railsapps.github.io

What is Ruby on Rails? · RailsApps

by Daniel Kehoe

Last updated 11 October 2013

In-depth primer on Ruby and Rails, “What is Ruby on Rails?” Explains what is important about Ruby and Rails.

This article introduces basic concepts of web development, explains why Ruby on Rails is a popular framework for web development, and looks at Rails from several perspectives so you’ll be prepared to learn more about Rails. You don’t need a technical background to read this article. Technical terms are introduced and explained in easy-to-understand language. If you are an experienced web developer, you’ll get an introduction to the specifics of Rails, with fundamental concepts explained systematically and comprehensively.

Get the Book

New to Rails? Get the book Learn Ruby on Rails.

Ruby

Ruby is a programming language. It was created 20 years ago by Yukihiro “Matz” Matsumoto. By most measures of programming language popularity, Ruby ranks among the top ten, though usually as tenth (or so) in popularity, and largely due to the popularity of Rails. Like Java or the C language, Ruby is a general-purpose programming language, though it is best known for its use in web programming.

Rails

Rails is a software library that extends the Ruby programming language. David Heinemeier Hansson is its creator. He gave it the name “Ruby on Rails,” though it is often just called “Rails.”

It is software code that is added to the Ruby programming language. Technically, it is a package library (specifically, a RubyGem), that is installed using the operating system command-line interface.

Rails is a framework for building websites. As such, Rails establishes conventions for easier collaboration and maintenance. These conventions are codified as the Rails API (the application programming interface, or directives that control the code). The Rails API is documented online and described in books, articles, and blog posts. Learning Rails means learning how to use the Rails conventions and its API.

Rails combines the Ruby programming language with HTML, CSS, and JavaScript to create a web application that runs on a web server. Because it runs on a web server, Rails is considered a server-side, or “back end,” web application development platform (the web browser is the “front end”). Later, this article will describe web applications in greater depth and show why a web development framework is needed to build complex websites.

Rails, in a larger sense, is more than a software library and an API. Rails is the central project of a vast community that produces software libraries that simplify the task of building complex websites. Members of the Rails community share many core values, often use the same tools, and support each other with an informal network that is built on volunteerism. Overlapping the informal community is an economic network that includes jobs, recruiters, consulting firms, conferences, businesses that build websites with Rails, and investors that fund startups. Rails is popular among web startups, significantly because the pool of open source software libraries (RubyGems, or "gems") makes it possible to build complex sites quickly.

Read on for a more detailed look at Rails. First, we'll look at why Rails uses the Ruby programming language. We'll consider why Rails is popular, review some basic concepts, then look at Rails from six different perspectives to understand it better.

Why Ruby?

In a podcast from This Developer's Life and in an interview from 2005, David Heinemeier Hansson, the creator of Rails, describes building an online project management application named BaseCamp in 2004. He had been using the PHP programming language because he could get things done quickly but was frustrated because of a lack of abstraction and frequently repetitive code that made PHP "dirty." Hansson wanted to use the "clean" software enginering abstractions supported in the Java programming language but found development in Java was cumbersome. He tried Ruby and was excited about the ease of use (he calls it pleasure) he found in the Ruby language.

Ruby is known among programmers for a terse, uncluttered syntax that doesn't require a lot of extra punctuation. Compared to Java, Ruby is streamlined, with less code required to create basic structures such as data fields. Ruby is a modern language that makes it easy to use high-level abstractions such as metaprogramming. In particular, metaprogramming makes it easy to develop a "domain specific language" that customizes Ruby for a particular set of uses (Rails and many gems use this "DSL" capability).

Ruby's key advantage is RubyGems, the package manager that makes it easy to create and share software libraries (gems) that extend Ruby. RubyGems provides a simple system to install gems. Anyone can upload a gem to the central RubyGems website, making the gem immediately available for installation by anyone. The RubyGems website is where you'll obtain the most recent version of Rails. And it is where you will obtain all the gems that help you build complex websites.

