Please see the post **Do not buy the print version of the Ruby on Rails Tutorial (yet)**

Ruby on Rails Tutorial

Learn Web Development with Rails

Michael Hartl

Contents

Foreword

My former company (CD Baby) was one of the first to loudly switch to Ruby on Rails, and then even more loudly switch back to PHP (Google me to read about the drama). This book by Michael Hartl came so highly recommended that I had to try it, and the *Ruby on Rails Tutorial* is what I used to switch back to Rails again.

Though I’ve worked my way through many Rails books, this is the one that finally made me “get” it. Everything is done very much “the Rails way”—a way that felt very unnatural to me before, but now after doing this book finally feels natural. This is also the only Rails book that does test-driven development the entire time, an approach highly recommended by the experts but which has never been so clearly demonstrated before. Finally, by including Git, GitHub, and Heroku in the demo examples, the author really gives you a feel for what it’s like to do a real-world project. The tutorial’s code examples are not in isolation.

The linear narrative is such a great format. Personally, I powered through the *Rails Tutorial* in three long days, doing all the examples and challenges at the end of each chapter. Do it from start to finish, without jumping around, and you’ll get the ultimate benefit.

Enjoy!

Derek Sivers (sivers.org)
Formerly: Founder, CD Baby
Currently: Founder, Thoughts Ltd.

Acknowledgments

The *Ruby on Rails Tutorial* owes a lot to my previous Rails book, *RailsSpace*, and hence to my coauthor Aurelius Prochazka. I'd like to thank Aure both for the work he did on that book and for his support of this one. I'd also like to thank Debra Williams Cauley, my editor on both *RailsSpace* and the *Ruby on Rails Tutorial*; as long as she keeps taking me to baseball games, I'll keep writing books for her.

I'd like to acknowledge a long list of Rubyists who have taught and inspired me over the years: David Heinemeier Hansson, Yehuda Katz, Carl Lerche, Jeremy Kemper, Xavier Noria, Ryan Bates, Geoffrey Grosenbach, Peter Cooper, Matt Aimonetti, Gregg Pollack, Wayne E. Seguin, Amy Hoy, Dave Chelimsky, Pat Maddox, Tom Preston-Werner, Chris Wanstrath, Chad Fowler, Josh Susser, Obie Fernandez, Ian McFarland, Steven Bristol, Wolfram Arnold, Alex Chaffee, Giles Bowkett, Evan Dorn, Long Nguyen, James Lindenbaum, Adam Wiggins, Tikhon Bernstam, Ron Evans, Wyatt Greene, Miles Forrest, the good people at Pivotal Labs, the Heroku gang, the thoughtbot guys, and the GitHub crew. Finally, many, many readers—far too many to list—have contributed a huge number of bug reports and suggestions during the writing of this book, and I gratefully acknowledge their help in making it as good as it can be.

About the author

Michael Hartl is the author of the *Ruby on Rails Tutorial*, the leading introduction to web development with Ruby on Rails. His prior experience includes writing and developing *RailsSpace*, an extremely obsolete Rails tutorial book, and developing Insoshi, a once-popular and now-obsolete social networking platform in Ruby on Rails. In 2011, Michael received a Ruby Hero Award for his contributions to the Ruby community. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

Copyright and license

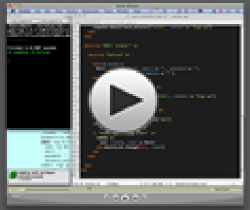
Chapter 6
Modeling users

In Chapter 5, we ended with a stub page for creating new users (Section 5.4); over the course of the next four chapters, we'll fulfill the promise implicit in this incipient signup page. The first critical step is to create a *data model* for users of our site, together with a way to store that data. In Chapter 7, we'll give users the ability to sign up for our site and create a user profile page. Once users can sign up, we'll let them sign in and sign out as well (Chapter 8), and in Chapter 9 (Section 9.2.1) we'll learn how to protect pages from improper access. Taken together, the material in Chapter 6 through Chapter 9 develops a full Rails login and authentication system. As you may know, there are various pre-built authentication solutions for Rails; Box 6.1 explains why, at least at first, it's probably a better idea to roll your own.

This is a long and action-packed chapter, and you may find it unusually challenging, especially if you are new to data modeling. By the end of it, though, we will have created an industrial-strength system for validating, storing, and retrieving user information.

Box 6.1. Roll your own authentication system

Virtually all web applications require a login and authentication system of some sort. As a result, most web frameworks have a plethora of options for implementing such systems, and Rails is no exception. Examples of authentication and authorization systems include Clearance, Authlogic, Devise, and CanCan (as well as non-Rails-specific solutions built on top of OpenID or OAuth). It's reasonable to ask why we should reinvent the wheel. Why not just use an off-the-shelf solution instead of rolling our own?



For one, practical experience shows that authentication on most sites requires extensive customization, and modifying a third-party product is often more work than writing the system from scratch. In addition, off-the-shelf systems can be "black boxes", with potentially mysterious innards; when you write your own system, you are far more likely to understand it. Moreover, recent additions to Rails (Section 6.3) make it easy to write a custom authentication system. Finally, if you *do* end up using a third-party system later on, you'll be in a much better position to understand and modify it if you've first built one yourself.

As usual, if you're following along using Git for version control, now would be a good time to make a topic branch for modeling users:

```
$ git checkout master
$ git checkout -b modeling-users
```

(The first line here is just to make sure that you start on the master branch, so that the **`modeling-users`** topic branch is based on **`master`**. You can skip that command if you're already on the master branch.)

6.1 User model

Although the ultimate goal of the next three chapters is to make a signup page for our site (mocked up in Figure 6.1), it would do little good now to accept information for new users: we don't currently have any place to put it. Thus, the first step in signing up users is to make a data structure to capture and store their information.

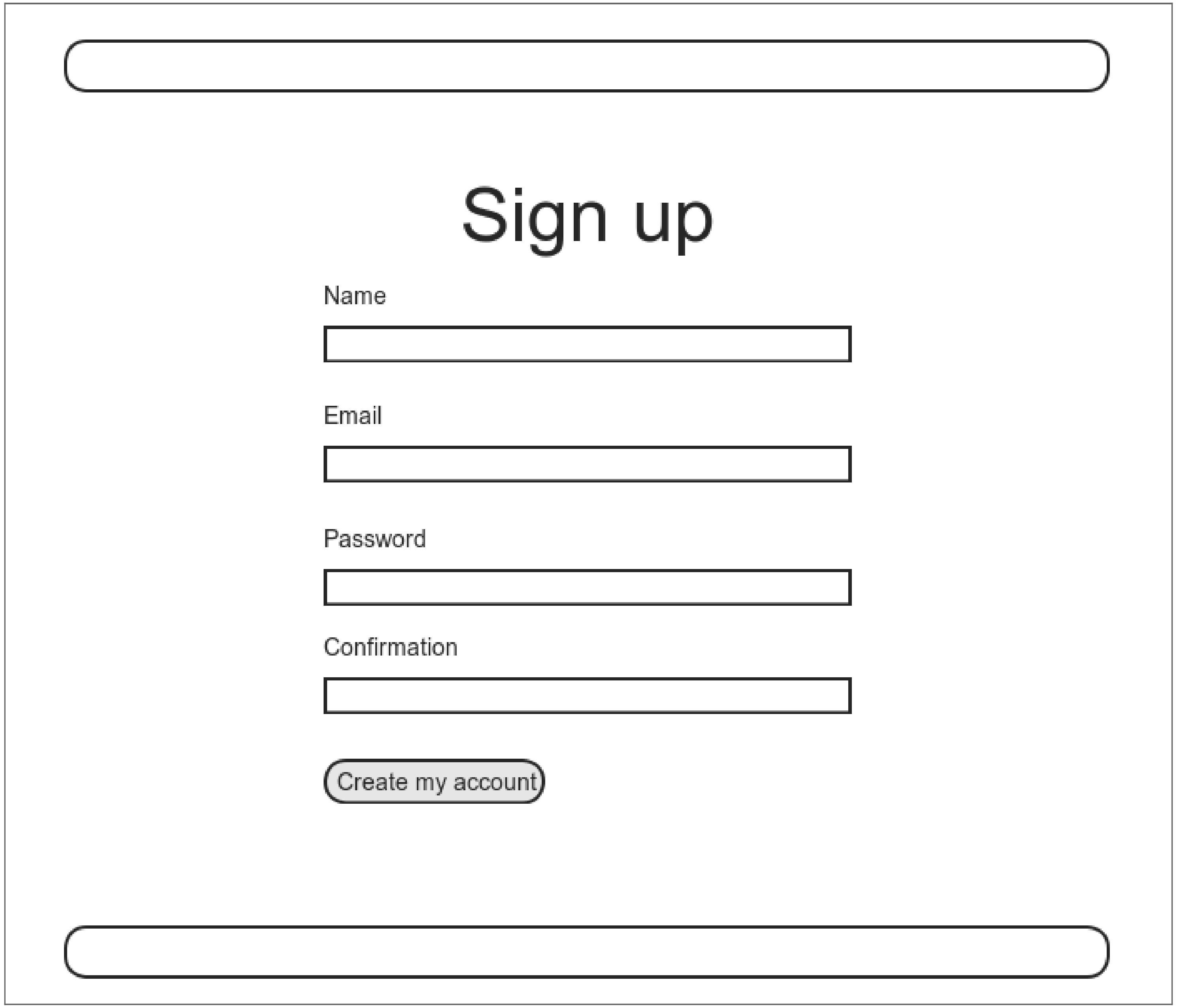


Figure 6.1: A mockup of the user signup page. [(full size)]

In Rails, the default data structure for a data model is called, naturally enough, a *model* (the M in

MVC from Section 1.2.6). The default Rails solution to the problem of persistence is to use a *database* for long-term data storage, and the default library for interacting with the database is called *Active Record*.[1] Active Record comes with a host of methods for creating, saving, and finding data objects, all without having to use the structured query language (SQL)[2] used by relational databases. Moreover, Rails has a feature called *migrations* to allow data definitions to be written in pure Ruby, without having to learn an SQL data definition language (DDL). The effect is that Rails insulates you almost entirely from the details of the data store. In this book, by using SQLite for development and PostgreSQL (via Heroku) for deployment (Section 1.4), we have developed this theme even further, to the point where we barely ever have to think about how Rails stores data, even for production applications.

