Chapter 13

User microposts

In the course of developing the core sample application, we've now encountered four resources—users, sessions, account activations, and password resets—but only the first of these is backed by an Active Record model with a table in the database. The time has finally come to add a second such resource: user *microposts*, which are short messages associated with a particular user.[1] We first saw microposts in larval form in Chapter 2, and in this chapter we will make a full-strength version of the sketch from Section 2.3 by constructing the Micropost data model, associating it with the User model using the `has_many` and `belongs_to` methods, and then making the forms and partials needed to manipulate and display the results (including, in Section 13.4, uploaded images). In Chapter 14, we'll complete our tiny Twitter clone by adding the notion of *following* users in order to receive a *feed* of their microposts.

13.1 A Micropost model

We begin the Microposts resource by creating a Micropost model, which captures the essential characteristics of microposts. What follows builds on the work from Section 2.3; as with the model in that section, our new Micropost model will include data validations and an association with the User model. Un-

[1] The name is motivated by the common description of Twitter as a *microblog*; since blogs have posts, microblogs should have *microposts*, which can be thought of as the generic equivalent of "tweets".

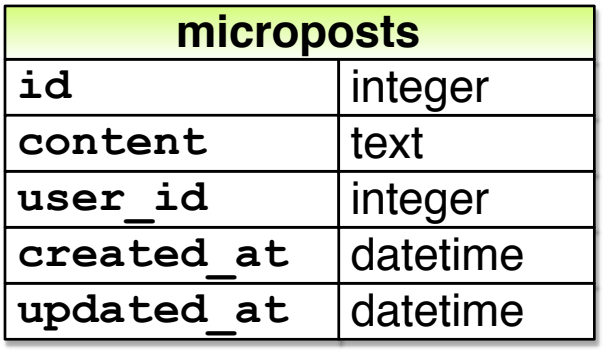

Figure 13.1: The Micropost data model.

like that model, the present Micropost model will be fully tested, and will also have a default *ordering* and automatic *destruction* if its parent user is destroyed.

If you're using Git for version control, I suggest making a topic branch at this time:

```
$ git checkout -b user-microposts
```

13.1.1 The basic model

The Micropost model needs only two attributes: a **content** attribute to hold the micropost's content and a **user_id** to associate a micropost with a particular user. The result is a Micropost model with the structure shown in Figure 13.1.

It's worth noting that the model in Figure 13.1 uses the **text** data type for micropost content (instead of **string**), which is capable of storing an arbitrary amount of text. Even though the content will be restricted to fewer than 140 characters (Section 13.1.2) and hence would fit inside the 255-character **string** type, using **text** better expresses the nature of microposts, which are more naturally thought of as blocks of text. Indeed, in Section 13.3.2 we'll use a text *area* instead of a text field for submitting microposts. In addition, using **text** gives us greater flexibility should we wish to increase the length limit at

a future date (as part of internationalization, for example). Finally, using the **text** type results in no performance difference in production,[2] so it costs us nothing to use it here.

As with the case of the User model (Listing 6.1), we generate the Micropost model using **generate model** (Listing 13.1).

Listing 13.1: Generating the Micropost model.

```
$ rails generate model Micropost content:text user:references
```

This migration leads to the creation of the Micropost model shown in Listing 13.2. In addition to inheriting from **ApplicationRecord** as usual (Section 6.1.2), the generated model includes a line indicating that a micropost **belongs_to** a user, which is included as a result of the **user:references** argument in Listing 13.1. We'll explore the implications of this line in Section 13.1.3.

Listing 13.2: The generated Micropost model.
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  belongs_to :user
end
```

The **generate** command in Listing 13.1 also produces a migration to create a **microposts** table in the database (Listing 13.3); compare it to the analogous migration for the **users** table from Listing 6.2. The biggest difference is the use of **references**, which automatically adds a **user_id** column (along with an index and a foreign key reference)[3] for use in the user/micropost association. As with the User model, the Micropost model migration automatically includes

[2] www.postgresql.org/docs/9.1/static/datatype-character.html

[3] The foreign key reference is a database-level constraint indicating that the user id in the microposts table refers to the id column in the users table. This detail will never be important in this tutorial, and the foreign key constraint isn't even supported by all databases. (It's supported by PostgreSQL, which we use in production, but not by the development SQLite database adapter.) We'll learn more about foreign keys in Section 14.1.2.

the **t.timestamps** line, which (as mentioned in Section 6.1.1) adds the magic **created_at** and **updated_at** columns shown in Figure 13.1. (We'll put the **created_at** column to work starting in Section 13.1.4.)

Listing 13.3: The Micropost migration with added index.
db/migrate/[timestamp]_create_microposts.rb

