



## Controle de versão e Implantação

### Introdução

Sistemas de controle de versão nos permite acompanhar as mudanças nos códigos do projeto, favorece o desenvolvimento colaborativo e permite que erros involuntários sejam revertidos. Hoje, saber um sistema de controle de versão é uma habilidade necessária para todos os desenvolvedores de software.

Grande parte do desenvolvimento Rails depende, de uma forma ou de outra, de controle de versões, e este é largamente feito pela comunidade Rails através do Git. O Git é um sistema de controle de versão originalmente desenvolvido por Linus Torvalds para hospedar o kernel do Linux.

Há muito o que se estudar sobre o Git, mas neste material veremos o suficiente para promover o versionamento de nossas aplicações Rails. Muitos outros materiais de altíssima qualidade podem ser encontrados online. Um bom começo é o livro **Pro Git by Scott Chacon (Apress, 2009)**.

Para finalizar, colocar o seu código-fonte sob controle de versão Git é altamente recomendável, não só porque é quase uma prática universal no mundo Rails, mas também porque vai permitir que você compartilhe o seu código mais facilmente assim como implante sua aplicação.

### Instalação e Configuração

O primeiro passo naturalmente é instalar o Git. No Ubuntu, basta digitar o comando abaixo em seu terminal. Para a instalação em outros sistemas, verifique os procedimentos em <http://www.git-scm.com/book/en/Getting-Started-Installing-Git>.

```
apt-get install git
```

*Digite git --version para verificar se a instalação foi bem sucedida.*

Depois da instalação, é interessante que algumas configurações sejam feitas. E essas configurações só precisam ser feitas uma vez por computador. Digite em seu terminal:

```
git config --global user.name "seu nome"  
git config --global user.email seu e-mail
```

E já podemos utilizar o Git para controlar as versões de nossas aplicações.

### Primeira Configuração do Repositório

Toda vez que for necessário criar um novo repositório, deverão ser executadas alguns passos. Para exemplificação, utilizaremos a aplicação **myFirstApp** criada no final da aula de configuração do ambiente de desenvolvimento. O primeiro deles é navegar até o diretório raiz do projeto e iniciar um novo repositório vazio com o comando:

```
git init
```

Sua saída deve ser semelhante a imagem.

```
joao@joao-R430-R480-R440: ~/myFirstApp
joao@joao-R430-R480-R440:~/myFirstApp$ git init
Initialized empty Git repository in /home/joao/myFirstApp/.git/
joao@joao-R430-R480-R440:~/myFirstApp$
```

Agora precisamos adicionar os arquivos de projeto para o repositório. Por padrão, o Git acompanha as alterações em todos os arquivos, e podemos não querer acompanhar todos eles. O Git oferece um mecanismo simples para ignorar esses arquivos, com a simples inclusão de um arquivo chamado **.gitignore** no diretório raiz do aplicativo com algumas regras que serão responsáveis por dizer ao Git quais arquivos serão ignorados. Se vocês verificarem os arquivos do diretório de nossa aplicação, perceberá que o Rails já cria, por padrão, esse arquivo junto com a criação da aplicação. O arquivo deve ser parecido com a imagem abaixo.

```
.gitignore
1 # See https://help.github.com/articles/ignoring-files for more about ignoring files.
2 #
3 # If you find yourself ignoring temporary files generated by your text editor
4 # or operating system, you probably want to add a global ignore instead:
5 #   git config --global core.excludesfile '~/gitignore_global'
6
7 # Ignore bundler config.
8 /.bundle
9
10 # Ignore the default SQLite database.
11 /db/*.sqlite3
12 /db/*.sqlite3-journal
13
14 # Ignore all logfiles and tempfiles.
15 /log/*.log
16 /tmp
17
```

Percebam que de acordo com este arquivo, o Git vai ignorar arquivos de log, arquivos temporários (TMP) e bancos de dados SQLite. A maioria destes arquivos mudam com muita frequência e automaticamente, sendo desnecessária sua inclusão no controle de versão. Ademais, no desenvolvimento colaborativo, essas mudanças podem gerar conflitos frustrantes. Quaisquer outros arquivos que devam ser ignorados, devem ser elucidados neste arquivo.