Ruby has several disadvantages (at least when programmers want to argue). Its processing performance is slow relative to C++ or Java. The execution speed of a language is seldom important, though, relative to the benefits gained by programmer productivity and the general level of performance required by most websites. For websites that require lots of simultaneous activity, Ruby is not well-suited to the sophisticated software engineering required to execute simultaneous activity efficiently (standard Ruby lacks "parallelism", though some versions support it). Lastly, some programmers complain that Ruby programs (and especially Rails) contain "too much magic" (that is, complex operations that are hidden behind simple directives). These concerns haven't stopped Rails from becoming a popular web development platform.

Do You Need to Study Ruby to Learn Rails?

Let's digress for a moment. If you are reading this because you want to learn Rails, you may want to know if you must learn the Ruby programming language before learning Rails.

The short answer is "no," with one caveat. To avoid feeing overwhelmed when you first begin learning Rails, it is advisable to spend at least an hour with an introduction to Ruby so you are comfortable with the syntax of the language. You should be prepared to recognize correct formatting when you type Ruby code in your text editor. You can find several good online tutorials listed on the Ruby and Rails resource page. Get started with a quick lesson from RubyMonk or Try Ruby.

Be assured that you will indeed learn Ruby as you develop proficiency with Rails. Whether you study Ruby or not, you'll develop Ruby skills as you learn Rails. Rails is largely a "domain specific language" that has its own set of directives distinct from the Ruby core. As you learn the Rails "language" you'll be using the Ruby language syntax.

Your hardest challenge will be to learn the names of the structures you see in code examples. This is why it is helpful to work your way through a short introduction to Ruby. You'll need to be able to recognize when you are looking at an array or a hash. You should recognize when you are looking at an iterator or the Ruby block syntax. Eventually, you'll recognize more exotic Ruby formulations such as the lambda. It is okay if you can't write a lambda function or even know when to use one; many Rails developers start work before learning Ruby thoroughly.

As experienced Rails developers, we hope you will make an effort to learn Ruby as you learn Rails. Don't be lazy; when you encounter a bit of Ruby you don't understand, make an effort to find out what is going on. Spend time with a Ruby textbook or interactive course as you get more proficient with Rails. By all means, if you love the precision and order of programming languages, dive into the study of Ruby from the beginning. But don't delay starting Rails while you learn Ruby; realistically, you'll retain more knowledge of Ruby if you learn it as you build things in Rails.

Why Rails?

Rails is popular and widely used because its conventions are pervasive and astute. Any web application has complex requirements that include basic functions such as generating HTML, processing form submissions, or accessing a database. Without a web application development framework, a programmer has a mammoth task to implement all the required infrastructure. Even with a web application development framework, a programmer can take an idiosyncratic approach, building something that no one else can easily take apart and understand. The singular virtue of Rails is that Heinemeier Hansson, and the core team that joined him, decided that there is one best way to implement much of the infrastructure required by a web application. Many of the implementation decisions appear arbitrary. In fact, though Heinemeier Hansson is often lambasted as autocratic in his approach to improving Rails, the Rails API reflects deep experience and intelligence in implementing the requirements of a web application development framework. The benefit is that every developer who learns the "Rails way" produces a web application that any other Rails developer can unravel and understand more quickly than if they encountered idiosyncratic code without as many conventions. That means collaboration is easier, development is quicker, and there's a larger pool of open source libraries to enhance Rails.

The advantage of establishing conventions might seem obvious, but when Rails was released in 2004, web development was dominated by PHP, which lent itself to idiosyncratic code produced by solo webmasters, and Java frameworks such as Struts, which were often seen as burdened by an excess of structure. Other frameworks, such as Apple's WebObjects, Adobe's ColdFusion, and Microsoft's .NET Framework, were in wide use but the frameworks were products controlled by the companies and built by small teams, which tended to restrict innovation. Today PHP, Java frameworks, and .NET remain popular, largely among solo webmasters (PHP), enterprise teams (Java), and Windows aficionados (.NET) but Rails has become very popular and has influenced development of other server-side frameworks.

