The **update** 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). Note that if any of the validations fail, such as when a password is required to save a record (as implemented in Section 6.3), the call to **update** will fail. If we need to update only a single attribute, using the singular **update_attribute** bypasses this restriction by skipping the validations:

```
>> user.update_attribute(:name, "El Duderino")
=> true
>> user.name
=> "El Duderino"
```

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. Update the user's name using assignment and a call to **save**.
2. Update the user's email address using a call to **update**.
3. Confirm that you can change the magic columns directly by updating the **created_at** column using assignment and a save. Use the value **1.year.ago**, which is a Rails way to create a timestamp one year before the present time.

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 log 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* (seen briefly before in Section 2.3.2). In this section, we'll cover several of the most common cases, validating *presence*, *length*, *format* and *uniqueness*. In Section 6.3.2 we'll add a final common validation, *confirmation*. And we'll see in Section 7.3 how validations give us convenient error messages when users make submissions that violate them.

6.2.1 A validity test

As noted in Box 3.3, test-driven development isn't always the right tool for the job, but model validations are exactly the kind of features for which TDD is a perfect fit. It's difficult to be confident that a given validation is doing exactly what we expect it to without writing a failing test and then getting it to pass.

Our method will be to start with a *valid* model object, set one of its attributes to something we want to be invalid, and then test that it in fact is invalid. As a safety net, we'll first write a test to make sure the initial model object is valid. This way, when the validation tests fail we'll know it's for the right reason (and not because the initial object was invalid in the first place).

In what follows, and when doing TDD generally, it's convenenient to work with your editor split into two *panes*, with test code on the left and application code on the right. My preferred setup with the cloud IDE is shown in Figure 6.7.

To get us started, the command in Listing 6.1 produced an initial test for testing users, though in this case it's practically blank (Listing 6.4).

Listing 6.4: The practically blank default User test.
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

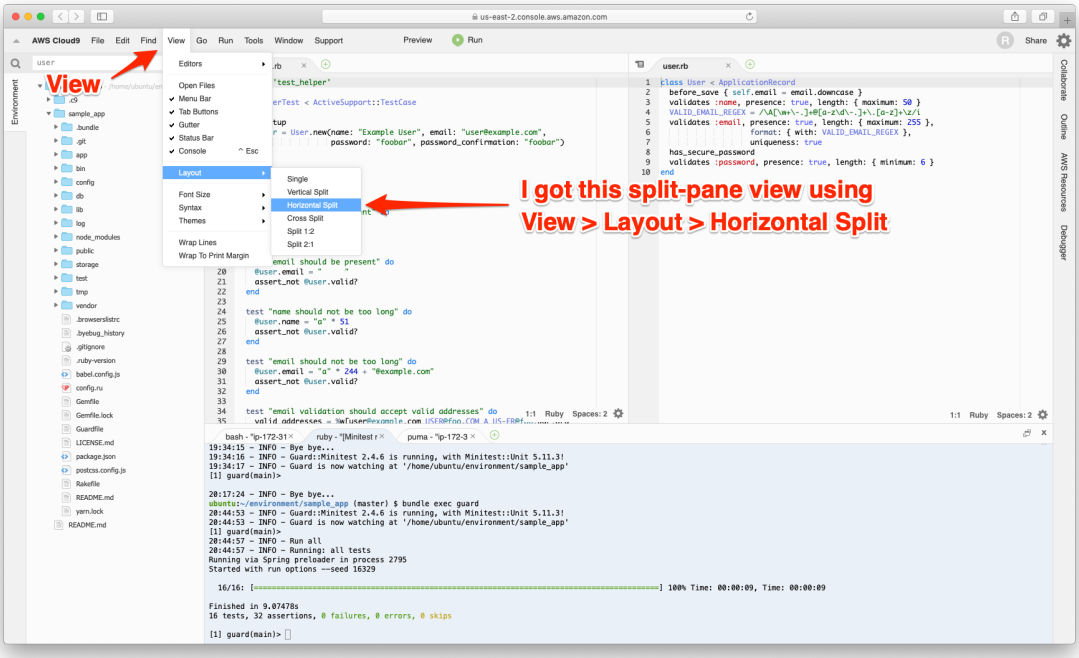

Figure 6.7: TDD with a split pane.

