[1.2.6 Model-view-controller (MVC)](http://ruby.railstutorial.org/chapters/beginning" \l "sec-mvc)

Even at this early stage, it’s helpful to get a high-level overview of how Rails applications work ([Figure 1.5](http://ruby.railstutorial.org/chapters/beginning#fig-MVC)). You might have noticed that the standard Rails application structure ([Figure 1.2](http://ruby.railstutorial.org/chapters/beginning#fig-directory_structure_rails)) has an application directory called **app/** with three subdirectories: **models**, **views**, and**controllers**. This is a hint that Rails follows the [model-view-controller](http://en.wikipedia.org/wiki/Model-view-controller) (MVC) architectural pattern, which enforces a separation between “domain logic” (also called “business logic”) from the input and presentation logic associated with a graphical user interface (GUI). In the case of web applications, the “domain logic” typically consists of data models for things like users, articles, and products, and the GUI is just a web page in a web browser.

When interacting with a Rails application, a browser sends a *request*, which is received by a web server 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.

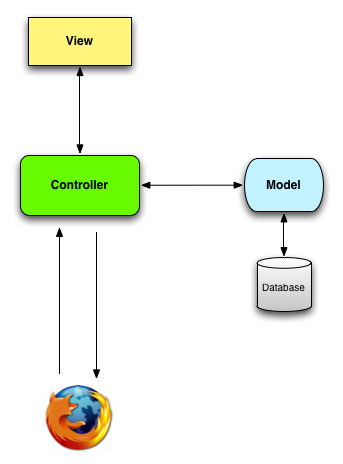


Figure 1.5: A schematic representation of the model-view-controller (MVC) architecture.

If this discussion seems a bit abstract right now, worry not; we’ll refer back to this section frequently. In addition, [Section 2.2.2](http://ruby.railstutorial.org/chapters/a-demo-app#sec-mvc_in_action) has a more detailed discussion of MVC in the context of the demo app. Finally, the sample app will use all aspects of MVC; we’ll cover controllers and views starting in [Section 3.1](http://ruby.railstutorial.org/chapters/static-pages#sec-static_pages), models starting in [Section 6.1](http://ruby.railstutorial.org/chapters/modeling-users#sec-user_model), and we’ll see all three working together in[Section 7.1.2](http://ruby.railstutorial.org/chapters/sign-up#sec-a_users_resource).

[1.3 Version control with Git](http://ruby.railstutorial.org/chapters/beginning#sec-version_control)

Now that we have a fresh and working Rails application, we’ll take a moment for a step that, while technically optional, would be viewed by many Rails developers as practically essential, namely, 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 software developer.

There are many options for version control, but the Rails community has largely standardized on[Git](http://git-scm.com/), 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, but there are many good free resources online; I especially recommend [*Pro Git*](http://git-scm.com/book) by Scott Chacon (Apress, 2009). 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 share your code more easily ([Section 1.3.4](http://ruby.railstutorial.org/chapters/beginning#sec-github)) and deploy your application right here in the first chapter ([Section 1.4](http://ruby.railstutorial.org/chapters/beginning#sec-deploying)).

[1.3.1 Installation and setup](http://ruby.railstutorial.org/chapters/beginning#sec-git_setup)

The first step is to install Git if you haven’t yet followed the steps in [Section 1.2.2.2](http://ruby.railstutorial.org/chapters/beginning#sec-install_git). (As noted in that section, this involves following the instructions in the [Installing Git section of *Pro Git*](http://git-scm.com/book/en/getting-started-installing-git).)

[First-time system setup](http://ruby.railstutorial.org/chapters/beginning#sec-1_3_1_1)

After installing Git, you should perform a set of one-time setup steps. These are *system* setups, meaning you only have to do them once per computer:

**$** git config --global user.name "Your Name"

**$** git config --global user.email your.email@example.com

I also like to use **co** in place of the more verbose **checkout** command, which we can arrange as follows:

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

This tutorial will usually use the full **checkout** command, which works for systems that don’t have**co** configured, but in real life I nearly always use **git co**.

As a final setup step, you can optionally set the editor Git will use for commit messages. If you use a graphical editor such as Sublime Text, TextMate, gVim, or MacVim, you need to use a flag to make sure that the editor stays attached to the shell instead of detaching immediately:[16](http://ruby.railstutorial.org/chapters/beginning" \l "fn-1_16)

**$** git config --global core.editor "subl -w"

Replace **"subl -w"** with **"mate -w"** for TextMate, **"gvim -f"** for gVim, or **"mvim -f"** for MacVim.

