

In this chapter, you'll see:

- Creating a new application
- Starting the server
- Accessing the server from a browser
- Producing dynamic content
- Adding hypertext links
- Passing data from the controller to the view
- Basic error recovery and debugging

## CHAPTER 2

# Instant Gratification

---

Let's write a simple application to verify that we have Rails snugly installed on our machines. Along the way, you'll get a peek at the way Rails applications work.

## Creating a New Application

When you install the Rails framework, you also get a new command-line tool, `rails`, that's used to construct each new Rails application you write.

Why do we need a tool to do this? Why can't we just hack away in our favorite editor and create the source for our application from scratch? Well, we could just hack. After all, a Rails application is just Ruby source code. But Rails also does a lot of magic behind the curtain to get our applications to work with a minimum of explicit configuration. To get this magic to work, Rails needs to find all the various components of your application. As you'll see later (in [Where Things Go, on page 291](#)), this means we need to create a specific directory structure, slotting the code we write into the appropriate places. The `rails` command creates this directory structure for us and populates it with some standard Rails code.

To create your first Rails application, pop open a shell window and navigate to a place in your filesystem where you want to create your application's directory structure. In our example, we'll be creating our projects in a directory called `work`. In that directory, use the `rails` command to create an application called `demo`. Be slightly careful here—if you have an existing directory called `demo`, you'll be asked if you want to overwrite any existing files. (Note: if you want to specify which Rails version to use, as described in [Choosing a Rails Version, on page 13](#), now is the time to do so.)

```

rubys> cd work
work> rails new demo
create
create  README.md
create  Rakefile
create  .ruby-version
      :      :      :
remove  config/initializers/cors.rb
remove  config/initializers/new_framework_defaults_7_0.rb
      run  bundle install
Fetching gem metadata from https://rubygems.org/.....
      :      :      :
append  config/importmap.rb
work>

```

The command has created a directory named `demo`. Pop down into that directory and list its contents (using `ls` on a Unix box or using `dir` on Windows). You should see a bunch of files and subdirectories:

```

work> cd demo
demo> ls -p
Gemfile          app/             db/             storage/
Gemfile.lock     bin/            lib/           test/
README.md        config/         log/           tmp/
Rakefile         config.ru       public/        vendor/

```

All these directories (and the files they contain) can be intimidating to start with, but you can ignore most of them for now. In this chapter, we'll only use two of them directly: the `bin` directory, where we'll find the Rails executables, and the `app` directory, where we'll write our application.

Examine your installation using the following command:

```
demo> bin/rails about
```

Windows users need to prefix the command with `ruby` and use a backslash:

```
demo> ruby bin\rails about
```

If you get a Rails version other than 7.0.4, reread [Choosing a Rails Version, on page 13](#).

This command also detects common installation errors. For example, if it can't find a JavaScript runtime, it provides you with a link to available runtimes.

As you can see from the `bin/` prefix, this is running the `rails` command from the `bin` directory. This command is a wrapper, or *binstub*, for the Rails executable. It serves two purposes: it ensures that you're running with the correct version of every dependency, and it speeds up the startup times of Rails commands by preloading your application.

If you see a bunch of messages concerning already initialized constants or a possible conflict with an extension, consider deleting the demo directory, creating a separate RVM gemset,<sup>1</sup> and starting over. If that doesn't work, use `bundle exec`<sup>2</sup> to run rails commands:

```
demo> bundle exec rails about
```

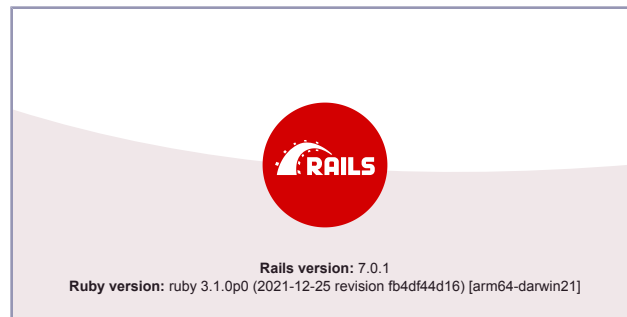
Once you get `bin/rails about` working, you have everything you need to start a stand-alone web server that can run our newly created Rails application. So without further ado, let's start our demo application:

```
demo> bin/rails server
=> Booting Puma
=> Rails 7.0.4 application starting in development
=> Run `bin/rails server --help` for more startup options
Puma starting in single mode...
* Puma version: 5.5.2 (ruby 3.1.9-p0) ("Zawgyi")
* Min threads: 5
* Max threads: 5
* Environment: development
* PID: 28763
* Listening on http://127.0.0.1:3000
* Listening on http://[::1]:3000
Use Ctrl-C to stop
```

