[2.3.2 Putting the](http://ruby.railstutorial.org/chapters/a-demo-app" \l "sec-putting_the_micro_in_microposts)*[micro](http://ruby.railstutorial.org/chapters/a-demo-app" \l "sec-putting_the_micro_in_microposts)*[in microposts](http://ruby.railstutorial.org/chapters/a-demo-app" \l "sec-putting_the_micro_in_microposts)

Any *micro*post worthy of the name should have some means of enforcing the length of the post. Implementing this constraint in Rails is easy with *validations*; to accept microposts with at most 140 characters (à la Twitter), we use a *length* validation. At this point, you should open the file**app/models/micropost.rb** in your text editor or IDE and fill it with the contents of[Listing 2.9](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_length_validation). (The use of **validates** in [Listing 2.9](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_length_validation) is characteristic of Rails 3; if you’ve previously worked with Rails 2.3, you should compare this to the use of **validates\_length\_of**.)

**Listing 2.9.** Constraining microposts to be at most 140 characters.   
**app/models/micropost.rb**

**class** **Micropost** < ActiveRecord::Base

validates :content, length: { maximum: 140 }

**end**

The code in [Listing 2.9](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_length_validation) may look rather mysterious—we’ll cover validations more thoroughly starting in [Section 6.2](http://ruby.railstutorial.org/chapters/modeling-users#sec-user_validations)—but its effects are readily apparent if we go to the new micropost page and enter more than 140 characters for the content of the post. As seen in [Figure 2.14](http://ruby.railstutorial.org/chapters/a-demo-app#fig-micropost_length_error_rails_3), Rails renders*error messages* indicating that the micropost’s content is too long. (We’ll learn more about error messages in [Section 7.3.3](http://ruby.railstutorial.org/chapters/sign-up#sec-signup_error_messages).)

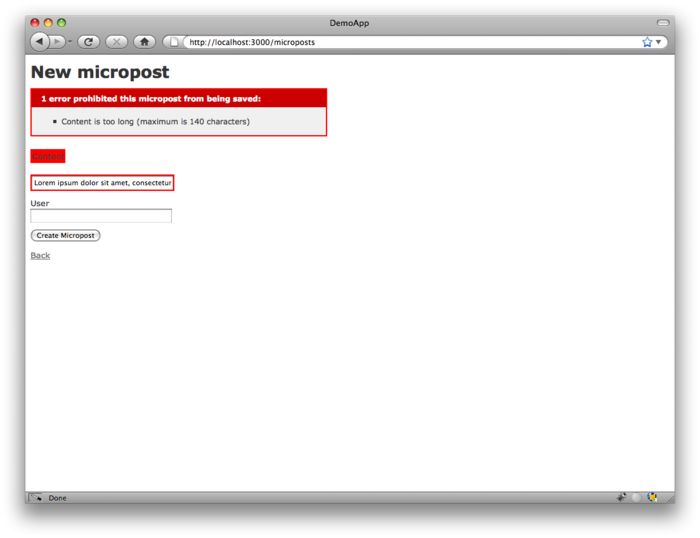


Figure 2.14: Error messages for a failed micropost creation. [(full size)](http://railstutorial.org/images/figures/micropost_length_error_rails_3-full.png)

[2.3.3 A user has\_many microposts](http://ruby.railstutorial.org/chapters/a-demo-app#sec-demo_user_has_many_microposts)

One of the most powerful features of Rails is the ability to form *associations* between different data models. In the case of our User model, each user potentially has many microposts. We can express this in code by updating the User and Micropost models as in [Listing 2.10](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_user_has_many_microposts) and [Listing 2.11](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_micropost_belongs_to_user).

**Listing 2.10.** A user has many microposts.   
**app/models/user.rb**

**class** **User** < ActiveRecord::Base

has\_many :microposts

**end**

**Listing 2.11.** A micropost belongs to a user.   
**app/models/micropost.rb**

**class** **Micropost** < ActiveRecord::Base

belongs\_to :user

validates :content, length: { maximum: 140 }

**end**

We can visualize the result of this association in [Figure 2.15](http://ruby.railstutorial.org/chapters/a-demo-app#fig-micropost_user_association). Because of the **user\_id** column in the**microposts** table, Rails (using Active Record) can infer the microposts associated with each user.