## Add e Commit

Agora vamos adicionar os arquivos do nosso projeto Rails ao repositório e depois “comitar” os resultados. Para adicionar todos os arquivos (exceto aqueles especificados no arquivo **.gitignore**), utilize o comando:

```
git add .
```

O ponto do comando referencia o diretório atual, e o Git adiciona todos os subdiretórios de forma recursiva. O comando **add** adiciona os arquivos do projeto em uma área de preparação, composta por alterações pendentes em seu projeto. Para verificar estas pendências, informe no terminal:

```
git status
```

Note que a saída para o comando apresenta as mudanças a serem submetidas.

```
joao@joao-R430-R480-R440: ~/myFirstApp

joao@joao-R430-R480-R440:~/myFirstApp$ git status
No ramo master

Submissão inicial.

Mudanças a serem submetidas:
(utilize "git rm --cached <arquivo>..." para não apresentar)

    new file:   .gitignore
    new file:   Gemfile
    new file:   Gemfile.lock
    new file:   Gemfile~
    new file:   README.rdoc
    new file:   Rakefile
    new file:   app/assets/images/.keep
    new file:   app/assets/javascripts/application.js
    new file:   app/assets/stylesheets/application.css
    new file:   app/controllers/application_controller.rb
```

Para dizer ao Git que queremos manter essas alterações, utilizamos o comando `commit`.

```
git commit -m "Initialize Repository"
```

A saída deve ser um pouco extensa, e tem seu início parecido com a imagem abaixo. O parâmetro `-m` permite que adicionemos uma mensagem ao `commit`, e isso é extremamente útil. Ainda é importante notar que os comandos Git são locais, isto é, são registrados apenas na máquina em que os comandos foram executados. O Git trabalha com um estilo de versionamento em duas partes lógicas (diferente do SVN). Na primeira parte acontece a gravação local das mudanças (`git commit`) e na segunda as mudanças são submetidas até um repositório remoto (`git push` – veremos mais sobre o `push` logo abaixo).

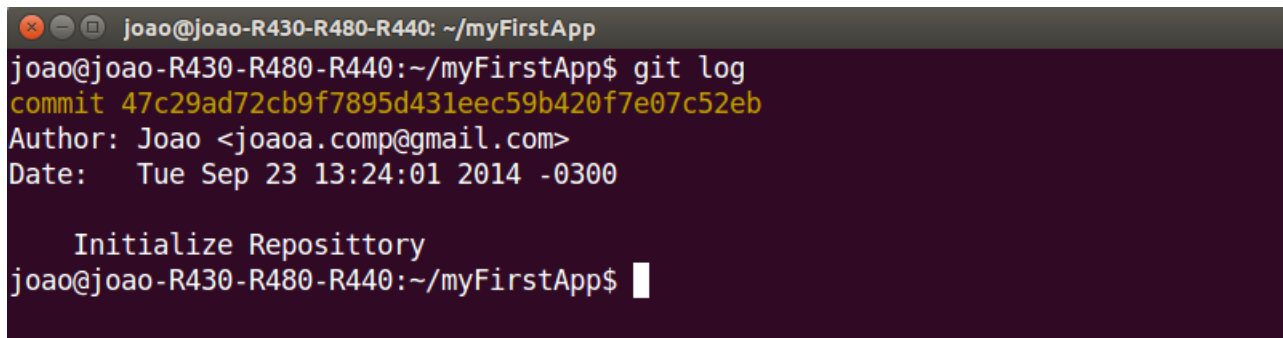
```
joao@joao-R430-R480-R440: ~/myFirstApp

joao@joao-R430-R480-R440:~/myFirstApp$ git commit -m "Initialize Repository"
[master (root-commit) 47c29ad] Initialize Repository
57 files changed, 951 insertions(+)
create mode 100644 .gitignore
create mode 100644 Gemfile
create mode 100644 Gemfile.lock
create mode 100644 Gemfile~
create mode 100644 README.rdoc
create mode 100644 Rakefile
create mode 100644 app/assets/images/.keep
create mode 100644 app/assets/javascripts/application.js
create mode 100644 app/assets/stylesheets/application.css
create mode 100644 app/controllers/application_controller.rb
create mode 100644 app/controllers/concerns/.keep
```

Para exibir uma lista de todos os `commits` efetuados até o momento, utilizamos o comando:

```
git log
```

Para o nosso exemplo, a saída:

A terminal window with a dark background. The title bar shows 'joao@joao-R430-R480-R440: ~/myFirstApp'. The prompt is 'joao@joao-R430-R480-R440:~/myFirstApp\$'. The command 'git log' has been executed, showing the output: 'commit 47c29ad72cb9f7895d431eec59b420f7e07c52eb', 'Author: Joao <joaoa.comp@gmail.com>', and 'Date: Tue Sep 23 13:24:01 2014 -0300'. Below this, the text 'Initialize Reposittory' is displayed. The prompt is now 'joao@joao-R430-R480-R440:~/myFirstApp\$' followed by a cursor.

```
joao@joao-R430-R480-R440: ~/myFirstApp
joao@joao-R430-R480-R440:~/myFirstApp$ git log
commit 47c29ad72cb9f7895d431eec59b420f7e07c52eb
Author: Joao <joaoa.comp@gmail.com>
Date: Tue Sep 23 13:24:01 2014 -0300

Initialize Reposittory
joao@joao-R430-R480-R440:~/myFirstApp$
```


Neste momento, talvez ainda não esteja claro o porquê de colocar nosso código sob controle de versionamento. Considere um cenário hipotético em que acidentalmente você apagou um arquivo qualquer da aplicação, por exemplo o arquivo Gemfile. Se você verificar o status do que está acontecendo, a saída do comando `git status` vai dizer que o arquivo foi apagado. Felizmente, as mudanças são apenas na “árvore de trabalho”, já que não foram “comitados” ainda. Então poderemos desfazer a alteração facilmente graças ao Git, através do comando `git checkout -f`, em que `-f` força que as alterações atuais sejam sobrescritas pelo `commit` anterior. Trabalharemos com esses comandos o tempo todo no desenvolvimento de nossas aplicações Rails.