6.1.1 Database migrations

You may recall from Section 4.4.5 that we have already encountered, via a custom-built **`User`** class, user objects with **`name`** and **`email`** attributes. That class served as a useful example, but it lacked the critical property of *persistence*: when we created a User object at the Rails console, it disappeared as soon as we exited. Our goal in this section is to create a model for users that won't disappear quite so easily.

As with the User class in Section 4.4.5, we'll start by modeling a user with two attributes, a **`name`** and an **`email`** address, the latter of which we'll use as a unique username.[3] (We'll add an attribute for passwords in Section 6.3.) In Listing 4.9, we did this with Ruby's **`attr_accessor`** method:

```
class User
  attr_accessor :name, :email
  .
  .
  .
end
```

In contrast, when using Rails to model users we don't need to identify the attributes explicitly. As noted briefly above, to store data Rails uses a relational database by default, which consists of *tables* composed of data *rows*, where each row has *columns* of data attributes. For example, to store users with names and email addresses, we'll create a **`users`** table with **`name`** and **`email`** columns (with each row corresponding to one user). By naming the columns in this way, we'll let Active Record figure out the User object attributes for us.

Let's see how this works. (If this discussion gets too abstract for your taste, be patient; the console examples starting in Section 6.1.3 and the database browser screenshots in Figure 6.3 and Figure 6.6 should make things clearer.) You may recall from Listing 5.28 that we created a Users controller (along with a **`new`** action) using the command

```
$ rails generate controller Users new --no-test-framework
```

There is an analogous command for making a model: **`generate model`**. Listing 6.1 shows the command to generate a User model with two attributes, **`name`** and **`email`**.

Listing 6.1. Generating a User model.

```
$ rails generate model User name:string email:string
      invoke  active_record
      create    db/migrate/[timestamp]_create_users.rb
      create    app/models/user.rb
      invoke    rspec
      create      spec/models/user_spec.rb
```

(Note that, in contrast to the plural convention for controller names, model names are singular: a Users controller, but a User model.) By passing the optional parameters **`name:string`** and **`email:string`**, we tell Rails about the two attributes we want, along with what types those attributes should be (in this case, **`string`**). Compare this with including the action names in

Listing 3.4 and Listing 5.28.

One of the results of the `generate` command in Listing 6.1 is a new file called a *migration*. Migrations provide a way to alter the structure of the database incrementally, so that our data model can adapt to changing requirements. In the case of the User model, the migration is created automatically by the model generation script; it creates a `users` table with two columns, `name` and `email`, as shown in Listing 6.2. (We'll see in Section 6.2.5 and again in Section 6.3 how to make a migration from scratch.)

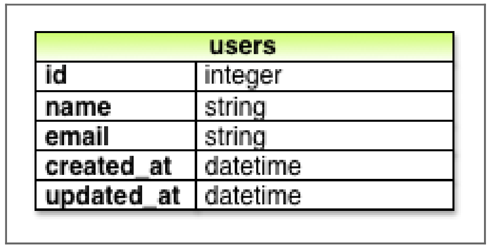
Listing 6.2. Migration for the User model (to create a `users` table).
`db/migrate/[timestamp]_create_users.rb`

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :name
      t.string :email

      t.timestamps
    end
  end
end
```

Note that the name of the migration file is prefixed by a *timestamp* based on when the migration was generated. In the early days of migrations, the filenames were prefixed with incrementing integers, which caused conflicts for collaborating teams if multiple programmers had migrations with the same number. Barring the improbable scenario of migrations generated the same second, using timestamps conveniently avoids such collisions.

The migration itself consists of a `change` method that determines the change to be made to the database. In the case of Listing 6.2, `change` uses a Rails method called `create_table` to create a *table* in the database for storing users. The `create_table` method accepts a block (Section 4.3.2) with one block variable, in this case called **`t`** (for "table"). Inside the block, the **`create_table`** method uses the **`t`** object to create **`name`** and **`email`** columns in the database, both of type **`string`**.[4] Here the table name is plural (**`users`**) even though the model name is singular (User), which reflects a linguistic convention followed by Rails: a model represents a single user, whereas a database table consists of many users. The final line in the block, **`t.timestamps`**, is a special command that creates two *magic columns* called **`created_at`** and **`updated_at`**, which are timestamps that automatically record when a given user is created and updated. (We'll see concrete examples of the magic columns starting in Section 6.1.3.) The full data model represented by this migration is shown in Figure 6.2.



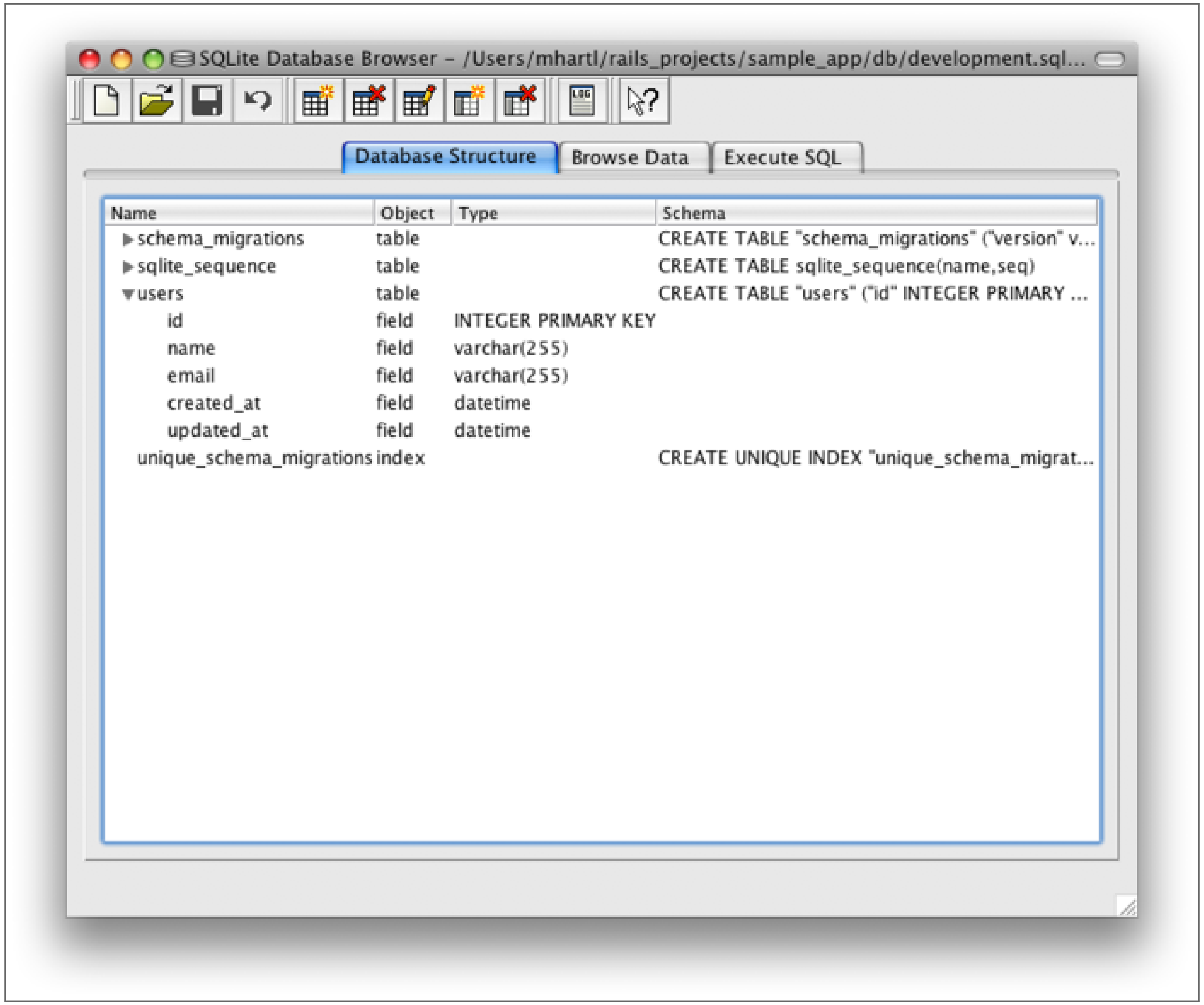
users	
id	integer
name	string
email	string
created_at	datetime
updated_at	datetime

Figure 6.2: The users data model produced by Listing 6.2.

We can run the migration, known as "migrating up", using the **`rake`** command (Box 2.1) as follows:

```
$ bundle exec rake db:migrate
```

(You may recall that we ran this command once before, in Section 2.2.) The first time **`db:migrate`** is run, it creates a file called **`db/development.sqlite3`**, which is an SQLite[5] database. We can see the structure of the database using the excellent SQLite Database Browser to open the **`db/development.sqlite3`** file (Figure 6.3); compare with the diagram in Figure 6.2. You might note that there's one column in Figure 6.3 not accounted for in the migration: the **`id`** column. As noted briefly in Section 2.2, this column is created automatically, and is used by Rails to identify each row uniquely.


SQLite Database Browser - /Users/mhartl/rails_projects/sample_app/db/development.sql...
Database Structure
Browse Data
Execute SQL

Name	Object	Type	Schema
▶schema_migrations	table		CREATE TABLE "schema_migrations" ("version" v...
▶sqlite_sequence	table		CREATE TABLE sqlite_sequence(name,seq)
▼users	table		CREATE TABLE "users" ("id" INTEGER PRIMARY ...
id	field	INTEGER PRIMARY KEY	
name	field	varchar(255)	
email	field	varchar(255)	
created_at	field	datetime	
updated_at	field	datetime	
unique_schema_migrations	index		CREATE UNIQUE INDEX "unique_schema_migrat...

Figure 6.3: The SQLite Database Browser with our new `users` table. (full size)

Most migrations, including all the ones in the *Rails Tutorial*, are *reversible*, which means we can "migrate down" and undo them with a single Rake task, called **`db:rollback`**:

```
$ bundle exec rake db:rollback
```

(See Box 3.1 for another technique useful for reversing migrations.) Under the hood, this command executes the **`drop_table`** command to remove the users table from the database. The reason this works is that the **`change`** method knows that **`drop_table`** is the inverse of **`create_table`**, which means that the rollback migration can be easily inferred. In the case of an irreversible migration, such as one to remove a database column, it is necessary to define separate **`up`** and **`down`** methods in place of the single **`change`** method. Read about migrations in the Rails Guides for more information.

If you rolled back the database, migrate up again before proceeding:

```
$ bundle exec rake db:migrate
```

6.1.2 The model file

We've seen how the User model generation in Listing 6.1 generated a migration file (Listing 6.2), and we saw in Figure 6.3 the results of running this migration: it updated a file called **`development.sqlite3`** by creating a table **`users`** with columns **`id`**, **`name`**, **`email`**, **`created_at`**, and **`updated_at`**. Listing 6.1 also created the model itself; the rest of this section is dedicated to understanding it.

We begin by looking at the code for the User model, which lives in the file `user.rb` inside the `app/models/` directory. It is, to put it mildly, very compact (Listing 6.3). (*Note*: The `attr_accessible` line will not appear if you are using Rails 3.2.2 or earlier. In this case, you should add it in Section 6.1.2.2.)

Listing 6.3. The brand new User model.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email
end
```

Recall from Section 4.4.2 that the syntax `class User < ActiveRecord::Base` means that the `User` class *inherits* from `ActiveRecord::Base`, so that the User model automatically has all the functionality of the `ActiveRecord::Base` class. Of course, knowledge of this inheritance doesn't do any good unless we know what `ActiveRecord::Base` contains, and we'll get a first look momentarily. Before we move on, though, there are two tasks to complete.

Model annotation

Although it's not strictly necessary, you might find it convenient to *annotate* your Rails models using the `annotate` gem (Listing 6.4).

Listing 6.4. Adding the `annotate` gem to the `Gemfile`.

```
source 'https://rubygems.org'
.
.
.
group :development, :test do
  gem 'sqlite3', '1.3.5'
  gem 'rspec-rails', '2.11.0'
```

```
end

gem 'annotate', '2.5.0', group: :development

group :test do
   .
   .
   .
end
```

(We place the `annotate` gem in a `group :development` block (analogous to `group :test`) because the annotations aren't needed in production applications.) We next install it with `bundle install`:

```
$ bundle install
```

This gives us a command called `annotate`, which simply adds comments containing the data model to the model file:

```
$ bundle exec annotate
Annotated (1): User
```

The results appear in Listing 6.5.

Listing 6.5. The annotated User model.
`app/models/user.rb`

```
# == Schema Information
#
# Table name: users
#
```

```
#  id         :integer         not null, primary key
#  name       :string(255)
#  email      :string(255)
#  created_at :datetime
#  updated_at :datetime
#

class User < ActiveRecord::Base
  attr_accessible :name, :email
end
```

I find that having the data model visible in the model files helps remind me which attributes the model has, but future code listings will omit the annotations for brevity. (Note that, if you want your annotations to be up-to-date, you'll have to run `annotate` again any time the data model changes.)

Accessible attributes

Let's revisit the User model, focusing now on the `attr_accessible` line (Listing 6.6). This line tells Rails which attributes of the model are *accessible*, i.e., which attributes can be modified automatically by outside users (such as users submitting requests with web browsers).

Listing 6.6. Making the `name` and `email` attributes accessible.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email
end
```

The code in Listing 6.6 doesn't do quite what you might think. By default, *all* model attributes are accessible. What Listing 6.6 does is to ensure that the `name` and `email` attributes—and *only* the `name` and `email` attributes—are automatically accessible to outside users. We'll see why this is important in Chapter 9: using **`attr_accessible`** is important for preventing a *mass assignment* vulnerability, a distressingly common and often serious security hole in many Rails applications.

6.1.3 Creating user objects

We've done some good prep work, and now it's time to cash in and learn about Active Record by playing with our newly created User model. As in Chapter 4, our tool of choice is the Rails console. Since we don't (yet) want to make any changes to our database, we'll start the console in a *sandbox*:

```
$ rails console --sandbox
Loading development environment in sandbox
Any modifications you make will be rolled back on exit
>>
```

As indicated by the helpful message "Any modifications you make will be rolled back on exit", when started in a sandbox the console will "roll back" (i.e., undo) any database changes introduced during the session.

In the console session in Section 4.4.5, we created a new user object with **`User.new`**, which we had access to only after requiring the example user file in Listing 4.9. With models, the situation is different; as you may recall from Section 4.4.4, the Rails console automatically loads the Rails environment, which includes the models. This means that we can make a new user object without any further work:

```
>> User.new
=> #<User id: nil, name: nil, email: nil, created_at: nil, updated_at: nil>
```

We see here the default console representation of a user object, which prints out the same attributes shown in Figure 6.2 and Listing 6.5.

When called with no arguments, **`User.new`** returns an object with all **`nil`** attributes. In Section 4.4.5, we designed the example User class to take an *initialization hash* to set the object attributes; that design choice was motivated by Active Record, which allows objects to be initialized in the same way:

```
>> user = User.new(name: "Michael Hartl", email: "mhartl@example.com")
=> #<User id: nil, name: "Michael Hartl", email: "mhartl@example.com",
created_at: nil, updated_at: nil>
```

Here we see that the name and email attributes have been set as expected.

If you've been tailing the development log, you may have noticed that no new lines have shown up yet. This is because calling **`User.new`** doesn't touch the database; it simply creates a new Ruby object in memory. To save the user object to the database, we call the **`save`** method on the **`user`** variable:

```
>> user.save
=> true
```

The **`save`** method returns **`true`** if it succeeds and **`false`** otherwise. (Currently, all saves should succeed; we'll see cases in Section 6.2 when some will fail.) As soon as you save, you should see a line in the development log with the SQL command to **`INSERT INTO "users"`**. Because of the many methods supplied by Active Record, we won't ever need raw SQL in this book, and I'll omit discussion of the SQL commands from now on. But you can learn a lot by watching the log.

You may have noticed that the new user object had **`nil`** values for the **`id`** and the magic columns **`created_at`** and **`updated_at`** attributes. Let's see if our **`save`** changed anything:

```
>> user
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2011-12-05 00:57:46", updated_at: "2011-12-05 00:57:46">
```

We see that the **`id`** has been assigned a value of **`1`**, while the magic columns have been assigned the current time and date.[6] Currently, the created and updated timestamps are identical; we'll see them differ in Section 6.1.5.

As with the User class in Section 4.4.5, instances of the User model allow access to their attributes using a dot notation:[7]

```
>> user.name
=> "Michael Hartl"
>> user.email
=> "mhartl@example.com"
>> user.updated_at
=> Tue, 05 Dec 2011 00:57:46 UTC +00:00
```

As we'll see in Chapter 7, it's often convenient to make and save a model in two steps as we have above, but Active Record also lets you combine them into one step with **`User.create`**:

```
>> User.create(name: "A Nother", email: "another@example.org")
#<User id: 2, name: "A Nother", email: "another@example.org", created_at:
"2011-12-05 01:05:24", updated_at: "2011-12-05 01:05:24">
>> foo = User.create(name: "Foo", email: "foo@bar.com")
#<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2011-12-05
01:05:42", updated_at: "2011-12-05 01:05:42">
```

Note that **`User.create`**, rather than returning **`true`** or **`false`**, returns the User object itself, which we can optionally assign to a variable (such as **`foo`** in the second command above).

The inverse of **`create`** is **`destroy`**:

```
>> foo.destroy
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2011-12-05
01:05:42", updated_at: "2011-12-05 01:05:42">
```

Oddly, **`destroy`**, like **`create`**, returns the object in question, though I can't recall ever having used the return value of **`destroy`**. Even odder, perhaps, is that the **`destroy`**ed object still exists in memory:

```
>> foo
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2011-12-05
01:05:42", updated_at: "2011-12-05 01:05:42">
```

How do we know if we really destroyed an object? And for saved and non-destroyed objects, how can we retrieve users from the database? It's time to learn how to use Active Record to find user objects.

6.1.4 Finding user objects

Active Record provides several options for finding objects. Let's use them to find the first user we created while verifying that the third user (**`foo`**) has been destroyed. We'll start with the existing user:

```
>> User.find(1)
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2011-12-05 00:57:46", updated_at: "2011-12-05 00:57:46">
```

Here we've passed the id of the user to **`User.find`**; Active Record returns the user with that id.

Let's see if the user with an **`id`** of **`3`** still exists in the database:

```
>> User.find(3)
ActiveRecord::RecordNotFound: Couldn't find User with ID=3
```

Since we destroyed our third user in Section 6.1.3, Active Record can't find it in the database. Instead, **`find`** raises an *exception*, which is a way of indicating an exceptional event in the execution of a program—in this case, a nonexistent Active Record id, which causes **`find`** to raise an **`ActiveRecord::RecordNotFound`** exception.[8]

In addition to the generic **`find`**, Active Record also allows us to find users by specific attributes:

```
>> User.find_by_email("mhartl@example.com")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2011-12-05 00:57:46", updated_at: "2011-12-05 00:57:46">
```

The **`find_by_email`** method is automatically created by Active Record based on the **`email`** attribute in the **`users`** table. (As you might guess, Active Record creates a **`find_by_name`** method as well.) Since we will be using email addresses as usernames, this sort of **`find`** will be useful when we learn how to let users sign in to our site (Chapter 7). If you're worried that **`find_by_email`** will be inefficient if there are a large number of users, you're ahead of the game; we'll cover this issue, and its solution via database indices, in Section 6.2.5.

We'll end with a couple of more general ways of finding users. First, there's **`first`**:

```
>> User.first
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2011-12-05 00:57:46", updated_at: "2011-12-05 00:57:46">
```

Naturally, **`first`** just returns the first user in the database. There's also **`all`**:

```
>> User.all
=> [#<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2011-12-05 00:57:46", updated_at: "2011-12-05 00:57:46">,
#<User id: 2, name: "A Nother", email: "another@example.org", created_at:
"2011-12-05 01:05:24", updated_at: "2011-12-05 01:05:24">]
```

No prizes for inferring that **`all`** returns an array (Section 4.3.1) of all users in the database.

6.1.5 Updating user objects

Once we've created objects, we often want to update them. There are two basic ways to do this. First, we can assign attributes individually, as we did in Section 4.4.5:

```
>> user           # Just a reminder about our user's attributes
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2011-12-05 00:57:46", updated_at: "2011-12-05 00:57:46">
>> user.email = "mhartl@example.net"
=> "mhartl@example.net"
>> user.save
=> true
```

Note that the final step is necessary to write the changes to the database. We can see what happens without a save by using **`reload`**, which reloads the object based on the database information:

```
>> user.email
=> "mhartl@example.net"
>> user.email = "foo@bar.com"
=> "foo@bar.com"
>> user.reload.email
=> "mhartl@example.net"
```

Now that we've updated the user, the magic columns differ, as promised in Section 6.1.3:

```
>> user.created_at
=> "2011-12-05 00:57:46"
>> user.updated_at
=> "2011-12-05 01:37:32"
```

The second way to update attributes is to use `update_attributes`:

```
>> user.update_attributes(name: "The Dude", email: "dude@abides.org")
=> true
>> user.name
=> "The Dude"
>> user.email
=> "dude@abides.org"
```

The `update_attributes` 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). It's worth noting that, once you have defined some attributes as accessible using `attr_accessible` (Section 6.1.2.2), *only* those attributes can be modified using `update_attributes`. If you ever find that your models mysteriously start refusing to update certain columns, check to make sure that those columns are included in the call to `attr_accessible`.

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 sign 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*. In this section, we'll cover several of the most common cases, validating *presence*, *length*, *format* and *uniqueness*. In Section 6.3.4 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 Initial user tests

As with the other features of our sample app, we'll add User model validations using test-driven development. Because we didn't pass the

```
--no-test-framework
```

flag when we generated the User model (unlike, e.g., Listing 5.28), the command in Listing 6.1 produces an initial spec for testing users, but in this case it's practically blank (Listing 6.7).

Listing 6.7. The practically blank default User spec.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
```

```
  pending "add some examples to (or delete) #{__FILE__}"
end
```

This simply uses the **`pending`** method to indicate that we should fill the spec with something useful. We can see its effect by running the User model spec:

```
$ bundle exec rspec spec/models/user_spec.rb
*


Finished in 0.01999 seconds
1 example, 0 failures, 1 pending

Pending:
  User add some examples to (or delete)
  /Users/mhartl/rails_projects/sample_app/spec/models/user_spec.rb
  (Not Yet Implemented)
```

On many systems, pending specs will be displayed in yellow to indicate that they are in between passing (green) and failing (red).

We'll follow the advice of the default spec by filling it in with some RSpec examples, shown in Listing 6.8.

Listing 6.8. Testing for the **`:name`** and **`:email`** attributes.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before { @user = User.new(name: "Example User", email: "user@example.com") }

  subject { @user }
  it { should respond_to(:name) }
  it { should respond_to(:email) }
end
```

The **`before`** block, which we saw in Listing 5.27), runs the code inside the block before each example—in this case, creating a new **`@user`** instance variable using **`User.new`** and a valid initialization hash. Then

```
subject { @user }
```

makes **`@user`** the default subject of the test example, as seen before in the context of the **`page`** variable in Section 5.3.4.

The two examples in Listing 6.8 test for the existence of **`name`** and **`email`** attributes:

```
it { should respond_to(:name) }
it { should respond_to(:email) }
```

These examples implicitly use the Ruby method **`respond_to?`**, which accepts a symbol and returns **`true`** if the object responds to the given method or attribute and **`false`** otherwise:

```
$ rails console --sandbox
>> user = User.new
>> user.respond_to?(:name)
=> true
>> user.respond_to?(:foobar)
=> false
```

(Recall from Section 4.2.3 that Ruby uses a question mark to indicate such true/false boolean methods.) The tests themselves rely on the *boolean convention* used by RSpec: the code

```
@user.respond_to?(:name)
```

can be tested using the RSpec code

```
@user.should respond_to(:name)
```

Because of `subject { @user }`, we can leave off `@user` in the test, yielding

```
it { should respond_to(:name) }
```

These kinds of tests allow us to use TDD to add new attributes and methods to our User model, and as a side-effect we get a nice specification for the methods that all `User` objects should respond to.

You should verify at this point that the tests fail:

```
$ bundle exec rspec spec/
```

Even though we created a development database with `rake db:migrate` in Section 6.1.1, the tests fail because the *test database* doesn't yet know about the data model (indeed, it doesn't yet exist at all). We can create a test database with the correct structure, and thereby get the tests to pass, using the `db:test:prepare` Rake task:

```
$ bundle exec rake db:test:prepare
```

This just ensures that the data model from the development database, `db/development.sqlite3`, is reflected in the test database, `db/test.sqlite3`. Failure to run this Rake task after a migration is a common source of confusion. In addition, sometimes the test database gets corrupted and needs to be reset. If your test suite is mysteriously breaking, be sure to try running `rake db:test:prepare` to see if that fixes the problem.

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 7.3.2, 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. Although the first step in TDD is to write a *failing* test (Section 3.2.1), in this case we don't yet know enough about validations to write the proper test, so we'll write the validation first, using the console to understand it. Then we'll comment out the validation, write a failing test, and verify that uncommenting the validation gets the test to pass. This procedure may seem pedantic for such a simple test, but I have seen many "simple" tests that actually test the wrong thing; being meticulous about TDD is simply the *only* way to be confident that we're testing the right thing. (This comment-out technique is also useful when rescuing an application whose application code is already written but—*quelle horreur!*—has no tests.)

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.

`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  validates :name, presence: true
end
```

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

```
class User < ActiveRecord::Base
  attr_accessible(:name, :email)

  validates(:name, presence: true)
end
```

Let's drop into the console to see the effects of adding a validation to our User model:[9]

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

Here **`user.save`** returns **`false`**, indicating a failed save. In the final command, we use 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.)

Now for the failing test. To ensure that our incipient test will fail, let's comment out the validation at this point (Listing 6.10).

Listing 6.10. Commenting out a validation to ensure a failing test.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  # validates :name, presence: true
end
```

The initial validation tests then appear as in Listing 6.11.

Listing 6.11. A failing test for validation of the `name` attribute.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end

  subject { @user }

  it { should respond_to(:name) }
  it { should respond_to(:email) }

  it { should be_valid }

  describe "when name is not present" do
    before { @user.name = " " }
    it { should_not be_valid }
  end
end
```

The first new example is just a sanity check, verifying that the `@user` object is initially valid:

```
it { should be_valid }
```

This is another example of the RSpec boolean convention we saw in Section 6.2.1: whenever an object responds to a boolean method `foo?`, there is a corresponding test method called `be_foo`. In this case, we can test the result of calling

```
@user.valid?
```

with

```
@user.should be_valid
```

As before, `subject { @user }` lets us leave off `@user`, yielding

```
it { should be_valid }
```

The second test first sets the user’s name to an invalid (blank) value, and then tests to see that the resulting `@user` object is invalid:

```
describe "when name is not present" do
  before { @user.name = " " }
  it { should_not be_valid }
end
```

This uses a `before` block to set the user’s name to an invalid (blank) value and then checks that the resulting user object is not valid.

