Nevertheless, I believe it is a mistake to use a pre-built system like Devise in a tutorial like this one. Off-the-shelf systems can be "black boxes" with potentially mysterious innards, and the complicated data models used by such systems would be utterly overwhelming for beginners (or even for experienced developers not familiar with data modeling). For learning purposes, it's essential to introduce the subject more gradually.

Happily, Rails makes it possible to take such a gradual approach while still developing an industrial-strength login and authentication system suitable for production applications. This way, even 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 to meet your particular needs.

6.1 User model

Although the ultimate goal of the next three chapters is to make a signup page for our site (as 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.

In Rails, the default data structure for a data model is called, naturally enough, a *model* (the M in MVC from Section 1.2.3). 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*

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

[2]Officially pronounced "ess-cue-ell", though the alternate pronunciation "sequel" is also common. You can differentiate an individual author's preference by the choice of indefinite article: those who write "a SQL database" prefer "sequel", whereas those who write "an SQL database" prefer "ess-cue-ell". As you'll soon see, I prefer the latter.

Figure 6.1: A mockup of the user signup page.

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 database. 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.

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 -b modeling-users
```

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 4.17, 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

[3] By using an email address as the username, we open the possibility of communicating with our users at a future date (Chapter 11 and Chapter 12).

users		
id	name	email
1	Michael Hartl	mhartl@example.com
2	Sterling Archer	archer@example.gov
3	Lana Kane	lana@example.gov
4	Mallory Archer	boss@example.gov

Figure 6.2: A diagram of sample data in a `users` table.

users	
id	integer
name	string
email	string

Figure 6.3: A sketch of the User data model.

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). An example of such a table appears in Figure 6.2, corresponding to the data model shown in Figure 6.3. (Figure 6.3 is just a sketch; the full data model appears in Figure 6.4.) By naming the columns `name` and `email`, we'll let Active Record figure out the User object attributes for us.

You may recall from Listing 5.38 that we created a Users controller (along with a `new` action) using the command

```
$ rails generate controller Users new
```

The analogous command for making a model is **`generate model`**, which we can use to generate a User model with **`name`** and **`email`** attributes, as shown in Listing 6.1.

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    test_unit
      create      test/models/user_test.rb
      create      test/fixtures/users.yml
```

(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 which types those attributes should be (in this case, **`string`**). Compare this with including the action names in Listing 3.7 and Listing 5.38.

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 starting in Section 6.2.5 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[6.0]
  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 the migration in Listing 6.2 is shown in Figure 6.4. (Note the addition of the magic columns, which weren't present in the sketch shown in Figure 6.3.)

We can run the migration, known as "migrating up", using the **db:migrate** command as follows:

[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.

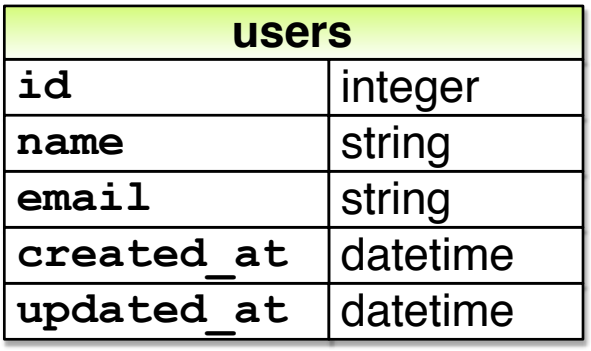

Figure 6.4: The User data model produced by Listing 6.2.

```
$ rails db:migrate
```

(You may recall that we ran this command in a similar context 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 by opening **development.sqlite3** with DB Browser for SQLite. (If you're using the cloud IDE, you should first download the database file to the local disk, as shown in Figure 6.5.) The result appears in Figure 6.6; compare with the diagram in Figure 6.4. You might note that there's one column in Figure 6.6 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.

Exercises

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

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

[5] Officially pronounced "ess-cue-ell-ite", although the (mis)pronunciation "sequel-ite" is also common.