[First-time repository setup](http://ruby.railstutorial.org/chapters/beginning#sec-1_3_1_2)

Now we come to some steps that are necessary each time you create a new *repository*. First navigate to the root directory of the first app and initialize a new repository:

**$** git init

Initialized empty Git repository in /Users/mhartl/rails\_projects/first\_app/.git/

The next step is to add the project files to the repository. There’s a minor complication, though: by default Git tracks the changes of *all* the files, but there are some files we don’t want to track. For example, Rails creates log files to record the behavior of the application; these files change frequently, and we don’t want our version control system to have to update them constantly. Git has a simple mechanism to ignore such files: simply include a file called **.gitignore** in the application root directory with some rules telling Git which files to ignore.[17](http://ruby.railstutorial.org/chapters/beginning#fn-1_17)

Looking again at [Table 1.1](http://ruby.railstutorial.org/chapters/beginning#table-rails_directory_structure), we see that the **rails** command creates a default **.gitignore** file in the application root directory, as shown in [Listing 1.6](http://ruby.railstutorial.org/chapters/beginning#code-default_gitignore).

**Listing 1.6.** The default **.gitignore** created by the **rails** command.

# See http://help.github.com/ignore-files/ for more about ignoring files.

#

# If you find yourself ignoring temporary files generated by your text editor

# or operating system, you probably want to add a global ignore instead:

# git config --global core.excludesfile '~/.gitignore\_global'

# Ignore bundler config.

/.bundle

# Ignore the default SQLite database.

/db/\*.sqlite3

/db/\*.sqlite3-journal

# Ignore all logfiles and tempfiles.

/log/\*.log

/tmp

[Listing 1.6](http://ruby.railstutorial.org/chapters/beginning#code-default_gitignore) causes Git to ignore files such as log files, Rails temporary (**tmp**) files, and SQLite databases. (For example, to ignore log files, which live in the **log/** directory, we use **log/\*.log** to ignore all files that end in **.log**.) Most of these ignored files change frequently and automatically, so including them under version control is unnecessary. Moreover, when collaborating with others, these irrelevant changes can cause frustrating conflicts.

The **.gitignore** file in [Listing 1.6](http://ruby.railstutorial.org/chapters/beginning#code-default_gitignore) is a good start, but for convenience and security ([Listing 3.2](http://ruby.railstutorial.org/chapters/static-pages#code-secret_token)) I recommend using [Listing 1.7](http://ruby.railstutorial.org/chapters/beginning#code-gitignore) instead. This augmented **.gitignore** arranges to ignore Rails documentation files, Vim and Emacs swap files, and (for OS X users) the weird **.DS\_Store**directories created by the Mac Finder application. If you want to use this broader set of ignored files, open up **.gitignore** in your favorite text editor and fill it with the contents of [Listing 1.7](http://ruby.railstutorial.org/chapters/beginning#code-gitignore).

**Listing 1.7.** An augmented **.gitignore** file.

# Ignore bundler config.

/.bundle

# Ignore the default SQLite database.

/db/\*.sqlite3

/db/\*.sqlite3-journal

# Ignore all logfiles and tempfiles.

/log/\*.log

/tmp

# Ignore other unneeded files.

database.yml

doc/

\*.swp

\*~

.project

.DS\_Store

.idea

.secret

[1.3.2 Adding and committing](http://ruby.railstutorial.org/chapters/beginning#sec-adding_and_committing)

Finally, we’ll add the files in your new Rails project to Git and then commit the results. You can add all the files (apart from those that match the ignore patterns in **.gitignore**) as follows:

**$** git add .

Here the dot ‘**.**’ represents the current directory, and Git is smart enough to add the files*recursively*, so it automatically includes all the subdirectories. This command adds the project files to a *staging area*, which contains pending changes to your project; you can see which files are in the staging area using the **status** command:[18](http://ruby.railstutorial.org/chapters/beginning" \l "fn-1_18)

**$** git status

**#** On branch master

**#**

**#** Initial commit

**#**

**#** Changes to be committed:

**#** (use "git rm --cached <file>..." to unstage)

**#**

**#** new file: README.rdoc

**#** new file: Rakefile

.

.

.

(The results are long, so I’ve used vertical dots to indicate omitted output.)

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

**$** git commit -m "Initialize repository"

[master (root-commit) df0a62f] Initialize repository

42 files changed, 8461 insertions(+), 0 deletions(-)

create mode 100644 README.rdoc

create mode 100644 Rakefile

.

