

Conhecendo Rails

Guia prático e mão na massa
para desenvolvimento web



Eustáquio Rangel

Conhecendo Rails

Eustáquio Rangel de Oliveira Jr.

Esse livro está à venda em <http://leanpub.com/conhecendo-rails>

Essa versão foi publicada em 2018-01-30



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2013 - 2018 Eustáquio Rangel de Oliveira Jr.

Tweet Sobre Esse Livro!

Por favor ajude Eustáquio Rangel de Oliveira Jr. a divulgar esse livro no [Twitter](#)!

A hashtag sugerida para esse livro é [#rails](#).

Descubra o que as outras pessoas estão falando sobre esse livro clicando nesse link para buscar a hashtag no Twitter:

[#rails](#)

Outras Obras De Eustáquio Rangel de Oliveira Jr.

Conhecendo o Git

Conhecendo Ruby

A minha família

Conteúdo

Sobre esse livro	2
Contato	3
Convenções utilizadas	3
O que é Rails	5
Princípios	5
DRY	9
ORM	9
MVC	10
Instalando	13
Criando o projeto	14
Tem IDE?	15
Precisamos de um runtime JavaScript!	16
Estrutura de arquivos e diretórios criada	18
Rake	20
Rack	21
Ambientes do Rails	25
Iniciando o servidor	28
Spring	30
Começando a construir a aplicação	32
Pluralização	32
Criando o primeiro scaffold	34
Migrations	35
Asset pipeline	45
Configurando a timezone da aplicação	46
Active Record	49
Logger	60
Verificando alterações no objeto	61
Testes unitários	62
Fixtures	63
Configurando o teste unitário	64
Traduzindo	72
Testes funcionais	79
Controllers	84
Callbacks nos modelos	89
Métodos e escopos no modelo	98

CONTEÚDO

ActiveSupport	103
Helpers	112
Controladores e sessões	115
Testes de sistema	118
Roteamento e REST	122
Roteamento	122
REST	123
Restringindo acesso com sessões e callbacks de controlador	134
Refatorando um pouco os controladores	137
Views e layouts	141
Usando layouts diferentes	141
Tomando mais cuidado ainda com o mass-assignment	145
Utilizando presenters	150
Criando um controlador para o público	154
Associações entre modelos	157
Associação do tipo um-para-um	160
Associação do tipo um-para-muitos	162
Associações de muitos para muitos	166
Associações de muitos para muitos, através	171
Acelerando as consultas nas associações	174
Juntando joins e includes	176
Pluck versus select	177
Criando um outro tipo de associação um-para-um	178
Upload de arquivos	179
Criando o modelo de imagem	179
Adaptando as views e controllers, com imagem	182
Redimensionando as imagens	183
Criando uma associação polimórfica	187
Criando um módulo compartilhado	192
Movendo a lógica para o modelo	193
Interface pública da loja	197
Strings seguras para HTML	200
Markdown	207
Turbolinks	209
Utilização	209
Desabilitando	210
Verificando o funcionamento	210
Interagindo	211
Carrinho de compras	213
Quantidade em estoque	213

CONTEÚDO

Configurando as sessões	215
Criando o carrinho	217
Comprando um produto	218
Removendo um produto do carrinho, com Ajax	221
Alterando a quantidade do produto, com Ajax	223
Fechando o pedido	225
Travando registros na aplicação	234
Optimistic Locking	234
Pessimistic Locking	238
E-mail	240
Variáveis de ambiente	240
Mailers	242
Anexos	245
Utilizando enums	246
Criando um job para processamento em background	249
Enviando o processamento para background	252
DelayedJob	252
Resque	255
Sidekiq	257
ActionCable	260
O que é	260
Criando o canal	260
Criando o JavaScript	260
Comunicando com o canal	263
Caching	269
Ativando o caching em páginas	269
Limpando o cache	271
Russian Doll caching	274
HTTP caching	278
Paginação	282
Busca e autocomplete	284
Utilizando o autocomplete do jQuery	284
Adaptando a rota e o controlador	286
Implementando a busca no modelo	287
Buscas com LIKE?	288
API	291
Versionando	291
Permitindo	297
Autenticando	299
Limitando	305

CONTEÚDO

Deploy	309
Sistema operacional	309
SSH	309
RVM	310
Servidor web e módulo para aplicações Rails	310
Usuário no servidor	312
Configurando o servidor	312
Configurando o banco de dados	314
Configurando o método de login no servidor	315
Configurando o método de deploy	316
Configurando deploy com uma ferramenta automatizada	320
Tomando conta dos seus logs	324
Sass	326
CoffeeScript	329
Extras	330
MiniTest	330
Guard	342
FactoryBot	344
Capybara	353
SimpleCov	360
Brakeman	367
Chartkick	371
PaperTrail	375
PaperClip	379
CarrierWave	386
Bootstrap	391
Tmux	396
ElasticSearch	399
Apenas API	412
Gerando um diagrama do banco de dados	418
Fim!	422

Copyright © 2017 Eustáquio Rangel de Oliveira Jr.

Todos os direitos reservados.

Nenhuma parte desta publicação pode ser reproduzida, armazenada em bancos de dados ou transmitida sob qualquer forma ou meio, seja eletrônico, eletrostático, mecânico, por fotocópia, gravação, mídia magnética ou algum outro modo, sem permissão por escrito do detentor do copyright.

Ilustração da capa por Ana Carolina Otero Rangel.

Sobre esse livro

Alguns anos atrás, mais precisamente em 22 de Abril de 2006, lancei um dos primeiros tutoriais de Rails daqui do Brasil, bem curtinho e mão na massa, seguido pelo “desaforo” monstro (no bom sentido) de 300 páginas do Ronaldo Ferraz, “Rails para sua diversão e lucro”, que na minha opinião foi o material que foi mais utilizado para impulsionar o *framework* por aqui. Pena que depois dele o Ronaldo não escreveu mais nada, pois a desenvoltura e eloquência dele para explicar o tema e metodologias associadas são ótimas.

Durante todos esses anos, venho trabalhado com Rails como ferramenta de preferência para projetos web, só não o utilizando se for algum requisito de projeto do cliente utilizar alguma outra linguagem (geralmente, para a parte web, PHP), mas venho negando algumas situações desse tipo por ter ficado mal acostumado com a qualidade e facilidade que o Rails proporciona as nossas aplicações. Sério, não é porque eu escrevi o primeiro livro de Ruby do Brasil ou coisa do tipo, eu costumo promover as coisas que eu realmente acredito que sejam boas.

Além de trabalhar com Rails, já ministrei muitos treinamentos, tanto em Ruby como Rails, através de cursos organizados por mim mesmo ou minha empresa, a Bluefish¹, ou de cursos de mini-cursos de Rails em faculdades e empresas. Minha meta nesse livro é tentar chegar no mesmo ponto em que eu chego através do material que levo e dos vários “extras” dos treinamentos ao vivo, apresentando o *framework* para quem está chegando agora e dando algum pitaco aqui ou ali das tecnologias associadas, apesar de acreditar que nunca vou conseguir chegar no ponto das aulas presenciais, pois eu falo mais que o homem da cobra e faço um monte de macaquices nos treinamentos. Enfim, é uma tentativa.

Nesse livro, basicamente vamos ver o que já vem embutido com o Rails, sem utilizar (muito) algumas outras *gems* que atualmente são bastante utilizadas e populares. Minha meta é mostrar como é o *framework* assim que “sai da caixa”, e mostrar como fazer determinadas coisas com os recursos mínimos apresentados e entregues por ele. Aqui e ali vão existir algumas dicas de como dar uma incrementada em determinada *feature*, mas esperem encontrar aqui um Rails puro-sangue, sem muitas modificações ou extras. Existem alguns outros livros muito interessantes, recomendados, a respeito desses outros recursos que podem e algumas vezes, devem ser utilizados para o desenvolvimento de uma aplicação Rails, mas existe também um certo exagero em ensinar o *framework*, chegando ao ponto de ficar parecido com uma piadinha antiga sobre uma pilha de livros de Java necessários para aprender Java e dois livros para aprender Rails². Hoje em dia, se não tomarmos cuidado, a coisa se inverte. Por isso que, mesmo abrindo mão de ensinar algumas coisas mais “cool” por aqui, minha meta é ensinar como desenvolver com Ruby on Rails em apenas um livro.

Acredito que para quem está chegando agora, é uma didática interessante. Alguns podem até me criticar depois por coisa do tipo “mas porque diabos ele me fez escrever tanto código se dava para usar somente uma *gem*” ou “porque ele não me mostrou a tecnologia X que é muito

¹<http://bluefish.com.br>

²<https://dl.dropboxusercontent.com/u/1482800/javaversusrubybooks.jpeg>

chique, e eu quero ser chique”, mas ninguém vai poder me criticar por mostrar como implementar determinados comportamentos e *features* com os recursos mínimos disponíveis, que no final, até reduzem determinadas facilidades que podem ser tornar complexidades. Espero que gostem.

Como complemento, e para quem está conhecendo o *framework* agora, fica a dica de ler o meu outro *ebook*, “Conhecendo Ruby”³. Para quem está conhecendo o *framework* antes da linguagem, **parem tudo**: leiam o livro sobre a linguagem e depois voltem por aqui, senão algumas das coisas que o Rails faz vão parecer bruxaria ou alguma coisa do tinhoso. A urgência de começar com uma aplicação no *framework* sem conhecer a linguagem em que ele é feito pode até ser justificada por algum prazo apertado ou mesmo a ânsia de meter logo a mão na massa, mas acreditem, vocês vão perder uma ótima chance de entender mais o *framework* logo no início se não conhecereem a linguagem Ruby de uma forma básica antes.

Ah, e tudo o que é mostrado aqui foi executado em um sistema operacional GNU/Linux. Rails vai rodar melhor em sistemas *Unix like* como Linux e o OSX (com certeza utilizando um Linux como servidor de produção), e com algum esforço, no Windows. Mas não me peçam dicas para esse último, está fora do escopo aqui do livro e até do meu dia-a-dia.

Contato

Meu site é <http://eustaquierangel.com>⁴, meu Twitter é [@taq](https://twitter.com/taq)⁵ e meu Github é <http://github.com/taq>⁶.

Convenções utilizadas

Durante o livro temos algumas convenções:

- Texto em *italico* para palavras não tão tradicionais no nosso Português;
- Texto em **negrito** para destacar alguma informação importante;
- Texto em fonte de tamanho fixo para código;
- Linhas iniciadas com \$ indicam que estão sendo executadas em um terminal;
- Em partes de código ou comandos no terminal em que forem encontrados três pontos (...), significa que acima ou abaixo dos pontos existe conteúdo que foi omitido para que não fosse apresentado todo o código. Também pode ser utilizado para indicar algumas operações como inserir conteúdo em um arquivo e logo após executar um comando no terminal;
- Encontrando essa barra \ no final de uma linha de código significa que a linha continua abaixo;

E temos algumas caixas de informações como:

³<http://leanpub.com/conhecendo-ruby>

⁴<http://eustaquierangel.com>

⁵[http://twitter.com/taq](https://twitter.com/taq)

⁶<http://github.com/taq/>



Essa é uma caixa de dica



Essa é uma caixa de alerta



Essa é uma caixa de algum desafio ou exercício

O que é Rails

Rails é um *framework* para desenvolvimento web, disponibilizado como código aberto, também chamado somente de *Rails* ou *RoR*, escrito na linguagem de programação Ruby⁷. Foi criado por David Heinemeier Hansson, também chamado de DHH, em 2003, tendo seu código liberado como código livre em 2004 e que atualmente conta com milhares de colaboradores.

Ele é “meta-framework”, cujas partes vamos ir conhecendo no decorrer do livro. Um *meta-framework* é baseado em vários componentes, que funcionam juntos mas podem ser substituídos. Podemos pegar como exemplo o ActiveRecord, que é um dos componentes que vem como padrão no Rails mas que pode não ser utilizado quando usamos, por exemplo, o MongoDB⁸ como banco de dados do nosso projeto.

Princípios

O Rails utiliza padrões e metodologias como o MVC (Model-View-Controller)⁹, DRY¹⁰, *Convention over Configuration*¹¹, ORM¹² (*Object Relational Mapping*), scaffolding¹³, etc., sendo que costumo dizer que ele desburocratiza e deixa menos chato de utilizar esses tipos de tecnologias que, dependendo da ferramenta utilizada, eram chatas e trabalhosas, e costumavam ficar fora do dia-a-dia dos desenvolvedores por conta disso.

Inclusive, temos publicado a “doutrina do Rails”¹⁴, onde são expostos alguns dos pilares, padrões e metodologias utilizadas no Rails. Vamos dar uma rápida olhada nos atuais, já que eles foram sendo adaptados durante vários anos, e podem estar diferentes na hora que você está lendo isso.

Otimizado para a felicidade dos programadores

Rails é feito em Ruby, uma linguagem em que o seu criador, Yukihiro Matsumoto, o popular “Matz”, sempre se refere à palavra “felicidade”. Em [uma entrevista](#)¹⁵, é perguntado ao Matz:

Em um artigo introdutório sobre Ruby, você escreveu: “Para mim o propósito da vida é em parte ter alegria. Os programadores geralmente sentem alegria quando podem se concentrar no lado criativo da programação, então Ruby foi projetada para tornar os programadores felizes”. Como Ruby pode tornar os programadores felizes?

⁷Para conhecer Ruby, veja o meu *ebook* gratuito [Conhecendo Ruby](#)

⁸<https://www.mongodb.org/>

⁹<http://pt.wikipedia.org/wiki/Mvc>

¹⁰http://pt.wikipedia.org/wiki/Don%27t_repeat_yourself

¹¹http://pt.wikipedia.org/wiki/Conven%C3%A7%C3%A3o_sobre_configura%C3%A7%C3%A3o

¹²http://pt.wikipedia.org/wiki/Mapeamento_objeto-relacional

¹³<http://pt.wikipedia.org/wiki/Scaffolding>

¹⁴<http://rubyonrails.org/doctrine/>

¹⁵<http://www.artima.com/intv/rubyP.html>

E Matz respondeu:

Você quer aproveitar a vida, não é? Se você termina seu trabalho rapidamente e seu trabalho é divertido, isso é bom, não é? Esse é o propósito da vida, em parte. Sua vida é melhor. Eu quero resolver problemas que eu encontro na vida diária usando computadores, então eu preciso escrever programas. Usando Ruby, eu quero concentrar nas coisas que eu faço, não nas regras mágicas da linguagem, como começar algo com `public void something something something` para dizer, “imprimir ‘oi mundo’”. Eu só quero dizer, “imprima isso!”. Eu não quero todas as palavras-chave mágicas ao redor. Eu só quero me concentrar na tarefa. Essa é a ideia básica. Então, eu tentei fazer o código Ruby conciso e sucinto.

Rails leva em conta uma outra visão pessoal do Matz, compartilhada por vários outros desenvolvedores, inclusive eu, que é “O Princípio da Mínima Surpresa” (PoLS), que determina que a linguagem deve se comportar do jeito que esperamos e não fazendo alguma coisa inesperada ou evitando que se use comportamento que é logicamente correto. Rails tem um princípio parecido, chamado “O Princípio do Sorriso Grande”, feito pelo seu criador, DHH, que diz que “APIs desenhadas com grande atenção me fazem sorrir mais e mais”. Como Matz, DHH diz que criou o Rails para seu próprio uso (que foi o que ele fez e faz ainda hoje), mas quando uma pessoa já pensa em alguma ferramenta bem desenhada, otimizada para resolver problemas de maneira mais fácil, voltada para deixar o seu próprio trabalho com mais satisfação e a libera como código aberto, todos sabemos onde isso vai dar em grande parte: mais pessoas satisfeitas com o seu trabalho.

Um problema é que hoje em dia muitos desenvolvedores tendem a estender e modificar tudo isso, até de forma que lembra muitas coisas “inchadas” de recursos “enterprise” de algumas tecnologias utilizadas nas últimas décadas, argumentando que buscam uma “simplificação” ou “organização” mais eficiente que no final acabam levando à uma base de código enorme que muitas vezes fazem justamente o contrário, complicam demais. Na minha opinião isso perverte o princípio básico original de simplicidade e satisfação e serve em grande parte para uma massagem no cérebro para se sentir mais “skilled”.

Novamente, na minha opinião, o verdadeiro valor está na simplicidade, na clareza e manutenção fácil do código, e não em um emaranhado de tecnologias e centenas linhas que aparentemente deveriam resolver o “problema” de dezenas de linhas de código!

Hoje em dia muitas vezes Ruby e principalmente Rails são apresentados para os desenvolvedores de maneiras tão complicadas que tudo o que Matz e DHH disseram acima vai por ralo abaixo. Muitas vezes eu escuto pessoas dizendo que “Rails é complicado”, por causa justamente da maneira que foram apresentadas ao Rails junto com dezenas de outras ferramentas associadas. Isso é irresponsabilidade com os desenvolvedores que estão chegando agora, e só serve para ou os afastar das tecnologias ou criar uma base de conhecimento muito mais complicada do que deveria ser.

Como vamos ver no decorrer do livro, vamos criar uma estrutura básica de “cadastro, onde utilizamos o recurso de `scaffold`, algo que algumas pessoas veêm como um “pecado”, com apenas 4 linhas, algo que parece mágica, mas que faz com que as pessoas fiquem maravilhadas com o poder que toda essa aplicação básica, criada de maneira muito rápida, entrega. Faça as pessoas

sorrirem, faça as pessoas ficarem empolgadas, faça elas ficarem animadas com uma simplificação do seu trabalho, em algumas vezes ter prazer novamente em desenvolver software, e depois, se necessário, apresente algumas outras ferramentas e metodologias que podem complicar um pouco mais tudo isso em favor de algum recurso que seja necessário na aplicação.

Algumas vezes já vi projetos em que as pessoas começam a sorrir e ficarem empolgadas novamente quando o que estava complicado começa a ser simplificado, o que é bom, mas nesse caso significa que muito esforço já foi gasto em lugares que não o justificavam.

Falando em “*scaffold* do pecado”, quando gostarem de alguma tecnologia (e se vocês estão chegando agora em Ruby e Rails tenho certeza de que vão gostar), por favor sejam “evangelistas” e não “pregadores”. Os primeiros tendem a divulgar suas idéias sem chamar as outras pessoas de pecadores ou possuídos, como fazem os segundos, que são mais chatos. Isso não leva a nada de positivo. Existe um ditado na web que diz “não alimente os trolls”, eu também digo “não alimente o desenvolvedor arrogante”. Ele é um porre.

Convenção acima da configuração

Convention Over Configuration significa que temos várias facilidades fornecidas pelo *framework* desde que respeitemos algumas convenções que ele nos pede para seguir. São convenções fáceis e bem estruturadas (algumas das quais podem dar aquele sentimento de “diabos, porque não havia pensado nisso antes, tudo ia ficar muito mais simples e eficiente”) mas que **não são obrigatórias** e que podem ser alteradas se necessário. Nesse caso, o *framework* deixa na mão do programador para resolver as pendências que serão necessárias para o seu bom funcionamento.

Um dos lemas de produtividade do Rails é “você não é um belo e único floco de neve”, onde abrindo mão um pouco da sua individualidade e aderindo à essas convenções bem estruturadas, podemos produzir mais. É uma questão de costume (“mas eu faço isso assim faz 20 anos!”) e também uma questão de ego, muitas vezes admitindo que o seu jeito não é o melhor e que você vai ter muito mais retorno reavaliando seu método de trabalho.

Isso também facilita a entrada de novos programadores utilizando a tecnologia, onde não precisam saber tanto de determinadas coisas para fazer aplicações bem legais, *deixando claro* que muitas dessas coisas que foram abstraídas pelas “mágicas” do Rails, mais tarde durante o processo de aprendizado, **devem ser reconhecidas a aprendidas**, correndo o risco de fazer como alguns programadores muito acostumados com aplicações desktop migrando para a web utilizando ferramentas automáticas sem nem ter idéia de como é o funcionamento básico de aplicações web. Inclusive, para ter uma idéia de como essas coisas funcionam, recomendo dar uma olhada no livro “*Desconstruindo a Web*”, do William Molinari¹⁶.

O menu é “omakase”

Sabe quando você chega em um restaurante e não faz ideia do que pedir, seja por desconhecer a culinária local ou pelo tanto de opções que tem no menu, e pede pela sugestão do chef, levando em conta que ele vai te recomendar algo bom, mesmo que você nem saiba direito o que é exatamente “bom”? Esse é o conceito de “omakase”, onde você não precisa ser um expert ou atirar no escuro,

¹⁶ <https://desconstruindoaweb.com.br/>

confiando em alguém mais experiente que você, que é o que fazem na *stack default* do Rails, que é compartilhada entre milhares de pessoas, com resultados como:

1. Cria uma experiência compartilhada, dá um senso mais forte de comunidade e deixa mais fácil ensinar e ajudar as pessoas.
2. As pessoas aperfeiçoam as mesmas ferramentas.
3. Substituições são possíveis, mas não necessárias. Quando você tiver uma ideia clara da ferramenta (e por clara, por favor, não se baseiem em quanto que você complicou determinadas funcionalidades básicas sem entender elas direito) você pode agir como seu próprio chefe, escolhendo cuidadosamente suas opções.

É um paradigma de ninguém

No Rails muitas ideias e paradigmas diferentes são implementados, de uma forma que os deixam ideologicamente flexíveis, permitindo lidar com uma série de problemas. Alguns paradigmas se comportam bem dentro do espaço onde são originalmente aplicados, mas ficam muito rígidos quando saem dessa esfera de conforto, e Rails os aplica de um modo que o resultado final no *framework* é mais sólido e capaz do que qualquer paradigma individual possa ser.

Exalta código bem feito

Código bem feito tem que ser perseguido com vigor, sendo que o que pode ser código bem feito e belo pode ser motivo de discussões intermináveis, mas antes de mais nada, a beleza vem da simplicidade e a eficiência embutida nessa simplicidade, deixando para o programador menos código para escrever, focando nas coisas que mais importam.

Ferramentas afiadas

A linguagem Ruby, em que Rails é feito, já dá por padrão várias ferramentas e ótimos recursos para o programador, com inclusive alguns deles sendo criticados como sendo muito poder para o programador, onde pessoas vindas de ambientes mais restritivos pensam em todos os tipos de calamidades por causa do imenso poder dessas ferramentas afiadas. Mas não retiramos as facas afiadas da cozinha e pedimos para as pessoas fatiarem tomates com esponjas: apenas criamos, através da educação e do bom senso, o hábito de deixar as facas um pouco mais longe de quem não as sabe utilizar e ensinamos que caso as alcancem, o uso errado pode trazer perigo, mas o uso correto vai trazer novas habilidades.

Rails fornece várias ferramentas afiadas, onde não vão pedir desculpa se você se cortar, mas vai ter paciência e ajudar até que você as saiba utilizar.

Valorização de ambientes integrados

Rails é em primeira instância um imenso monólito, desenhado para resolver o problema completo, cobrindo desde o banco de dados até o JavaScript no *front end*. Dessa forma, ele sai da frente do programador permitindo que as aplicações possam ser construídas de maneira imediata sem a necessidade de integrar mais nada, fornecendo o poder de aplicações em ajustadas e customizadas com o uso de um único ambiente integrado.

Progresso e estabilidade

Rails não tem medo de evoluir durante os anos, onde valoriza o progresso mesmo ao custo da estabilidade, mas utilizando o bom senso. Uma das cicatrizes recentes foi a migração de aplicações da versão 2 para a versão 3, onde muitas coisas foram alteradas e ainda refletem hoje em dia, onde vemos que apesar de dolorosas, foram necessárias.

Arma uma tenda grande

Ok, essa frase traduzida ficou estranha, mas ... os desenvolvedores do Rails procuram escutar e discutir ideias e códigos de quem quer que os aponte. Não tentam ser tudo para todos, mas se Rails fosse uma festa, deixariam as pessoas trazer as próprias bebidas, para que possam ser combinadas talvez em uma nova. Nunca se sabe quando alguém vai começar arrumando alguma coisa escrita errada na documentação e vai começar a fazer o novo próximo grande recurso!

Alguns outros princípios e metodologias aplicados no Rails são:

DRY

DRY significa “Don’t Repeat Yourself”, ou seja, não faça a mesma coisa mais de uma vez. Podemos identificar em alguns sistemas comportamentos repetidos que fazem a mesma coisa, que aumentam o custo de manutenção e a chance de, quando precisar de alguma mudança, mudar apenas em um lugar e esquecer dos outros, o que leva fatalmente à mais problemas e desperdiça a chance de fazer a coisa bem feita da primeira vez. A reutilização de código que Ruby nos permite com suas características de orientação à objetos são uma mão na roda para isso.

ORM

O ORM é uma camada encarregada de conectar objetos com informações no banco de dados. Se você detesta escrever SQL¹⁷ (se não gosta de escrever tudo bem, mas conhecer e conhecer como fazer alguns *tunings* em SQL é uma coisa importante e recomendável de saber) vai adorar saber que utilizando Rails não vamos precisar escrever uma linha sequer de SQL, já deixando claro que se quisermos, é perfeitamente possível de ser feito. Mas utilizando o poder da orientação à objetos de Ruby, essa camada fica descomplicada, simples e prazerosa de utilizar, diferente de algumas outras opções de ORM que são até populares mas que fica bem mais complicado de utilizar.

Utilizando o COC (descrito acima), o mapeamento de uma classe em Ruby chamada, por exemplo, Pessoa, para uma tabela chamada Pessoas (ou seja, no plural, onde um objeto representa uma linha da tabela, sendo singular, e uma tabela, por ser uma coleção de objetos, fica no plural) é feito com apenas 2 linhas de código.

¹⁷<https://pt.wikipedia.org/wiki/SQL>

MVC

Se você já sabe o que é o padrão MVC e está careca de ver o pessoal escrevendo disso, pule essa parte. Vou tentar fazer uma descrição bem didática para o pessoal que nunca ouviu falar nisso (ei, ninguém nasceu sabendo, certo, e não tem problema algum em explicar isso de uma forma mais, digamos, simples) com alguns conceitos baseados em ilustrações e casos de uso.

Model, view, controller

O padrão MVC¹⁸ é, antes de mais nada, na minha opinião, é um jeito de “fatiar” a sua aplicação de modo a deixar ela mais:

1. Gerenciável
2. “Testável”
3. Organizada

Lembrem-se de César e Napoleão Bonaparte com o seu “Dividir para Conquistar”¹⁹ e imaginem que uma aplicação, apesar de ser uma coisa só, pode ser manipulada como algumas camadas “lógicas” que permitem, além de trabalhar mais focado em determinado comportamento, dividir as tarefas entre equipes diferentes. Como um exemplo, geralmente o pessoal focado em código não tem muito jeito para algumas questões de *interface*, e podemos pensar que nesse caso podemos passar para o pessoal do *front-end*, que vai se preocupar mais com as questões visuais e de usabilidade do sistema. As partes são independentes, desacopladas, mas funcionam de maneira complementar.

Vamos olhar isoladamente cada cama, mas fora de ordem ali da sigla. Vamos começar com a letra C e ver qual o conceito de um controller.

Controller

O controller, em uma aplicação web, é quem recebe o que digitamos no navegador, descobre para onde que tem que enviar a requisição que fizemos e retorna o resultado de um modo que possa ser interpretado de alguma forma pelo navegador. Assim, quando digitamos algo como

¹ <http://leanpub.com/conhecendo-rails>

É o controlador que no final das contas vai tratar a nossa requisição. Quando digitamos uma URL como essa:

¹⁸ <http://pt.wikipedia.org/wiki/Mvc>

¹⁹ https://pt.wikipedia.org/wiki/Dividir_para_conquistar

1 <http://eustaquierangel.com/posts/255>

Podemos decompor em `controller/action/id`, indicando que é um controlador que chama `posts` que vai fazer alguma ação com alguma coisa que tem o `id` igual à 255. O código presente nos `controllers` tem que ser enxuto, ou seja, pouco código, comparado ao resto da aplicação. Lógico que podemos colocar quanto código quisermos, mas não é recomendado.

Tudo isso também é verificado de acordo com os verbos HTTP²⁰ que são enviados junto com a requisição. Vamos comentar mais sobre esses verbos quando chegarmos na parte em que mencionamos REST.

Vamos imaginar que a nossa aplicação é alguma repartição pública (onde, nunca, eu digo, nunca, deixem ela funcionar na mesma velocidade ou com a mesma burocracia de uma repartição de verdade) onde vamos para fazer algum tipo serviço como protocolar documentos.

Chegando lá, para encontrarmos onde que devemos nos dirigir, tem logo na frente várias placas, que indicam as **rotas** para cada serviço disponibilizado. Depois que encontrarmos para onde devemos ir, encontrarmos alguns porteiros bem simpáticos (sério, existe!), entre eles o “Seu” Zé *Controlador*, que apesar de ser um cara esforçado e bom no atendimento, é meio esquecido e não sabe fazer nada mais que **delegar** o que recebe, no final entregando o que foi pedido, e ainda às vezes ficando com os méritos do trabalho ... enfim, o Zé Controlador é um cara que conhece tudo por ali mas precisa de um guia dos departamentos para encontrar para quem encaminhar o que pedimos.

O Zé também controla algumas coisas como a autorização de entrada no local, através de alguns crachás que ele distribui com algumas informações escritas em uma área chamada **cookies** (que está no crachá, ou seja, com o visitante), e anota essa informação em um livro que fica com ele chamado **sessões**, que fica guardado lá com ele.

Ah, ele também pode atender requisições pelo telefone ou email. Essa analogia vamos ver mais tarde, mas é muito importante pois permite que aplicações conversem com aplicações e não só com humanos, um fator muito importante na web moderna.

Model

O `model` é um objeto que, além de se conectar com o banco de dados através do `ORM`, é a camada do MVC mais “gordinha” e esperta, onde geralmente podemos descrever como contendo nossas “regras de negócio” em relação aos resultados que esperamos obter e inserir no nosso banco de dados. Algumas coisas como comportamentos de *triggers* e *stored procedures* inseridas anteriormente no banco de dados ficam nessa camada.

Antes que alguém ache estranho isso, sim, no código, afinal, o banco de dados na minha opinião tem que se prestar bem à função definida à ele, ou seja, fornecer dados de forma rápida, segura e consistente, não armazenar regras da aplicação! Inserir esse tipo de coisa no banco de dados além de perverter a função original dele, faz a aplicação ficar dependente do banco de dados, pois cada um tem a sua linguagem definida para esse tipo de coisa, deixando o seu sistema menos portável e com muito mais manutenção, especialmente se você tiver para a mesma aplicação clientes que utilizam bancos de dados distintos.

²⁰http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

Dentro de um `model`, podemos ter muito mais código do que temos em um `controller`, lógico, tentando otimizar o máximo que pudermos.

Podemos imaginar o `model` como um *nerd* gordinho introvertido que fica no fundo da repartição, o João Modelão. Ele quem faz a maior parte do serviço pesado, mas não gosta de lidar com gente, nunca aparece e é para ele quem o Zé Controlador geralmente pede a maioria das coisas que precisa saber para atender o pessoal.

View

O João Modelão é um sujeito muito inteligente, mas ele não tem vocação nenhuma para fazer alguma coisa mais bonita e organizada. Eu tenho um amigo que me lembra ele: a mesa do sujeito é um mar de coisas, mas ele consegue se organizar naquela bagunça. O João Modelão resolve qualquer problemas com os dados que é requisitado, porém antes de devolver os resultados para que o Zé Controlador retorne para quem os pediu, ele envia para o Louis Laview, que é um sujeito que tem o dom de fazer a **apresentação** das informações do jeito que foi requisitado, seja ele `HTML`, `JSON`, `XML`, imagens, etc.

Além disso, ele é responsável pelo design dos formulários que são preenchidos e entregues para o Zé Controlador.

Então, no MVC, a `view` é a camada visual exposta ao usuário, é o que, fora a URL que é enviada para o `controller`, o que vemos na aplicação.

Instalando

O Rails é instalado através do RubyGems, o gerenciador de pacotes da linguagem Ruby, e é composto por várias *gems* que são instaladas como dependência da *gem* principal, `rails`. Se você não conhece o funcionamento das *gems*, dê uma olhada no meu outro *ebook*, “[Conhecendo Ruby](#)”²¹.

Geralmente podemos instalar o Rails sem especificar uma versão, pois vai ser instalada sempre a mais recente. Vamos utilizar nesse livro a **versão 4.2** do Rails e fazer algumas observações para que está utilizando a versão 3 em alguma aplicação que já tenha sido feita.

Inclusive, enquanto iniciei o livro estávamos na versão 5, e alguns dias antes de ter todo o material formatado foi lançada a versão 5.1, com algumas mudanças. Vamos levar em conta que quando você estiver lendo esse material ele vai estar atualizado (e sendo atualizado) para alguma versão acima da 5.1. É importante também utilizar uma versão mais moderna do Ruby, que atualmente é a **2.4.1**. Recomendo a leitura do capítulo de instalação da linguagem no sistema operacional lá do meu outro *ebook*, “[Conhecendo Ruby](#)”, citado acima.

Instalar o Rails é fácil utilizando o seguinte comando, executado em um terminal:

```
1 $ gem install rails  
2 Fetching: rails-5.1.0.gem (100%)  
3 ...
```

Verificando a versão instalada:

```
1 $ rails -v  
2 Rails 5.1.0
```



Atenção

Vamos precisar de uma versão igual ou maior que a 5.

Logo a seguir vamos verificar se todas as *gems* necessárias estão instaladas, criando um projeto “zerado” do Rails que vai executar automaticamente o Bundler²², que é um gerenciador de dependências.



Dica

Vamos criar um aplicativo de uma livraria, onde os exemplos e modelagem podem não estar de acordo com algumas das melhores práticas, modelagem e otimização, mas onde foi prezada a didática. Não adianta eu tentar fazer discurso bonito se não dá para entender.

²¹<http://leanpub.com/conhecendo-ruby>

²²<http://gembundler.com>

Criando o projeto

Vamos criar o nosso projeto `bookstore`, onde podemos ver que o `bundler` é executado automaticamente no final para nós:

```
1 $ cd ~  
2 $ rails new bookstore  
3 create  
4 create README  
5 create Rakefile  
6 create config.ru  
7 create .gitignore  
8 create Gemfile  
9 create app  
10 ...  
11 run bundle install  
12 ...
```



Dica

O nome do projeto é importante na hora da criação, pois além do nome do diretório a ser criado, são configurados alguns parâmetros no arquivo da aplicação. Vamos escolher um nome de projeto em Inglês, mas nada impede que seja utilizado um em Português.



Dica

Vamos começar a contar com quantas linhas fazemos uma aplicação básica. Essa é a primeira linha.



O diretório atual é utilizado para a criação da aplicação. Se estiver em uma distribuição GNU/Linux como o Ubuntu, procure evitar o diretório `/tmp` pois ele é limpo sempre que o computador é desligado ou reiniciado.

O `Bundler` é definido como “o melhor jeito de gerenciar as dependências da sua aplicação”, as quais são definidas em um arquivo chamado `Gemfile`, presente no diretório da aplicação criada:

```

1 source 'https://rubygems.org'
2
3 git_source(:github) do |repo_name|
4   repo_name = "#{repo_name}/#{repo_name}" unless repo_name.include?("/")
5   "https://github.com/#{repo_name}.git"
6 end
7
8 # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
9 gem 'rails', '~> 5.1.0'
10 ...

```

Essas são algumas as dependências da aplicação, especificadas como uma série de *gems*, com números de versão opcionais especificadas, e no caso do `sass-rails`, utilizando um recurso chamado **Pessimistic Version Constraint**²³.