The design decisions that went into the first version of Rails anchored a virtuous circle that led to Rails's growth. Within the first year, Rails caught the attention of prominent software engineers, notably Martin Fowler and Dave Thomas (proponents of agile software development methodologies). Rails is well-matched to the practices of agile software development, particular in its emphasis on software testing and "convention over configuration." The interest and advocacy of opinion leaders from the agile camp led to greater visibility in the wider open source community, culminating in a keynote lecture by Heinemeier Hansson at the 2005 O'Reilly Open Source Convention. Because Rails was adopted by software engineers who are influencers and trend setters, it is often said that Rails is favored by "the cool kids." If that is so, it is largely because Rails is well-suited to software engineering practices that are promoted by thought leaders like Fowler and Thomas.

Understanding Rails Guiding Principles

The popularity of Rails is an outgrowth of the Rails "philosophy" or guiding principles.

Rails is Opinionated

In the mid-1990s, web applications were often written in Perl, a programming language that promised, "There's more than one way to do it." Perl is a prime example of "non-opinionated" software; there's no "right way" or "best way" to solve programming problems in Perl. Famously, Perl's documentation states, "In general, [Perl's built-in functions] do what you want, unless you want consistency."

In contrast, Rails is said to be "opinionated." There is a "Rails way" for many of the problems that must be solved by a web application developer. If you follow the Rails conventions, you'll have fewer decisions to make and you'll find more of what you need is already built. The benefit is faster development, improved collaboration, and easier maintenance.

Rails is Omakase

Omakase is a Japanese phrase that means "I'll leave it to you." Customers at sushi restaurants can order *omakase*, entrusting the chef to make a pleasing selection instead of making their own à la carte choices. In a famous essay Heinemeier Hansson declared Rails is Omakase, and said, "A team of chefs picked out the ingredients, designed the APIs, and arranged the order of consumption on your behalf according to their idea of what would make for a tasty full-stack framework…. When we, or in some cases I — as the head chef of the omakase experience that is Rails — decide to include a dish, it's usually based on our distilled tastes and preferences. I've worked in this establishment for a decade. I've poured well in the excess of ten thousand hours into Rails. This doesn't make my tastes right for you, but it certainly means that they're well formed."

Understanding that Rails is omakase means accepting that many of the opinions enshrined in the Rails API are the decisions of a Benevolent Dictator for Life, informed by discussion with other developers who have made significant contributions to the Rails code base. For the most part, Heinemeier Hansson's "opinions" will serve you well.

Convention Over Configuration

"Convention over configuration" is an example of Rails as "opinionated software." It is an extension of the concept of a default, a setting or value automatically assigned without user intervention. Some software systems, notably Java web application frameworks, need multiple configuration files, each with many settings. For example, a configuration file might specify that a database table named "sales" corresponds to a class named "Sales." The configuration file permits flexibility (a developer can easily change the setting if the table is named "items_sold"). Instead of relying on extensive configuration files, Rails makes assumptions. By convention, if you create a model object in Rails named

"User," it will save data to a database table named "users" without any configuration required. Rails will also assume the table name is plural if the class name is singular.

"Convention over configuration" means you'll be productive. You won't spend time setting up configuration files. You'll spend less time thinking about where things go and what names to assign. And, because other developers have learned the same conventions, it is easier to collaborate.

Don't Repeat Yourself

Known by the acronym DRY, "Don't Repeat Yourself" is a principle of software development formulated by Andy Hunt and Dave Thomas and widely advocated among Rails developers. In its simplest form, it is an admonition to avoid duplication. When code is duplicated, an application becomes more complex, making it more difficult to maintain and more vulnerable to unintended behavior (bugs). The DRY principle can be extended to development processes as well as code. For example, manual testing is repetititive; automated testing is DRY. Software design patterns that introduce abstraction or indirection can make code more DRY; for example, by eliminating repetitive *if-then* logic.