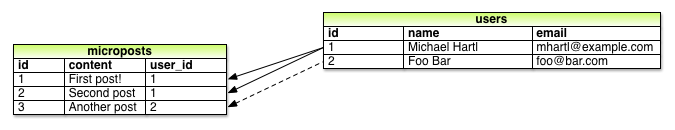


Figure 2.15: The association between microposts and users.

In [Chapter 10](http://ruby.railstutorial.org/chapters/user-microposts#top) and [Chapter 11](http://ruby.railstutorial.org/chapters/following-users#top), we will use the association of users and microposts both to display all a user’s microposts and to construct a Twitter-like micropost feed. For now, we can examine the implications of the user-micropost association by using the *console*, which is a useful tool for interacting with Rails applications. We first invoke the console with **rails console** at the command line, and then retrieve the first user from the database using **User.first** (putting the results in the variable **first\_user**):[7](http://ruby.railstutorial.org/chapters/a-demo-app#fn-2_7)

$ rails console

**>>** first\_user = User.first

=> #<User id: 1, name: "Michael Hartl", email: "michael@example.org",

created\_at: "2013-03-06 02:01:31", updated\_at: "2013-03-06 02:01:31">

**>>** first\_user.microposts

=> [#<Micropost id: 1, content: "First micropost!", user\_id: 1, created\_at:

"2013-03-06 02:37:37", updated\_at: "2013-03-06 02:37:37">, #<Micropost id: 2,

content: "Second micropost", user\_id: 1, created\_at: "2013-03-06 02:38:54",

updated\_at: "2013-03-06 02:38:54">]

**>>** exit

(I include the last line just to demonstrate how to exit the console, and on most systems you can Ctrl-d for the same purpose.) Here we have accessed the user’s microposts using the code**first\_user.microposts**: with this code, Active Record automatically returns all the microposts with **user\_id** equal to the id of **first\_user** (in this case, **1**). We’ll learn much more about the association facilities in Active Record in [Chapter 10](http://ruby.railstutorial.org/chapters/user-microposts#top) and [Chapter 11](http://ruby.railstutorial.org/chapters/following-users#top).

[2.3.4 Inheritance hierarchies](http://ruby.railstutorial.org/chapters/a-demo-app#sec-inheritance_hierarchies)

We end our discussion of the demo application with a brief description of the controller and model class hierarchies in Rails. This discussion will only make much sense if you have some experience with object-oriented programming (OOP); if you haven’t studied OOP, feel free to skip this section. In particular, if you are unfamiliar with *classes* (discussed in [Section 4.4](http://ruby.railstutorial.org/chapters/rails-flavored-ruby#sec-ruby_classes)), I suggest looping back to this section at a later time.

We start with the inheritance structure for models. Comparing [Listing 2.12](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_user_class) and [Listing 2.13](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_micropost_class), we see that both the User model and the Micropost model inherit (via the left angle bracket **<**) from**ActiveRecord::Base**, which is the base class for models provided by ActiveRecord; a diagram summarizing this relationship appears in [Figure 2.16](http://ruby.railstutorial.org/chapters/a-demo-app#fig-demo_model_inheritance). It is by inheriting from**ActiveRecord::Base** that our model objects gain the ability to communicate with the database, treat the database columns as Ruby attributes, and so on.

**Listing 2.12.** The **User** class, with inheritance.   
**app/models/user.rb**

**class** **User** < ActiveRecord::Base

.

.

.

**end**

**Listing 2.13.** The **Micropost** class, with inheritance.   
**app/models/micropost.rb**

**class** **Micropost** < ActiveRecord::Base

.

.

.

**end**

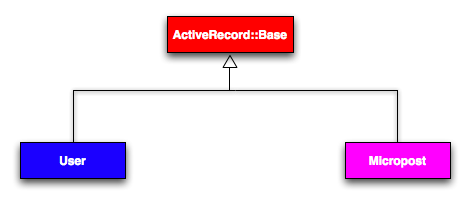


Figure 2.16: The inheritance hierarchy for the User and Micropost models.