To write a test for a valid object, we'll create an initially valid User model object `@user` using the special `setup` method (discussed briefly in the Chapter 3 exercises), which automatically gets run before each test. Because `@user` is an instance variable, it's automatically available in all the tests, and we can test its validity using the `valid?` method (Section 6.1.3). The result appears in Listing 6.5.

Listing 6.5: A test for an initially valid user. 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")
  end

  test "should be valid" do
    assert @user.valid?
  end
end
```

Listing 6.5 uses the plain `assert` method, which in this case succeeds if `@user.valid?` returns `true` and fails if it returns `false`.

Because our User model doesn't currently have any validations, the initial test should pass:

Listing 6.6: GREEN

```
$ rails test:models
```

Here we've used `rails test:models` to run just the model tests (compare to `rails test:integration` from Section 5.3.4).

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. In the console, confirm that a new user is currently valid.
2. Confirm that the user created in Section 6.1.3 is also valid.

6.2.2 Validating presence

Perhaps the most elementary validation is *presence*, which simply verifies that a given attribute is present. For example, in this section we'll ensure that both the name and email fields are present before a user gets saved to the database. In Section 7.3.3, we'll see how to propagate this requirement up to the signup form for creating new users.

We'll start with a test for the presence of a **`name`** attribute by building on the test in Listing 6.5. As seen in Listing 6.7, all we need to do is set the **`@user`** variable's **`name`** attribute to a blank string (in this case, a string of spaces) and then check (using the **`assert_not`** method) that the resulting User object is not valid.

Listing 6.7: A test for validation of the **`name`** attribute. 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")
  end

  test "should be valid" do
    assert @user.valid?
  end

  test "name should be present" do
    @user.name = "     "
    assert_not @user.valid?
  end
end
```

At this point, the model tests should be RED:

Listing 6.8: RED

```
$ rails test:models
```

As we saw briefly before in the Chapter 2 exercises, the way to validate the presence of the name attribute is to use the `validates` method with argument `presence: true`, as shown in Listing 6.9. The `presence: true` argument 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 6.9: Validating the presence of a `name` attribute. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true
end
```

Listing 6.9 may look like magic, but `validates` is just a method. An equivalent formulation of Listing 6.9 using parentheses is as follows:

```
class User < ApplicationRecord
  validates(:name, presence: true)
end
```

Let's drop into the console to see the effects of adding a validation to our User model:[10]

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

[10] I'll omit the output of console commands when they are not particularly instructive—for example, the results of `User.new`.

Here we check the validity of the **user** variable using the **valid?** method, which returns **false** when the object fails one or more validations, and **true** when all validations pass. 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 4.4.3.)

Because the user isn't valid, an attempt to save the user to the database automatically fails:

```
>> user.save
=> false
```

As a result, the test in Listing 6.7 should now be GREEN:

Listing 6.10: GREEN

```
$ rails test:models
```

Following the model in Listing 6.7, writing a test for **email** attribute presence is easy (Listing 6.11), as is the application code to get it to pass (Listing 6.12).

Listing 6.11: A test for validation of the **email** attribute. 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")
```

```
  end

  test "should be valid" do
    assert @user.valid?
  end

  test "name should be present" do
    @user.name = ""
    assert_not @user.valid?
  end

  test "email should be present" do
    @user.email = "     "
    assert_not @user.valid?
  end
end
```

Listing 6.12: Validating the presence of an `email` attribute. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  validates :name,  presence: true
  validates :email, presence: true
end
```

At this point, the presence validations are complete, and the test suite should be GREEN:

Listing 6.13: GREEN

