

Chapter 1

From zero to deploy

Welcome to the [Ruby on Rails Tutorial](#)! The purpose of this tutorial is to teach you how to develop custom web applications. The resulting skillset will put you in a great position to get a job as a web developer, start a career as a freelancer, or found a company of your own. If you already know how to develop web applications, this tutorial will quickly get you up to speed with Ruby on Rails.

The focus throughout the Ruby on Rails Tutorial is on general skills that are useful no matter which specific technology you end up using. Once you understand how web apps work, learning another framework can be done with *much* less effort. That being said, the framework of choice in this tutorial—namely, [Ruby on Rails](#)—has never been a better choice for learning web development ([Box 1.1](#)).

Box 1.1. The many advantages of Rails

Ruby on Rails (or just “Rails” for short) is a free and open-source web development framework written in the [Ruby](#) programming language. Upon its debut, Ruby on Rails rapidly became one of the most popular tools for building dynamic web applications. Rails is used by companies as varied as [Airbnb](#), [SoundCloud](#), [Disney](#), [Hulu](#), [GitHub](#), and [Shopify](#), as well as by innumerable freelancers, independent development shops, and startups.

Although there are many choices in web development, Rails stands apart for its elegance, power, and integrated approach to web applications. Using Rails, even novice developers can build a full-stack web application without ever leaving the framework—a huge boon for people learning web development for the first time. Rails also gives you flexibility going forward—for example, serving as a great [back end](#) if you want to build a [single-page application](#) or mobile app sometime down the line.

One big advantage is that Rails is not prone to the “new hotness” problem that plagues some development communities (notably JavaScript/Node.js), in which a dizzyingly complex set of technologies seems to change every six months. As Rails creator [David Heinemeier Hansson](#) once [noted](#):

Back then the complexity merchant of choice was J2EE, but the complaints are uncannily similar to those leveled against JavaScript today... The core premise of Rails remains in many ways as controversial today as it was when it premiered. That by formalizing conventions, eliminating valueless choices, and offering a full-stack framework that provides great defaults for anyone who wants to create a complete application, we can make dramatic strides of productivity.

Due in part to this philosophy, Rails has remained so stable at its core much of this tutorial has been the same since the third edition, launched in 2014. The things you learn here won't go out of date soon.

And yet, Rails continues to innovate. For example, the [Rails 6 release](#) includes major new features for email routing, text formatting, parallel testing, and multiple-database support. A big part of Rails 6 is being “[scalable by default](#)”, which means that *Rails scales* no matter how big your app gets. All this while maintaining rock-solid dependability—indeed, the wildly popular developer platform [GitHub](#), the hugely successful online store-builder [Shopify](#), and the collaboration tool (and very first Rails app) [Basecamp](#) all run their sites on the pre-release versions of Rails. This means that *new versions of Rails are immediately tested by some of the largest, most successful web apps in existence*.

Not bad for a little side project cooked up by a freelance Danish web developer way back in 2004. What was an edgy choice then is an easy choice now: with its proven track-record, productive feature-set, and helpful community, Rails is a fantastic framework for building modern web applications.

There are no formal prerequisites for this book, which contains integrated tutorials for the [Ruby](#) programming language, the Unix command line, [HTML](#), [CSS](#), a small amount of [JavaScript](#), and even a little [SQL](#). That's a lot of material to absorb, though, and if you're new to software development I recommend starting with the tutorials at [Learn Enough](#), especially *[Learn Enough Command Line to Be Dangerous](#)* and *[Learn Enough Ruby to Be Dangerous](#)*.¹ On the other hand, a surprising number of complete beginners have gotten through this tutorial, so don't let me stop you if you're excited to build web apps.

The principal teaching method of this tutorial is building real working software through a series of example applications of increasing sophistication, starting with a minimal *hello* app ([Figure 1.1](#), [Section 1.2](#)), a slightly more capable *toy* app ([Figure 1.2](#), [Chapter 2](#)), and a real *sample* app ([Figure 1.3](#), [Chapter 3](#) through [Chapter 14](#)). As implied by their generic names, these applications focus on general principles, which are applicable to practically any kind of web application. In particular, the full sample application includes all the major features needed by professional-grade web apps, including user signup, login, and account management. The final version of the sample app, developed in [Chapter 14](#), also bears more than a passing resemblance to [Twitter](#)—a website which, coincidentally, was also originally written in Rails.

Let's get started!

¹Adding the rest of the [Learn Enough sequence](#) would certainly provide excellent preparation for this tutorial, but if you're in a hurry you can probably get by with just Command Line and Ruby. *[Learn Enough Ruby to Be Dangerous](#)* in particular has a chapter on building a simple web application using Sinatra, a Ruby-based micro-framework that serves as excellent preparation for Rails. If you get stuck in the present tutorial, I suggest giving *[Learn Enough Ruby to Be Dangerous](#)* and its prerequisites a try, then loop back here to see how it goes the second time.

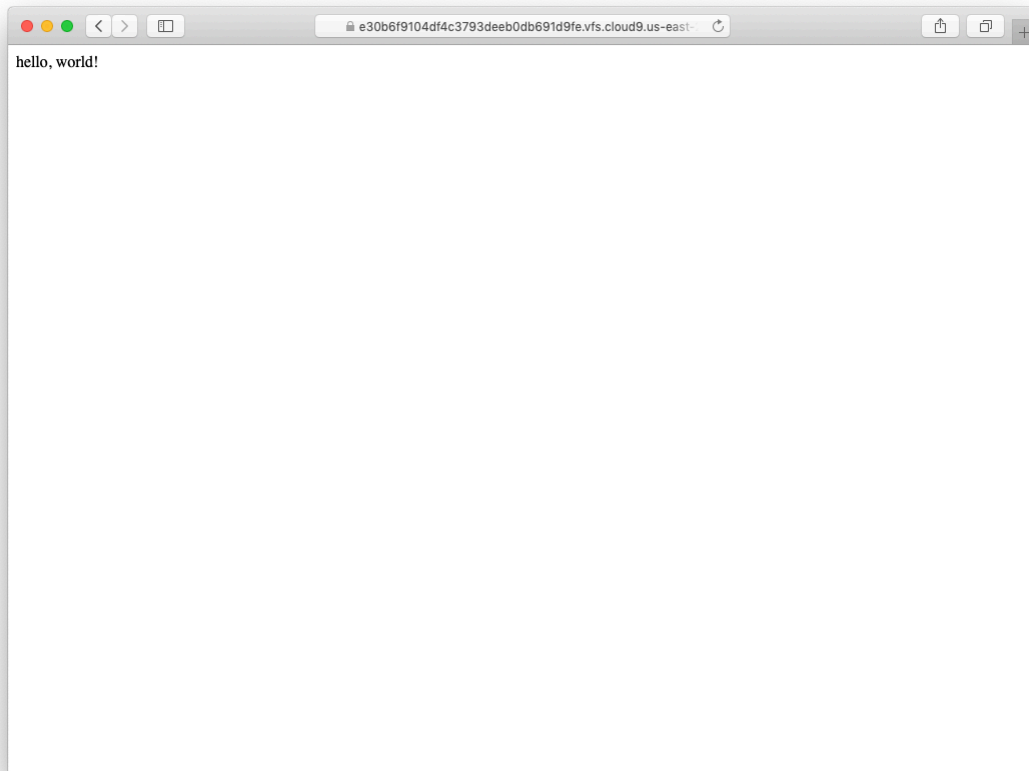


Figure 1.1: The beginning hello app.

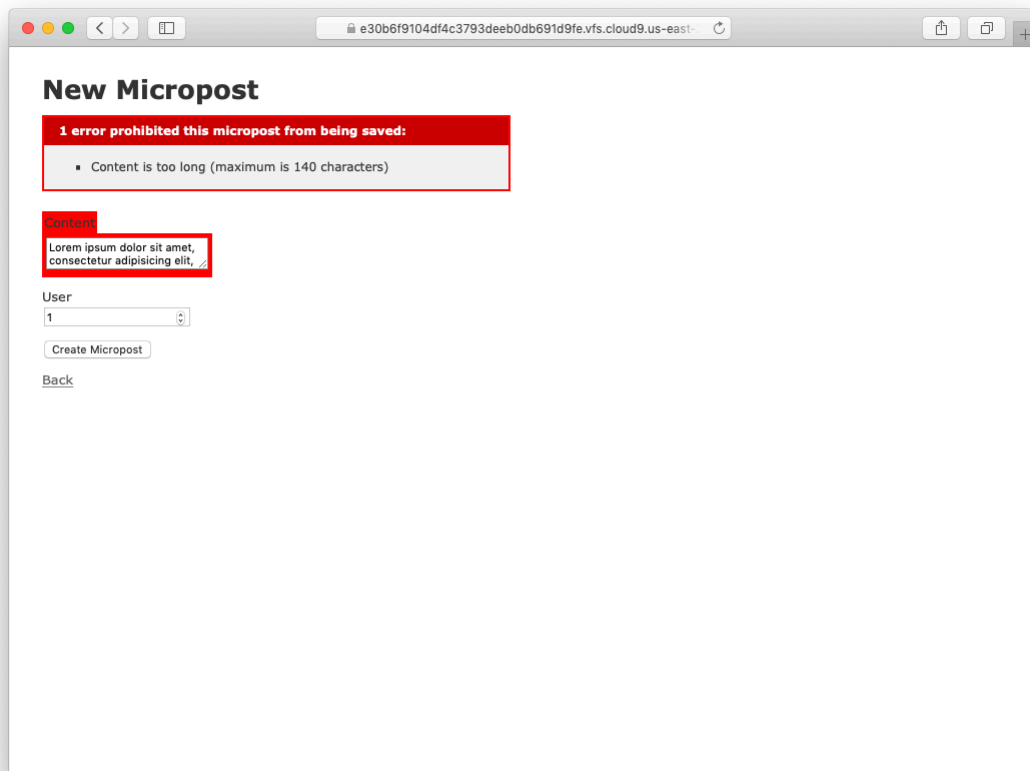


Figure 1.2: An intermediate toy app.

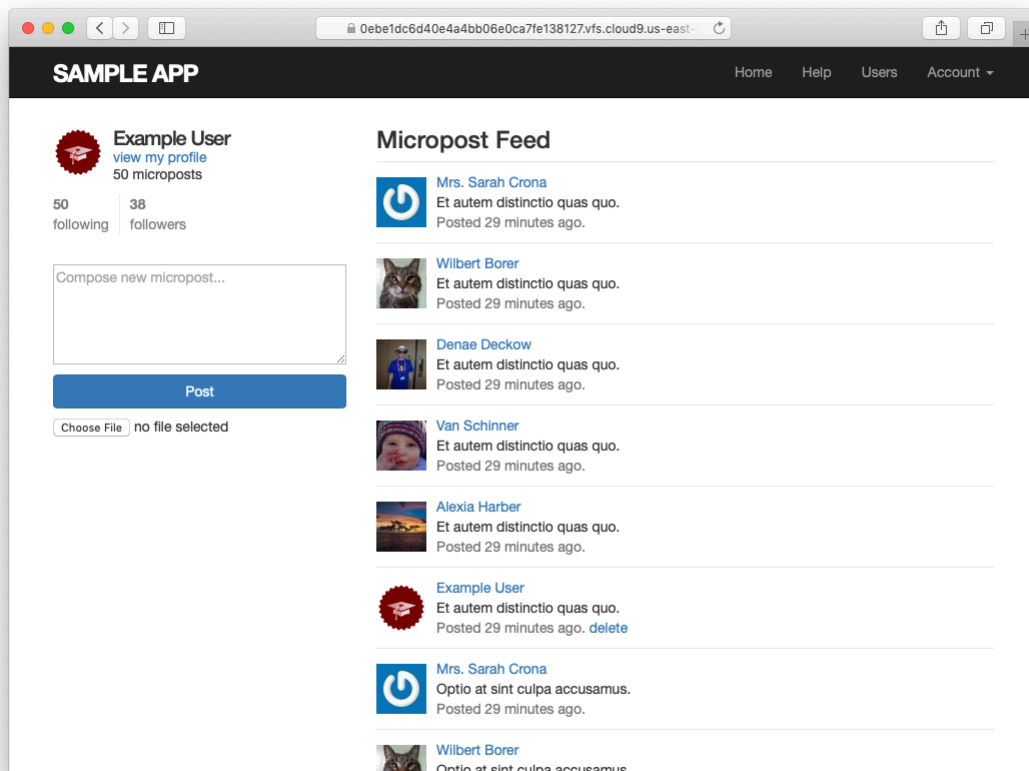


Figure 1.3: The final sample app.

1.1 Up and running

One advantage of using this tutorial is that you can get up and running fast. In particular, the Rails Tutorial has a long-running partnership with [AWS Cloud9](#), a development environment that runs in your browser. The result is a complete system for developing all the software in this tutorial.

This is important because, even for experienced developers, installing Ruby, Rails, and all the associated supporting software can be quite challenging. Compounding the problem is the multiplicity of environments: different operating systems, version numbers, preferences in text editors, etc.

This is why the recommended solution, especially for newer users, is to sidestep most installation and configuration issues by using a *cloud integrated development environment*, or cloud IDE ([Section 1.1.1](#)). The cloud IDE used in this tutorial runs inside an ordinary web browser, and hence works the same across different platforms. It also maintains the current state of your work, so you can take a break from the tutorial and come back to the system just as you left it.

A second possibility is to set up your native system (Windows, macOS, or Linux) for Rails development. It is definitely recommended that you do this eventually, but it can represent significant overhead, and is likely to require a healthy amount of *technical sophistication* ([Box 1.2](#)). Instructions for setting up your native system can be found in the “[Native OS setup](#)” section of *Learn Enough Dev Environment to Be Dangerous*. (Note in particular that you’ll need Ruby 2.6 or greater to run Rails 6.) If you go this route, be sure to complete the configuration and Rails installation steps in [Section 1.1.2](#) as well.

Box 1.2. Technical sophistication

The [Ruby on Rails Tutorial](#) is part of the [Learn Enough](#) family of tutorials, which develop the theme of *technical sophistication*: the combination of hard and soft skills that make it seem like you can magically solve any technical problem (as illustrated in “[Tech Support Cheat Sheet](#)” from [xkcd](#)).

Knowing how to code is an important component of technical sophistication, but there's more to it than that—you also have to know how to click around menu items to learn the capabilities of a particular application, how to clarify a confusing error message by [googling it](#), or when to give up and just reboot the darn thing.

Because web applications have so many moving parts, they offer ample opportunities to develop your technical sophistication. In the context of Rails web development, some specific examples of technical sophistication include making sure you're using the right Ruby gem versions, running `bundle install` or `bundle update`, and restarting the local webserver if something doesn't work. (Don't worry if all this sounds like gibberish; we'll cover everything mentioned here in the course of completing this tutorial.)

As you proceed through this tutorial, in all likelihood you will occasionally be tripped up by things not immediately working as expected. Although some particularly tricky steps are explicitly highlighted in the text, it is impossible to anticipate all the things that can go wrong. I recommend you embrace these inevitable stumbling blocks as opportunities to work on improving your technical sophistication. Or, as we say in [geek speak](#): *It's not a bug, it's a feature!*

1.1.1 Development environment

Considering various idiosyncratic customizations, there are probably as many development environments as there are Rails programmers. To avoid this complexity, the *Ruby on Rails Tutorial* standardizes on the excellent cloud development environment [Cloud9](#), part of Amazon Web Services (AWS). The resulting workspace environment comes pre-configured with most of the software needed for Ruby on Rails web development, including Ruby, RubyGems, Git. (Indeed, the only big piece of software we'll install separately is Rails itself, and this is intentional ([Section 1.1.2](#)).)

The cloud IDE includes the three essential components needed to develop web applications: a command-line terminal, a filesystem navigator, and a text editor ([Figure 1.4](#)). Among other features, the cloud IDE's text editor supports

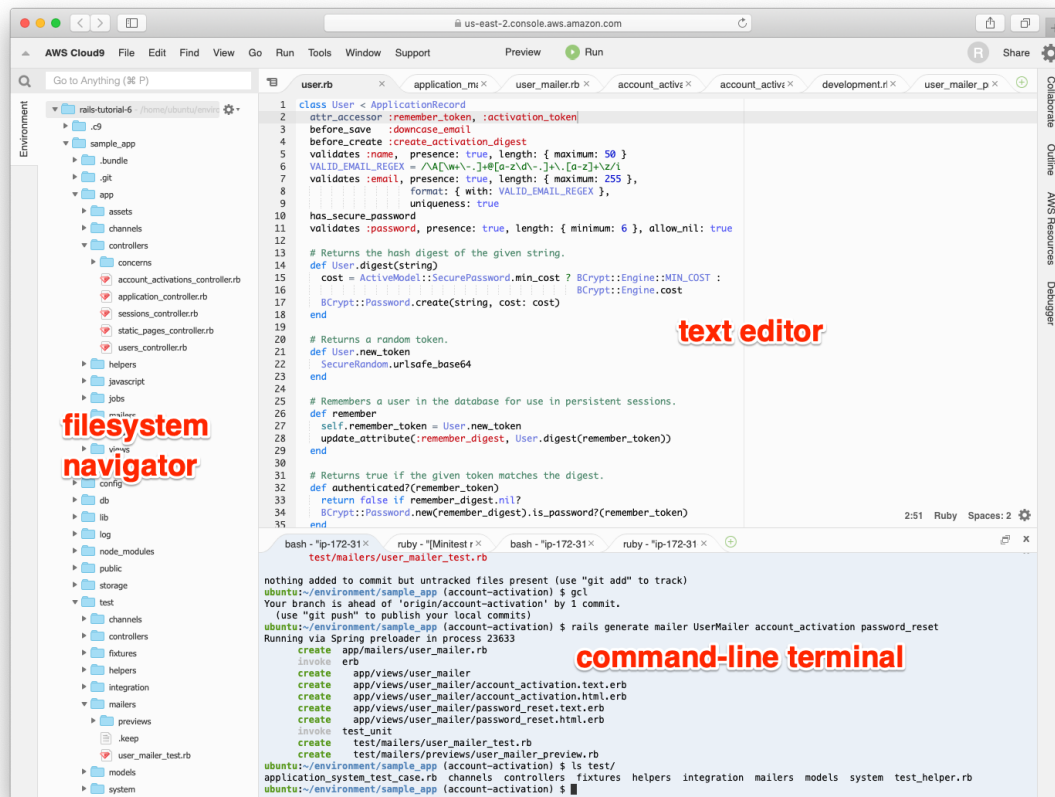


Figure 1.4: The anatomy of the cloud IDE.

the “Find in Files” global search that I consider essential to navigating any large Ruby or Rails project. Finally, even if you decide not to use the cloud IDE exclusively in real life (and I certainly recommend learning other tools as well), it provides an excellent introduction to the general capabilities of command-line terminals, text editor, and other development tools.

