learn-rails.com

What Is Ruby on Rails

A tutorial by Daniel Kehoe

Contents

Chapter 1

About RailsApps

The best way to learn is by doing; when it comes to code, that means building applications.

The RailsApps project provides open source example applications for Rails developers, for free. Our work is supported by subscribers who want access to our in-depth tutorials.

Our tutorials take you on a guided path starting with absolute basics. With each tutorial you will gain knowledge as you build a real-world Rails application. As you build, you'll feel genuine satisfaction at each step. Hands-on learning with real Rails applications is the key to absorbing and retaining knowledge.

Monthly Subscription

The RailsApps project is supported by a monthly subscription. When you subscribe, you get:

- the book *Learn Ruby on Rails*
- the book *Rails and Bootstrap*
- additional beginner and intermediate-level tutorials

In addition you'll support development of:

- new example applications
- more tutorials
- tools to create starter applications
- a growing community resource

I invite you to join the RailsApps project and support our work for the Rails community.

You can join the RailsApps project and get access to all the tutorials for only $19 per month. You'll need a credit card to sign up. Your subscription starts immediately and you can cancel anytime.

For your subscription, sign up here:

- Join RailsApps

Thank you! I appreciate your support for our work.

Chapter 2

Concepts

This chapter provides the background, or "big picture," you will need to understand Rails.

This chapter is excerpted from the book Learn Ruby on Rails by Daniel Kehoe. Read the book for a complete introduction.

Here are the key concepts you'll need to know before you try to use Rails.

How the Web Works

We start with absolute basics, as promised.

When you "visit a website on the Internet" you use a *web browser* such as Safari, Chrome, Firefox, or Internet Explorer.

Web browsers are *applications* (software programs) that work by reading *files*.

Compare a *word processing program* with a *web browser*. Both word processing programs and web browsers read files. Microsoft Word reads files that are stored on your computer to display documents. A web browser retrieves files from remote computers called *servers* to display web pages. Knowing that everything comes from files will help you build a web application.

A web browser uses four kinds of files to display web pages:

- HTML – *structure* (layout) and *content* (text)
- CSS – *stylesheets* to set visual appearance
- JavaScript – *programming* to alter the page
- Images – plus video or other multimedia

At a minimum, a web page requires an HTML file. If a web browser receives just an HTML file, it will display text, with default styles applied by the browser.

If the page is always the same, every time it is displayed by the web browser, we say it is *static*. Webmasters don't need software such as Rails to deliver static documents; they just create files for delivery by an ordinary *web server* program.

Static websites are ideal for particle physics papers (which was the original use of the World Wide Web). But most sites on the web, especially those that allow a user to sign in, post comments, or order products and services, generate web pages *dynamically*.

Dynamic websites often combine web pages with information from a database. A database stores information such as a user's name, comments, Facebook likes, advertisements, or any other repetitive, structured data. A database *query* can provide a selection of data that customizes a webpage for a particular user or changes the web page so it varies with each visit.

Dynamic websites use a programming language such as Ruby to assemble HTML, CSS, and JavaScript files on the fly from component files or a database. A software program written in Ruby and organized using the Rails *development framework* is a Rails *web application*. A web server program that runs Rails applications to generate dynamic web pages is an *application server* (but usually we just call it a web server).

Software such as Rails can access a database, combining the results of a database query with static content to be delivered to a web browser as HTML, CSS, and JavaScript files. Keep in mind that the web browser only receives ordinary HTML, CSS, and JavaScript files; the files themselves are assembled dynamically by the Rails application running on the server.

Even if you are not going to use a database, there are other good reasons to generate a website using a programming language. For example, if you are creating several web pages, it often makes sense to assemble an HTML file from smaller components. For example, you might make a small file that will be included on every page to make a footer (Rails calls these "partials"). Just as importantly, if you are using Rails, you can add features to your website with code that has been developed and tested by other people so you don't have to build everything yourself.

The widespread practice of sharing code with other developers for free, and collaborating with strangers to build applications or tools, is known as *open source* software development. Rails is at the heart of a vibrant open source development community, which means you leverage the work of tens of thousands of skilled developers when you build a Rails application. When Ruby code is packaged up for others to share, the package is called a *gem*. The name is apt because shared code is valuable.

