to add the `bcrypt` gem to our **Gemfile** (Listing 6.37).[21]

Listing 6.37: Adding **bcrypt** to the **Gemfile**.

```
source 'https://rubygems.org'

gem 'rails',          '6.0.1'
gem 'bcrypt',         '3.1.13'
gem 'bootstrap-sass', '3.4.1'
.
.
.
```

Then run **bundle install** as usual:

```
$ bundle install
```

6.3.2 User has secure password

Now that we've supplied the User model with the required **password_digest** attribute and installed bcrypt, we're ready to add **has_secure_password** to the User model, as shown in Listing 6.38.

Listing 6.38: Adding **has_secure_password** to the User model. RED
app/models/user.rb

```
class User < ApplicationRecord
  before_save { self.email = email.downcase }
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
```

[21] As always, you should use the version numbers listed at gemfiles-6th-ed.railstutorial.org instead of the ones listed here.

```
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
end
```

As indicated by the RED indicator in Listing 6.38, the tests are now failing, as you can confirm at the command line:

Listing 6.39: RED

```
$ rails test
```

The reason is that, as noted in Section 6.3.1, **has_secure_password** enforces validations on the virtual **password** and **password_confirmation** attributes, but the tests in Listing 6.26 create an **@user** variable without these attributes:

```
def setup
  @user = User.new(name: "Example User", email: "user@example.com")
end
```

So, to get the test suite passing again, we just need to add a password and its confirmation, as shown in Listing 6.40.

Listing 6.40: Adding a password and its confirmation. GREEN
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end
  .
  .
  .
end
```

Note that the first line inside the `setup` method includes an additional comma at the end, as required by Ruby's hash syntax (Section 4.3.3). Leaving this comma off will produce a syntax error, and you should use your technical sophistication (Box 1.2) to identify and resolve such errors if (or, more realistically, when) they occur.

At this point the tests should be GREEN:

Listing 6.41: GREEN

```
$ rails test
```

We'll see in just a moment the benefits of adding `has_secure_password` to the User model (Section 6.3.4), but first we'll add a minimal requirement on password security.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Confirm that a user with valid name and email still isn't valid overall.

2. What are the error messages for a user with no password?

6.3.3 Minimum password standards

It's good practice in general to enforce some minimum standards on passwords to make them harder to guess. There are many options for enforcing password strength in Rails, but for simplicity we'll just enforce a minimum length and the requirement that the password not be blank. Picking a length of 6 as a reasonable minimum leads to the validation test shown in Listing 6.42.

Listing 6.42: Testing for a minimum password length. RED
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end
  .
  .
  .
  test "password should be present (nonblank)" do
    @user.password = @user.password_confirmation = " " * 6
    assert_not @user.valid?
  end

  test "password should have a minimum length" do
    @user.password = @user.password_confirmation = "a" * 5
    assert_not @user.valid?
  end
end
```

Note the use of the compact multiple assignment

```
@user.password = @user.password_confirmation = "a" * 5
```

in Listing 6.42. This arranges to assign a particular value to the password and its confirmation at the same time (in this case, a string of length 5, constructed using string multiplication as in Listing 6.14).

You may be able to guess the code for enforcing a **minimum** length constraint by referring to the corresponding **maximum** validation for the user's name (Listing 6.16):

```
validates :password, length: { minimum: 6 }
```

Combining this with a **presence** validation (Section 6.2.2) to ensure nonblank passwords, this leads to the User model shown in Listing 6.43. (It turns out the

has_secure_password method includes a presence validation, but unfortunately it applies only to records with *empty* passwords, which allows users to create invalid passwords like `'      '` (six spaces).)

Listing 6.43: The complete implementation for secure passwords. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  before_save { self.email = email.downcase }
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }
end
```

At this point, the tests should be GREEN:

Listing 6.44: GREEN

```
$ rails test:models
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Confirm that a user with valid name and email but a too-short password isn't valid.

2. What are the associated error messages?

6.3.4 Creating and authenticating a user

Now that the basic User model is complete, we'll create a user in the database as preparation for making a page to show the user's information in Section 7.1. We'll also take a more concrete look at the effects of adding **has_secure_password** to the User model, including an examination of the important **authenticate** method.

Since users can't yet sign up for the sample application through the web—that's the goal of Chapter 7—we'll use the Rails console to create a new user by hand. For convenience, we'll use the **create** method discussed in Section 6.1.3, but in the present case we'll take care *not* to start in a sandbox so that the resulting user will be saved to the database. This means starting an ordinary **rails console** session and then creating a user with a valid name and email address together with a valid password and matching confirmation:

```
$ rails console
>> User.create(name: "Michael Hartl", email: "michael@example.com",
?>             password: "foobar", password_confirmation: "foobar")
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 03:15:38", updated_at: "2019-08-22 03:15:38",
password_digest: [FILTERED]>
```

To check that this worked, let's look at the resulting **users** table in the development database using DB Browser for SQLite, as shown in Figure 6.10.[22] (If you're using the cloud IDE, you should download the database file as in Figure 6.5.) Note that the columns correspond to the attributes of the data model defined in Figure 6.9.