Code reuse is a fundamental technique in software development. It existed long before Andy Hunt and Dave Thomas promoted the DRY principle. Rails takes advantage of Ruby's metaprogramming features to not just reuse code but eliminate code where possible. With a knowledge of Rails conventions, it's possible to create entire simple web applications with only a few lines of code.

Where Rails Gets Complicated

It helps to understand the guiding principles of Rails. But it's even more helpful to know how (and why) Rails is complicated by departures from the guiding principles.

When Rails has No Opinion

As you gain experience with Rails, you may encounter areas where Rails doesn't state an opinion. For example, as of early 2013, there is no "official" approach to queueing background jobs. (Tasks that take time, such as contacting a remote server, are best handled as "background jobs" that won't delay display of a web page.) Much of the lively debate that drives development of new versions of Rails is focused on thrashing out the "opinions" that eventually will be enshrined in the Rails API.

Omakase But Substitutions Are Allowed

Implicit in the notion of "Rails is omakase" is an understanding that "substitutions are allowed." Most of Heinemeier Hansson's preferences are accepted by all Rails developers. However, many experienced developers substitute items on the menu at the Rails café. This has led to the notion that Rails has Two Default Stacks, as described in an essay by Steve Klabnik. Professional developers often substitute an alternative testing framework or use a different syntax for creating page views than the "official" version chosen by Heinemeier Hansson. This complicates learning because introductory texts often focus on the omakase selections but you'll encounter alternatives in blog posts and example code.

Conventions or Magic?

One of the joys of programming is knowing that everything that happens in an application is explained by the code. If you know where to look, you'll see the source of any behavior. For a skilled programmer, "convention over configuration" adds obscurity. Without a configuration file, there is no obvious code that reveals that data from a class named "Person" is saved to a datatable named "people." As a beginner, you'll simply accept the magic and not confound yourself trying to find how it works. It's not always easy to learn the conventions. For example, you may have a User object and a "users" datatable. Rails will also expect you to create a "controller object." Should it be named "UserController" (singular) or "UsersController" (plural)? You'll only know if you let Rails generate the code or you pay close attention to tutorials and example code.

DRY to Obscurity

The risk that "convention over configuration" leads to obscurity is compounded by the "Don't Repeat Yourself" principle. To avoid repetitive code, Rails often will offer default behavior that looks like magic because the underlying implementation is hidden in the depths of the Rails code library. You can implement a simple web application with only a few lines of custom code but you may wonder where all the behavior comes from. This can be frustrating when, as a beginner, you attempt to customize your application to do more than what's shown in simple tutorials.

How Rails Works

With an understanding in place of the Rails guiding principles and its challenges, let's start from scratch to get a clear picture of how Rails works. We'll start by explaining how a web application is used to create a website. You may already know most of this, but let's review because it will help explain how Rails works.

You're running a web browser on your computer (probably to read this article). A web browser combines three kinds of files—HTML, CSS, and JavaScript—to display web pages. A browser obtains the files from a web server. The web server can be remote (connected by the Internet) or on your own computer (when you are doing development).

You might have learned how to create HTML, CSS, and JavaScript files. HTML (HyperText Markup Language) is a convention for creating structured documents that combine content (such as text, images, or video) with generic typographic, layout and design elements (such as headlines and tables). CSS (Cascading Style Sheets) directives apply a specific appearance to typographic, layout and design elements. JavaScript is a programming language supported by all web browsers that is used to manipulate HTML and CSS elements, particularly for implementing interactive features such as tabs and modal windows. You can learn about HTML, CSS, and JavaScript in a typical "Introduction to Web Design" course that teaches how to make "static websites."

Web servers deliver HTML, CSS, and JavaScript, either from static files that are stored on the server, or from an "application server" that creates files dynamically using a programming language such as Ruby. A software program written in Ruby and organized using Rails conventions is a "web application." Rails combines the Ruby programming language with HTML, CSS, and JavaScript to create a web application. Rails uses Ruby to dynamically assemble HTML, CSS, and JavaScript files from component files (often adding content from a database).