Here are the steps for getting started with the cloud development environment:²

1. Because Cloud9 is part of Amazon Web Services (AWS), if you already

²Due to the constantly evolving nature of sites like AWS, details may vary; use your technical sophistication (Box 1.2) to resolve any discrepancies.

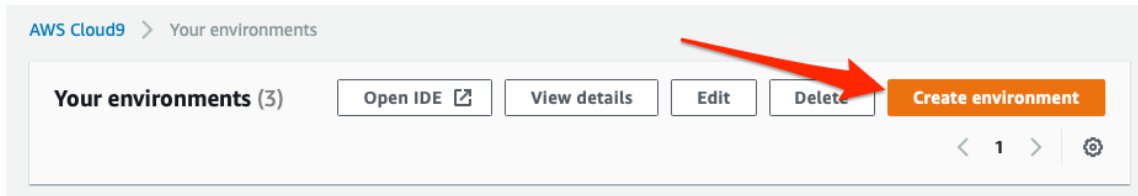


Figure 1.5: Creating an environment on AWS Cloud9.

have an AWS account you can just [sign in](#).³ To create a new Cloud9 workspace environment, go to the [AWS console](#) and type “Cloud9” in the search box.

2. If you don’t already have an AWS account, you should [sign up for a free account at AWS Cloud9](#).⁴ In order to prevent abuse, AWS requires a valid credit card for signup, but the workspace is 100% free (for a year as of this writing), and your card will not be charged. You might have to wait up to 24 hours for the account to be activated, but in my case it was ready in about ten minutes.
3. Once you’ve successfully gotten to the Cloud9 administrative page ([Figure 1.5](#)), clicking on “Create environment” and fill in the information as shown in [Figure 1.6](#), including the name “rails-tutorial”.⁵ fill in the description as shown in [Figure 1.6](#). On the next page, choose **Ubuntu Server** (*not* Amazon Linux) ([Figure 1.7](#)), and then click “Next step”.

clicking the confirmation buttons to accept the default settings until AWS starts provisioning the IDE ([Figure 1.9](#)). You may run into a warning message about being a “root” user, which you can safely ignore at this early stage. (We’ll discuss the preferred but more complicated practice, called an Identity and Access Management (IAM) user, in [Section 13.4.4](#).)

³<https://aws.amazon.com/>

⁴<https://www.railstutorial.org/cloud9-signup>

⁵If you’ve previously done this tutorial, you may want to use a fresh environment, with a name like “rails-tutorial-6”.

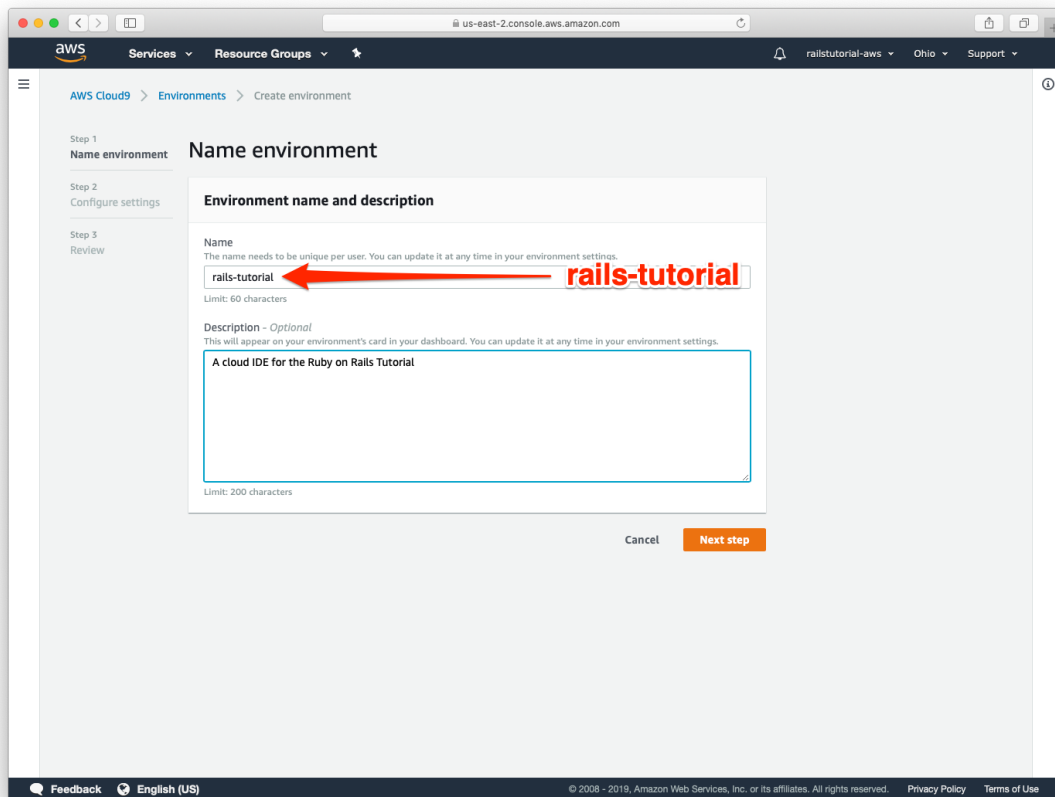


Figure 1.6: Naming a new work environment at AWS Cloud9.

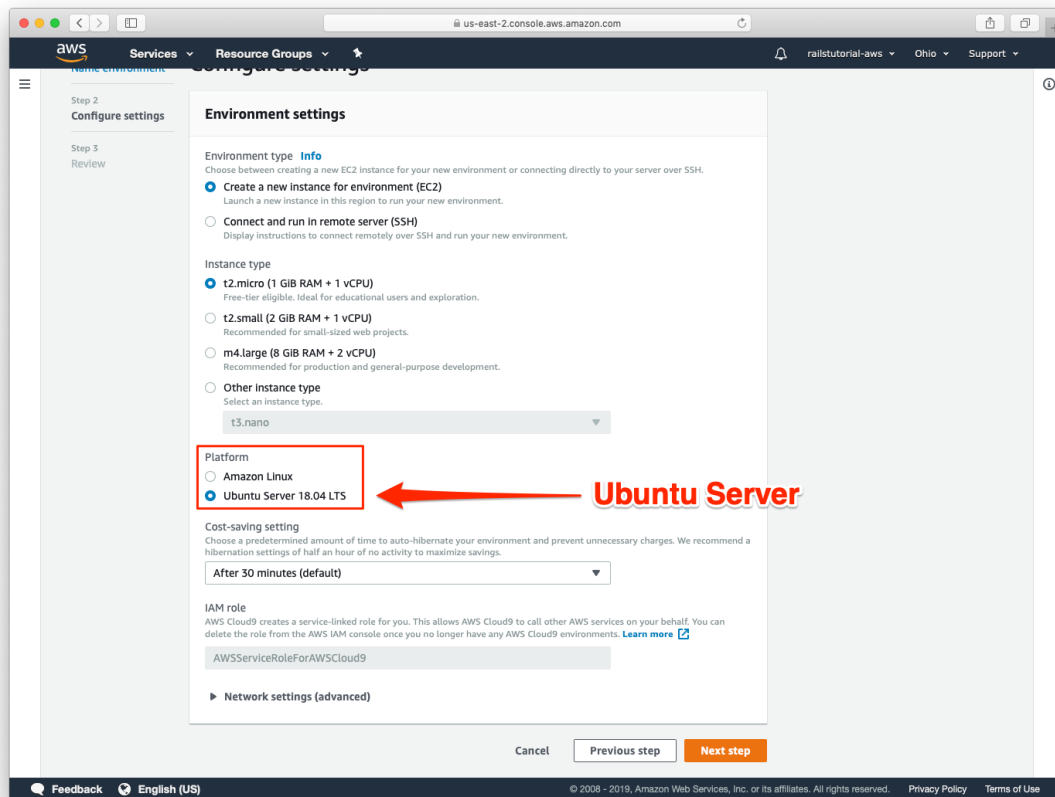


Figure 1.7: Selecting Ubuntu Server.

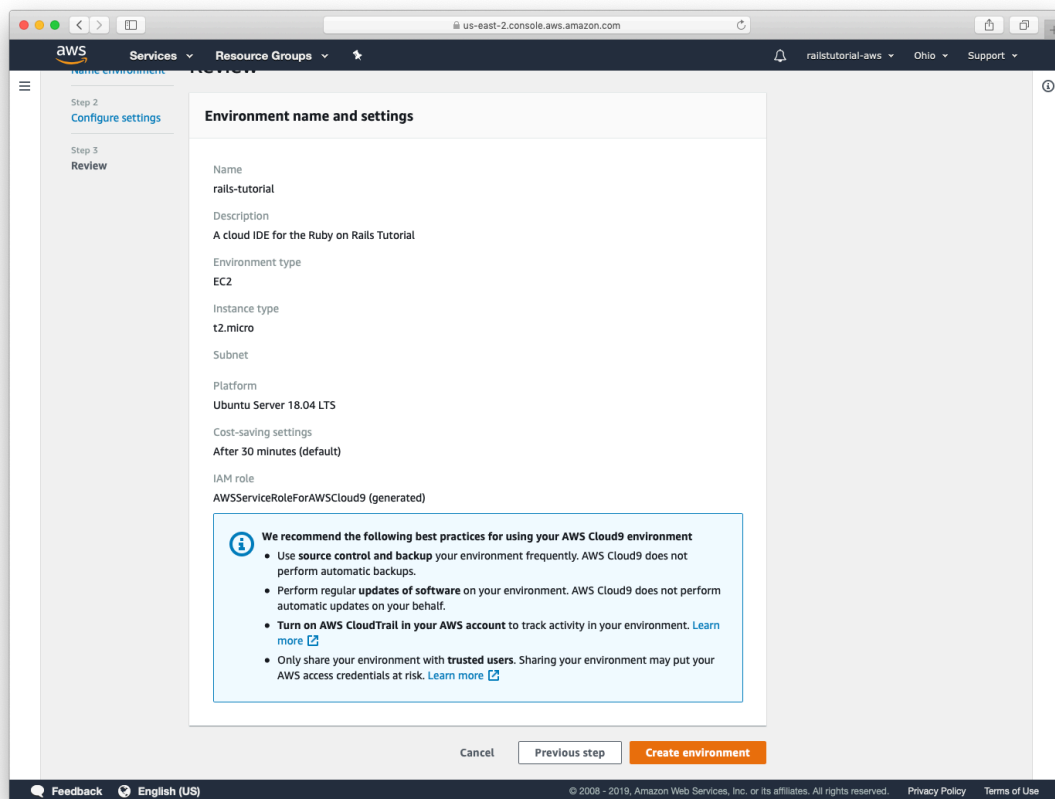


Figure 1.8: The final step before provisioning the IDE.

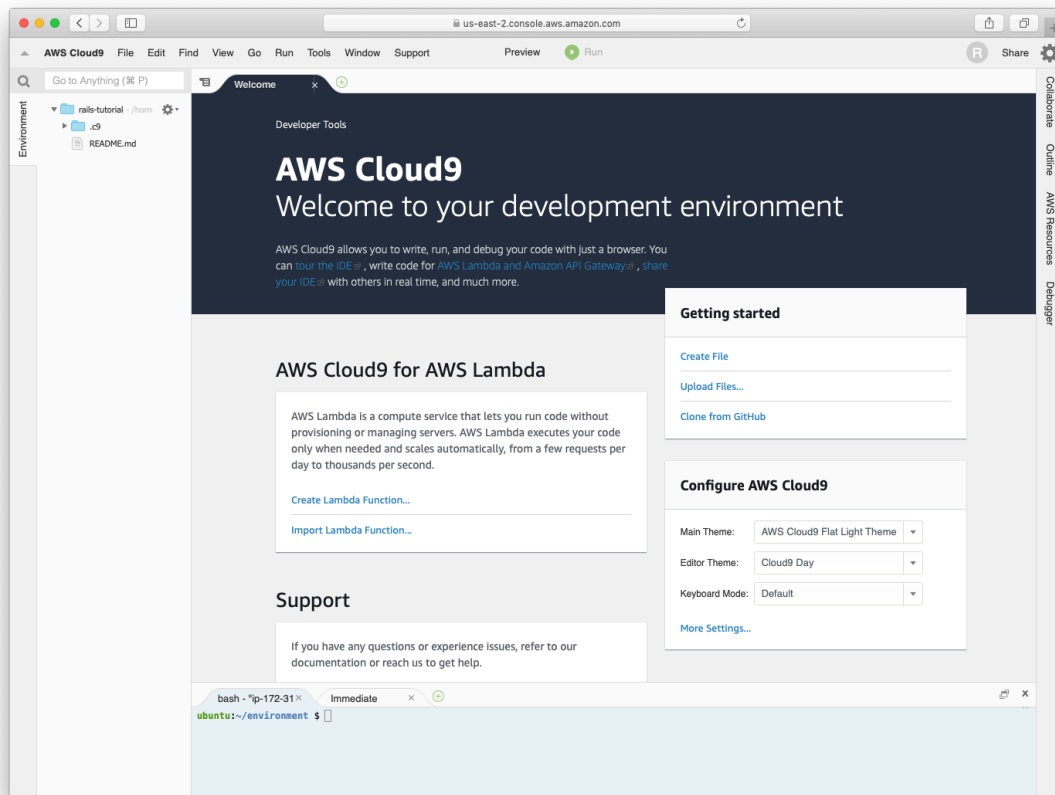


Figure 1.9: The default cloud IDE.

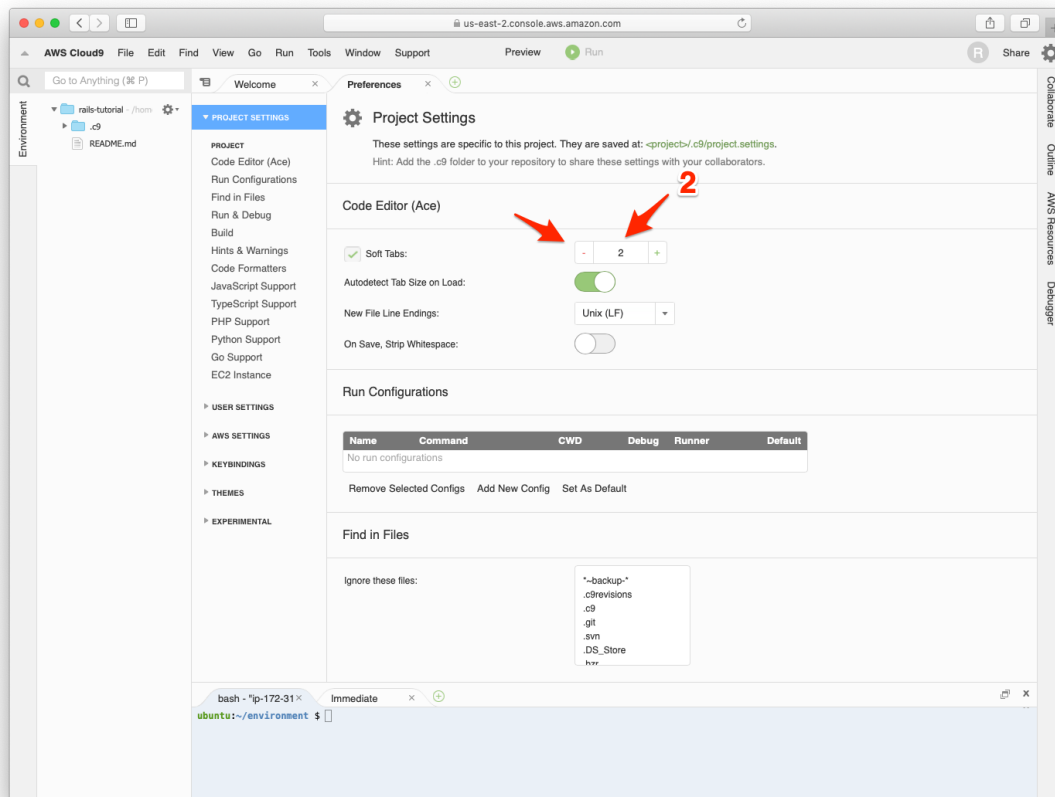


Figure 1.10: Setting Cloud9 to use two spaces for indentation.

Because using two spaces for indentation is a near-universal convention in Ruby, I also recommend changing the editor to use two spaces instead of the default four. As shown in [Figure 1.10](#), you can do this by clicking the gear icon in the upper right and then clicking the minus sign in the “Soft Tabs” setting until it reaches 2. (Note that this takes effect immediately; you don’t need to click a “Save” button.)

1.1.2 Installing Rails

The development environment from [Section 1.1.1](#) includes all the software we need to get started except for Rails itself. This is by design, as installing the exact version of Rails used in this tutorial is important for getting predictable results.

First, we'll do a little preparation by adding configuration settings to prevent the time-consuming installation of local Ruby documentation, as shown in [Listing 1.1](#).⁶ Note that this step needs to be done only once per system. (For more information on the command-line and other conventions in this book, see [Section 1.6](#).)

Listing 1.1: Configuring the `.gemrc` file to skip the installation of Ruby documentation.

```
$ echo "gem: --no-document" >> ~/.gemrc
```

To install Rails, we'll use the `gem` command provided by the *RubyGems* package manager, which involves typing the command shown in [Listing 1.2](#) into your command-line terminal. (If developing on your local system, this means using a regular terminal window; if using the cloud IDE, this means using the command-line area shown in [Figure 1.4](#).)