```
$ rails test
```

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. Make a new user called `u` and confirm that it's initially invalid. What are the full error messages?

2. Confirm that `u.errors.messages` is a hash of errors. How would you access just the email errors?

6.2.3 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 6.2.2, this step is easy.

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. In addition, although it's unlikely ever to be a problem, there's a chance that a user's email address could overrun the maximum length of strings, which for many databases is 255. Because the format validation in Section 6.2.4 won't enforce such a constraint, we'll add one in this section for completeness. Listing 6.14 shows the resulting tests.

Listing 6.14: Tests for `name` and `email` length validations. 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")
  end
  .
  .
  .
  test "name should not be too long" do
    @user.name = "a" * 51
    assert_not @user.valid?
  end

  test "email should not be too long" do
    @user.email = "a" * 244 + "@example.com"
    assert_not @user.valid?
  end
end
```

For convenience, we've used "string multiplication" in Listing 6.14 to make a string 51 characters long. We can see how this works using the console:

```
>> "a" * 51
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
>> ("a" * 51).length
=> 51
```

The email length validation arranges to make a valid email address that's one character too long:

```
>> "a" * 244 + "@example.com"
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaa@example.com"
>> ("a" * 244 + "@example.com").length
=> 256
```

At this point, the tests in Listing 6.14 should be RED:

Listing 6.15: RED

```
$ rails test
```

To get them to pass, we need to use the validation argument to constrain length, which is just **`length`**, along with the **`maximum`** parameter to enforce the upper bound (Listing 6.16).

Listing 6.16: Adding a length validation for the **`name`** attribute. GREEN
`app/models/user.rb`

```
class User < ApplicationRecord
  validates :name,  presence: true, length: { maximum: 50 }
  validates :email, presence: true, length: { maximum: 255 }
end
```

Now the tests should be GREEN:

Listing 6.17: GREEN

```
$ rails test
```

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

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. Make a new user with too-long name and email and confirm that it's not valid.
2. What are the error messages generated by the length validation?

6.2.4 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 the more stringent requirement of being a valid email address. 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 of tests involving collections of valid and invalid addresses. To make these collections, it's worth knowing about the useful **`%w[]`** technique for making arrays of strings, as seen in this console session:

```
>> %w[foo bar baz]
=> ["foo", "bar", "baz"]
>> addresses = %w[USER@foo.COM THE_US-ER@foo.bar.org first.last@foo.jp]
=> ["USER@foo.COM", "THE_US-ER@foo.bar.org", "first.last@foo.jp"]
```

```
>> addresses.each do |address|
?>   puts address
>> end
USER@foo.COM
THE_US-ER@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.

Because email format validation is tricky and error-prone, we'll start with some passing tests for *valid* email addresses to catch any errors in the validation. In other words, we want to make sure not just that invalid email addresses like *user@example,com* are rejected, but also that valid addresses like *user@example.com* are accepted, even after we impose the validation constraint. (Right now they'll be accepted because all non-blank email addresses are currently valid.) The result for a representative sample of valid email addresses appears in Listing 6.18.

Listing 6.18: Tests for valid email formats. 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")
  end
  .
  .
  .
  test "email validation should accept valid addresses" do
    valid_addresses = %w[user@example.com USER@foo.COM A_US-ER@foo.bar.org
                         first.last@foo.jp alice+bob@baz.cn]
    valid_addresses.each do |valid_address|
      @user.email = valid_address
      assert @user.valid?, "#{valid_address.inspect} should be valid"
    end
  end
end
```

Note that we've included an optional second argument to the assertion with a custom error message, which in this case identifies the address causing the test to fail:

```
assert @user.valid?, "#{valid_address.inspect} should be valid"
```

(This uses the interpolated **inspect** method mentioned in Section 4.3.3.) Including the specific address that causes any failure is especially useful in a test with an **each** loop like Listing 6.18; otherwise, any failure would merely identify the line number, which is the same for all the email addresses, and which wouldn't be sufficient to identify the source of the problem.

Next we'll add tests for the *invalidity* of a variety of invalid email addresses, such as *user@example,com* (comma in place of dot) and *user_at_foo.org* (missing the '@' sign). As in Listing 6.18, Listing 6.19 includes a custom error message to identify the exact address causing any failure.