Dica

O uso do **Pessimistic Version Constraint** funciona da seguinte forma: se especificarmos algo como:

```
1 gem 'library', '~> 2.2'
```

o último dígito vai ser descartado, e o que sobrou vai ser incrementado. Nesse caso, estamos esperando uma versão da *gem* entre 2.2 e 3. Se tivéssemos escrito `-> 2.2.0`, seria maior ou igual que 2.2.0 e menor que 2.3.0.

Tem IDE?

Sim, existem algumas IDEs para se trabalhar com Rails, mas aqui não vamos precisar nada mais do que um terminal e um bom editor de texto. Eu gosto de sempre focar nos requisitos mínimos para desenvolver um projeto, então o meu ambiente é composto de uma distribuição GNU/Linux (atualmente o Ubuntu) com o terminal padrão da distribuição e o agora super-ultra-mega-master editor Vim (que alguns anos atrás chamavam de chulé-difícil-bagaraio-atrasado, até que caiu no *hype* de uma galera aí e deu no que deu) com vários *plugins* que facilitam *e muito* o desenvolvimento. Tem uma [apresentação](#)²⁴ que utilizo nos meus *workshops* de Vim, junto com o [repositório](#)²⁵ com os arquivos já todos configurados para começar a utilizar o Vim com os plugins que eu utilizo para desenvolvimento.

Mas acredititem, depois que nos acostumamos com um terminal e um bom editor com alguns recursos extras, as coisas paradoxalmente ficam mais fáceis.

Mas mesmo assim, se alguém quiser dar uma olhada em IDEs, existem algumas opções disponíveis, tanto de IDEs como editores mais “vitaminados”:

²³ <http://docs.rubygems.org/read/chapter/16#page74>

²⁴ <https://speakerdeck.com/taq/conhecendo-o-vim>

²⁵ <https://github.com/taq/workshop-vim>

- [Aptana Studio²⁶](#)
- [EasyEclipse²⁷](#)
- [RDE²⁸](#)
- [RubyMine²⁹](#)
- [Komodo³⁰](#)
- [Redcar³¹](#)
- [Arcadia³²](#)

Mas, como não uso nenhum, não posso recomendar qual seria o melhor.

Precisamos de um runtime JavaScript!

Uma das coisas que vamos precisar já, antes de mais nada, é de uma ferramenta de *runtime* JavaScript para executar a nossa aplicação, que vai depender dela para podermos utilizar o CoffeeScript, que nos permite gerar código JavaScript a partir de uma sintaxe parecida com Ruby. Para isso, temos duas opções.

The Ruby Racer

Como primeira opção, podemos instalar a *gem* do “The Ruby Racer”, que vai utilizar o *engine* V8³³. Para isso, se ainda não foi feito, vamos remover o comentário do começo da seguinte linha (#) do nosso Gemfile:

```
1 gem 'therubyracer', :platforms => :ruby
```

e rodando o bundler, onde, no meu caso, a gem já se encontrava instalada:

```
1 $ bundle install
2 ...
3 Using therubyracer (0.11.4)
4 ...
```

Node.JS

Ou, como segunda opção (e até mais fácil), podemos instalar o Node.JS³⁴. No Ubuntu podemos utilizar:

²⁶<http://aptana.com/>

²⁷<http://www.easyeclipse.org/site/distributions/ruby-rails.html>

²⁸http://homepage2.nifty.com/sakazuki/rde_en/index.html

²⁹<http://www.jetbrains.com/ruby/index.html>

³⁰<http://komodoide.com/>

³¹<http://redcareditor.com/>

³²<http://arcadia.rubyforge.org/>

³³<http://code.google.com/p/v8/>

³⁴<http://nodejs.org/>

```
1 sudo apt-get install nodejs
```

Inclusive, o Node.JS é uma plataforma para construção de aplicações de rede server side, em JavaScript. Podemos fazer um pequeno teste para ver como funciona, inserindo o seguinte conteúdo em um arquivo chamado `node.js`:

```
1 var http = require('http');
2 http.createServer(function (req, res) {
3   res.writeHead(200, {'Content-Type': 'text/plain'});
4   res.end('Hello World\n');
5 }).listen(1337, '127.0.0.1');
6 console.log('Server running at http://127.0.0.1:1337/');
```

E agora rodando o servidor:

```
1 $ node node.js
2 Server running at http://127.0.0.1:1337/
```

Digitando `http://127.0.0.1:1337` no navegador, obtemos:

```
1 Hello, World
```

Reparam que no método `writeHead`, são enviados os dois primeiros argumentos de forma similar à uma aplicação Rack, que veremos logo a seguir. Podemos, como outro teste do Node.JS, fazer um servidor TCP que vai repetir tudo o que foi enviado para ele, inserindo o seguinte conteúdo em um arquivo chamado `echo.js`:

```
1 var net = require('net');
2
3 var server = net.createServer(function (socket) {
4   socket.write('Echo server\r\n');
5   socket.pipe(socket);
6 });
7
8 server.listen(1337, '127.0.0.1');
```

Conectando via `telnet`³⁵ em `127.0.0.1` na porta `1337`, podemos ver que tudo que digitamos é repetido:

³⁵<http://pt.wikipedia.org/wiki/Telnet>

```
1 $ telnet 127.0.0.1 1337
2 Trying 127.0.0.1...
3 Connected to 127.0.0.1.
4 Escape character is '^]'.
5 Echo server
6 Oi!
7 Oi!
8 Testando o Node.JS!
9 Testando o Node.JS!
```

Agora temos tudo o que precisamos para a nossa aplicação.

Para interromper o processamento em ambos os casos, devemos apertar **Ctrl+C** (apesar que no caso do Telnet, ele vai ser mais teimoso).

Estrutura de arquivos e diretórios criada

Após criarmos a nossa aplicação, vamos ver que foi criado um diretório novo abaixo do diretório corrente com o nome da aplicação que requisitamos, no caso, bookstore. Vamos dar uma olhada na estrutura de diretórios e arquivos criados:

```
1 $ ls
2 drwxr-xr-x 13 taq taq 4,0K .
3 drwxr-xr-x 10 taq taq 4,0K ..
4 drwxr-xr-x 10 taq taq 4,0K app
5 drwxr-xr-x  2 taq taq 4,0K bin
6 drwxr-xr-x  5 taq taq 4,0K config
7 -rw-r--r--  1 taq taq 130 config.ru
8 drwxr-xr-x  2 taq taq 4,0K db
9 -rw-r--r--  1 taq taq 2,0K Gemfile
10 -rw-r--r--  1 taq taq 4,7K Gemfile.lock
11 drwxr-xr-x  7 taq taq 4,0K .git
12 -rw-r--r--  1 taq taq 536 .gitignore
13 drwxr-xr-x  4 taq taq 4,0K lib
14 drwxr-xr-x  2 taq taq 4,0K log
15 -rw-r--r--  1 taq taq  67 package.json
16 drwxr-xr-x  2 taq taq 4,0K public
17 -rw-r--r--  1 taq taq 227 Rakefile
18 -rw-r--r--  1 taq taq 374 README.md
19 drwxr-xr-x  9 taq taq 4,0K test
20 drwxr-xr-x  3 taq taq 4,0K tmp
21 drwxr-xr-x  2 taq taq 4,0K vendor
```

- **app** - É onde fica separado o MVC. Vamos dar uma olhada o que tem dentro:

```

1   $ ls app/
2   total 32K
3   drwxr-xr-x  8 taq taq .
4   drwxr-xr-x 11 taq taq ..
5   drwxr-xr-x  5 taq taq assets
6   drwxr-xr-x  5 taq taq channels
7   drwxr-xr-x  3 taq taq controllers
8   drwxr-xr-x  2 taq taq helpers
9   drwxr-xr-x  2 taq taq jobs
10  drwxr-xr-x  2 taq taq mailers
11  drwxr-xr-x  3 taq taq models
12  drwxr-xr-x  3 taq taq views
13
14
15  Ali podemos ver que temos diretórios chamados `models`, `views` e
16  `controllers`, entre outros. Esses com certeza vão ser os diretórios em
17  que mais vamos trabalhar em nossa aplicação.

```

- bin - Onde estão presentes os *scripts* bundle, rails e rake, entre outros.
- config - Aqui é onde ficam várias parametrizações da nossa aplicação, entre elas informações pertinentes aos ambientes (mais sobre eles abaixo), as configurações de rotas, internacionalização, configurações do banco de dados, de sessões, etc.
- config.ru - Utilizado pelo Rack. Mais sobre isso logo abaixo.
- db - Diretório onde são armazenados informações do bancos de dados (e até o próprio banco de dados de desenvolvimento, como vamos ver daqui a pouco), informações das migrações necessárias para adequar o banco e o arquivo seeds.rb que é utilizado para fazer a carga inicial no banco.
- Gemfile - É o arquivo que indica que gems que vamos utilizar na nossa aplicação. As gems são pacotes de funcionalidades/recursos extras para a linguagem Ruby, inclusive o próprio Rails sendo uma delas. Para mais informações sobre gems, consulte o meu *ebook* gratuito “[Conhecendo Ruby](#)”³⁶.
- Gemfile.lock - Arquivo que guarda as versões das gems que foram utilizadas na aplicação. Dessa maneira, podemos ter certeza que sempre que executarmos a aplicação, as gems das versões necessárias serão requisitadas.
- .gitignore - Arquivo que indica para o Git³⁷ quais os arquivos que ele deve ignorar no projeto, se formos utilizar o Git como o controle de versão da aplicação. Para um tutorial rápido e conciso sobre Git, consulte o meu *ebook* “[Conhecendo o Git](#)”³⁸.
- lib - Código extra que não faz parte de nenhuma gem.
- log - Diretório onde são gravados os logs da aplicação.
- package.json - Arquivo de gerenciamento de pacotes JavaScript pelo Yarn.
- public - Arquivos estáticos para serem servidos pela aplicação.
- Rakefile - Utilizado pelo comando rake, descrito logo abaixo.

³⁶<http://leanpub.com/conhecendo-ruby>

³⁷<http://git-scm.com/>

³⁸<http://leanpub.com/conhecendo-o-git>

- README.md - Arquivo com instruções/informações sobre a aplicação, escrito em [Mark-down³⁹](#).
- test - Onde ficam os testes da aplicação.
- vendor- Onde ficam os *plugins* que não são *gems*.

Rake

Aquele arquivo `Rakefile` que vimos ali acima é um arquivo que contém “receitas” ou *tasks* para o uso com o comando `rake`. O `rake` é um programa similar ao `make40` que utilizamos em C para a automatização de tarefas, facilitando nosso trabalho. As tarefas customizadas que por acaso desejarmos criar podem ser inseridas no diretório `lib/tasks` e podemos ver quais são as disponíveis por padrão na nossa aplicação utilizando o seguinte comando:

```

1 $ rails --tasks
2 rake about                               # List versions of all Rails framework
3 ks and the environment
4 rake app:template                         # Applies the template supplied by LO\
5 CATION=(/path/to/template) or URL
6 rake app:update                            # Update configs and some other initial
7 ally generated files (or use just update:configs or update:bin)
8 rake assets:clean[keep]                   # Remove old compiled assets
9 rake assets:clobber                       # Remove compiled assets
10 ...

```



Dica

A partir da versão 5, os comandos que eram executados explicitamente com o comando `rake` são executados com o comando `rails`. Antes era `rake` para uma coisa, `rails` para outra, e resolveram uniformizar de vez.

As tarefas também podem ser encadeadas, o que as deixa mais rápido. Levando em conta algumas das que vimos acima, podemos utilizar como exemplo:

```
1 $ rails db:drop db:create
```

Vamos utilizar algumas dessas tarefas durante todo o desenvolvimento da nossa aplicação. Não é necessário saber todas (uau, são muitas), mas acabamos por decorar algumas poucas delas. Se esquecer quais são, ou quiser ver quais são as mais novas após alguma alteração nas *gems* da aplicação, é só executar `rails -T` novamente, como mostrado acima.

Uma tarefa bem interessante é a `rails stats`, que vai nos retornar estatísticas da nossa aplicação:

³⁹ <https://pt.wikipedia.org/wiki/Markdown>

⁴⁰ <http://pt.wikipedia.org/wiki/Make>

	Name	Lines	LOC	Classes	Methods	M/C	LOC/M	
1	Controllers	3	3	1	0	0	0	
2	Helpers	2	2	0	0	0	0	
3	Jobs	2	2	1	0	0	0	
4	Models	3	3	1	0	0	0	
5	Mailers	4	4	1	0	0	0	
6	Channels	8	8	2	0	0	0	
7	Javascripts	29	4	0	1	0	2	
8	Libraries	0	0	0	0	0	0	
9	Tasks	0	0	0	0	0	0	
10	Controller tests	0	0	0	0	0	0	
11	Helper tests	0	0	0	0	0	0	
12	Model tests	0	0	0	0	0	0	
13	Mailer tests	0	0	0	0	0	0	
14	Integration tests	0	0	0	0	0	0	
15	Total	51	26	6	1	0	24	
16	Code LOC:	26	Test LOC:	0	Code to Test Ratio:	1:0.0		

Podemos ver que temos em um eixo as partes do sistema (controladores, modelos, etc) e nas colunas temos:

- Quantas linhas tem o arquivo
- Quantas linhas de código (Lines Of Code, excluindo comentários) tem no arquivo
- Quantas classes tem no arquivo
- Quantos métodos tem no arquivo
- Métodos por classe
- Linhas de código por método

E logo abaixo, a proporção de linhas de testes por linhas de código. Essa é uma métrica interessante para medir como está a cobertura de testes da sua aplicação.

Rack

Segundo o próprio site do Rack⁴¹, ele “provê uma interface mínima, modular e adaptável para o desenvolvimento de aplicações web em Ruby. Por lidar com requisições e respostas HTTP no jeito mais simples possível, unifica e destila a API para servidores web, frameworks web e software entre eles (os tão chamados *middlewares*) em umas uma chamada de método.”

⁴¹<https://github.com/rack/rack>

Ele foi criado para criar um padrão no jeito que os vários servidores e *frameworks* feitos em Ruby lidam com as requisições e respostas HTTP, consequentemente criando maior flexibilidade e facilidade de integração e uso. Se você ficou curioso sobre outros vários servidores e *frameworks* que utilizam (e utilizaram) o Rack, podemos listar:

Servidores e handlers populares

- WEBrick
- Thin
- Glassfish v3
- Phusion Passenger (que é o mod_rack para o Apache e nginx)
- Puma
- Unicorn

Servidores e handlers já obsoletos ou não tão utilizados

- Mongrel
- EventedMongrel
- SwiftipliedMongrel
- FCGI
- CGI
- SCGI
- LiteSpeed
- Ebb
- Fuzed
- Rainbows!
- Reel
- unixrack
- uWSGI
- Zbatery

Frameworks populares

- Ruby on Rails
- Sinatra
- Lotus
- Ramaze

Frameworks já obsoletos ou não tão utilizados

- Camping
- Cosest
- Espresso
- Halcyon
- Mack
- Maveric
- Merb
- Racktools::SimpleApplication
- Rum
- Sin
- Vintage
- Waves
- Wee

E muitos outros. Podemos fazer uma aplicação rack com apenas uma linha, digitando o seguinte conteúdo em um arquivo chamado `ola.ru` (o nome pode ser livre, mas a extensão deve ser `.ru`):

```
1 run ->(env) {[200, {"Content-Type" => "text/html"}, ["Olá, mundo, agora são #\
2 {Time.now.strftime('%H:%M')}!"]}]
```

Disparar a “aplicação” com o comando `rackup` (da gem `rack`, que já deve estar instalada por aí no terminal):

```
1 $ rackup ola.ru
2 [2014-04-08 20:30:21] INFO  WEBrick 1.3.1
3 [2014-04-08 20:30:21] INFO  ruby 2.1.1 (2014-02-24) [i686-linux]
4 [2014-04-08 20:30:21] INFO  WEBrick::HTTPServer#start: pid=31383 port=9292
```

Isso disparou um servidor web na porta 9292, e podemos verificar facilmente o seu conteúdo acessando `http://localhost:9292` no navegador, ou utilizando, como no exemplo abaixo, o comando `curl`:

```
1 $ curl http://localhost:9292
2 Olá, mundo, agora são 20:30!
```

Para retornar o conteúdo para o navegador, só precisamos de uma Proc/lambda (*ebook* de Ruby[^conhecendo-ruby] para quem não souber o que é!) onde são retornados:

1. Um código de status HTTP⁴², muito importante conhecer isso em desenvolvimento web!

⁴²<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

2. O content (MIME) type do documento⁴³, outra coisa muito importante
3. O texto da resposta, em formato de um Array com uma ou várias linhas

Os códigos de status HTTP, como mencionado acima, são uma coisa muito importante para conhecer quando estamos nos envolvendo com desenvolvimento web. Eles fornecem um protocolo de comunicação eficiente de aplicação para aplicação, ou seja, de computador para computador, diferente do tipo de resultado que vemos em interfaces visuais de comunicação de computador para humano. Não precisamos saber todos os códigos, mas é bom manter em mente alguns essenciais:

- **200** - Conteúdo ok - Padrão de resposta para solicitações HTTP sucesso. A resposta real dependerá do método de solicitação usado. Em uma solicitação GET, a resposta conterá uma entidade que corresponde ao recurso solicitado. Em uma solicitação POST a resposta conterá a descrição de uma entidade, ou contendo o resultado da ação.
- **301** - Movido permanentemente para outra URL. Segundo práticas de SEO, esse é o melhor tipo de redirecionamento.
- **302** - Movido temporariamente para outra URL.
- **404** - Não encontrado. A URL requisitado não foi encontrada.
- **500** - Erro interno do servidor. Alguma coisa ocorreu de errado na sua aplicação.

⁴³http://en.wikipedia.org/wiki/MIME_type

Ambientes do Rails

Agora que já temos a aplicação básica criada, podemos especificar qual ambiente em que ela está sendo executada. O Rails permite termos configurações diferentes para cada ambiente, de forma a fazer com que a aplicação mude o seu comportamento baseada nas necessidades de cada um deles. Os ambientes em que podemos executar nossa aplicação no Rails são 3:

- **Desenvolvimento** é o ambiente onde escrevemos e desenvolvemos o código e os testes da aplicação. Nesse ambiente qualquer alteração que fizemos na aplicação vai automaticamente entrar em vigor, o que facilita muito enquanto estamos desenvolvendo.
- **Teste** é o ambiente onde rodamos os testes da nossa aplicação.
- **Produção** é o ambiente onde nossa aplicação vai rodar para valer, onde existe várias otimizações sendo feitas, como por exemplo, o *caching* dos arquivos carregados, e onde diferentemente do ambiente de desenvolvimento, precisaremos efetuar um *restart* (não, não é aquela banda coloridinha que fez sucesso e sumiu) a nossa aplicação.

Controlando qual ambiente está ativo

O controle de qual ambiente que está ativo é feito através de uma variável de ambiente ⁴⁴ chamada RAILS_ENV. Se ela não estiver definida, o ambiente de desenvolvimento development será utilizado. Podemos verificar dentro da aplicação qual ambiente está sendo utilizado utilizando Rails.env, que vai retornar o ambiente como uma *String*, ou testar o ambiente utilizando Rails.env.test?, Rails.env.development? e Rails.env.production?.



Dica

Podemos executar no terminal qualquer comando indicando qual ambiente ele vai utilizar especificando o valor da variável RAILS_ENV logo antes do comando, como no exemplo abaixo, usando production:

```
RAILS_ENV=production <comando>
```

Bancos de dados dos ambientes

Para cada ambiente desses, o Rails tem uma configuração de banco de dados separada. Essas configurações podem ser vistas no arquivo config/database.yml:

⁴⁴http://pt.wikipedia.org/wiki/Vari%C3%A1vel_de_ambiente

```
1 # SQLite version 3.x
2 #   gem install sqlite3
3 #
4 #   Ensure the SQLite 3 gem is defined in your Gemfile
5 #   gem 'sqlite3'
6 #
7 default: &default
8   adapter: sqlite3
9   pool: 5
10  timeout: 5000
11
12 development:
13   <<: *default
14   database: db/development.sqlite3
15
16 # Warning: The database defined as "test" will be erased and
17 # re-generated from your development database when you run "rake".
18 # Do not set this db to the same as development or production.
19 test:
20   <<: *default
21   database: db/test.sqlite3
22
23 production:
24   <<: *default
25   database: db/production.sqlite3
```

Essas configurações são ótimas pois isolam os 3 ambientes, nos permitindo não fazer nenhuma lambança no ambiente de produção. Para não ter desculpa de começar a já meter a mão na massa, o Rails já prepara uma configuração de banco de dados simples para começarmos a desenvolver, e outra para que executemos nossos testes, o que não nos dá a desculpa que fazer testes é alguma coisa complicada de se fazer. **Façam testes na sua *app*, eles já estão prontos para receberem conteúdo.**

Podemos ver que cada banco de dados do SQLite3 recebeu um nome de arquivo diferente. Podemos adequar esse arquivo à vontade para as configurações dos ambientes que precisamos, e também dos bancos de dados utilizados, bastando instalar os *drivers* dos bancos e mudar o valor da chave em *adapter*.

Podemos ver que não temos nenhum arquivo do banco criado em nosso diretório db:

```
1 $ ls db/
2 seeds.rb
```

E agora pedir para serem criados (não tem problema se der alguma mensagem de arquivo já existe):

```
1 $ rails db:create
2 Created database 'db/development.sqlite3'
3 Created database 'db/test.sqlite3'
4
5 $ ls db/
6 -rw-r--r-- 1 taq taq    0 development.sqlite3
7 -rw-r--r-- 1 taq taq  370 seeds.rb
8 -rw-r--r-- 1 taq taq    0 test.sqlite3
```

A partir desse momento, temos nossos ambientes, em relação ao banco de dados, totalmente isolados. Convém prestar muita atenção pois, dada a praticidade do Rails de funcionar “out-of-the-box”⁴⁵, podemos esquecer de trocar a configuração do ambiente de produção e o sistema entrar em produção utilizando um banco de dados SQLite!

⁴⁵http://en.wikipedia.org/wiki/Out_of_the_box

Iniciando o servidor

Podemos iniciar o servidor utilizando

```
1 $ rails s
```

ou

```
1 $ rails server
```

o que vai resultar no servidor Puma sendo inicializado no diretório corrente, “subindo” a nossa aplicação na porta 3000:

```
1 => Booting Puma
2 => Rails 5.0.1 application starting in development on http://localhost:3000
3 => Run `rails server -h` for more startup options
4 Puma starting in single mode...
5 * Version 3.7.0 (ruby 2.3.1-p112), codename: Snowy Sagebrush
6 * Min threads: 5, max threads: 5
7 * Environment: development
8 * Listening on tcp://0.0.0.0:3000
9 Use Ctrl-C to stop
```



Dica

Essa é a segunda linha da nossa aplicação básica.



Dica

Para inicializar o servidor em uma porta diferente, podemos passar a opção `-p` na linha de comando. Se quisermos a porta 8081:

```
1 $ rails s -p 8081
```



Dica

Para testarmos uma aplicação sem utilizar `localhost` ou `127.0.0.1`, podemos utilizar `http://1vh.me`, que vai apontar para a máquina local. Isso é bem útil quando estamos desenvolvendo aplicações que tem subdomínios.

Se abrirmos a URL `http://localhost:3000` em um navegador, podemos ver uma página como a seguinte:



Yay! You're on Rails!



Rails version: 5.0.1

Ruby version: 2.3.1 (x86_64-linux)

Página inicial do Rails

Esse é o `index.html`, que é gerado (a partir do Rails 4) dinamicamente para nossa aplicação. Após começarmos a desenvolver a nossa *app*, podemos ou deixar com alguma mensagem especificando que o site está sendo desenvolvido (não aquele operário com picareta com a mensagem “em construção”, por favor ;-), inserindo um arquivo `index.html` no diretório `public`:

```

1 $ ls -lah public/
2 total 32K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 13 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 728 404.html
6 -rw-r--r-- 1 taq taq 711 422.html
7 -rw-r--r-- 1 taq taq 728 500.html
8 -rw-r--r-- 1 taq taq 0 favicon.ico
9 -rw-r--r-- 1 taq taq 204 robots.txt

```

Ou podemos configurar a rota do root da aplicação para outro lugar, como vamos fazer no decorrer do livro.

No diretório `public` se encontram alguns arquivos estáticos para respostas de status do servidor⁴⁶, como:

- **404.html** Retornada quando a URL encontrada não existe.
- **422.html** Retornada para erros semânticos indicando que o documento é malformado.
- **500.html** Retornada indicando que ocorreu algum erro interno no servidor.

Também temos os arquivos `favicon.ico` e `robots.txt`.

Spring

A partir do Rails 4.1, temos já pronto para uso o Spring⁴⁷, que é um “pré-carregador” da aplicação Rails. e é uma mão na roda em modo de desenvolvimento, pois mantém a aplicação rodando em *background*, não sendo necessário recarregá-la quando executando diversos comandos e procedimentos que vamos ver ao decorrer do livro, minimizando o tempo de *boot*.

Para verificar se o Spring está ativo no projeto, **no ambiente de desenvolvimento**, podemos verificar se existe a seguinte linha no arquivo `Gemfile`:

```
1 gem 'spring', group: :development
```

E verificar se os arquivos foram `springfados` (ei, não fui eu quem inventei o termo ;-):

```

1 $ bundle exec spring binstub --all
2 * bin/rake: spring already present
3 * bin/rails: spring already present

```

Para verificarmos se está ativo quando rodando o servidor, é só digitar no terminal:

⁴⁶http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

⁴⁷<https://github.com/rails/spring>

```
1 $ spring status
```

O que deve retornar algo como:

```
1 Spring is running:  
2  
3 12017 spring server | bookstore | started 15 secs ago
```

Ou, se não estiver rodando:

```
1 $ spring status  
2 Spring is not running.
```

Algumas vezes, podemos ter alguns problemas quando precisamos recarregar de vez a aplicação, e ela se encontra carregada pelo Spring. nesse caso, precisamos pedir para que os processos sejam finalizados utilizando `spring stop`:

```
1 $ spring status  
2 Spring is running:  
3  
4 12883 spring server | bookstore | started 5 mins ago  
5  
6 $ spring stop  
7 Spring stopped.  
8  
9 $ spring status  
10 Spring is not running.
```



Dica

Se alguma coisa que você configurou/alterou aparentemente não deu resultado nem que a vaca tussa, pode ser que o estado anterior esteja ainda no Spring. Nesse caso, utilize algum dos métodos acima para pará-lo e recarregar a aplicação.

Também é possível parar o Spring quando o terminal é fechado. Podemos também desabilitar o Spring para algum comando que vamos rodar declarando a variável `DISABLE_SPRING` com qualquer valor, no terminal, antes de rodar o comando:

```
1 $ export DISABLE_SPRING=1  
2 $ <comando>
```

Começando a construir a aplicação

Pluralização

Um importante no nosso projeto é decidir se vamos criar nossos modelos (diga-se “tabelas do banco”) em Português em ou Inglês. Digo isso pois o Rails utiliza muito os recursos de *pluralização* e *singularização*, sendo que em Português temos muitos detalhes para esse tipo de coisa, e em Inglês, além de ter menos, o Rails já está pronto para isso.

Podemos verificar como funciona a pluralização e singularização abrindo o terminal do Rails, que é um REPL⁴⁸ Ruby (o `irb`) que permite acesso aos recursos do Rails:

```
1 $ rails c
2 Loading development environment
3 "person".pluralize
4 => "people"
5
6 "people".singularize
7 => "person"
8
9 "city".pluralize
10 => "cities"
11
12 "countries".singularize
13 => "country"
14
15 "book".pluralize
16 => "books"
17
18 "livro".pluralize
19 => "livros"
20
21 "países".singularize
22 => "paíse"
```

Podemos resolver isso com o uso de **inflexões**, alterando o arquivo `config/initializers/inflections.rb`:

⁴⁸http://en.wikipedia.org/wiki/Read%20%93eval%20%93print_loop

```

1 ActiveSupport::Inflector.inflections do |inflect|
2   inflect.plural /(ao)$/i, 'oes'
3   inflect.singular /oes$/i, 'ao'
4   inflect.plural /(or)$/i, '\1es'
5   inflect.singular /(or)es$/i, '\1'
6   inflect.plural /el$/i, '\1eis'
7
8   inflect.singular /(eis)$/i, 'el'
9   inflect.plural /il$/i, 'is'
10  inflect.singular /vis$/i, 'vil'
11  inflect.plural /is$/i, 'ises'
12  inflect.singular /ises$/i, 'is'
13 end

```

Testando novamente:

```

1 $ rails c
2 "livro".pluralize
3 => "livros"
4
5 "países".singularize
6 => "país"
7
8 "racão".pluralize
9 => "rácões"
10
11 "autor".pluralize
12 => "autores"
13
14 > "imóvel".pluralize
15 => "imóveis"
16
17 "civil".pluralize
18 => "cívics"

```

Um exemplo de inflexões em Português bem completo pode ser encontrado em [nesse arquivo⁴⁹](#). Mas vamos construir nossa aplicação usando o suporte nativo do Rails para a língua Inglesa para evitarmos perder muito tempo com detalhes desse tipo agora que estamos aprendendo, então, vamos deixar o arquivo do jeito que estava antes, removendo as linhas que utilizamos para ver o recurso sendo utilizado em Português.

⁴⁹ https://raw.github.com/tapajos/brtraducao/master/lib/brtraducao/inflector_portuguese.rb

Criando o primeiro scaffold

Tudo acertado na parte de pluralização (vamos utilizar o Inglês mesmo), podemos criar nosso primeiro scaffold⁵⁰, que é uma técnica utilizada por alguns *frameworks*, como o Rails, para prover uma estrutura CRUD⁵¹, que nos permite *criar, recuperar, atualizar e deletar* informações, dessa forma agilizando muitas das tarefas que precisamos fazer rotineiramente em uma aplicação.

Vamos criar um modelo chamado Person, um controlador chamado people, juntamente com as views necessárias. Esse modelo, inicialmente, vai conter os seguintes campos/propriedades:

- **name** - nome
- **email** - email
- **password** - senha
- **born_at** - data de nascimento
- **admin** - flag de administrador

Podemos, no momento de criar nosso scaffold, designar (ou não) as colunas/propriedades do nosso modelo, utilizando no terminal o comando `rails g`, que é uma abreviação de `rails generator`. Mas é **muito indicado** que já façamos isso, pois as views vão ser criadas já com as colunas nos tipos que precisamos:

```

1 $ rails g scaffold Person name:string email:string password:string born_at:da\
2 te admin:boolean
3     invoke  active_record
4     create    db/migrate/20170226142016_create_people.rb
5     create    app/models/person.rb
6     invoke  test_unit
7     create    test/models/person_test.rb
8     create    test/fixtures/people.yml
9     invoke  resource_route
10    route    resources :people
11    ...

```



Dica

Essa é a terceira linha da nossa aplicação básica.

Usando a *convention over configuration (COC)*, o Rails automaticamente cria uma coluna chamada `id` como chave primária da tabela, auto-incrementável, e também colunas chamadas `created_at` e `updated_at`, que são utilizadas, respectivamente, como seus nomes já denunciam, para marcar quando que um registro foi criado e atualizado. A coluna `id` é criada como uma coluna numérica, que não permite valores nulos, e com a característica de auto-incremento, ou seja, vai sempre incrementar em 1 a cada valor que for criado.

⁵⁰ [http://en.wikipedia.org/wiki/Scaffold_\(programming\)](http://en.wikipedia.org/wiki/Scaffold_(programming))

⁵¹ <http://pt.wikipedia.org/wiki/CRUD>

Migrations

As migrations baseiam-se no princípio que podemos fazer alterações no nosso banco de dados tanto para frente, como para trás, através de um *rollback*, no Rails 3.0.x mapeando os métodos up e down, respectivamente, e no Rails 3.1.x, através do método change. Podemos escolher qual a abordagem que vamos seguir, lembrando que o método change só permite reversões para os seguintes métodos:

- add_column
- add_index
- add_timestamp
- create_table
- remove_timestamps
- rename_column
- rename_index
- rename_table

São comportamentos muito úteis no desenvolvimento quando em produção, quando temos a praticidade do nosso banco de produção ser alterado de acordo com as últimas alterações que fizemos de uma maneira prática, rápida e segura. Vamos dar uma olhada na migration que foi criada no scaffold executado acima. As migrations se encontram, em ordem de nome de arquivo (gerado de acordo com a data de criação, o seu **com certeza** vai estar com números diferentes no início do nome do arquivo), no diretório db/migrate:

```
1 $ ls db/migrate/
2 total 12K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 3 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 246 20170226142016_create_people.rb
```

Agora vendo o conteúdo do arquivo:

```
1 $ cat db/migrate/20170226142016_create_people.rb
2 class CreatePeople < ActiveRecord::Migration[5.0]
3   def change
4     create_table :people do |t|
5       t.string :name
6       t.string :email
7       t.string :password
8       t.date :born_at
9       t.boolean :admin
10      t.timestamps
11    end
12  end
13 end
```

Ali podemos ver:

1. ActiveRecord::Migration[5.0] - o número de versão é uma novidade a partir do Rails 5.
2. create_table :people - cria a tabela people (o plural, em Inglês, de Person)
3. t.string :name - cria uma coluna do tipo string com o nome name
4. t.string :email - cria uma coluna do tipo string com o nome email
5. t.string :password - cria uma coluna do tipo string com o nome password
6. t.date :born_at - cria uma coluna do tipo date com o nome born_at
7. t.boolean :admin - cria uma coluna do tipo boolean com o nome admin
8. t.timestamps - cria as colunas created_at e updated_at, ambas com o tipo timestamp, e já utiliza null: false para indicar que as colunas não podem ter conteúdo nulo.



Dica

Se não quisermos que sejam criadas as colunas created_at e updated_at, deixando o Rails automaticamente tomar conta delas quando criarmos ou atualizarmos um registro, é só remover a linha da `t.timestamps` da migration.

Nomes de tabelas e chaves primárias customizadas

Não vamos utilizar isso na nossa aplicação, portanto, não alterem nada no código como mostrado a seguir, mas se quisermos e precisarmos (por um bom motivo!), podemos sobreescrivê-lo que o Rails lida com nossas tabelas e chaves primárias, abrindo mão nesse caso de um pouco do *convention over configuration*, especificando o nome da tabela mapeada pelo objeto e sua(s) chave(s) primária(s), utilizando `set_table_name` e `set_primary_key` no arquivo criado para o modelo, que se encontra em `app/models/person.rb`:

```
1 class Person < ActiveRecord::Base
2   self.table_name :pessoa
3   self.primary_key :cpf
4 end
```



Atenção

Para utilizar como chave primária outra coluna que não seja a `id`, temos que especificar na *migration* que não é para ela criar a coluna `id`, da seguinte forma:

```
1 create_table :people, id: false do |t|
```

Para chaves primárias compostas, podemos instalar a gem `composite_primary_keys`⁵²:

⁵²<http://compositekeys.rubyforge.org>

```
1 $ gem install composite_primary_keys
```

Que nos permite fazer algo como:

```
1 class Person < ActiveRecord::Base
2   set_table_name :pessoa
3   set_primary_keys :cpf, :rg
4 end
```

No caso de precisar criar uma nova chave primária que precise ser auto incrementada utilizando *sequences*⁵³, temos também que usar o método `set_sequence_name`, como por exemplo:

```
1 class Person < ActiveRecord::Base
2   set_table_name :pessoa
3   set_primary_key :codigo
4   set_sequence_name "PessoasCodigos"
5 end
```

Se precisarmos lidar com tabelas legadas com tipos de dados fora do padrão, como por exemplo, uma coluna chamada `ativa`, cujo valor é S ou N, que seria um valor booleano de `true` ou `false`, a partir do Rails 5 temos disponíveis o método `attribute`, onde descrevemos o atributo que queremos interceptar e o tipo de dados o qual ele será convertido. Se essa coluna estivesse com um valor que representasse um booleano em Inglês, como por exemplo, yes ou t, ou um valor como 1 ou 0, poderíamos utilizar o `attribute` dessa maneira:

```
1 class Person < ActiveRecord::Base
2   attribute :ativa, :boolean
3 end
```

Como estamos com ela “aportuguesada” (o que também deve ser levado em conta junto com as outras convenções demonstradas acima, como a pluralização, etc), podemos especificar uma classe para fazer a conversão, criando um tipo de atributo customizado. Antes de mais nada temos que ter a classe, aqui no exemplo criada em `app/services/sim_nao_type.rb`, chamada `SimNaoType`:

⁵³http://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_6015.htm

```

1 class SimNaoType < ActiveRecord::Type::Boolean
2   # do objeto para o banco
3   def serialize(value)
4     return value if value.kind_of?(String)
5     value ? 'S' : 'N'
6   end
7
8   # do banco para o objeto
9   def deserialize(value)
10    return value if [ true, false ].include?(value)
11    value.to_s.upcase == 'S'
12  end
13 end

```

Ali temos o método `serialize`, que vai converter o valor presente no objeto para o que vai ser persistido no banco (ou seja, converter de `true` ou `false` para `S` ou `N`) e o método `deserialize`, que vai converter o valor persistido no banco para o valor necessário no objeto (fazendo o inverso da operação anterior, ou seja, convertendo de `S` e `N` para `true` ou `false`).