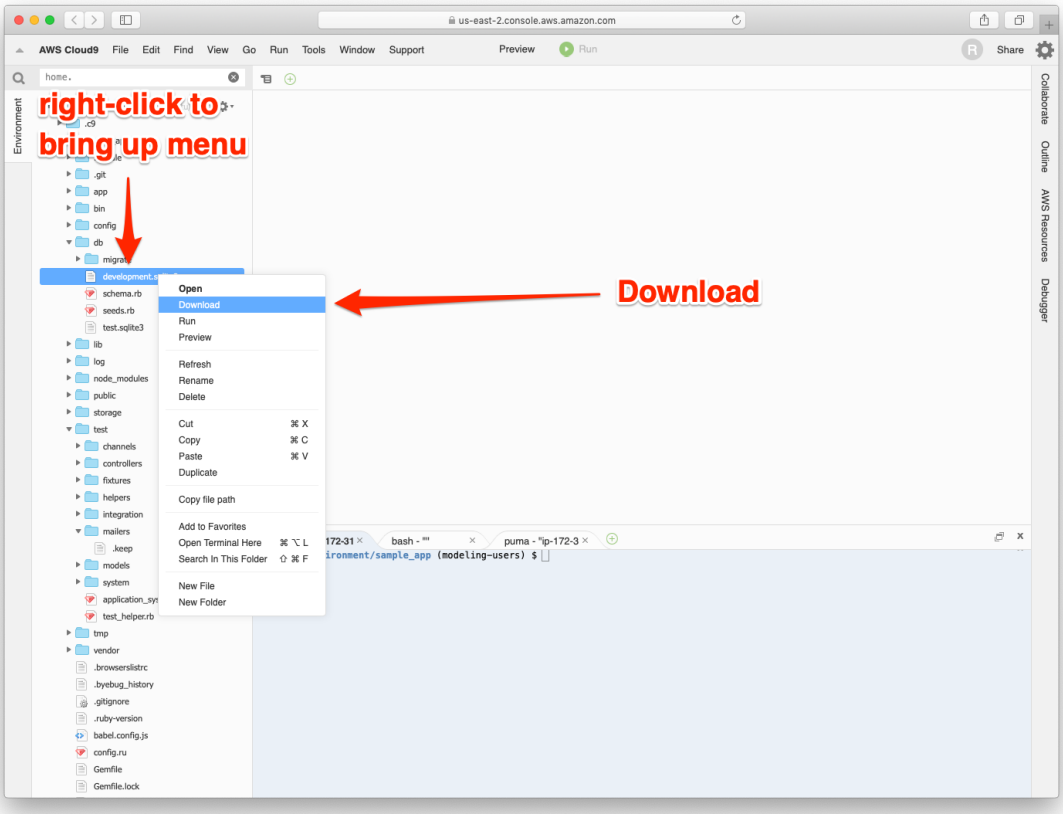

Figure 6.5: Downloading a file from the cloud IDE.