Listing 6.19: Tests for email format validation. 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")
  end
  .
  .
  .
  test "email validation should reject invalid addresses" do
    invalid_addresses = %w[user@example,com user_at_foo.org user.name@example.
                           foo@bar_baz.com foo@bar+baz.com]
    invalid_addresses.each do |invalid_address|
      @user.email = invalid_address
      assert_not @user.valid?, "#{invalid_address.inspect} should be invalid"
    end
  end
end
```

At this point, the tests should be RED:

Listing 6.20: RED

```
$ rails test
```

The application code for email format validation uses the **`format`** validation, which works like this:

```
validates :email, format: { with: /<regular expression>/ }
```

This validates the attribute with the given *regular expression* (or *regex*), which is a powerful (and often cryptic) language for matching patterns in strings. This means we need to construct a regular expression to match valid email addresses while *not* matching invalid ones.

There actually exists a full regex for matching email addresses according to the official email standard, but it's enormous, obscure, and quite possibly counter-productive.[11] In this tutorial, we'll adopt a more pragmatic regex that has proven to be robust in practice. Here's what it looks like:

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

To help understand where this comes from, Table 6.1 breaks it into bite-sized pieces.[12]

Although you can learn a lot by studying Table 6.1, to really understand regular expressions I consider using an interactive regular expression matcher like Rubular to be essential (Figure 6.8).[13] 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

[11] For example, 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.

[12] 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.

[13] 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.

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 valid email regex.

open to Rubular—no amount of reading about regular expressions can replace playing with them interactively. (*Note*: If you use the regex from Table 6.1 in Rubular, I recommend leaving off the `\A` and `\z` characters so that you can match more than one email address at a time in the given test string. Also note that the regex consists of the characters *inside* the slashes **`/.../`**, so you should omit those when using Rubular.)

Applying the regular expression from Table 6.1 to the **`email`** format validation yields the code in Listing 6.21.

Listing 6.21: Validating the email format with a regular expression. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  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 }
end
```

Here the regex **`VALID_EMAIL_REGEX`** is a *constant*, indicated in Ruby by a name starting with a capital letter. The code