You should verify that the tests fail at this point:

```
$ bundle exec rspec spec/models/user_spec.rb
...F
4 examples, 1 failure
```

Now uncomment the validation (i.e., revert Listing 6.10 back to Listing 6.9) to get the tests to pass:

```
$ bundle exec rspec spec/models/user_spec.rb
....
4 examples, 0 failures
```

Of course, we also want to validate the presence of email addresses. The test (Listing 6.12) is analogous to the one for the `name` attribute.

Listing 6.12. A test for presence of the `email` attribute.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when email is not present" do
    before { @user.email = " " }
    it { should_not be_valid }
  end
end
```

The implementation is also virtually the same, as seen in Listing 6.13.

Listing 6.13. Validating the presence of the `name` and `email` attributes.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  validates :name,  presence: true
  validates :email, presence: true
end
```

Now all the tests should pass, and the presence validations are complete.

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.

We start with a test. 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 (Listing 6.14).

Listing 6.14. A test for `name` length validation.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when name is too long" do
    before { @user.name = "a" * 51 }
    it { should_not be_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 test in Listing 6.14 should fail. To get it to pass, we need to know about the validation argument to constrain length, `:length`, along with the `:maximum` parameter to enforce the upper bound (Listing 6.15).

Listing 6.15. Adding a length validation for the `name` attribute.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  validates :name,  presence: true, length: { maximum: 50 }
  validates :email, presence: true
end
```

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

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 more stringent requirements. 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 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 (Listing 6.16).

Listing 6.16. Tests for email format validation.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when email format is invalid" do
    it "should be invalid" do
      addresses = %w[user@foo,com user_at_foo.org example.user@foo.
                     foo@bar_baz.com foo@bar+baz.com]
      addresses.each do |invalid_address|
        @user.email = invalid_address
        @user.should_not be_valid
      end
    end
  end
```

```
    describe "when email format is valid" do
      it "should be valid" do
        addresses = %w[user@foo.COM A_US-ER@f.b.org frst.lst@foo.jp a+b@baz.cn]
        addresses.each do |valid_address|
          @user.email = valid_address
          @user.should be_valid
        end
      end
    end
  end
```

As noted above, these are far from exhaustive, but we do check the common valid email forms `user@foo.COM`, `THE_US-ER@foo.bar.org` (uppercase, underscores, and compound domains), and `first.last@foo.jp` (the standard corporate username `first.last`, with a two-letter top-level domain `jp`), along with several invalid forms.

The application code for email format validation uses a *regular expression* (or *regex*) to define the format, along with the `:format` argument to the `validates` method (Listing 6.17).

Listing 6.17. Validating the email format with a regular expression.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  validates :name,  presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, 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

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

ensures that only email addresses that match the pattern will be considered valid. (Because it starts with a capital letter, **`VALID_EMAIL_REGEX`** is a Ruby *constant*, so its value can't change.)

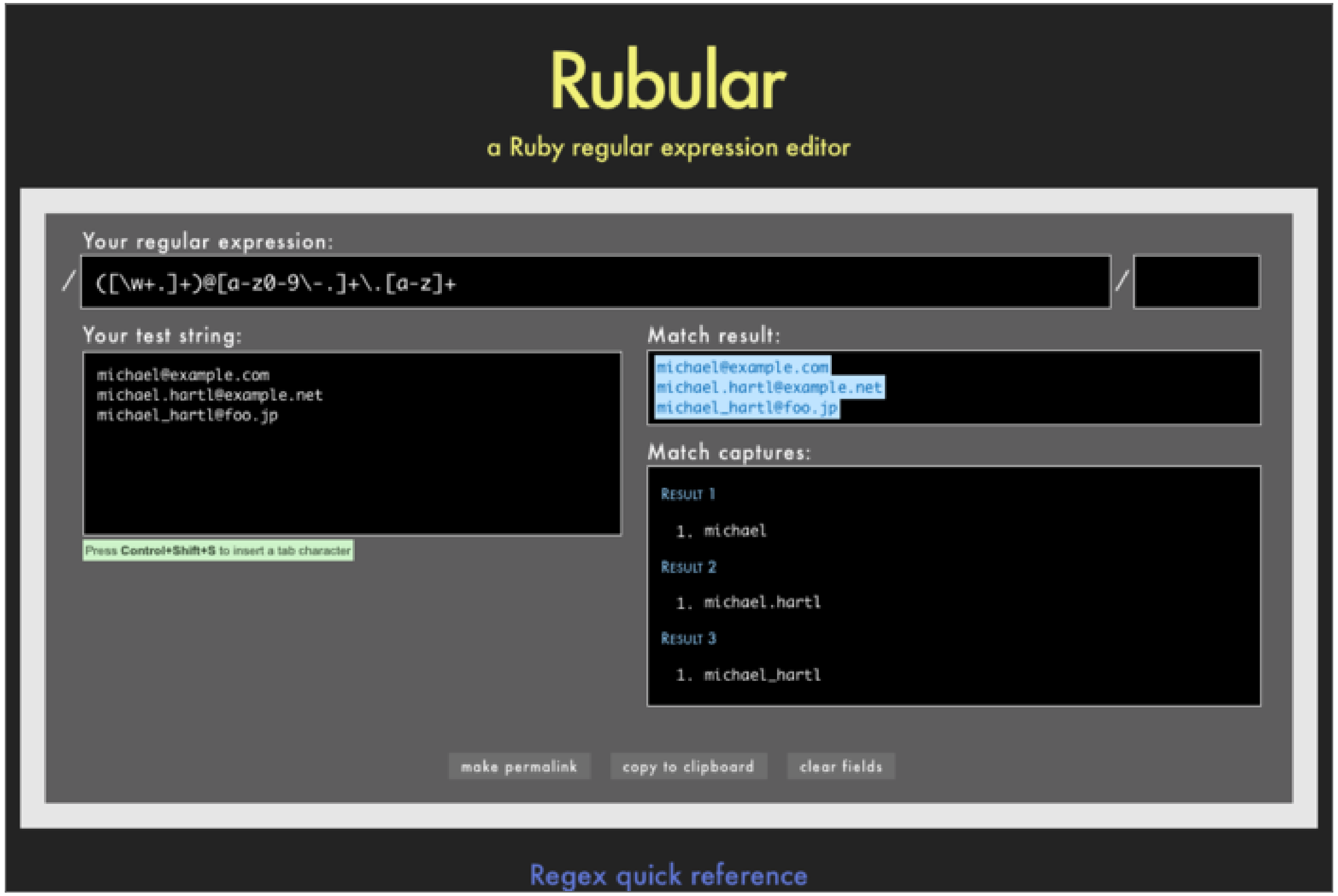
So, where does the pattern come from? Regular expressions consist of a terse (some would say unreadable) language for matching text patterns; learning to construct regexes is an art, and to get you started I've broken **`VALID_EMAIL_REGEX`** into bite-sized pieces (Table 6.1).[10] To really learn about regular expressions, though, I consider the amazing Rubular regular expression editor (Figure 6.4) to be simply essential.[11] 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 open to Rubular—no amount of reading about regular expressions can replace a couple of hours playing with Rubular. (Note: If you use the regex from Listing 6.17 in Rubular, you should leave off the `\A` and `\z` characters.)

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 email regex from Listing 6.17.

By the way, there actually exists a full regex for matching email addresses according to the official standard, but it's really not worth the trouble. The one in Listing 6.17 is fine, maybe even better than the official one.[12]



```
/([\w+.]+)@[a-z0-9\-.]+\.[a-z]+/
```

```
michael@example.com
michael.hartl@example.net
michael_hartl@foo.jp
```

```
michael@example.com
michael.hartl@example.net
michael_hartl@foo.jp
```

```
Result 1
  1. michael
Result 2
  1. michael.hartl
Result 3
  1. michael_hartl
```

Figure 6.4: The awesome Rubular regular expression editor. (full size)

The tests should all be passing now. (In fact, the tests for valid email addresses should have been passing all along; since regexes are notoriously error-prone, the valid email tests are there mainly as a sanity check on **`VALID_EMAIL_REGEX`**.) This means that there's only one constraint left: enforcing the email addresses to be unique.

6.2.5 Uniqueness validation

To enforce uniqueness of email addresses (so that we can use them as usernames), we'll be using the `:unique` 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, as usual, with our 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.[13] The (first) duplicate email test appears in Listing 6.18.

Listing 6.18. A test for the rejection of duplicate email addresses.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when email address is already taken" do
    before do
      user_with_same_email = @user.dup
      user_with_same_email.save
    end

    it { should_not be_valid }
  end
end
```

The method here is to make a user with the same email address as `@user`, which we accomplish using `@user.dup`, which creates a duplicate user with the same attributes. Since we then save that user, the original `@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 6.18 to pass with the code in Listing 6.19.

Listing 6.19. Validating the uniqueness of email addresses.
app/models/user.rb

```
class User < ActiveRecord::Base
  .
  .
  .
  validates :email, presence: true, format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
end
```

We're not quite done, though. Email addresses are case-insensitive—**foo@bar.com** goes to the same place as **FOO@BAR.COM** or **FoO@BAr.coM**—so our validation should cover this case as well. We test for this with the code in Listing 6.20.

Listing 6.20. A test for the rejection of duplicate email addresses, insensitive to case.
spec/models/user_spec.rb

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end
  .
  .
  .
  describe "when email address is already taken" do
    before do
      user_with_same_email = @user.dup
      user_with_same_email.email = @user.email.upcase
      user_with_same_email.save
    end

    it { should_not be_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 first duplicate email test, but with an upper-case 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"
>> user_with_same_email = user.dup
>> user_with_same_email.email = user.email.upcase
>> user_with_same_email.valid?
=> true
```

Of course, `user_with_same_email.valid?` is `true`, because the uniqueness validation is currently case-sensitive, but we want it to be `false`. Fortunately, `:uniqueness` accepts an option, `:case_sensitive`, for just this purpose (Listing 6.21).