Note, if you're using a virtual machine, you need to run Rails like so:

```
demo> bin/rails server -b 0.0.0.0
```

As the second line of the startup tracing indicates, we started a web server on port 3000. The `localhost` part of the address means that the Puma web server will only accept requests that originate from your machine. We can access the application by pointing a browser at the URL <http://localhost:3000>. The result is shown in the following screenshot.



1. <https://rvm.io/gemsets/basics/>
2. [http://gembundler.com/v1.3/bundle\\_exec.html](http://gembundler.com/v1.3/bundle_exec.html)

If you look at the window where you started the server, you can see tracing showing that you started the application. We're going to leave the server running in this console window. Later, as we write application code and run it via our browser, we'll be able to use this console window to trace the incoming requests. When the time comes to shut down your application, you can press `Ctrl-C` in this window to stop the server. (Don't do that yet—we'll be using this particular application in a minute.)

If you want to enable this server to be accessed by other machines on your network, either you'll need to list each server you want to have access separately or you can enable everybody to access your development server by adding the following to `config/environments/development.rb`:

```
config.hosts.clear
```

You'll also need to specify `0.0.0.0` as the host to bind to the following code:

```
demo> bin/rails server -b 0.0.0.0
```

At this point, we have a new application running, but it has none of our code in it. Let's rectify this situation.

## Hello, Rails!

We can't help it—we just have to write a Hello, World! program to try a new system. Let's start by creating a simple application that sends our cheery greeting to a browser. After we get that working, we'll embellish it with the current time and links.

As you'll explore further in [Chapter 3, The Architecture of Rails Applications, on page 37](#), Rails is a model-view-controller (MVC) framework. Rails accepts incoming requests from a browser, decodes the request to find a controller, and calls an action method in that controller. The controller then invokes a particular view to display the results to the user. The good news is that Rails takes care of most of the internal plumbing that links all these actions. To write our Hello, World! application, we need code for a controller and a view, and we need a route to connect the two. We don't need code for a model, because we're not dealing with any data. Let's start with the controller.

In the same way that we used the `rails` command to create a new Rails application, we can also use a generator script to create a new controller for our project. This command is `rails generate`. So to create a controller called `say`, we make sure we're in the `demo` directory and run the command, passing in the name of the controller we want to create and the names of the actions we intend for this controller to support:

```
demo> bin/rails generate controller Say hello goodbye
       create  app/controllers/say_controller.rb
       route   get 'say/hello'
get 'say/goodbye'
       invoke  erb
       create  app/views/say
       create  app/views/say/hello.html.erb
       create  app/views/say/goodbye.html.erb
       invoke  test_unit
       create  test/controllers/say_controller_test.rb
       invoke  helper
       create  app/helpers/say_helper.rb
       invoke  test_unit
       invoke  assets
       invoke  scss
       create  app/assets/stylesheets/say.scss
```

The rails generate command logs the files and directories it examines, noting when it adds new Ruby scripts or directories to our application. For now, we're interested in one of these scripts and (in a minute) the .html.erb files.

The first source file we'll be looking at is the controller. You can find it in the app/controllers/say\_controller.rb file.

Let's take a look at it:

```
rails7/demo1/app/controllers/say_controller.rb
class SayController < ApplicationController
  >   def hello
  >   end

   def goodbye
   end
end
```

Pretty minimal, eh? SayController is a class that inherits from ApplicationController, so it automatically gets all the default controller behavior. What does this code have to do? For now, it does nothing—we simply have empty action methods named hello() and goodbye(). To understand why these methods are named this way, you need to look at the way Rails handles requests.

## Rails and Request URLs

Like any other web application, a Rails application appears to its users to be associated with a URL. When you point your browser at that URL, you're talking to the application code, which generates a response to you.

Let's try it now. Navigate to the URL <http://localhost:3000/say/hello> in a browser. You'll see something that looks like the following screenshot.