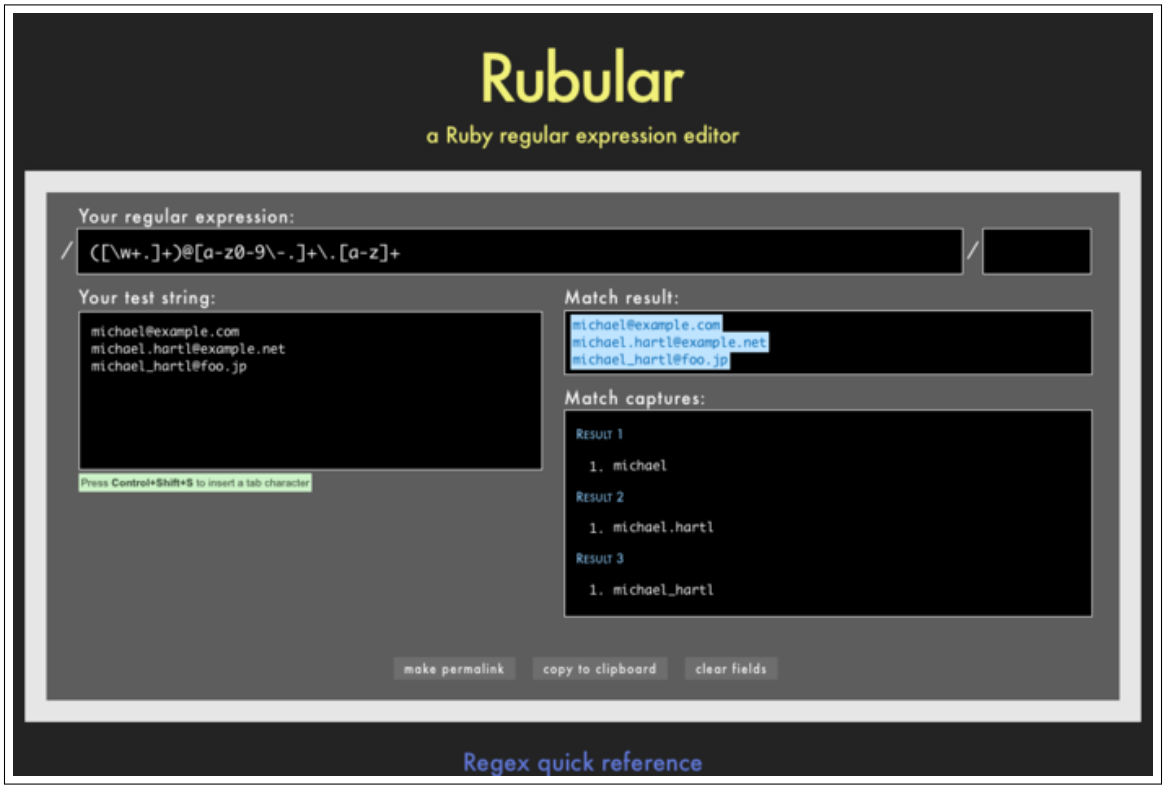

Figure 6.8: The awesome Rubular regular expression editor.

```
VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
validates :email, presence: true, length: { maximum: 255 },
                  format: { with: VALID_EMAIL_REGEX }
```

ensures that only email addresses that match the pattern will be considered valid. (The expression above has one minor weakness: it allows invalid addresses that contain consecutive dots, such as **foo@bar..com**. Updating the regex in Listing 6.21 to fix this blemish is left as an exercise (Section 6.2.4).)

At this point, the tests should be GREEN:

Listing 6.22: GREEN

```
$ rails test:models
```

This means that there's only one constraint left: enforcing email uniqueness.

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. By pasting in the valid addresses from Listing 6.18 and invalid addresses from Listing 6.19 into the test string area at Rubular, confirm that the regex from Listing 6.21 matches all of the valid addresses and none of the invalid ones.

2. As noted above, the email regex in Listing 6.21 allows invalid email addresses with consecutive dots in the domain name, i.e., addresses of the form *foo@bar..com*. Add this address to the list of invalid addresses in Listing 6.19 to get a failing test, and then use the more complicated regex shown in Listing 6.23 to get the test to pass.

3. Add *foo@bar..com* to the list of addresses at Rubular, and confirm that the regex shown in Listing 6.23 matches all the valid addresses and none of the invalid ones.

Listing 6.23: Disallowing double dots in email domain names. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-]+(\.[a-z\d\-]+)*\.[a-z]+\z/i
  validates :email, presence:   true, length: { maximum: 255 },
                    format:     { with: VALID_EMAIL_REGEX }
end
```

6.2.5 Uniqueness validation

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

We'll start with some short 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.[14] The initial duplicate email test appears in Listing 6.24.

Listing 6.24: A test for the rejection of duplicate email addresses. 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")
  end
  .
  .
  .
  test "email addresses should be unique" do
    duplicate_user = @user.dup
    @user.save
    assert_not duplicate_user.valid?
```

[14] As noted briefly in the introduction to this section, there is a dedicated test database, **`db/test.sqlite3`**, for this purpose.

```
  end
end
```

The method here is to make a user with the same email address as **`@user`** using **`@user.dup`**, which creates a duplicate user with the same attributes. Since we then save **`@user`**, the duplicate user has an email address that already exists in the database, and hence should not be valid.

We can get the new test in Listing 6.24 to pass by adding **`uniqueness: true`** to the **`email`** validation, as shown in Listing 6.25.