.

.

The **-m** flag lets you add a message for the commit; if you omit **-m**, Git will open the editor you set in [Section 1.3.1](http://ruby.railstutorial.org/chapters/beginning#sec-git_setup) and have you enter the message there.

It is important to note that Git commits are *local*, recorded only on the machine on which the commits occur. This is in contrast to the popular open-source version control system called Subversion, in which a commit necessarily makes changes on a remote repository. Git divides a Subversion-style commit into its two logical pieces: a local recording of the changes (**git commit**) and a push of the changes up to a remote repository (**git push**). We’ll see an example of the push step in [Section 1.3.5](http://ruby.railstutorial.org/chapters/beginning#sec-git_commands).

By the way, you can see a list of your commit messages using the **log** command:

**$** git log

commit df0a62f3f091e53ffa799309b3e32c27b0b38eb4

Author: Michael Hartl <michael@michaelhartl.com>

Date: Thu Oct 15 11:36:21 2009 -0700

Initialize repository

To exit **git log**, you may have to type **q** to quit.

[1.3.3 What good does Git do you?](http://ruby.railstutorial.org/chapters/beginning#sec-1_3_3)

It’s probably not entirely clear at this point why putting your source under version control does you any good, so let me give just one example. (We’ll see many others in the chapters ahead.) Suppose you’ve made some accidental changes, such as (D’oh!) deleting the critical **app/controllers/**directory:

**$** ls app/controllers/

application\_controller.rb

**$** 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. 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’s up:

**$** git status

**#** On branch master

**#** Changed but not updated:

**#** (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

**#**

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 easily by having Git check out the previous commit with the **checkout** command (and a **-f** flag to force overwriting the current changes):

**$** git checkout -f

**$** git status

**#** On branch master

nothing to commit (working directory clean)

**$** ls app/controllers/

application\_controller.rb

The missing directory and file are back. That’s a relief!

[1.3.4 GitHub](http://ruby.railstutorial.org/chapters/beginning#sec-github)

Now that you’ve put your project under version control with Git, it’s time to push your code up to[GitHub](http://github.com/), a social coding site optimized for hosting and sharing Git repositories. 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. This step is optional, but being a GitHub member will open the door to participating in a wide variety of open-source projects.

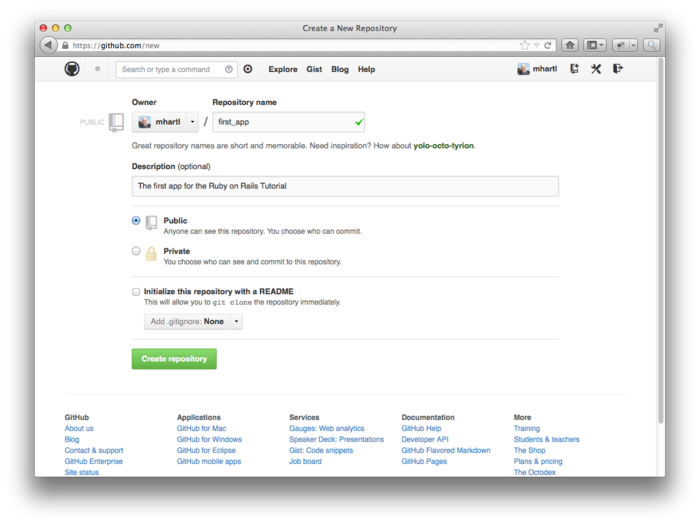


Figure 1.6: Creating the first app repository at GitHub. [(full size)](http://railstutorial.org/images/figures/create_first_repository_4_0-full.png)

GitHub has a variety of paid plans, but for open-source code their services are free, so sign up for a[free GitHub account](https://github.com/signup/free) if you don’t have one already. (You might have to follow the [GitHub tutorial on creating SSH keys](http://help.github.com/key-setup-redirect) first.) After signing up, click on the link to [create a repository](http://github.com/new) and fill in the information as in [Figure 1.6](http://ruby.railstutorial.org/chapters/beginning#fig-create_first_repository). (Take care *not* to initialize the repository with a **README** file, as **rails new** creates one of those automatically.) After submitting the form, push up your first application as follows:

**$** git remote add origin https://github.com/<username>/first\_app.git

**$** git push -u origin master

These commands tell Git that you want to add GitHub as the origin for your main (*master*) branch and then push your repository up to GitHub. (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> with your actual username. For example, the command I ran was

**$** git remote add origin https://github.com/mhartl/first\_app.git

The result is a page at GitHub for the first application repository, with file browsing, full commit history, and lots of other goodies ([Figure 1.7](http://ruby.railstutorial.org/chapters/beginning#fig-github_repository_page)).