Agora vamos indicar no modelo que ele usará esse tipo de atributo customizado:

```

1 class Pessoa < ApplicationRecord
2   attribute :ativa, SimNaoType.new
3 end

```

Com isso, agora podemos lidar com nosso objeto sem nos preocupar com os `Ss` e `Ns`, levando em conta somente os valores booleanos deles:

```

1 > p = Person.create(name: 'Eustáquio Rangel', ativa: true)
2   (0.1ms) begin transaction
3   SQL (0.3ms) INSERT INTO "people" ("name", "ativa", "created_at",
4   "updated_at") VALUES (?, ?, ?, ?)  [["name", "Eustáquio Rangel"], ["ativa",
5   "S"], ["created_at", 2017-04-14 16:47:27 UTC], ["updated_at", 2017-04-14
6   16:47:27 UTC]] (12.0ms) commit transaction
7 => #<Person id: 1, name: "Eustáquio Rangel", ativa: true, created_at:
8   "2017-04-14 16:47:27", updated_at: "2017-04-14 16:47:27">
9
10 > Person.where(ativa: true)
11 Person Load (0.2ms) SELECT "people".* FROM "people" WHERE "people"."ativa"\=
12 =
13   ?  [["ativa", "S"]]
14 => #<ActiveRecord::Relation [#<Person id: 1, name: "Eustáquio Rangel", ativa:
15   true, created_at: "2017-04-14 16:47:27", updated_at: "2017-04-14 16:47:27">]\=
16 >
17
18 > p.ativa = false

```

```

19 => false
20
21 > p.save
22   (0.1ms) begin transaction
23 SQL (0.3ms) UPDATE "people" SET "ativa" = ?, "updated_at" = ? WHERE
24 "people"."id" = ? [{"ativa": "N"}, {"updated_at": "2017-04-14 16:47:52 UTC"}, 
25 ["id", 1]]
26   (7.9ms) commit transaction
27 => true
28
29 > Person.where(ativa: true)
30 Person Load (0.1ms) SELECT "people".* FROM "people" WHERE "people"."ativa"\ 
31 =
32 ? [{"ativa": "S"}]
33 => #<ActiveRecord::Relation []>
34
35 > Person.where(ativa: false)
36 Person Load (0.1ms) SELECT "people".* FROM "people" WHERE "people"."ativa"\ 
37 =
38 ? [{"ativa": "N"}]
39 => #<ActiveRecord::Relation [#<Person id: 1, name: "Eustáquio Rangel", ativa: 
40 false, created_at: "2017-04-14 16:47:27", updated_at: "2017-04-14 16:47:52">\ 
41 ]>

```

Podemos ver que todas as consultas no banco levaram em conta o tipo de dados persistido lá, ou seja, Strings com S ou N.

Uma palestra bem interessante sobre isso é Migrating legacy data⁵⁴.

Com a migration pronta, devemos preparar o banco de dados, ou seja, criar a tabela que vai ser mapeada pelo nosso objeto Person, que acabou de ser criado no diretório que contém os nossos modelos, app/models, aqui sem qualquer das configurações extras mostradas acima:

```

1 $ ls app/models/
2 person.rb
3
4 $ cat app/models/person.rb
5 class Person < ActiveRecord::Base
6 end

```

⁵⁴<http://www.slideshare.net/phuesler/migrating-legacy-data-1861106>



Após termos o nosso ambiente Rails configurado com tudo o que precisamos (linguagem Ruby e gems necessárias para o Rails instaladas), o Rails dificilmente vai ficar na frente do desenvolvedor na hora de começar a desenvolver sua aplicação, tanto que nem vamos precisar configurar o banco de dados para começar a desenvolver, pois já existem **bancos de dados configurados e prontos para o desenvolvimento, testes e produção**, utilizando o banco de dados SQLite, que pode ser utilizado sem problemas para as fases de desenvolvimento e testes, mas que **deve ser trocado** para a produção. As configurações dos bancos no Rails se encontram no arquivo `config/database.yml`.

Como visto, o mapeamento de uma classe com uma tabela no esquema ORM em Rails chega a ser ridículo de simples comparados com alguns outros *frameworks* com ORM: basta criar a classe herdando de `ActiveRecord::Base`. No conceito de **convention over configuration**, o nome da classe vai ser pluralizado para refletir o nome da tabela, e no nosso exemplo o objeto vai ser `Person` e a tabela, `People`.

Podemos (e geralmente, devemos) alterar essa `migration` antes de executá-la, para se adequar mais precisamente às nossas necessidades:

```

1 class CreatePeople < ActiveRecord::Migration[5.0]
2   def change
3     create_table :people do |t|
4       t.string :name, limit: 50, null: false
5       t.string :email, limit: 100, index: { unique: true }
6       t.string :password, limit: 100
7       t.date :born_at
8       t.boolean :admin
9       t.timestamps null: false
10    end
11  end
12 end

```

Ali utilizamos `limit` para especificar o tamanho da coluna, `null` para indicar que ela não pode ser nula, `index` para indicar que queremos criar um índice na coluna `email` e `unique` indicando que é um índice único (`unique: true`). Em versões anteriores à 4, para criar um índice devemos chamar explicitamente o método `add_index`.



Dica

A partir do Rails 5, podemos incluir comentários sobre as tabelas no banco de dados. Para isso podemos passar os comentários através da opção `comment` na `migration`:

```
1   create_table :people, comment: 'Tabela de pessoas' do |t|
```

Vamos dar uma olhada no nosso banco, antes de executar a `migration`, com o comando `rails dbconsole`:

```
1 $ rails dbconsole
2 SQLite version 3.14.1 2016-08-11 18:53:32
3 Enter ".help" for usage hints.
4 sqlite> select name from sqlite_master where type='table' order by name;
5 sqlite>
```



Dica

Podemos listar as tabelas de um banco de dados SQLite com o comando mostrado acima.

Não foi listada nenhuma tabela, o banco está “pelado”. Podemos inclusive ver que no diretório onde estão os bancos de dados (ei, podem ser vários, temos 3 ambientes!), que é o db, não consta nenhum arquivo referente aos bancos de dados (aquele seeds.rb vamos ver mais tarde):

```
1 $ ls db/
2 total 12K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 12 taq taq 4,0K ..
5 drwxr-xr-x 2 taq taq 4,0K migrate
6 -rw-r--r-- 1 taq taq 370 seeds.rb
```

Agora podemos executar a migration. Para isso, utilizamos o comando `rails db:migrate`:

```
1 $ rails db:migrate
2 == 20170226142016 CreatePeople: migrating ======\n
3 ==
4 -- create_table(:people)
5   -> 0.0026s
6 == 20170226142016 CreatePeople: migrated (0.0027s)
7 =====
```



Dica

Essa é a quarta linha da nossa aplicação básica.

Verificando novamente no banco de dados:

```

1 SQLite version 3.14.1 2016-08-11 18:53:32
2 Enter ".help" for usage hints.
3 sqlite> select name from sqlite_master where type='table' order by name;
4 ar_internal_metadata
5 people
6 schema_migrations
7 sqlite_sequence
8 sqlite>
```

Pudemos ver que a nossa tabela foi criada, junto com algumas outras de controle do *framework*, como a `schema_migrations`, que armazena o número da última `migration` executada. Dando outra olhada no diretório db:

```

1 $ ls db/
2 total 52K
3 drwxr-xr-x 3 taq taq 4,0K .
4 drwxr-xr-x 12 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 32K development.sqlite3
6 drwxr-xr-x 2 taq taq 4,0K migrate
7 -rw-r--r-- 1 taq taq 1,2K schema.rb
8 -rw-r--r-- 1 taq taq 370 seeds.rb
```

Podemos ver que foram criados dois arquivos novos, `development.sqlite3`, que é o banco de dados utilizado para o modo de desenvolvimento, e o arquivo `schema.rb`, que armazena informações da estrutura do banco de dados.

Se por acaso erramos ou esquecemos de algo na `migration`, podemos tanto fazer `outra migration` para consertar, ou executar um `rollback` (igual nos bancos de dados), que vai devolver o nosso banco de dados à um estado prévio do que se encontra atualmente. Podemos executar vários `rollbacks` se necessário:

```

1 $ rails db:rollback
2 == 20170226142016 CreatePeople: reverting =====\\
3 ==
4 -- drop_table(:people)
5   -> 0.0014s
6 == 20170226142016 CreatePeople: reverted (0.0066s)
7 =====
8
9 $ rails dbconsole
10 SQLite version 3.14.1 2016-08-11 18:53:32
11 Enter ".help" for usage hints.
12 sqlite> select name from sqlite_master where type='table' order by name;
13 ar_internal_metadata
14 schema_migrations
15 sqlite_sequence
16 sqlite>
```

Agora temos que rodar a nossa migration com db:migrate novamente, para deixar o nosso banco de dados pronto para o que foi requisitado no scaffold. Não esqueçam de rodar novamente a migration!



Dica

Para ver como está o estado corrente das migrations, podemos utilizar o comando `rails db:migrate:status`.

Nesse ponto, já temos o resultado do nosso scaffold visível no navegador. Como criamos um controlador chamado `people`, podemos acessá-lo através de `http://localhost:3000/people:`

People

Name	Email	Password	Born at	Admin

[New Person](#)

Controlador de pessoas



A URL acima especifica:

- `http://` - O protocolo utilizado
- `localhost` - O host
- `:3000` - A porta
- `people` - O caminho

A RFC das URLs, feita por Tim Berners-Lee, se encontra aqui: [aqui⁵⁵](#) e outro artigo bem legal [aqui⁵⁶](#)

Fazendo aqui uma recapitulação de como criamos nossa aplicação básica:

1. `rails new bookstore`
2. `rails s`
3. `rails g scaffold Person name:string email:string password:string born_at:date admin:boolean`
4. `rails db:migrate`

⁵⁵ <http://www.ietf.org/rfc/rfc1738.txt>

⁵⁶ <http://blog.lunatech.com/2009/02/03/what-every-web-developer-must-know-about-url-encoding>

Para aplicações baseadas em `scaffolds`, é só seguir os passos 2 e 3 até satisfazer as necessidades. Lógico que podemos fazer muito mais do que isso, mas se você está aprendendo Rails agora, pode usar isso como “colinha” para impressionar os amigos e fazer uma aplicação básica em menos de 2 minutos. ;-)

Asset pipeline

A partir do Rails 3.1.x, temos suporte para a *asset pipeline*, que é uma forma de otimizar os arquivos CSS e JavaScript da nossa aplicação, concatenando-os e comprimindo-os (por exemplo, removendo espaços em branco que para nós, desenvolvedores, são essenciais para melhor compreensão do código, mas que não fazem a menor diferença para o navegador), entregando aos navegadores compridos utilizando o algoritmo Gzip⁵⁷ (suportado na maioria dos servidores web e navegadores) e também para a utilização de escrever esses arquivos a partir de outros recursos como o Sass e o CoffeeScript (que geram, respectivamente, CSS e JavaScript padrões para os navegadores, a partir de sintaxes próprias e diferenciadas), com o intuito de aumentar a produtividade.

A *library* que cuida desse processamento é a Sprockets, que podemos verificar que já está instalada:

```
1 $ gem search -l sprockets
2
3 *** LOCAL GEMS ***
4 sprockets (3.7.1)
5 sprockets-rails (3.2.0)
```

Podemos verificar também se a `gem` faz parte das alocadas para o projeto, utilizando o `bundle`:

```
1 $ bundle show sprockets
2 /home/taq/.rvm/gems/ruby-2.3.1/gems/sprockets-3.7.1
```

A *asset pipeline* é habilitada por padrão, e se quisermos, podemos desabilitá-la no arquivo `config/initializers/assets.rb` indicando `config.assets.enabled` como `false`. Se quisermos desabilitar logo quando criamos a aplicação, podemos utilizar as opções `-S` ou `--skip-sprockets` juntamente com `rails new <nome da aplicação>`.

Os arquivos para uso da *asset pipeline* se encontram em `app/assets`:

```
1 $ ls app/assets/
2 total 20K
3 drwxrwxr-x 5 taq taq 4,0K .
4 drwxrwxr-x 8 taq taq 4,0K ..
5 drwxrwxr-x 2 taq taq 4,0K config
6 drwxrwxr-x 2 taq taq 4,0K images
7 drwxrwxr-x 2 taq taq 4,0K javascripts
8 drwxrwxr-x 2 taq taq 4,0K stylesheets
```

⁵⁷<http://pt.wikipedia.org/wiki/Gzip>

Opa, mas olhem só, temos um diretório `images` lá! O que a *asset pipeline* vai fazer com ele, grudar todas as imagens? Até daria para fazer algo do tipo em imagens menores para utilizar *spriting*⁵⁸, mas todos os arquivos ali são utilizados com o conceito de *fingerprinting*, que baseia o nome do arquivo no conteúdo do arquivo, como forma de otimizar o carregamento e entrega do arquivo para o navegador, enviando, através de cabeçalhos HTTP, indicação se uma possível cópia do *cache* deve ser utilizada ou não.

A técnica utilizada atualmente é na forma de inserir uma *hash* do conteúdo do arquivo *antes* da extensão. Anteriormente, era utilizada a *hash* *depois* da extensão, na forma de parâmetros da URL:

```
1 antes: arquivo.css?123456
2 atual: arquivo-123456.css
```

O recurso de *fingerprinting* pode ser desabilitado configurando a opção `config.assets.digest`, também no arquivo `config/initializers/assets.rb`. Importante notar que os assets, em modo de desenvolvimento, são apenas pré compilados, dando mais rapidez no desenvolvimento da aplicação, e no ambiente de produção, após utilizar o comando (ou task):

```
1 rails assets:precompile
```

todo o processo é feito, dando mais rapidez na velocidade da aplicação.

Configurando a timezone da aplicação

Aproveitando que estamos dando uma olhada nos arquivos do diretório `initializers`, vamos aproveitar para configurar a *timezone* da nossa aplicação. Podemos ver quais são as *timezones* disponíveis rodando o seguinte comando no terminal:

```
1 $ rails time:zones:all
2
3 * UTC -11:00 *
4 American Samoa
5 International Date Line West
6 Midway Island
7 ...
8
9 * UTC -03:00 *
10 Brasilia
11 ...
```

⁵⁸<http://compass-style.org/help/tutorials/spriting/>

No meu caso, estou na *timezone* de Brasília. Seria legal seguir as convenções e inserir uma configuração para isso em um arquivo separado nos *initializers*, chamado por exemplo *config/initializers/timezone.rb*, mas por alguns motivos a configuração da *timezone* não faz efeito ali, e deve ser configurada ou no arquivo *config/application.rb* ou nos arquivos dos ambientes, no nosso caso aqui, o *config/environments/development.rb*, o qual eu acho mais indicado do que ficar alterando o *application.rb*. Vamos inserir a seguinte linha logo antes do *end* no final do arquivo:

```
1 config.time_zone = 'Brasilia'
```

Podemos abrir o *console* do Rails e verificar se foi devidamente configurada:

```
1 $ rails c
2 > Time.zone.name
3 => "Brasilia"
```

A configuração da *timezone* é importante para a utilização da aplicação, pois todas as datas e horas gravadas no *ActiveRecord* vão utilizar UTC e convertidas na *timezone* que especificarmos. Inclusive, quando precisarmos converter data e hora a partir de uma *String*, é recomendável utilizar já com a *timezone* corrente:

```
1 > Time.parse("2017-02-26 12:00:00")
2 => 2017-02-26 12:00:00 -0300
3 > Time.zone.parse("2017-02-26 12:00:00")
4 => Sun, 26 Feb 2017 12:00:00 BRT -03:00
```

Os métodos que nos auxiliam com os cálculos de datas e horas também utilizam a configuração da *timezone*. Temos alguns métodos muito úteis que nos permitem fazer coisas como:

```
1 > 2.days.ago
2 => Fri, 24 Feb 2017 12:55:01 BRT -03:00
3 > 2.hours.ago
4 => Sun, 26 Feb 2017 10:55:06 BRT -03:00
5 > 2.days.from_now
6 => Tue, 28 Feb 2017 12:55:13 BRT -03:00
7 > 2.hours.from_now
8 => Sun, 26 Feb 2017 14:55:18 BRT -03:00
9 > Time.current + 1.hour + 20.minutes + 35.seconds
10 => Sun, 26 Feb 2017 14:16:25 BRT -03:00
```

Vamos preferir fazer conversões de data para hora utilizando também as *timezones*:

```
1 > Date.today.in_time_zone
2 => Sun, 26 Feb 2017 00:00:00 BRT -03:00
3 > Date.current.in_time_zone
4 => Sun, 26 Feb 2017 00:00:00 BRT -03:00
5 > Date.today.to_time
6 => 2017-02-26 00:00:00 -0300
7 > Date.current.to_time
8 => 2017-02-26 00:00:00 -0300
```



Dica

Sempre utilizem o método `current` e não o `now`, pois o primeiro já vem com a `timezone` retornada.



Dica

Em uma API, retorne sempre a data em formato UTC para que possa ser convertida de maneira segura por quem a for consumir, e utilize o padrão ISO-8601:

```
1 > Time.current.utc.iso8601
2 => "2017-02-26T16:03:42Z"
```

Active Record

Antes de brincar um pouco no navegador, vamos abrir o console do Rails para aprender um pouco sobre o que podemos fazer com o nosso modelo. Para isso, basta digitar (se você não gostar de abreviações como o c, pode digitar rails console mesmo):

```
1 $ rails c
2 Loading development environment
3 >
```

Antes de mais nada, vamos verificar quantos registros temos de Person no banco de dados:

```
1 Person.count
2 (0.2ms) SELECT COUNT(*) FROM "people"
3 => 0
```



Viram como o ORM gerou o comando SQL necessário e mostrou no console para nós?

Criando registros

Como não temos nenhum registro, vamos criar algum, e pedir a contagem de registros novamente:

```
1 > person = Person.new
2 => #<Person id: nil, name: nil, email: nil, password: nil, born_at: nil, admi\
3 n:
4 nil, created_at: nil, updated_at: nil>
5
6 > person.name = "Eustáquio Rangel"
7 => "Eustáquio Rangel"
8
9 > person.email = "taq@bluefish.com.br"
10 => "taq@bluefish.com.br"
11
12 > person.save
13   (0.1ms) begin transaction
14   SQL (0.3ms) INSERT INTO "people" ("name", "email", "created_at",
15 "updated_at") VALUES (?, ?, ?, ?)  [["name", "Eustáquio Rangel"], ["email",
16 "taq@bluefish.com.br"], ["created_at", 2017-02-26 16:11:42 UTC],
17 ["updated_at", 2017-02-26 16:11:42 UTC]]
18   (11.3ms) commit transaction
```

```

19 => true
20
21 > Person.count
22 (0.2ms) SELECT COUNT(*) FROM "people"
23 => 1

```

Um atalho para criar um registro é com o método `create`:

```

1 > Person.create(name: "Ana Carolina", email: "carol@bluefish.com.br")
2
3   (0.1ms) begin transaction
4   SQL (0.2ms) INSERT INTO "people" ("name", "email", "created_at",
5     "updated_at") VALUES (?, ?, ?, ?)  [["name", "Ana Carolina"], ["email",
6     "carol@bluefish.com.br"], ["created_at", 2017-02-26 16:13:17 UTC],
7     ["updated_at", 2017-02-26 16:13:17 UTC]]
8   (6.1ms) commit transaction
9
10 => #<Person id: 2, name: "Ana Carolina", email:
11   "carol@bluefish.com.br", password: nil, born_at: nil, admin: nil,
12   created_at: "2017-02-26 16:13:17", updated_at: "2017-02-26 16:13:17">
13
14 > Person.count
15 (0.2ms) SELECT COUNT(*) FROM "people"
16 => 2

```



Reparam nos pontos de interrogação utilizados no comando `INSERT` acima: o Rails, a partir do 3.1, dá uma agilizada na nossa aplicação utilizando *prepared statements*. Mais detalhes [nesse link⁵⁹](#).

Se dermos uma olhada na variável que atribuímos para o objeto `Person` novo, logo antes dessa chamada, podemos verificar que o atributo `id`, que é um número auto incremental, foi preenchido automaticamente:

```

1 > person
2 => #<Person id: 1, name: "Eustáquio Rangel", email: "taq@bluefish.com.br",
3   password: nil, born_at: nil, admin: nil, created_at: "2013-07-21 20:44:17",
4   updated_at: "2013-07-21 20:44:17">

```

Consultando registros

E podemos procurar algum registro por esse `id`, utilizando o método `find`:

⁵⁹ <http://patsburghessy.net/2011/10/22/show-some-love-for-prepared-statements-in-rails-3-1>

```

1 > person = Person.find(2)
2
3 Person Load (0.3ms) SELECT "people".* FROM "people" WHERE "people"."id" = ?
4 LIMIT 1 [["id", 2]] => #<Person id: 2, name: "Ana Carolina", email:
5 "carol@bluefish.com.br", password: nil, born_at: nil, admin: nil, created_at:
6 "2013-07-21 20:48:57", updated_at: "2013-07-21 20:48:57">

```



Atenção

Se for consultado com `find` um `id` que não existe, vai ser disparada uma exceção:

```

1 Person.find(10)
2 ActiveRecord::RecordNotFound: Couldn't find Person with 'id'=10

```

Para evitarmos isso, podemos utilizar um `rescue` logo após o método `find`:

```
1 Person.find(10) rescue nil
```

Ou interceptarmos a exceção, como veremos mais para frente.

Agora podemos utilizar alguns métodos fornecidos pelo `Arel`⁶⁰, que é o gerador de *queries* utilizado por padrão no `ActiveRecord`:

```

1 > person = Person.where("id <= 2").order(:name).limit(1).first
2
3 => #<Person id: 2, name: "Ana Carolina", email: "carol@bluefish.com.br",
4 password: nil, born_at: nil, admin: nil, created_at: "2013-07-21 20:48:57",
5 updated_at: "2013-07-21 20:48:57">

```



Alerta de versões anteriores

Em versões anteriores à 4, existiam alguns *finders* disponíveis que eram gerados dinamicamente (através de `method_missing`) baseados nos atributos do modelo, como por exemplo, `find_by_name`, `find_by_email`, etc. Eles ainda não foram removidos a partir do Rails 4, mas correm o risco de ser, e mesmo se não forem, tem um problema de performance.

Podemos utilizar marcadores nas consultas, úteis para algumas conversões de tipos de dados:

⁶⁰<https://github.com/rails/arel>

```

1 > Person.where(["id <= ?", 2])
2
3 => #<ActiveRecord::Relation [<Person id: 1, name: "Eustaquio Rangel", email:
4 "taq@bluefish.com.br", password: nil, born_at: nil, admin: nil, created_at:
5 "2013-07-21 20:44:17", updated_at: "2013-07-21 20:44:17">, <Person id: 2, name: "Ana Carolina", email: "carol@bluefish.com.br", password: nil, born_at: nil, admin: true, created_at: "2013-07-21 20:48:57", updated_at: "2013-07-21 21:52:18">]
6
7
8
9

```



Tente selecionar os registros que o atributo `admin` é verdadeiro.

Também podemos utilizar marcadores nomeados, indicando onde o valor deve ser encaixado através de uma Hash:

```

1 > Person.where("id <= :id", id: 2)
2
3 Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE (id<=2)
4 => #<ActiveRecord::Relation [<Person id: 1, name: "Eustaquio Rangel", email:
5 : "taq@bluefish.com.br", password: nil,
6 born_at: nil, admin: nil, created_at: "2014-04-09 14:18:19", updated_at: "2014-04-09 14:18:19">, <Person id: 2,
7 name: "Ana Carolina", email: "carol@bluefish.com.br", password: nil, born_at:
8 : nil, admin: nil, created_at: "2014-04-09 14:18:47", updated_at: "2014-04-09 14:18:47">]
9
10

```

Selecionar de um Array:

```

1 > Person.where(id: [1, 2])
2
3 Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE "people"."id" IN
4 (1, 2) => #<ActiveRecord::Relation [<Person id: 1, name: "Eustaquio Rangel",
5 : "taq@bluefish.com.br", password: nil, born_at: nil, admin: nil, created_at: "2014-04-09 14:18:19", updated_at: "2014-04-09 14:18:19">, <Person id: 2, name: "Ana Carolina", email: "carol@bluefish.com.br", password: nil, born_at: nil, admin: nil, created_at: "2014-04-09 14:18:47", updated_at: "2014-04-09 14:18:47">]
6
7
8
9
10

```

Ou utilizar uma Range:

```
1 > Person.where(id: (1..2))
2
3   Person Load (0.4ms)  SELECT "people".* FROM "people" WHERE ("people"."id"
4     BETWEEN 1 AND 2) => #<ActiveRecord::Relation [<Person id: 1, name: "Eustaq\
5 uio
6   Rangel", email: "taq@bluefish.com.br", password: nil, born_at: nil, admin:
7   nil, created_at: "2014-04-09 14:18:19", updated_at: "2014-04-09 14:18:19">,
8   <Person id: 2, name: "Ana Carolina", email: "carol@bluefish.com.br",
9   password: nil, born_at: nil, admin: nil, created_at: "2014-04-09 14:18:47",
10  updated_at: "2014-04-09 14:18:47">]
```

Podemos verificar se um determinado registro existe, utilizando `exists?`:

```
1 > Person.exists?(1)
2 => true
3
4 > Person.exists?(3)
5 => false
```

Retornar todos os registros com `all`:

```
1 > Person.all
2
3 => [<Person id: 1, name: "Eustaquio Rangel", email: "taq@bluefish.com.br", \
4
5   password: nil, born_at: nil, admin: nil, created_at: "2013-07-21 20:44:17",
6   updated_at: "2013-07-21 20:44:17">, <Person id: 2, name: "Ana Carolina",
7   email: "carol@bluefish.com.br", password: nil, born_at: nil, admin: true,
8   created_at: "2013-07-21 20:48:57", updated_at: "2013-07-21 21:52:18">]
```

Consultando em lotes

Temos alguns métodos para auxiliar em consultas com muitos resultados retornados, sendo que o primeiro que vamos ver é o `find_each`, que utiliza o `find_in_batches` para retornar os registros em “lotes” com tamanho *default* de 1000 registros, ou de acordo com o que for especificado. Vamos testar o comportamento padrão:

```

1 > Person.find_each { |person| puts person.name }
2
3 Scoped order and limit are ignored, it's forced to be batch order and batch s\
4 ize
5 Person Load (0.1ms)  SELECT  "people".* FROM "people"    ORDER BY "people"."id"
6 ASC LIMIT 1000
7 Eustáquio Rangel de Oliveira Jr.
8 Ana Carolina

```

Podemos ver ali que foi utilizada uma cláusula `LIMIT` para limitar o resultado retornado em até 1000 registros. Podemos especificar o número de registros que queremos por lote utilizando a opção `:batch_size`:

```

1 > Person.find_each(batch_size: 50) { |person| puts person.name }
2
3 Scoped order and limit are ignored, it's forced to be batch order and batch s\
4 ize
5 Person Load (0.3ms)  SELECT  "people".* FROM "people"    ORDER BY "people"."id"
6 ASC LIMIT 50
7 Eustáquio Rangel de Oliveira Jr.
8 Ana Carolina

```

E forçar um `offset` de onde deve ser o início com `:start`:

```

1 > Person.find_each(batch_size: 50, start: 2) { |person| puts person.name }
2
3 Scoped order and limit are ignored, it's forced to be batch order and batch s\
4 ize
5 Person Load (0.3ms)  SELECT  "people".* FROM "people"  WHERE ("people"."id" >=
6 2) ORDER BY "people"."id" ASC LIMIT 50
7 Ana Carolina

```

Como dito acima, o `find_each` utiliza o `find_in_batches`, que tem a diferença de retornar o “lote” de registros:

```

1 > Person.find_in_batches do |group|
2   group.each { |person| puts person.name }
3 end
4
5 Person Load (0.3ms)  SELECT  "people".* FROM "people"    ORDER BY "people"."id"
6 " ASC LIMIT 1000
7 Eustáquio Rangel de Oliveira Jr.
8 Ana Carolina

```

As opções `:batch_size` e `:start` também funcionam com `find_in_batches`.

Se quisermos ter controle do número do lote que está sendo processado, podemos utilizar `find_-in_batches` com `with_index`:

```
1 > Person.find_in_batches(batch_size: 1).with_index do |group, batch|
2   puts "processando batch #{batch}"
3   group.each { |person| puts person.name }
4 end
5
6   Person Load (0.2ms)  SELECT  "people".* FROM "people"    ORDER BY "people"."\
7 id"
8   ASC LIMIT 1 processando batch 0
9 Eustáquio Rangel de Oliveira Jr.
10  Person Load (0.2ms)  SELECT  "people".* FROM "people"  WHERE ("people"."id"\\
11 >
12  1) ORDER BY "people"."id" ASC LIMIT 1
13
14 processando batch 1
15 Ana Carolina
16  Person Load (0.1ms)  SELECT  "people".* FROM "people"  WHERE ("people"."id"\\
17 >
18  2) ORDER BY "people"."id" ASC LIMIT 1
19 => nil
```

Atualizando registros

Podemos atualizar um registro usando o método `save`:

```
1 > person.admin = true
2 => true
3
4 > person.save
5 => true
```

Ou utilizando `update_attribute` ou `update_attributes`, para atualizar vários atributos de uma vez, enviando uma Hash:

```
1 > person.update_attribute(:admin, true)
2 => true
3
4 > person.update_attributes(admin: false)
5 => true
```

Convertendo registros

Podemos converter um objeto para JSON:

```
1 > puts person.to_json
2 {"id":1,"name":"Eustáquio Rangel de Oliveira Jr.","email":"taq@bluefish.com.b\
3 r",
4 "password":null,"born_at":null,"admin":null,"created_at":"2017-02-26T13:11:42\
5 .721-03:00",
6 "updated_at":"2017-02-26T13:21:11.599-03:00"}
```



Dica

Utilizamos `puts` para mostrar as quebras de linhas.

Até no Rails 4, existia a opção de converter para XML, mas ei, quem está utilizando esse monstro ainda, correto? Ah, é, o governo adora XML. Então, se precisarmos converter os registros ainda para XML, temos que instalar a gem `activemodel-serializers-xml` no nosso `Gemfile`, rodar um `bundle install` e indicar no modelo que ele pode ser convertido para XML incluindo o módulo:

```
1 class Person < ApplicationRecord
2   include ActiveModel::Serializers::Xml
3 end
```

Isso nos permite converter para XML sem problemas:

```
1 2.3.1 :004 > puts person.to_xml
2 <?xml version="1.0" encoding="UTF-8"?>
3 <person>
4   <id type="integer">1</id>
5   <name>Eustáquio Rangel de Oliveira Jr.</name>
6   <email>taq@bluefish.com.br</email>
7   <password nil="true"/>
8   <born-at nil="true"/>
9   <admin nil="true"/>
10  <created-at type="dateTime">2017-02-26T13:11:42-03:00</created-at>
11  <updated-at type="dateTime">2017-02-26T13:21:11-03:00</updated-at>
12 </person>
```

Apagando registros

Podemos apagar o registro utilizando `destroy`:

```

1 > person = Person.create(name: "José Mané", email: "jose.mane@gmail.com")
2
3   (0.1ms) begin transaction
4 SQL (0.3ms) INSERT INTO "people" ("name", "email", "created_at",
5     "updated_at") VALUES (?, ?, ?, ?)  [["name", "José Mané"], ["email",
6     "jose.mane@gmail.com"], ["created_at", "2017-02-26 16:36:25 UTC"],
7     ["updated_at", "2017-02-26 16:36:25 UTC"]]
8   (6.5ms) commit transaction
9
10  => #<Person id: 3, name: "José Mané", email: "jose.mane@gmail.com",
11    password: nil, born_at: nil, admin: nil, created_at: "2017-02-26
12    16:36:25", updated_at: "2017-02-26 16:36:25">
13
14 > Person.count
15 > => 3
16
17 > person.destroy
18   (0.1ms) begin transaction
19 SQL (0.3ms) DELETE FROM "people" WHERE "people"."id" = ?  [["id", 3]]
20   (10.2ms) commit transaction
21 => #<Person id: 3, name: "José Mané", email: "jose.mane@gmail.com",
22   password: nil, born_at: nil, admin: nil, created_at: "2017-02-26
23   16:36:25", updated_at: "2017-02-26 16:36:25">
24
25 > Person.count
26 => 2

```

Sandbox

Podemos trabalhar dentro do console usando um modo em que quaisquer alterações que fizermos serão descartadas no final. Para isso, devemos enviar a opção `--sandbox` junto com o comando para abrir o console:

```

1 $ rails c --sandbox
2 Loading development environment in sandbox (Rails 5.0.1)
3 Any modifications you make will be rolled back on exit

```

Ali já recebemos o aviso quaisquer alterações serão revertidas através de um `rollback` no final. Podemos experimentar utilizando um método para apagar todos os registros, `destroy_all`, e verificar o total de registros no final:

```

1 > Person.destroy_all
2
3   Person Load (1.5ms)  SELECT "people".* FROM "people"
4     (0.1ms)  SAVEPOINT active_record_1
5   SQL (2.4ms)  DELETE FROM "people" WHERE "people"."id" = ?  [["id", 1]]
6     (0.1ms)  RELEASE SAVEPOINT active_record_1
7     (0.0ms)  SAVEPOINT active_record_1
8   SQL (0.2ms)  DELETE FROM "people" WHERE "people"."id" = ?  [["id", 2]]
9     (0.1ms)  RELEASE SAVEPOINT active_record_1
10    => [#<Person id: 1, name: "Eustaquio Rangel", email: "taq@bluefish.com.br", \
11 password: nil, born_at: nil, admin: nil, created_at: "2014-04-09 14:18:19", u\
12 pdated_at: "2014-04-09 14:18:19">, #<Person id: 2, name: "Ana Carolina", emai\
13 l: "carol@bluefish.com.br", password: nil, born_at: nil, admin: nil, created_\
14 at: "2014-04-09 14:18:47", updated_at: "2014-04-09 14:18:47">]
15 2.1.1p76 :002 > Person.count
16   (0.2ms)  SELECT COUNT(*) FROM "people"
17 => 0

```

Fechando o console e abrindo novamente, vamos ver que nossas alterações foram descartadas e os registros continuam por lá:

```

1 $ rails c
2 Loading development environment (Rails 4.1.0)
3 Person.count
4   (0.1ms)  SELECT COUNT(*) FROM "people"
5 => 2

```



Tomem cuidado que em certas situações o sandbox deixa o banco de dados em estado `locked`, ou seja, travado para determinadas operações.