## GitHub

Até este momento, nosso projeto está sob controle de versão com o Git, mas apenas localmente. Vamos ver agora como empurrar nosso código até o GitHub, um site otimizado para hospedagem e compartilhamento de repositórios Git. Ter uma cópia de seu repositório Git no GitHub tem dois propósitos. Primeiro, é um backup completo do seu código (incluindo o histórico completo de `commits`), e isso faz todo o desenvolvimento colaborativo muito mais fácil. Segundo, sendo um membro GitHub vai abrir as portas para participar de uma grande variedade de projetos de código aberto.

O GitHub oferece uma variedade de planos pagos, mas para o código-fonte aberto seus serviços são gratuitos. Para a disciplina, é suficiente que criemos uma conta gratuita no GitHub (se você ainda não tem). Basta se inscrever no site: <https://github.com/join>. Com a conta criada, talvez seja necessário seguir o tutorial disponível pelo próprio GitHub para a criação de chaves SSH ( <https://help.github.com/articles/generating-ssh-keys> ).

Clique no link para criar um novo repositório e preencha as informações necessárias, de acordo com a imagem abaixo. Apenas tome cuidado para não inicializar o repositório com um arquivo README, já que o comando `rails new` para criação da aplicação já cria este arquivo automaticamente.

**Owner**  **Repository name**  ✓

Great repository names are short and memorable. Need inspiration? How about [miniature-octo-spice](#).

**Description** (optional)



---

☒ **Public**  
Anyone can see this repository. You choose who can commit.

☐ **Private**  
You choose who can see and commit to this repository.

---

☐ **Initialize this repository with a README**  
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** ▾ | Add a license: **None** ▾ ⓘ


---

**Create repository**

Feitas as configurações, vamos enviar nossa aplicação para o repositório remoto, com os comandos.

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

Se vocês atualizarem seu repositório no GitHub, já poderão visualizar seu projeto no repositório remoto.



[joao-a-silva / myFirstApp](#)
Unwatch ▾ 1

---

First app for the Ruby on Rails — Edit


1 commit
1 branch
0 releases
1 contributor







---


branch: master ▾
[myFirstApp](#) / +
 ⋮

---

Initialize Repository

 **joao-a-silva** authored 2 hours ago latest commit 47c29ad72c

 <a href="#">app</a>	Initialize Repository	2 hours ago
 <a href="#">bin</a>	Initialize Repository	2 hours ago
 <a href="#">config</a>	Initialize Repository	2 hours ago
 <a href="#">db</a>	Initialize Repository	2 hours ago
 <a href="#">lib</a>	Initialize Repository	2 hours ago
 <a href="#">log</a>	Initialize Repository	2 hours ago

## Branch, Edit, Commit e Merge

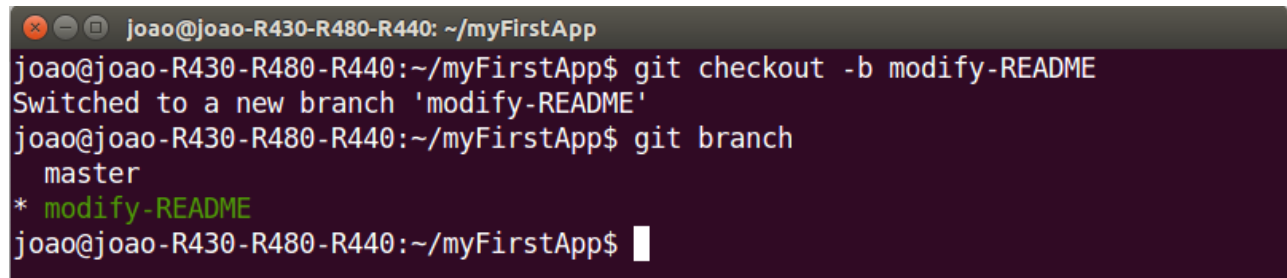
Se você observar, o GitHub mostra automaticamente o conteúdo do arquivo LEIA-ME na página de repositório principal. No nosso caso, já que o projeto é uma aplicação Rails gerado usando o comando `rails new`, o arquivo README é aquele que vem com o Rails. Devido à extensão `.rdoc` sobre o arquivo, GitHub garante que ele está bem formatado, mas o conteúdo não é muito útil. Então vamos fazer a nossa primeira edição, alterando o README para descrever nosso projeto, em vez de descrever o próprio Rails. No processo, vamos ver um primeiro exemplo do fluxo `branch`, `edit`, `commit`, `merge`, muito recomendado quando se usa o Git.