```
class CreateMicroposts < ActiveRecord::Migration[6.0]
  def change
    create_table :microposts do |t|
      t.text :content
      t.references :user, foreign_key: true

      t.timestamps
    end
    add_index :microposts, [:user_id, :created_at]
  end
end
```

Because we expect to retrieve all the microposts associated with a given user id in reverse order of creation, Listing 13.3 adds an index (Box 6.2) on the **user_id** and **created_at** columns:

```
add_index :microposts, [:user_id, :created_at]
```

By including both the **user_id** and **created_at** columns as an array, we arrange for Rails to create a *multiple key index*, which means that Active Record uses *both* keys at the same time.

With the migration in Listing 13.3, we can update the database as usual:

```
$ rails db:migrate
```

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. Using **`Micropost.new`** in the console, instantiate a new Micropost object called **`micropost`** with content "Lorem ipsum" and user id equal to the id of the first user in the database. What are the values of the magic columns **`created_at`** and **`updated_at`**?

2. What is **`micropost.user`** for the micropost in the previous exercise? What about **`micropost.user.name`**?

3. Save the micropost to the database. What are the values of the magic columns now?

13.1.2 Micropost validations

Now that we've created the basic model, we'll add some validations to enforce the desired design constraints. One of the necessary aspects of the Micropost model is the presence of a user id to indicate which user made the micropost. The idiomatically correct way to do this is to use Active Record *associations*, which we'll implement in Section 13.1.3, but for now we'll work with the **`Micropost`** model directly.

The initial micropost tests parallel those for the User model (Listing 6.7). In the **`setup`** step, we create a new micropost while associating it with a valid user from the fixtures, and then check that the result is valid. Because every micropost should have a user id, we'll add a test for a **`user_id`** presence validation. Putting these elements together yields the test in Listing 13.4.

Listing 13.4: Tests for the validity of a new micropost. GREEN
test/models/micropost_test.rb

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase

  def setup
    @user = users(:michael)
    # This code is not idiomatically correct.
    @micropost = Micropost.new(content: "Lorem ipsum", user_id: @user.id)
  end
```

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

  test "user id should be present" do
    @micropost.user_id = nil
    assert_not @micropost.valid?
  end
end
```

As indicated by the comment in the **setup** method, the code to create the micropost is not idiomatically correct, which we'll fix in Section 13.1.3.

As with the original User model test (Listing 6.5), the first test in Listing 13.4 is just a reality check, but the second is a test of the presence of the user id, for which we'll add the presence validation shown in Listing 13.5.

Listing 13.5: A validation for the micropost's **user_id**. GREEN

app/models/micropost.rb

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  validates :user_id, presence: true
end
```

By the way, as of Rails 5 the tests in Listing 13.4 actually pass without the validation in Listing 13.5, but only when using the idiomatically incorrect line highlighted in Listing 13.4. The user id presence validation is necessary after switching to the idiomatically correct code in Listing 13.12, so we include it here for convenience.

With the code in Listing 13.5 the tests should (still) be GREEN:

Listing 13.6: GREEN

```
$ rails test:models
```

Next, we'll add validations for the micropost's **content** attribute (following the example from Section 2.3.2). As with the **user_id**, the **content** attribute must be present, and it is further constrained to be no longer than 140 characters (which is what puts the *micro* in micropost).

As with the User model validations (Section 6.2), we'll add the micropost content validations using test-driven development. The resulting tests generally follow the examples from the User model validation tests, as shown in Listing 13.7.

Listing 13.7: Tests for the Micropost model validations. RED
test/models/micropost_test.rb

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase

  def setup
    @user = users(:michael)
    @micropost = Micropost.new(content: "Lorem ipsum", user_id: @user.id)
  end

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

  test "user id should be present" do
    @micropost.user_id = nil
    assert_not @micropost.valid?
  end

  test "content should be present" do
    @micropost.content = "   "
    assert_not @micropost.valid?
  end

  test "content should be at most 140 characters" do
    @micropost.content = "a" * 141
    assert_not @micropost.valid?
  end
end
```

As in Section 6.2, the code in Listing 13.7 uses string multiplication to test the micropost length validation:

```
$ rails console
>> "a" * 10
=> "aaaaaaaaaa"
>> "a" * 141
=> "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
```

The corresponding application code is virtually identical to the **name** validation for users (Listing 6.16), as shown in Listing 13.8.

Listing 13.8: The Micropost model validations. GREEN
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  belongs_to :user
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
end
```

At this point, the full test suite should be GREEN:

Listing 13.9: 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. At the console, instantiate a micropost with no user id and blank content. Is it valid? What are the full error messages?