## Say#hello

Find me in `app/views/say/hello.html.erb`

### Our First Action

At this point, we can see not only that we've connected the URL to our controller but also that Rails is pointing the way to our next step—namely, to tell Rails what to display. That's where views come in. Remember when we ran the script to create the new controller? That command added several files and a new directory to our application. That directory contains the template files for the controller's views. In our case, we created a controller named `say`, so the views will be in the `app/views/say` directory.

By default, Rails looks for templates in a file with the same name as the action it's handling. In our case, that means we need to edit a file called `hello.html.erb` in the `app/views/say` directory. (Why `.html.erb`? We'll explain in a minute.) For now, let's put some basic HTML in there:

```
rails7/demo1/app/views/say/hello.html.erb
<h1>Hello from Rails!</h1>
```

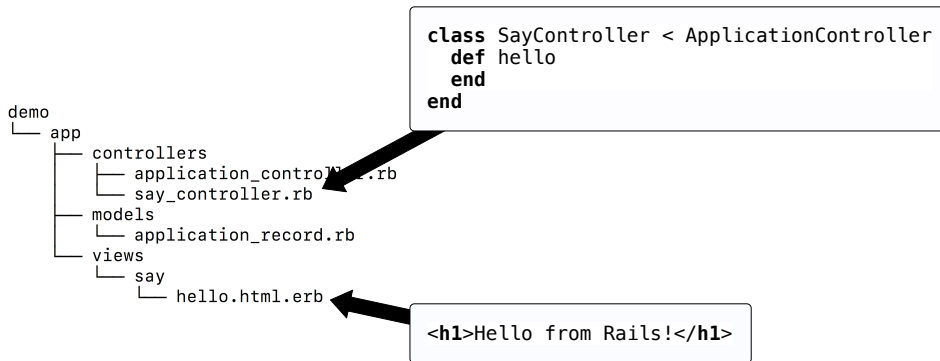
Save the `hello.html.erb` file, and refresh your browser window. You should see it display our friendly greeting, as in the following screenshot.

## Hello from Rails!

In total, we've looked at two files in our Rails application tree. We looked at the controller, and we modified a template to display a page in the browser. These files live in standard locations in the Rails hierarchy: controllers go into `app/controllers`, and views go into subdirectories of `app/views`. You can see this structure in the diagram [shown on page 27](#).

### Making It Dynamic

So far, our Rails application is boring—it just displays a static page. To make it more dynamic, let's have it show the current time each time it displays the page.



To do this, we need to change the template file in the view—it now needs to include the time as a string. That raises two questions. First, how do we add dynamic content to a template? Second, where do we get the time from?

### Dynamic Content

You can create dynamic templates in Rails in many ways. The most common way, which we'll use here, is to embed Ruby code in the template. That's why the template file is named `hello.html.erb`; the `.html.erb` suffix tells Rails to expand the content in the file using a system called ERB.

ERB is a filter, installed as part of the Rails installation, that takes an `.erb` file and outputs a transformed version. The output file is often HTML in Rails, but it can be anything. Normal content is passed through without being changed. However, content between `<%=` and `%>` is interpreted as Ruby code and executed. The result of that execution is converted into a string, and that value is substituted in the file in place of the `<%=...%>` sequence. For example, change `hello.html.erb` to display the current time:

```
rails7/demo2/app/views/say/hello.html.erb
```

```
<h1>Hello from Rails!</h1>
```

```

➤ <p>
➤   It is now <%= Time.now %>
➤ </p>

```

When we refresh our browser window, we see the time displayed using Ruby's standard format, as shown in the following screenshot.

## Hello from Rails!

It is now 2022-01-06 22:41:40 -0500

Notice that the time displayed updates each time the browser window is refreshed. It looks as if we're really generating dynamic content.

### Making Development Easier

You might have noticed something about the development we've been doing so far. As we've been adding code to our application, we haven't had to restart the running application. It's been happily chugging away in the background. And yet each change we make is available whenever we access the application through a browser. What gives?

It turns out that the Rails dispatcher is pretty clever. In development mode (as opposed to testing or production), it automatically reloads application source files when a new request comes along. That way, when we edit our application, the dispatcher makes sure it's running the most recent changes. This is great for development.

However, this flexibility comes at a cost: it causes a short pause after you enter a URL before the application responds. That's caused by the dispatcher reloading stuff. For development it's a price worth paying, but in production it would be unacceptable. For this reason, this feature is disabled for production deployment.