### Branch

O Git é incrivelmente bom em fazer branches, que em outras palavras pode ser descrito como cópias de um repositório onde podemos fazer mudanças (possivelmente experimentais) sem modificar os arquivos originais. Na maioria dos casos, o repositório pai é o branch `master`, e podemos criar um novo branch filho usando o comando `checkout` com o parâmetro `-b`. O comando `git branch` serve para listar todos os branches locais, e o asterico indica o branch que estamos atualmente.

```
git checkout -b modify-README
git branch
```

A saída para os comandos pode ser observada abaixo. Note que o branch `modify-README` é o branch atual, isso porque o comando `git checkout -b` tanto cria um novo branch quanto faz a mudança para ele.

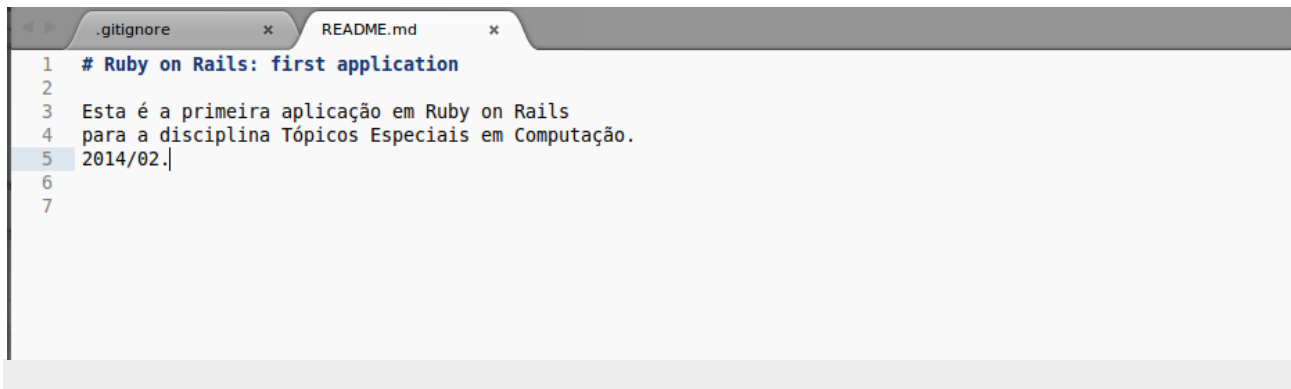
A terminal window with a dark background and light text. The prompt is 'joao@joao-R430-R480-R440: ~/myFirstApp'. The first command is 'git checkout -b modify-README', and the output is 'Switched to a new branch 'modify-README''. The second command is 'git branch', and the output shows 'master' and '\* modify-README', with the latter being highlighted in green. The prompt returns to 'joao@joao-R430-R480-R440:~/myFirstApp\$'.

O real potencial dos branches torna-se bem mais claro quando se trabalha em um projeto com vários desenvolvedores. Particularmente, o branch `master` é isolado de todas as alterações que fazemos para o branch criado, por isso mesmo que nós realmente façamos alguma coisa errada, sempre poderemos descartar as alterações retornando ao branch `master` e excluindo o branch filho. Vamos ver como fazer isso daqui a pouco. Vale comentar que para uma mudança tão pequena como esta que estamos fazendo, é desnecessária a criação de um novo branch, mas nunca é cedo demais para começar a praticar bons hábitos.

### Edit

Uma vez criado o branch acima, vamos editá-lo para torná-lo um pouco mais descritivo. Vamos utilizar a linguagem de marcação Markdown para o RDoc padrão para este fim. E se você usar a extensão de arquivo `.md` então GitHub irá formatar automaticamente bem. Primeiro vamos usar a versão do Git do `mv` ("move") para mudar o nome do arquivo, e, em seguida, preenchê-lo com o conteúdo como descrito abaixo.