Listing 1.2: Installing Rails with a specific version number.

```
$ gem install rails -v 6.0.1
```

Here the `-v` flag ensures that the specified version of Rails gets installed. You can confirm that the installation succeeded by passing the `-v` flag to the `rails` command itself:

⁶This uses the `echo` and `>>` (append) commands covered in [Section 1.3](#) and [Section 2.1](#) of *Learn Enough Command Line to Be Dangerous*. Note that if the file being appended to doesn't exist, `>>` is smart enough to create it.


```
$ rails -v
Rails 6.0.1
```

The version number output by this command should match the version installed in [Listing 1.2](#).

There's one more configuration step, which is to install [Yarn](#), a program to manage software dependencies. If you're using your native OS, you should follow the [Yarn installation instructions](#) for your platform. If you're on the cloud IDE, you can run this command, which downloads and executes the necessary commands from the Learn Enough [CDN](#):

```
$ source <(curl -sL https://cdn.learnenough.com/yarn_install)
```

From time to time, you'll probably get a warning message that looks like this:

```
=====
Your Yarn packages are out of date!
Please run `yarn install --check-files` to update.
=====
```

All you need to do if this happens is execute the suggested **yarn** command:

```
$ yarn install --check-files
```

That's it! You've now got a system fully configured for Ruby on Rails web development.

1.2 The first application

Following a [long tradition](#) in computer programming, our goal for the first application is to write a “hello, world” program. In particular, we will create a simple application that displays the string “hello, world!” on a web page, both on our development environment ([Section 1.2.4](#)) and on the live web ([Section 1.4](#)).

Virtually all Rails applications start the same way, by running the **rails new** command. This handy command creates a skeleton Rails application in a directory of your choice. To get started, users *not* using the Cloud9 IDE recommended in [Section 1.1.1](#) should make a **environment** directory for your Rails projects if it doesn't already exist ([Listing 1.3](#)) and then change into the directory.⁷

Listing 1.3: Making an **environment** directory for Rails projects.

```
# These steps are not needed on the cloud IDE.
$ cd # Change to the home directory.
$ mkdir environment # Make an environment directory.
$ cd environment/ # Change into the environment directory.
```

[Listing 1.3](#) uses the Unix commands **cd** and **mkdir**; see [Box 1.3](#) if you are not already familiar with these commands.

Box 1.3. A crash course on the Unix command line

For readers coming from Windows or macOS, the Unix command line may be unfamiliar. Luckily, if you are using the recommended cloud environment, you automatically have access to a Unix (Linux) command line running a standard **shell** (command-line interface) known as **Bash**.

The basic idea of the command line is simple: by issuing short commands, users can perform a large number of operations, such as creating directories (**mkdir**), moving and copying files (**mv** and **cp**), and navigating the filesystem by changing directories (**cd**). Although the command line may seem primitive to users mainly familiar with graphical user interfaces (GUIs), appearances are deceiving: the command line is one of the most powerful tools in the developer's toolbox. Indeed, you will rarely see the desktop of an experienced developer without several open terminal windows running command-line shells.

⁷This step is designed to unify the treatment of native systems and the cloud IDE by using identical directory structures. If you are confident in your technical sophistication, feel free to omit this step, and use a directory of your choice.

Description	Command	Example
list contents	ls	\$ ls -l
make directory	mkdir <dirname>	\$ mkdir environment
change directory	cd <dirname>	\$ cd environment/
cd one directory up		\$ cd ..
cd to home directory		\$ cd - or just \$ cd
cd to path incl. home dir		\$ cd ~/environment/
move file (rename)	mv <source> <target>	\$ mv foo bar
copy file	cp <source> <target>	\$ cp foo bar
remove file	rm <file>	\$ rm foo
remove empty directory	rmdir <directory>	\$ rmdir environment/
remove nonempty directory	rm -rf <directory>	\$ rm -rf tmp/
concatenate & display file contents	cat <file>	\$ cat ~/.ssh/id_rsa.pub

Table 1.1: Some common Unix commands.

The general subject is deep, but for the purposes of this tutorial we will need only a few of the most common Unix command-line commands, as summarized in Table 1.1. For a more thorough introduction to the basics of the command line, see the first [Learn Enough](#) tutorial, *Learn Enough Command Line to Be Dangerous*.

The next step on both local systems and the cloud IDE is to create the first application using the command in Listing 1.4. Note that Listing 1.4 explicitly includes the Rails version number as part of the command. This ensures that the same version of Rails we installed in Listing 1.2 is used to create the first application's file structure.

Listing 1.4: Running `rails new` (with a specific version number).

```
$ cd ~/environment
$ rails _6.0.1_ new hello_app
  create
  create  README.md
  create  Rakefile
  create  .ruby-version
  create  config.ru
  create  .gitignore
  create  Gemfile
  run    git init from "."
```

```
Initialized empty Git repository in /home/ubuntu/environment/hello_app/.git/  
  create  package.json  
  create  app  
  create  app/assets/config/manifest.js  
  create  app/assets/stylesheets/application.css  
  create  app/channels/application_cable/channel.rb  
  create  app/channels/application_cable/connection.rb  
  create  app/controllers/application_controller.rb  
  create  app/helpers/application_helper.rb  
  .  
  .  
  .
```

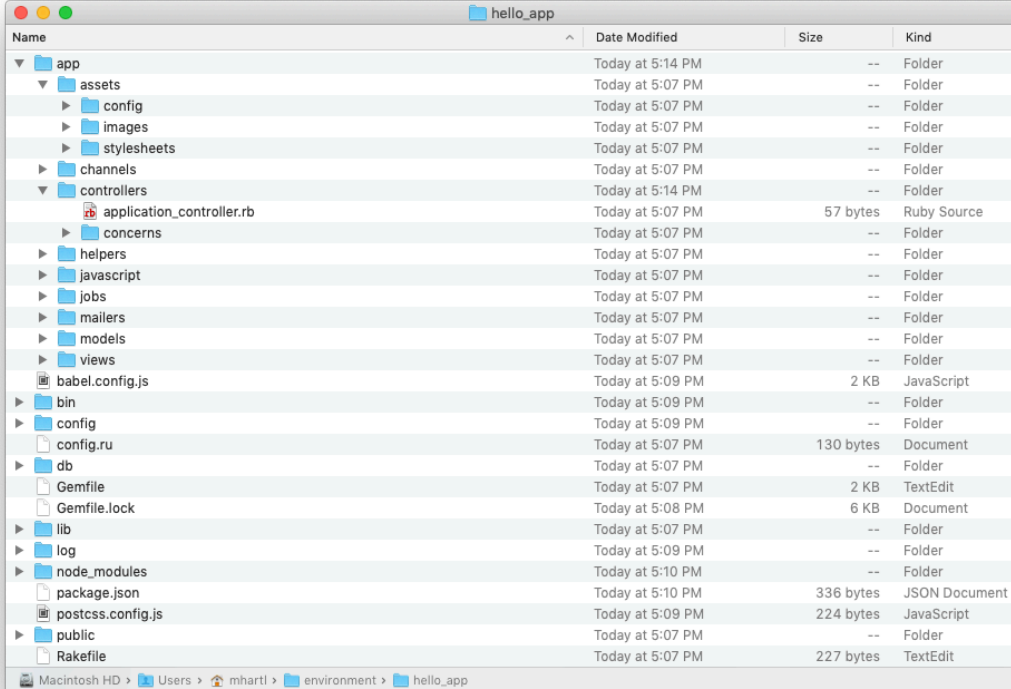
Notice how many files and directories the **rails** command creates. This standard directory and file structure (Figure 1.11) is one of the many advantages of Rails: it immediately gets you from zero to a functional (if minimal) application. Moreover, since the structure is common to all Rails apps, you can immediately get your bearings when looking at someone else’s code.

A summary of the default Rails files appears in Table 1.2. We’ll learn about most of these files and directories throughout the rest of this book. In particular, starting in Section 5.2.1 we’ll discuss the **app/assets** directory, part of the *asset pipeline* that makes it easy to organize and deploy assets such as Cascading Style Sheets and image files.

1.2.1 Bundler

After creating a new Rails application, the next step is to use *Bundler* to install and include the gems needed by the app. Bundler is run automatically (via **bundle install**) by the **rails** command in Listing 1.4, but in this section we’ll make some changes to the default application gems and run Bundler again. This involves opening the **Gemfile** with a text editor. (With the cloud IDE, this involves clicking the arrow in the file navigator to open the sample app directory and double-clicking the **Gemfile** icon.) Although the exact version numbers and details may differ slightly, the results should look something like Figure 1.12 and Listing 1.5. (The code in this file is Ruby, but don’t worry at this point about the syntax; Chapter 4 will cover Ruby in more depth.)

If the files and directories don’t appear as shown in Figure 1.12, click on



The screenshot shows a file explorer window titled 'hello_app' with a table of files and folders. The table has columns for Name, Date Modified, Size, and Kind. The directory structure is as follows:

Name	Date Modified	Size	Kind
app	Today at 5:14 PM	--	Folder
assets	Today at 5:07 PM	--	Folder
config	Today at 5:07 PM	--	Folder
images	Today at 5:07 PM	--	Folder
stylesheets	Today at 5:07 PM	--	Folder
channels	Today at 5:07 PM	--	Folder
controllers	Today at 5:14 PM	--	Folder
application_controller.rb	Today at 5:07 PM	57 bytes	Ruby Source
concerns	Today at 5:07 PM	--	Folder
helpers	Today at 5:07 PM	--	Folder
javascript	Today at 5:07 PM	--	Folder
jobs	Today at 5:07 PM	--	Folder
mailers	Today at 5:07 PM	--	Folder
models	Today at 5:07 PM	--	Folder
views	Today at 5:07 PM	--	Folder
babel.config.js	Today at 5:09 PM	2 KB	JavaScript
bin	Today at 5:09 PM	--	Folder
config	Today at 5:09 PM	--	Folder
config.ru	Today at 5:07 PM	130 bytes	Document
db	Today at 5:07 PM	--	Folder
Gemfile	Today at 5:07 PM	2 KB	TextEdit
Gemfile.lock	Today at 5:08 PM	6 KB	Document
lib	Today at 5:07 PM	--	Folder
log	Today at 5:09 PM	--	Folder
node_modules	Today at 5:10 PM	--	Folder
package.json	Today at 5:10 PM	336 bytes	JSON Document
postcss.config.js	Today at 5:09 PM	224 bytes	JavaScript
public	Today at 5:07 PM	--	Folder
Rakefile	Today at 5:07 PM	227 bytes	TextEdit

Figure 1.11: The directory structure for a newly created Rails app.

File/Directory	Purpose
app/	Core application (app) code, including models, views, controllers, and helpers
app/assets	Applications assets such as Cascading Style Sheets (CSS) and images
bin/	Binary executable files
config/	Application configuration
db/	Database files
doc/	Documentation for the application
lib/	Library modules
log/	Application log files
public/	Data accessible to the public (e.g., via web browsers), such as error pages
bin/rails	A program for generating code, opening console sessions, or starting a local server
test/	Application tests
tmp/	Temporary files
README.md	A brief description of the application
Gemfile	Gem requirements for this app
Gemfile.lock	A list of gems used to ensure that all copies of the app use the same gem versions
config.ru	A configuration file for Rack middleware
.gitignore	Patterns for files that should be ignored by Git

Table 1.2: A summary of the default Rails directory structure.

the file navigator’s gear icon and select “Refresh File Tree”. (As a general rule, you should refresh the file tree any time files or directories don’t appear as expected.)⁸

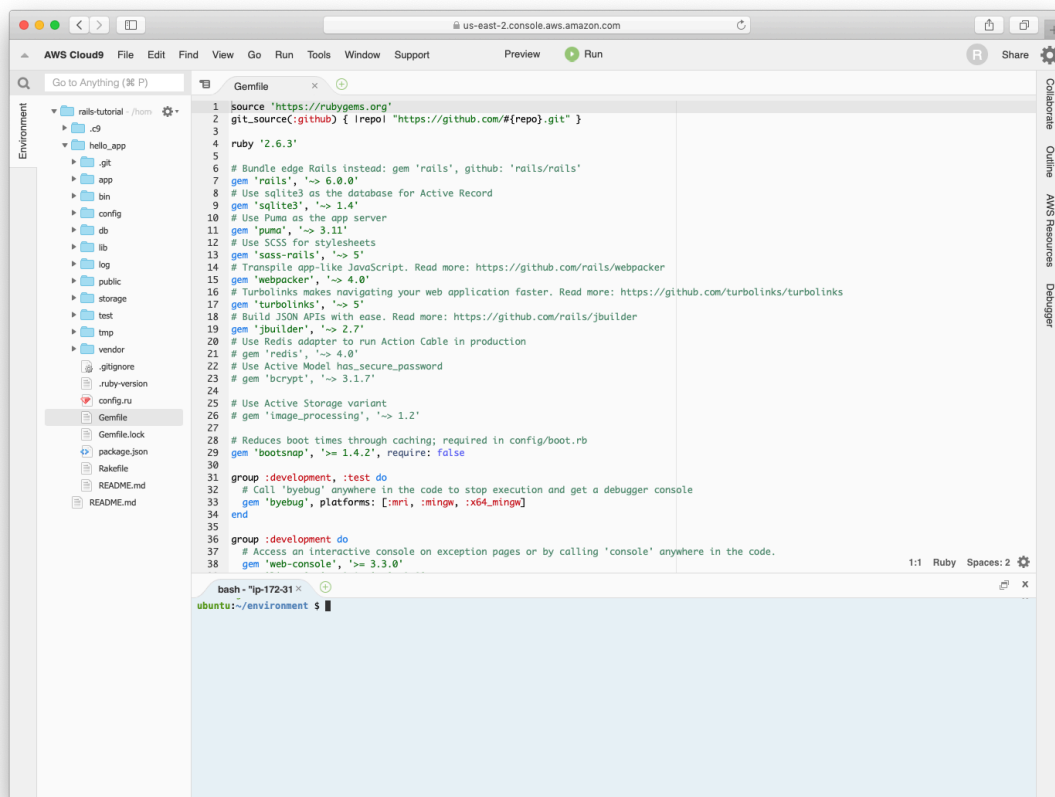
Listing 1.5: The default **Gemfile** in the **hello_app** directory.

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

ruby '2.6.3'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 6.0.1'
# Use sqlite3 as the database for Active Record
gem 'sqlite3', '~> 1.4'
# Use Puma as the app server
gem 'puma', '~> 3.11'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5'
# Transpile app-like JavaScript. Read more: https://github.com/rails/webpacker
gem 'webpacker', '~> 4.0'
# Turbolinks makes navigating your web application faster.
```

⁸This is a typical example of technical sophistication (Box 1.2).



```
1 source 'https://rubygems.org'
2 git_source(:github) { |repo| "https://github.com/#{repo}.git" }
3
4 ruby '2.6.3'
5
6 # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
7 gem 'rails', '~> 6.0.0'
8 # Use sqlite3 as the database for Active Record
9 gem 'sqlite3', '~> 1.4'
10 # Use Puma as the app server
11 gem 'puma', '~> 3.11'
12 # Use SCSS for stylesheets
13 gem 'sass-rails', '~> 5'
14 # Transpile app-like JavaScript. Read more: https://github.com/rails/webpacker
15 gem 'webpacker', '~> 4.0'
16 # Turbolinks makes navigating your web application faster. Read more: https://github.com/turbolinks/turbolinks
17 gem 'turbolinks', '~> 5'
18 # Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
19 gem 'jbuilder', '~> 2.7'
20 # Use Redis adapter to run Action Cable in production
21 gem 'redis', '~> 4.0'
22 # Use Active Model has_secure_password
23 gem 'bcrypt', '~> 3.1.7'
24
25 # Use Active Storage variant
26 gem 'image_processing', '~> 1.2'
27
28 # Reduces boot times through caching; required in config/boot.rb
29 gem 'bootsnap', '>= 1.4.2', require: false
30
31 group :development, :test do
32   # Call 'byebug' anywhere in the code to stop execution and get a debugger console
33   gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
34 end
35
36 group :development do
37   # Access an interactive console on exception pages or by calling 'console' anywhere in the code.
38   gem 'web-console', '>= 3.3.0'
```

Figure 1.12: The default **Gemfile** open in a text editor.

```
# Read more: https://github.com/turbolinks/turbolinks
gem 'turbolinks', '~> 5'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.7'
# Use Redis adapter to run Action Cable in production
# gem 'redis', '~> 4.0'
# Use Active Model has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Use Active Storage variant
# gem 'image_processing', '~> 1.2'

# Reduces boot times through caching; required in config/boot.rb
gem 'bootsnap', '>= 1.4.2', require: false

group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a
  # debugger console
  gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  # Access an interactive console on exception pages or by calling 'console'
  # anywhere in the code.
  gem 'web-console', '>= 3.3.0'
  gem 'listen', '>= 3.0.5', '< 3.2'
  # Spring speeds up development by keeping your application running in the
  # background. Read more: https://github.com/rails/spring
  gem 'spring'
  gem 'spring-watcher-listen', '~> 2.0.0'
end

group :test do
  # Adds support for Capybara system testing and selenium driver
  gem 'capybara', '>= 2.15'
  gem 'selenium-webdriver'
  # Easy installation and use of web drivers to run system tests with browsers
  gem 'webdrivers'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

Many of these lines are commented out with the hash symbol `#` (Section 4.2); they are there to show you some commonly needed gems and to give examples of the Bundler syntax. For now, we won't need any gems other than the defaults.

Unless you specify a version number to the `gem` command, Bundler will automatically install the latest requested version of the gem. This is the case,

for example, in the code

```
gem 'sqlite3'
```

There are also two common ways to specify a gem version range, which allows us to exert some control over the version used by Rails. The first looks like this:

```
gem 'capybara', '>= 2.15'
```

This installs the latest version of the `capybara` gem (which is used in testing) as long as it's greater than or equal to version **2.15**—even if it's, say, version **7.2**.