### Adding the Time

Our original problem was to display the time to users of our application. We now know how to make our application display dynamic data. The second issue we have to address is working out where to get the time from.

We've shown that the approach of embedding a call to Ruby's `Time.now()` method in our `hello.html.erb` template works. Each time they access this page, users will see the current time substituted into the body of the response. And for our trivial application, that might be good enough. In general, though, we probably want to do something slightly different. We'll move the determination of the time to be displayed into the controller and leave the view with the job of displaying it. We'll change our action method in the controller to set the time value into an instance variable called `@time`:

```
rails7/demo3/app/controllers/say_controller.rb
class SayController < ApplicationController
  def hello
    @time = Time.now
  end

  def goodbye
  end
end
```



In the `.html.erb` template, we'll use this instance variable to substitute the time into the output:

```
rails7/demo3/app/views/say/hello.html.erb
```

```
<h1>Hello from Rails!</h1>
<p>
  It is now <%= @time %>
</p>
```

When we refresh our browser window, we again see the current time, showing that the communication between the controller and the view was successful.

Why did we go to the extra trouble of setting the time to be displayed in the controller and then using it in the view? Good question. In this application, it doesn't make much difference, but by putting the logic in the controller instead, we buy ourselves some benefits. For example, we may want to extend our application in the future to support users in many countries. In that case, we'd want to localize the display of the time, choosing a time appropriate to the user's time zone. That would require a fair amount of application-level code, and it would probably not be appropriate to embed it at the view level. By setting the time to display in the controller, we make our application more flexible: we can change the time zone in the controller without having to update any view that uses that time object. The time is *data*, and it should be supplied to the view by the controller. We'll see a lot more of this when we introduce models into the equation.

## The Story So Far

Let's briefly review how our current application works.

1. The user navigates to our application. In our case, we do that using a local URL such as <http://localhost:3000/say/hello>.
2. Rails then matches the route pattern, which it previously split into two parts and analyzed. The `say` part is taken to be the name of a controller, so Rails creates a new instance of the Ruby `SayController` class (which it finds in `app/controllers/say_controller.rb`).
3. The next part of the pattern, `hello`, identifies an action. Rails invokes a method of that name in the controller. This action method creates a new `Time` object holding the current time and tucks it away in the `@time` instance variable.
4. Rails looks for a template to display the result. It searches the `app/views` directory for a subdirectory with the same name as the controller (`say`) and in that subdirectory for a file named after the action (`hello.html.erb`).

5. Rails processes this file through the ERB templating system, executing any embedded Ruby and substituting in values set up by the controller.
6. The result is returned to the browser, and Rails finishes processing this request.

This isn't the whole story. Rails gives you lots of opportunities to override this basic workflow (and we'll be taking advantage of them shortly). As it stands, our story illustrates convention over configuration, one of the fundamental parts of the philosophy of Rails. Rails applications are typically written using little or no external configuration. That's because Rails provides convenient defaults, and because you apply certain conventions to how a URL is constructed, which file a controller definition is placed in, or which class name and method names are used. Things knit themselves together in a natural way.

## Linking Pages Together

It's a rare web application that has just one page. Let's see how we can add another stunning example of web design to our Hello, World! application.

Normally, each page in our application will correspond to a separate view. While we'll also use a new action method to handle the new page, we'll use the same controller for both actions. This needn't be the case, but we have no compelling reason to use a new controller right now.

We already defined a goodbye action for this controller, so all that remains is to update the scaffolding that was generated in the `app/views/say` directory. This time the file we'll be updating is called `goodbye.html.erb` because by default templates are named after their associated actions:

```
rails7/demo4/app/views/say/goodbye.html.erb
```

```
<h1>Goodbye!</h1>
<p>
  It was nice having you here.
</p>
```

Fire up your trusty browser again, but this time point to our new view using the URL <http://localhost:3000/say/goodbye>. You should see something like this screenshot.

**Goodbye!**

It was nice having you here.

Now we need to link the two screens. We'll put a link on the hello screen that takes us to the goodbye screen, and vice versa. In a real application, we might want to make these proper buttons, but for now we'll use hyperlinks.

We already know that Rails uses a convention to parse the URL into a target controller and an action within that controller. So a simple approach would be to adopt this URL convention for our links.