Calculations

Quanto tivermos alguns atributos numéricos, também podemos utilizar alguns métodos do tipo *calculations*, que são:

- `average(<atributo>)` - Calcula a média
- `sum(<atributo>)` - Calcula o total
- `minimum(<atributo>)` - Calcula o valor mínimo
- `maximum(<atributo>)` - Calcula o valor máximo
- `count` - Esse nós já vimos, calcula a quantidade de registros.

**Dica**

Também podemos passar condições para os métodos do tipo *calculation*, utilizando *conditions*, como no exemplo:

```
1 Person.where(email: "taq@bluefish.com.br").count
```

SQL Injection

Todas as formas acima que vimos em que enviamos um argumento para um marcador na consulta nos previne de ataques que utilizam uma técnica chamada **SQL Injection**⁶¹, onde são enviados códigos SQL maliciosos para as nossas consultas. Vamos imaginar que eu tenho a seguinte consulta, inserindo o argumento de forma fixa como uma String dentro dela:

```
1 > arg = "taq@bluefish.com.br"
2 => "taq@bluefish.com.br"
3
4 > Person.where("email = '#{arg}'")
5
6   Person Load (0.4ms)  SELECT "people".* FROM "people" WHERE
7     (email='taq@bluefish.com.br') ORDER BY "people"."name" ASC
8 => #< ActiveRecord::Relation [<Person id: 1, name: "Eustáquio Rangel", email:
9   "taq@bluefish.com.br", password: "9733340c840c719779f234407ee0bac26ae8904b",
10  born_at: "1971-04-10", admin: true, created_at: "2014-04-09 14:18:19",
11  updated_at: "2014-04-18 15:55:53">]
```

O que aconteceria se alguém, que não soubesse quais emails que temos cadastrados no sistema e gostaria de pegar alguns deles, enviasse um argumento como:

```
1 > arg = "user@site.com') or (1=1) --"
2 => "user@site.com') or (1=1) --"
3
4 > Person.where("email='#{arg}'")
5
6   Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE
7     (email='user@site.com') or (1=1) --' ORDER BY "people"."name" ASC
8 => #< ActiveRecord::Relation [<Person id: 1, name: "Eustáquio Rangel", email:
9   "taq@bluefish.com.br", password: "9733340c840c719779f234407ee0bac26ae8904b",
10  born_at: "1971-04-10", admin: true, created_at: "2014-04-09 14:18:19",
11  updated_at: "2014-04-18 15:55:53">, <Person id: 2, name: "Ana Carolina",
12  email: "carol@bluefish.com.br", password:
13  "9733340c840c719779f234407ee0bac26ae8904b", born_at: "1979-04-10", admin:
```

⁶¹https://pt.wikipedia.org/wiki/Inje%C3%A7%C3%A3o_de_SQL

```
14   false, created_at: "2014-04-09 14:18:47", updated_at: "2014-04-11 00:32:16"> \
15 ]>
```

Olhando a consulta SQL gerada, vemos que vamos ter problemas. Para evitar isso, utilizando marcadores:

```
1 > Person.where(["email = ?", "user@site.com') or (1=1) --"])
2
3   Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE
4     (email='user@site.com')) or (1=1) --') ORDER BY "people"."name" ASC
5 => #<ActiveRecord::Relation []>
6
7 > Person.where(["email = ?", "user@site.com) or (1=1) --"])          \
8                                         \
9
10
11    Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE
12      (email='user@site.com) or (1=1) --') ORDER BY "people"."name" ASC
13 => #<ActiveRecord::Relation []>
14
15 > Person.where(["email = :email", email: "user@site.com') or (1=1) --"])
16                                         \
17
18
19   Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE
20     (email='user@site.com')) or (1=1) --') ORDER BY "people"."name" ASC
21 => #<ActiveRecord::Relation []>
22
23 > Person.where(["email = :email", email: "user@site.com) or (1=1) --"])
24
25   Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE
26     (email='user@site.com) or (1=1) --') ORDER BY "people"."name" ASC
27 => #<ActiveRecord::Relation []>
```

Nunca dêem mole inserindo Strings direto dos comandos SQL, sempre prefiram marcadores.

Logger

Podemos customizar o log do Rails customizando o valor de `ActiveRecord::Base.logger`. Por padrão, ele é direcionado para um objeto do `ActiveSupport::Logger`, apontando para `STDOUT`:

```
1 ActiveRecord::Base.logger = ActiveSupport::Logger.new(STDOUT)
```

mas pode ser alterado para outra classe, como por exemplo o `Log4r`.

O nível de log que temos é indicado por `Rails.logger.level` e corresponde à uma das posições do Array abaixo, lembrando que é baseado em 0, verificando se o nível da mensagem é igual ou mais alto que a seleção corrente:

```
1 [:debug, :info, :warn, :error, :fatal, :unknown]
```

Podemos enviar três tipos de informações para o log, através dos seguintes métodos:

- logger.debug
- logger.info
- logger.fatal

Verificando alterações no objeto

Podemos verificar com o Active Record Dirty se houveram, e se sim, quais foram, as alterações em um objeto:

```
1 > p = Person.find(1)
2
3 Person Load (0.1ms)  SELECT "people".* FROM "people" WHERE "people"."id" = ?
4 ORDER BY name LIMIT 1  [["id", 1]] => #<Person id: 1, name: "Eustáquio Rangel\\
5 ",\\
6 email: "taq@bluefish.com.br", password:\\
7 "9733340c840c719779f234407ee0bac26ae8904b", born_at: "1971-04-06", admin: tru\\
8 e,\\
9 created_at: "2013-07-06 22:51:34", updated_at: "2013-07-13 18:43:16">>
10
11 > p.changed?
12 => false
13
14 > p.admin = false
15 => false
16
17 > p.changed?
18 => true
19
20 > p.changed_attributes
21 => {"admin"=>true}
22
23 > p.admin_was
24 => true
25
26 > p.admin_change
27 => [true, false]
28
29 p.changes
30 => {"admin"=>[true, false]}
```

Esse comportamento é interessante para registrar (e opcionalmente restaurar) os dados que foram alterados no registro. Existe uma gem especializada em fazer isso, o PaperTrail⁶², que vamos abordar na parte extra do livro.

Testes unitários

Podemos ver que, se não fossem algumas *constraints*⁶³ criadas pela migration no banco de dados, nossos registros poderiam ficar pouco inconsistentes, afinal, poderíamos criar um registro de Person sem especificar o seu nome, se não tivéssemos especificado que ele não poderia ser nulo no banco:

```

1 > person = Person.new.save
2
3 SQLite3::ConstraintException: people.name may not be NULL: INSERT
4 INTO "people" ("admin", "born_at", "created_at", "email", "name",
5 "password", "updated_at") VALUES (?, ?, ?, ?, ?, ?, ?, ?)
6 ActiveRecord::StatementInvalid: SQLite3::ConstraintException:
7 people.name may not be NULL: INSERT INTO "people" ("admin",
8 "born_at", "created_at", "email", "name", "password",
9 "updated_at") VALUES (?, ?, ?, ?, ?, ?, ?, ?)
10 ...

```

Algumas coisas para atentar aqui:

1. Sabemos que o banco de dados é uma área crítica na performance de um sistema. Quanto mais acessos, mais afetada a performance pode ficar e quanto mais acessos ruins, mais desperdício vamos gerar.
2. O tipo de exceção acima é feio pra chuchu, e precisaríamos de blocos com `begin ... rescue` para capturá-las.
3. Não é todo tipo de regra que podemos parametrizar no banco de dados, ou que é fácil de parametrizar no banco de dados sem cair no uso de recursos específicos de cada banco.

Ao invés de deixar a requisição ir direto para o banco e explodir, podemos parametrizar nossos objetos com algumas regras. Para testar essas regras, vamos testar o comportamento do nosso modelo através de **testes unitários**, que são testes relativos ao **modelo**, ao banco de dados.

Vamos especificar as seguintes condições:

1. O nome é obrigatório.
2. O email não é obrigatório, mas quando presente, tem que validar com uma expressão regular.
3. O email não pode se repetir.

⁶²https://github.com/airblade/paper_trail

⁶³[http://en.wikipedia.org/wiki/Constraint_\(database\)#Constraints](http://en.wikipedia.org/wiki/Constraint_(database)#Constraints)

4. A data de nascimento não é obrigatória, mas tem que ser de um maior de 16 anos quando estiver presente.

Para testarmos e exercitarmos essas regras, precisamos de um banco de dados de teste, que já está pronto, e de alguns dados para tentarmos inserir no banco.

Fixtures

Todos sabemos que inserir dados para testes não é uma tarefa muito agradável. Por isso o Rails utiliza o conceito de fixtures para realizar isso automaticamente para nós. As fixtures se encontram no diretório `test/fixtures` e são criadas automaticamente quando criamos um scaffold ou um modelo:

```
1 ls -lah test/fixtures/
2 total 12K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 7 taq taq 4,0K ..
5 drwxr-xr-x 7 taq taq 4,0K files
6 -rw-r--r-- 1 taq taq    0 .keep
7 -rw-r--r-- 1 taq taq  272 people.yml
```



Tem pessoas que consideram as fixtures uma coisa enviada dos quintos dos infernos e preferem utilizar o conceito de *factories* ao invés delas. Vamos aprender aqui os recursos padrões que já vem com o Rails, então vamos utilizar as fixtures, mas se você ficou curioso sobre *factories*, pode verificar a gem `FactoryGirl`⁶⁴ e um Railscast sobre isso⁶⁵.

Vamos dar uma olhada no conteúdo da fixture criada para Person, já vista aqui pluralizada no arquivo `people.yml`:

```
1 # Read about fixtures at http://api.rubyonrails.org/classes/ActiveRecord/Fixt\
2 ureSet.html
3
4 one:
5   name: MyString
6   email: MyString
7   password: MyString
8   born_at: 2017-02-26
9   admin: false
10
11 two:
12   name: MyString
```

⁶⁴https://github.com/thoughtbot/factory_girl

⁶⁵<http://railscasts.com/episodes/158-factories-not-fixtures>

```

13   email: MyString
14   password: MyString
15   born_at: 2017-02-26
16   admin: false

```

Temos ali uma Hash com Hashes dentro dela. Podemos carregar o arquivo de fixtures no console para dar uma olhada:

```

1 $ rails c
2 > h = YAML.load(File.read("test/fixtures/people.yml"))
3 => {"one"=>{"name"=>"MyString", "email"=>"MyString", "password"=>"MyString",
4 "born_at"=>Sun, 26 Feb 2017, "admin"=>false}, "two"=>{"name"=>"MyString",
5 "email"=>"MyString", "password"=>"MyString", "born_at"=>Sun, 26 Feb 2017,
6 "admin"=>false}
7
8 > h["one"]
9 => {"name"=>"MyString", "email"=>"MyString", "password"=>"MyString",
10 "born_at"=>Sun, 26 Feb 2017, "admin"=>false}

```

Agora podemos customizar um pouco esse arquivo. Vamos trocar os nomes das chaves de one e two para admin e autor, e preencher com alguns valores:

```

1 admin:
2   name: Eustáquio Rangel
3   email: taq@bluefish.com.br
4   password: secret
5   born_at: 1970-01-01
6   admin: true
7
8 autor:
9   name: Ana Carolina
10  email: carol@bluefish.com.br
11  password: secretagain
12  born_at: 1976-01-01
13  admin: false

```

As fixtures são carregadas para os testes funcionais e unitários, em três passos:

1. Os dados da tabela correspondente da fixture são removidos todos da tabela
2. Os dados das fixtures são carregados na tabela
3. Os dados das fixture são disponibilizados em uma variável para o caso de quisermos acessar diretamente.

Configurando o teste unitário

O nosso teste unitário para o modelo Person já se encontra pronto, no diretório test/models (até na versão anterior era em test/unit):

```
1 $ ls -lah test/models/person_test.rb
2 -rw-r--r-- 1 taq taq 120 test/models/person_test.rb
```

Vamos dar uma olhada no seu conteúdo, e já adequá-lo para as condições que especificamos acima, deixando-o com o seguinte conteúdo:

```
1 require 'test_helper'
2
3 class PersonTest < ActiveSupport::TestCase
4   setup do
5     @person = people(:admin)
6   end
7
8   test 'tem que ser válido sem alterações' do
9     assert @person.valid?
10  end
11
12  test 'não pode ter nome vazio' do
13    @person.name = ''
14    assert !@person.valid?
15  end
16
17  test 'não pode ter nome maior que 50 caracteres' do
18    @person.name = '*' * 51
19    assert !@person.valid?
20  end
21
22  test 'pode ter email vazio' do
23    @person.email = ''
24    assert @person.valid?
25  end
26
27  test 'não pode ter email inválido' do
28    @person.email = 'foo@bar'
29    assert !@person.valid?
30  end
31
32  test 'não pode ter email repetido' do
33    new_person = Person.new(@person.attributes)
34    assert !new_person.valid?
35  end
36
37  test 'a data de nascimento não pode ser menor que 16 anos' do
38    @person.born_at = Date.current - 15.years
39    assert !@person.valid?
40  end
```

```

41
42   test 'a data de nascimento pode ser maior que 16 anos' do
43     @person.born_at = Date.current - 17.years
44     assert @person.valid?
45   end
46 end

```

Podemos reparar que conseguimos acesso às variáveis das fixtures na linha 6, através do método `people`, que retorna o “registro” contido na chave que especificamos para ele. De quebra, ganhamos alguns métodos para trabalhar com datas.

Vamos rodar os testes unitários utilizando `rails test:models` (com o *output* mostrado aqui de forma resumida):

```

1 $ rails test:models
2 1) Failure: PersonTest#test_a_data_de_nascimento_não_pode_ser_menor_que_16_an\
3 os
4 2) Failure: PersonTest#test_não_pode_ter_email_inválido
5 3) Failure: PersonTest#test_não_pode_ter_email_repetido
6 4) Failure: PersonTest#test_não_pode_ter_nome_vazio
7 5) Failure: PersonTest#test_não_pode_ter_nome_maior_que_50_caracteres
8
9 8 tests, 8 assertions, 5 failures, 0 errors, 0 skips

```

Podemos ver que o nosso teste unitário apresentou uma série de falhas, pois o nosso modelo ainda se encontra “cru”. Agora vamos alterar o modelo para obedecer às regras que desejamos. Os modelos do sistema se encontram no diretório `app/models`, e no caso do modelo `Person`, no arquivo `person.rb`, que mostramos aqui já alterado:

```

1 class Person < ApplicationRecord
2   validates :name, presence: true, length: { maximum: 50 }
3   validates :email, allow_blank: true, allow_nil: true, uniqueness: true, for\
4   mat: { with: /\A[a-zA-Z0-9_.-]+@[a-zA-Z0-9_.-]+\.[a-zA-Z]{2,4}\z/ }
5   validates :born_at, presence: true
6   validate :age_limit
7
8   private
9
10  def age_limit
11    if self.born_at.nil? || Date.current.year - self.born_at.year < 16
12      errors.add(:born_at, 'tem que ser maior que 16 anos')
13      throw(:abort)
14    end
15  end
16 end

```

Rodando nossos testes unitários novamente:

```

1 $ rails test:models
2 Run options: --seed 57328
3
4 # Running:
5
6 .....
7
8 Finished in 0.054606s, 146.5036 runs/s, 146.5036 assertions/s.
9
10 8 runs, 8 assertions, 0 failures, 0 errors, 0 skips

```

Agora está tudo de acordo!



Dica

A partir do Rails 5, temos que obrigatoriamente utilizar `throw(:abort)` para interromper a verificação do objeto quando encontramos algum erro, do contrário, o restante das verificações irá prosseguir a cadeia de *callbacks*.



Reparam na expressão regular: ao invés de utilizarmos os caracteres de início de linha `\^` e de fim de linha `\$`, utilizamos, respectivamente, `\A` e `\z`, e isso tem uma boa razão. Se houver um caractere de quebra de linha (`\n`) no meio do texto avaliado, a expressão regular só vai verificar até na quebra de linha. Assim, é fácil passar pela expressão utilizando um caractere desses, ao contrário de quando utilizamos `\^` e `\$`, quando é avaliado mesmo do início ao final da linha. Isso permitia alguns problemas com XSS (Cross-site Scripting)⁶⁶, mas se tentarmos utilizar os primeiros caracteres hoje, vamos receber uma mensagem de erro perguntando se esquecemos desse pequeno detalhe ou se queremos mesmo uma validação de várias linhas, onde nesse caso devemos especificar obrigatoriamente a opção `multiline: true` junto da expressão regular.

Podemos testar no console:

```

1 > person = Person.new
2 => #<Person id: nil, name: nil, email: nil, password: nil, born_at: nil, admi\
3 n:
4 nil, created_at: nil, updated_at: nil>
5
6 > person.save
7 => false
8
9 > person.errors.messages
10 => { :name=> [ "can't be blank" ] }
11

```

⁶⁶http://pt.wikipedia.org/wiki/Cross-site_scripting

```
12 > person.errors.full_messages
13 => ["Name can't be blank", "Born at can't be blank", "Born at tem que ser mai\ 
14 or que 16 anos"]
15
16 > person.name = "Teste!"
17 => "Teste!"
18
19 > person.email = "foo@bar"
20 => "foo@bar"
21
22 > person.save
23 => false
24
25 > person.errors.messages
26 => {:email=>["is invalid"], :born_at=>["can't be blank", "tem que ser maior q\ 
27 ue 16 anos"]}
28
29 > person.errors.full_messages
30 => ["Email is invalid", "Born at can't be blank", "Born at tem que ser maior \
31 que 16 anos"]
32
33 > person.email = "foo@bar.com"
34 => "foo@bar.com"
35
36 > person.born_at = Date.today-10.years
37 => Sat, 21 Jul 2003
38
39 > person.save
40 => false
41
42 > person.errors.messages
43 => {:base=>["tem que ser maior que 16 anos"]}
44
45 > person.errors.full_messages
46 => ["Born at tem que ser maior que 16 anos"]
47
48 > person.born_at = Date.today-20.years
49 => Sat, 21 Jul 1993
50
51 > person.save
52 => true
53
54 > person.destroy
55 => true
```

Regras de validação

Pudemos ver acima que utilizamos tanto o método `validates`, que insere as regras de validação para os atributos, como o `validate`, que especifica que um determinado método (geralmente, um método privado) deve ser utilizado para a validação. Especificamos as seguintes condições:

- name não pode ser vazio, utilizando `presence: true`
- name não pode mais que 50 caracteres, utilizando `length: { maximum: 50 }`
- email pode ser vazio, utilizando `allow_blank: true`
- email pode ser nulo, utilizando `allow_nil: true`
- email tem que ser único se foi preenchido, utilizando `uniqueness: true`. Nesse caso, ele é combinado com as regras de vazio e nulo acima.
- email tem que combinar com a expressão regular enviada, utilizando `format {with: <regex>}`. Nesse caso, ele é combinado com as regras de vazio e nulo acima.
- A pessoa tem que ser maior que 16 anos, validando utilizando o `método age_limit`, que adiciona uma mensagem de erro na lista de erros do modelo se for menor que a idade especificada.

As validações podem ser consultadas na documentação oficial do Rails⁶⁷, e algumas das mais comuns são:

- **format** - Testa se o atributo combina com a expressão regular enviada:

```
1 validates :legacy_code, format: { with: /\A[a-zA-Z]+\z/, message: "only all\\
2 ows letters" }
```

- **inclusion** - Testa se o atributo está incluído em um determinado conjunto:

```
1 validates :size, inclusion: { in: %w(small medium large), message: "%{value}\\
2 } is not a valid size" }
```

- **length** - Testa o comprimento do atributo:

```
1 validates :name, length: { minimum: 2 }
2 validates :bio, length: { maximum: 500 }
3 validates :password, length: { in: 6..20 }
4 validates :registration_number, length: { is: 6 }
```

- **numericality** - Testa se o atributo é numérico.

```
1 validates :points, numericality: true
2 validates :games_played, numericality: { only_integer: true }
```

- **presence** - Testa se o atributo está preenchido.

⁶⁷http://edgeguides.rubyonrails.org/active_record_validations.html

```
1 validates :name, :login, :email, presence: true
```

- **uniqueness** - Testa se o atributo é único

```
1 validates :email, uniqueness: true
```

Também podemos utilizar `:if` e `:unless` no final de `validates`, passando um símbolo de um método como argumento, onde a validação só vai ser executada se atendida a condição especificada, validada pelo retorno do método. Por exemplo:

```
1 validates :born_at, presence: true, unless: :admin?
```

Para mais validações e opções enviadas para as validações, consulte a documentação oficial do Rails⁶⁸.

Validadores customizados

Como às vezes utilizamos uma validação várias vezes, podemos criar nossos validadores customizados para evitar repetir código e manter o DRY da aplicação e até mesmo para deixar ela mais limpa e clara.

No nosso modelo de pessoa em Person, utilizamos a seguinte validação:

```
1 validates :email, allow_blank: true, allow_nil: true, uniqueness: true, forma\
2 t: { with: /\A[a-zA-Z0-9_.-]+@[a-zA-Z0-9_ -]+\.\+[a-zA-Z]{2,4}\z/ }
```

Aquela expressão regular é bem compridinha e além de “poluir” o código, pode ser utilizada em alguma outra parte vindoura. Vamos extrair essa validação para um validador customizado.

Validando o registro

O primeiro jeito que podemos fazer é separar as validações em uma classe separada específica para um determinado modelo, onde *podemos executar várias validações no objeto*, como por exemplo, criando em `appValidators` (esse diretório não existe, mas pode ser criado, inclusive, com o nome que melhor atender) um arquivo chamado `person_validator.rb` com o seguinte conteúdo:

⁶⁸http://edgeguides.rubyonrails.org/active_record_validations.html

```

1 class PersonValidator < ActiveModel::Validator
2   EXPRESSION = /\A[a-zA-Z0-9_.-]+@[a-zA-Z0-9_.-]+\.\+[a-zA-Z]{2,4}\z/
3
4   def validate(record)
5     return true if record.email.blank? || record.email.match?(EXPRESSION)
6     record.errors[:email] << 'deve ser um email válido'
7   end
8 end

```

e no modelo, em `person.rb`, utilizar:

```

1 class Person < ApplicationRecord
2   validates_with PersonValidator
3   validates :email, allow_blank: true, allow_nil: true, uniqueness: true
4 ...

```

Validando um atributo

Particularmente, para efeito de reaproveitamento e granularidade, eu prefiro fazer os validadores por atributo. Nesse caso, vamos criar um novo arquivo lá em `appValidators`, chamado `email_validator.rb`, herdando de `EachValidator`, para validar somente um atributo:

```

1 class EmailValidator < ActiveModel::EachValidator
2   EXPRESSION = /\A[a-zA-Z0-9_.-]+@[a-zA-Z0-9_.-]+\.\+[a-zA-Z]{2,4}\z/
3
4   def validate_each(record, attr, value)
5     return true if value.blank? || value.match?(EXPRESSION)
6     record.errors[attr] << (options[:message] || 'deve ser um email válido')
7   end
8 end

```

e no modelo, em `person.rb`, utilizar:

```

1 class Person < ApplicationRecord
2   validates :email, allow_blank: true, allow_nil: true, uniqueness: true, ema\il: true
3 ...

```

Reparam que utilizei `email: true`, fica bem mais *clean* e também que temos no validador a opção de enviar uma mensagem, então podemos sobrescrever a mensagem padrão enviando uma customizada como:

```

1 validates :email, allow_blank: true, allow_nil: true, uniqueness: true, email\
2 : { message: 'está com um formato maluco' }

```

E até já traduzir (assunto que vamos ver logo a seguir) a mensagem. Nesse caso, prefiro fazer dessa forma:

```

1 class EmailValidator < ActiveRecord::EachValidator
2   EXPRESSION = /\A[a-zA-Z0-9_\.]+@[a-zA-Z0-9_\.]+\.[a-zA-Z]{2,4}\z/
3
4   def validate_each(record, attr, value)
5     return true if value.blank? || value.match?(EXPRESSION)
6     record.errors[attr] << message
7   end
8
9   private
10
11  def message
12    return options[:message] if options[:message].present?
13    I18n.t('activerecord.errors.messages.invalid_email', default: 'deve ser u\
14 m email válido')
15  end
16 end

```

Outra dica é que o validador é procurado pelo símbolo que é enviado. Como enviamos `email: true`, vai ser procurado o `EmailValidator`. Se enviarmos `gororoba: true`, vai ser procurado o `GororobaValidator`.

Traduzindo

O Rails permite que utilizemos arquivos de tradução para o nosso sistema, rodando em uma linguagem `default` e/ou alterando a linguagem de acordo com o necessário. Os arquivos de tradução ficam no diretório `config/locales`:

```

1 $ ls -lah config/locales/
2 total 12K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 5 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 214 en.yml

```

Para inserirmos uma tradução para outra língua, temos que gravar nesse diretório um arquivo YAML para a língua desejada. Os arquivos podem ser pegos no Github no projeto `rails-i18n`⁶⁹. No nosso caso, vamos pegar o arquivo `pt-BR.yml` e gravar no diretório `config/locales`:

⁶⁹ <https://github.com/svenfuchs/rails-i18n>

```

1 $ cd config/locales
2 $ curl https://raw.githubusercontent.com/svenfuchs/rails-i18n/master/rails/lo\
3 cale/pt-BR.yml -o pt-BR.yml
4 $ ls -lah
5 total 20K
6 drwxr-xr-x 2 taq taq .
7 drwxr-xr-x 5 taq taq ..
8 -rw-r--r-- 1 taq taq en.yml
9 -rw-r--r-- 1 taq taq pt-BR.yml

```



Dica

Se vocês não tem a ferramenta `curl` instalada, instalem. Ferramenta indispensável para desenvolvimento web hoje em dia.

Agora precisamos indicar para a nossa aplicação qual vai ser a língua utilizada como *default*. Para isso, criamos o arquivo `locale.rb` no diretório `config/initializers`, com o seguinte conteúdo:

```
1 Rails.application.config.i18n.default_locale = 'pt-BR'
```

Testando agora no console:

```

1 > person = Person.new
2 => #<Person id: nil, name: nil, email: nil, password: nil, born_at: nil, admi\
3 n:
4 nil, created_at: nil, updated_at: nil>
5
6 > person.email = "foo@bar"
7 => "foo@bar"
8
9 > person.save
10 => false
11
12 > person.errors.messages
13 => {:name=>["não pode ficar em branco"], :email=>["não é válido"], \
14 :born_at=>["não pode ficar em branco", "tem que ser maior que 16 anos"]}

```

Modelos da aplicação quase todo traduzido! Para por a “cereja no bolo” da tradução, podemos customizar o nosso arquivo de tradução com a tradução dos atributos dos modelos. No caso acima, se utilizássemos o método `full_messages` para verificar quais seriam as Strings completas retornadas nas mensagens de erros, teríamos:

```

1 > person.errors.full_messages
2 => ["Name não pode ficar em branco", "Email não é válido",
3 "Born at não pode ficar em branco", "Born at tem que ser maior que 16 anos"]

```

Mas se traduzirmos os atributos dos modelos, temos que inserir conteúdo no arquivo `pt-BR.yml` que fizemos *download*, ou, até melhor, criar um arquivo separado para indicar onde ficam nossas traduções customizadas. Dessa forma, podemos atualizar o arquivo de tradução sempre que for necessário, sem perder as customizações que fizemos.

Podemos criar o arquivo `config/locales/models.pt-BR.yml` com o seguinte conteúdo:

```

1 ---
2 pt-BR:
3   activerecord:
4     models:
5       person: "Pessoa"
6     attributes:
7       person:
8         name: "Nome"
9         email: "E-mail"
10        password: "Senha"
11        born_at: "Data de nascimento"
12        admin: "Administrador"

```



Dica

Sempre utilize espaços nos arquivos de tradução, o processador YAML para as traduções atualmente é meio ... sensível.

Agora teremos:

```

1 > person = Person.new
2 => #<Person id: nil, name: nil, email: nil, password: nil, born_at: nil, admi\
3 n:
4 nil, created_at: nil, updated_at: nil>
5
6 > person.email = "foo@bar"
7 => "foo@bar"
8
9 > person.save
10 => false
11
12 > person.errors.full_messages
13 => ["Nome não pode ficar em branco", "E-mail não é válido",
14 "Data de nascimento não pode ficar em branco",
15 "Data de nascimento tem que ser maior que 16 anos"]

```

Para trocarmos o locale em tempo de execução, podemos utilizar:

```
1 I18n.locale = <locale>
```

Por exemplo, vamos fazer *download* de um arquivo em espanhol:

```
1 $ cd config/locales/
2 $ curl https://raw.githubusercontent.com/svenfuchs/rails-i18n/master/rails/lo\
3 cale/es.yml -o es.yml
4
5 $ ls -lah
6 drwxr-xr-x 2 taq taq 4,0K .
7 drwxr-xr-x 2 taq taq 4,0K ..
8 -rw-r--r-- 5 taq taq 214 en.yml
9 -rw-r--r-- 1 taq taq 5,3K es.yml
10 -rw-r--r-- 1 taq taq 5,7K pt-BR.yml
11
12 $ cd ../..
13 $ rails c
14 Loading development environment (Rails 4.1.0)
15
16 > person = Person.new
17 => #<Person id: nil, name: nil, email: nil, password: nil, born_at: nil, admi\
18 n: nil, created_at: nil, updated_at: nil>
19
20 > person.email = "foo@bar"
21 => "foo@bar"
22
23 > person.save
24 => false
25
26 > person.errors.full_messages
27 => ["Nome não pode ficar em branco", "E-mail não é válido",
28 "Data de nascimento não pode ficar em branco",
29 "Data de nascimento tem que ser maior que 16 anos"]
30
31 > I18n.locale = "es"
32 => "es"
33
34 > person.save
35 => false
36
37 > person.errors.full_messages
38 => ["Name no puede estar en blanco",
39 "Email no es válido", "Born at no puede estar en blanco",
40 "Born at tem que ser maior que 16 años"]
```

Reparam que os atributos voltaram a ficar em Inglês, pois não customizamos o arquivo `es.yml` como fizemos no `pt-BR.yml`, e a mensagem sobre ser maior que 16 anos ficou fixa.