The second method looks like this:

```
gem 'rails', '~> 6.0.1'
```

This installs the gem `rails` as long as it's version **6.0.1** or newer but *not* **6.1** or newer. In other words, the `>=` notation always installs the latest gem, whereas the `-> 6.0.1` notation will install **6.0.2** (if available) but not **6.1.0**.⁹

Unfortunately, experience shows that even minor point releases can break Rails applications, so for the *Ruby on Rails Tutorial* we'll err on the side of caution by including exact version numbers for all gems. You are welcome to use the most up-to-date version of any gem, including using the `->` construction in the **Gemfile** (which I generally recommend for more advanced users), but be warned that this may cause the tutorial to act unpredictably.

Converting the **Gemfile** in Listing 1.5 to use exact gem versions results in the code shown in Listing 1.6.¹⁰ Note that we've also taken this opportunity to arrange for the `sqlite3` gem to be included only in a development or test environment (Section 7.1.1), which prevents potential conflicts with the database

⁹Similarly, `-> 6.0` would install version **6.9** of a gem but not **7.0**. This is especially useful if the project in question uses *semantic versioning* (also called “semver”), which is a convention for numbering releases designed to minimize the chances of breaking software dependencies.

¹⁰You can determine the exact version number for each gem by running `gem list <gem name>` at the command line, but Listing 1.6 saves you the trouble.

used by Heroku (Section 1.4). Finally, we've removed the line from Listing 1.5 specifying the exact Ruby version number; as noted in Section 7.5.4, it's recommended to keep this line in a mission-critical app, but keeping it in a tutorial of this nature introduces potential errors and complexity. (That said, if your app fails to work without that line, you should definitely restore it.)

Important note: For all the Gemfiles in this book, you should use the version numbers listed at gemfiles-6th-ed.railstutorial.org instead of the ones listed below (although they should be identical if you are reading this online).

Listing 1.6: A **Gemfile** with an explicit version for each Ruby gem.

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',          '6.0.1'
gem 'puma',           '3.12.1'
gem 'sass-rails',    '5.1.0'
gem 'webpacker',     '4.0.7'
gem 'turbolinks',    '5.2.0'
gem 'jbuilder',      '2.9.1'
gem 'bootsnap',      '1.4.4', require: false

group :development, :test do
  gem 'sqlite3', '1.4.1'
  gem 'byebug',  '11.0.1', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  gem 'web-console',          '4.0.1'
  gem 'listen',               '3.1.5'
  gem 'spring',               '2.1.0'
  gem 'spring-watcher-listen', '2.0.1'
end

group :test do
  gem 'capybara',             '3.28.0'
  gem 'selenium-webdriver',   '3.142.4'
  gem 'webdrivers',           '4.1.2'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

Once you've placed the contents of Listing 1.6 into the application's **Gem-**

file, install the gems using **bundle install**:¹¹

```
$ cd hello_app/  
$ bundle install  
Fetching source index for https://rubygems.org/  
.  
.  
.
```

The **bundle install** command might take a few moments, but when it's done our application will be ready to run.

By the way, when you run **bundle install** it's possible that you'll get a message saying you need to run **bundle update** first. In this case you should... run **bundle update** first! (Learning not to panic when things don't go exactly as planned is a key part of technical sophistication, and you'll be amazed at how often the “error” message contains the exact instructions you need to fix the problem at hand.)

1.2.2 rails server

Thanks to running **rails new** in Section 1.2 and **bundle install** in Section 1.2.1, we already have an application we can run—but how? Happily, Rails comes with a command-line program, or *script*, that runs a *local* webserver to assist us in developing our application: **rails server**.

Before running **rails server**, it's necessary on some systems (including the cloud IDE) to allow connections to the local web server. To enable this, you should navigate to the file **config/environments/development.rb** and paste in the two extra lines shown in Listing 1.7 and Figure 1.13.

Listing 1.7: Allowing connections to the local web server.

```
config/environments/development.rb
```

```
Rails.application.configure do
```

```
.
```

¹¹As noted in Table 3.1, you can even leave off **install**, as the **bundle** command by itself is an alias for **bundle install**.

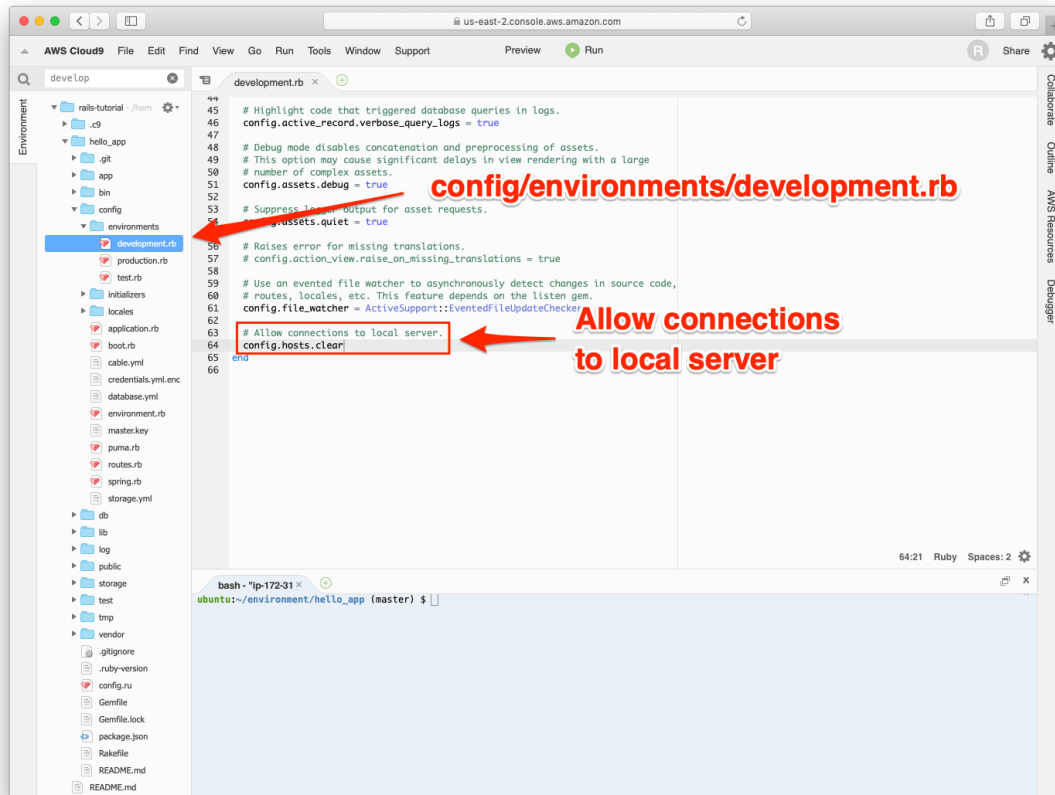


Figure 1.13: Allowing Cloud9 to connect to the Rails server.

```

.
.
# Allow connections to local server.
config.hosts.clear
end

```

The **rails server** command appears in Listing 1.8, which I recommend you run in a second terminal tab so that you can still issue commands in the first tab, as shown in Figure 1.14 and Figure 1.15. Note from Listing 1.8 that you can shut the server down using Ctrl-C.¹²

¹²Here “C” refers to the character on the keyboard, not the capital letter, so there’s no need to hold down the Shift key to get a capital “C”.

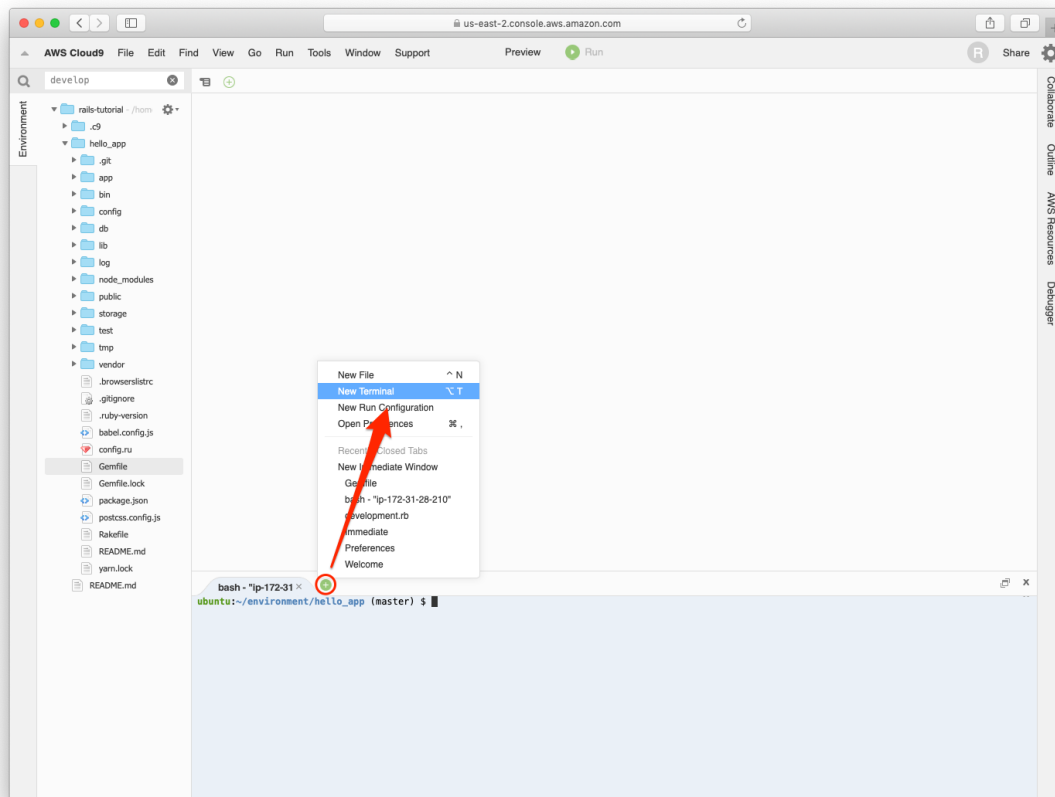


Figure 1.14: Opening a new terminal tab.

Listing 1.8: Running the Rails server.

```
$ cd ~/environment/hello_app/  
$ rails server  
=> Booting Puma  
=> Ctrl-C to shutdown server
```

To view the result of **rails server** on a native OS, paste the URL <http://localhost:3000> into the address bar of your browser. On the cloud IDE, go to Preview and click on Preview Running Application (Figure 1.16), and then open it in a full browser window or tab (Figure 1.17). In either case, the result should look something like Figure 1.18.

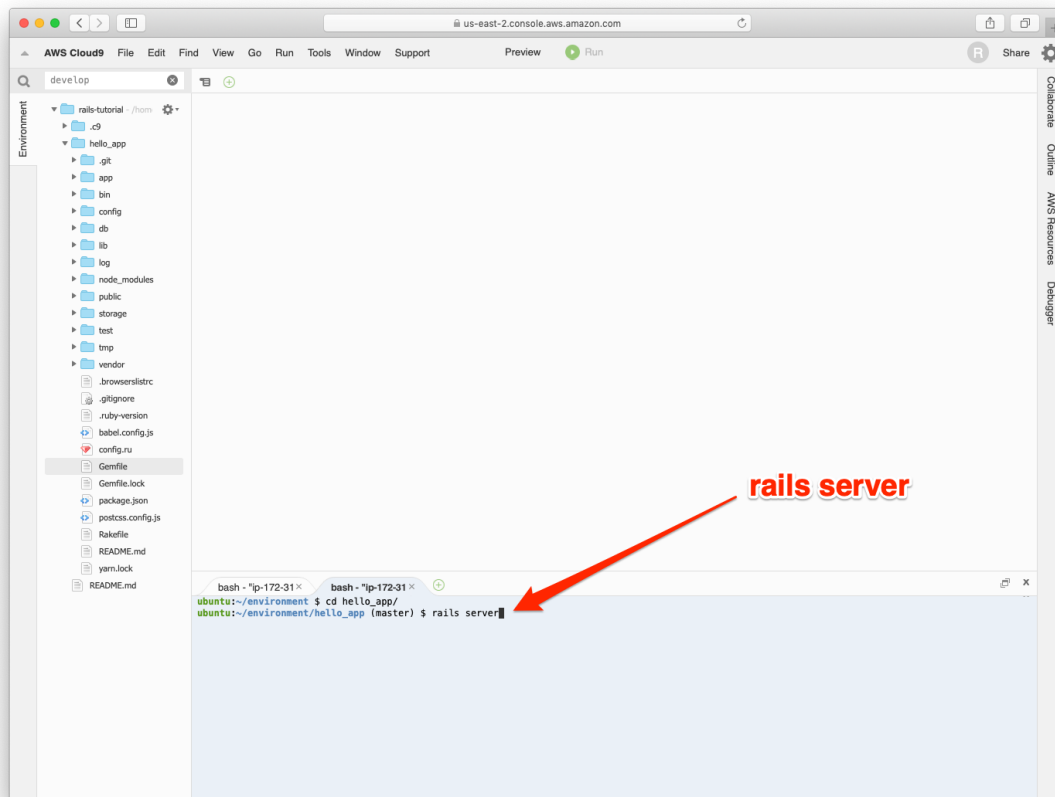


Figure 1.15: Running the Rails server in a separate tab.

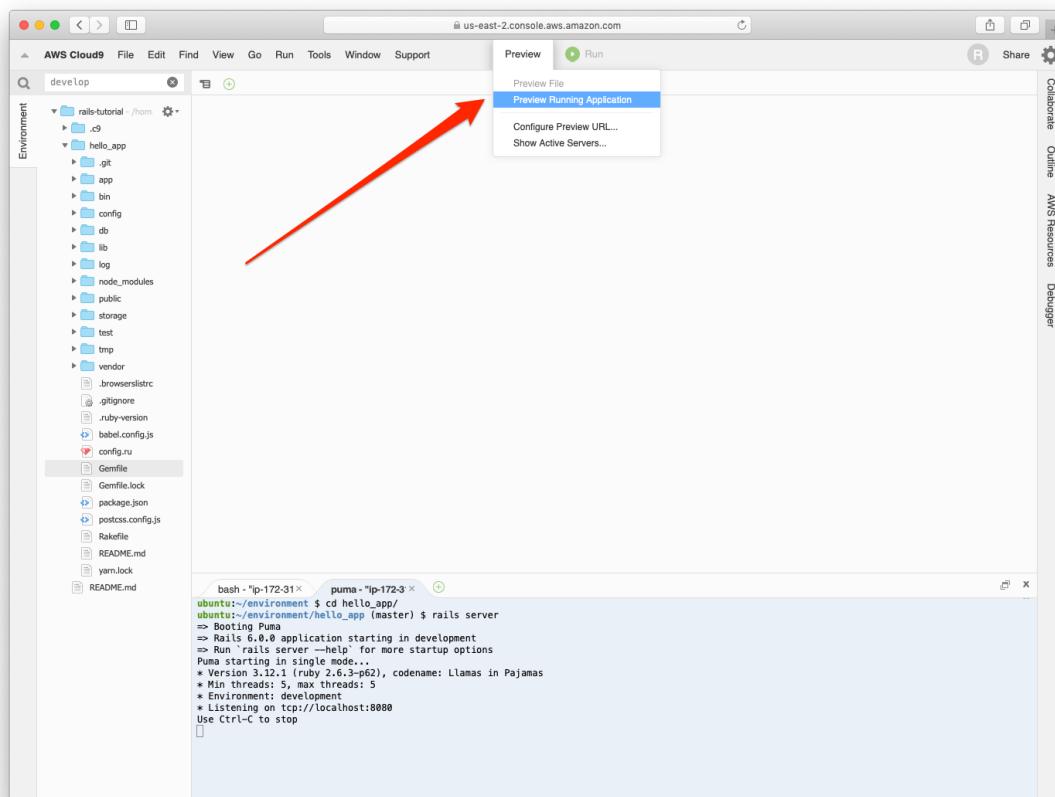


Figure 1.16: Sharing the local server running on the cloud workspace.

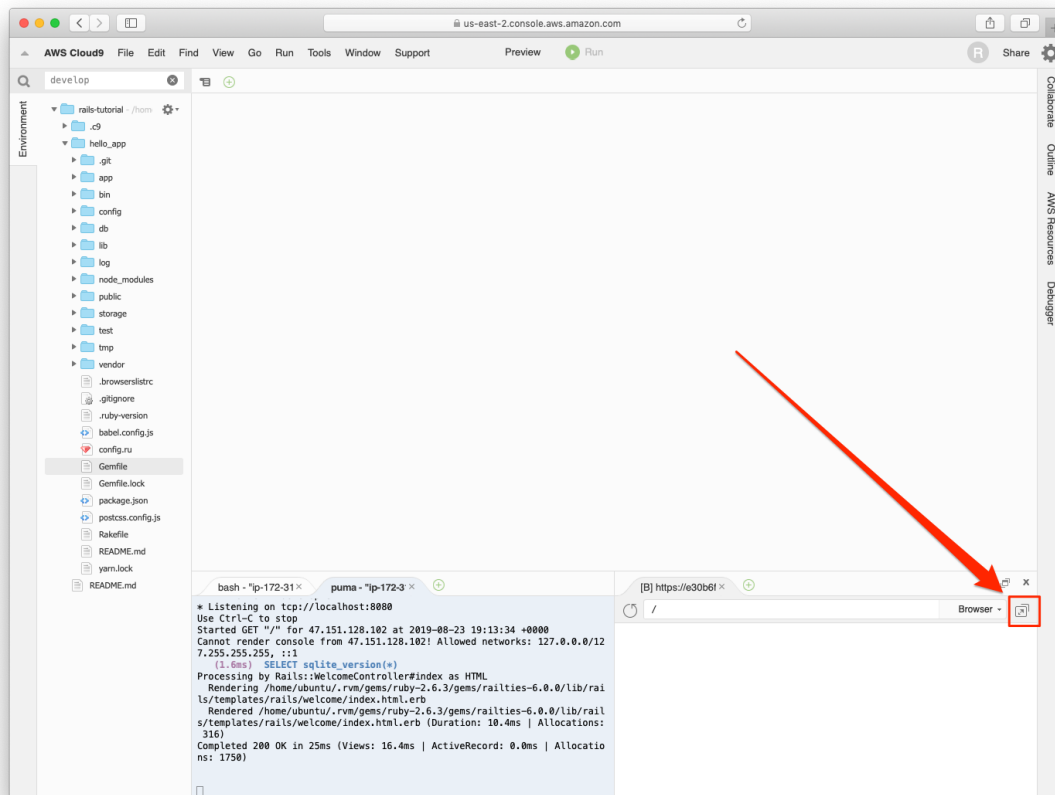


Figure 1.17: Opening the running app in a full browser window or tab.