2. At the console, instantiate a second micropost with no user id and content that's too long. Is it valid? What are the full error messages?

13.1.3 User/Micropost associations

When constructing data models for web applications, it is essential to be able to make *associations* between individual models. In the present case, each micropost is associated with one user, and each user is associated with (potentially) many microposts—a relationship seen briefly in Section 2.3.3 and shown

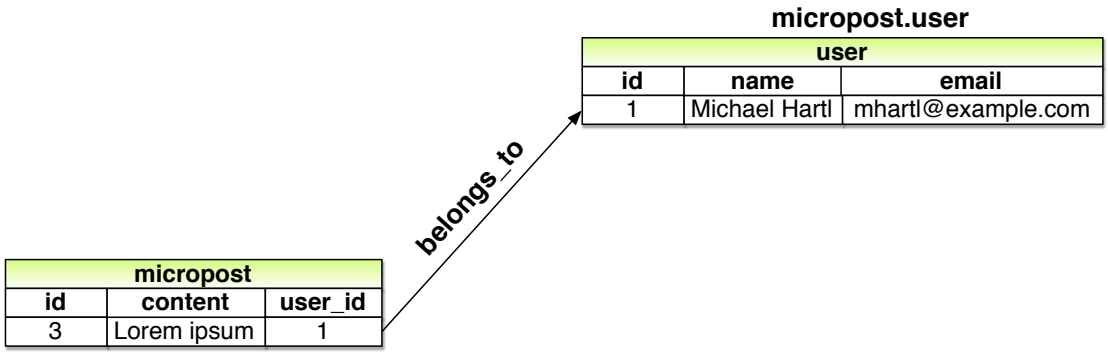

Figure 13.2: The **belongs_to** relationship between a micropost and its associated user.

schematically in Figure 13.2 and Figure 13.3. As part of implementing these associations, we'll write tests for the Micropost model and add a couple of tests to the User model.

Using the **belongs_to**/**has_many** association defined in this section, Rails constructs the methods shown in Table 13.1. Note from Table 13.1 that instead of

```
Micropost.create
Micropost.create!
Micropost.new
```

we have

```
user.microposts.create
user.microposts.create!
user.microposts.build
```

These latter methods constitute the idiomatically correct way to make a micropost, namely, *through* its association with a user. When a new micropost is made in this way, its **user_id** is automatically set to the right value. In particular, we can replace the code

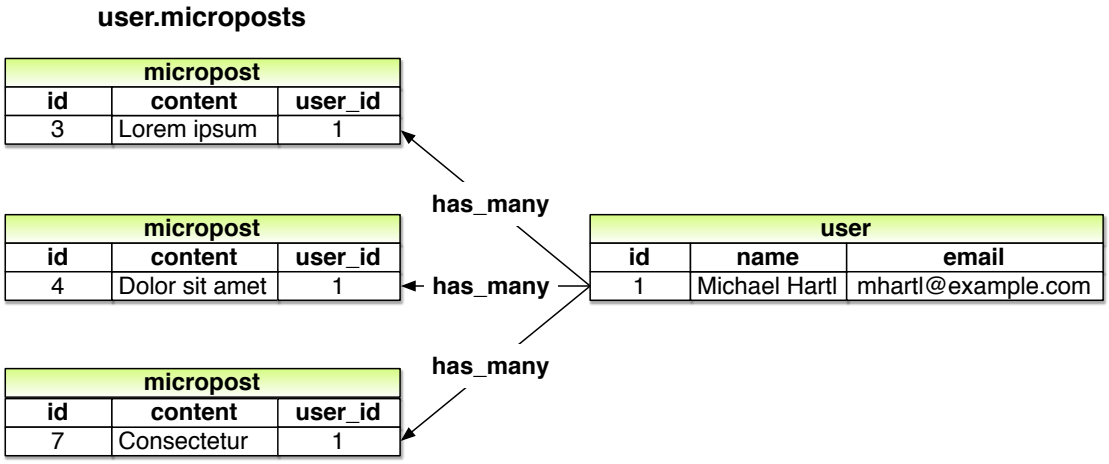

Figure 13.3: The **has_many** relationship between a user and its microposts.

```
@user = users(:michael)
# This code is not idiomatically correct.
@micropost = Micropost.new(content: "Lorem ipsum", user_id: @user.id)
```

from Listing 13.4 with this:

```
@user = users(:michael)
@micropost = @user.microposts.build(content: "Lorem ipsum")
```

(As with **new**, **build** returns an object in memory but doesn't modify the database.) Once we define the proper associations, the resulting **@micropost** variable will automatically have a **user_id** attribute equal to its associated user's id.