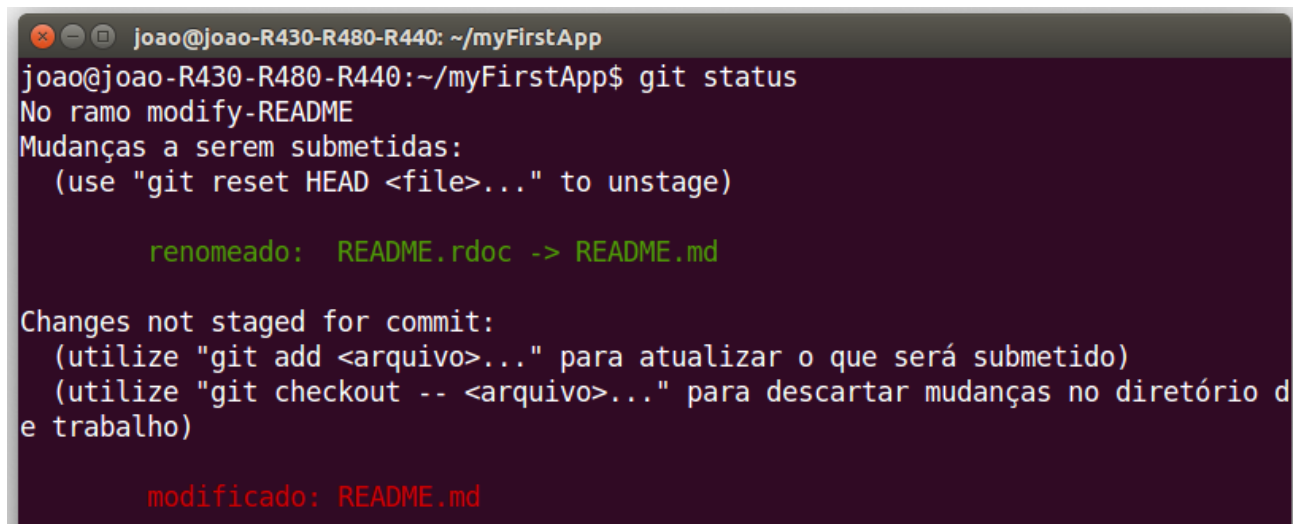
```
git mv README.rdoc README.md
```



## Commit

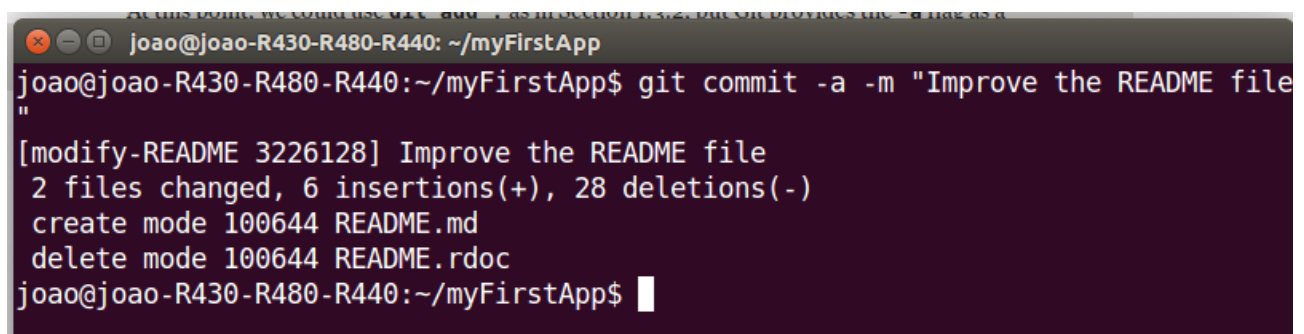
Verifiquemos o estado de nosso branch:

```
git status
```



Neste ponto, podemos usar `git add .`, mas não faz muito sentido já que não criamos nenhum novo arquivo. O Git fornece o parâmetro `-a` como um atalho para o caso (muito comum) de “comitarmos” todas as modificações para os arquivos existentes. Então:

```
git commit -a -m "Improve the README file"
```



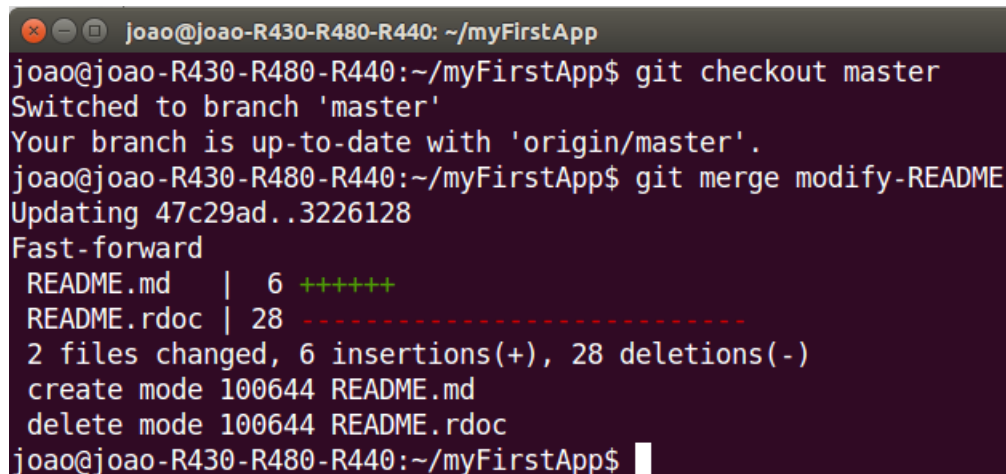


Muito cuidado ao usar o parâmetro `-a`. Se você tiver adicionado quaisquer novos arquivos ao projeto desde o último commit, você ainda tem que adicionar eles usando `git add` primeiro.