The `hello.html.erb` file would contain the following:

```
...
<p>
  Say <a href="/say/goodbye">Goodbye</a>!
</p>
...
```

And the `goodbye.html.erb` file would point the other way:

```
...
<p>
  Say <a href="/say/hello">Hello</a>!
</p>
...
```

This approach would certainly work, but it's a bit fragile. If we were to move our application to a different place on the web server, the URLs would no longer be valid. It also encodes assumptions about the Rails URL format into our code; it's possible a future version of Rails could change that format.

Fortunately, these aren't risks we have to take. Rails comes with a bunch of *helper methods* that can be used in view templates. Here, we'll use the `link_to()` helper method, which creates a hyperlink to an action. (The `link_to()` method can do a lot more than this, but let's take it gently for now.) Using `link_to()`, `hello.html.erb` becomes the following:

```
rails7/demo5/app/views/say/hello.html.erb
<h1>Hello from Rails!</h1>
<p>
  It is now <%= @time %>
</p>
> <p>
>   Time to say
>   <%= link_to "Goodbye", say_goodbye_path %>!
> </p>
```

A `link_to()` call is within an ERB `<%=...%>` sequence. This creates a link to a URL that will invoke the `goodbye()` action. The first parameter in the call to `link_to()` is the text to be displayed in the hyperlink, and the next parameter tells Rails to generate the link to the `goodbye()` action.

Let's stop for a minute to consider how we generated the link. We wrote this:

```
link_to "Goodbye", say_goodbye_path
```

First, `link_to()` is a method call. (In Rails, we call methods that make it easier to write templates *helpers*.) If you come from a language such as Java, you might be surprised that Ruby doesn't insist on parentheses around method parameters. You can always add them if you like.

`say_goodbye_path` is a precomputed value that Rails makes available to application views. It evaluates to the `/say/goodbye` path. Over time, you'll see that Rails provides the ability to name all the routes that you use in your application.

Let's get back to the application. If we point our browser at our hello page, it now contains the link to the goodbye page, as shown in the following screenshot.

## Hello from Rails!

It is now 2022-01-06 22:41:43 -0500

Time to say [Goodbye!](#)

We can make the corresponding change in `goodbye.html.erb`, linking it back to the initial hello page:

```
rails7/demo5/app/views/say/goodbye.html.erb
<h1>Goodbye!</h1>
<p>
  It was nice having you here.
</p>
> <p>
>   Say <%= link_to "Hello", say_hello_path %> again.
> </p>
```

So far, we've just done things that should work, and—unsurprisingly—they've worked. But the true test of the developer friendliness of a framework is how it responds when things go wrong. As we've not invested much time into this code yet, now is a perfect time to try to break things.

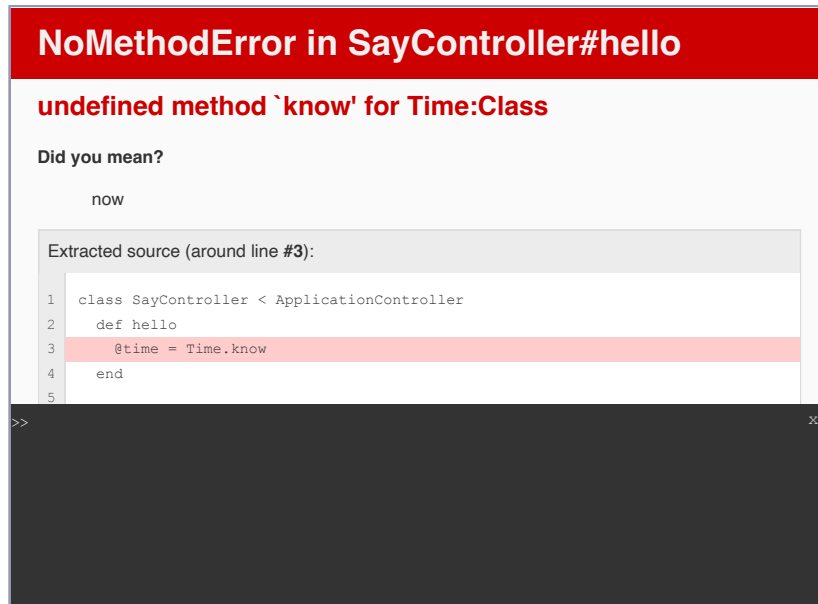
## When Things Go Wrong

Let's start by introducing a typo in the source code—one that perhaps is introduced by a misfiring autocorrect function in your favorite editor:

```
rails7/demo5/app/controllers/say_controller.rb
class SayController < ApplicationController
  def hello
    @time = Time.know
  end

  def goodbye
  end
end
```

Refresh the following page in your browser: <http://localhost:3000/say/hello>. You should see something like the following screenshot.