To get code like **@user.microposts.build** to work, we need to update the User and Micropost models with code to associate them. The first of these was included automatically by the migration in Listing 13.3 via **belongs_to :user**, as shown in Listing 13.10. The second half of the association, **has_many :microposts**, needs to be added by hand, as shown in (Listing 13.11).

Method	Purpose
`micropost.user`	Returns the User object associated with the micropost
`user.microposts`	Returns a collection of the user's microposts
`user.microposts.create(arg)`	Creates a micropost associated with `user`
`user.microposts.create!(arg)`	Creates a micropost associated with `user` (exception on failure)
`user.microposts.build(arg)`	Returns a new Micropost object associated with `user`
`user.microposts.find_by(id: 1)`	Finds the micropost with id `1` and `user_id` equal to `user.id`

Table 13.1: A summary of user/micropost association methods.

Listing 13.10: A micropost `belongs_to` a user. GREEN
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  belongs_to :user
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
end
```

Listing 13.11: A user `has_many` microposts. GREEN
app/models/user.rb

```
class User < ApplicationRecord
  has_many :microposts
  .
  .
  .
end
```

With the association thus made, we can update the `setup` method in Listing 13.4 with the idiomatically correct way to build a new micropost, as shown in Listing 13.12.

Listing 13.12: Using idiomatically correct code to build a micropost. GREEN
test/models/micropost_test.rb

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase
```

```
  def setup
    @user = users(:michael)
    @micropost = @user.microposts.build(content: "Lorem ipsum")
  end

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

  test "user id should be present" do
    @micropost.user_id = nil
    assert_not @micropost.valid?
  end
  .
  .
  .
end
```

Of course, after this minor refactoring the test suite should still be GREEN:

Listing 13.13: 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. Set `user` to the first user in the database. What happens when you execute the command `micropost = user.microposts.create(content: "Lorem ipsum")`?
2. The previous exercise should have created a micropost in the database. Confirm this by running `user.microposts.find(micropost.id)`. What if you write `micropost` instead of `micropost.id`?
3. What is the value of `user == micropost.user`? How about `user.microposts.first == micropost`?

13.1.4 Micropost refinements

In this section, we'll add a couple of refinements to the user/micropost association. In particular, we'll arrange for a user's microposts to be retrieved in a specific *order*, and we'll also make microposts *dependent* on users so that they will be automatically destroyed if their associated user is destroyed.

Default scope

By default, the **`user.microposts`** method makes no guarantees about the order of the posts, but (following the convention of blogs and Twitter) we want the microposts to come out in reverse order of when they were created so that the most recent post is first.[4] We'll arrange for this to happen using a *default scope*.

This is exactly the sort of feature that could easily lead to a spurious passing test (i.e., a test that would pass even if the application code were wrong), so we'll proceed using test-driven development to be sure we're testing the right thing. In particular, let's write a test to verify that the first micropost in the database is the same as a fixture micropost we'll call **`most_recent`**, as shown in Listing 13.14.

Listing 13.14: Testing the micropost order. RED
test/models/micropost_test.rb

```
require 'test_helper'

class MicropostTest < ActiveSupport::TestCase
  .
  .
  .
  test "order should be most recent first" do
    assert_equal microposts(:most_recent), Micropost.first
  end
end
```

Listing 13.14 relies on having some micropost fixtures, which we can define in analogy with the user fixtures, last seen in Listing 11.5. In addition to the

[4] We briefly encountered a similar issue in Section 10.5 in the context of the users index.

content attribute defined in Section 13.1.1, we also need define the associated **user**. Conveniently, Rails includes a way to build associations in fixtures, like this:

```
orange:
  content: "I just ate an orange!"
  created_at: <%= 10.minutes.ago %>
  user: michael
```

By identifying the **user** as **michael**, we tell Rails to associate this micropost with the corresponding user in the users fixture:

```
michael:
  name: Michael Example
  email: michael@example.com
  .
  .
  .
```

The full micropost fixtures appear in Listing 13.15.