Para alterarmos os atributos, podemos primeiro alterar o método `age_limit` do modelo para utilizar o método `t` do `I18n`, que vai retornar a tradução do que estiver escrito na chave dos níveis que especificarmos, nesse caso, onde vamos alterar nos arquivos de tradução, `activerecord.errors.messages.older_than_16`:

```
1 def age_limit
2   if self.born_at.nil? || Date.today.year - self.born_at.year < 16
3     errors.add(:born_at, I18n.t('activerecord.errors.messages.older_than_16\
4 '))
5     throw(:abort)
6   end
7 end
```

E agora inserindo em `config/locales/errors.pt-BR.yml`:

```
1 ---
2 pt-BR:
3   activerecord:
4     errors:
5       messages:
6         older_than_16: 'deve ser maior que 16 anos'
```

E em `config/locales/models.es.yml`, aproveitando para traduzir já os atributos:

```
1 ---
2 es:
3   activerecord:
4     models:
5       person: "Persona"
6     attributes:
7       person:
8         name: "Nombre"
9         email: "E-mail"
10        password: "Contraseña"
11        born_at: "Fecha de nacimiento"
12        admin: "Administrador"
```

E agora em `config/locales/errors.es.yml`, com a mensagem de erro:

```

1  ---
2  es:
3    activerecord:
4      errors:
5        messages:
6          older_than_16: "debe ser mayor que 16 años"

```

Testando novamente:

```

1 > person = Person.new
2 => #<Person id: nil, name: nil, email: nil, password: nil, born_at: nil,
3   admin: nil, created_at: nil, updated_at: nil>
4
5 > person.email = "foo@bar"
6 => "foo@bar"
7
8 > person.save
9 => false
10
11 > person.errors.full_messages
12 => ["Nome não pode ficar em branco", "E-mail não é válido",
13   "Data de nascimento não pode ficar em branco",
14   "Data de nascimento deve ser maior que 16 anos"]
15
16 > I18n.locale = "es"
17 => "es"
18
19 > person.save
20 => false
21
22 > person.errors.full_messages
23 => ["Nombre no puede estar en blanco", "E-mail no es válido",
24   "Fecha de nacimiento no puede estar en blanco",
25   "Fecha de nacimiento debe ser mayor que 16 años"]

```

Podemos também traduzir os atributos em uma *view*, onde existem várias partes utilizando texto *hardcoded*, ou seja, fixo, utilizando o método do modelo chamado `human_attribute_name`, como em `app/views/people/show.html.erb`:

```

1  <p>
2    <b><%= Person.human_attribute_name(:name) %></b>
3    <%= @person.name %>
4  </p>

```

Vamos aproveitar que estamos trabalhando nessa *view* e contornar um problema comum em páginas que mostram emails: a coleta dos mesmos por “robozinhos” que circulam na web, como

forma de utilizar os emails para os mais diversos usos. Ao invés de apresentar o email da pessoa diretamente, vamos utilizar um helper chamado `mail_to`, que vai converter o endereço de email em um link clicável:

```

1 <p>
2   <strong><%= Person.human_attribute_name(:email) %></strong>
3   <%= mail_to @person.email %>
4 </p>
```

Isso não resolve o nosso problema, já que o código HTML só criou o link:

```

1 <p>
2   <strong>Email:</strong>
3   <a href="mailto:taq@bluefish.com.br">taq@bluefish.com.br</a>
4 </p>
```

Porém, se instalarmos a gem `actionview-encoded_mail_to` no nosso `Gemfile`, executarmos o `bundler` (não esqueçam de reiniciar o servidor após instalar!):

```

1 gem 'actionview-encoded_mail_to'
2 ...
3 Installing actionview-encoded_mail_to 1.0.4
```

e alterarmos o `mail_to` para:

```

1 <p>
2   <strong><%= Person.human_attribute_name(:email) %></strong>
3   <%= mail_to @person.email, nil, encode: 'javascript' %>
4 </p>
```

vamos ter no código HTML algo como:

```

1 <p>
2   <strong>E-mail:</strong>
3   <script id="mail_to-baq1b6a1">eval(decodeURIComponent('%76%61%72%20%73%63%
4 %72%69%70%74%20%3d%20%64%6f%63%75%6d%65%6e%74%2e%67%65%74%45%6c%65%6d%65%6e%7
5 4%42%79%49%64%28%27%6d%61%69%6c%5f%74%6f%2d%62%61%71%31%62%36%61%31%27%29%3b%7
6 76%61%72%20%61%20%3d%20%64%6f%63%75%6d%65%6e%74%2e%63%72%65%61%74%65%45%6c%65%
7 %6d%65%6e%74%28%27%61%27%29%3b%61%2e%73%65%74%41%74%74%72%69%62%75%74%65%28%2
8 7%68%72%65%66%27%2c%20%27%6d%61%69%6c%74%6f%3a%74%61%71%40%62%6c%75%65%66%69%7
9 73%68%2e%63%6f%6d%2e%62%72%27%29%3b%61%2e%61%70%70%65%6e%64%43%68%69%6c%64%28%
10 %64%6f%63%75%6d%65%6e%74%2e%63%72%65%61%74%65%54%65%78%74%4e%6f%64%65%28%27%7
11 4%61%71%40%62%6c%75%65%66%69%73%68%2e%63%6f%6d%2e%62%72%27%29%3b%73%63%72%6
12 69%70%74%2e%70%61%72%65%6e%74%4e%6f%64%65%2e%69%6e%73%65%72%74%42%65%66%6f%72%
13 %65%28%61%2c%73%63%72%69%70%74%29%3b' ))</script>
14 </p>
```

Também podemos alterar o `encode` para `hex`:

```

1 <p>
2   <strong><%= Person.human_attribute_name(:email) %></strong>
3   <%= mail_to @person.email, nil, encode: 'hex' %>
4 </p>

```

que resulta em algo como:

```

1 <p>
2   <strong>E-mail:</strong>
3   <a href="#109;ailto:%74%61%71@%62%6c%75%65%6\"
4 6%69%73%68.%63%6f%6d.%62%72">#116;aq@blue#1\%
5 02;ish.com.br</a>
6 </p>

```

Testes funcionais

Se rodarmos nossos testes funcionais, que são testes relativos aos controllers e às views, vamos ter resultado similar à:

```

1 $ rails test test/controllers
2 Run options:
3
4 # Running tests:
5
6 EEEEEEE
7
8 Finished tests in 0.248808s, 28.1341 tests/s, 0.0000 assertions/s.
9
10 1) Error:
11 test_should_create_person(PeopleControllerTest):
12 StandardError: No fixture with name 'one' found for table 'people'

```

Já podemos ver claramente a causa das falhas. Lembram-se que mudamos os nomes das chaves das fixtures de one e two para admin e autor? O Rails constrói todos os seus testes utilizando os nomes antigos, e apesar da clareza que alterar os nomes nos dá, podemos considerar deixar os testes com os nomes originais. Mas isso vai de cada um. Vamos dar uma olhada no arquivo do nosso teste funcional para Person, que se encontra no diretório test/controllers:

```
1 $ cat test/controllers/people_controller_test.rb
2 require 'test_helper'
3
4 class PeopleControllerTest < ActionDispatch::IntegrationTest
5   setup do
6     @person = people(:one)
7   end
8
9   test "should get index" do
10    get people_url
11    assert_response :success
12  end
13
14  test "should get new" do
15    get new_person_url
16    assert_response :success
17  end
18
19  test "should create person" do
20    assert_difference('Person.count') do
21      post people_url, params: { person: { admin: @person.admin, born_at:
22        @person.born_at, email: @person.email, name: @person.name, password:
23        @person.password } }
24    end
25
26    assert_redirected_to person_url(Person.last)
27  end
28
29  test "should show person" do
30    get person_url(@person)
31    assert_response :success
32  end
33
34  test "should get edit" do
35    get edit_person_url(@person)
36    assert_response :success
37  end
38
39  test "should update person" do
40    patch person_url(@person), params: { person: { admin: @person.admin,
41      born_at: @person.born_at, email: @person.email, name: @person.name,
42      password: @person.password } }
43    assert_redirected_to person_url(@person)
44  end
45
46  test "should destroy person" do
```

```
47     assert_difference('Person.count', -1) do
48       delete person_url(@person)
49     end
50     assert_redirected_to people_url
51   end
52 end
```

Se trocarmos todas as ocorrências de `people(:one)` (uma só, nesse caso) para `people(:admin)`, podemos rodar nosso teste novamente e vamos ter algo como:

```
1 $ rake test
2 Run options: --seed 47509
3
4 # Running tests:
5
6 F.....
7
8 Finished tests in 2.358057s, 2.9685 tests/s, 4.6649 assertions/s.
9
10 1) Failure:
11 PeopleControllerTest#test_should_create_person
12 [/bookstore/test/controllers/people_controller_test.rb:20]:
13 "Person.count" didn't change by 1.
14 Expected: 3
15 Actual: 2
16
17 7 tests, 8 assertions, 1 failures, 0 errors, 0 skips
```

O único teste que falhou foi o “*should create person*”, lebram-se que parametrizamos no nosso modelo que o e-mail não poderia se repetir? Podemos alterar apenas o conteúdo desse teste para:

```
1 test "should create person" do
2   assert_difference('Person.count') do
3     post people_url, params: { person: { admin: @person.admin, born_at:
4       @person.born_at, email: "functional@test.com", name: @person.name, password:
5       @person.password } }
6   end
7   assert_redirected_to person_url(Person.last)
8 end
```

Rodando novamente os testes funcionais:

```
1 $ rails test test/controllers
2 Run options: --seed 54088
3
4 # Running:
5 .....
6
7 Finished in 0.559707s, 12.5065 runs/s, 16.0798 assertions/s.
8
9 7 runs, 9 assertions, 0 failures, 0 errors, 0 skips
```

Agora que nossos testes unitários e funcionais passaram, podemos executar somente `rails test`, que vai executar nossos testes já definidos, unitários e funcionais:

```
1 $ rails test
2 Running via Spring preloader in process 26305
3 Run options: --seed 23235
4
5 # Running:
6
7 .....
8
9 Finished in 0.550002s, 27.2726 runs/s, 30.9090 assertions/s.
10
11 15 runs, 17 assertions, 0 failures, 0 errors, 0 skips
```

Dando uma olhada nos testes, podemos notar como que são executados. Assim como nos testes unitários, existe um método `setup` que é *executado sempre antes de qualquer outro teste*, criando uma instância nova de `Person` através dos dados carregados da `fixture` com a chave especificada, no caso, `:admin`:

```
1 setup do
2   @person = people(:admin)
3 end
```

No teste para verificar se a ação `index` está correta, é executado o método `get`, que envia o verbo HTTP `GET`, para a URL especificada. A partir do Rails 5, não podemos fazer mais como no Rails 4 e enviar a URL como uma `action`, pois os testes funcionais são praticamente testes de integração agora e necessitam que seja indicada uma das rotas disponíveis. As rotas podem ser consultadas com o comando `rails routes`:

```

1 $ rails routes
2   Prefix Verb URI Pattern      Controller#Action
3     people GET  /people(.:format)    people#index
4           POST /people(.:format)    people#create
5   new_person GET  /people/new(.:format)  people#new
6 edit_person GET  /people/:id/edit(.:format) people#edit
7   person GET   /people/:id(.:format)   people#show
8           PATCH /people/:id(.:format)   people#update
9           PUT   /people/:id(.:format)   people#update
10          DELETE /people/:id(.:format)  people#destroy

```

Podemos procurar as rotas de um determinado controlador utilizando a opção `-c <controlador>` e também `-g` para procurar um termo genérico em prefixos e ações. Como só temos um controlador, no primeiro caso o resultado ficaria igual se procurássemos por `people`, mas podemos já ver que as rotas são filtradas se procuramos por, por exemplo, `POST`:

```

1 $ rails routes -g POST
2 Prefix Verb URI Pattern      Controller#Action
3     POST /people(.:format)    people#create

```

Para cada rota são disponibilizadas os sufixos `_url` e `_path`, ou seja, para a primeira rota mostrada acima em `people`, temos `people_url` e `people_path`. A diferença entre elas é que `_url` provê o endereço completo, com o protocolo, o nome do servidor, a porta etc (por exemplo, `http://localhost:3000/people`) enquanto que `_path` provê somente o caminho relativo (por exemplo, `/people`). Podemos ver que os testes funcionais a preferência é por `_url`. No teste abaixo, é acionada a URL `http://localhost:3000/people` com o método `GET`:

```

1 test "should get index" do
2   get people_url
3   assert_response :success
4 end

```

Uma pequena grande diferença no Rails 5 é que ele exige que os parâmetros enviados para as rotas sejam explicitamente declarados em uma chave `params` de uma Hash. Em versões anteriores, isso não era necessário, e em algumas vezes as coisas podiam se confundir com as outras opções enviadas por ali. Mesmo que que vá dar um bom trabalho converter os testes de versões anteriores, vai compensar pela questão da organização. No exemplo abaixo podemos ver o envio de `params` com uma outra chave, `person`, que carrega todos os atributos necessários (por enquanto) para criar uma pessoa no sistema:

```
1 test "should create person" do
2   assert_difference('Person.count') do
3     post people_url, params: { person: { admin: @person.admin, born_at:
4       @person.born_at, email: "functional@test.com", name: @person.name, passwo\
5     rd:
6       @person.password } }
7   end
8 end
```

Controllers

Vamos dar uma olhada em nosso controller de Person:

```
1 class PeopleController < ApplicationController
2   before_action :set_person, only: [:show, :edit, :update, :destroy]
3
4   # GET /people
5   # GET /people.json
6   def index
7     @people = Person.all
8   end
9
10  # GET /people/1
11  # GET /people/1.json
12  def show
13  end
14
15  # GET /people/new
16  def new
17    @person = Person.new
18  end
19
20  # GET /people/1/edit
21  def edit
22  end
23
24  # POST /people
25  # POST /people.json
26  def create
27    @person = Person.new(person_params)
28
29    respond_to do |format|
30      if @person.save
31        format.html { redirect_to @person, notice: 'Person was successfully c\
32     reated.' }
```

```
33     format.json { render :show, status: :created, location: @person }
34   else
35     format.html { render :new }
36     format.json { render json: @person.errors, status: :unprocessable_ent\
37 ity }
38   end
39 end
40 end
41
42 # PATCH/PUT /people/1
43 # PATCH/PUT /people/1.json
44 def update
45   respond_to do |format|
46     if @person.update(person_params)
47       format.html { redirect_to @person, notice: 'Person was successfully u\
48 pdated.' }
49       format.json { render :show, status: :ok, location: @person }
50     else
51       format.html { render :edit }
52       format.json { render json: @person.errors, status: :unprocessable_ent\
53 ity }
54     end
55   end
56 end
57
58 # DELETE /people/1
59 # DELETE /people/1.json
60 def destroy
61   @person.destroy
62   respond_to do |format|
63     format.html { redirect_to people_url, notice: 'Person was successfully \
64 destroyed.' }
65     format.json { head :no_content }
66   end
67 end
68
69 private
70   # Use callbacks to share common setup or constraints between actions.
71   def set_person
72     @person = Person.find(params[:id])
73   end
74
75   # Never trust parameters from the scary internet, only allow the white li\
76 st through.
77   def person_params
78     params.require(:person).permit(:name, :email, :password, :born_at, :adm\
```

```
79 in)
80 end
81 end
```

Antes de mais nada, prestem atenção nos comentários: eles indicam como a URL de cada ação/método vai se parecer, levando em conta a implementação REST do controller. Desde o Rails 3.x, podemos enxugar todo esse código e utilizar os responders, que a partir do Rails 4.2, deve ser instalado como uma gem, inserindo o conteúdo no Gemfile e rodando o bundler:

```
1 ...
2 gem 'responders'
3 ...
4 $ bundle install
5 ...
6 Installing responders 2.3.0
7 ...
```

Agora sim podemos alterar o código para:

```
1 class PeopleController < ApplicationController
2   before_action :set_person, only: [:show, :edit, :update, :destroy]
3   respond_to :html
4
5   def index
6     @people = Person.all
7   end
8
9   def show
10  end
11
12  def new
13    @person = Person.new
14  end
15
16  def edit
17  end
18
19  def create
20    @person = Person.new(person_params)
21    flash[:notice] = 'Pessoa salva' if @person.save
22    respond_with @person
23  end
24
25  def update
26    flash[:notice] = 'Pessoa atualizada' if @person.update(person_params)
```

```

27     respond_with @person
28   end
29
30   def destroy
31     flash[:notice] = 'Pessoa apagada' if @person.destroy
32     respond_with @person
33   end
34
35   private
36   # Use callbacks to share common setup or constraints between actions.
37   def set_person
38     @person = Person.find(params[:id])
39   end
40
41   # Never trust parameters from the scary internet, only allow the white li\
42 st through.
43   def person_params
44     params.require(:person).permit(:name, :email, :password, :born_at, :adm\
45 in)
46   end
47 end

```

Bem melhor, não? Reparem que não foi em todos os métodos que precisamos utilizar `respond_with @person`.

Se precisarmos de suporte para mais alguns formatos como JSON e XML, podemos utilizar:

```
1 respond_to :html, :json, :xml
```

Reparam que utilizamos um `callback` (que já vem pronto no `scaffold`) chamado `before_action`, que executa uma determinada ação (o método `set_person`) para todos ou um conjunto de métodos (`:show`, `:edit`, `:update`, `:destroy`) para exercer o DRY em nosso controlador, carregando o objeto referente à pessoa em apenas um método (privado):

```

1 before_action :set_person, only: [:show, :edit, :update, :destroy]
2 ...
3 private
4   # Use callbacks to share common setup or constraints between actions.
5   def set_person
6     @person = Person.find(params[:id])
7   end

```

Dessa forma temos apenas um ponto que carrega os dados da pessoa que precisamos para o nosso controller. Vamos ver mais alguns `callbacks` utilizados nos controladores no decorrer do livro.



Dica

Reparem que esse método está logo abaixo de `private`, o que indica que é privado. Para mais informações sobre modificadores de controle de acesso de métodos, deem uma olhada no meu *e-book* de Ruby, “Conhecendo Ruby”.

Também utilizamos ali uma `Hash` chamada `flash`. Não, não é aquele software que faz animações que todo mundo adorava na década de 90 e um monte de gente (inclusive eu) era contra aqueles tipos de sites feitos inteiramente usando ele, dizíamos que ia dar meleca a longo prazo e ninguém acreditava (e deu). Pensem nessa `Hash` como uma forma de enviar mensagens entre o controlador e a `view`. O que for inserido ali vai ser consumido em algum ponto da `view`, onde podemos utilizar chaves como `:notice`, `:error`, `:info` e `:warning`.

Como podemos ter certeza que não fizemos alguma coisa de errado nesse refatoramento do controlador que fizemos acima com a `gem responders`? Simples, rodamos a suíte de testes e vemos se o resultado continua o mesmo:

```
1 $ rails test
2 Running via Spring preloader in process 14222
3 Run options: --seed 1467
4
5 # Running:
6
7 .....
8
9 Finished in 1.010068s, 14.8505 runs/s, 16.8305 assertions/s.
10
11 15 runs, 17 assertions, 0 failures, 0 errors, 0 skips
```

Callbacks nos modelos

Um *callback* é um método que chamamos em determinada situação. Em alguns jeitos de desenvolver software, eles são as chamadas *triggers* no banco de dados, disparadas quando acontece alguma coisa com um registro na tabela. Aqui vamos delegar essa responsabilidade para o código, eximindo o banco de dados dessa tarefa.

Temos vários *callbacks* que podem ser chamados, mas vamos nos ater agora à um determinado exemplo. Nós indicamos, quando fizemos nosso modelo, que o email deve ser único, mas o que acontece se digitarmos o email com maiúsculas e minúsculas? O Rails tem suporte para tratar disso em *alguns bancos* que suportam, junto com a cláusula `uniqueness`, também a `case_sensitive`, mas vamos já garantir que o email é convertido em minúsculas **antes** de ser salvo no banco. Para isso, vamos utilizar o *callback* `before_save`. Antes de implementar, vamos escrever um teste no arquivo `test/models/person_test.rb`:

```
1 test 'o email deve ser salvo em minúsculas' do
2   email = 'TAQ@BLUEFISH.COM.BR'
3   @person.email = email
4   assert @person.save
5   assert_equal email.downcase, @person.email
6 end
```

Rodando os testes de modelo, podemos ver que eles quebram:

```
1 $ rails test:models
2 Run options: --seed 44934
3
4 # Running:
5
6 ....F
7
8 Failure:
9 PersonTest#test_o_email_deve_ser_salvo_em_minúsculas
10 [/home/taq/git/conhecendo-rails/apps/bookstore/test/models/person_test.rb:51]:
11 Expected: "taq@bluefish.com.br"
12 Actual: "TAQ@BLUEFISH.COM.BR"
```

Agora vamos escrever o código para converter o email em minúsculas, no arquivo do modelo em `app/models/person.rb`, inserindo o seguinte conteúdo:

```
1 before_save :convert_email  
2  
3 def convert_email  
4   email.downcase!  
5 end
```

Rodando os testes novamente, tudo certo:

```
1 $ rails test:models  
2 Run options: --seed 8169  
3  
4 # Running:  
5  
6 .....  
7  
8 Finished in 0.060790s, 148.0509 runs/s, 164.5010 assertions/s.  
9  
10 9 runs, 10 assertions, 0 failures, 0 errors, 0 skips
```

Temos mais alguns outros *callbacks*⁷⁰ além do `before_save` que vimos acima. Aqui tem uma lista:

- `:after_initialize`
- `:after_find`
- `:after_touch`
- `:before_validation`
- `:after_validation`
- `:before_save`
- `:around_save`
- `:after_save`
- `:before_create`
- `:around_create`
- `:after_create`
- `:before_update`
- `:around_update`
- `:after_update`
- `:before_destroy`
- `:around_destroy`
- `:after_destroy`
- `:after_commit`
- `:after_rollback`

⁷⁰<http://api.rubyonrails.org/classes/ActiveRecord/Callbacks.html>

A partir do Rails 5, retornando `false` de algum método de *callback* não interrompe mais a transação de onde ele foi chamado. Por exemplo, em métodos como o `before_save` utilizados para **validação** e não **transformação** como fizemos no nosso exemplo, era só retornar `false` que a transação era interrompida. Agora precisamos explicitamente chamar o método `throw` com `:abort`. Um pequeno exemplo seria:

```
1 class Person < ApplicationRecord
2 ...
3 before_save :check_annoying_person
4 ...
5 def check_annoying_person
6   if email == 'mala@semalca.com'
7     errors.add(:base, 'malas sem alças não são permitidos.')
8     throw(:abort)
9   end
10 end
11 end
```



Dica

Podemos pensar que, para customizar o objeto no momento da sua criação, é só sobre-escrever o método `initialize` do objeto e chamar o método `super`, que vai chamar o anterior, mas nesses casos a customização é feita através do método `after_initialize`, que, se existir, vai ser chamado após a criação do objeto do modelo.

Atributos virtuais

Vamos aproveitar que estamos falando sobre *callbacks* e mencionar os atributos virtuais. Os atributos virtuais são métodos nos modelos que são utilizados para processar determinada informação, sem persistir ela no banco. Podem ser utilizados para manipular essas informações de forma a transformar alguma informação a ser persistida e podem ser utilizados diretamente no código do modelo explicitamente ou através de alguma *gem* que forneça os recursos necessários.

Vimos que temos um atributo chamado `password` no nosso modelo, mas com certeza não é uma boa idéia guardar uma senha em texto puro. Para remediar essa situação, vamos utilizar um atributo virtual no nosso modelo, criando um modo de encriptar a senha informada como texto puro, gravando já encriptada em nosso modelo.

Senhas e criptografia

O Rails já dá suporte para utilizar a criptografia utilizando BCrypt. Para utilizarmos esse recurso, temos primeiro que inserir a gem `bcrypt` no nosso `Gemfile` e rodar o `bundler`:

```

1 gem 'bcrypt'
2 ...
3 $ bundle install

```

Depois, precisamos de uma coluna na tabela de pessoas com o nome de `password_digest` e o tipo `string`. Vai ser nessa coluna que a senha criptografada vai ser armazenada, e não na coluna `password`. Nesse caso, precisamos renomear a coluna da tabela. Para isso, vamos fazer uma `migration`:

```

1 $ rails g migration RenamePasswordToPasswordDigestOnPeople
2     invoke  active_record
3         create
4             db/migrate/20170228121821_rename_password_to_password_digest_on_p\\
5 eople.rb

```

Vamos customizar o conteúdo do arquivo gerado:

```

1 class RenamePasswordToPasswordDigestOnPeople < ActiveRecord::Migration[5.0]
2   def change
3     rename_column :people, :password, :password_digest
4   end

```

E rodar a `migration`:

```

1 $ rails db:migrate
2 == 20170228121821 RenamePasswordToPasswordDigestOnPeople: migrating =====\\
3 ==
4 -- rename_column(:people, :password, :password_digest)
5 -> 0.0097s
6 == 20170228121821 RenamePasswordToPasswordDigestOnPeople: migrated (0.0097\\
7 s) ==

```



Atenção

Quando precisar alterar alguma coisa no banco de dados, **sempre** faça uma `migration` que será rodada na sequência. **Não** altere direto no banco. **Não** altere uma `migration` anterior achando que não vai dar problema mais tarde. Vão por mim, já tentei dar uma de “esperto” com isso e nunca funciona.

Bom também já alterar nas `fixtures` para o nome novo da coluna, aproveitando o fato que as `fixtures` permitem a **inserção de código para ser avaliado**, onde geramos uma senha utilizando o BCrypt:

```

1 admin:
2   name: Eustáquio Rangel
3   email: taq@bluefish.com.br
4   password_digest: <%= BCrypt::Password.create('secret') %>
5   born_at: 1970-01-01
6   admin: true
7
8 autor:
9   name: Ana Carolina
10  email: carol@bluefish.com.br
11  password_digest: <%= BCrypt::Password.create('secretagain') %>
12  born_at: 1976-01-01
13  admin: false

```

Após isso, temos que indicar que o modelo vai ter suporte a senhas seguras utilizando `has_secure_password`, como no exemplo abaixo:

```

1 class Person < ActiveRecord::Base
2 ...
3   has_secure_password
4 ....
5 end

```

A partir desse momento, temos métodos para indicar e validar a senha e a confirmação, e também para fazer a validação da senha. Como não é fornecido nenhum validador da senha, podemos indicar que ela deve estar presente quando um registro for criado, evitando que seja requisitada sempre que um registro for atualizado:

```

1 class Person < ActiveRecord::Base
2 ...
3   has_secure_password
4   validates :password, presence: { on: :create }, length: { minimum: 8, all\ 
5   low_blank: true }
6 ....
7 end

```

Atentem que não temos mais uma **coluna** chamada `password`, mas temos um *atributo virtual* chamado `password`, criado pela `gem`, que vai receber a senha em texto puro e encriptar para gravar na coluna `password_digest`.

Agora podemos alterar as `views` para incluírem ambos `password` e `password_confirmation` (esse último inserido também como um atributo virtual), não esquecendo de liberá-los nos `strong parameters` no arquivo do controlador:

```

1 def person_params
2   params.require(:person).permit(:name, :email, :password,
3   :password_confirmation, :born_at, :admin)
4 end

```



A partir do Rails 4, temos o conceito de **strong parameters** nos controllers, onde são filtrados os atributos que são recebidos pelo controller, evitando atributos indesejados. Em versões anteriores do Rails, isso era feito liberando os atributos nos modelos, onde não era realmente a função dele. No controller, podemos verificar que qualquer inclusão ou alteração do objeto vai pegar os valores de `person_params`, onde é feito o filtro, que basicamente pega tudo o que foi enviado em `params` e utiliza o método `slice` (do 'ActiveSupport') para selecionar apenas o que indicamos (o método que faz o inverso é `except`):

```

1 {a: 1, b: 2, c: 3}.slice(:a, :c)
2 => {:a=>1, :c=>3}

```



Com certeza o atributo `admin` também seria removido do `person_params` acima!

Também temos que alterar o teste funcional para incluir uma senha que atenda as condições (tamanho mínimo 8) no momento em que é criado o registro e verificar que pode estar vazia (não vai ser alterada) no momento em que o registro é atualizado e vai ser alterada se não estiver vazia:

```

1 test "should create person" do
2   assert_difference('Person.count') do
3     post people_url, params: { person: { admin: @person.admin, born_at:
4       @person.born_at, email: "functional@test.com", name: @person.name, passwo\
5     rd:
6       'testetdetamanhobom' } }
7   end
8   assert_redirected_to person_url(Person.last)
9 end
10
11 ...
12
13 test "should update person without changing password" do
14   old_digest = @person.password_digest
15   patch person_url(@person), params: { person: { admin: @person.admin, born_a\
16 t:
17   @person.born_at, email: @person.email, name: @person.name } }

```

```

18 assert_redirected_to person_url(@person)
19 @person.reload
20 assert_equal old_digest, @person.password_digest
21 end
22
23 test "should update person changing password" do
24   old_digest = @person.password_digest
25   patch person_url(@person), params: { person: { admin: @person.admin, born_a\
t:
27   @person.born_at, email: @person.email, name: @person.name, password:
28   'anotherpassword' } }
29   assert_redirected_to person_url(@person)
30   @person.reload
31   assert_not_equal old_digest, @person.password_digest
32 end

```

Na *view* o código fica assim, já alterando o arquivo pt-BR para incluir `password_confirmation`:

```

1 <div class="field">
2   <%= form.label :password %>
3   <%= form.password_field :password %>
4 </div>
5
6 <div class="field">
7   <%= form.label :password_confirmation %>
8   <%= form.password_field :password_confirmation %>
9 </div>

```

A *gem* também nos fornece um método chamado `authenticate`, que deve ser utilizado para verificar se a senha indicada é válida com a senha persistida. Reparem que é informada uma senha em texto puro e comparada com a senha persistida encriptada. Vamos fazer um teste para verificar que a pessoa é autenticada e outro para verificar que não é:

```

1 test 'deve autenticar a pessoa' do
2   assert @person.authenticate('secret')
3 end
4
5 test 'não deve autenticar a pessoa' do
6   assert !@person.authenticate('nananinanão')
7 end

```

Para verificar se uma pessoa é válida através do email e senha, podemos fazer:

```

1 person = Person.where(email: email).first
2 if person && person.authenticate(password)
3   # autenticado!
4 end

```

Isso é interessante de deixar disponível como um método no modelo. Como já temos o método `authenticate` fornecido pela `gem`, podemos criar outro, chamado, por exemplo, `auth`, que vai receber o email e senha e retornar o objeto da pessoa ou nulo se a autenticação falhar. Primeiro, testes:

```

1 test 'deve ter um método para autenticar a pessoa através de email e senha' do
2   assert_respond_to Person, :auth
3 end
4
5 test 'deve autenticar com email e senha' do
6   assert_not_nil Person.auth(@person.email, 'secret')
7 end
8
9 test 'não deve autenticar com email e senha' do
10  assert_nil Person.auth(@person.email, 'nananinanão')
11 end

```

Agora a implementação do código no modelo:

```

1 def self.auth(email, senha)
2   person = Person.where(email: email).first
3   person && person.authenticate(senha) ? person : nil
4 end

```

Aproveitando que estamos falando sobre senhas, podemos ter aquela costumeira situação onde a pessoa perde a sua senha e geramos um *token* randômico e único para a recuperação da senha. Antes do Rails 5, costumávamos utilizar o `SecureRandom` para a geração do *token*, seja utilizando o método `hex`:

```

1 > SecureRandom.hex(12)
2 => "96831c30081b8c6e7c6a3287"

```

ou o `uuid`, removendo os hífens:

```

1 > SecureRandom.uuid.gsub(/-/ , '')
2 => "f720365ee6d04b0cae32479208710548"

```

mas a partir do Rails 5, temos o método `has_secure_token`, que vai gerar um *token* novo sempre que salvarmos um registro, salvando em uma coluna chamada por `default_token` ou alguma outra indicada para ele, como no exemplo, `password_reset_token`:

```
1 class Person < ApplicationRecord
2 ...
3   has_secure_token :password_reset_token
4 ...
5 end
```

O *token* a ser gravado é o exemplo do SecureRandom acima, chamando o método `hex` com um tamanho de 12 caracteres.

Dando uma garibada na view

Aparentemente agora está tudo ok, mas temos um pequeno problema com a nossa *view*, que não inclui a faixa de anos válida para a faixa de idade necessária para fazer cadastro no sistema, que é 16 anos. Podemos corrigir em `app/views/people/_form.html.erb`, alterando a linha que contém `date_select` para:

```
1 <%= form.date_select :born_at, id: :person_born_at, start_year: 100.years.ago \
2 .year, end_year: 16.years.ago.year %>
```

Isso vai mostrar os anos a partir de 100 anos (se Charles Montgomery “Monty” Burns, o popular Sr. Burns⁷¹ for utilizar o sistema, é capaz de precisarmos ir um pouco mais para trás) até 16 atrás do ano corrente.

⁷¹http://pt.wikipedia.org/wiki/Montgomery_Burns

Editing person

Nome

Eustáquio Rangel

E-mail

taq@bluefish.com.br

Senha

teste

Data de nascimento

26 ▾ Julho ▾ 1971 ▾

Administrador

Atualizar Pessoa

[Show](#) | [Back](#)

Editando o registro da pessoa

Um conceito importante que temos em Rails, como forma de otimizar o nosso código e respeitar o DRY, é o conceito de `partials`. As `partials` nada mais são do que conteúdo que pode ser incluído em um determinado ponto do *script* corrente, sendo uma analogia os arquivos que podemos utilizar `include(_once)` ou `require(_once)` no PHP. Um ponto importante é que os arquivos de `partials` devem todos começar com um sublinhado (`_`) no começo do nome do arquivo, como `_form.html.erb` que vimos acima. Se tivéssemos dois formulários, um para criar a pessoa e um para editar, seria uma violação do DRY e mais manutenção e probabilidade de erro no sistema.

Métodos e escopos no modelo

Uma regrinha básica é que os `models` tem que ser inteligentes, e os `controllers` e `views`, burros. O que precisarmos agregar de lógica vai nos modelos. Pudemos ver que criamos acima um método para autenticar as pessoas através de email e senha, criado direto como um método de classe (estático) no modelo.

Também temos o que chamamos de escopos no modelo, que podem nos retornar uma coleção de objetos que atendem à uma determinada condição. Supondo que queremos filtrar todos as

pessoas que tenham o *flag* de administrador ativo. Reparem que esse é um método do **modelo**, assim como o auth apresentado acima, e não de uma **instância** do modelo. No teste:

```
1 test "deve ter um escopo para retornar administradores" do
2   assert_respond_to Person, :admins
3   assert_equal 1, Person.admins.size
4 end
```

No modelo:

```
1 scope :admins, -> { where(admin: true) }
```

Testando no *console*:

```
1 > reload!
2 Reloading...
3 => true
4
5 > Person.admins
6 Person Load (0.2ms)  SELECT "people".* FROM "people" WHERE "people"."admin" =
7 ?  [["admin", true]]
8 => #<ActiveRecord::Relation [#<Person id: 1, name: "Eustáquio Rangel de
9 Oliveira Jr.", email: "taq@bluefish.com.br", password_digest:
10 "$2a$10$sOOLPajS7e3JrqrB10WMKeZnWT/skuQ.m0h2SM9Rhhd...", born_at:
11 "1970-01-01", admin: true, created_at: "2017-02-28 11:36:28", updated_at:
12 "2017-02-28 12:54:10"]>
```



Dica

Em modo de desenvolvimento, podemos utilizar o método `reload!` para carregar nossas alterações.

A partir do Rails 4, **todos os nossos escopos tem que ter uma lambda associada**. Em versões anteriores, o escopo acima poderia ser escrito como

```
1 scope :admins, where(admin: true)
```

fica até mais simples, mas deixa espaço para algumas “pegadinhas”, pois o escopo é avaliado no momento da sua criação, e algo como

```
1 scope :recent, where(["created_at >= ?", 7.days.ago])
```

seria avaliado no **momento em que o escopo fosse criado** e com certeza não iria refletir os registros corretos que foram criados na última semana, ao passo que

```
1 scope :recent, -> { where(["created_at >= ?", 7.days.ago]) }
```

funcionaria corretamente pois a lambda (usando o operador `stab`, ou seja, `->`) é avaliada sempre que executada, retornando a data corrente correta menos os 7 dias. Então, nos escopos é **obrigatório** usar lambdas.



O escopo criado acima, `recent`, tem uma pegadinha. Apesar de termos configurado a `timezone` da aplicação, se executarmos o escopo no console vamos ver que `created_at` é sempre procurado com data e hora com 3 horas a mais do que a data e hora locais. Isso porque a data gravada no banco de dados é gravada na `timezone` UTC, e na consulta calculada de acordo com a `timezone` local.

Podemos trocar a `timezone` atual e verificar a diferença usando `Time.now` e `Time.zone.now`, que vão retornar, respectivamente, a data e hora local e a data e hora local de acordo com a `timezone` configurada:

```
1 Time.now
2 => 2013-07-13 20:51:32 -0300
3 Time.zone.now
4 => Sat, 13 Jul 2013 20:51:35 BRT -03:00
5 Time.zone = 'UTC'
6 => "UTC"
7 Time.now
8 => 2013-07-13 20:59:42 -0300
9 Time.zone.now
10 => Sat, 13 Jul 2013 23:59:45 UTC +00:00
```



Viram como calculamos a data utilizando `7.days.ago`? Esses são alguns dos métodos que o `ActiveSupport`⁷² nos fornece para facilitar nossas vidas. São métodos que funcionam como extensões dos objetos da linguagem Ruby e que vamos ver daqui a pouco.

Podemos também enviar argumentos para o escopo. Vamos fazer um teste:

```
1 test "deve ter um escopo para retornar usuarios por dominio" do
2   assert_respond_to Person, :by_domain
3   assert_equal 2, Person.by_domain("bluefish.com.br").size
4 end
```

E configurar o escopo no modelo:

⁷²http://guides.rubyonrails.org/active_support_core_extensions.html

```
1 scope :by_domain, ->(domain) { where("email like ?", "%@#{domain}") }
```

Testando no console:

```
1 > reload!
2 Reloading...
3 => true
4
5 > Person.by_domain("bluefish.com.br")
6 Person Load (0.4ms) SELECT "people".* FROM "people" WHERE (email like
7 '%@bluefish.com.br')
8 => #<ActiveRecord::Relation [#<Person id: 1, name: "Eustáquio Rangel de
9 Oliveira Jr.", email: "taq@bluefish.com.br", password_digest:
10 "$2a$10$soOLPajS7e3JrqrB10WMKeZnWT/skuQ.m0h2SM9Rhhd...", born_at:
11 "1970-01-01", admin: true, created_at: "2017-02-28 11:36:28", updated_at:
12 "2017-02-28 12:54:10">, #<Person id: 2, name: "Ana Carolina", email:
13 "carol@bluefish.com.br", password_digest: nil, born_at: "1976-01-01", admi\
14 n:
15 false, created_at: "2017-02-28 11:36:28", updated_at: "2017-02-28
16 11:36:28">]
```

Também podemos especificar a ordem padrão das consultas ao nosso modelo, por exemplo, em ordem alfabética.

No teste:

```
1 test "deve ter um escopo padrão para retornar os usuários em ordem alfabética\
2 " do
3   people = Person.all
4   assert people.first.name < people.last.name, "#{people.last.name} deveria
5   estar antes de #{people.first.name}"
6 end
```

No modelo:

```
1 default_scope -> { order(:name) }
```



Dica

Para evitar algum escopo definido no modelo, podemos utilizar o método `unscoped`:

```
1 Person.all.first.name
2 Person Load (0.4ms) SELECT "people".* FROM "people" ORDER BY name
3 => "Ana Carolina"
4
5 Person.unscoped.all.first.name
6 Person Load (0.3ms) SELECT "people".* FROM "people"
7 => "Eustaquio Rangel"
```



O teste funcional pode quebrar agora que está ordenado por nome. Podemos, no teste de criação do registro, trocar o nome para um lá do final do alfabeto para satisfazer o teste:

```
@person.name = "Zaratsuira"
```

Escopos também podem ser utilizados encadeados, como uma interface fluente, como no exemplo:

```
1 > Person.by_domain("bluefish.com.br").admins
2   Person Load (0.2ms)  SELECT "people".* FROM "people" WHERE (email like
3     '%@bluefish.com.br') AND "people"."admin" = ?  [["admin", true]]
4   => #< ActiveRecord::Relation [<Person id: 1, name: "Eustáquio Rangel de
5     Oliveira Jr.", email: "taq@bluefish.com.br", password_digest:
6     "$2a$10$sOOLPajS7e3JrqrB10WMKeZnWT/skuQ.m0h2SM9Rhhd...", born_at:
7     "1970-01-01", admin: true, created_at: "2017-02-28 11:36:28", updated_at:
8     "2017-02-28 12:54:10">>]
```

Pensem nos escopos como “bloquinhos de construção”. Cada um deles, se bem definido (não saia criando a escopos a torto e direito só porque é legal!) e bem testado (uma outra grande vantagem) é muito útil e evita repetição de código e provê lógica bem definida. Comparando com algumas metodologias de desenvolvimento, eles podem substituir *views* no banco de dados com todas as vantagens que temos através do código.