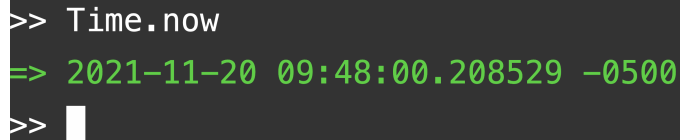
For security reasons, the web console is configured to only be shown when accessed from the same machine that the web server is running on. If you're running on a different machine, you'll need to adjust the configuration to see this. For example, to enable the web console to be seen by all, add the following to `config/environments/development.rb` and restart your server:

```
config.web_console.whitelisted_ips = %w( 0.0.0.0/0 ::/0 )
```

What you see is that Ruby tells you about the error ("undefined method 'know'"), and Rails shows you the extracted source where the code can be found (Rails.root), the stack traceback, and request parameters (at the moment, None). It also provides the ability to toggle the display of session and environment dumps.

You'll even see a suggestion: "Did you mean? now." What a nice touch.

At the bottom of the window you see an area consisting of white text on a black background, looking much like a command-line prompt. This is the Rails *web console*. You can use it to try out suggestions and evaluate expressions. Let's try it out, as shown in the following screenshot.



```
>> Time.now
=> 2021-11-20 09:48:00.208529 -0500
>>
```

All in all, helpful stuff.

We've broken the code. Now, let's break the other thing we've used so far: the URL. Visit the following page in your browser: <http://localhost:3000/say/hullo>. You should see something like the [screenshot on page 35](#).

This is similar to what we saw before, but in place of source code we see a list of possible routes, how they can be accessed, and the controller action they're associated with. We'll explain this later in detail, but for now look at the Path Match input field. If you enter a partial URL in there, you can see a list of routes that match. That's not needed right now, as we have only two routes, but can be helpful later when we have many.

At this point, we've completed our toy application and in the process verified that our installation of Rails is functioning properly and provides helpful information when things go wrong. After a brief recap, it's now time to move on to building a real application.

## What We Just Did

We constructed a toy application that showed you the following:

- How to create a new Rails application and how to create a new controller in that application
- How to create dynamic content in the controller and display it via the view template
- How to link pages together
- How to debug problems in the code or the URL

This is a great foundation, and it didn't take much time or effort. This experience will continue as we move on to the next chapter and build a much bigger application.

Routing Error

No route matches [GET] "/say/hullo"

Rails.root: /Users/rubys/git/awdwr/edition4/work/demol

[Application Trace](#) | [Framework Trace](#) | [Full Trace](#)

**Routes**

Routes match in priority from top to bottom

Helper	HTTP Verb	Path
<a href="#">Path / Url</a>		<input type="text" value="Path Match"/>
say_hello_path	GET	/say/hello(.:format)
say_goodbye_path	GET	/say/goodbye(.:format)

```

>>
tu
tu

```

## Playtime

Here's some stuff to try on your own:

- Experiment with the following expressions:
  - Addition: `<%= 1+2 %>`
  - Concatenation: `<%= "cow" + "boy" %>`
  - Time in one hour: `<%= 1.hour.from_now.localtime %>`
- A call to the following Ruby method returns a list of all the files in the current directory:
 

```
@files = Dir.glob('*')
```

Use it to set an instance variable in a controller action, and then write the corresponding template that displays the filenames in a list on the browser.

Hint—you can iterate over a collection using something like this:

```

<% @files.each do |file| %>
  file name is: <%= file %>
<% end %>

```

Note that the first and last lines of this loop use `<%` *without* an equal sign. This causes the code embedded in these markers to be executed *without* inserting the results returned into the output.

You might want to use a `<ul>` for the list.

## Cleaning Up

Maybe you've been following along and writing the code in this chapter. If so, chances are that the application is still running on your computer. When we start coding our next application in [Chapter 6, Task A: Creating the Application, on page 69](#), we'll get a conflict the first time we run it because it'll also try to use the computer's port 3000 to talk with the browser. Now is a good time to stop the current application by pressing `Ctrl-C` in the window you used to start it. Microsoft Windows users may need to press `Ctrl-Pause/Break` instead.

Now let's move on to an overview of Rails.