Listing 6.25: Validating the uniqueness of email addresses. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  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
end
```

We're not quite done, though. Email addresses are typically processed as if they were case-insensitive—i.e., **`foo@bar.com`** is treated the same as **`FOO@BAR.COM`** or **`FoO@BAr.coM`**—so our validation should incorporate this as well.[15] It's thus important to test for case-insensitivity, which we do with the code in Listing 6.26.

Listing 6.26: Testing case-insensitive email uniqueness. RED
test/models/user_test.rb

```
require 'test_helper'
```

[15]Technically, only the domain part of the email address is case-insensitive: *foo@bar.com* is actually different from *Foo@bar.com*. In practice, though, it is a bad idea to rely on this fact; as noted at about.com, "Since the case sensitivity of email addresses can create a lot of confusion, interoperability problems and widespread headaches, it would be foolish to require email addresses to be typed with the correct case. Hardly any email service or ISP does enforce case sensitive email addresses, returning messages whose recipient's email address was not typed correctly (in all upper case, for example)." Thanks to reader Riley Moses for pointing this out.

```
class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  test "email addresses should be unique" do
    duplicate_user = @user.dup
    duplicate_user.email = @user.email.upcase
    @user.save
    assert_not duplicate_user.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 initial duplicate email test, but with an uppercase email address instead. If this test feels a little abstract, go ahead and fire up the console:

```
$ rails console --sandbox
>> user = User.create(name: "Example User", email: "user@example.com")
>> user.email.upcase
=> "USER@EXAMPLE.COM"
>> duplicate_user = user.dup
>> duplicate_user.email = user.email.upcase
>> duplicate_user.valid?
=> true
```

Of course, **duplicate_user.valid?** is currently **true** because the uniqueness validation is case-sensitive, but we want it to be **false**. Fortunately, **:uniqueness** accepts an option, **:case_sensitive**, for just this purpose (Listing 6.27).

Listing 6.27: Validating the uniqueness of email addresses, ignoring case.
GREEN

app/models/user.rb

```
class User < ApplicationRecord
  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: { case_sensitive: false }
end
```

Note that we have simply replaced **`true`** in Listing 6.25 with **`case_sensitive: false`** in Listing 6.27. (Rails infers that **`uniqueness`** should be **`true`** as well.)

At this point, our application—with an important caveat—enforces email uniqueness, and our test suite should pass:

Listing 6.28: GREEN

```
$ rails test
```

There's just one small problem, which is that *the Active Record uniqueness validation does not guarantee uniqueness at the database level*. Here's a scenario that explains why:

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 can happen on any Rails website with significant traffic (which I once learned the hard way). Luckily, the solution is straightforward to implement: we just need to enforce uniqueness at the database level as well as at the model level. Our method is

to create a database *index* on the email column (Box 6.2), and then require that the index be unique.

Box 6.2. Database indices

When creating a column in a database, it is important to consider whether 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 log in to the sample app starting in Chapter 7, 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—which means we might have to examine *every* row (since the user could be the last one in the database). 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"—the paper version of a full-table scan. With a book index, on the other hand, you can just look up "foobar" in the index to see all the pages containing "foobar". A database index works essentially the same way.

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:

```
$ rails generate migration add_index_to_users_email
```

Unlike the migration for users, the email uniqueness migration is not predefined, so we need to fill in its contents with Listing 6.29.[16]

Listing 6.29: The migration for enforcing email uniqueness.
db/migrate/[timestamp]_add_index_to_users_email.rb

```
class AddIndexToUsersEmail < ActiveRecord::Migration[6.0]
  def change
    add_index :users, :email, unique: true
  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:

```
$ rails db:migrate
```

(If the migration fails, make sure to exit any running sandbox console sessions, which can lock the database and prevent migrations.)

At this point, the test suite should be RED due to a violation of the uniqueness constraint in the *fixtures*, which contain sample data for the test database. User fixtures were generated automatically in Listing 6.1, and as shown in Listing 6.30 the email addresses are not unique. (They're not *valid* either, but fixture data doesn't get run through the validations.)

Listing 6.30: The default user fixtures. RED
test/fixtures/users.yml

```
# Read about fixtures at https://api.rubyonrails.org/classes/ActiveRecord/
# FixtureSet.html
```

[16] 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.