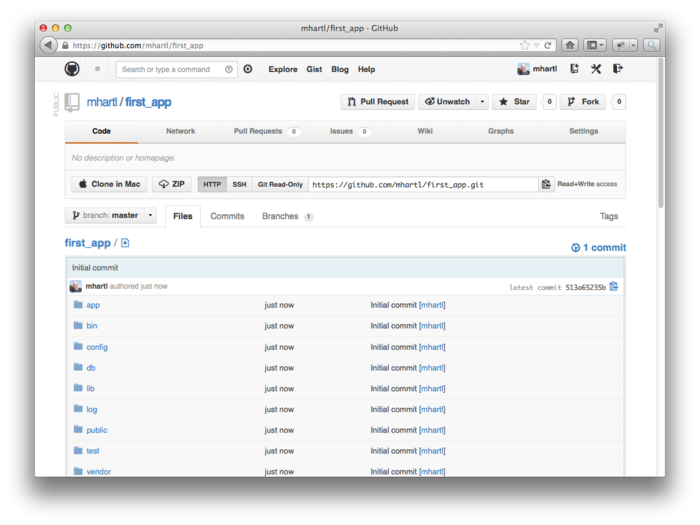


Figure 1.7: A GitHub repository page. [(full size)](http://railstutorial.org/images/figures/github_repository_page_4_0-full.png)

GitHub also has native applications to augment the command-line interface, so if you’re more comfortable with GUI apps you might want to check out [GitHub for Windows](http://windows.github.com/) or [GitHub for Mac](http://mac.github.com/). (GitHub for Linux is still just Git, it seems.)

[1.3.5 Branch, edit, commit, merge](http://ruby.railstutorial.org/chapters/beginning#sec-git_commands)

If you’ve followed the steps in [Section 1.3.4](http://ruby.railstutorial.org/chapters/beginning#sec-github), you might notice that GitHub automatically shows the contents of the **README** file on the main repository page. In our case, since the project is a Rails application generated using the **rails** command, the **README** file is the one that comes with Rails ([Figure 1.8](http://ruby.railstutorial.org/chapters/beginning#fig-rails_readme)). Because of the **.rdoc** extension on the file, GitHub ensures that it is formatted nicely, but the contents aren’t helpful at all, so in this section we’ll make our first edit by changing the**README** to describe our project rather than the Rails framework itself. In the process, we’ll see a first example of the branch, edit, commit, merge workflow that I recommend using with Git.

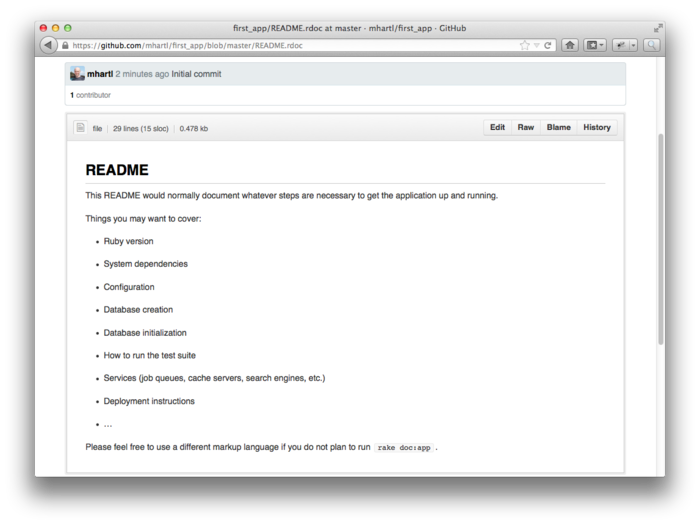


Figure 1.8: The initial **README** file for our project at GitHub. [(full size)](http://railstutorial.org/images/figures/rails_readme_4_0-full.png)

[Branch](http://ruby.railstutorial.org/chapters/beginning#sec-git_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. (If you set up the **co** alias in [Section 1.3](http://ruby.railstutorial.org/chapters/beginning#sec-version_control), you can use **git co -b modify-README** instead.)