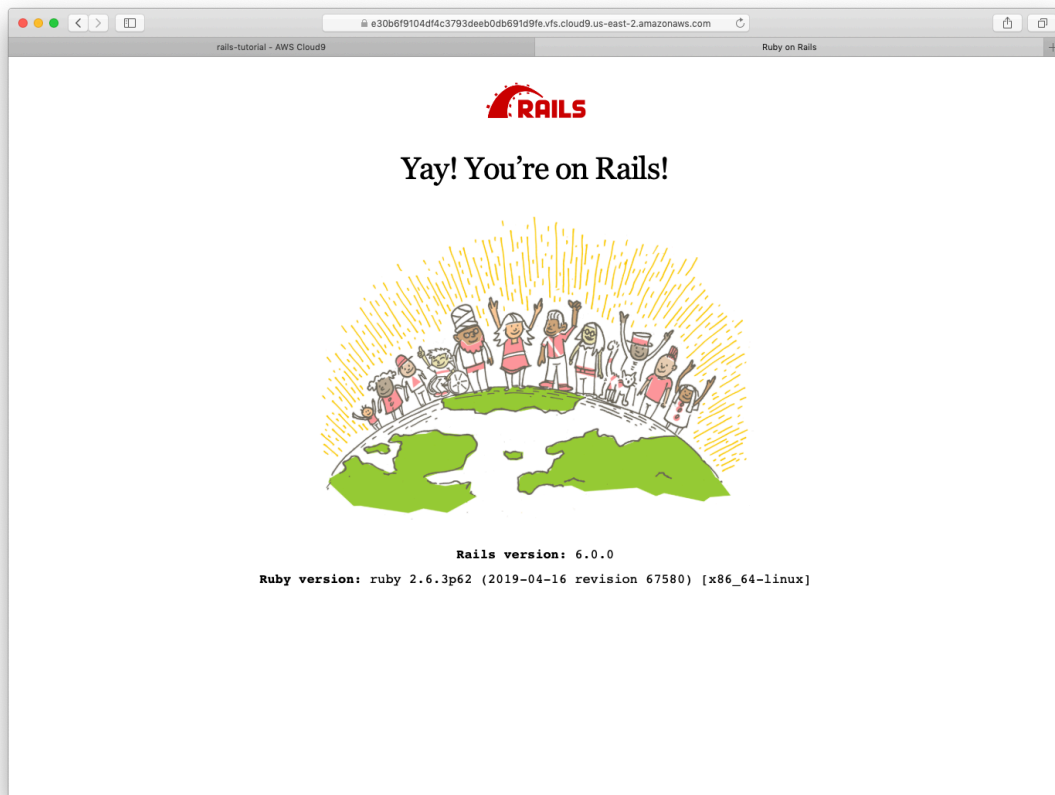


Figure 1.18: The default Rails page served by **rails server**.

Exercises

The [Ruby on Rails Tutorial](#) contains a large number of exercises. Solving them as you proceed through the tutorial is strongly recommended.

In order to keep the main discussion independent of the exercises, the solutions are not generally incorporated into subsequent code listings. (In the rare circumstance that an exercise solution is used subsequently, it is explicitly solved in the main text.) This means that over time your code may diverge from the code shown in the tutorial due to differences introduced in the exercises. Learning how to resolve such discrepancies is a valuable exercise in technical sophistication ([Box 1.2](#)).

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](#).

Many of the exercises are challenging, but we'll start out with some easy ones just to get warmed up:

1. According to the default Rails page, what is the version of Ruby on your system? Confirm by running `ruby -v` at the command line.
2. What is the version of Rails? Confirm that it matches the version installed in [Listing 1.2](#).

1.2.3 Model-View-Controller (MVC)

Even at this early stage, it's helpful to get a high-level overview of how Rails applications work, as illustrated in [Figure 1.19](#). You might have noticed that the standard Rails application structure ([Figure 1.11](#)) has an application directory called `app/`, which includes subdirectories called `models`, `views`, and `controllers` (among others). This is a hint that Rails follows the [model-view-controller](#) (MVC) architectural pattern, which enforces a separation between the data in the application (such as user information) and the code used to display it, which is a common way of structuring a graphical user interface (GUI).

When interacting with a Rails application, a browser sends a *request*, which is received by a webserver and passed on to a Rails *controller*, which is in charge

of what to do next. In some cases, the controller will immediately render a *view*, which is a template that gets converted to HTML and sent back to the browser. More commonly for dynamic sites, the controller interacts with a *model*, which is a Ruby object that represents an element of the site (such as a user) and is in charge of communicating with the database. After invoking the model, the controller then renders the view and returns the complete web page to the browser as HTML.

If this discussion seems a bit abstract right now, don't worry; we'll cover these ideas in more detail later in this book. In particular, [Section 1.2.4](#) shows a first tentative application of MVC, while [Section 2.2.2](#) includes a more detailed discussion of MVC in the context of the toy app. Finally, the full sample app will use all aspects of MVC: we'll cover controllers and views starting in [Section 3.2](#), models starting in [Section 6.1](#), and we'll see all three working together in [Section 7.1.2](#).

1.2.4 Hello, world!

As a first application of the MVC framework, we'll make a [wafer-thin](#) change to the first app by adding a *controller action* to render the string “hello, world!” to replace the default Rails page from [Figure 1.18](#). (We'll learn more about controller actions starting in [Section 2.2.2](#).)

As implied by their name, controller actions are defined inside controllers. We'll call our action **hello** and place it in the Application controller. Indeed, at this point the Application controller is the only controller we have, which you can verify by running

```
$ ls app/controllers/*_controller.rb
```

to view the current controllers. (We'll start creating our own controllers in [Chapter 2](#).) [Listing 1.9](#) shows the resulting definition of **hello**, which uses the **render** function to return the HTML text “hello, world!”. (Don't worry about the Ruby syntax right now; it will be covered in more depth in [Chapter 4](#).)

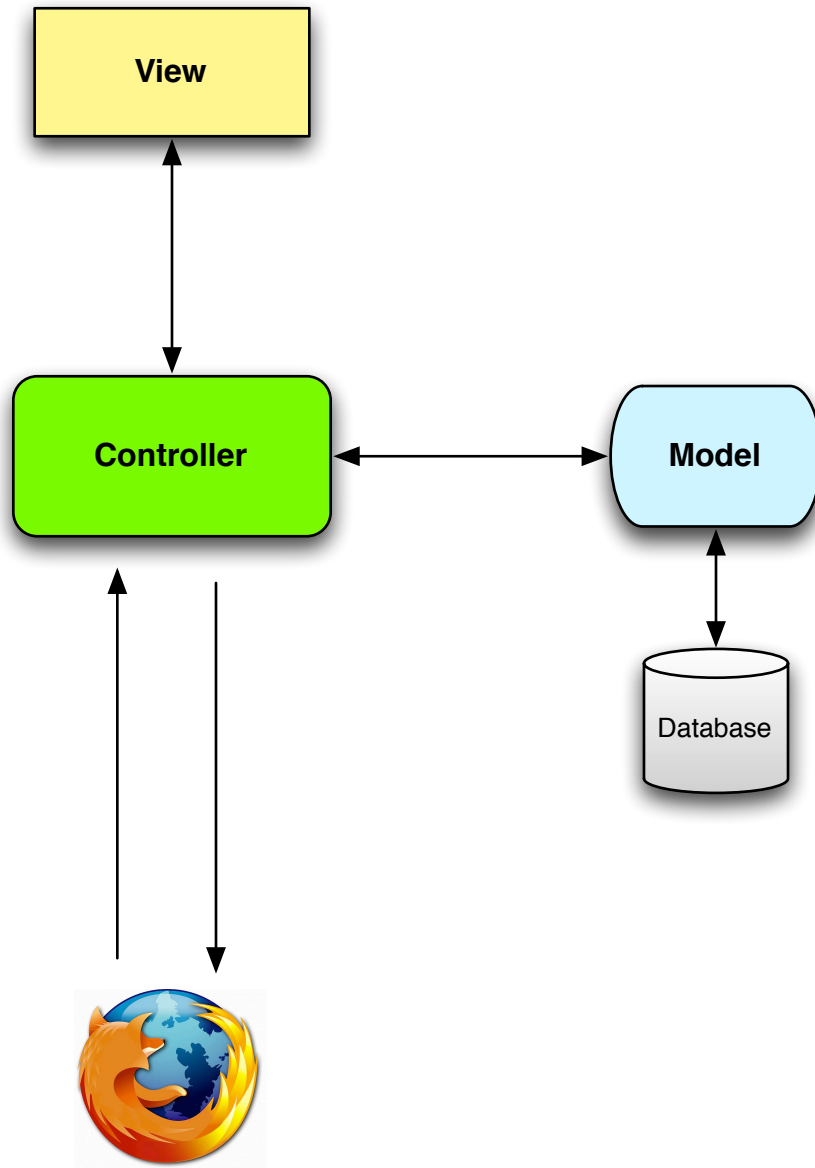


Figure 1.19: A schematic representation of the model-view-controller (MVC) architecture.

Listing 1.9: Adding a **hello** action to the Application controller.

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  def hello
    render html: "hello, world!"
  end
end
```

Having defined an action that returns the desired string, we need to tell Rails to use that action instead of the default page in [Figure 1.18](#). To do this, we'll edit the Rails *router*, which sits in front of the controller in [Figure 1.19](#) and determines where to send requests that come in from the browser. (I've omitted the router from [Figure 1.19](#) for simplicity, but we'll discuss it in more detail starting in [Section 2.2.2](#).) In particular, we want to change the default page, the *root route*, which determines the page that is served on the *root URL*. Because it's the URL for an address like `http://www.example.com/` (where nothing comes after the final forward slash), the root URL is often referred to as `/` ("slash") for short.

As seen in [Listing 1.10](#), the Rails routes file (`config/routes.rb`) includes a comment directing us to the [Rails Guide on Routing](#), which includes instructions on how to define the root route. The syntax looks like this:

```
root 'controller_name#action_name'
```

In the present case, the controller name is **application** and the action name is **hello**, which results in the code shown in [Listing 1.11](#).

Listing 1.10: The default routing file (formatted to fit).

config/routes.rb

```
Rails.application.routes.draw do
  # For details on the DSL available within this file,
  # see https://guides.rubyonrails.org/routing.html
end
```

Listing 1.11: Setting the root route.*config/routes.rb*

```
Rails.application.routes.draw do
  root 'application#hello'
end
```

With the code from [Listing 1.9](#) and [Listing 1.11](#), the root route returns “hello, world!” as required ([Figure 1.20](#)).¹³ Hello, world!

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. Change the content of the **hello** action in [Listing 1.9](#) to read “hola, mundo!” instead of “hello, world!”.
2. Show that Rails supports non-[ASCII](#) characters by including an inverted exclamation point, as in “¡Hola, mundo!” ([Figure 1.21](#)).¹⁴ To get a ¡ character on a Mac, you can use Option-1; otherwise, you can always copy-and-paste the character into your editor.
3. By following the example of the **hello** action in [Listing 1.9](#), add a second action called **goodbye** that renders the text “goodbye, world!”. Edit the routes file from [Listing 1.11](#) so that the root route goes to **goodbye** instead of to **hello** ([Figure 1.22](#)).

¹³The base URL for the Rails Tutorial Cloud9 shared URLs has changed from rails-tutorial-c9-mhartl.c9.io to one on Amazon Web Services, but in many cases the screenshots are identical, so the browser address bar will show old-style URLs in some figures (such as [Figure 1.20](#)). This is the sort of minor discrepancy you can resolve using your technical sophistication ([Box 1.2](#)).

¹⁴Your editor may display a message like “invalid multibyte character”, but this is not a cause for concern. You can [Google the error message](#) if you want to learn how to make it go away.

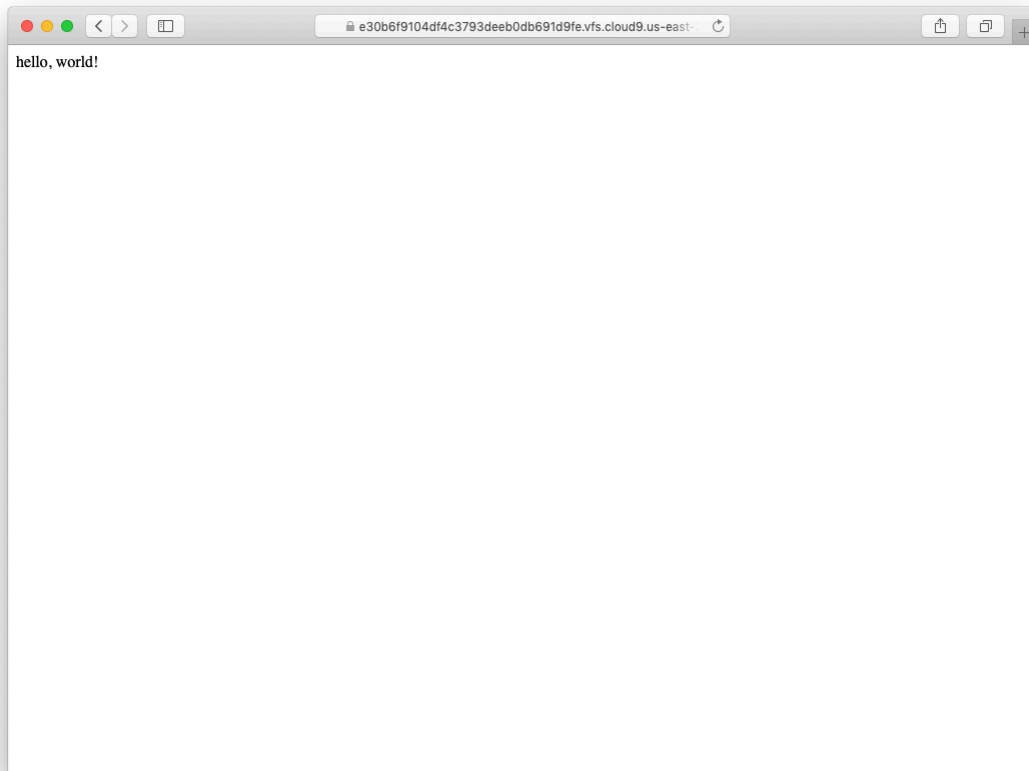


Figure 1.20: Viewing “hello, world!” in the browser.

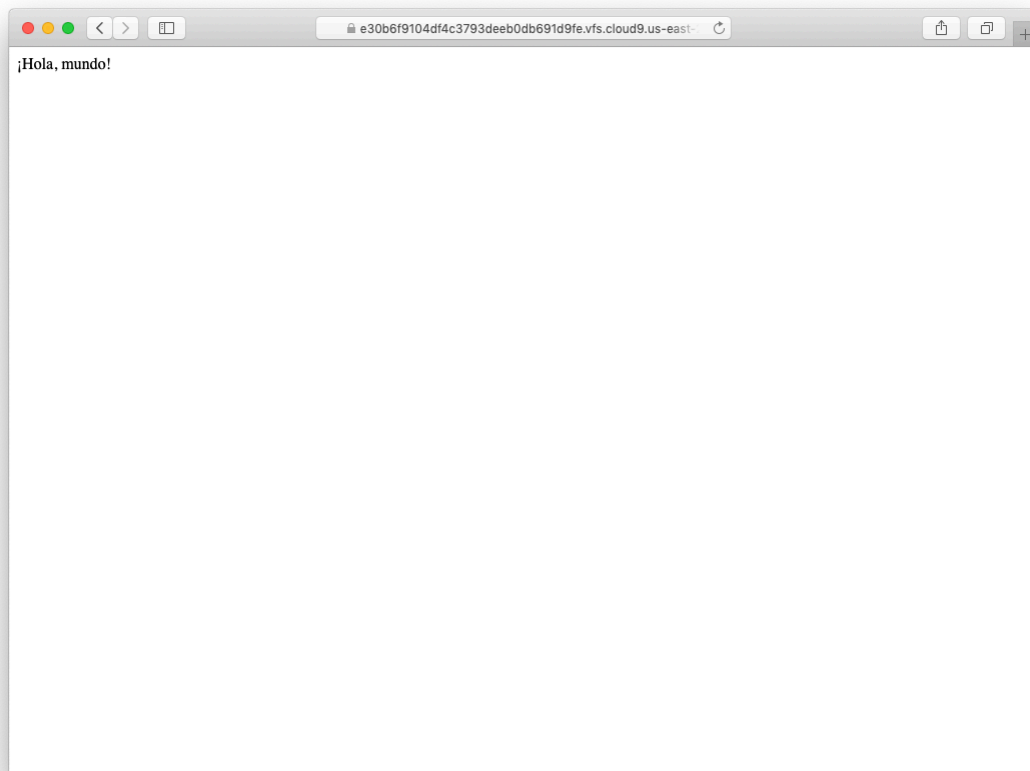


Figure 1.21: Changing the root route to return “¡Hola, mundo!”.

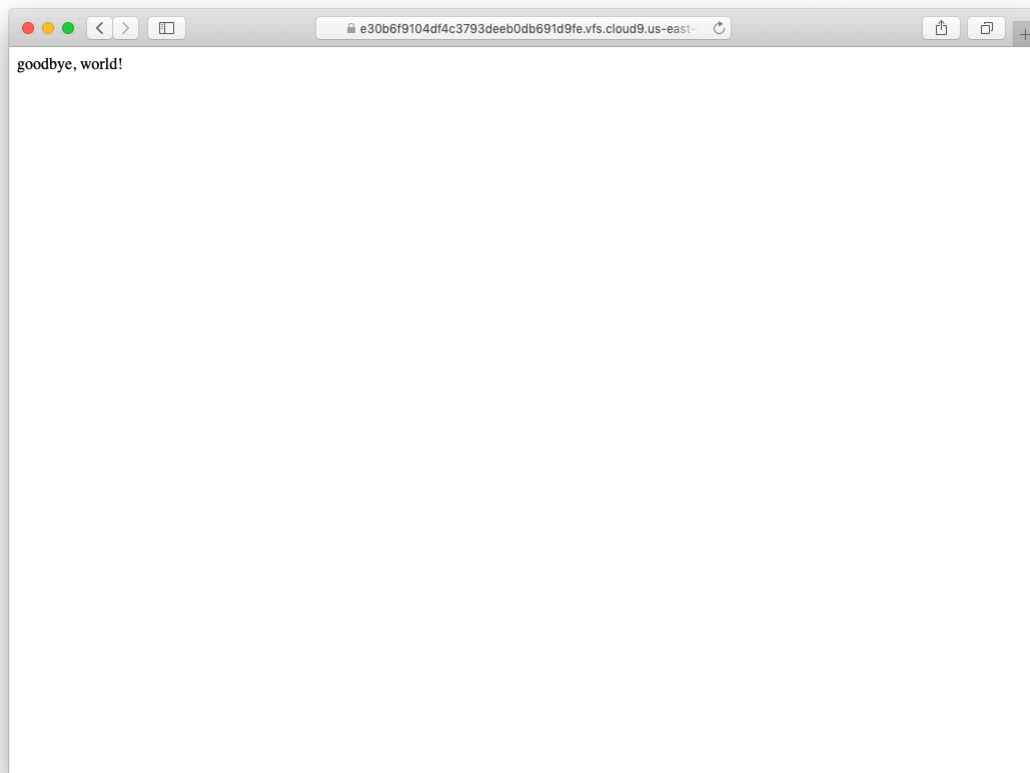


Figure 1.22: Changing the root route to return “goodbye, world!”.

1.3 Version control with Git

Now that we have a working “hello, world” application, we’ll take a moment for a step that, while technically optional, would be viewed by many experienced software developers as practically essential: placing our application source code under *version control*. Version control systems allow us to track changes to our project’s code, collaborate more easily, and roll back any inadvertent errors (such as accidentally deleting files). Knowing how to use a version control system is a required skill for every professional-grade software developer.

There are many options for version control, but the software development community has largely standardized on [Git](#), a distributed version control system originally developed by Linus Torvalds to host the [Linux kernel](#). Git is a large subject, and we’ll only be scratching the surface in this book; for a more thorough introduction to the basics, see [Learn Enough Git to Be Dangerous](#).

Putting your source code under version control with Git is *strongly* recommended, not only because it’s nearly a universal practice in the Rails world, but also because it will allow you to back up and share your code more easily ([Section 1.3.3](#)) and deploy your application right here in the first chapter ([Section 1.4](#)).

1.3.1 Installation and setup

The cloud IDE recommended in [Section 1.1.1](#) includes Git by default, so no installation is necessary in this case. Otherwise, [Learn Enough Git to Be Dangerous](#) includes [instructions](#) for installing Git on your system.

First-time system setup

Before using Git, you should perform a few one-time setup steps. These are *system* setups, meaning you only have to do them once per computer.

The first (and required) step is to configure your name and email address, as shown in [Listing 1.12](#).

Listing 1.12: Configuring the name and email fields for Git.