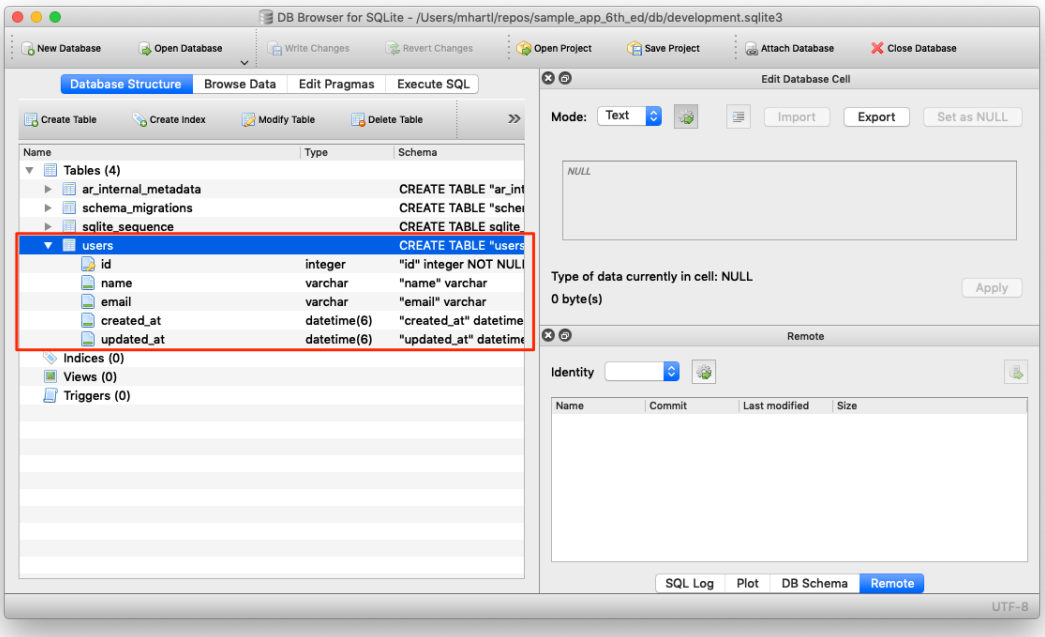

Figure 6.6: DB Browser with our new `users` table.

1. Rails uses a file called **schema.rb** in the **db/** directory to keep track of the structure of the database (called the *schema*, hence the filename). Examine your local copy of **db/schema.rb** and compare its contents to the migration code in Listing 6.2.

2. Most migrations (including all the ones in this tutorial) are *reversible*, which means we can "migrate down" and undo them with a single command, called **db:rollback**:

```
$ rails db:rollback
```

   After running this command, examine **db/schema.rb** to confirm that the rollback was successful. (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.

3. Re-run the migration by executing **rails db:migrate** again. Confirm that the contents of **db/schema.rb** have been restored.

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 6.6 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).

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

```
class User < ApplicationRecord
end
```

Recall from Section 4.4.2 that the syntax **class User < ApplicationRecord** means that the **User** class *inherits* from the **ApplicationRecord** class, which in turn inherits from **ActiveRecord::Base** (Figure 2.19), so that the User model automatically has all the functionality of the **ActiveRecord::Base** class. Of course, this knowledge doesn't do us any good unless we know what **ActiveRecord::Base** contains, so let's get started with some concrete examples.

Exercises

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

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

1. In a Rails console, use the technique from Section 4.4.4 to confirm that **User.new** is of class **User** and inherits from **ApplicationRecord**.
2. Confirm that **ApplicationRecord** inherits from **ActiveRecord::Base**.

6.1.3 Creating user objects

As in Chapter 4, our tool of choice for exploring data models 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 4.17. 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.

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

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

The notion of *validity* is important for understanding Active Record model objects. We'll explore this subject in more depth in Section 6.2, but for now it's worth noting that our initial **`user`** object is valid, which we can verify by calling the boolean **`valid?`** method on it:

```
>> user.valid?
true
```

So far, we haven't touched the database: `User.new` only creates an object *in memory*, while `user.valid?` merely checks to see if the object is valid. In order to save the User object to the database, we need to call the `save` method on the `user` variable:

```
>> user.save
   (0.1ms)  SAVEPOINT active_record_1
  SQL (0.8ms)  INSERT INTO "users" ("name", "email", "created_at",
  "updated_at") VALUES (?, ?, ?, ?)  [["name", "Michael Hartl"],
  ["email", "michael@example.com"], ["created_at", "2019-08-22 01:51:03.453035"],
  ["updated_at", "2019-08-22 01:51:03.453035"]]
   (0.1ms)  RELEASE SAVEPOINT active_record_1
=> true
```

The `save` method returns `true` if it succeeds and `false` otherwise. (Currently, all saves should succeed because there are as yet no validations; we'll see cases in Section 6.2 when some will fail.) For reference, the Rails console also shows the SQL command corresponding to `user.save` (namely, `INSERT INTO "users"`...). We'll hardly ever need raw SQL in this book,[6] and I'll omit discussion of the SQL commands from now on, but you can learn a lot by reading the SQL corresponding to Active Record commands.