The full value of branching only becomes clear when working on a project with multiple developers,[19](http://ruby.railstutorial.org/chapters/beginning" \l "fn-1_19) but branches are helpful even for a single-developer tutorial such as this one. In particular, the master branch is insulated from any changes we make to the topic branch, so even if we *really* screw 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, but it’s never too early to start practicing good habits.

[Edit](http://ruby.railstutorial.org/chapters/beginning#sec-git_edit)

After creating the topic branch, we’ll edit it to make it a little more descriptive. I prefer the[Markdown markup language](http://daringfireball.net/projects/markdown/) to the default RDoc for this purpose, and if you use the file extension**.md** then GitHub will automatically format it nicely for you. So, first we’ll use Git’s version of the Unix **mv** (“move”) command to change the name, and then fill it in with the contents of [Listing 1.8](http://ruby.railstutorial.org/chapters/beginning#code-new_readme):

**$** git mv README.rdoc README.md

**$** subl README.md

**Listing 1.8.** The new **README** file, **README.md**.

# Ruby on Rails Tutorial: first application

This is the first application for the

[\*Ruby on Rails Tutorial\*](http://railstutorial.org/)

by [Michael Hartl](http://michaelhartl.com/).

[Commit](http://ruby.railstutorial.org/chapters/beginning#sec-git_commit)

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

**$** git status

**#** On branch modify-README

**#** Changes to be committed:

**#** (use "git reset HEAD <file>..." to unstage)

**#**

**#** renamed: README.rdoc -> README.md

**#**

**#** Changed but not updated:

**#** (use "git add <file>..." to update what will be committed)

**#** (use "git checkout -- <file>..." to discard changes in working directory)

**#**

**#** modified: README.md

**#**

At this point, we could use **git add .** as in [Section 1.3.2](http://ruby.railstutorial.org/chapters/beginning#sec-adding_and_committing), but Git provides the **-a** flag as a shortcut for the (very common) case of committing all modifications to existing files (or files created using **git mv**, which don’t count as new files to Git):

**$** git commit -a -m "Improve the README file"

2 files changed, 5 insertions(+), 243 deletions(-)

delete mode 100644 README.rdoc

create mode 100644 README.md

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** first.

Note that we write the commit message in the *present* tense. 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 the GitHub post [Shiny new commit styles](https://github.com/blog/926-shiny-new-commit-styles) for more information.

[Merge](http://ruby.railstutorial.org/chapters/beginning#sec-git_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 34f06b7..2c92bef

Fast forward

README.rdoc | 243 --------------------------------------------------

README.md | 5 +

2 files changed, 5 insertions(+), 243 deletions(-)

delete mode 100644 README.rdoc

create mode 100644 README.md

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 2c92bef).

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 screw up the branch>

$ git add .

$ git commit -a -m "Major screw up"

$ 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.

[Push](http://ruby.railstutorial.org/chapters/beginning#sec-git_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 1.3.4](http://ruby.railstutorial.org/chapters/beginning#sec-github)), on most systems we can omit **origin master**, and simply run **git push**:

**$** git push

As promised, GitHub nicely formats the new file using Markdown ([Figure 1.9](http://ruby.railstutorial.org/chapters/beginning#fig-new_readme)).

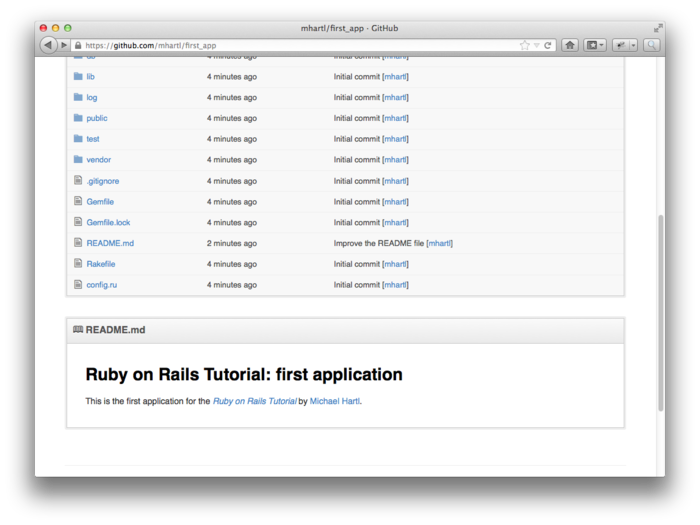


Figure 1.9: The improved **README** file formatted with Markdown. [(full size)](http://railstutorial.org/images/figures/new_readme_4_0-full.png)