Listing 6.21. Validating the uniqueness of email addresses, ignoring case.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  .
  .
  .
  validates :email, presence: true, format: { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }
end
```

Note that we have simply replaced `true` with `case_sensitive: false`; Rails infers in this case that `:uniqueness` should be `true`. At this point, our application—with an important caveat—enforces email uniqueness, and our test suite should pass.

The uniqueness caveat

There's just one small problem, the caveat alluded to above:

Using `validates :uniqueness` does not guarantee uniqueness.

D'oh! But what can go wrong? Here's what:

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. Luckily, the solution is straightforward to implement; we just need to enforce uniqueness at the database level as well. Our method is to create a database *index* on the email column, and then require that the index be unique.

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 pre-defined, so we need to fill in its contents with Listing 6.22.[14]

Listing 6.22. The migration for enforcing email uniqueness.
`db/migrate/[timestamp]_add_index_to_users_email.rb`

```
class AddIndexToUsersEmail < ActiveRecord::Migration
  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:

```
$ bundle exec rake db:migrate
```

(If this fails, try exiting any running sandbox console sessions, which can lock the database and prevent migrations.) If you're interested in seeing the practical effect of this, take a look at the file `db/schema.rb`, which should now include a line like this:

```
add_index "users", ["email"], :name => "index_users_on_email", :unique => true
```

Unfortunately, there's one more change we need to make to be assured of email uniqueness, which is to make sure that the email address is all lower-case before it gets saved to the database. The reason is that not all database adapters use case-sensitive indices.[15] The way to do this is with a *callback*, which is a method that gets invoked at a particular point in the lifetime of an Active Record object (see the Rails API entry on callbacks). In the present case, we'll use a `before_save` callback to force Rails to downcase the email attribute before saving the user to the database, as shown in Listing 6.23.

Listing 6.23. Ensuring email uniqueness by downcasing the email attribute.
`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email

  before_save { |user| user.email = email.downcase }
  .
  .
  .
end
```

The code in Listing 6.23 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. This code is a little advanced, and at this point I suggest you simply trust that it works; if you're skeptical, comment out the uniqueness validation from Listing 6.19 and try to create users with identical email addresses to see the error that results. (We'll see this technique again in Section 8.2.1.)

Now the Alice scenario above will work fine: the database will save a user record based on the first request, and will reject the second save for violating the uniqueness constraint. (An error will appear in the Rails log, but that doesn't do any harm. You can actually catch the `ActiveRecord::StatementInvalid` exception that gets raised—see Insoshi for an example—but in this tutorial we won't bother with this step.) Adding this index on the email attribute accomplishes a second goal, alluded to briefly in Section 6.1.4: it fixes an efficiency problem in `find_by_email` (Box 6.2).

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 sign 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. 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". 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.

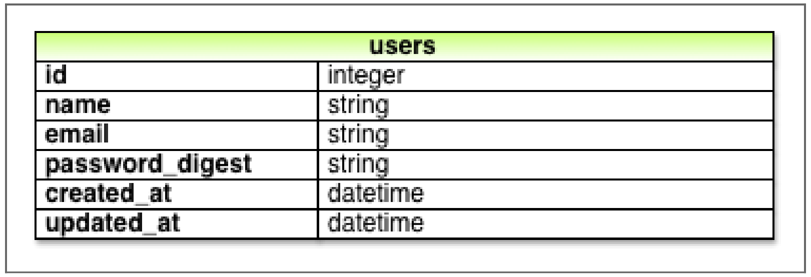
6.3 Adding a secure password

In this section we'll add the last of the basic User attributes: a secure password used to authenticate users of the sample application. The method is to require each user to have a password (with a password confirmation), and then store an encrypted version of the password in the database. 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 sign in to the site.

The method for authenticating users will be to take a submitted password, encrypt it, and compare the result to the encrypted value stored in the database. If the two match, then the submitted password is correct and the user is authenticated. By comparing encrypted values instead of raw passwords, we will be able to authenticate users without storing the passwords themselves, thereby avoiding a serious security hole.

6.3.1 An encrypted password

We'll start with the necessary change to the data model for users, which involves adding a **`password_digest`** column to the **`users`** table (Figure 6.5). The name *digest* comes from the terminology of cryptographic hash functions, and the exact name **`password_digest`** is necessary for the implementation in Section 6.3.4 to work. By encrypting the password properly, we'll ensure that an attacker won't be able to sign in to the site even if he manages to obtain a copy of the database.



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

Figure 6.5: The User model with an added **`password_digest`** attribute.

We'll use the state-of-the-art hash function called bcrypt to irreversibly encrypt the password to form the password hash. To use bcrypt in the sample application, we need to add the **`bcrypt-`**

ruby gem to our **Gemfile** (Listing 6.24).

Listing 6.24. Adding **bcrypt-ruby** to the **Gemfile**.

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

gem 'rails', '3.2.8'
gem 'bootstrap-sass', '2.0.4'
gem 'bcrypt-ruby', '3.0.1'
.
.
.
```

Then run **bundle install**:

```
$ bundle install
```

On some systems, you may get the warning

```
make: /usr/bin/gcc-4.2: No such file or directory
```

To fix this, reinstall RVM using the **clang** flag:

```
$ rvm reinstall 1.9.3 --with-gcc=clang
```

Since we want users to have a password digest column, a user object should respond to **password_digest**, which suggests the test shown in Listing 6.25.

Listing 6.25. Ensuring that a User object has a `password_digest` column.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

  before do
    @user = User.new(name: "Example User", email: "user@example.com")
  end

  subject { @user }

  it { should respond_to(:name) }
  it { should respond_to(:email) }
  it { should respond_to(:password_digest) }
  .
  .
  .
end
```

To get the test to pass, we first generate an appropriate migration for the `password_digest` column:

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

Here the first argument is the migration name, and we've also supplied a second argument with the name and type of attribute we want to create. (Compare this to the original generation of the `users` table in Listing 6.1.) We can choose any migration name we want, but it's convenient to end the name with `_to_users`, since in this case Rails automatically constructs a migration to add columns to the `users` table. Moreover, by including the second argument, we've given Rails enough information to construct the entire migration for us, as seen in Listing 6.26.

Listing 6.26. The migration to add a **`password_digest`** column to the **`users`** table.
`db/migrate/[ts]_add_password_digest_to_users.rb`

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

This code uses the **`add_column`** method to add a **`password_digest`** column to the **`users`** table.

We can get the failing test from Listing 6.25 to pass by migrating the development database and preparing the test database:

```
$ bundle exec rake db:migrate
$ bundle exec rake db:test:prepare
$ bundle exec rspec spec/
```

6.3.2 Password and confirmation

As seen in the mockup in Figure 6.1, we expect to have users confirm their passwords, a common practice on the web meant to minimize typos. We could enforce this at the controller layer, but it's conventional to put it in the model and use Active Record to enforce the constraint. The method is to add **`password`** and **`password_confirmation`** attributes to the User model, and then require that the two attributes match before the record is saved to the database. Unlike the other attributes we've seen so far, the password attributes will be *virtual*—they will only exist temporarily in memory, and will not be persisted to the database.

We'll start with **`respond_to`** tests for a password and its confirmation, as seen in Listing 6.27.

Listing 6.27. Testing for the `password` and `password_confirmation` attributes.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

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

  subject { @user }

  it { should respond_to(:name) }
  it { should respond_to(:email) }
  it { should respond_to(:password_digest) }
  it { should respond_to(:password) }
  it { should respond_to(:password_confirmation) }

  it { should be_valid }
  .
  .
  .
end
```

Note that we've added `:password` and `:password_confirmation` to the initialization hash for `User.new`:

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

We definitely don't want users to enter a blank password, so we'll add another test to validate password presence:

```
describe "when password is not present" do
  before { @user.password = @user.password_confirmation = " " }
  it { should_not be_valid }
end
```

Since we'll be testing password mismatch in a moment, here we make sure to test the *presence* validation by setting both the password and its confirmation to a blank string. This uses Ruby's ability to make more than one assignment in a line. For example, in the console we can set both `a` and `b` to `3` as follows:

```
>> a = b = 3
>> a
=> 3
>> b
=> 3
```

In the present case, we use it to set both password attributes to `" "`:

```
@user.password = @user.password_confirmation = " "
```

We also want to ensure that the password and confirmation match. The case where they *do* match is already covered by `it { should be_valid }`, so we only need to test the case of a mismatch:

```
describe "when password doesn't match confirmation" do
  before { @user.password_confirmation = "mismatch" }
  it { should_not be_valid }
end
```

In principle, we are now done, but there is one case that doesn't quite work. What if the password confirmation is blank? If it is empty or consists of whitespace but the password is valid, then the two don't match and the confirmation validation will catch it. If both the password and its confirmation are empty or consist of whitespace, then the password presence validation will catch it. Unfortunately, there's one more possibility, which is that the password confirmation is *nil*. This can never happen through the web, but it can at the console:

```
$ rails console
>> User.create(name: "Michael Hartl", email: "mhartl@example.com",
?>             password: "foobar", password_confirmation: nil)
```

When the confirmation is **`nil`**, Rails doesn't run the confirmation validation, which means that we can create users at the console without password confirmations. (Of course, right *now* we haven't added the validations yet, so the code above will work in any case.) To prevent this, we'll add a test to catch this case:

```
describe "when password confirmation is nil" do
  before { @user.password_confirmation = nil }
  it { should_not be_valid }
end
```

(This behavior strikes me as a minor bug in Rails, and perhaps it will be fixed in a future version, and in any case adding the validation does no harm.)