Returning to the console, we can see the effect of **has_secure_password** from Listing 6.43 by looking at the **password_digest** attribute:

[22] If for any reason something went wrong, you can always reset the database as follows:

1. Quit the console.
2. Run **$ rm -f development.sqlite3** at the command line to remove the database. (We'll learn a more elegant method for doing this in Chapter 7.)
3. Re-run the migrations using **$ rails db:migrate**.
4. Restart the console.

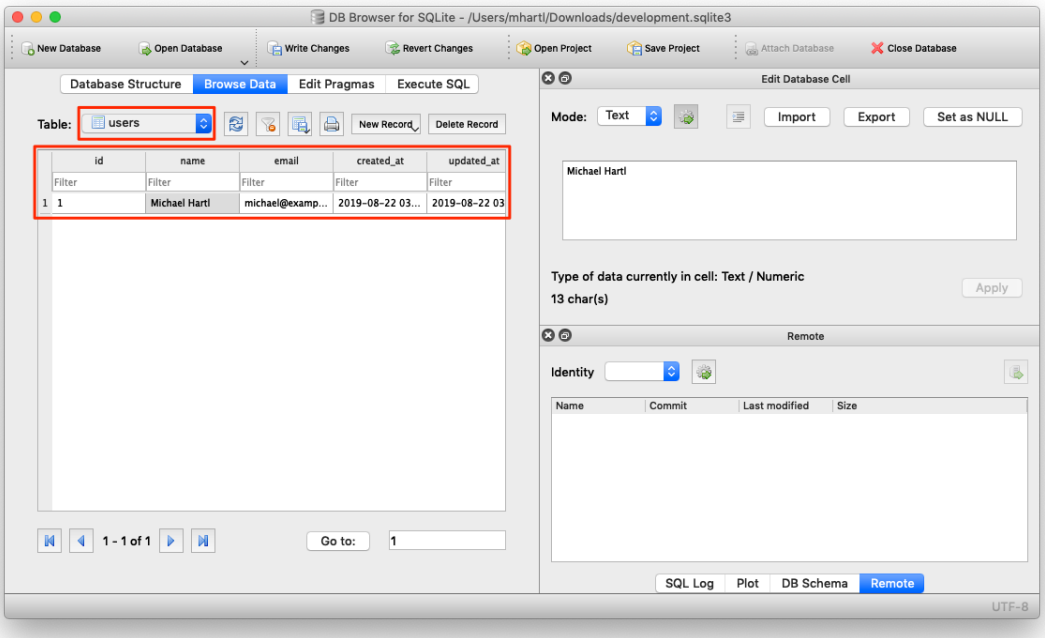

Figure 6.10: A user row in the SQLite database `db/development.sqlite3`.

```
>> user = User.find_by(email: "michael@example.com")
>> user.password_digest
=> "$2a$12$WgjER5ovLFjC2hmCItmbTe6nAXzT3bO66GiAQ83Ev03eVp32zyNYG"
```

This is the hashed version of the password (**"foobar"**) used to initialize the user object. Because it's constructed using bcrypt, it is computationally impractical to use the digest to discover the original password.[23]

As noted in Section 6.3.1, **has_secure_password** automatically adds an **authenticate** method to the corresponding model objects. This method determines if a given password is valid for a particular user by computing its digest and comparing the result to **password_digest** in the database. In the case of the user we just created, we can try a couple of invalid passwords as follows:

```
>> user.authenticate("not_the_right_password")
false
>> user.authenticate("foobaz")
false
```

Here **user.authenticate** returns **false** for invalid password. If we instead authenticate with the correct password, **authenticate** returns the user itself:

```
>> user.authenticate("foobar")
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 03:15:38", updated_at: "2019-08-22 03:15:38",
password_digest: [FILTERED]>
```

In Chapter 8, we'll use the **authenticate** method to sign registered users into our site. In fact, it will turn out not to be important to us that **authenticate** returns the user itself; all that will matter is that it returns a value that is **true** in a boolean context. Recalling from Section 4.2.2 that **!!** converts an object to its corresponding boolean value, we can see that **user.authenticate** does the job nicely:

[23] By design, the bcrypt algorithm produces a *salted hash*, which protects against two important classes of attacks (dictionary attacks and rainbow table attacks).

```
>> !!user.authenticate("foobar")
=> true
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Quit and restart the console, and then find the user created in this section.
2. Try changing the name by assigning a new name and calling **`save`**. Why didn't it work?
3. Update **`user`**'s name to use your name. *Hint*: The necessary technique is covered in Section 6.1.5.

6.4 Conclusion

Starting from scratch, in this chapter we created a working User model with name, email, and password attributes, together with validations enforcing several important constraints on their values. In addition, we have the ability to securely authenticate users using a given password. This is a remarkable amount of functionality for only twelve lines of code.

In Chapter 7, we'll make a working signup form to create new users, together with a page to display each user's information. In Chapter 8, we'll then use the authentication machinery from Section 6.3 to let users log into the site.

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

```
$ rails test
$ git add -A
$ git commit -m "Make a basic User model (including secure passwords)"
```