Why create a web application? A web browser needs only a single HTML file to display a web page (CSS and JavaScript files are optional). However, if you are creating several web pages, you might want to assemble the 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"). Consider another example: If you are displaying content from a database, you might not want the complex programming code that accesses the database mixed into your HTML (programmers call this separation of concerns and it makes for more modular, maintainable programs). Finally, if you are using a web application server such as the one supplied with Rails, you can add features to your website that have been developed and tested by other people so you don't have to build everything yourself. These are all reasons to create a web application.

Now that you've seen how Rails combines the Ruby programming language with HTML, CSS, and JavaScript to create a web application, we can examine Rails more closely.

Six Perspectives On a Rails Application

To understand Rails, let's take a look at different aspects of a Rails web application.

Like the blind men encountering the elephant, you'll understand Rails better by examining it from six different perspectives.

Web Browser's Perspective

As we've seen, 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 them on the server side but you can view these files by using the web developer tools that are built in to every browser (choose the menu item "Web Developer Tools" in Google Chrome or "Web Developer Toolbar" in Firefox).

Coder's Perspective

What do you see on your computer if you are looking at a Rails application?

You'll see a set of files that you can edit with your text editor to create your web application. The files are 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. To use Rails, you must learn about this structure and the purpose of each of the folders and files.

Here is how Rails files are organized:

```
+-app | +-assets | | +-images | | +-javascripts | | +-stylesheets | +-controllers | +-helpers |
+-mailers | +-models | +-views +-config +-db +-features +-lib +-log +-public +-script +-spec
```

The Rails file structure exists for the convenience of developers. “Assets” such as images, CSS stylesheet files, and JavaScript files get their own folders. Other folders are a place for configuration files (“config”) and testing files (“features” and “spec”) As you investigate the Rails file structure, you’ll see other folders that don’t have an evident purpose. For example, the folders for “controllers,” “models,” and “views” are important but may not have an obvious function. These folders correspond to a more abstract organizational system, one imposed by software architecture.

From a coder’s perspective, we can say that Rails is a set of files organized in a specific way. We use text editors to edit the files to make a web application. But what’s in the files?

Software Architect’s Perspective

The contents of the files (particularly the files written in the Ruby language) are organized according to a higher level abstraction we call “software architecture.”

Software is purely conceptual; it takes shape as text files but it comes into being in the mind of programmers. Aside from certain features that are required by computer design (programs must read strings of characters from filesystems), a software program can be a conceptual abstraction that is absolutely unique to the mind of its creator. In practice, most programming languages impose a standard set of abstractions that reflect the common needs of most programmers. We keep lists with an abstraction we call an array; we perform an operation on each element of an array with an abstraction we call an iterator. Every programming language documents these basic abstractions in a language reference (for example, see the Ruby API).

In Ruby (as in many other object-oriented languages), an “object” is the fundamental abstraction which is the basis for all other abstractions. Ruby tutorials say, “In Ruby, everything is an object,” and then show that all objects have state (data or “attributes”), behavior (procedures or “methods”), and identity (unique existence among all other objects). Objects are described with a class definition that describes attributes and methods; objects are used by a program as instances that have identical methods but differing data. Ruby provides a standard syntax for defining classes, creating instances, and calling methods.

At a higher level of abstraction, programmers find that code can be often be optimally organized according to patterns that other experienced developers will recognize. Code is easier to write when it fits a pattern you’ve encountered before; it’s easier to analyze and understand code that another programmer has written if it matches a widely known software design pattern. One example is the Model–view–controller pattern that is fundamental to organizing Rails applications. Though the MVC pattern is enshrined in the file structure of a Rails application, it also exists in the form of a hierarchy of Rails classes, notably ActionController (controller), ActionView (view), and ActiveRecord (model), which are Objects that can be subclassed to be used as components of your web application.

From the perspective of a software architect, a Rails web application is organized as a hierarchy of classes defined by the Rails API.