Putting everything together gives the (failing) tests in Listing 6.28. We'll get them to pass in Section 6.3.4.

Listing 6.28. Test for the password and password confirmation.

spec/models/user_spec.rb

```
require 'spec_helper'

describe User do

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

  subject { @user }

  it { should respond_to(:name) }
  it { should respond_to(:email) }
  it { should respond_to(:password_digest) }
  it { should respond_to(:password) }
  it { should respond_to(:password_confirmation) }

  it { should be_valid }
  .
  .
  .
  describe "when password is not present" do
    before { @user.password = @user.password_confirmation = " " }
    it { should_not be_valid }
  end

  describe "when password doesn't match confirmation" do
    before { @user.password_confirmation = "mismatch" }
    it { should_not be_valid }
  end

  describe "when password confirmation is nil" do
    before { @user.password_confirmation = nil }
    it { should_not be_valid }
  end
end
```

6.3.3 User authentication

The final piece of our password machinery is a method to retrieve users based on their email and passwords. This divides naturally into two parts: first, find a user by email address; second, authenticate the user with a given password.

The first step is simple; as we saw in Section 6.1.4, we can find a user with a given email address using the **`find_by_email`** method:

```
user = User.find_by_email(email)
```

The second step is then to use an **`authenticate`** method to verify that the user has the given password. In Chapter 8, we'll retrieve the current (signed-in) user using code something like this:

```
current_user = user.authenticate(password)
```

If the given password matches the user's password, it should return the user; otherwise, it should return **`false`**.

As usual, we can express the requirement for **`authenticate`** using RSpec. The resulting tests are more advanced than the others we've seen, so let's break them down into pieces; if you're new to RSpec, you might want to read this section a couple of times. We start by requiring a User object to respond to **`authenticate`**:

```
it { should respond_to(:authenticate) }
```

We then cover the two cases of password match and mismatch:

```
describe "return value of authenticate method" do
  before { @user.save }
  let(:found_user) { User.find_by_email(@user.email) }

  describe "with valid password" do
    it { should == found_user.authenticate(@user.password) }
  end

  describe "with invalid password" do
    let(:user_for_invalid_password) { found_user.authenticate("invalid") }

    it { should_not == user_for_invalid_password }
    specify { user_for_invalid_password.should be_false }
  end
end
```

The **`before`** block saves the user to the database so that it can be retrieved using **`find_by_email`**, which we accomplish using the **`let`** method:

```
let(:found_user) { User.find_by_email(@user.email) }
```

We've used **`let`** in a couple of exercises, but this is the first time we've seen it in the body of the tutorial. Box 6.3 covers **`let`** in more detail.

The two **`describe`** blocks cover the case where **`@user`** and **`found_user`** should be the same (password match) and different (password mismatch); they use the "double equals" **`==`** test for object equivalence (Section 4.3.1). Note that the tests in

```
describe "with invalid password" do
  let(:user_for_invalid_password) { found_user.authenticate("invalid") }

  it { should_not == user_for_invalid_password }
  specify { user_for_invalid_password.should be_false }
end
```

use `let` a second time, and also use the `specify` method. This is just a synonym for `it`, and can be used when writing `it` would sound unnatural. In this case, it sounds good to say "it [i.e., the user] should not equal wrong user", but it sounds strange to say "user: user with invalid password should be false"; saying "specify: user with invalid password should be false" sounds better.

Box 6.3. Using `let`

RSpec's `let` method provides a convenient way to create local variables inside tests. The syntax might look a little strange, but its effect is similar to variable assignment. The argument of `let` is a symbol, and it takes a block whose return value is assigned to a local variable with the symbol's name. In other words,

```
let(:found_user) { User.find_by_email(@user.email) }
```

creates a `found_user` variable whose value is equal to the result of `find_by_email`. We can then use this variable in any of the `before` or `it` blocks throughout the rest of the test. One advantage of `let` is that it *memoizes* its value, which means that it remembers the value from one invocation to the next. (Note that *memoize* is a technical term; in particular, it's *not* a misspelling of "memorize".) In the present case, because `let` memoizes the `found_user` variable, the `find_by_email` method will only be called once whenever the User model specs are run.

Finally, as a security precaution, we'll test for a length validation on passwords, requiring that they be at least six characters long:

```
describe "with a password that's too short" do
```

```
  before { @user.password = @user.password_confirmation = "a" * 5 }
  it { should be_invalid }
end
```

Putting together all the tests above gives Listing 6.29.

Listing 6.29. Test for the `authenticate` method.
`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do

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

  subject { @user }
  .
  .
  .
  it { should respond_to(:authenticate) }
  .
  .
  .
  describe "with a password that's too short" do
    before { @user.password = @user.password_confirmation = "a" * 5 }
    it { should be_invalid }
  end

  describe "return value of authenticate method" do
    before { @user.save }
    let(:found_user) { User.find_by_email(@user.email) }

    describe "with valid password" do
      it { should == found_user.authenticate(@user.password) }
    end

    describe "with invalid password" do
```

```
      let(:user_for_invalid_password) { found_user.authenticate("invalid") }

      it { should_not == user_for_invalid_password }
      specify { user_for_invalid_password.should be_false }
    end
  end
end
```

As noted in Box 6.3, **`let`** memoizes its value, so that the first nested **`describe`** block in Listing 6.29 invokes **`let`** to retrieve the user from the database using **`find_by_email`**, but the second **`describe`** block doesn't hit the database a second time.

6.3.4 User has secure password

In previous versions of Rails, adding a secure password was difficult and time-consuming, as seen in the Rails 3.0 version of the *Rails Tutorial*[16], which covers the creation of an authentication system from scratch. But web developers' understanding of how best to authenticate users has matured enough that it now comes bundled with the latest version of Rails. As a result, we'll complete the implementation of secure passwords (and get to a green test suite) using only a few lines of code.

First, we need to make the **`password`** and **`password_confirmation`** columns accessible (Section 6.1.2.2) so that we can instantiate new users with an initialization hash:

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

Following the model in Listing 6.6, we do this by adding the appropriate symbols to the list of accessible attributes:

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

Second, we need presence and length validations for the password, the latter of which uses the **`:minimum`** key in analogy with the **`:maximum`** key from Listing 6.15:

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

Next, we need to add **`password`** and **`password_confirmation`** attributes, require the presence of the password, require that they match, and add an **`authenticate`** method to compare an encrypted password to the **`password_digest`** to authenticate users. This is the only nontrivial step, and in the latest version of Rails all these features come for free with one method, **`has_secure_password`**:

```
has_secure_password
```

As long as there is a **`password_digest`** column in the database, adding this one method to our model gives us a secure way to create and authenticate new users. (If **`has_secure_password`** seems a bit too magical for your taste, I suggest taking a look at the source code for `secure_password.rb`, which is well-documented and quite readable. You'll see that, among other things, it automatically includes a validation for the **`password_digest`** attribute. In Chapter 7, we'll see that this is a mixed blessing.)

Finally, we need a presence validation for the password confirmation:

```
validates :password_confirmation, presence: true
```

Putting these three elements together yields the User model shown in Listing 6.30, which completes the implementation of secure passwords.

Listing 6.30. The complete implementation for secure passwords.
app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :name, :email, :password, :password_confirmation
  has_secure_password

  before_save { |user| user.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,
                    format:     { with: VALID_EMAIL_REGEX },
                    uniqueness: { case_sensitive: false }
  validates :password, presence: true, length: { minimum: 6 }
  validates :password_confirmation, presence: true
end
```

You should confirm at this point that the test suite passes:

```
$ bundle exec rspec spec/
```