Listing 13.15: Micropost fixtures.
test/fixtures/microposts.yml

```
orange:
  content: "I just ate an orange!"
  created_at: <%= 10.minutes.ago %>
  user: michael

tau_manifesto:
  content: "Check out the @tauday site by @mhartl: https://tauday.com"
  created_at: <%= 3.years.ago %>
  user: michael

cat_video:
  content: "Sad cats are sad: https://youtu.be/PKffm2uI4dk"
  created_at: <%= 2.hours.ago %>
  user: michael

most_recent:
  content: "Writing a short test"
  created_at: <%= Time.zone.now %>
  user: michael
```

Note that Listing 13.15 explicitly sets the **created_at** column using embedded Ruby. Because it's a "magic" column automatically updated by Rails, setting **created_at** by hand isn't ordinarily possible, but it is possible in fixtures.[5]

With the code in Listing 13.14 and Listing 13.15, the test suite should be RED:

Listing 13.16: RED

```
$ rails test test/models/micropost_test.rb
```

We'll get the test to pass using a Rails method called **default_scope**, which among other things can be used to set the default order in which elements are retrieved from the database. To enforce a particular order, we'll include the **order** argument in **default_scope**, which lets us order by the **created_at** column as follows:

```
order(:created_at)
```

Unfortunately, this orders the results in *ascending* order from smallest to biggest, which means that the oldest microposts come out first. To pull them out in reverse order, we can push down one level deeper and include a string with some raw SQL:

```
order('created_at DESC')
```

Here **DESC** is SQL for "descending", i.e., in descending order from newest to oldest.[6] In older versions of Rails, using this raw SQL used to be the only option to get the desired behavior, but as of Rails 4.0 we can use a more natural pure-Ruby syntax as well:

[5] In practice this might not be necessary, and in fact on many systems the fixtures are created in the order in which they appear in the file. In this case, the final fixture in the file is created last (and hence is most recent), but it would be foolish to rely on this behavior, which is brittle and probably system-dependent.

[6] SQL is case-insensitive, but it is conventional to write SQL keywords (such as **DESC**) in all-caps.

```
order(created_at: :desc)
```

Adding this in a default scope for the Micropost model gives Listing 13.17.

Listing 13.17: Ordering the microposts with **`default_scope`**. GREEN
app/models/micropost.rb

```
class Micropost < ApplicationRecord
  belongs_to :user
  default_scope -> { order(created_at: :desc) }
  validates :user_id, presence: true
  validates :content, presence: true, length: { maximum: 140 }
end
```

Listing 13.17 introduces the "stabby lambda" syntax for an object called a *Proc* (procedure) or *lambda*, which is an *anonymous function* (a function created without a name). The stabby lambda **`->`** takes in a block (Section 4.3.2) and returns a Proc, which can then be evaluated with the **`call`** method. We can see how it works at the console:

```
>> -> { puts "foo" }
=> #<Proc:0x007fab938d0108@(irb):1 (lambda)>
>> -> { puts "foo" }.call
foo
=> nil
```

(This is a somewhat advanced Ruby topic, so don't worry if it doesn't make sense right away.)

With the code in Listing 13.17, the tests should be GREEN:

Listing 13.18: GREEN

```
$ rails test
```

Dependent: destroy

Apart from proper ordering, there is a second refinement we'd like to add to microposts. Recall from Section 10.4 that site administrators have the power to *destroy* users. It stands to reason that, if a user is destroyed, the user's microposts should be destroyed as well.

We can arrange for this behavior by passing an option to the **has_many** association method, as shown in Listing 13.19.

Listing 13.19: Ensuring that a user's microposts are destroyed along with the user.
app/models/user.rb

```
class User < ApplicationRecord
  has_many :microposts, dependent: :destroy
  .
  .
  .
end
```

Here the option **dependent: :destroy** arranges for the dependent microposts to be destroyed when the user itself is destroyed. This prevents userless microposts from being stranded in the database when admins choose to remove users from the system.

We can verify that Listing 13.19 is working with a test for the User model. All we need to do is save the user (so it gets an id) and create an associated micropost. Then we check that destroying the user reduces the micropost count by 1. The result appears in Listing 13.20. (Compare to the integration test for "delete" links in Listing 10.62.)

Listing 13.20: A test of **dependent: :destroy**. 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
  .
  .
  .
  test "associated microposts should be destroyed" do
    @user.save
    @user.microposts.create!(content: "Lorem ipsum")
    assert_difference 'Micropost.count', -1 do
      @user.destroy
    end
  end
end
```

If the code in Listing 13.19 is working correctly, the test suite should still be GREEN:

Listing 13.21: 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. How does the value of **`Micropost.first.created_at`** compare to **`Micropost.last.created_at`**?

2. What are the SQL queries for Micropost.first and Micropost.last? *Hint*: They are printed out by the console.

3. Let **`user`** be the first user in the database. What is the id of its first micropost? Destroy the first user in the database using the **`destroy`** method, then confirm using **`Micropost.find`** that the user's first micropost was also destroyed.