```
$ git config --global user.name "Your Name"
$ git config --global user.email your.email@example.com
```

Note that the name and email address you use in your Git configuration will be available in any repositories you make public.

If you're using the cloud IDE, the next step is to configure a default editor for the times when Git needs one (such as editing, or “**amending**” changes to projects). We'll use the **nano** editor, which is relatively friendly to beginners and is the default on the cloud IDE. As of this writing, the default editor gets reset on logout, and the path is also incorrect, so we need to execute [Listing 1.13](#), which creates a *symbolic link* (or “symlink”) to the correct location of the **nano** executable.¹⁵ (The command in [Listing 1.13](#) is a little advanced, so certainly don't worry about understanding it if it looks confusing.)

Listing 1.13: Configuring the default editor on the cloud IDE.

```
$ sudo ln -sf `which nano` /usr/bin
```

Next, we'll take an optional but convenient step and set up an *alias*, or synonym, for the commonly used **checkout** command, as shown in [Listing 1.14](#).

Listing 1.14: Setting up **git co** as a checkout alias.

```
$ git config --global alias.co checkout
```

In this tutorial, I'll always use the full **git checkout** command for maximum compatibility, but in practice I almost always use **git co** for short.

The final step is to prevent Git from asking for your password every time you want to use commands like **push** or **pull** ([Section 1.3.4](#)). The options for doing

¹⁵Vim is actually my Git preferred editor in this context, and is recommended for people who have Minimum Viable Vim™ or better (as described in [Learn Enough Text Editor to Be Dangerous](#)). To use **vim** in [Listing 1.13](#), just replace ``which nano`` with ``which vim``

this are system-dependent; see the article “[Caching your GitHub password in Git](#)” if you’re using anything other than Linux (including the cloud IDE). If you are using Linux (including of course the cloud IDE), you can simply set a *cache timeout* as shown in [Listing 1.15](#).

Listing 1.15: Configuring Git to remember passwords for a set length of time.

```
$ git config --global credential.helper "cache --timeout=86400"
```

[Listing 1.15](#) configures Git to remember any passwords you use for 86,400 seconds (one day).¹⁶ If you’re highly security-conscious, you can use a shorter timeout, such as the default 900 seconds, or 15 minutes.

First-time repository setup

Now we come to some steps that are necessary each time you create a new *repository* (sometimes called a *repo* for short). The first step is to navigate to the root directory of the hello app and initialize a new repository:

```
$ cd ~/environment/hello_app # Just in case you weren't already there
$ git init
Reinitialized existing Git repository in
/home/ubuntu/environment/hello_app/.git/
```

Note that Git outputs a message that the repository has been *reinitialized*. This is because, as of Rails 6, running **rails new** ([Listing 1.4](#)) automatically initialize a Git repository (a strong indication of how ubiquitous Git’s use is in tech). Thus, the **git init** step isn’t technically necessary in our case, but this won’t hold for general Git repositories, so always running **git init** is a good habit to cultivate.

The next step is to add all the project files to the repository using **git add -A**.¹⁷

¹⁶In theory, you could use a longer timeout, but on the cloud IDE the timer seems to get reset every day or so, so entering a timeout of more than 86,400 seconds appears to have little effect in this case.

¹⁷Many developers use the [nearly equivalent](#) **git add .**, where **.** (“dot”) represents the current directory. In the rare cases where the two differ, what you usually want is **git add -A**, and this is what’s used in the [official Git documentation](#), so that’s what we go with here.

```
$ git add -A
```

This command adds all the files in the current directory apart from those that match the patterns in a special file called **.gitignore**. The **rails new** command automatically generates a **.gitignore** file appropriate to a Rails project, but you can add additional patterns as well.¹⁸

The added files are initially placed in a *staging area*, which contains pending changes to our project. We can see which files are in the staging area using the **status** command:

```
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

       new file:   .browserslistrc
       new file:   .gitignore
       new file:   .ruby-version
       new file:   Gemfile
       new file:   Gemfile.lock
       .
       .
       .
```

To tell Git we want to keep the changes, we use the **commit** command:

```
$ git commit -m "Initialize repository"
[master (root-commit) df0a62f] Initialize repository
.
.
.
```

The **-m** flag lets us add a *message* for the commit; if we omit **-m**, Git will open the system's default editor and have us enter the message there. (All the examples in this tutorial will use the **-m** flag.)

¹⁸Although we'll never need to edit it in the main tutorial, an example of adding a rule to the **.gitignore** file appears in [Section 3.6.2](#), which is part of the optional advanced testing setup in [Section 3.6](#).

It is important to note that Git commits are *local*, recorded only on the machine on which the commits occur. We'll see how to push the changes up to a remote repository (using `git push`) in [Section 1.3.4](#).

By the way, we can see a list of the commit messages using the `log` command:

```
$ git log
commit b981e5714e4d4a4f518aeca90270843c178b714e (HEAD -> master)
Author: Michael Hartl <michael@michaelhartl.com>
Date:   Sun Aug 18 17:57:06 2019 +0000

    Initialize repository
```

Depending on the length of the repository's log history, you may have to type `q` to quit. (As explained in [Learn Enough Git to Be Dangerous](#), `git log` uses the `less` interface covered in [Learn Enough Command Line to Be Dangerous](#).)

1.3.2 What good does Git do you?

If you've never used version control before, it may not be entirely clear at this point what good it does you, so let's look at just one example. Suppose you've made some accidental changes, such as (D'oh!) deleting the critical `app/controllers/` directory.

```
$ ls app/controllers/
application_controller.rb  concerns/
$ rm -rf app/controllers/
$ ls app/controllers/
ls: app/controllers/: No such file or directory
```

Here we're using the Unix `ls` command to list the contents of the `app/controllers/` directory and the `rm` command to remove it ([Table 1.1](#)). As noted in [Learn Enough Command Line to Be Dangerous](#), the `-rf` flag means "recursive force", which recursively removes all files, directories, subdirectories, and so on, without asking for explicit confirmation of each deletion.

Let's check the status to see what changed:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

       deleted:    app/controllers/application_controller.rb
       deleted:    app/controllers/concerns/.keep

no changes added to commit (use "git add" and/or "git commit -a")
```

We see here that a file has been deleted, but the changes are only on the “working tree”; they haven’t been committed yet. This means we can still undo the changes using the **checkout** command with the **-f** flag to force overwriting the current changes:

```
$ git checkout -f
$ git status
On branch master
nothing to commit, working tree clean
$ ls app/controllers/
application_controller.rb  concerns/
```

The missing files and directories are back. That’s a relief!

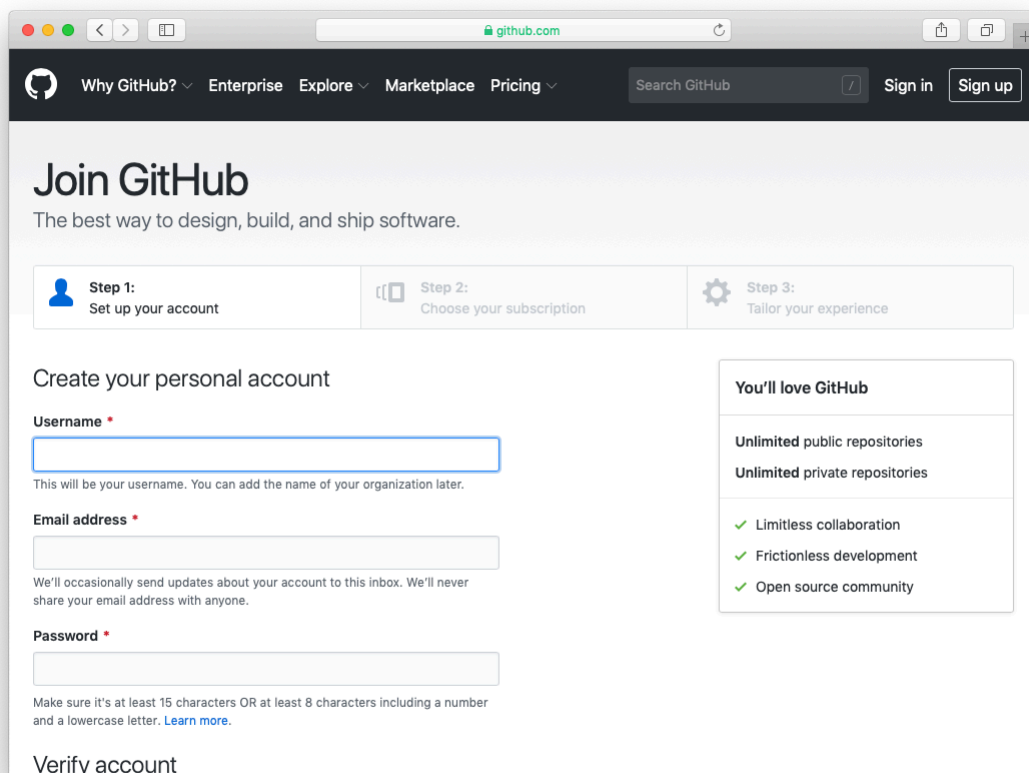
1.3.3 GitHub

Now that we’ve put our project under version control with Git, it’s time to push our code up to [GitHub](#), a site optimized for hosting and sharing Git repositories.¹⁹ Putting a copy of your Git repository at GitHub serves two purposes: it’s a full backup of your code (including the full history of commits), and it makes any future collaboration much easier.

Getting started with GitHub is straightforward: just [sign up for a GitHub account](#) if you don’t already have one (Figure 1.23).

Once you’ve signed up or signed in, click on the + sign dropdown menu and select “New repository” (Figure 1.24).

¹⁹[Bitbucket](#) and [GitLab](#) are also excellent choices. Like GitHub, GitLab is written in Rails.



The screenshot shows the GitHub sign-up page in a browser window. The browser's address bar displays 'github.com'. The navigation bar includes links for 'Why GitHub?', 'Enterprise', 'Explore', 'Marketplace', and 'Pricing', along with a search bar and 'Sign in' and 'Sign up' buttons. The main heading is 'Join GitHub' with the tagline 'The best way to design, build, and ship software.' Below this, a progress bar indicates three steps: 'Step 1: Set up your account', 'Step 2: Choose your subscription', and 'Step 3: Tailor your experience'. The 'Create your personal account' section contains three input fields: 'Username', 'Email address', and 'Password', each with a red asterisk indicating a required field. The 'Username' field has a note: 'This will be your username. You can add the name of your organization later.' The 'Email address' field has a note: 'We'll occasionally send updates about your account to this inbox. We'll never share your email address with anyone.' The 'Password' field has a note: 'Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)' To the right of the form is a box titled 'You'll love GitHub' containing the following text: 'Unlimited public repositories', 'Unlimited private repositories', '✓ Limitless collaboration', '✓ Frictionless development', and '✓ Open source community'. At the bottom of the form is a 'Verify account' link.

Figure 1.23: Signing up for GitHub.

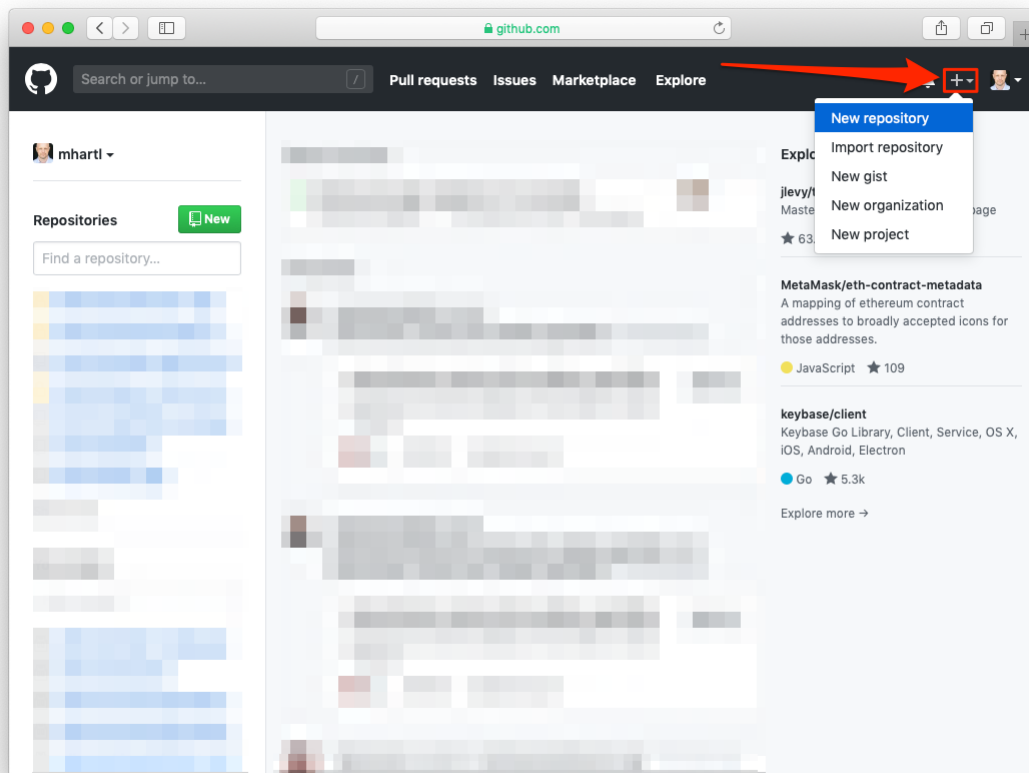


Figure 1.24: Selecting the “New repository” option.

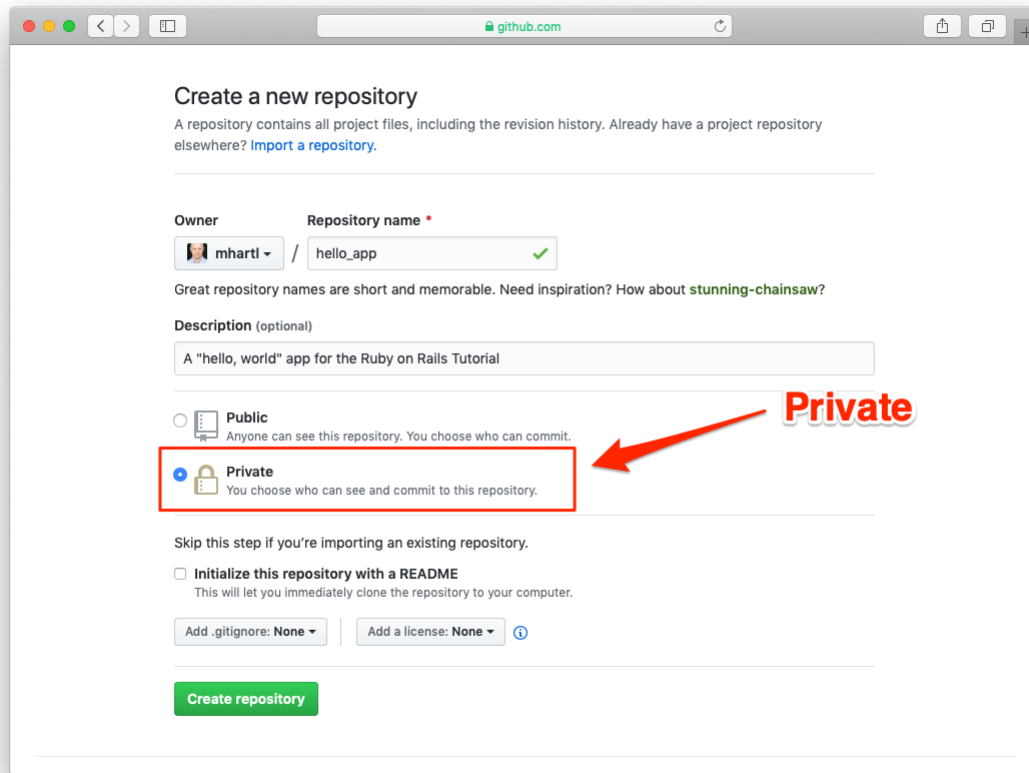


Figure 1.25: Creating a private repository at GitHub.