ActiveSupport

Como vimos acima, temos algumas extensões no Rails que complementam e facilitam vários comportamentos dos objetos padrões que temos em Ruby, tendo algumas delas inclusive sendo portadas para a linguagem. Vamos dar uma olhada em algumas das que temos disponíveis, com toda a lista, descrições e exemplos de uso na página sobre o ActiveSupport do Rails Guides⁷³.

Extensões para objetos

blank?

Os seguintes valores podem ser testados como “vazios” no Rails:

- nil e false
- Strings compostas apenas por espaços em branco
- Arrays e Hashes vazias
- Objetos que implementem o método empty? e retornam true quando o método é chamado

present?

O inverso de blank?.

presence

Verifica se determinado valor está presente.

```
1 nil.presence
2 => nil
3 a = [1, 2, 3]
4 => [1, 2, 3]
5 a[0].presence
6 => 1
7 a[10].presence
8 => nil
9 h = {um: 1, dois: 2, tres: 3}
10 => {:um=>1, :dois=>2, :tres=>3}
11 h[:um].presence
12 => 1
13 h[:dez].presence
14 => nil
```

duplicable?

Indica se um objeto pode ser duplicado. Alguns valores em Ruby são *immediate values* ou *singletons*, e não são duplicáveis.

⁷³http://guides.rubyonrails.org/active_support_core_extensions.html

```

1 "oi".duplicable?
2 => true
3 1.duplicable?
4 => false

```

deep_dup

Duplica um objeto de modo profundo, ou seja, *deep*. Se você não sabe a diferença entre duplicar de modo *raso* (*shallow*) e *profundo* (*deep*), consulte o meu *ebook* [Conhecendo Ruby](#)⁷⁴, mas podemos resumir como a duplicação de um objeto com todas as referências de outros objetos dentro dele.

```

1 array = ["oi"]
2 => ["oi"]
3 novo_array = array.deep_dup
4 => ["oi"]
5 novo_array[0].upcase!
6 => "OI"
7 array
8 => ["oi"]
9 novo_array
10 => ["OI"]

```

try

Chama um método em um objeto garantindo que, se o objeto for nulo, vai ser retornado nulo, ao invés de disparada uma exceção.

```

1 nil.upcase
2 NoMethodError: undefined method `upcase' for nil:NilClass
3 nil.try(:upcase)                                \
4                                         \
5
6 => nil

```

Se enviarmos um bloco, ele vai ser executado somente se o objeto não for nulo:

```

1 nil.try{|ref| ref.upcase}
2 => nil
3 "oi".try{|ref| ref.upcase}
4 => "OI"

```

class_eval

Avalia código no contexto da classe *singleton/eigenclass* do objeto. O exemplo abaixo usa dois blocos na mesma linha, não façam isso, foi feito somente para economizar espaço!

⁷⁴<http://leanpub.com/conhecendo-ruby>

```
1 msg = "oi"
2 => "oi"
3 new_msg = msg.dup
4 => "oi"
5 new_msg.class_eval { define_method(:hello) { puts "hello" } }
6 => :hello
7 new_msg.hello
8 hello
9 => nil
10 msg.hello
11 NoMethodError: undefined method `hello' for "oi":String
```

acts_like?(duck)

Se você quer que determinada classe indique que se comporta como uma outra, por exemplo, uma `String`, podemos criar um método na classe (o valor de retorno não importa):

```
1 def acts_like_string?
2 end
```

E testar da seguinte maneira:

```
1 classe.acts_like?(:string)
```

to_param

Todos os objetos em Rails respondem a esse método, que os representa como uma `String` para consulta, ou fragmentos de URL. Por padrão ele apenas chama o método `to_s` para fazer a conversão em `String`:

```
1 1.to_param
2 => "1"
```

Podemos criar esse método em classes do nosso sistema, que vão agir em conjunto com alguns outros métodos (e no exemplo abaixo, usando o `parameterize`) para algo como:

```

1 class Person < ActiveRecord::Base
2 ...
3   def to_param
4     "#{id}-#{name.parameterize}"
5   end
6 ...
7 end
8
9 Person.last.to_param
10 => "1-eustaquio-rangel"

```

Isso inclusive nos dá uma slug⁷⁵, uma parte da URL onde tentamos representar o recurso de maneira mais legível. Utilizando parameterize temos praticamente 3 tipos de operações sendo feitas:

1. A transliteração [^transliteration] para remover os acentos dos caracteres
2. A conversão para caracteres minúsculos
3. A troca de espaços por hífens (-)

Seria mais ou menos isso:

```

1 I18n.transliterate(Person.last.name).downcase.gsub(/\s/, '-')
2 => "eustaquio-rangel"

```

to_query

Podemos usar em conjunto com `to_param` para mapear a parte da consulta retornada para a chave associada, como nos exemplos:

```

1 person.to_query("person") \\
2
3 => "person=1-eustaquio-rangel"
4 person.to_query(person.class.to_s.downcase)
5 => "person=1-eustaquio-rangel"

```

with_options

Permite enviar através de um bloco uma série de opções em comum para vários métodos, como por exemplo:

⁷⁵[http://en.wikipedia.org/wiki/Slug_\(web_publishing\)#Slug](http://en.wikipedia.org/wiki/Slug_(web_publishing)#Slug)

```
1 with_options presence: true do |attr|
2   attr.validates :name
3   attr.validates :born_at
4 end
```

instance_values

Retorna o nome das variáveis de instância de um objeto, sem a arroba (@) na frente.

instance_variable_names

Retorna o nome das variáveis de instância de um objeto, com a arroba (@) na frente.

quietly

Executa o conteúdo do bloco de maneira silenciosa, omitindo o *output* de STDOUT e STDERR:

```
1 quietly { Person.first }
2 => #<Person id: 2, name: "Ana Carolina", email: "carol@bluefish.com.br"
3 ...
```

silence_stream

Faz a mesma coisa que `quietly`, porém podemos escolher qual o fluxo que vai ser silenciado. Inclusive, vemos que o *output* do SQL no console é enviado para STDERR:

```
1 silence_stream(STDOUT) { Person.first }
2   Person Load (0.4ms)  SELECT  "people".* FROM "people"    ORDER BY "people"."\
3 name" ASC LIMIT 1
4 => #<Person id: 2, name: "Ana Carolina", email: "carol@bluefish.com.br", pas\
5 sword:
6 ...
7 silence_stream(STDERR) { Person.first }
8 => #<Person id: 2, name: "Ana Carolina", email: "carol@bluefish.com.br", pas\
9 sword:
10 ...
```

delegate

Delega a chamada de um método para outro objeto. Vai ser útil quando tivermos associações entre nossos modelos, mas já podemos ver um exemplo como em:

```

1 class Parent
2   attr_accessor :child
3   delegate :love, to: :child
4
5   def initialize(child)
6     @child = child
7   end
8 end
9
10 class Child
11   def love
12     puts "meus pais me amam!"
13   end
14 end
15
16 parent = Parent.new(Child.new)
17 => #<Parent:0x9aed888 @child=#<Child:0x9aed8b0>>
18 parent.love
19 meus pais me amam!

```

Podemos utilizar um prefixo:

```

1 class Parent
2   delegate :love, to: :child, prefix: true
3   ...
4 end
5
6 parent.child_love
7 meus pais me amam!

```

subclasses

Retorna as subclasses do receiver:

```

1 class A; end
2 class B < A; end
3 class C < B; end
4 A.subclasses
5 => [B]
6 B.subclasses
7 => [C]
8 C.subclasses
9 => []

```

descendants

Retorna todas as classes que são herdadas do receiver. Levando em conta o exemplo acima:

```

1 A.descendants
2 => [B, C]
3 B.descendants
4 => [C]
5 C.descendants
6 => []

```

html_safe e html_safe?

Por padrão, todas as `Strings` no Rails são marcadas como não-seguras para serem inseridas diretamente dentro de código HTML e são transformadas automaticamente nas `views` em entidades HTML⁷⁶, assim evitando problemas com inserção de código malicioso. Para que possamos inserir as `Strings` com o seu conteúdo original, devemos explicitamente marcá-las como seguras utilizando o método `html_safe`, e testá-las utilizando `html_safe?`:

```

1 s = "<script type='text/javascript'>alert('te peguei!');</script>"
2 => "<script type='text/javascript'>alert('te peguei!');</script>""
3 s.html_safe?
4 => false
5 s = "<script type='text/javascript'>alert('te peguei!');</script>".html_safe
6 => "<script type='text/javascript'>alert('te peguei!');</script>""
7 s.html_safe?
8 => true

```

remove

Remove o padrão:

```

1 "Comprei 2 canetas na papelaria".remove(/\s\d+\s\w+/)
2 => "Comprei na papelaria"

```

squish

Remove espaços, quebras e retornos de linha, tabulações:

```

1 "que\n      formatação\r\n\t mais\t    maluca colocaram\taqui".squish
2 => "que formatação mais maluca colocaram aqui"

```

truncate

Retorna uma cópia “truncada” do `receiver`, onde podemos passar um separador para deixar de forma mais natural:

⁷⁶http://en.wikipedia.org/wiki/List_of_XML_and_HTML_character_entity_references

```

1 "batatinha quando nasce esparrama pelo chão".truncate(27)
2 => "batatinha quando nasce e..."
3 "batatinha quando nasce esparrama pelo chão".truncate(27, separator: ' ')
4 => "batatinha quando nasce..."

```

camelize

Retorna uma String em *camel case*, levando em conta os sublinhados (*underlines*, `_`) da String:

```

1 "person".camelize
2 => "Person"
3 "person_user".camelize
4 => "PersonUser"

```

underscore

Faz o inverso de `camelize`, convertendo uma String de *camel case* para a versão com sublinhados:

```

1 "Person".underscore
2 => "person"
3 "PersonUser".underscore
4 => "person_user"

```

dasherize

Converte uma String com sublinhados em uma versão separada por hífens (-):

```

1 "person_user".dasherize
2 => "person-user"

```

to_date, to_time, to_datetime

Retornam uma String convertida em um objeto Date, podendo em `to_time` ser especificada uma timezone (o *default* é UTC):

```

1 "01/02/1970".to_date
2 => Sun, 01 Feb 1970
3 "01/02/1970 01:02:03".to_time
4
5
6 => 1970-02-01 01:02:03 -0300
7 "01/02/1970 01:02:03".to_datetime
8 => Sun, 01 Feb 1970 01:02:03 +0000
9 "01/02/1970 01:02:03".to_time(:utc)

```

```
10
11
12 => 1970-02-01 01:02:03 UTC
13 "01/02/1970 01:02:03".to_time(:local)
14 => 1970-02-01 01:02:03 -0300
```

Medidas em bytes

```
1 1.byte
2 => 1
3 1.kilobyte
4 => 1024
5 1.megabyte
6 => 1048576
7 1.gigabyte
8 => 1073741824
9 1.terabyte
10 => 1099511627776
11 1.petabyte
12 => 1125899906842624
```

Medidas em tempo

```
1 1.second.from_now
2 => Mon, 14 Apr 2014 22:29:23 BRT -03:00
3 1.minute.from_now
4
5
6 => Mon, 14 Apr 2014 22:30:27 BRT -03:00
7 1.hour.from_now
8
9
10 => Mon, 14 Apr 2014 23:29:32 BRT -03:00
11 1.day.from_now
12
13
14 => Tue, 15 Apr 2014 22:29:38 BRT -03:00
15 1.month.from_now
16
17
18 => Wed, 14 May 2014 22:29:43 BRT -03:00
19 1.year.from_now
20
21
22 => Tue, 14 Apr 2015 22:29:47 BRT -03:00
23 1.day.ago
```

```

24 => Sun, 13 Apr 2014 22:30:40 BRT -03:00
25 1.day.from_now + 1.hour + 10.minutes + 30.seconds
26 => Tue, 15 Apr 2014 23:41:41 BRT -03:00

```

sum

Soma todos os elementos de uma coleção:

```

1 [1,2,3].sum
2 => 6
3 (0..10).sum
4 => 55

```

to_sentence

Transforma os elementos de uma coleção em uma sentença:

```

1 %w(Moe Larry Curly).to_sentence
2 => "Moe, Larry e Curly"

```

to_xml

Converte em XML:

```

1 {name: "Eustaquio", email: "taq@bluefish.com.br"}.to_xml
2 <?xml version="1.0" encoding="UTF-8"?>
3 <hash>
4   <name>Eustaquio</name>
5   <email>taq@bluefish.com.br</email>
6 </hash>

```

to_json

Converte em JSON:

```

1 {name: "Eustaquio", email: "taq@bluefish.com.br"}.to_json
2 {"name": "Eustaquio", "email": "taq@bluefish.com.br"}

```

Helpers

Aproveitando que estamos falando de alguns métodos que nos auxiliam, temos no Rails os `helpers`, que como o próprio nome diz, são “ajudantes” para várias tarefas que executamos nas `views`, simplificando-as. Temos alguns `helpers` já disponíveis como o `link_to`, que podemos ver em várias das `views` geradas pelo `scaffold`, e vários outros que criam elementos HTML.

Também podemos criar nossos `helpers` customizados, inclusive, quando utilizamos o `scaffold`, é gerado um `helper` vazio para o controlador criado, no diretório `app/helpers`, juntamente com o `application_helper.rb`, que é o `helper` compartilhado entre todos os controladores:

```

1 $ ls app/helpers/
2 total 28K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 9 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 29 application_helper.rb
6 -rw-r--r-- 1 taq taq 24 people_helper.rb

```

Qualquer método que queremos que esteja disponível nas nossas `views`, se for específico de um controlador, podemos por no `helper` correspondente, ou se for de escopo global para a aplicação, podemos por no `application_helper.rb`.

[^transliteration] <https://pt.wikipedia.org/wiki/Translitera%C3%A7%C3%A3o>

JavaScript não-obstrutivo

Podemos reparar no link criado pelo `helper link_to` utilizado para apagar um registro na `view index.html.erb` que é disparada um diálogo com uma mensagem de confirmação se queremos apagar o registro correspondente.

O código é algo como:

```

1 <%= link_to 'Apagar', person, method: :delete, data: { confirm: 'Tem certeza?' \
2 ' } %>

```

Nesse código, o Rails faz uso da técnica de **JavaScript não-obstrutivo**⁷⁷, que permite que, ao invés de entupirmos o nosso código HTML com comportamento escrito em JavaScript que iria funcionar, mas iria poluir o código de marcação HTML que temos, e iria agregar funcionalidade extra que não é a função desse código de marcação.

O Rails faz isso através de alguns *data-attributes* que são criados no código HTML, como o visto acima:

```

1 <a data-confirm="Tem certeza?" data-method="delete" href="/people/2-ana-carol\ \
2 ina" rel="nofollow">Apagar</a>

```

Nesse caso, foi criado o *data-attribute* chamado `data-confirm`, que vai ser encontrado (assim como qualquer outro que se encontre na página) e feita a vinculação de código como:

⁷⁷ http://en.wikipedia.org/wiki/Unobtrusive_JavaScript

```
1 function(event) {  
2     if(!confirm('Tem certeza?')) {  
3         event.preventDefault();  
4         return false;  
5     }  
6     return true;  
7 }
```

para quando o link for clicado, deixando nosso código de marcação HTML bem limpo e organizado.

Controladores e sessões

Agora que temos um modelo com suporte à autenticação, podemos restringir o acesso à algumas partes do nosso sistema, inclusive no próprio cadastro de pessoas, para somente quem se identificou.

Para isso, vamos construir um controlador responsável por ações de login e logout, armazenando algumas informações do usuário identificado através do recurso de sessões, que permitem manter algum **estado** em nossa aplicação web.

Vamos construir um controlador chamado sessions, que será *RESTful* (daqui a pouco vamos falar sobre isso) apenas para algumas ações, como new, create e destroy, que vão ser utilizadas para mostrar o formulário de login, criar (login) e destruir (logout) a sessão.

Primeiro vamos adicionar as rotas, no arquivo config/routes.rb:

```
1 # sessions
2 resources :sessions, only: [:new, :create, :destroy]
```

No arquivo do controlador, em app/controllers/sessions_controller.rb, vamos inserir o seguinte conteúdo:

```
1 class SessionsController < ApplicationController
2   def new
3   end
4
5   def create
6   end
7
8   def destroy
9   end
10 end
```

Agora vamos fazer testes funcionais para verificar que o comportamento esperado é alcançado. Para acessar variáveis de sessão, podemos usar a *hash session*, que pode ser acessada inclusive nos testes funcionais, porém como os testes funcionais no Rails 5 estão mais parecidos com testes de integração, não podemos mais atribuir valores à session, somente ler dela. No arquivo em test/controllers/sessions_controller_test.rb vamos inserir o seguinte conteúdo:

```
1 require 'test_helper'
2
3 class SessionsControllerTest < ActionDispatch::IntegrationTest
4   setup do
5     @person = people(:admin)
6   end
7
8   test 'deve ter um formulário de login' do
9     get new_session_url
10    assert_response :success
11    assert_select 'form[action=?]', sessions_path do
12      assert_select 'input[type=text][name=\'email\']'
13      assert_select 'input[type=password][name=\'senha\']'
14      assert_select 'input[type=submit]'
15    end
16  end
17
18  test 'não deve fazer login com senha errada' do
19    post sessions_url, params: { email: @person.email, senha: 'nananinanao' }
20    assert_nil session[:id]
21    assert_redirected_to new_session_url
22  end
23
24  test 'deve fazer login' do
25    post sessions_url, params: { email: @person.email, senha: 'secret' }
26    assert_equal @person.id, session[:id]
27    assert_equal @person.name, session[:name]
28    assert_equal @person.admin, session[:admin]
29    assert_equal "Olá, #{@person.name}!", flash[:notice]
30    assert_redirected_to people_path
31  end
32
33  test 'deve fazer logout' do
34    delete session_url(@person)
35    assert_nil session[:id]
36    assert_nil session[:name]
37    assert_nil session[:admin]
38    assert_redirected_to new_session_url
39  end
40 end
```

Podemos notar que no primeiro teste, estamos, com `assert_select`, testando os elementos HTML que se encontram na página, testando o atributo `name` deles, sendo que `name` é o mais importante para o processo de login, pois vai ser o que vai ser enviado para o servidor (`id` serve para a identificação do elemento no HTML), tendo certeza que na hora que abrirmos o navegador, vamos encontrar um formulário com todos os elementos necessários.

O assert_select funciona inclusive de maneira aninhada, no exemplo, testando os elementos que se encontram dentro dele, como pudemos ver no exemplo de form, onde são procurados os elementos dentro dele.

Os outros testes especificam que não é possível fazer login com a senha errada, que deve ser possível fazer login com a senha correta, indicando uma mensagem de boas vindas, redirecionando para a página de pessoas, guardando o id do usuário que fez o login na session, na chave :id, e o último que deve ser feito o logout, atribuindo nil ao valor de session[:id].

Agora podemos incluir código no nosso controlador:

```

1 class SessionsController < ApplicationController
2   def new
3   end
4
5   def create
6     person = Person.auth(params[:email], params[:senha])
7     if !person
8       redirect_to new_session_url
9     return
10    end
11    session[:id]      = person.id
12    session[:name]    = person.name
13    session[:admin]   = person.admin
14    flash[:notice]   = "Olá, #{person.name}!"
15    redirect_to people_path
16  end
17
18  def destroy
19    session[:id] = nil
20    redirect_to new_session_url
21  end
22 end

```

E finalmente a view, que está em app/views/sessions/new.html.erb, onde vamos criar o nosso formulário:

```

1 <h1>Autenticação</h1>
2 <p><%= flash[:notice] %></p>
3
4 <%= form_tag sessions_path do %>
5   <p>
6     <%= label_tag :email %>
7     <%= text_field_tag :email %>
8   </p>
9   <p>
10    <%= label_tag :senha %>

```

```

11   <%= password_field_tag :senha %>
12   </p>
13   <p>
14     <%= submit_tag 'Autenticar' %>
15   </p>
16 <% end %>
```

Agora podemos rodar nossos testes e verificar que está tudo ok:

```

1 $ rails test
2 Run options: --seed 59157
3
4 # Running:
5
6 .....
7
8 Finished in 0.896757s, 13.3816 runs/s, 30.1085 assertions/s.
9
10 12 runs, 27 assertions, 0 failures, 0 errors, 0 skips
```

Testes de sistema

A partir do Rails 5.1, foram introduzidos os testes de sistema. Esses tipos de testes eram feitos antes como testes de integração utilizando a gem [Capybara](#)⁷⁸, que agora já vem por padrão no `Gemfile`. Esses testes vão simular - e dependendo do *driver* do [Selenium](#)⁷⁹ especificado, utilizar - o navegador utilizando nossa aplicação, inclusive testando recursos dinâmicos gerados, por exemplo, por JavaScript.

Vamos criar nosso primeiro teste de sistema utilizando a autenticação que fizemos acima. Antes de mais nada, temos que especificar o *driver* que iremos utilizar.

Por padrão, vem configurado o [chromedriver](#)⁸⁰, mas particularmente eu prefiro utilizar o [Poltergeist](#)⁸¹, que não abre a janela do navegador (mais rápido, mas que dá para fazer com o chromedriver também) e é mais fácil de instalar.

Temos um pequeno problema que algum tempo atrás o mantenedor do `PhantomJS`, que é utilizado no `Poltergeist`, decidiu [sair do projeto](#)⁸². Como é um projeto de fonte aberta, ele pode ser mantido por outras pessoas, mas o pessoal ficou meio preocupado com as razões dele ter feito isso, podendo indicar uma complexidade suficiente no código e no tocar do projeto que possa compromete-lo. Enfim, vamos ver como configurar os dois *drivers*.

Chromedriver

Se estivermos rodando no Linux, temos que instalar o navegador `Chromium` e o `chromedriver`:

⁷⁸<https://github.com/teamcapybara/capybara>

⁷⁹<https://github.com/SeleniumHQ/selenium/wiki/Ruby-Bindings>

⁸⁰<https://github.com/SeleniumHQ/selenium/wiki/ChromeDriver>

⁸¹<https://github.com/teampoltergeist/poltergeist>

⁸²<https://github.com/teampoltergeist/poltergeist/issues/882>

```
1 $ sudo apt install chromium-browser chromedriver
```

Temos que colocar o chromedriver no PATH:

```
1 $ export PATH=$PATH:/usr/lib/chromium-browser/
```

Vamos instalar a gem capybara-screenshot para registrar as capturas de tela, inserindo no Gemfile:

```
1 group :development, :test do
2   ...
3   gem 'capybara-screenshot'
4   ...
5 end
```

e, como sempre, rodar o bundler:

```
1 $ bundle
2 ...
3 Fetching capybara-screenshot 1.0.17
4 Installing capybara-screenshot 1.0.17
5 ...
```

Agora vamos configurar o arquivo test/application_system_test_case.rb para indicar que vamos utilizar o chromedriver. Do jeito que já está o arquivo quando é criado pela aplicação, já está pronto para uso, mas vamos configurar para utilizar a opção headless e também as capturas de tela:

```
1 require 'test_helper'
2
3 class ApplicationSystemTestCase < ActionDispatch::SystemTestCase
4   Capybara.register_driver :chrome do |app|
5     Capybara::Selenium::Driver.new app, browser: :chrome,
6       options: Selenium::WebDriver::Chrome::Options.new(args: %w[headless disable-gpu])
7   end
8
9
10  Capybara::Screenshot.register_driver :chrome do |driver, path|
11    driver.save_screenshot(path)
12  end
13
14  driven_by :chrome
15 end
```

Poltergeist

Para instalá-lo, vamos inserir a seguinte linha no Gemfile e executar o bundler:

```
1 gem 'poltergeist'  
2 ...  
3 $ bundle
```

Agora vamos configurar o arquivo `test/application_system_test_case.rb` para indicar que vamos utilizar o Poltergeist:

```
1 require 'test_helper'  
2 require 'capybara/poltergeist'  
3  
4 class ApplicationSystemTestCase < ActionDispatch::SystemTestCase  
5   Capybara.register_driver :poltergeist do |app|  
6     Capybara::Poltergeist::Driver.new(app, js_errors: false)  
7   end  
8   driven_by :poltergeist  
9 end
```

Rodando os testes

Vamos criar os nossos testes de sistema no arquivo `test/system/login_test.rb` com o seguinte conteúdo:

```
1 require "application_system_test_case"  
2  
3 class UsersTest < ApplicationSystemTestCase  
4   setup do  
5     @person = people(:admin)  
6   end  
7  
8   test 'cant login' do  
9     visit new_session_url  
10    assert_selector 'h1', text: 'Autenticação'  
11    fill_in 'Email', with: @person.email  
12    fill_in 'Senha', with: 'bla'  
13    click_button 'Autenticar'  
14    assert_selector 'p', text: 'Você precisa se autenticar no sistema'  
15  end  
16  
17  test 'login' do  
18    visit new_session_url  
19    assert_selector 'h1', text: 'Autenticação'  
20    fill_in 'Email', with: @person.email  
21    fill_in 'Senha', with: 'secret'  
22    click_button 'Autenticar'  
23    assert_selector 'p#notice', text: "Olá, #{@person.name}!"
```

```
24     end  
25 end
```

Podemos rodar nossos testes de sistema utilizando `rails test:system`:

```
1 rails test:system  
2 Run options: --seed 63084  
3  
4 # Running:  
5  
6 Puma starting in single mode...  
7 * Version 3.8.2 (ruby 2.4.1-p111), codename: Sassy Salamander  
8 * Min threads: 0, max threads: 1  
9 * Environment: test  
10 * Listening on tcp://0.0.0.0:32899  
11 Use Ctrl-C to stop  
12 ..  
13  
14 Finished in 2.790193s, 0.7168 runs/s, 1.4336 assertions/s.  
15 2 runs, 4 assertions, 0 failures, 0 errors, 0 skips
```

Vale notar que os testes de sistema **não são executados com o restante da suíte de testes**, então temos que executar explicitamente como demonstrado acima.

Roteamento e REST

Roteamento

Apesar do controlador estar funcionando perfeitamente, convém deixar um modo de acessar as nossas ações de login e logout para URLs como /autenticar e /sair.

Para isso vamos novamente alterar o nosso arquivo de rotas em config/routes.rb, antes fazendo alguns testes funcionais no controlador de sessões para garantir que essas rotas estão presentes.

```
1 test 'should have a login url' do
2   assert_recognizes({ controller: 'sessions', action: 'new' }, 'autenticar')
3 end
4
5 test 'should have a logout url' do
6   assert_recognizes({ controller: 'sessions', action: 'destroy' }, { path: 'sair',
7   method: 'delete' })
8 end
```

E agora vamos alterar novamente nossas rotas, no arquivo config/routes.rb:

```
1 get "autenticar" => "sessions#new"
2 post "autenticar" => "sessions#create"
3 delete "sair"      => "sessions#destroy"
```

Rodando os testes funcionais, tudo deve estar ok.

Agora reparem o que acontece na view app/views/sessions/new.html.erb, se ao invés de utilizarmos path, utilizarmos url, ficando

```
1 <%= form_tag sessions_path do %>
```

que resulta no seguinte código HTML:

```
1 <form action="http://localhost:3000/sessions" accept-charset="UTF-8" method="\
2 post">
```

Foi inserida a URL completa do destino. Isso é útil para alguns casos, como quando queremos que o formulário seja enviado para uma localização segura, utilizando *HTTPS*, o que precisaria da URL completa. Nesse caso, poderíamos especificar:

```
1 <%= form_tag sessions_url(protocol: 'https') do %>
```

Que gera, no HTML:

```
1 <form action="https://localhost:3000/sessions" accept-charset="UTF-8" method=\n2 "post">
```

Ou para fazer redicionamentos obedecendo atentamente algumas regras do protocolo *HTTP* quando utilizando o método `redirect_to`. Como essa é uma página de autenticação, poderíamos redirecioná-la para um ambiente seguro no caso de não estar em um ambiente seguro, direto no controlador:

```
1 def new\n2   if !request.ssl?\n3     redirect_to sessions_url(protocol: 'https')\n4   end\n5\n6   ...
```

Lógico que isso não vai funcionar se você não tiver um ambiente SSL devidamente configurado.

REST

Pudemos ver que quando foi criado o recurso de login e logout, foi mencionado que o controlador seria *RESTful*, mas o que é isso e melhor, o que é REST?

REST significa *Representational State Transfer* (Transferência de Estado Representacional) e é um modelo para sistemas distribuídos, como a World Wide Web. O termo foi criado por Roy Fielding⁸³, um dos principais autores da especificação do protocolo HTTP que é utilizado por sites da internet. Os sistemas que seguem os princípios REST são frequentemente chamados de RESTful.

No REST, existem os verbos HTTP que indicam qual é o comportamento da aplicação em recursos marcados como RESTful. Os verbos são GET, POST, PUT/PATCH e DELETE, sendo que:

- GET retorna a representação de um recurso
- POST requisita a criação de um novo recurso
- PUT/PATCH requisita que o recurso seja alterado
- DELETE requisita que o recurso seja removido

Podemos ver que quando criamos um *scaffold*, automaticamente o gerador cria um recurso RESTful com o nome do controlador no arquivo de rotas automaticamente, através do método `resource`:

⁸³http://pt.wikipedia.org/wiki/Roy_Fielding

```

1   Rails.application.routes.draw do
2     resources :people
3     ...

```



A partir do Rails 4, o método PUT foi substituído por PATCH, mas continua funcionando (pelo menos por enquanto) do mesmo jeito. A RFC do PATCH pode ser encontrada [aqui⁸⁴](http://www.rfc-editor.org/info/rfc5789) e a discussão da implementação do PATCH [aqui⁸⁵](https://github.com/rails/rails/issues/348).

Com o uso de resource, também são criados algumas rotas baseadas no nome do controlador, como no caso de people, que podemos ver utilizando rails routes -c people:

```

1 $ rails routes -c people
2   Prefix Verb  URI Pattern          Controller#Action
3   people GET   /people(.:format)    people#index
4         POST  /people(.:format)    people#create
5 new_person GET   /people/new(.:format) people#new
6 edit_person GET   /people/:id/edit(.:format) people#edit
7 person GET   /people/:id(.:format)  people#show
8         PATCH /people/:id(.:format)  people#update
9         PUT   /people/:id(.:format)  people#update
10        DELETE /people/:id(.:format) people#destroy

```

E métodos correspondentes para as rotas:

- **people_path** ou **people_url** - Listar as pessoas (/people/index)
- **new_person_path** ou **new_person_url** - Criar uma pessoa nova (/people/new)
- **edit_person_path(<id>)** ou **edit_person_url(<id>)** - Editar uma pessoa, especificada pelo id (/people/<id>/edit)
- **person_path(<id>)** ou **person_url(<id>)** - Mostrar uma pessoa, especificada pelo id (/people/<id>)

Rotas extras nos resources RESTful

Coleções

E se desejarmos incluir rotas extras em nossos resources *RESTful*? No caso de rotas que operam em uma coleção, podemos utilizar collection, como no caso de quisermos uma URL listando apenas os administradores (/people/admins).

Primeiro, lógico, os testes, funcionais, para o controlador people:

⁸⁴ <http://www.rfc-editor.org/info/rfc5789>

⁸⁵ <https://github.com/rails/rails/issues/348>

```

1 test 'should have an admins route' do
2   assert_routing({ path: '/people/admins' }, { controller: 'people', action: \
3 'admins' })
4 end
5
6 test 'should list all the admins' do
7   get admins_people_url
8   assert_response :success
9   assert_select 'table' do
10    assert_select 'tbody' do
11      assert_select 'tr', 1
12    end
13  end
14 end

```

Agora a ação no controlador:

```

1 def admins
2   @admins = Person.admins
3 end

```

Na nova view app/views/people/admins.html.erb:

```

1 <h1>Administradores</h1>
2 <table>
3   <thead>
4     <tr>
5       <th>Nome</th>
6     </tr>
7   </thead>
8   <tbody>
9     <% for admin in @admins %>
10      <tr>
11        <td><%= admin.name %></td>
12        <td><%= link_to "Mostrar", admin %></td>
13        <td><%= link_to "Voltar", people_path %></td>
14      </tr>
15    <% end %>
16  </tbody>
17 </table>

```

E finalmente, no arquivo de rotas:

```

1 resources :people do
2   collection do
3     get :admins
4   end
5 end

```

Agora podemos acessar /people/admins no navegador, e ganhamos o método `admins_people_path` (e também `admins_people_url`, reparem no plural, `people`), que podemos utilizar com o método `link_to` e vários outros métodos que geram URLs.

Recurso único

Podemos também inserir uma rota, no singular, para apenas um recurso, utilizando `member`. Vamos supor que desejamos mostrar as datas de criação e alteração de um determinado registro através de uma ação chamada `changed`.

Lá vamos nós nos testes novamente:

```

1 test "should have a changed route" do
2   assert_routing({path: "/people/#{@person.id}/changed"}, {controller: "people",
3   action: "changed", id: @person.id.to_param})
4 end
5
6 test "should show info about when a person has changed" do
7   get changed_person_url(@person.id)
8   assert_response :success
9   assert_select 'p#name', text: "Nome: #{@person.name}"
10  assert_select 'p#created', text: "Criado em: #{I18n.localize(@person.created_at)}"
11  assert_select 'p#updated', text: "Alterado em: #{I18n.localize(@person.updated_at)}"
12 end
13
14 end

```



Dica

Utilizamos `I18n.localize` acima pois o método `l10n`, presente nas `views`, não é acessível pelos testes funcionais, mas ele nada mais é do que um *alias* para `localize`. Esse método, utilizado depois nas `views`, formata uma data de acordo com o padrão especificado no arquivo de traduções.

No controlador, vamos criar uma nova *action* chamada `changed`:

```

1 def changed
2 end

```

Alterar para carregar a pessoa também no `before_action`:

```
1 before_action :set_person, only: [:show, :edit, :update, :destroy, :changed]
```

E criar uma nova *view*, chamada app/views/people/changed.html.erb:

```
1 <h1>Alterações</h1>
2 <p id="name">Nome: <%= @person.name %></p>
3 <p id="created">Criado em: <%= l(@person.created_at) %></p>
4 <p id="updated">Alterado em: <%= l(@person.updated_at) %></p>
```

E no arquivo de rotas:

```
1 resources :people do
2   collection do
3     get :admins
4   end
5   member do
6     get :changed
7   end
8 end
```

Agora podemos utilizar o método changed_person_path(<person>) em nos controladores e *views*, enviando como argumento o id ou o objeto da pessoa. Rodando os testes unitários, tudo ok:

```
1 $ rails test
2 Run options: --seed 47735
3
4 # Running:
5
6 .
7
8 Finished in 0.942184s, 19.1046 runs/s, 45.6387 assertions/s.
9
10 18 runs, 43 assertions, 0 failures, 0 errors, 0 skips
```

O que acontece se indicarmos um id que não existe?

Simples, explode. A aplicação, do jeito que está, explode com uma exceção mais feia que brigar com a mãe por causa da “mistura” do almoço. Se eu requisitar, por exemplo, /people/134214/changed, vou ter algo do tipo:

```

1 ActiveRecord::RecordNotFound in PeopleController#changed
2 Couldn't find Person with 'id'=134214

```

Para evitarmos isso, podemos por uma cláusula `rescue` direto no `find` utilizado por `set_person`, onde também podemos por código para a validação se encontrou ou não o registro:

```

1 def set_person
2   @person = Person.find(params[:id]) rescue nil
3   if !@person
4     flash[:notice] = "Pessoa não encontrada"
5     redirect_to action: "index"
6     return
7   end
8 end

```

O problema é que várias partes da aplicação podem ter esse tipo de verificação, o que nos faria escrever código repetitivo e pior, esquecer de lidar com esse tipo de coisa em algum momento, o que fatalmente levaria a aplicação a explodir como aconteceu acima.