## Merge

Já com as alterações terminadas, estamos prontos para mesclar os resultados de volta para o nosso branch `master`:

```
git checkout master
git merge modify-README
```



```
joao@joao-R430-R480-R440: ~/myFirstApp
joao@joao-R430-R480-R440:~/myFirstApp$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
joao@joao-R430-R480-R440:~/myFirstApp$ git merge modify-README
Updating 47c29ad..3226128
Fast-forward
 README.md | 6 ++++++
 README.rdoc | 28 -----
 2 files changed, 6 insertions(+), 28 deletions(-)
 create mode 100644 README.md
 delete mode 100644 README.rdoc
joao@joao-R430-R480-R440:~/myFirstApp$
```

Seus resultados não deverão ser muito diferentes da saída mostrada acima. Depois de mescladas as mudanças, podemos excluir o branch criado com o comando:







```
git branch -d modify-README
```

Esta etapa é opcional, e é muito comum manter o branch criado. Desta forma, você pode alternar entre branches, que se fundem em mudanças toda vez que você chegar a um ponto de parada natural. Ainda é possível abandonar as alterações do branch com o comando `git branch -D`, que ao contrário do parâmetro `-d` apaga o branch mesmo que a mesclagem das alterações tenha sido realizadas.

## Push

Agora que temos o README atualizado, podemos empurrar as mudanças até GitHub para ver o resultado. Já fizemos um primeiro push, então só precisamos do comando `git push`. Como é de se esperar, o GitHub formata o novo arquivo usando Markdown.



 <a href="#">Gemfile</a>	Initialize Reposittory	3 hours ago
 <a href="#">Gemfile.lock</a>	Initialize Reposittory	3 hours ago
 <a href="#">Gemfile~</a>	Initialize Reposittory	3 hours ago
 <a href="#">README.md</a>	Improve the README file	17 minutes ago
 <a href="#">Rakefile</a>	Initialize Reposittory	3 hours ago
 <a href="#">config.ru</a>	Initialize Reposittory	3 hours ago

## README.md

# Ruby on Rails: first application

Esta é a primeira aplicação em Ruby on Rails para a disciplina Tópicos Especiais em Computação.  
2014/02.

## Implantação

Mesmo no início já nos preocuparemos em colocar nossa aplicação Rails em produção, ainda que ela não tenha nenhuma funcionalidade implementada. É uma etapa opcional, mas quanto antes fizermos a implantação mais cedo poderemos observar qualquer problema. Após o esforço adicional de se colocar o projeto em produção em um ambiente alternativo, podem surgir alguns problemas de integração difíceis de serem resolvidos a medida que o projeto cresce.

A implantação de aplicações desenvolvidas em Rails já foi bem trabalhosa, mas amadureceu rapidamente de modo que hoje temos várias ótimas opções. Estas opções vão desde hosts compartilhados até servidores privados.

Na disciplina utilizaremos o Heroku, serviço de implantação em nuvens, uma plataforma construída especificamente para a implantação do Rails e outras aplicações web. Há quem diga que com o Heroku a implantação de aplicações Rails seja ridiculamente fácil, a partir do momento que o projeto esteja sobre controle de versão com Git.

## Configuração do Heroku

O Heroku trabalha o banco de dados PostgreSQL. Então precisaremos adicionar a gem `pg` no ambiente de produção para permitir a comunicação do Rails com PostgreSQL. Para isto, substitua a linha `gem 'sqlite3'` no arquivo Gemfile da aplicação pelas linhas abaixo:

```
group :development do
  gem 'sqlite3', '1.3.8'
end

group :production do
  gem 'pg', '0.15.1'
  gem 'rails_12factor', '0.0.2'
end
```

Note-se também a adição da gem `rails_12factor`, que é utilizada pelo Heroku para trabalhar ativos estáticos, como imagens e folhas de estilo. Após salvar o arquivo, execute o comando `bundle install`, com o parâmetro especial abaixo:

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

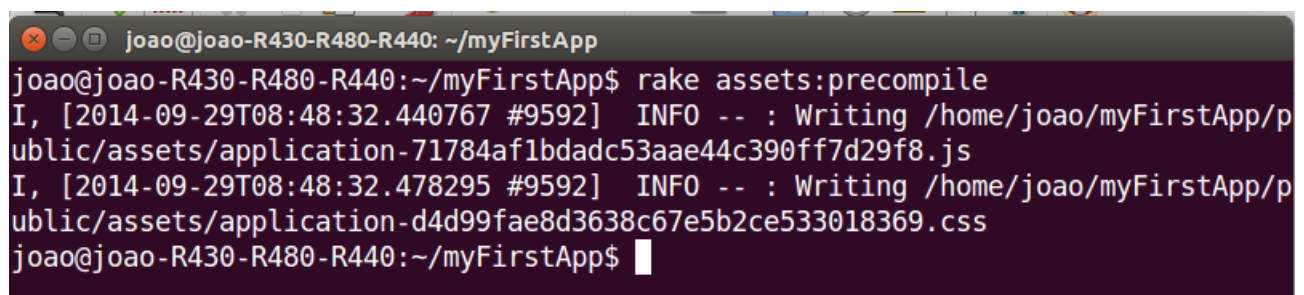
A opção `--without production` impede a instalação das gems necessárias somente no ambiente de produção. E mesmo que elas só serão utilizadas neste ambiente, elas são necessárias para atualizar o arquivo **Gemfile.lock**. Vamos dar um commit para confirmar a atualização:

```
git commit -a -m "Update Gemfile.lock for Heroku"
```

Agora faremos uma última configuração, a saber, a criação de arquivos Heroku que servirão como ativos estáticos, como imagens e CSS.

```
rake assets:precompile  
git add .  
git commit -m "Add precompiled assets for Heroku"
```

Sua saída dever ser algo parecido com a imagem abaixo:



```
joao@joao-R430-R480-R440: ~/myFirstApp  
joao@joao-R430-R480-R440:~/myFirstApp$ rake assets:precompile  
I, [2014-09-29T08:48:32.440767 #9592] INFO -- : Writing /home/joao/myFirstApp/public/assets/application-71784af1bdadc53aae44c390ff7d29f8.js  
I, [2014-09-29T08:48:32.478295 #9592] INFO -- : Writing /home/joao/myFirstApp/public/assets/application-d4d99fae8d3638c67e5b2ce533018369.css  
joao@joao-R430-R480-R440:~/myFirstApp$
```

Falaremos mais do comando `rake` no decorrer das aulas.

O próximo passo é configurar uma conta no Heroku. Para isso, se inscreva em <https://id.heroku.com/signup>. Verifique seu e-mail para finalizar a criação da conta e instale o aplicativo Heroku com o comando:

```
wget -q0- https://toolbelt.heroku.com/install-ubuntu.sh | sh
```

Para a instalação em outras plataformas, verificar o site <https://toolbelt.heroku.com/>. Em seguida, faça o login no Heroku com o comando abaixo, e informe suas credenciais.

```
heroku login
```

Finalmente, na raiz do projeto, digite o comando:

```
heroku create
```

```
joao@joao-R430-R480-R440:~/myFirstApp$ heroku create
Creating whispering-hamlet-2971... done, stack is cedar
http://whispering-hamlet-2971.herokuapp.com/ | git@heroku.com:whispering-hamlet-2971.git
Git remote heroku added
joao@joao-R430-R480-R440:~/myFirstApp$ clear
```

O comando cria um novo subdomínio apenas para o nosso aplicativo, disponível para visualização imediata. Percebam que o nome do subdomínio é gerado aleatoriamente, e não há nada lá ainda. Para a implantação da aplicação, são necessários dois passos. O primeiro consiste em usar o Git para empurrar os arquivos até o Heroku:

```
git push heroku master
```

Caso o comando acima dê o erro “*Agent admitted failure to sign using the key. Permission denied (publickey) fatal: The remote end hung up unexpectedly*”, ou algo do tipo, execute o seguinte comando e tente o anterior novamente:

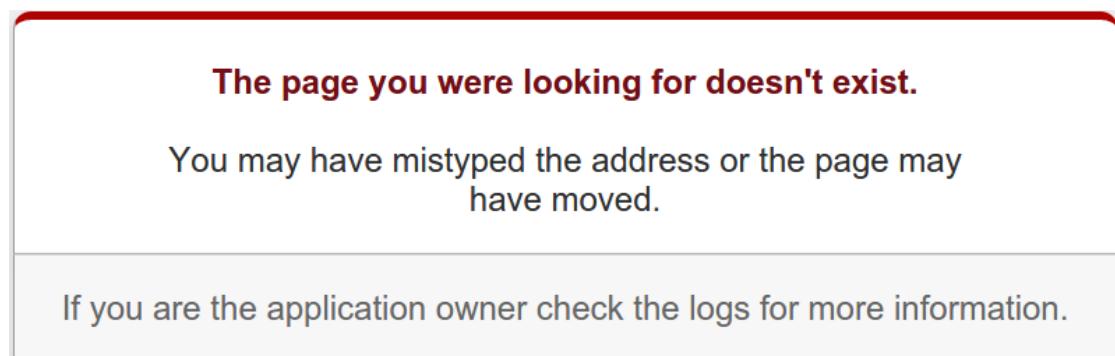
```
ssh-add ~/.ssh/id_rsa
```

Se tudo ocorreu bem, o início da saída em seu terminal dever ser:

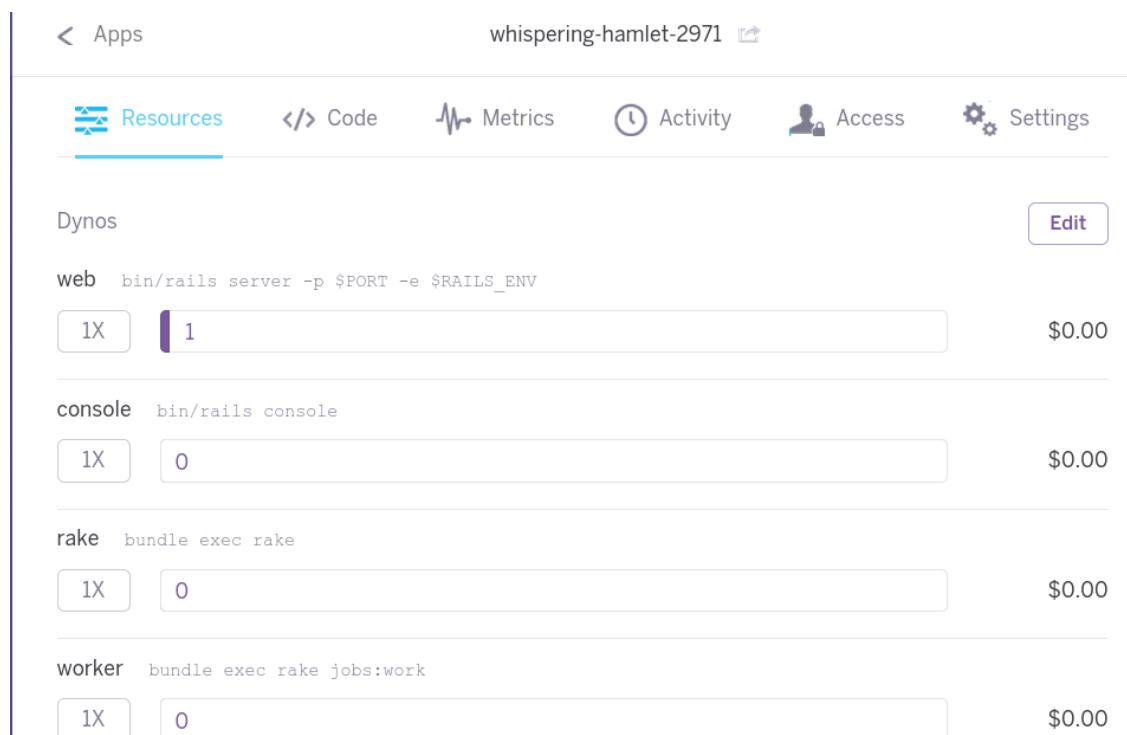
```
joao@joao-R430-R480-R440:~/myFirstApp$ git push heroku master
Initializing repository, done.
Counting objects: 67, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (56/56), done.
Writing objects: 100% (67/67), 15.91 KiB | 0 bytes/s, done.
Total 67 (delta 5), reused 0 (delta 0)

-----> Ruby app detected
-----> Compiling Ruby/Rails
-----> Using Ruby version: ruby-2.0.0
-----> Installing dependencies using 1.6.3
    Running: bundle install --without development:test --path vendor/bundle -
-binstubs vendor/bundle/bin -j4 --deployment
    Fetching gem metadata from https://rubygems.org/.....
    Installing i18n 0.6.11
    Installing rake 10.3.2
    Installing thread_safe 0.3.4
    Installing minitest 5.4.2
    Installing erubis 2.7.0
    Installing builder 3.2.2
    Installing json 1.8.1
```

Na verdade, não existe um segundo passo, o aplicativo já está implantado! Para visualizá-lo, você pode visitar o endereço criado pelo comando `heroku create`, ou usar o comando `heroku open`, que abre o navegador no endereço correto. E voilá, nossa aplicação já está disponível para uso.



A má notícia é que a página exibida é um erro, decorrente de problemas técnicos de integração do Heroku com o Rails 4.0. A boa notícia é que o erro é corrigido quando trabalharmos com as rotas no decorrer do curso. Após implantado com sucesso, o Heroku fornece uma interface amigável para administrar e configurar o aplicativo.



## Comandos Heroku

Há dezenas de comandos Heroku, e nós não os estudaremos no curso. Vamos verificar apenas um deles que serve para renomeação da aplicação:

```
heroku rename <nome>
```

É interessante manter o subdomínio aleatório criado pelo Heroku, assim, alguém poderia visitar o seu site somente se você lhes deu o endereço. E vale lembrar que o Heroku também aceita domínios personalizados.