On the new repository page, fill the fields with the repository name (**hello_app**) and optional description, and take special care to select the “Private” option, as shown in Figure 1.25. Although Rails apps are in principle safe to expose as public repositories, so many things can go wrong (such as accidentally exposing passwords or [private keys](#)) that making all such repositories private is a prudent default.²⁰

After clicking the “Create repository” button, you should see something like Figure 1.26, with commands for adding an existing repository to GitHub.

²⁰GitHub allows unlimited public and private repositories.

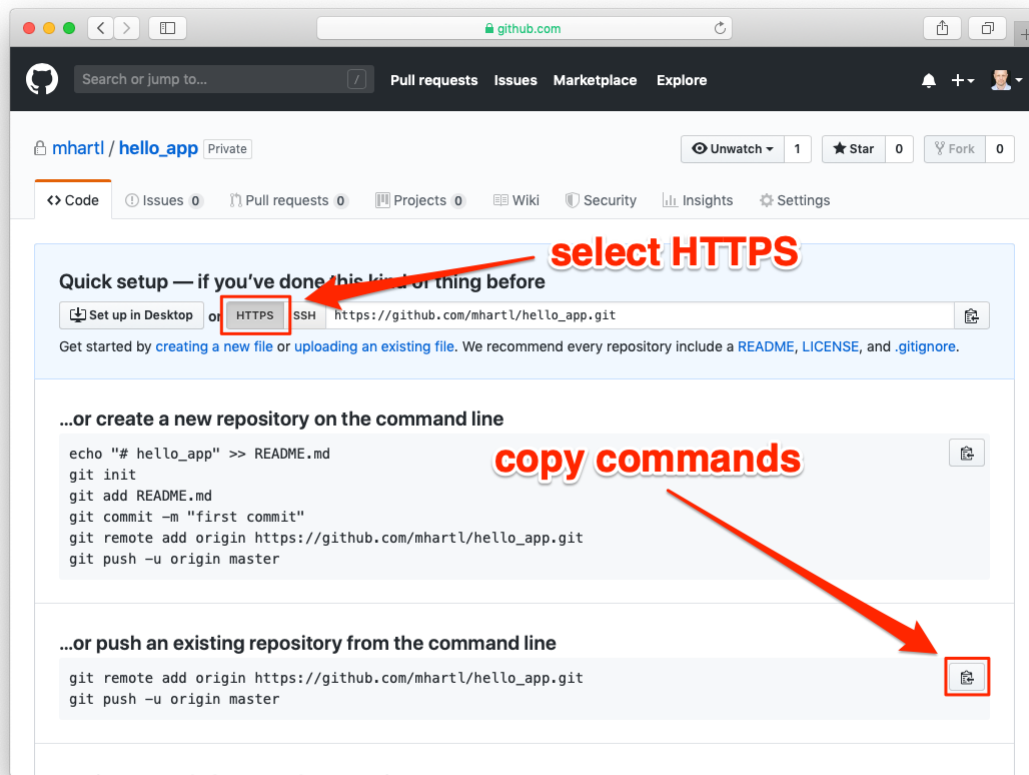


Figure 1.26: Code for adding an existing repository.

Click on the HTTPS option,²¹ and then copy the commands in the section for an existing repository. I suggest clicking the small icon on the right side of the screen, which automatically copies the commands shown in Listing 1.16 to your pasteboard buffer, allowing you to paste them into the command-line terminal.

Finally, run the commands in Listing 1.16. You will have to type your GitHub password, but you won't the next time (as long as it's within the cache timeout period) due to the configuration in Listing 1.15.

²¹The SSH option shown in Figure 1.26 is excellent for more advanced users, so feel free to use it if you're comfortable with generating and configuring SSH keys. Among other things, this option allows your system to cache your password automatically, rendering the setup step in Listing 1.15 unnecessary.

Listing 1.16: Adding GitHub as a remote origin and pushing up the repository.

```
$ git remote add origin https://github.com/<username>/hello_app.git
$ git push -u origin master
```

The commands in [Listing 1.16](#) first tell Git that you want to add GitHub as the *origin* for your repository, and then push your repository up to the remote origin. (Don’t worry about what the `-u` flag does; if you’re curious, do a web search for “[git set upstream](#)”.) Of course, you should replace `<username>` in [Listing 1.16](#) with your actual username. For example, the command I ran looked like this:

```
$ git remote add origin https://github.com/mhartl/hello_app.git
```

The result is a page at GitHub for the `hello_app` repository, with file browsing, full commit history, and lots of other features ([Figure 1.27](#)).

1.3.4 Branch, edit, commit, merge

If you’ve followed the steps in [Section 1.3.3](#), you might notice that GitHub automatically rendered the repository’s README file, as shown in [Figure 1.28](#). This file, called `README.md`, was generated automatically by the command in [Listing 1.4](#). As indicated by the filename extension `.md`, it is written in *Markdown*,²² a human-readable markup language designed to be easy to convert to HTML—which is exactly what GitHub has done.

This automatic rendering of the README is convenient, but of course it would be better if we tailored the contents of the file to the project at hand. In this section, we’ll customize the README by adding some Rails Tutorial-specific content. In the process, we’ll see a first example of the branch, edit, commit, merge workflow that I recommend using with Git.²³

²²See [Learn Enough Text Editor to Be Dangerous](#) and [Learn Enough Git to Be Dangerous](#) for more information about Markdown.

²³For a convenient way to visualize Git repositories, take a look at [Atlassian’s SourceTree app](#).

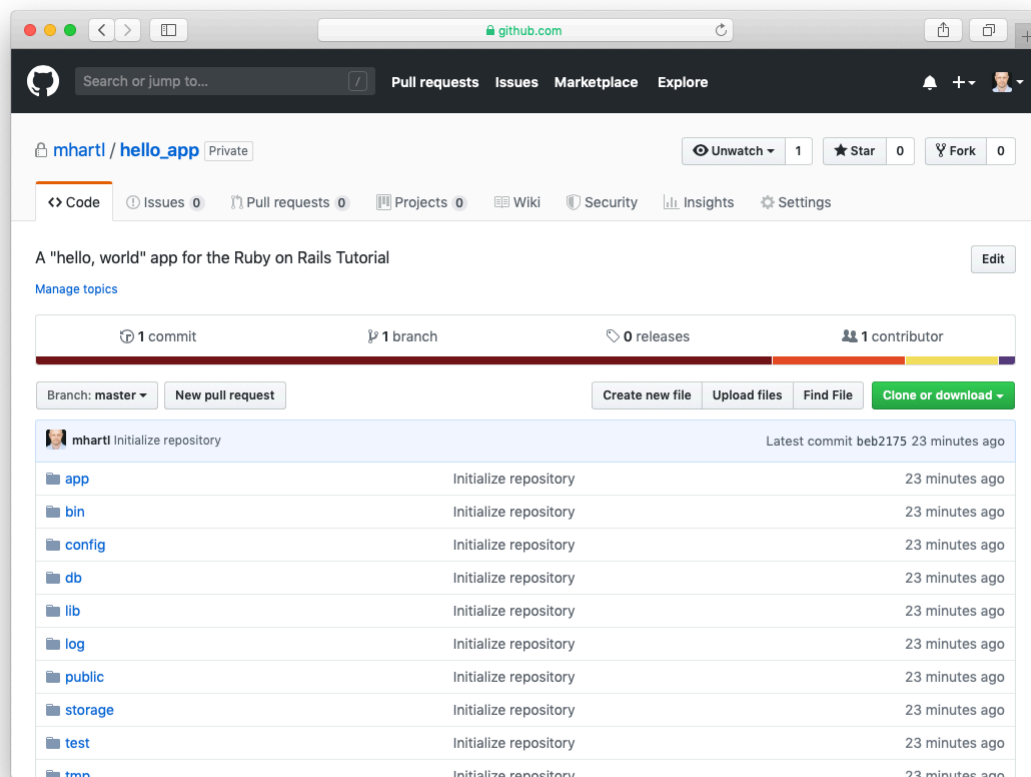


Figure 1.27: A GitHub repository page.

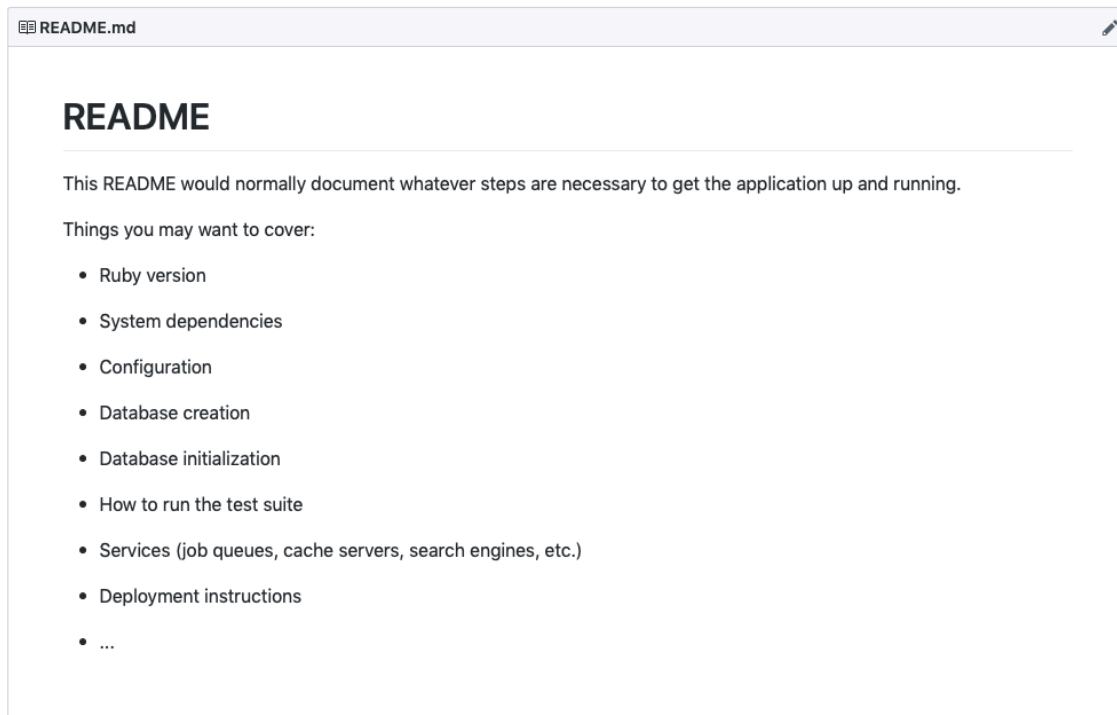


Figure 1.28: GitHub's rendering of the default Rails README.

Branch

Git is incredibly good at making *branches*, which are effectively copies of a repository where we can make (possibly experimental) changes without modifying the parent files. In most cases, the parent repository is the *master* branch, and we can create a new topic branch by using **checkout** with the **-b** flag:

```
$ git checkout -b modify-README
Switched to a new branch 'modify-README'
$ git branch
  master
* modify-README
```

Here the second command, **git branch**, just lists all the local branches, and the asterisk ***** identifies which branch we're currently on. Note that **git checkout -b modify-README** both creates a new branch and switches to it, as indicated by the asterisk in front of the **modify-README** branch.

The full value of branching only becomes clear when working on a project with multiple developers,²⁴ but branches are helpful even for a single-developer tutorial such as this one. In particular, because the master branch is insulated from any changes we make to the topic branch, even if we *really* mess things up we can always abandon the changes by checking out the master branch and deleting the topic branch. We'll see how to do this at the end of the section.

By the way, for a change as small as this one I wouldn't normally bother with a new branch (opting instead to work directly on the master branch), but in the present context it's a prime opportunity to start practicing good habits.

Edit

After creating the topic branch, we'll edit the README to add custom content, as shown in [Listing 1.17](#) and [Figure 1.29](#).

²⁴See, for example, the section on [Collaborating](#) in *Learn Enough Git to Be Dangerous*.

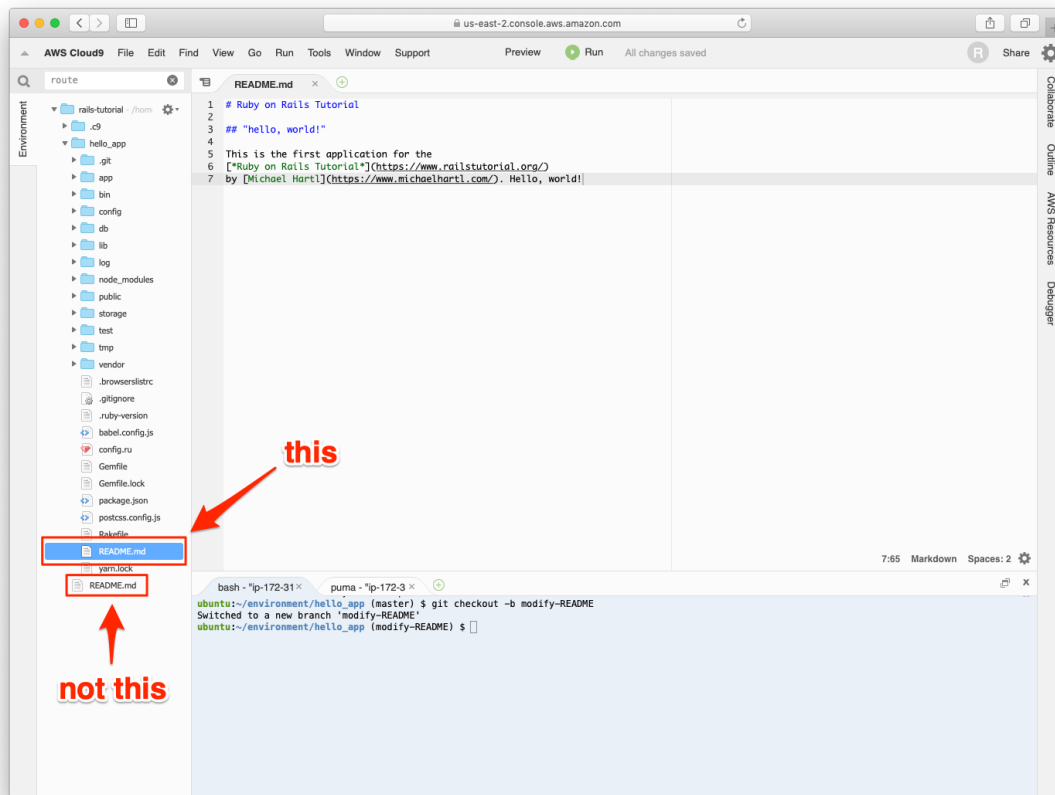


Figure 1.29: Editing the README file.