```
one:
  name: MyString
  email: MyString

two:
  name: MyString
  email: MyString
```

Because we won't need fixtures until Chapter 8, for now we'll just remove them, leaving an empty fixtures file (Listing 6.31).

Listing 6.31: An empty fixtures file. GREEN
test/fixtures/users.yml

```
# empty
```

Having addressed the uniqueness caveat, there's one more change we need to make to be assured of email uniqueness. Some database adapters use case-sensitive indices, considering the strings "Foo@ExAMPle.CoM" and "foo@example.com" to be distinct, but our application treats those addresses as the same. To avoid this incompatibility, we'll standardize on all lower-case addresses, converting "Foo@ExAMPle.CoM" to "foo@example.com" before saving it to the database. The way to do this is with a *callback*, which is a method that gets invoked at a particular point in the lifecycle of an Active Record object.

In the present case, that point is before the object is saved, so we'll use a **before_save** callback to downcase the email attribute before saving the user.[17] The result appears in Listing 6.32. (This is just a first implementation; we'll discuss this subject again in Section 11.1, where we'll use the preferred *method reference* convention for defining callbacks.)

Listing 6.32: Ensuring email uniqueness by downcasing the email attribute. RED
app/models/user.rb

[17] See the Rails API entry on callbacks for more information on which callbacks Rails supports.

```
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
end
```

The code in Listing 6.32 passes a block to the **before_save** callback and sets the user's email address to a lower-case version of its current value using the **downcase** string method. Note also that Listing 6.32 reverts the **uniqueness** constraint back to **true**, since case-sensitive matching works fine if all of the emails are lower-case. Indeed, this practice prevents problems applying the database index from Listing 6.29, since many databases have difficulty using an index when combined with a case-insensitive match.[18]

Restoring the original constraint does break the test in Listing 6.26, but that's easy to fix by reverting the test to its previous form from Listing 6.24, as shown again in Listing 6.33.

Listing 6.33: Restoring the original email uniqueness test. 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")
  end
  .
  .
  .
  test "email addresses should be unique" do
    duplicate_user = @user.dup
    @user.save
    assert_not duplicate_user.valid?
  end
end
```

[18] Thanks to reader Alex Friedman for pointing this out.

By the way, in Listing 6.32 we could have written the assignment as

```
self.email = self.email.downcase
```

(where **self** refers to the current user), but inside the User model the **self** keyword is optional on the right-hand side:

```
self.email = email.downcase
```

We encountered this idea briefly in the context of **reverse** in the **palindrome** method (Section 4.4.2), which also noted that **self** is *not* optional in an assignment, so

```
email = email.downcase
```

wouldn't work. (We'll discuss this subject in more depth in Section 9.1.)

At this point, the Alice scenario above will work fine: the database will save a user record based on the first request, and it will reject the second save because the duplicate email address violates the uniqueness constraint. (An error will appear in the Rails log, but that doesn't do any harm.) Moreover, adding this index on the email attribute accomplishes a second goal, alluded to briefly in Section 6.1.4: as noted in Box 6.2, the index on the **email** attribute fixes a potential efficiency problem by preventing a full-table scan when finding users by email address.

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. Add a test for the email downcasing from Listing 6.32, as shown in Listing 6.34. This test uses the **reload** method for reloading a value from

the database and the **assert_equal** method for testing equality. To verify that Listing 6.34 tests the right thing, comment out the **before_save** line to get to RED, then uncomment it to get to GREEN.

2. By running the test suite, verify that the **before_save** callback can be written using the "bang" method **email.downcase!** to modify the **email** attribute directly, as shown in Listing 6.35.

Listing 6.34: A test for the email downcasing from Listing 6.32.
test/models/user_test.rb

```
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  test "email addresses should be unique" do
    duplicate_user = @user.dup
    @user.save
    assert_not duplicate_user.valid?
  end

  test "email addresses should be saved as lower-case" do
    mixed_case_email = "Foo@ExAMPle.CoM"
    @user.email = mixed_case_email
    @user.save
    assert_equal mixed_case_email.downcase, @user.reload.email
  end
end
```

Listing 6.35: An alternate callback implementation. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  before_save { 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
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
```