The inheritance structure for controllers is only slightly more complicated. Comparing [Listing 2.14](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_users_controller_class)and [Listing 2.15](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_microposts_controller_class), we see that both the Users controller and the Microposts controller inherit from the Application controller. Examining [Listing 2.16](http://ruby.railstutorial.org/chapters/a-demo-app#code-demo_application_controller_class), we see that **ApplicationController** itself inherits from **ActionController::Base**; this is the base class for controllers provided by the Rails library Action Pack. The relationships between these classes is illustrated in [Figure 2.17](http://ruby.railstutorial.org/chapters/a-demo-app#fig-demo_controller_inheritance).

**Listing 2.14.** The **UsersController** class, with inheritance.   
**app/controllers/users\_controller.rb**

**class** **UsersController** < ApplicationController

.

.

.

**end**

**Listing 2.15.** The **MicropostsController** class, with inheritance.   
**app/controllers/microposts\_controller.rb**

**class** **MicropostsController** < ApplicationController

.

.

.

**end**

**Listing 2.16.** The **ApplicationController** class, with inheritance.   
**app/controllers/application\_controller.rb**

**class** **ApplicationController** < ActionController::Base

.

.

.

**end**

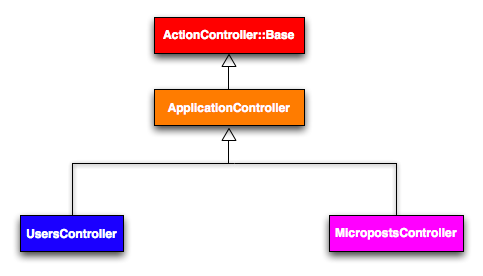


Figure 2.17: The inheritance hierarchy for the Users and Microposts controllers.

As with model inheritance, by inheriting ultimately from **ActionController::Base** both the Users and Microposts controllers gain a large amount of functionality, such as the ability to manipulate model objects, filter inbound HTTP requests, and render views as HTML. Since all Rails controllers inherit from **ApplicationController**, rules defined in the Application controller automatically apply to every action in the application. For example, in [Section 8.2.1](http://ruby.railstutorial.org/chapters/sign-in-sign-out#sec-remember_me) we’ll see how to include helpers for signing in and signing out of all of the sample application’s controllers.

[2.3.5 Deploying the demo app](http://ruby.railstutorial.org/chapters/a-demo-app#sec-deploying_the_demo_app)

With the completion of the Microposts resource, now is a good time to push the repository up to GitHub:

**$** git add .

**$** git commit -m "Finish demo app"

**$** git push

Ordinarily, you should make smaller, more frequent commits, but for the purposes of this chapter a single big commit at the end is fine.

At this point, you can also deploy the demo app to Heroku as in [Section 1.4](http://ruby.railstutorial.org/chapters/beginning#sec-deploying):

**$** heroku create

**$** git push heroku master

(As noted in [Section 1.4.1](http://ruby.railstutorial.org/chapters/beginning#sec-heroku_setup), some readers have reported needing to precompile static assets (such as CSS and images), which can be included by hand as follows:

**#** This should only be used **if** your Heroku deploy fails without it.

**$** rake assets:precompile

**$** git add .

**$** git commit -m "Add precompiled assets for Heroku"

**$** git push heroku master

This shouldn’t be necessary, and I have been unable to reproduce the issue, but the reports are common enough that I include it here for reference.)

To get the application’s database to work, you’ll also have to migrate the production database:

**$** heroku run rake db:migrate

This updates the database at Heroku with the necessary user/micropost data model.

Rake is *Ruby make*, a make-like language written in Ruby. Rails uses Rake extensively, especially for the innumerable little administrative tasks necessary when developing database-backed web applications. The **rake db:migrate** command is probably the most common, but there are many others; you can see a list of database tasks using **-T db**:

$ bundle exec rake -T db

To see all the Rake tasks available, run

$ bundle exec rake -T

The list is likely to be overwhelming, but don’t worry, you don’t have to know all (or even most) of these commands. By the end of the *Rails Tutorial*, you’ll know all the most important ones.