Ruby is a programming language; Rails is a development framework. That means Rails is a set of *structures and conventions* for building a web application using the Ruby language. Rails is also a *library* or collection of *gems* that developers use as the core of any Rails web application. By using Rails, you get well-tested code that implements many of the most-needed features of a dynamic website.

With Rails, you will be using shared standard practices that make it easier to collaborate with others and maintain your application. As an example, consider the code that is used to access a database. Using Ruby without the Rails framework, or using another language such as PHP, you could mix the complex programming code that accesses the database with the code that generates HTML. With the insight of years of developers' collective experience in maintaining and debugging such code, Rails provides a library of code that segregates

database access from the code that displays pages, enforcing *separation of concerns*, and making more modular, maintainable programs.

In a nutshell, that's how the web works, and why Rails is useful.

For a history of Rails, and an explanation of why it is popular, see the article What is Ruby on Rails?

JavaScript and Ruby

JavaScript and Ruby are both general-purpose programming languages.

Ruby is the programming language you'll use when creating web applications that run on your local computer or a remote server using the Rails web application development framework.

JavaScript is the programming language that controls every web browser. The companies that build web browsers (Google, Apple, Microsoft, Mozilla, and others) agreed to use JavaScript as the standard browser programming language. You might imagine an alternative universe in which Ruby was the browser programming language; then you would only have to learn one language for front-end and back-end programming. That's not the real world; plus it would be boring, as learning more than one language makes us smarter and better programmers.

Though most of the code in Rails applications is written in Ruby, developers add JavaScript to Rails applications to implement features such as browser-based visual effects and user interaction.

There is another universe where JavaScript is used on servers to run web applications. System administrators can install the Node.js code library to enable servers to run JavaScript. Server-side JavaScript web application frameworks are available, such as Express and Meteor, but none are as popular as Ruby on Rails.

What is Rails?

So far, I've defined Rails in two ways: as *structures and conventions* for building a web application, and as a *library* or collection of code.

To really understand Rails, and succeed in building Rails applications, we need to consider Rails from six other perspectives. Like six blind men encountering an elephant, it can be difficult to understand Rails unless you look at it from multiple points of view.

Here are six different ways of looking at Rails, summarized from the article What is Ruby on Rails?

From the **perspective of the web browser**, Rails is simply a program that generates HTML, CSS, and JavaScript files. These files are generated dynamically. You can't see the files on the server side but you can view these files by using the web developer tools that are built in to every browser. Later you'll examine these files when you learn to troubleshoot a Rails application.

From the **perspective of a programmer**, Rails is a set of files organized with a specific structure. The structure is the same for every Rails application; this commonality is what makes it easy to collaborate with other Rails developers. We use text editors to edit these files to make a web application.

From the **perspective of a software architect**, Rails is a structure of *abstractions* that enable programmers to collaborate and organize their code. Thinking in abstractions means we group things in categories and analyze relationships. Conceptual categories and relationships can be made "real" in code. Software programs are built of "concepts made real" that are the moving parts of a software machine. To a software architect, *classes* are the basic parts of a software machine. A class can represent something in the physical world as a collection of various attributes or properties (for example, a User with a name, password, and email address). Or a class can describe another abstraction, such as a Number, with attributes such as quantity, and behavior, such as "can be added and subtracted." You'll get a better grasp of classes in the chapter, "Just Enough Ruby."

To a software architect, Rails is a pre-defined set of classes that are organized into a higher level of abstraction known as an API, or *application programming interface*. The Rails API is organized to conform to certain widely known *software design patterns*. You'll become familiar with these abstractions as you build a Rails application. Later in the tutorial, we'll learn about the *model–view–controller* design pattern. As a beginner, you will see the MVC design pattern reflected in the file structure of a Rails application.

We can look at Rails from the **perspective of a gem hunter**. Rails is popular because developers have written and shared many software libraries (RubyGems, or "gems") that provide useful features for building websites. We can think of a Rails application as a collection of gems that provide basic functionality, plus custom code that adds unique features for a particular website. Some gems are required by every Rails application. For example, database adaptors enable Rails to connect to databases. Other gems are used to make development easier, for example, gems for testing that help programmers find bugs. Still other gems add functionality to the website, such as gems for logging in users or processing credit cards. Knowing what gems to use, and why, is an important aspect of learning Rails. This tutorial will show you how to build a web application using some of the most commonly used gems.