Listing 1.17: The new **README** file.*README.md*

```
# Ruby on Rails Tutorial

## "hello, world!"

This is the first application for the
[*Ruby on Rails Tutorial*](https://www.railstutorial.org/)
by [Michael Hartl](https://www.michaelhartl.com/). Hello, world!
```


Commit

With the changes made, we can take a look at the status of our branch:

```
$ git status
On branch modify-README
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

At this point, we could use `git add -A` as in [Section 1.3.1](#), but `git commit` provides the `-a` flag as a shortcut for the (very common) case of committing all modifications to existing files:

```
$ git commit -a -m "Improve the README file"
[modify-README 34bb6a5] Improve the README file
1 file changed, 5 insertions(+), 22 deletions(-)
```

Be careful about using the `-a` flag improperly; if you have added any new files to the project since the last commit, you still have to tell Git about them using `git add -A` first.

Note that we write the commit message in the *present* tense (and, technically speaking, the *imperative mood*). Git models commits as a series of patches, and in this context it makes sense to describe what each commit *does*, rather than what it did. Moreover, this usage matches up with the commit messages generated by Git commands themselves. See [Committing to Git](#) from *Learn Enough Git to Be Dangerous* for more information.

Merge

Now that we've finished making our changes, we're ready to *merge* the results back into our master branch:

```
$ git checkout master
Switched to branch 'master'
$ git merge modify-README
Updating b981e57..015008c
Fast-forward
 README.md | 27 +++++-----
 1 file changed, 5 insertions(+), 22 deletions(-)
```

Note that the Git output frequently includes things like **34f06b7**, which are related to Git's internal representation of repositories. Your exact results will differ in these details, but otherwise should essentially match the output shown above.

After you've merged in the changes, you can tidy up your branches by deleting the topic branch using **git branch -d** if you're done with it:

```
$ git branch -d modify-README
Deleted branch modify-README (was 015008c).
```

This step is optional, and in fact it's quite common to leave the topic branch intact. This way you can switch back and forth between the topic and master branches, merging in changes every time you reach a natural stopping point.

As mentioned above, it's also possible to abandon your topic branch changes, in this case with **git branch -D**:

```
# For illustration only; don't do this unless you mess up a branch
$ git checkout -b topic-branch
$ <really mess up the branch>
$ git add -A
$ git commit -a -m "Make major mistake"
$ git checkout master
$ git branch -D topic-branch
```

Unlike the **-d** flag, the **-D** flag will delete the branch even though we haven't merged in the changes.

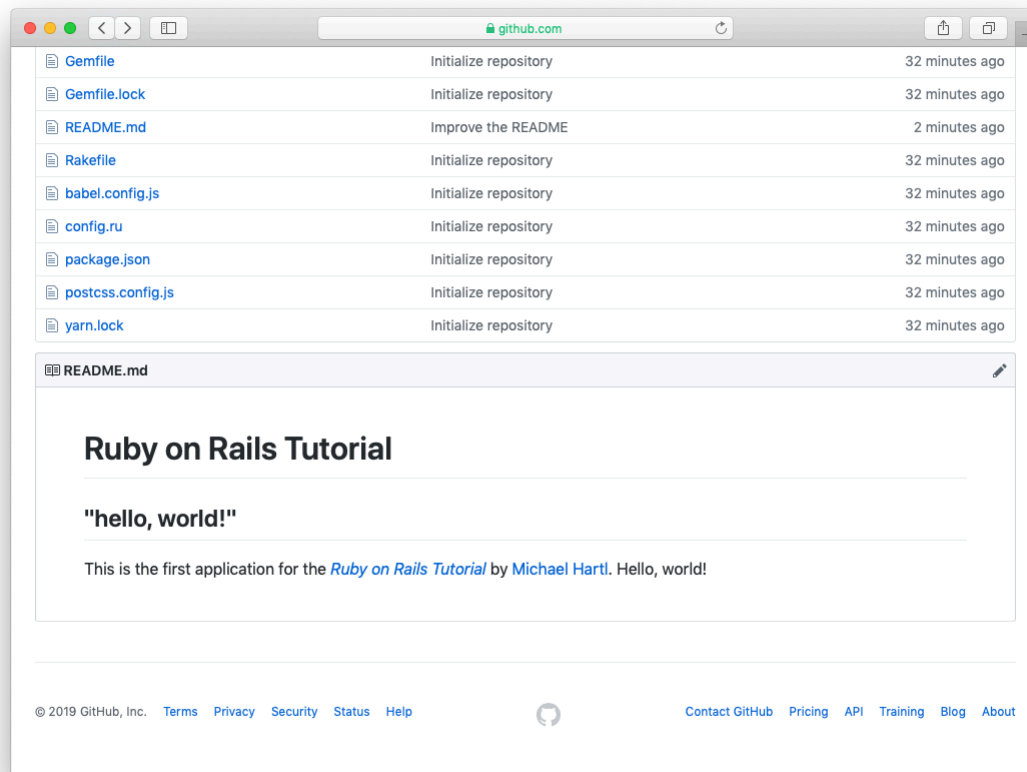


Figure 1.30: The improved **README** file at GitHub.

Push

Now that we've updated the **README**, we can push the changes up to GitHub to see the result. Since we have already done one push (Section 1.3.3), on most systems we can omit **origin master**, and simply run **git push**:

```
$ git push
```

As with the default **README**, GitHub nicely converts the Markdown in our updated **README** to HTML (Figure 1.30).

1.4 Deploying

Even though this is only the first chapter, we’re already going to deploy our Rails application to production! As with the version control setup in [Section 1.3](#), this step is technically optional, but deploying early and often allows us to catch any deployment problems early in our development cycle. The alternative—deploying only after laborious effort sealed away in a development environment—often leads to terrible integration headaches when launch time comes.²⁵

Deploying Rails applications used to be a pain, but the Rails deployment ecosystem has matured rapidly in the past few years, and now there are several great options. These include shared hosts or virtual private servers running [Phusion Passenger](#) (a module for the Apache and Nginx²⁶ web servers), full-service deployment companies such as [Engine Yard](#) and [Rails Machine](#), and cloud deployment services such as [Engine Yard Cloud](#) and [Heroku](#).

My favorite Rails deployment option is Heroku, which is a hosted platform built specifically for deploying Rails and other web applications. (As you might guess, Heroku itself is written in Rails.) Heroku makes deploying Rails applications ridiculously easy, as long as your source code is under version control with Git—which is yet another reason to follow the Git setup steps in [Section 1.3](#) if you haven’t already. In addition, for many purposes, including for this tutorial, Heroku’s free tier is more than sufficient.

The rest of this section is dedicated to deploying our first application to Heroku. Some of the ideas are fairly advanced, so don’t worry about understanding all the details; what’s important is that by the end of the process we’ll have deployed our application to the live web.

1.4.1 Heroku setup and deployment

Heroku uses the [PostgreSQL](#) database (pronounced “post-gres-cue-ell”, and often called “Postgres” for short), which means that we need to add the `pg` gem

²⁵Though it shouldn’t matter for the example applications in the *Rails Tutorial*, if you’re worried about accidentally making your app public too soon there are several options; see [Section 1.4.3](#) for one.

²⁶Pronounced “Engine X”.

in the production environment to allow Rails to talk to Postgres:

```
group :production do
  gem 'pg', '1.1.4'
end
```

Also be sure to incorporate the changes made in [Listing 1.6](#) preventing the `sqlite3` gem from being included in a production environment, since the [SQLite database](#) isn't supported at Heroku.²⁷

```
group :development, :test do
  gem 'sqlite3', '1.4.1'
  gem 'byebug', '11.0.1', platforms: [:mri, :mingw, :x64_mingw]
end
```

The resulting **Gemfile** appears as in [Listing 1.18](#).

Important note: For all the Gemfiles in this book, you should use the version numbers listed at gemfiles-6th-ed.railstutorial.org instead of the ones listed below (although they should be identical if you are reading this online).

Listing 1.18: A **Gemfile** with added and rearranged gems.

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',      '6.0.1'
gem 'puma',       '3.12.1'
gem 'sass-rails', '5.1.0'
gem 'webpacker', '4.0.7'
gem 'turboinks',  '5.2.0'
gem 'jbuilder',   '2.9.1'
gem 'bootsnap',   '1.4.4', require: false

group :development, :test do
  gem 'sqlite3', '1.4.1'
  gem 'byebug',  '11.0.1', platforms: [:mri, :mingw, :x64_mingw]
end
```

²⁷SQLite is widely used as an embedded database—for instance, it's ubiquitous in mobile phones—and Rails uses it locally by default because it's so easy to set up, but it isn't designed for database-backed web applications. See [Section 3.1](#) for more information.

```
end

group :development do
  gem 'web-console',      '4.0.1'
  gem 'listen',          '3.1.5'
  gem 'spring',          '2.1.0'
  gem 'spring-watcher-listen', '2.0.1'
end

group :test do
  gem 'capybara',        '3.28.0'
  gem 'selenium-webdriver', '3.142.4'
  gem 'webdrivers',      '4.1.2'
end

group :production do
  gem 'pg', '1.1.4'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

To prepare the system for deployment to production, we run **bundle install** with a special flag to prevent the local installation of any production gems (which in this case consists of the `pg` gem), as shown in [Listing 1.19](#).

Listing 1.19: Bundling without production gems.

```
$ bundle install --without production
```

Because the only gem added in [Listing 1.18](#) is restricted to a production environment, right now the command in [Listing 1.19](#) doesn't actually install any additional local gems, but it's needed to update **Gemfile.lock** with the `pg` gem. We can commit the resulting change as follows:

```
$ git commit -a -m "Update Gemfile for Heroku"
```

Next we have to create and configure a new Heroku account. The first step is to [sign up for Heroku](#). Then check to see if your system already has the Heroku command-line client installed:

```
$ heroku --version
```

This will display the current version number if the **heroku** command-line interface (CLI) is available, but on most systems it will be necessary to install the [Heroku CLI](#) by hand.²⁸ In particular, if you're working on the cloud IDE, you can install Heroku using the command shown in [Listing 1.20](#).

Listing 1.20: The command to install Heroku on the cloud IDE.

```
$ source <(curl -sL https://cdn.learnenough.com/heroku_install)
```

After running the command in [Listing 1.20](#), you should now be able to verify the installation by displaying the current version number (details may vary):

```
$ heroku --version
heroku/7.27.1 linux-x64 node-v11.14.0
```

Once you've verified that the Heroku command-line interface is installed, use the **heroku** command to log in with the mail address and password you used when signing up (the **--interactive** option prevents **heroku** from trying to spawn a browser):

```
$ heroku login --interactive
```

Finally, use the **heroku create** command to create a place on the Heroku servers for the sample app to live ([Listing 1.21](#)).

Listing 1.21: Creating a new application at Heroku.

```
$ heroku create
Creating app... done, ⬢ blooming-bayou-75897
https://blooming-bayou-75897.herokuapp.com/ |
https://git.heroku.com/blooming-bayou-75897.git
```

²⁸toolbelt.heroku.com

The **heroku** command creates a new subdomain just for our application, available for immediate viewing. There’s nothing there yet, though, so let’s get busy deploying.

1.4.2 Heroku deployment, step one

At this point, we’re ready to deploy to Heroku.

Step 1

The first step is to use Git to push the master branch up to Heroku:

```
$ git push heroku master
```

(You may see some warning messages, which you should ignore for now. We’ll discuss them further in [Section 7.5](#).)

Step 2

There is no step two! We’re already done. To see your newly deployed application, visit the address that you saw when you ran **heroku create** (i.e., [Listing 1.21](#)).²⁹ The result appears in [Figure 1.31](#). The page is identical to [Figure 1.20](#), but now it’s running in a production environment on the live web.³⁰

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. By making the same change as in [Section 1.2.4](#), arrange for your production app to display “hola, mundo!”.

²⁹If you’re working on your local machine instead of the cloud IDE, you can use **heroku open** to open the site automatically in a web browser.

³⁰Your results may differ if you completed the exercises in [Section 1.2.4](#).

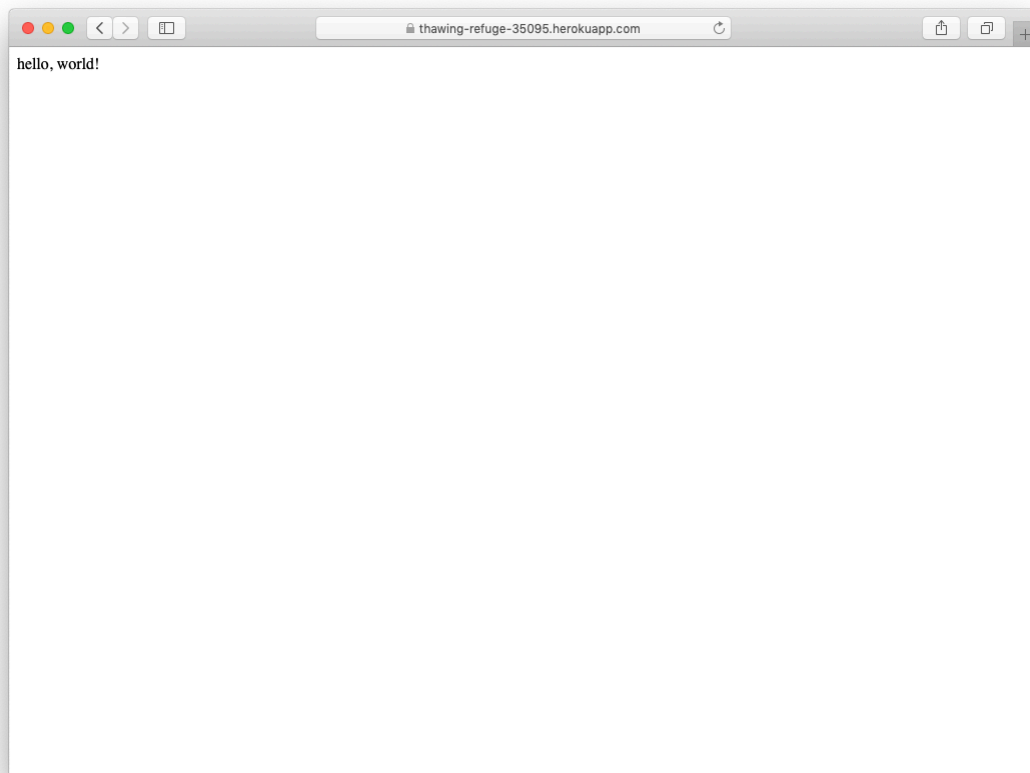


Figure 1.31: The first Rails Tutorial application running on Heroku.

2. As in [Section 1.2.4](#), arrange for the root route to display the result of the **goodbye** action. When deploying, confirm that you can omit **master** in the Git push, as in **git push heroku**.

1.4.3 Heroku commands

There are many [Heroku commands](#), and we'll barely scratch the surface in this book. Let's take a minute to show just one of them by renaming the application as follows:

```
$ heroku rename rails-tutorial-hello
```

Don't use this name yourself; it's already taken by me! In fact, you probably shouldn't bother with this step right now; using the default address supplied by Heroku is fine. But if you do want to rename your application, you can arrange for it to be reasonably secure by using a random or obscure subdomain, such as the following:

```
hwpcbmze.herokuapp.com  
seyjhflo.herokuapp.com  
jhyicevg.herokuapp.com
```

With a random subdomain like this, someone could visit your site only if you gave them the address.³¹ (By the way, as a preview of Ruby's compact awesomeness, here's the code I used to generate the random subdomains:

```
('a'..'z').to_a.shuffle[0..7].join
```

³¹This solution, known as "security through obscurity", is fine for hobby projects, but for sites that require greater initial security I recommend using [Rails HTTP basic authentication](#). This is a much more advanced technique, though, and requires significantly more technical sophistication ([Box 1.2](#)) to implement. (Thanks to Alfie Pates for raising this issue.)

We'll return to this bit of code in Chapter 4.)³²

In addition to supporting subdomains, Heroku also supports custom domains. (In fact, the [Ruby on Rails Tutorial site](#) lives at Heroku; if you're reading this book online, you're looking at a Heroku-hosted site right now!) See the [Heroku documentation](#) for more information about custom domains and other Heroku topics.

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. Run `heroku help` to see a list of Heroku commands. What is the command to display logs for an app?
2. Use the command identified in the previous exercise to inspect the activity on your application. What was the most recent event? (This command is often useful when debugging production apps.)

1.5 Conclusion

We've come a long way in this chapter: development environment setup, installation, version control, and deployment. In the next chapter, we'll build on the foundation from Chapter 1 to make a database-backed *toy app*, which will give us our first real taste of what Rails can do.

If you'd like to share your progress at this point, feel free to send a tweet or Facebook status update with something like this:

[I'm learning Ruby on Rails with the @railstutorial!](#)
<https://www.railstutorial.org/>

³²As is often the case, this code can be made even *more* compact using a built-in part of Ruby, in this case something called `sample: ('a'..'z').to_a.sample(8).join`. Thanks to alert reader Stefan Pochmann for pointing this out—I didn't even know about `sample` until he told me!

I also recommend signing up for the [Rails Tutorial email list](#)³³, which will ensure that you receive priority updates (and exclusive coupon codes) regarding the *Ruby on Rails Tutorial*.

1.5.1 What we learned in this chapter

- Ruby on Rails is a web development framework written in the Ruby programming language.
- Installing Rails, generating an application, and editing the resulting files is easy using a pre-configured cloud environment.
- Rails comes with a command-line command called `rails` that can generate new applications (`rails new`) and run local servers (`rails server`).
- We added a controller action and modified the root route to create a “hello, world” application.
- We protected against data loss while enabling collaboration by placing our application source code under version control with Git and pushing the resulting code to a private repository at GitHub.
- We deployed our application to a production environment using Heroku.

1.6 Conventions used in this book

The conventions used in this book are mostly self-explanatory. In this section, we’ll go over some that may not be.

This tutorial makes frequent use of [command-line](#) commands. For simplicity, all command line examples use a Unix-style command-line prompt (a dollar sign), as follows:

³³railstutorial.org/email

```
$ echo "hello, world"
hello, world
```

Rails comes with many commands that can be run at the command line. For example, in [Section 1.2.2](#) we'll run a local development webserver with the **rails server** command:

```
$ rails server
```

As with the command-line prompt, the *Rails Tutorial* uses the Unix convention for directory separators (i.e., a forward slash `/`). For example, the sample application **production.rb** configuration file appears as follows:

```
config/environments/production.rb
```

This *file path* should be understood as being relative to the application's root directory, which will vary by system. For example, on the cloud IDE ([Section 1.1.1](#)) it looks like this:

```
/home/ubuntu/environment/sample_app/
```

Thus, the full path to **production.rb** is

```
/home/ubuntu/environment/sample_app/config/environments/production.rb
```

I will typically omit the application path and write just **config/environments/production.rb** for short.

The *Rails Tutorial* often shows output from various programs. Because of the innumerable small differences between different computer systems, the output you see may not always agree exactly with what is shown in the text, but this is not cause for concern. In addition, some commands may produce errors depending on your system; rather than attempt the [Sisyphean](#) task of documenting

all such errors in this tutorial, I will delegate to the “Google the error message” algorithm, which among other things is good practice for real-life software development ([Box 1.2](#)). If you run into any problems while following the tutorial, I suggest consulting the resources listed at the [Rails Tutorial Help page](#).³⁴

Because the *Rails Tutorial* covers testing of Rails applications, it is often helpful to know if a particular piece of code causes the test suite to fail (indicated by the color red) or pass (indicated by the color green). For convenience, code resulting in a failing test is thus indicated with **RED**, while code resulting in a passing test is indicated with **GREEN**.

Finally, for convenience the *Ruby on Rails Tutorial* adopts two conventions designed to make the many code samples easier to understand. First, some code listings include one or more highlighted lines, as seen below:

```
class User < ApplicationRecord
  validates :name, presence: true
  validates :email, presence: true
end
```

Such highlighted lines typically indicate the most important new code in the given sample, and often (though not always) represent the difference between the present code listing and previous listings. Second, for brevity and simplicity many of the book’s code listings include vertical dots, as follows:

```
class User < ApplicationRecord
  .
  .
  .
  has_secure_password
end
```

These dots represent omitted code and should not be copied literally.

³⁴<https://www.railstutorial.org/help>