Time Traveler’s Perspective

So far, we’ve seen how a web application is organized from the perspective of a web browser, the filesystem, and software architecture. There’s a temporal perspective that is just as important. Every software development project has this temporal aspect; it’s not unique to Rails, but it’s important to understand if you are going to do development with Rails.

A web application is developed over time; often we make mistakes and need to roll back to a previous version of our work. Computers are data storage machines as much as they are machines for manipulating or communicating data. However, the original concept of a computer file system didn’t allow for travel backward in time; files in disk storage preserve only the most recent version of your work. As programmers, we’ve had to add version control systems (sometimes known as revision control or source control systems) to recover earlier versions of our work.

There are several popular revision control systems; git is used most often by Rails developers. The metaphor of computer folders and files is easy to understand, at least for people who have worked where 20th century office supplies are still used. Git doesn’t offer any similar real-world metaphor which means it is much more difficult to understand how it works. Nonetheless, you must learn to use git if you intend to build more than the simplest Rails application.

Superficially, you can think of git as a series of snapshots of the your project’s filesystem. You can save a snapshot at any time with a “git commit” command in the Terminal. Then you can recover files from that snapshot.

Git is important not just for backing up and recovering files (after all, backup software such as Apple’s Time Machine can do that). You can use git with GitHub for remote backup of your projects. More importantly, git and GitHub are the primary mechanisms for collaborating on Rails application development, whether open source or proprietary projects. You will use git and GitHub constantly on any real-world Rails project.

Strictly speaking, git and GitHub are not part of Rails (they are tools that can be used on any development project). But Rails and the gems that go into a complex web application would not exist without git and GitHub so it's important to recognize the essential role of these tools as part of a Rails project.

From the time traveler's perspective, we can say that a Rails project is a series of snapshots of files stored in a git repository.

Gem Hunter's Perspective

You read earlier that Ruby's key advantage is RubyGems, the package manager that makes it easy to create and share software libraries (gems) that extend Ruby. Rails is popular because developers have written and shared so many 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. In fact, when a developer begins work on an existing Rails web project, he or she will often first investigate what gems are used by the application.

Every Rails application has a Gemfile in the application root directory. The Gemfile lists each gem that is used by the application. The Gemfile is used by a utility program (named Bundler) that adds each gem to the Ruby programming environment. Some gems are required by every Rails application, for example, a database adaptor that allows 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.

The art of selecting gems is one of the skills of an experienced Rails developer. There are gems that are recommended by Rails original creators (notably, David Heinemeier Hansson). Skilled developers often replace these "official" gems with alternative sets of gems that have gained popularity more recently. Knowing what gems to use, or why, is an aspect of learning Rails that is seldom explicit.

As you learn more about Rails, you'll learn about the gems that are most frequently added to Rails projects.

Tester's Perspective

Among development platforms for the web, Rails uniquely includes a "baked-in" test framework. This reflects the platform creator's commitment to a methodology of software development known as *Test Driven Development* (TDD). Though test driven development is optional, not required, for any Rails development project, it is so often used (and expected) that we can't fully understand Rails without considering the tester's perspective.

Software is always tested. Sometimes only by its users (who don't know they are guinea pigs) but more often (and more responsibly), before it is released. Historically, on large corporate software projects, a quality assurance process might involve a team of engineers systematically testing every feature of a software product from the point of view of the user. To the QA team, *integration testing* meant every component was tested to work when assembled together. *Acceptance testing*, which can be the same as integration testing, meant the team evaluated the product to make sure it performed as described in a requirements specification. In this context of large enterprise software projects, QA testing was something of a software engineering specialty and automated testing was part of the practice, since it often was practical to write software programs to test software programs. If a testing regimen is thorough, automated testing will include *unit tests*, small test programs to test discrete parts of a software program, in isolation from the rest of the program, often at the class or method level.