We can also look at Rails from the **perspective of a time traveler** in order to understand the importance of *software version control*. Specifically, we use the Git revision control system to record a series of snapshots of your project's filesystem. Git makes it easy to back up and recover files; more importantly, Git lets you make exploratory changes, trying out code you may decide to discard, without disturbing work you've done earlier. You can use Git with GitHub, a popular "social coding" website, for remote backup of your projects and community collaboration. Git can keep multiple versions ("branches") of your local code in

sync with a remote GitHub repository, making it possible to collaborate with others on open source or proprietary projects. Strictly speaking, Git and GitHub are not part of Rails (they are tools that can be used on any development project). And there are several other version control systems that are used in open source development. But a professional Rails developer uses Git and GitHub constantly on any real-world Rails project.

Finally, we can consider a Rails application from the **perspective of a tester**. Software testing is part of Rails culture; Rails is the first web development platform to make testing an integrated part of development. Before Rails, automated testing was rarely part of web development. A web application would be tested by users and (maybe) a QA team. If automated tests were used, the tests were often written after the web application was largely complete. Rails introduced the discipline of Test-Driven Development (TDD) to the wider web development community. With TDD, tests are written *before* any implementation coding. It may seem odd to write tests first, but for a skilled TDD practitioner, it brings coherence to the programming process. First, the developer will give thought to what needs to be accomplished and think through alternatives and edge cases. Second, the developer will have complete test coverage for the project. With good test coverage, it is easier to *refactor*, rearranging code to be more elegant or efficient. Running a test suite after refactoring provides assurance that nothing inadvertently broke after the changes.

TDD is seen as a necessary skill of an experienced Rails developer. Because this is a tutorial for beginners, it *will not* introduce you to techniques of Test-Driven Development. As you work through more advanced tutorials, you'll be introduced to Test-Driven Development.

Stacks

To understand Rails from the perspective of a professional Rails developer, you'll need to grasp the idea of a *technology stack* and recognize that Rails can have more than one stack.

A technology stack is a set of technologies or software libraries that are used to develop an application or deliver web pages. "Stack" is a term that is used loosely and descriptively. There is no organization that sets the rules about what goes into a stack. As a technologist, your choice of stack reflects your experience, values, and personal preference, just like religion or favorite beverage.

For example, Mark Zuckerberg developed Facebook in 2004 using the LAMP application stack:

- Linux (operating system)
- Apache (web server)
- MySQL (database)
- PHP (programming language)

For this tutorial, your application stack will be:

- Mac OS X, Linux, or Windows
- WEBrick (web server)
- SQLite (database)
- Ruby on Rails (language and framework)

Sometimes when we talk about a stack, we only care about part of a larger stack. For example, a Rails stack includes the gems we choose to add features to a website or make development easier. When we select the gems we'll use for a Rails application, we're choosing a stack.

Sometimes the choice of components is driven by the requirements of an application. At other times, the stack is a matter of personal preference. Just as craftsmen and aficionados debate the merits of favorite tools and techniques in any profession, Rails developers avidly dispute what's the best Rails stack for development.

The company 37signals, where the creator of Rails works, uses this Rails stack:

- ERB for view templates
- MySQL for databases
- MiniTest for testing

It is not important (at this point) to know what the acronyms mean (we'll learn later).

Another stack is more popular among Rails developers:

- Haml for view templates
- PostgreSQL for databases
- RSpec for testing

We'll learn later what the terms mean. For now, just recognize that parts of the Rails framework can be swapped out, just like making substitutions when you order from a menu at a restaurant.

You can learn much about Rails by following the experts' debates about the merits of a favorite stack. The debates are a source of much innovation and improvement for the Rails framework. In the end, the power of the crowd prevails; usually the best components in the Rails stack are the most popular.

The proliferation of choices for the Rails stack can make learning difficult, particularly because the components used by many leading Rails developers are not the components used in many beginner tutorials. In this tutorial, we stick to solid ground where there is no debate. In more advanced tutorials, we'll explore stack choices and choose components that are most often used by professional developers.