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: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
```

We see that the `id` has been assigned a value of `1`, while the magic columns have

[6] The only exception is in Section 14.3.3.

been assigned the current time and date.[7] 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:

```
>> user.name
=> "Michael Hartl"
>> user.email
=> "michael@example.com"
>> user.updated_at
=> Thu, 22 Aug 2019 01:51:03 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:
"2019-08-22 01:53:22", updated_at: "2019-08-22 01:53:22">
>> foo = User.create(name: "Foo", email: "foo@bar.com")
#<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2019-08-22
01:54:03", updated_at: "2019-08-22 01:54:03">
```

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
   (0.1ms)  SAVEPOINT active_record_1
  SQL (0.2ms)  DELETE FROM "users" WHERE "users"."id" = ?  [["id", 3]]
```

[7] The timestamps are recorded in Coordinated Universal Time (UTC), which for most practical purposes is the same as Greenwich Mean Time. But why call it UTC? 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.

```
   (0.1ms)  RELEASE SAVEPOINT active_record_1
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2019-08-22
01:54:03", updated_at: "2019-08-22 01:54:03">
```

Like **`create`**, **`destroy`** returns the object in question, though I can't recall ever having used the return value of **`destroy`**. In addition, the destroyed object still exists in memory:

```
>> foo
=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2019-08-22
01:54:03", updated_at: "2019-08-22 01:54:03">
```

So 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? To answer these questions, we need to learn how to use Active Record to find user objects.

Exercises

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

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

1. Confirm that **`user.name`** and **`user.email`** are of class **`String`**.
2. Of what class are the **`created_at`** and **`updated_at`** attributes?

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: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
```

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, leading **`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: "michael@example.com")
=> #<User id: 1, name: "Michael Hartl", email: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
```

Since we will be using email addresses as usernames, this sort of **`find`** will be useful when we learn how to let users log in to our site (Chapter 7). If you're worried that **`find_by`** 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: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
```

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

[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 14.4.1.

```
>> User.all
=> #<ActiveRecord::Relation [#<User id: 1, name: "Michael Hartl", email:
"michael@example.com", created_at: "2019-08-22 01:51:03", updated_at:
"2019-08-22 01:51:03">, #<User id: 2, name: "A Nother", email:
"another@example.org", created_at: "2019-08-22 01:53:22", updated_at:
"2019-08-22 01:53:22">]>
```

As you can see from the console output, **`User.all`** returns all the users in the database as an object of class **`ActiveRecord::Relation`**, which is effectively an array (Section 4.3.1).

Exercises

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

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

1. Find the user by **`name`**. Confirm that **`find_by_name`** works as well. (You will often encounter this older style of **`find_by`** in legacy Rails applications.)

2. For most practical purposes, **`User.all`** acts like an array, but confirm that in fact it's of class **`User::ActiveRecord_Relation`**.

3. Confirm that you can find the length of **`User.all`** by passing it the **`length`** method (Section 4.2.2). Ruby's ability to manipulate objects based on how they act rather than on their formal class type is called *duck typing*, based on the aphorism that "If it looks like a duck, and it quacks like a duck, it's probably a duck."

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: "michael@example.com",
created_at: "2019-08-22 01:51:03", updated_at: "2019-08-22 01:51:03">
>> 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 by running **`user.save`**, the magic columns differ, as promised in Section 6.1.3:

```
>> user.created_at
=> Thu, 22 Aug 2019 01:51:03 UTC +00:00
>> user.updated_at
=> Thu, 22 Aug 2019 01:58:08 UTC +00:00
```

The second main way to update multiple attributes is to use **`update`**:[9]

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

[9] Formerly **`update_attributes`**.

The **update** method accepts a hash of attributes, and on success performs both the update and the save in one step (returning **true** to indicate that the save went through). Note that if any of the validations fail, such as when a password is required to save a record (as implemented in Section 6.3), the call to **update** will fail. If we need to update only a single attribute, using the singular **update_attribute** bypasses this restriction by skipping the validations:

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

Exercises

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

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

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

6.2 User validations

The User model we created in Section 6.1 now has working **name** and **email** attributes, but they are completely generic: any string (including an empty one) is currently valid in either case. And yet, names and email addresses are more specific than this. For example, **name** should be non-blank, and **email** should match the specific format characteristic of email addresses. Moreover, since