Testing serves several purposes. First, it improves the user experience by finding problems before the product is released to customers. Second, it provides oversight for managers, as a verification that the project has been implemented as specified or promised. However, to a software engineer who is skilled at writing tests, the function of testing in the service of quality assurance is often secondary to testing as a method of programming. Instead of tests simply verifying that what you've written works, tests can become an integral part of the programming process. Rather than writing unit tests after you've written a piece of code, you can write your tests *first*. This is the essence of Test Driven Development. It may seem odd, or time-consuming, but for a skilled TDD practitioner, it brings coherence to the programming process. First, by writing tests before a writing a specific implementation, the developer will give thought to what needs to be accomplished and think through alternatives and edge cases. Second, by writing tests before writing implementation code, the developer will have complete test coverage for the project. Software products typically contain much hidden interdependency. With good test coverage, programmers (especially those on a team) can rip out and replace code to change features and (just as important) can refactor code, rearranging code to be more obvious or more efficient. Running a test suite after refactoring provides assurance that nothing inadvertently breaks after the changes.

Just as writing unit tests can bring coherence to writing portions of a software application, the overall process of developing an application can be improved by writing automated acceptance tests before implementing any parts of a project. Behavior Driven Development (BDD) is similar to Test Driven Development but focuses on writing specifications for a project in the form of descriptive stories that can be the basis for automated acceptance tests.

Before Rails, automated testing was rarely part of web development. A web application would be tested by users and (maybe) a QA team. Rails introduced the discipline of Test Driven Development to the wider web development community. Behavior Driven Development is less commonly used on Rails projects (less so among startups, more frequently among consulting firms and enterprise projects) but TDD is common and seen as a necessary skill of an experienced Rails developer.

Rails Stack

The set of technologies or software libraries that are used to develop an application or deliver web pages is called a technology "stack." For example, Mark Zuckerberg developed Facebook in 2004 using the LAMP stack consisting of Linux (operating system), Apache (web server), MySQL (database), and PHP (programming language). Rails, as a web development framework, and Ruby, as a programming language, can be used with a choice of operating systems (Linux, Mac OS X, Windows), databases (SQLite, MySQL, PostgreSQL, and others), and web servers (Apache, Nginx, and others).

As a beginner, your technology stack will include Ruby on Rails, the WEBrick web server (it comes with Ruby), the SQLite database, and the Linux, Mac OS X, or Windows operating system (whatever is installed on your computer).

The Rails stack can also include a variety of software libraries (gems) that add features to a website or make development easier. Sometimes the choice of components in a stack 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 other profession, Rails developers avidly dispute what's the best Rails stack for development. At times it seems discussion leads to more heat than light.

Sometimes it is not easy to determine what's at stake or even what's being discussed. You'll be challenged as you seek to sort out technical issues from matters of style or preference. But you can learn much about Rails by following these debates. These debates are a source for much innovation and improvement of 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 set of components promoted by David Heinemeier Hansson (the creator of Rails) is not the same set of components used by many leading Rails developers. For example, Heinemeier Hansson recommends the default Test::Unit library for test-driven development; many developers add the RSpec gem instead.

Tutorials for the RailsApps project resolve the debate by using components for the Rails stack that are most often used by the leading developers.

What's Next?

At this point, if you've read the entire article, you've gotten an excellent introduction to Rails. You haven't see any Rails code yet. But you've got the context and background that will help make sense of everything you encounter in Rails.

Get the Book

The book Learn Ruby on Rails is best Rails book for beginners.

Ready to Install Rails?

The Install Rails article from the RailsApps project will show you how to install Rails.

The article doesn't offer shortcuts. It is an installation guide that professional developers use to configure their working environment for real-world Rails development.

Would You Like to Build a Rails Application?

The RailsApps project provides example applications that developers use as starter apps. Hundreds of developers use the apps, report problems as they arise, and propose solutions. Rails changes frequently; each application is known to work and serves as your personal “reference implementation” so you can stay up to date. Each is accompanied by a tutorial so there is no mystery code.

You can use the Rails Composer tool to generate any of the examples as a starter app. Then customize the code for a complete working Rails web application.