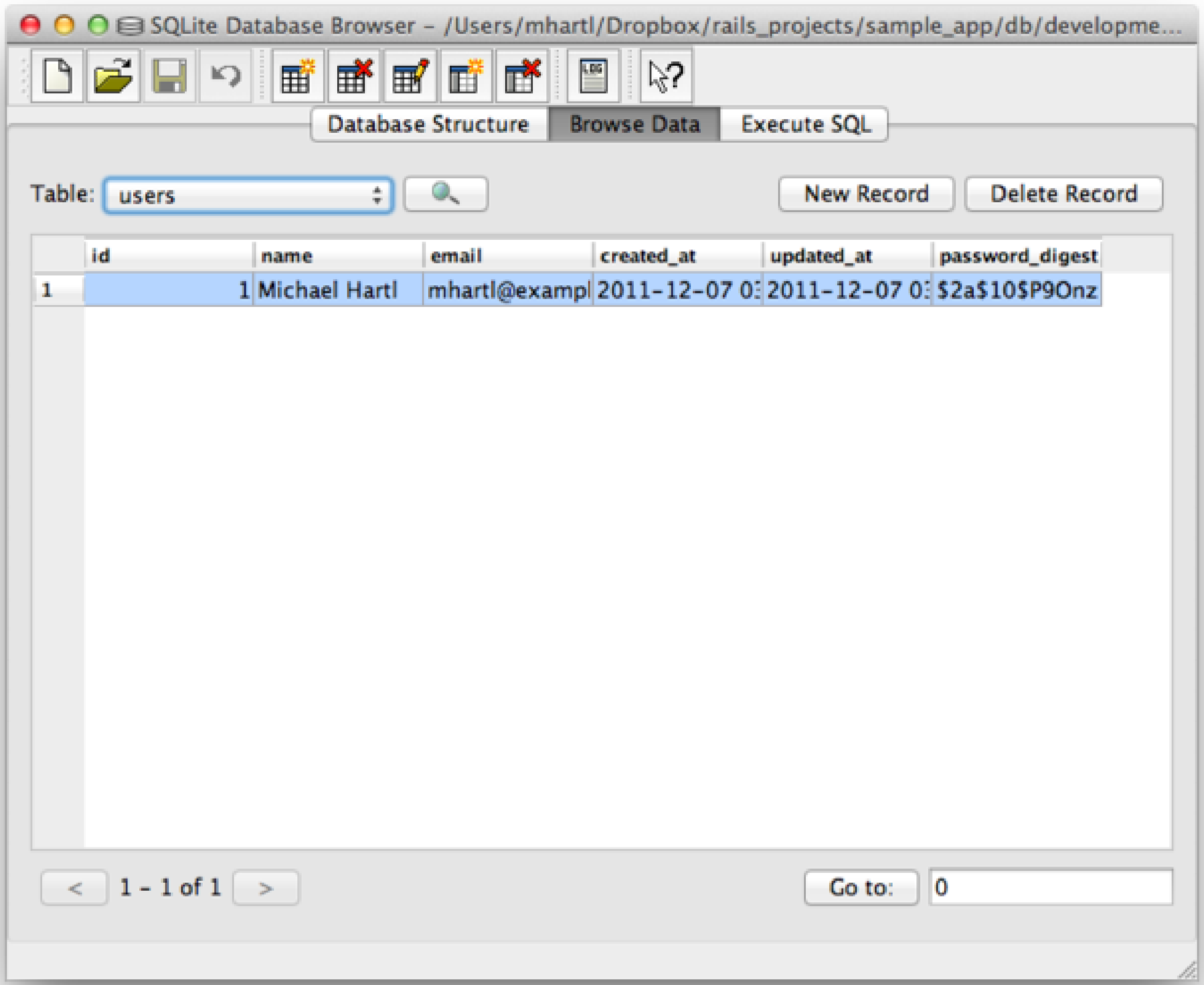
6.3.5 Creating a user

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. This also gives us a chance to make the work from the previous sections feel more concrete; merely getting the test suite to pass may seem anti-climactic, and it will be gratifying to see an actual user record in the development database.

Since we can't yet sign up through the web—that's the goal of Chapter 7—we'll use the Rails console to create a new user by hand. In contrast to Section 6.1.3, in this section we'll take care *not* to start in a sandbox, since this time the whole point is to save a record to the database:

```
$ rails console
>> User.create(name: "Michael Hartl", email: "mhartl@example.com",
?>             password: "foobar", password_confirmation: "foobar")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2011-12-07 03:38:14", updated_at: "2011-12-07 03:38:14",
password_digest: "$2a$10$P9OnzpdCON80yuMVk3jGr.LMA16VwOExJgjlw0G4f21y...">
```

To check that this worked, let's look at the row in the development database (**`db/development.sqlite3`**) using the SQLite Database Browser (Figure 6.6). Note that the columns correspond to the attributes of the data model defined in Figure 6.5.



	id	name	email	created_at	updated_at	password_digest
1	1	Michael Hartl	mhartl@exampl	2011-12-07 0:	2011-12-07 0:	$2a$10$P9Onz

Figure 6.6: A user row in the SQLite database **`db/development.sqlite3`**. (full size)

Returning to the console, we can see the effect of **`has_secure_password`** from Listing 6.30 by looking at the **`password_digest`** attribute:

```
>> user = User.find_by_email("mhartl@example.com")
>> user.password_digest
=> "$2a$10$P9OnzpdCON80yuMVk3jGr.LMA16VwOExJgjlw0G4f21yZIMSH/xoy"
```

This is the encrypted version of the password (**`"foobar"`**) used to initialize the user object. We can also verify that the **`authenticate`** command is working by first using an invalid password and then a valid one:

```
>> user.authenticate("invalid")
=> false
>> user.authenticate("foobar")
=> #<User id: 1, name: "Michael Hartl", email: "mhartl@example.com",
created_at: "2011-12-07 03:38:14", updated_at: "2011-12-07 03:38:14",
password_digest: "$2a$10$P9OnzpdCON80yuMVk3jGr.LMA16VwOExJgjlw0G4f21y...">
```

As required, **`authenticate`** returns **`false`** if the password is invalid and the user itself if the password is valid.

6.4 Conclusion

Starting from scratch, in this chapter we created a working User model with **`name`**, **`email`**, and various password attributes, together with validations enforcing several important constraints on their values. In addition, we can securely authenticate users using a given password. In previous versions of Rails, such a feat would have taken more than twice as much code, but because of the compact **`validates`** method and **`has_secure_password`**, we were able to build a complete User model in only ten source lines of code.

In the next chapter, 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 use the authentication machinery from Section 6.3 to let users sign 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:

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

Then merge back into the master branch:

```
$ git checkout master
$ git merge modeling-users
```

6.5 Exercises

1. Add a test for the email downcasing from Listing 6.23, as shown in Listing 6.31. By commenting out the `before_save` line, verify that Listing 6.31 tests the right thing.
2. By running the test suite, verify that the `before_save` callback can be written as shown in Listing 6.32.
3. Read through the Rails API entry for `ActiveRecord::Base` to get a sense of its capabilities.
4. Study the entry in the Rails API for the `validates` method to learn more about its capabilities and options.
5. Spend a couple of hours playing with Rubular.

Listing 6.31. A test for the email downcasing from Listing 6.23.

`spec/models/user_spec.rb`

```
require 'spec_helper'

describe User do
  .
  .
  .
  describe "email address with mixed case" do
    let(:mixed_case_email) { "Foo@ExAMPle.CoM" }

    it "should be saved as all lower-case" do
      @user.email = mixed_case_email
      @user.save
      @user.reload.email.should == mixed_case_email.downcase
    end
  end
  .
  .
  .
end
```

Listing 6.32. An alternate implementation of the `before_save` callback.

`app/models/user.rb`

```
class User < ActiveRecord::Base
  attr_accessible :name, :email, :password, :password_confirmation
  has_secure_password

  before_save { self.email.downcase! }
  .
  .
  .
end
```

« Chapter 5 Filling in the layout   Chapter 7 Sign up »

1. The name comes from the “active record pattern”, identified and named in *Patterns of Enterprise Application Architecture* by Martin Fowler. ↑

2. Pronounced “ess-cue-ell”, though the alternate pronunciation “sequel” is also common. ↑

3. By using an email address as the username, we open the theoretical possibility of communicating with our users at a future date. ↑

4. Don’t worry about exactly how the `t` object manages to do this; the beauty of *abstraction layers* is that we don’t have to know. We can just trust the `t` object to do its job. ↑

5. Officially pronounced “ess-cue-ell-ite”, although the (mis)pronunciation “sequel-ite” is also common. ↑

6. In case you’re curious about `"2011-12-05 00:57:46"`, I’m not writing this after midnight; the timestamps are recorded in Coordinated Universal Time (UTC), which for most practical purposes is the same as Greenwich Mean Time. From the NIST Time and Frequency FAQ: **Q:** Why is UTC used as the acronym for Coordinated Universal Time instead of CUT? **A:** In 1970 the Coordinated Universal Time system was devised by an international advisory group of technical experts within the International Telecommunication Union (ITU). The ITU felt it was best to designate a single abbreviation for use in all languages in order to minimize confusion. Since unanimous agreement could not be achieved on using either the English word order, CUT, or the French word order, TUC, the acronym UTC was chosen as a compromise. ↑

7. Note the value of `user.updated_at`. Told you the timestamp was in UTC. ↑

8. Exceptions and exception handling are somewhat advanced Ruby subjects, and we won’t need them much in this book. They are important, though, and I suggest learning about them using one of the Ruby books recommended in Section 1.1.1. ↑

9. I’ll omit the output of console commands when they are not particularly instructive—for example,

the results of `User.new`. ↑

10. 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. ↑

11. 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. ↑

12. 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. If you don't have an email address that contains only letters, numbers, underscores, and dots, then I recommend getting one. N.B. The regex in Listing 6.17 allows plus signs, too, because Gmail (and possibly other email services) does something useful with them: to filter email from example.com, you can use `username+example@gmail.com`, which will go to the Gmail address `username@gmail.com`, allowing you to filter on the string `example`. ↑

13. As noted briefly in the introduction to this section, there is a dedicated test database, `db/test.sqlite3`, for this purpose. ↑

14. 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. ↑

15. Direct experimentation with SQLite on my system and PostgreSQL on Heroku show that this step is, in fact, necessary. ↑

16. http://railstutorial.org/book?version=3.0 ↑