Para resolver isso de forma mais eficiente, podemos indicar que o controlador vai recuperar de uma exceção `RecordNotFound`, como a disparada nesse tipo de erro, redirecionando para um determinado método. Já que podemos utilizar esse método para vários controladores, que geralmente em uma ação `index`, vamos inserir o método no `ApplicationController`, que é compartilhado por todos os controladores. Primeiro vamos capturar a exceção:

```

1 class ApplicationController < ApplicationController
2   rescue_from ActiveRecord::RecordNotFound, with: :record_not_found
3   ...

```

E agora implementar o método:

```

1 def record_not_found
2   flash[:notice] = 'Registro não encontrado'
3   redirect_to action: 'index'
4 end
5 ...

```



Cuidado para não redirecionar para um método que busque também algum registro que não exista, senão você vai ficar em loop eterno!

Vamos aproveitar que estamos implementando isso e alterar o nosso controlador de sessões. Vamos criar um tipo de exceção customizada, chamada `NotAuthenticated`, vamos indicar uma mensagem e um método para redirecionar. Primeiro, no `ApplicationController`, a exceção, que herda de `StandardError`:

```

1 class ApplicationController < ActionController::Base
2   protect_from_forgery with: :exception
3
4   NotAuthenticated = Class.new(StandardError)
5
6   rescue_from ActiveRecord::RecordNotFound, with: :record_not_found
7   rescue_from NotAuthenticated, with: :not_authenticated
8   ...

```

Agora o método:

```

1 def not_authenticated
2   flash[:notice] = 'Você precisa se autenticar no sistema'
3   redirect_to new_session_url
4 end

```

E por fim, vamos alterar o controlador de sessões para disparar a exceção:

```

1 def create
2   person = Person.auth(params[:email], params[:senha])
3   raise NotAuthenticated if !person

```

Se rodarmos os testes funcionais agora, vamos ver que tudo continua funcionando da maneira desejada, somente que agora refatoramos o código de um jeito mais eficiente.

Utilizando um REPL para depurar erros

Vamos imaginar que alguma exceção ocorra e não utilizamos o recurso mostrado acima para capturar seu tipo específico. Em **ambiente de desenvolvimento** seria bem útil termos um jeito de ver o que aconteceu e interagir com o código, especialmente com um REPL. Para testar, vamos alterar a ação `index` do controlador `People` para retornar uma coleção com `nil` ao invés dos registros ou uma coleção vazia. Vamos alterar de:

```

1 def index
2   @people = Person.all
3   ...

```

para

```

1 def index
2   @people = nil
3   ...

```

Quando visitarmos `http://localhost:3000/people`, vamos ter uma tela como essa:

```

Showing /home/taq/git/conhecendorails/bookstore/app/views/people/index.html.erb where line #18 raised:
undefined method `each' for nil:NilClass

Extracted source (around line #18):
15   </thead>
16
17   <tbody>
18   => @people.each do |person| %>
19     <tr>
20       <td><%= person.name %></td>
21       <td><%= person.email %></td>

Rails.root: /home/taq/git/conhecendorails/bookstore

Application Trace | Framework Trace | Full Trace
app/views/people/index.html.erb:18:in `__app_views_people_index_html_erb____583521268____620107588'
app/controllers/people_controller.rb:11:in `index'

```

Tela de erro

Agora vamos inserir as gems `better_errors`⁸⁶ e `binding_of_caller`⁸⁷ no **grupo de desenvolvimento** do Gemfile e rodar o bundler. É muito importante inserir nesse grupo, a não ser que desejemos que em ambiente de produção algum maluco fique fuçando no nosso código quando ocorrer uma exceção:

```

1 group :development do
2   gem 'better_errors'
3   gem 'binding_of_caller'
4 ...
5
6 $ bundle install
7 ...
8 Installing better_errors 2.0.0
9 Installing debug_inspector 0.0.2
10 Installing binding_of_caller 0.7.2
11 ...

```

Precisamos reiniciar o servidor do Rails para que as alterações entrem em efeito. Após isso, devemos ter uma tela como essa:

⁸⁶https://github.com/charliesome/better_errors

⁸⁷https://github.com/banister/binding_of_caller

The screenshot shows a browser window with a stack trace and a REPL interface. The stack trace on the left lists:

- NoMethodError at /people
- undefined method `each' for nil:NilClass
- Application Frames: All Frames
- #<Class:0xac3e0d8>#_app_views_people_index_html_erb_1...
- app/views/people/index.html.erb, line 18
- PeopleController#index
- app/controllers/people_controller.rb, line 11

The code in app/views/people/index.html.erb (line 18) is highlighted:

```

13   <th></th>
14   </tr>
15   </thead>
16
17   <tbody>
18     <% @people.each do |person| %>
19       <tr>
20         <td><= person.name %></td>
21         <td><= person.email %></td>
22         <td><= l(person.born_at) %></td>
23         <td><= person.admin %></td>

```

The REPL interface on the right shows the result of the command >> @people:

```

>> @people
=> nil
>> |

```

A note below the REPL says: ▲ This is a live shell. Type in here.

Request info

Request parameters	{"action"=>"index", "controller"=>"people"}
Rack session	#<ActionDispatch::Request::Session:0xadeb9e4 @by=#<ActionDispatch::Session::ActiveRecordStore:0xac89674 @app=#<ActionDispatch::ParamsParser:0xac89330 @app=<Rack::Head:0xac88ad4 @app=<Rack::ConditionalGet:0xac8882c @app=<Rack::ETag:0xac880e4 @app=<ActionDispatch::Routing::RouteSet:0xa895e40>, @cache_control="max-age=0, private, must-revalidate", @no_cache_control="no-cache">>, @parsers={#<Mime::Type:0x9c0bbe8 @synonyms=["text/x-json", "application/jsonrequest"], @symbol=:json, @string="application/json">>:json}>>, @default_options={:path=>"/",

Gem better_errors

Vejam que ali eu tenho um REPL habilitado no navegador, onde digitei e recebi o resultado:

```

1 >> @people
2 => nil
3 >>

```

Também conseguimos navegar pelo lado esquerdo em “Application frames” para ver a stack do ponto atual que estamos, e no lado direito inferior temos várias informações da aplicação. Vale lembrar, **não esqueçam de retornar o código do controlador ao que era antes do teste**.

Também podemos utilizar o `web-console` que vem a partir do Rails 4.2. Para isso, é só inserir `<%= console %>` em uma `view`, como por exemplo na `show` de pessoas, que vai mostrar um `console` como esse no final da página, onde eu já pedi para mostrar o conteúdo da variável `@people` (depois que retornei `Person.all` no controlador):

```

>> @people
=> #<ActiveRecord::Relation [#<Person id: 2, name: "Ana Carolina", email: "carol@bluefish.com.br", password: nil, born_at: nil, admin: true, created_at: "2015-01-29 12:47:58", updated_at: "2015-02-20 21:42:39">, #<Person id: 1, name: "Eustáquio Rangel Jr.", email: "taq@bluefish.com.br", password: "6cdadbb275a6b2bf9e7261042c872814119ba446", born_at: "1971-06-04", admin: nil, created_at: "2015-01-29 12:47:22", updated_at: "2015-02-20 12:00:46">]
>> |

```

Web console

Namespaces

Podemos criar rotas também com namespaces, que permitem que possamos construir nossa aplicação com controladores com o mesmo nome porém separados por diretório e por nome de classe, como forma de organização.

Aqui no livro não vamos utilizar para deixar (sempre) a coisa mais fácil. Por exemplo, podemos criar um controlador com o nome `Admin::PeopleController`, que vai ser gravado em `app/controllers/admin/people_controller.rb` utilizando o seguinte comando:

```

1 $ rails g scaffold_controller admin::person
2     create  app/controllers/admin/people_controller.rb
3     invoke  erb
4     create    app/views/admin/people
5     create    app/views/admin/people/index.html.erb
6     create    app/views/admin/people/edit.html.erb
7     create    app/views/admin/people/show.html.erb
8     create    app/views/admin/people/new.html.erb
9     create    app/views/admin/people/_form.html.erb
10    ...

```

O gerador utilizado ali é novo, é o `scaffold_controller`, que vai gerar um controlador com as ações RESTful para nós, juntamente com as `views` do mesmo, mas sem levar em conta os atributos do modelo.

Utilizamos um prefixo antes do nome do controlador, `admin::`, que vai fazer com que o novo controlador se chame `Admin::PeopleController` e que seja gravado no diretório `app/controllers/admin/people_controller.rb` e suas `views` em `app/views/admin/people`.

Vemos ali uma versão resumida do resultado do gerador, mas ele não cria as rotas necessárias para acessar o controlador novo. Para isso, vamos alterar o nosso arquivo de rotas `config/routes.rb` para utilizar um namespace:

```

1 namespace :admin do
2   resources :people
3 end

```

Agora podemos acessar a URL do controlador novo, que é `http://localhost:3000/admin/people`. Um detalhe **muito** importante dentro do novo controlador é que ele vai utilizar o prefixo que utilizamos em sua criação `admin` antes dos nomes dos modelos utilizados dentro do controlador, então temos que alterar

```

1 def index
2   @admin_people = Admin::Person.all
3 end

```

para

```

1 def index
2   @admin_people = Person.all
3 end

```

que vai referenciar o modelo correto (e também todas as outras ocorrências disso no controlador) e também alterar as `views`, como no formulário de

```

1 <%= form_for(@admin_person) do |f| %>

```

que vai enviar o formulário para a `people` já existente, para utilizar o *namespace* dessa forma:

```

1 <%= form_for([:admin, @admin_person]) do |f| %>

```

que vai alterar o *path* do formulário para o o caminho correto:

```

1 <form accept-charset="UTF-8" action="/admin/people"

```

Também podemos gerar um `scaffold` inteiro utilizando um `namespace`, mas depois vai nos fazer limpar um pouco de código, já que vai ser criado outro modelo dentro do diretório `admin`, etc. Particularmente, eu prefiro fazer dessa forma, pois o `scaffold` ainda dá resultados bem mais rápidos do que fazer tudo na mão.



Dica

Podemos tratar rotas que não existem inserindo no `routes.rb`:

```

1 match "/notfound" => "pub#index", via: [:get, :post]

```

E algo como isso no controlador e ação para onde foi redirecionado:

```

1 def index
2   flash[:notice] = "Desculpe, #{$params[:notfound]} não fo\
3 i encontrado".html_safe if params[:notfound]
4   @books = Book.all
5 end

```

Não esquecendo de adicionar algo para mostrar o `notice` na `view`. Utilizamos `not found`, mas pode ser qualquer identificador na rota que seja utilizado depois no controlador.

Restringindo acesso com sessões e callbacks de controlador

Agora que já temos implementado nosso processo de *login*, podemos restringir o acesso em determinados controladores apenas para quem fez o *login* no sistema. Podemos identificar isso através da verificação da presença de conteúdo na chave `id` na `Hash session`. Se houver conteúdo, o *login* é validado, se não, disparamos a exceção customizada que criamos, `NotAuthenticated`, o que vai fazer com que o usuário seja redirecionado para a página de autenticação.

Vamos fazer um teste para verificar que um usuário não tem acesso à ação `index` do controlador `Person` (as outras ações também devem ser testadas, vamos fazer apenas uma nesse caso para ganhar tempo na didática). Em versões anteriores do Rails, podíamos simular uma sessão atribuindo valores direto na `Hash session`, porém agora não podemos mais utilizar esse recurso e temos que acionar **mesmo** a ação de autenticação, o que nos dá mais segurança.

Como vamos utilizar esse método de autenticação direto nos testes, é interessante o colocar dentro do arquivo `test_helper.rb`:

```
1 def sign_in(email, password)
2   post sessions_url, params: { email: email, senha: password }
3 end
```

Vamos inserir esse método antes de cada teste funcional atual do nosso controlador de pessoas. Assim:

```
1 ...
2 test "should get index" do
3   sign_in @person.email, 'secret'
4   get people_url
5   assert_response :success
6 end
7 ...
```

Temos que fazer isso para **todos** os testes. Existem alguns *frameworks* de testes que permitem dividir os testes em contextos onde podemos executar o código de autenticação para um conjunto de testes e não para outros, mas vamos fazer disso só na parte dos extras do livro.

Agora vamos inserir um teste para demonstrar que, sem autenticação, vamos ser direcionados para a URL de login:

```
1 test 'should redirect to login page' do
2   get people_url
3   assert_redirected_to new_session_url
4 end
```

Para fazer a verificação no nosso controlador, podemos utilizar alguns métodos que filtram o acesso ao nosso controlador. Nesse caso específico, podemos utilizar `before_action` no código do controlador das pessoas, que vai chamar o método `logged?`, que verifica se a Hash de sessão tem o valor de `id` preenchido, senão, dispara a exceção `NotAuthenticated`:

```

1 class PeopleController < ApplicationController
2   respond_to :html
3   before_action :logged?, only: [:index]
4   ...
5
6   def logged?
7     raise NotAuthenticated unless session[:id]
8 end

```



Dica

Podemos limitar as ações onde o *callback* vai ser aplicado utilizando `only`, como no exemplo, ou onde vai ser ignorado, usando `except`.

Agora tudo funciona bem, e podemos testar o comportamento no navegador, com sucesso.

Só tem um pequeno detalhe que pode ferir o DRY nesse caso: essa verificação de login pode ser utilizada em mais de um controlador. Para resolver isso, vamos mover o código de `logged?` para o controlador geral e compartilhado por todos os outros, o `ApplicationController`, que se encontra em `app/controllers/application_controller`, removendo o código do método do controlador `people`:

```

1 class ApplicationController < ActionController::Base
2   protect_from_forgery
3
4   def logged?
5     raise NotAuthenticated unless session[:id]
6   end
7 end

```

Podemos rodar nossos testes que tudo deve estar rodando de acordo.



Dica

Os *callbacks* também aceitam blocos, onde é enviada uma referência do controlador:

```

1 before_action do |controller|
2   logger.info "Recebi #{controller.request.request_method} de #{controller\"
3   .request.remote_ip}"
4 end

```



Dica

Os *callbacks* também aceitam classes, para onde é enviada uma referência do controlador para o método correspondente ao filtro sendo acionado (`before`,`after`,`around`):

```

1  class Teste
2    def self.before(controller)
3      controller.logger.info "estou em BEFORE de #{controller}"
4    end
5    def self.after(controller)
6      controller.logger.info "estou em AFTER de #{controller}"
7    end
8  end
9
10 class PeopleController < ApplicationController
11   respond_to :html, :json, :xml
12   before_action :logged?, only: [:index]
13   before_action Teste
14   after_action Teste
15   ...

```



Para sacanear, insiram essa classe no `after_action` (ela pode ser declarada no começo do próprio controlador, já que só vamos fazer um teste e logo remover), depois de inserir a *gem* Nokogiri no Gemfile e executar `bundle install`:

```

1  class Reverse
2    def self.after(controller)
3      content = controller.response.body.to_s
4      doc      = Nokogiri::HTML(controller.response.body)
5      doc.xpath("//a").each do |node|
6          element = node.to_s
7          text    = node.text
8          content.sub!(element, element.sub(text, text.reverse.capitalize))
9      end
10     controller.response.body = content
11   end
12 end

```

Manipulamos o conteúdo da resposta que vai ser retornado (através de `response.body`) para inverter o texto de todos os links que existirem. Lógico que isso pode ser aproveitado de maneira mais produtiva do que sacanear o usuário da aplicação. ;-)

Além de `before_action`, também temos na sequência de *callbacks*: `after_action`, para ser executado após uma ação (onde nesse caso, lógico, não pode inviabilizar a ação e só tem acesso aos dados enviados), e `around_action`, que é chamado e executado como um `before_action` até

encontrar uma chamada à `yield`, onde vai executar a ação, e continuar com o código dele logo após `yield`.



Os `callbacks _filter` tiveram seus nomes alterados para `_action` a partir da versão 4 do Rails.⁸⁸

Refatorando um pouco os controladores

Como forma de fazer um exercício de refatoração e aproveitamento de código, vamos ver como podemos otimizar a nossa aplicação utilizando muito pouco código nos controladores, exercitando o princípio do DRY e de os controladores tem que ser “magros”. A seguir vamos ver esse exercício, e fica a critério de cada um implementar no resto do livro (é só alterar uma linha nos controladores que o `scaffold` nos dá!).

Como vamos ter outros controladores na nossa aplicação que vão ter o seu acesso limitado através de login, para que não tenhamos que utilizar o callback `before_action` que criamos em todos, podemos criar um novo controlador, chamado por exemplo `LoggedController` em `app/controllers/logged_controller.rb`:

```
1 class LoggedController < ApplicationController
2   before_action :logged?, only: [:index]
3 end
```

E agora indicar que o controlador `PeopleController` não herda mais de `ApplicationController`, e sim do novo `LoggedController`:

```
1 class PeopleController < LoggedController
2   before_action :set_person, only: [:show, :edit, :update, :destroy, :changed]
3   respond_to :html
4
5   def index
6     ...
```

Se rodarmos os testes, vamos ver que o comportamento do sistema foi mantido, e agora mais otimizado ainda. Nos outros controladores que precisarmos de restrição de acesso, é só indicar que herdam de `LoggedController`.

Fazendo um controlador mágico

Aproveitando que também os outros controladores vão todos utilizar os `responders` e serem RESTful, vamos fazer um exercício aqui e criar um novo controlador chamado `MagicController`:

⁸⁸<https://github.com/rails/rails/commit/9d62e04838f01f5589fa50b0baa480d60c815e2c>

```
1 class MagicController < LoggedController
2   before_action :logged?, only: [:index]
3   before_action :set_references
4   before_action :set_collection, only: [:index]
5   before_action :set_resource, only: [:show, :edit, :update, :destroy, :chang\
6 ed]
7   respond_to :html
8
9   def index
10    respond_with instance_variable_get("@#{@plural_ref}")
11 end
12
13  def show
14 end
15
16  def new
17    instance_variable_set("@{@singular_ref}", Object.const_get(@singular_ref\
18 .camelize).new)
19 end
20
21  def create
22    resource = instance_variable_set("@{@singular_ref}", Object.const_get(@s\
23 ingular_ref.camelize).new(send(@params)))
24    flash[:notice] = "Registro de #{@name_ref.downcase} salvo" if resource.sa\
25 ve
26    respond_with resource
27 end
28
29  def update
30    resource = instance_variable_get("@{@singular_ref}")
31    flash[:notice] = "Registro de #{@name_ref.downcase} atualizado" if resour\
32 ce.update(send(@params))
33    respond_with resource
34 end
35
36  def edit
37 end
38
39  def destroy
40    resource = instance_variable_get("@{@singular_ref}")
41    flash[:notice] = "Registro de #{@name_ref.downcase} apagado" if resource.\\
42 destroy
43    respond_with resource
44 end
45
46 private
```

```

47
48  def set_references
49    @plural_ref   = /([a-z]+)(_)?(controller)?/.match(controller_name)[1]
50    @singular_ref = @plural_ref.singularize
51    @params        = "#{@singular_ref}_params"
52    @name_ref      = I18n.translate("activerecord.models.#{@singular_ref}")
53  end
54
55  def set_collection
56    instance_variable_set("@#{@plural_ref}", Object.const_get(@singular_ref.c\
amelize).all)
57  end
58
59
60  def set_resource
61    instance_variable_set("@#{@singular_ref}", Object.const_get(@singular_ref)\.
camelize).find(params[:id]))
62  end
63
64 end

```



Vejam que por causa da liberdade criativa do exemplo na `before_action` eu inseri o método `changed`, que só faz sentido no controlador de pessoas.

Indicando que um controlador herda dele, transforma o controlador de pessoas em algo como:

```

1  class PeopleController < MagicController
2    def admins
3      @admins = Person.admins
4    end
5
6    def changed
7    end
8
9    private
10
11   def person_params
12     params.require(:person).permit(:name, :email, :password, :password_confirmatio\
mation, :born_at, :admin)
13   end
14
15 end

```

Uau.

Se você não entendeu o que algumas partes do código faz, especialmente aqueles métodos `instance_variable_*`, dê uma olhada no meu [ebook de Ruby, “Conhecendo Ruby”⁸⁹](#).

⁸⁹ <http://leanpub.com/conhecendo-ruby>

Como vamos trabalhar no resto do livro com alguns outros scaffolds e como eu sempre digo, vamos utilizar os recursos padrões do Rails e manter os controladores do jeito que vem “de fábrica”, para não complicar para quem está chegando agora.

Dá dó não utilizar o que fizemos acima, mas vai de cada um.

Views e layouts

Diferente de algumas outras linguagens/*frameworks*, onde inserimos alguns arquivos parecidos com partials no arquivo corrente, em Rails utilizamos um conceito de *layouts*, onde o conteúdo dinâmico das nossas *views* é inserido.

Podemos ver o exemplo padrão de *layout* em app/views/layouts/application.html.erb:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Bookstore</title>
5          <%= csrf_meta_tags %>
6          <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-\>
7          track': 'reload' %>
8          <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reloa\>
9          d' %>
10     </head>
11     <body>
12         <%= yield %>
13     </body>
14 </html>
```

Podemos ver ali o método `yield`, que vai inserir o conteúdo dinâmico processado de nossas *views* no ponto em que é especificado, ou seja, todo o resultado da *view* corrente é inserido naquele ponto.

Usando layouts diferentes

Vamos especificar *layouts* diferentes para condições diferentes. Como acabamos de restringir o acesso do nosso controlador de pessoas para apenas os usuários que tiverem feito o login, criamos um ambiente restrito que podemos chamar de “interface de administração”, onde vamos inserir menus, títulos diferentes, etc.

Vamos customizar esse *layout* (app/views/layouts/application.html.erb) dessa maneira:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>Bookstore</title>
5      <%= stylesheet_link_tag     "application", media: "all", "data-turbolinks-tr\
6 ack" => true %>
7      <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
8      <%= csrf_meta_tags %>
9  </head>
10 <body>
11     <header>
12         <h1>Interface de administração</h1>
13     </header>
14     <nav>
15         <ul>
16             <li><%= link_to "Pessoas", people_path %></li>
17         </ul>
18     </nav>
19     <section id="main">
20         <%= yield %>
21     </section>
22     <footer>
23         Desenvolvido no ebook "Conhecendo Rails"
24     </footer>
25 </body>
26 </html>
```

Ficou legal, mas deêm uma olhada na página de autenticação:

Interface de administração

- Pessoas

Autenticação

Email

taq@bluefish.com.br

Senha

.....

Autenticar

Desenvolvido no ebook "Conhecendo Rails"

Isso está meio estranho

Ainda não estamos na interface administrativa e nem queremos que um usuário que ainda não efetuou o login veja o menu da interface de administração. A solução para isso é criar um arquivo de *layout* novo, chamado de pub.html.erb:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Bookstore</title>
5     <%= csrf_meta_tags %>
6     <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-\
7 track': 'reload' %>
8     <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reloa\
9 d' %>
10    </head>
11    <body>
12      <header>
```

```

13      <h1>Bookstore - A maior livraria virtual (depois da Amazon)</h1>
14  </header>
15  <nav>
16    <ul>
17      <li><%= link_to 'Inicial', '/' %></li>
18    </ul>
19  </nav>
20  <section id="main">
21    <%= yield %>
22  </section>
23  <footer>
24    Desenvolvido no ebook "Conhecendo Rails"
25  </footer>
26  </body>
27 </html>

```

E indicamos em nosso controlador de login o nome do *layout* a ser utilizado:

```

1 class SessionController < ApplicationController
2   layout 'pub'
3   ...

```

E agora está tudo certo:

Bookstore - A maior livraria virtual (depois da Amazon)

- [Inicial](#)

Autenticação

Email

Senha

Desenvolvido no ebook "Conhecendo Rails"

Layout correto para página de login

Podemos inclusive ter um método que retorne qual layout que será utilizado, enviando o nome do método para o método layout:

```
1 class SessionController < ApplicationController
2   layout :layout_chooser
3
4   ...
5   private
6
7   def layout_chooser
8     case Time.current.hour
9     when 0..5
10      'madrugada'
11    when 6..12
12      'manha'
13    when 13..18
14      'tarde'
15    when 18..24
16      'noite'
17    else
18      'seila'
19    end
20  end
21  ...
```

Tomando mais cuidado ainda com o mass-assignment

Nesse ponto que já temos nossos métodos no modelo para fazer autenticação do usuário e nossas views e controladores prontos, podemos tomar mais um pouco de cuidado com o que recebemos do formulário. Lembram que foi mencionado que não é uma boa enviar `admin` pelo formulário? Isso poderia fazer com que um usuário “normal” ativasse esse *flag* e se tornasse um administrador do sistema.

A primeira coisa que precisamos fazer é evitar que esse atributo seja acessível através do formulário. Vamos alterar a *view* do formulário para não indicar mais `admin` como parte da *Hash* `person` que é enviada pelo formulário, e recebida no controlador, verificando com um teste funcional em `tests/controllers/people_controller_test.rb`:

```
1 test "should not show admin as a person form element" do
2   get edit_person_url(@person)
3   assert_select "input[name='person[admin]']", 0
4 end
```

E agora alterar a *view*, trocando de:

```

1 <div class="field">
2   <%= form.label :admin %><br>
3   <%= form.check_box :admin %>
4 </div>
```

para:

```

1 <div class="field">
2   <%= label_tag :admin %><br/>
3   <%= check_box_tag :admin %>
4 </div>
```

Reparem que removemos `form.label` e `form.check_box`, substituindo respectivamente por `label_tag` e `check_box_tag`. Isso vai criar elementos HTML desvinculados da *Hash* enviada pelo formulário. A partir desse momento, a indicação se uma pessoa é um administrador ou não está inutilizada para envio no objeto do modelo no formulário.

Para fechar a porta de vez, utilizando o *strong parameters*, vamos remover o atributo `admin` da lista de atributos permitidos para recebimento no controlador, dessa forma:

```

1 def person_params
2   params.require(:person).permit(:name, :email, :password, :password_confirmation, :born_at)
3 end
```



Dica

Reparem como os métodos utilizados em um formulário geram os elementos HTML. No caso do formulário de Person, são criados elementos com nomes como:

- `person[name]`
- `person[email]`

Isso permite que na hora que o controlador recebe os dados do formulário, possa criar um novo objeto de Person utilizando apenas

```

1 def create
2   @person = Person.new(params[:person])
3   ...
```

recebendo apenas os valores relativos à um objeto Person do formulário. O que fizemos acima exclui `admin` dos valores enviados em `person`, e lógico, que agora com o *strong parameters*, é altamente não recomendável passar `params` direto para criar ou atualizar um modelo, nesse caso, sendo utilizado `person_params`, já filtrado.

Removemos o atributo `admin` dos pontos dos nossos testes funcionais onde estava sendo enviado (nos testes de inclusão e atualização de pessoas) e fazemos um teste funcional simulando que se for enviado `admin` pelo formulário, não vai ativar o *flag* no novo registro:

```

1 test "should not set admin from mass assignment" do
2   sign_in @person.email, 'secret'
3
4   assert_difference('Person.count') do
5     post people_url, params: { person: { admin: true, born_at: @person.born_a\
6 t,
7     email: "functional@test.com", name: @person.name, password:
8     'testetedetamanhobom' } }
9   end
10
11   assert !Person.where(email: "functional@test.com").take.admin?
12 end

```

Agora precisamos de uma maneira de somente mostrar os atributos HTML para indicar se uma pessoa pode ser administrador ou não somente para quem é administrador. Vamos fazer dois testes, um com um administrador ou sistema e outro com um não-administrador, que deve e não deve mostrar os elementos HTML:

```

1 test "should show admin elements if current user is admin" do
2   assert @person.update_attribute(:admin, true)
3   sign_in @person.email, 'secret'
4
5   get edit_person_url(@person)
6   assert_select "input[type=checkbox][name=admin]", 1
7 end
8
9 test "should not show admin elements if current user is not admin" do
10  assert @person.update_attribute(:admin, false)
11  sign_in @person.email, 'secret'
12
13  get edit_person_url(@person)
14  assert_select "input[type=checkbox][name=admin]", 0
15 end

```

Para satisfazer os testes, alteramos a *view* do formulário para:

```

1 <% if session[:admin] %>
2   <div class="field">
3     <%= label_tag :admin %><br />
4     <%= check_box_tag :admin, true, @person.admin %>
5   </div>
6 <% end %>

```

Agora somente os administradores estão vendo esses elementos, mas espera: como fazer para marcar alguém como administrador, mesmo sendo um administrador que requisitou?

Para isso, vamos alterar direto no controlador, onde, nas ações de inserir e atualizar pessoas, vai ser verificado se o usuário corrente é um administrador, dessa maneira ajustando explicitamente o atributo admin.

Vamos verificar esse comportamento com testes, primeiro para inclusão:

```
1 test "should set the admin flag when asked and current user is an admin" do
2   assert @person.update_attributes(admin: true)
3   sign_in @person.email, 'secret'
4
5   assert_difference('Person.count') do
6     post people_url, params: { admin: true, person: { born_at: @person.born_a\
7 t, email: "functional@test.com", name: @person.name, password: 'testdetamanh\
8 obom' } }
9   end
10  assert Person.where(email: "functional@test.com").take.admin?
11 end
12
13 test "should not set the admin flag when asked and current user is an admin" \
14 do
15   assert @person.update_attributes(admin: false)
16   sign_in @person.email, 'secret'
17
18   assert_difference('Person.count') do
19     post people_url, params: { admin: false, person: { born_at: @person.born_\
20 at, email: "functional@test.com", name: @person.name, password: 'testdetaman\
21 hobom' } }
22   end
23   assert !Person.where(email: "functional@test.com").take.admin?
24 end
```

e agora para atualização:

```
1 test "should update person with admin flag when asked and current user is an \
2 admin" do
3   assert @person.update_attributes(admin: true)
4   sign_in @person.email, 'secret'
5   regular = people(:autor)
6   assert !regular.admin?
7
8   put person_url(regular), params: { admin: true, person: regular.attributes }
9   regular.reload
10  assert regular.admin?
11 end
12
13 test "should update person with admin flag as false when asked and current us\
14 \
```

```
14 er is an admin" do
15   assert @person.update_attributes(admin: true)
16   sign_in @person.email, 'secret'
17
18   put person_url(@person), params: { admin: false, person: @person.attributes\
19 }
20   @person.reload
21   assert !@person.admin?
22 end
23
24 test "should not update person with admin flag when asked and current user is\
25 not an admin" do
26   assert @person.update_attributes(admin: false)
27   sign_in @person.email, 'secret'
28
29   put person_url(@person), params: { admin: false, person: @person.attributes\
30 }
31   @person.reload
32   assert !@person.admin?
33 end
```

Para satisfazer os testes e a condição que queremos agora, vamos alterar o controlador para indicar explicitamente se queremos - e podemos - ajustar o atributo de administrador:

```
1 ...
2 def create
3   @person = Person.new(person_params)
4   @person.admin = params[:admin] if session[:admin]
5   flash[:notice] = "Pessoa salva" if @person.save
6   respond_with @person
7 end
8
9 def update
10  @person.admin = params[:admin] if session[:admin]
11  flash[:notice] = "Pessoa atualizada" if @person.update(person_params)
12  respond_with @person
13 end
14 ...
```

Utilizando presenters

Podemos ver que na lista de pessoas estamos testando, na *view*, se uma pessoa é um administrador ou não. Também estamos localizando a data de nascimento e podem aparecer mais alguns tipos de situação em que não desejamos que a *view* se encarregue de analisar o que está acontecendo para escolher um melhor jeito de apresentar a informação. É aí que entram os *presenters*, que utilizam o conceito de *decorators*.

Para utilizarmos os *presenters*, vamos criar um diretório chamado `app/presenters`, e vamos criar um arquivo chamado `person_presenter.rb`.

Após o Rails 3.x todo o conteúdo do diretório e subdiretórios em `app/*` serem carregados. Após isso, vamos inserir conteúdo no arquivo `app/presenters/person_presenter.rb`:

```
1 class PersonPresenter
2   attr_reader :person
3   delegate :id, :name, :email, to: :person
4
5   def initialize(person)
6     @person = person
7   end
8
9   def admin
10    @person.admin ? 'Sim' : 'Não'
11  end
12
13  def born_at
14    helpers.l(@person.born_at)
15  end
16
17  def password
18    '*' * 10
19  end
20
21 private
22
23  def helpers
24    ApplicationController.helpers
25  end
26 end
```

Com o *presenter* criado, podemos alterar o controlador para criar um *presenter*, reparando no `link_to` onde apontamos para o registro de `person`:

```
1 def show
2   @person = PersonPresenter.new(@person)
3   respond_with @person
4 end
```

E agora alterar a *view* para utilizar o *presenter*:

```
1 <p id="notice"><%= notice %></p>
2
3 <p>
4   <strong><%= Person.human_attribute_name(:name) %></strong>
5   <%= @person.name %>
6 </p>
7
8 <p>
9   <strong><%= Person.human_attribute_name(:email) %></strong>
10  <%= mail_to @person.email, nil, encode: 'hex' %>
11 </p>
12
13 <p>
14   <strong><%= Person.human_attribute_name(:password) %></strong>
15   <%= @person.password %>
16 </p>
17
18 <p>
19   <strong><%= Person.human_attribute_name(:born_at) %></strong>
20   <%= @person.born_at %>
21 </p>
22
23 <p>
24   <strong><%= Person.human_attribute_name(:admin) %></strong>
25   <%= @person.admin %>
26 </p>
27
28 <%= link_to 'Editar', edit_person_path(@person.person) %> |
29 <%= link_to 'Voltar', people_path %>
```

Isso vai nos dar um resultado em *show* como esse:

Interface de administração

- Pessoas

Nome: Eustáquio Rangel de Oliveira Jr.

E-mail: tag@bluefish.com.br

Senha: *****

Data de nascimento: 01/01/1970

Administrador: Sim

[Editar](#) | [Voltar](#)

Desenvolvido no ebook "Conhecendo Rails"

Mostrando usuário com presenters

Pudemos ver que estamos listando os métodos não implementados no *presenter* e delegados para person através do método `delegate`. Quando temos poucos métodos, não tem problema, mas quando existem muitos métodos para serem listados, é melhor utilizar uma classe como a `SimpleDelegator90` e definir nosso *presenter* como no exemplo abaixo:

⁹⁰<https://ruby-doc.org/stdlib-2.4.1/libdoc/delegate/rdoc/SimpleDelegator.html>

```
1 class PersonPresenter < SimpleDelegator
2   def admin
3     super ? 'Sim' : 'Não'
4   end
5
6   def born_at
7     helpers.l(super)
8   end
9
10  def password
11    '*' * 10
12  end
13
14  private
15
16  def helpers
17    ApplicationController.helpers
18  end
19 end
```

Dessa maneira, os métodos que não são encontrados na classe corrente são automaticamente delegados para o objeto `@person`, que foi utilizado na criação do *presenter*. Reparem que por estarmos definindo métodos que tem o mesmo nome do objeto “interno”, utilizamos `super` para chamar o método homônimo. Nesse caso, temos que retornar o `link_to` da `show.html.erb` para o jeito que era antes, “normal”, porque agora não vamos ter mais a referência de `person` no `decorator`, que vai lidar com tudo de maneira transparente!

Existe uma gem chamada `draper`⁹¹ que faz um trabalho interessante com os *decorators* e vale uma olhada.

⁹¹<https://github.com/drapergem/draper>

Criando um controlador para o público

Vamos aproveitar e agora criar um controlador para expor nossa aplicação para o público. Existem outras maneiras de fazer isso, como direto nos controladores já criados expor ou não determinadas informações e controles, a utilização de *namespaces*⁹², como já demonstrado anteriormente.

Mas vamos simplificar nesse momento e definir que os acessos sem autenticação, pelo público em geral, será feito através de um controlador chamado pub, que no momento da sua criação terá duas ações, index e sobre:

```

1 $ rails g controller pub index sobre
2   create  app/controllers/pub_controller.rb
3     route  get 'pub/sobre'
4     route  get 'pub/index'
5   invoke  erb
6 ...

```

Agora vamos definir alguns testes funcionais para esse controlador, inclusive verificando se o endereço *default* do nosso site (o raiz, ou /) aponta para a ação index, preenchendo o arquivo test/controllers/pub_controller_test.rb com o seguinte conteúdo:

```

1 require 'test_helper'
2
3 class PubControllerTest < ActionDispatch::IntegrationTest
4   test "should get index" do
5     get root_path
6     assert_response :success
7
8     assert_select "h1", "Bookstore - A maior livraria virtual (depois da Amaz\"
9 on)"
10    assert_routing({ path: root_path }, {controller: "pub", action: "index" })
11  end
12
13  test "should get sobre" do
14    get pub_sobre_url
15    assert_response :success
16  end
17 end

```

Vamos comentar a rota criada (no arquivo config/routes.rb) para pub#index e inseri-la como root, de forma que sera exibida sempre que for aberta a aplicação no endereço do seu domínio:

⁹²<http://guides.rubyonrails.org/routing.html#controller-namespaces-and-routing>

```
1   Rails.application.routes.draw do
2     #get 'pub/index'
3     root 'pub#index'
```

Que vai nos produzir o seguinte conteúdo quando acessarmos `http://localhost:3000:`

Bookstore - A maior livraria virtual (depois da Amazon)

- [Inicial](#)

Pub#index

Find me in `app/views/pub/index.html.erb`

Desenvolvido no ebook "Conhecendo Rails"

Página inicial

Reparam que não precisamos especificar no controlador o nome do *layout* a ser utilizado, só com o nome do controlador já vai assumir o nome do *layout* correspondente.

Agora que já temos um controlador para o público, vamos alterar o nosso controlador de pessoas para validar se o usuário está autenticado e é um administrador. Para isso, vamos fazer um novo controlador, `AdminController`, que vai herdar de `LoggedController` e inserir uma nova ação verificação com o método `admin?:`

```
1 class AdminController < LoggedController
2   before_action :admin?
3
4   private
5
6   def admin?
7     raise NotAdmin unless session[:admin]
8   end
9 end
```

Alterar o controlador de pessoas para herdar desse controlador novo:

```
1 class PeopleController < AdminController
2   ...
```

Alterar o `ApplicationController` para definir a nova exceção o jeito de lidar com ela:

```
1 class ApplicationController < ActionController::Base
2   protect_from_forgery with: :exception
3
4   NotAuthenticated = Class.new(StandardError)
5   NotAdmin = Class.new(StandardError)
6
7   rescue_from ActiveRecord::RecordNotFound, with: :record_not_found
8   rescue_from NotAuthenticated, with: :not_authenticated
9   rescue_from NotAdmin, with: :not_admin
10 ...
11
12 def not_admin
13   flash[:notice] = 'Você precisa ser administrador'
14   redirect_to root_path
15 end
```

Esse método novo, `admin?`, vai verificar se o usuário foi identificado como administrador no momento em que foi autenticado, quando vamos gravar o *flag* de administrador em uma variável de sessão, que vai ser verificada por esse método.

Temos que alterar o teste funcional para refletir o novo redirecionamento, ao invés da página de login, para a página principal, junto com a mensagem:

```
1 test 'should redirect to login page' do
2   get people_url
3   assert_redirected_to root_path
4   assert_equal 'Você precisa ser administrador', flash[:notice]
5 end
```

E remover o teste que verificava que não poderia atualizar a pessoa como administrador se a pessoa que fez login não for administrador, porque agora nem tem mais jeito de entrar no controlador se não for um administrador. Ao invés de remover, podemos também alterar o teste para:

```
1 test "should not set the admin flag when asked and current user is an admin" \
2 do
3   assert @person.update_attributes(admin: false)
4   sign_in @person.email, 'secret'
5
6   assert_no_difference('Person.count') do
7     post people_url, params: { admin: true, person: { born_at: @person.born_a\
8 t,
9     email: "functional@test.com", name: @person.name, password:
10    'testdetamanhobom' } }
11   end
12   assert_redirected_to root_path
13 end
```

Associações entre modelos

Para testar as associações entre os nossos modelos, temos que criar um novo, pois só temos um. Vamos criar um *scaffold* novo com o modelo Book, que terá os seguintes atributos:

- **title** - Título
- **published_at** - Data de publicação
- **text** - Texto descritivo do livro
- **value** - Valor do livro
- **person_id** - Autor do livro

Criando o novo *scaffold*:

```
1 $ rails g scaffold Book title:string published_at:date text:text value:decimal\\
2 1 person:references
3   invoke active_record
4   create db/migrate/20170311144612_create_books.rb
5   create app/models/book.rb
6   invoke test_unit
7   create test/models/book_test.rb
8   create test/fixtures/books.yml
9   ...
10  ...
11  ...
12  ...
```

Agora é adaptar e rodar as *migrations*, alterar os testes unitários e funcionais, limitar o acesso ao controlador de livros para somente quem tiver feito o login, verificar o *layout* do controlador e rodar os testes para ver se está tudo ok.

Nem vamos escrever por aqui como faz isso, pois é basicamente o que fizemos com o controlador de pessoas, somente algumas observações para a *migration*, vista aqui já alterada:

```
1 class CreateBooks < ActiveRecord::Migration[5.0]
2   def change
3     create_table :books do |t|
4       t.string :title, limit: 100, null: false
5       t.date :published_at, null: false
6       t.text :text, null: false
7       t.decimal :value, precision: 10, scale: 2, null: false
8       t.references :person, foreign_key: true
9       t.timestamps
10    end
11  end
12 end
```

Dando uma olhada no que customizamos nas colunas da tabela do banco de dados:

- A coluna `title` vai ter um limite de 100 caracteres (`limit: 100`) e não pode ter o seu valor nulo (`null: false`).
- A coluna `published_at` não pode ter o seu valor nulo.
- A coluna `text` não pode ter o seu valor nulo.
- A coluna `value` não pode ter o seu valor nulo, tem que ter precisão de 10 dígitos (`precision: 10`) sendo que 2 dígitos (`scale: 2`) são utilizados como casas decimais, ou seja, temos um tamanho de 8 dígitos antes da casa decimal.
- Foi criada uma *referência* para outra tabela, através de `:person` (ou seja, referenciando a tabela `People`, de acordo com as convenções), sendo definida como chave estrangeira⁹³. Isso leva a criar uma coluna chamada `person_id` na tabela `Book`, que é a chave estrangeira que aponta para o `id` da tabela `People` que está referenciado em `person_id`.

Vamos adaptar a nossa *fixture* de livro:

```

1 one:
2   title: Conhecendo Ruby
3   published_at: 2013-06-29
4   text: Livro prático sobre a linguagem Ruby
5   value: 1.00
6   person: admin
7
8 two:
9   title: Conhecendo o Git
10  published_at: 2013-06-24
11  text: Quer aprender Git de forma rápida e prática?
12  value: 10.00
13  person: admin

```

Reparam que utilizei, ao invés de `person_id`, o **nome da associação**, `person`, e a chave da *fixture* de pessoas, `admin`, para indicar na *fixture* que o livro está associado com a pessoa da outra *fixture*. Eu poderia ter utilizado `autor`, mas fui xarope e utilizei `admin` para fazer propagandas dos meus livros e *ebooks*. ;-)

Isso funciona pois automagicamente quando utilizamos `references` ao criar a *migration*, o Rails já embutiu código dentro do modelo, que vamos ver logo abaixo. Antes de mais nada vamos alterar nossos testes (que, após a adaptação da *fixture* acima já devem estar rodando de boa) para refletir o **comportamento** que queremos que o modelo do livro apresente, seja baseado no que definimos no banco de dados (que é o que vamos fazer aqui, limitar pelas *constraints* inseridas no banco) ou em alguma regra específica para ele.

Vamos alterar nosso teste do modelo `Book`, presente em `test/models/book_test.rb` para:

⁹³https://pt.wikipedia.org/wiki/Chave_estrangeira

```
1 require 'test_helper'
2
3 class BookTest < ActiveSupport::TestCase
4   setup do
5     @book = books(:one)
6   end
7
8   # título
9   test 'deve ter um título' do
10    @book.title = nil
11    assert !@book.valid?
12  end
13
14  test 'não pode ter mais que 100 caracteres' do
15    @book.title = '*' * 101
16    assert !@book.valid?
17  end
18
19  # data de publicação
20  test 'deve ter data de publicação' do
21    @book.title = nil
22    assert !@book.valid?
23  end
24
25  # texto
26  test 'deve ter texto' do
27    @book.text = nil
28    assert !@book.valid?
29  end
30
31  # valor
32  test 'deve ter valor' do
33    @book.value = nil
34    assert !@book.valid?
35  end
36
37  test 'deve ser um número' do
38    @book.value = 'bla'
39    assert !@book.valid?
40  end
41
42  test 'não deve ser maior que o permitido' do
43    @book.value = 100000000.00
44    assert !@book.valid?
45  end
46
```

```

47  # pessoa
48  test 'deve ter pessoa' do
49    @book.person = nil
50    assert !@book.valid?
51  end
52 end

```

**Dica**

O momento de definição dos testes é o melhor momento para se pensar no **comportamento** do seu objeto, antes de pensar em como será a **implementação** desse comportamento.

Associação do tipo um-para-um

Nosso livro (o da aplicação, não esse que você está lendo) precisa de uma pessoa que seja o autor, que desejamos acessar como um método chamado `person`, em um objeto representando um livro. Esse foi um dos testes que inserimos acima, o último, por sinal, no arquivo de teste unitário de livro.

Vamos dar uma olhada em arquivo do modelo de Book, ainda sem as validações necessárias para passar nos testes, mas já com o resultado da automagicamente criada coluna especificada na *migration* através de `references`:

```

1 class Book < ApplicationRecord
2   belongs_to :person
3 end

```

Ali no modelo foi indicado automagicamente (lógico que podemos inserir e alterar isso na hora que quisermos) que um livro pertence à uma pessoa, utilizando o método `belongs_to`, ou seja *pertence à*. Dessa forma, foi criada uma associação no ORM indicando uma dependência (forte, por sinal, é uma *foreign key* do livro para a pessoa).

Essa associação, por padrão, a partir do Rails 5, **não pode ficar vazia**, ou seja, se um livro pertence à uma pessoa, deve obrigatoriamente conter um registro válido, uma *foreign key* válida, ali. Se por acaso desejarmos por algum motivo que essa verificação seja afrouxada, podemos especificar no final a Hash `optional: true`, dessa forma:

```

1 class Book < ApplicationRecord
2   belongs_to :person, optional: true
3 end

```

Isso é útil para migrar aplicações de versões anteriores que permitiam esse comportamento. Podemos até definir esse comportamento para a aplicação inteira, utilizando o seguinte código em um `initializer`:

```
1 Rails.application.config.active_record.belongs_to_required_by_default = false
```

Mas vamos deixar **sem** o `optional` e manter o que já vem definido por padrão.

Vamos terminar de inserir as validações necessárias para o modelo passar nos testes:

```
1 class Book < ApplicationRecord
2   validates :title, presence: true, length: { maximum: 100 }
3   validates :published_at, presence: true
4   validates :text, presence: true
5   validates :value, presence: true, numericality: { less_than_or_equal_to: 99 \
6   999999.99 }
7   validates :person, presence: true
8
9   belongs_to :person
10 end
```

Vamos ver como funcionam essas validações logo mais no livro, mas tem algumas ali que até já são auto-explicativas. Se rodarmos nossos testes agora, todos devem passar.

Como temos a associação com pessoa, em nossos objetos de livros, ela já pode ser testada no `console` ou no navegador:

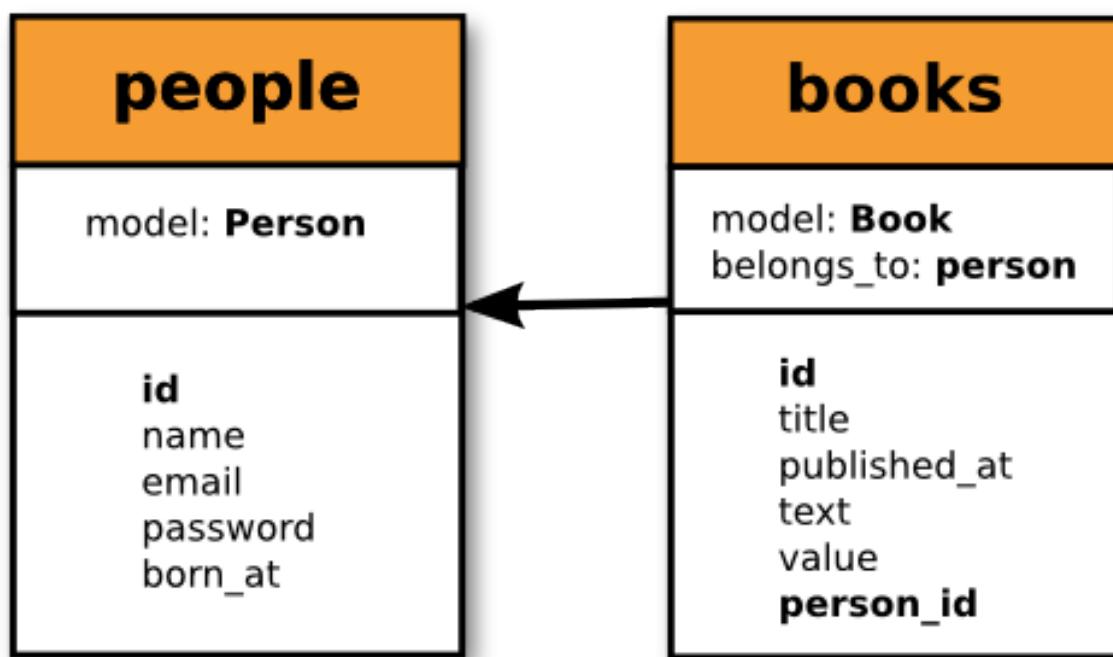
```
1 book = Book.new
2 => #<Book id: nil, title: nil, published_at: nil, text: nil, value: nil, pers\
3 on_id: nil, created_at: nil, updated_at: nil>
4
5 book.title = "Ruby - Conhecendo a Linguagem"
6 => "Ruby - Conhecendo a Linguagem"
7
8 book.published_at = "2006-03-01"
9 => "2006-03-01"
10
11 book.text = "Tinha, mas acabou."
12 => "Tinha, mas acabou."
13
14 book.value = 40.00
15 => 40.0
16
17 book.person = Person.last
18 Person Load (0.3ms)  SELECT "people".* FROM "people" ORDER BY name DESC LIMIT \
19 1
20 => #<Person id: 2, name: "Eustáquio Rangel", email: "taq@bluefish.com.br", \
21 password: "9733340c840c719779f234407ee0bac26ae8904b", born_at: "1971-04-06", \
22 admin: true, created_at: "2013-07-06 22:51:34", updated_at: "2013-07-07 \
23 22:48:20">
```

```

24
25 book.person.name
26 => "Eustáquio Rangel"
27
28 book.save
29 (0.1ms) begin transaction
30 SQL (28.9ms) INSERT INTO "books" ("created_at", "person_id", "published_at",
31 "text", "title", "updated_at", "value") VALUES (?, ?, ?, ?, ?, ?, ?)
32 [[["created_at", Tue, 09 Jul 2013 15:15:46 UTC +00:00], ["person_id", 2],
33 ["published_at", Wed, 01 Mar 2006], ["text", "Tinha, mas acabou."], ["title",
34 "Ruby - Conhecendo a Linguagem"], ["updated_at", Tue, 09 Jul 2013 15:15:46 UTC
35 +00:00], ["value", #<BigDecimal:ae70608, '0.4E2',9(45)>]]
36 (114.1ms) commit transaction
37 => true

```

Ficando assim:



Livro pertence à pessoa

Associação do tipo um-para-muitos

Já que um livro pertence à uma pessoa, agora podemos especificar que uma pessoa pode ter vários livros. Primeiro, no teste unitário:

```
1 test "deve ter uma coleção de livros" do
2   person = people(:admin)
3   assert_respond_to person, :books
4   assert_kind_of Book, person.books.first
5 end
```

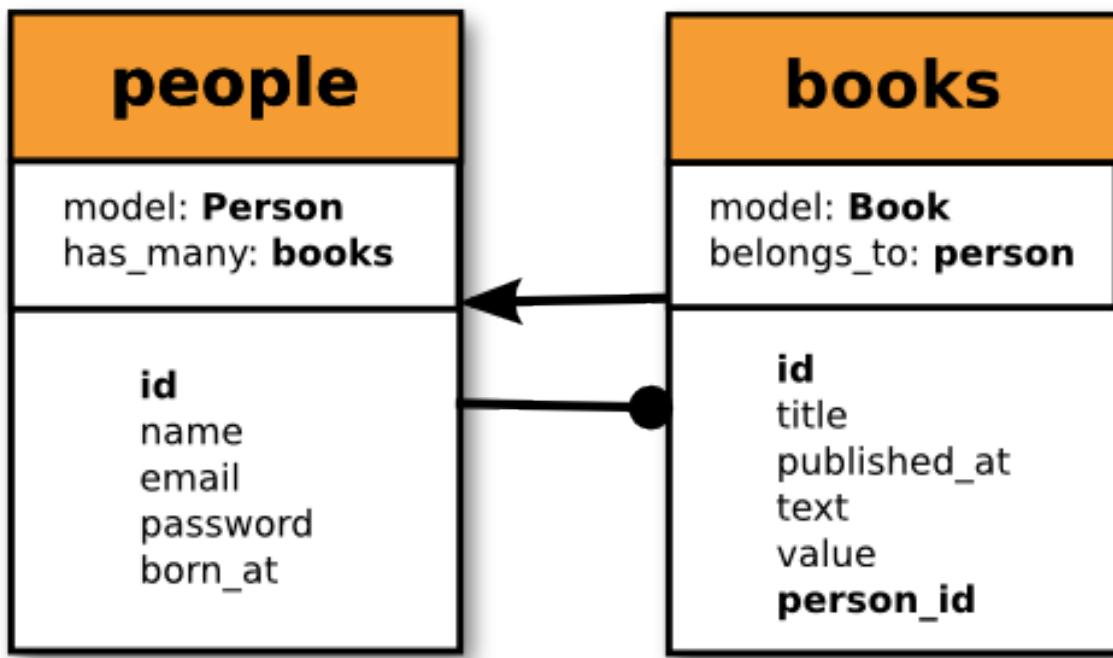
E no modelo, estabelecendo a associação, usando `has_many`:

```
1 has_many :books
```

Através do `has_many` temos uma coleção de livros no objeto da pessoa, como podemos verificar pelo *console*:

```
1 p = Person.last
2 Person Load (0.3ms)  SELECT "people".* FROM "people" ORDER BY name DESC LIMIT\ 
3 1
4 => #<Person id: 2, name: "Eustáquio Rangel", email: "taq@bluefish.com.br",
5 password: "9733340c840c719779f234407ee0bac26ae8904b", born_at: "1971-04-06",
6 admin: true, created_at: "2013-07-06 22:51:34", updated_at: "2013-07-07
7 22:48:20">
8
9 p.books
10 Book Load (0.1ms)  SELECT "books".* FROM "books" WHERE "books"."person_id" = 2
11 => [#<Book id: 1, title: "Ruby - Conhecendo a Linguagem", published_at:
12 "2006-03-01", text: "Tinha, mas acabou.", value:
13 #<BigDecimal:aaaae9c0,'0.4E2',9(36)>, person_id: 2, created_at: "2013-07-09
14 15:15:46", updated_at: "2013-07-09 15:15:46">]
15
16 p.book_ids
17 => [1]
18
19 p.books.last.title
20 => "Ruby - Conhecendo a Linguagem"
```

Ficando assim:



Pessoa tem muitos livros

Fica aqui uma dica sobre o que acontece quando apagamos o registro da pessoa com vários livros. O comportamento padrão é não-intrusivo e não faz nada, mas podemos especificar o que deve ser feito através de :dependent, da seguinte forma:

```
1 has_many :books, dependent: :destroy
```

Os seguintes comportamentos estão disponíveis para :dependent:

- **destroy** - Os objetos associados são destruídos junto com o objeto corrente, chamando os métodos `destroy` de cada objeto.
- **delete_all** - Os objetos associados são apagados sem chamar o método `destroy` de cada um.
- **nullify** - Todas as chaves estrangeiras dos objetos associados são transformadas em nulo, sem chamar os *callbacks* de atualização dos objetos.
- **restrict_with_error** - Evita que o objeto seja apagado, retornando `false` se tem mais objetos associados.
- **restrict_with_exception** - Evita que o objeto seja apagado, disparando uma exceção se tem mais objetos associados.



Atenção

Muito cuidado com o `:dependent`: `:destroy`, ainda mais se for utilizado o `callback before_destroy`. Existem situações em que, mesmo se o `callback` retornar `false`, indicando que o objeto em questão não deve ser destruído, os objetos da coleção em `has_many` já foram! Deêm uma olhada em uma [discussão de como isso funciona⁹⁴](#) e fiquem prevenidos.

Podemos fazer testes unitários em `Person` para verificar que o comportamento de `:restrict_with_exception` ou `:destroy` são obedecidos, e para termos certeza de que o modelo está configurado corretamente de acordo com a nossa escolha.

Para o teste de `:restrict_with_exception`, nesse caso alterando o modelo para

```
1 has_many :books, dependent: :restrict_with_exception
```

podemos utilizar:

```
1 test "não pode apagar a pessoa se ela tiver livros" do
2   person = people(:admin)
3   assert person.books.size > 0
4
5   assert_no_difference('Person.count') do
6     assert_no_difference('Book.count') do
7       assert_raise ActiveRecord::DeleteRestrictionError do
8         assert !person.destroy, "não deveria apagar com #{person.books.size} \
9 livro(s)"
10      end
11    end
12  end
13 end
```

Rodando os testes, podemos ver que tudo está funcionando de acordo. Agora, para o teste de `:destroy`, vamos alterar o modelo para

```
1 has_many :books, dependent: :destroy
```

e nos testes podemos comentar o teste anterior e inserir o seguinte código:

⁹⁴<https://github.com/rails/rails/issues/3458>

```

1 test "deve apagar os livros quando apagar a pessoa" do
2   person = people(:admin)
3   assert person.books.size > 0
4
5   assert_difference('Person.count', -1) do
6     assert_difference('Book.count', person.books.size * -1) do
7       assert person.destroy, "deveria apagar a pessoa"
8     end
9   end
10 end

```

Podemos notar que o primeiro comportamento e teste é para *evitar* que a pessoa seja apagada se tiver itens na coleção, chutando o pau da barraca e disparando uma `Exception` se isso ocorrer, enquanto que o segundo é para *garantir* que os itens da coleção sejam apagados se a pessoa for apagada.



Dica

Se qualquer um dos comportamentos acima for implementado, vamos ter um erro no teste de controlador de pessoas, justamente onde o registro é apagado. Para corrigir isso, podemos apagar todos os livros da pessoa com `@person.books.destroy_all` antes de fazer a chamada no controlador.

Associações de muitos para muitos

Vamos criar um novo `scaffold` para cadastrarmos as categorias que nossos livros pertencem. Os livros cadastrados até aqui já pertencem à duas categorias que podemos definir como “tecnologia” e “programação”. Vamos criar mais um `scaffold` bem simples para gerenciar as categorias, e já rodar as `migrations`:

```

1 $ rails g scaffold Category name:string
2   invoke  active_record
3   create    db/migrate/20170312135948_create_categories.rb
4   create    app/models/category.rb
5   invoke  test_unit
6 $ rails db:migrate

```

Após o `scaffold` criado, podemos acessar `http://localhost:3000/categories` e criar as categorias mencionadas acima. Se quiserem adicionar outras para ver como fica a seleção, fiquem à vontade. Vamos adaptar também as `fixtures` criadas para refletir nas chaves algo mais identificável nas categorias, não esquecendo de trocar nos testes funcionais as chaves onde está sendo referenciado `one` para alguma das que vamos criar agora:

```
1 tech:  
2   name: Tecnologia  
3  
4 dev:  
5   name: Desenvolvimento
```

Para deixar as categorias disponíveis no gerenciamento de livros, no controlador de livros devemos pedir para que sejam carregadas, através do método `before_action`:

```
1 class BooksController < ApplicationController  
2   before_action :set_book, only: [:show, :edit, :update, :destroy]  
3   before_action :load_categories, only: [:new, :edit, :create, :update]  
4   ...  
5  
6   private  
7   def load_categories  
8     @categories = Category.all  
9   end  
10  ...  
11 end
```

Agora que temos disponíveis as categorias, precisamos de um meio de indicar que um livro tem várias categorias, e que a categoria tem vários livros. Para isso vamos utilizar uma tabela auxiliar, ou *join table*, para armazenar o id do livro e os ids (zero ou mais) das categorias dos livros. Vamos utilizar o método `has_and_belongs_to_many`, que até o Rails 4 ficou na controvérsia se seria ainda utilizado ou se seria marcado como *deprecated*, mas que até agora está funcionando muito bem, e é bem simples de implementar.

Vamos criar a *join table* com os nomes dos dois modelos, **em ordem alfabética**, isso é muito importante, criando a seguinte *migration*:

```
1 $ rails g migration CreateJoinTableBookCategory book category  
2       invoke active_record  
3       create db/migrate/20170312145600_create_join_table_book_category.rb
```

Ela vai ter um conteúdo como esse, já removidos os comentários:

```

1 class CreateJoinTableBookCategory < ActiveRecord::Migration[5.0]
2   def change
3     create_join_table :books, :categories do |t|
4       t.index [:book_id, :category_id]
5       t.index [:category_id, :book_id]
6     end
7   end
8 end

```

Rodando a *migration*, vai ser criada a tabela books_categories, que **não vai ter um modelo associado, e nem um id**. Esses são detalhes muito importantes em uma *join table* desse tipo:

```

1 $ rails db:migrate
2 == 20170312145600 CreateJoinTableBookCategory: migrating =====
3 ==
4 -- create_join_table(:books, :categories)
5 -> 0.0076s
6 == 20170312145600 CreateJoinTableBookCategory: migrated (0.0076s)
7 =====

```

Agora vamos indicar no modelo de livro que ele tem vários relacionamentos com esse modelo novo, e que tem várias categorias. Antes de mais nada, testes no modelo de livro:

```

1 # categorias
2 test 'deve ter categorias' do
3   assert_respond_to @book, :categories
4 end
5
6 test 'deve ter categorias do tipo correto' do
7   @book.categories << categories(:one)
8   assert_kind_of Category, @book.categories.first
9 end

```

E agora sim podemos alterar o modelo de livro indicando que ele pertence à várias categorias:

```

1 class Book < ApplicationRecord
2 ...
3   has_and_belongs_to_many :categories
4 ...
5 end

```

Isso nos dá os seguintes métodos em Book:

```

1 > b = Book.first
2 Book Load (0.1ms)  SELECT "books".* FROM "books" ORDER BY "books"."id" ASC
3 LIMIT ?  [{"LIMIT": 1}] => #<Book id: 14, title: "Conhecendo Ruby",
4 published_at: "2013-06-29", text: "Livro prático sobre a linguagem Ruby",
5 value: #<BigDecimal:1701a10,'0.1E1',9(18)>, person_id: 15, created_at:
6 "2017-03-12 15:22:07", updated_at: "2017-03-12 15:22:07">
7
8 > b.categories
9 Category Load (0.2ms)  SELECT "categories".* FROM "categories" INNER JOIN
10 "books_categories" ON "categories"."id" = "books_categories"."category_id"
11 WHERE "books_categories"."book_id" = ?  [{"book_id": 14}] =>
12 #<ActiveRecord::Associations::CollectionProxy [#<Category id: 1, name:
13 "Tecnologia", created_at: "2017-03-12 14:02:51", updated_at: "2017-03-12
14 14:02:51">, #<Category id: 2, name: "Programação", created_at: "2017-03-12
15 14:02:51", updated_at: "2017-03-12 14:02:51">] >
16
17 > b.category_ids
18 => [1, 2]

```

E agora podemos fazer a mesma coisa no modelo de categoria:

```

1 class Category < ApplicationRecord
2   has_and_belongs_to_many :books
3 end

```

Que nos dá os métodos:

```

1 > c = Category.first
2 Category Load (0.4ms)  SELECT "categories".* FROM "categories" ORDER BY
3 "categories"."id" ASC LIMIT ?  [{"LIMIT": 1}]
4 => #<Category id: 1, name: "Tecnologia", created_at: "2017-03-12 14:02:51",
5 updated_at: "2017-03-12 14:02:51">
6
7 > c.books
8 Book Load (0.3ms)  SELECT "books".* FROM "books" INNER JOIN "books_categories"
9 ON "books"."id" = "books_categories"."book_id" WHERE
10 "books_categories"."category_id" = ?  [{"category_id": 1}] =>
11 #<ActiveRecord::Associations::CollectionProxy [#<Book id: 14, title:
12 "Conhecendo Ruby", published_at: "2013-06-29", text: "Livro prático sobre a
13 linguagem Ruby", value: #<BigDecimal:3799958,'0.1E1',9(18)>, person_id: 15,
14 created_at: "2017-03-12 15:22:07", updated_at: "2017-03-12 15:22:07">, #<Bo
15 ok
16 id: 15, title: "Conhecendo o Git", published_at: "2013-06-24", text: "Quer
17 aprender Git de forma rápida e prática?", value:

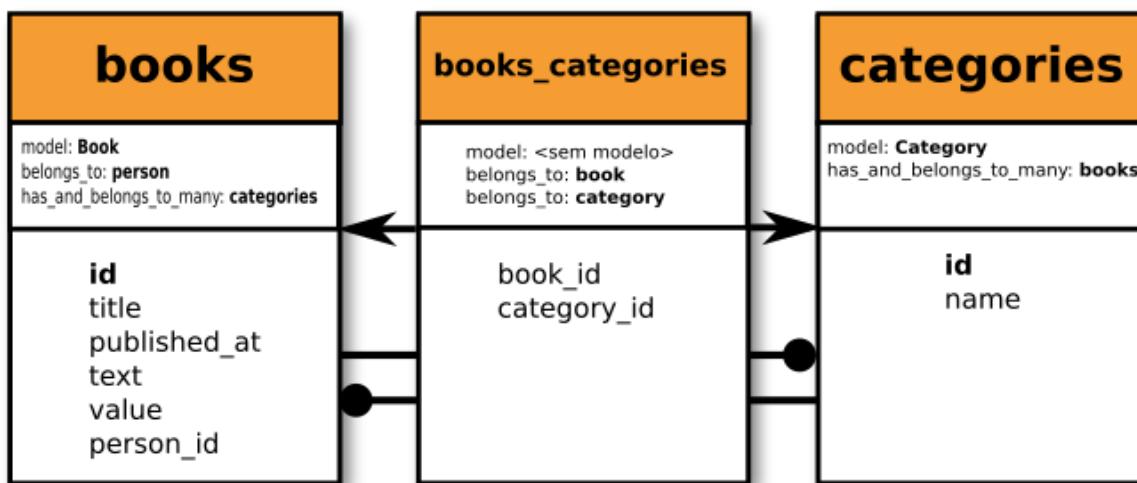
```

```

19  #<BigDecimal:378fef8, '0.1E2',9(18)>, person_id: 15, created_at: "2017-03-12
20  15:22:07", updated_at: "2017-03-12 15:22:07">] >
21
22 > c.book_ids
23 => [14, 15]

```

Ficando assim:



Livros e categorias

Vamos aproveitar o método category_ids de Book para fazer a nossa escolha de categorias. Primeiro, vamos liberar nos **strong parameters** do controlador dos livros o envio desse atributo, dessa forma:

```

1 def book_params
2   params.require(:book).permit(:title, :published_at, :text, :value, :person_\
3   id, category_ids: [])
4 end

```



Reparem como que foi liberado o atributo, permitindo o envio de um Array

E listamos as categorias disponíveis no formulário dos livros em app/views/books/_form.html.erb através da coleção já presente no controlador na variável @categories, com o método collection_check_boxes:

```

1 <div class="field">
2   <h2>Categorias</h2>
3   <%= collection_check_boxes :book, :category_ids, @categories, :id, :name \ 
4 %>
5 </div>

```

O que vai nos dar uma lista com as categorias cadastradas na forma de *checkboxes* no formulário de edição dos livros. Ficou legal, mas podemos melhorar para deixar com uma semântica mais adequada. Particularmente, eu prefiro gerar um elemento `ul` com um elemento `li` para cada categoria. Podemos personalizar da seguinte forma:

```

1 <ul>
2 <%= collection_check_boxes :book, :category_ids, @categories, :id, :name do | \
3 builder| %>
4   <li>
5     <%= builder.label { builder.check_box + builder.text } %>
6   </li>
7 <% end %>
8 </ul>

```

Agora sim ficou legal. Vamos alterar a view `app/views/books/show.html.erb` para listar as categorias cadastradas do livro:

```

1 <h2>Categorias</h2>
2 <ul>
3   <% for category in @book.categories %>
4     <li><%= category.name %></li>
5   <% end %>
6 </ul>

```

Associações de muitos para muitos, através

Agora que temos uma pessoa com vários livros, um livro que pertence à uma pessoa e tem várias categorias, sendo que uma categoria tem vários livros, podemos pensar o seguinte: e se quisermos saber quais as *categorias* de *livros* que uma *pessoa* tem? Podemos ver que está tudo conectado na modelagem conforme descrito, só precisamos de um jeito de, quando estivemos na pessoa, recuperarmos a categoria *através* (essa palavra é muito importante) aqui dos livros. Para isso, vamos usar a associação `has_many :through`. Primeiro, vamos alterar a *fixture* dos livros para incluírem as categorias, enviadas como um Array, dessa forma:

```

1 one:
2   title: Conhecendo Ruby
3   published_at: 2013-06-29
4   text: Livro prático sobre a linguagem Ruby
5   value: 1.00
6   person: admin
7   categories: [ tech, dev ]
8
9 two:
10  title: Conhecendo o Git
11  published_at: 2013-06-24
12  text: Quer aprender Git de forma rápida e prática?
13  value: 10.00
14  person: admin
15  categories: [ tech, dev ]

```

Agora um teste de pessoa:

```

1 # categorias
2 test 'deve ter várias categorias, através de livros' do
3   assert_respond_to @person, :categories
4 end
5
6 test 'deve ter uma categoria do tipo correto' do
7   assert_kind_of Category, @person.categories.first
8 end
9
10 test 'deve ter apenas duas categorias' do
11   assert_equal 2, @person.categories.count
12 end

```

Convém atentar para o último teste. Ele procura garantir que são encontradas apenas 2 categorias, mas temos 2 livros com 2 categorias cada um, de modo que vão ser retornadas 4 categorias. Vamos resolver isso já, mas antes vamos ver o método tradicional de indicar que *pessoa* tem várias *categorias* através de *livro*, inserindo no arquivo do modelo de pessoa a seguinte linha:

```

1 ...
2 has_many :categories, through: :books
3 ...

```

Novamente, uma configuração bem simples, que nos dá as categorias da pessoa:

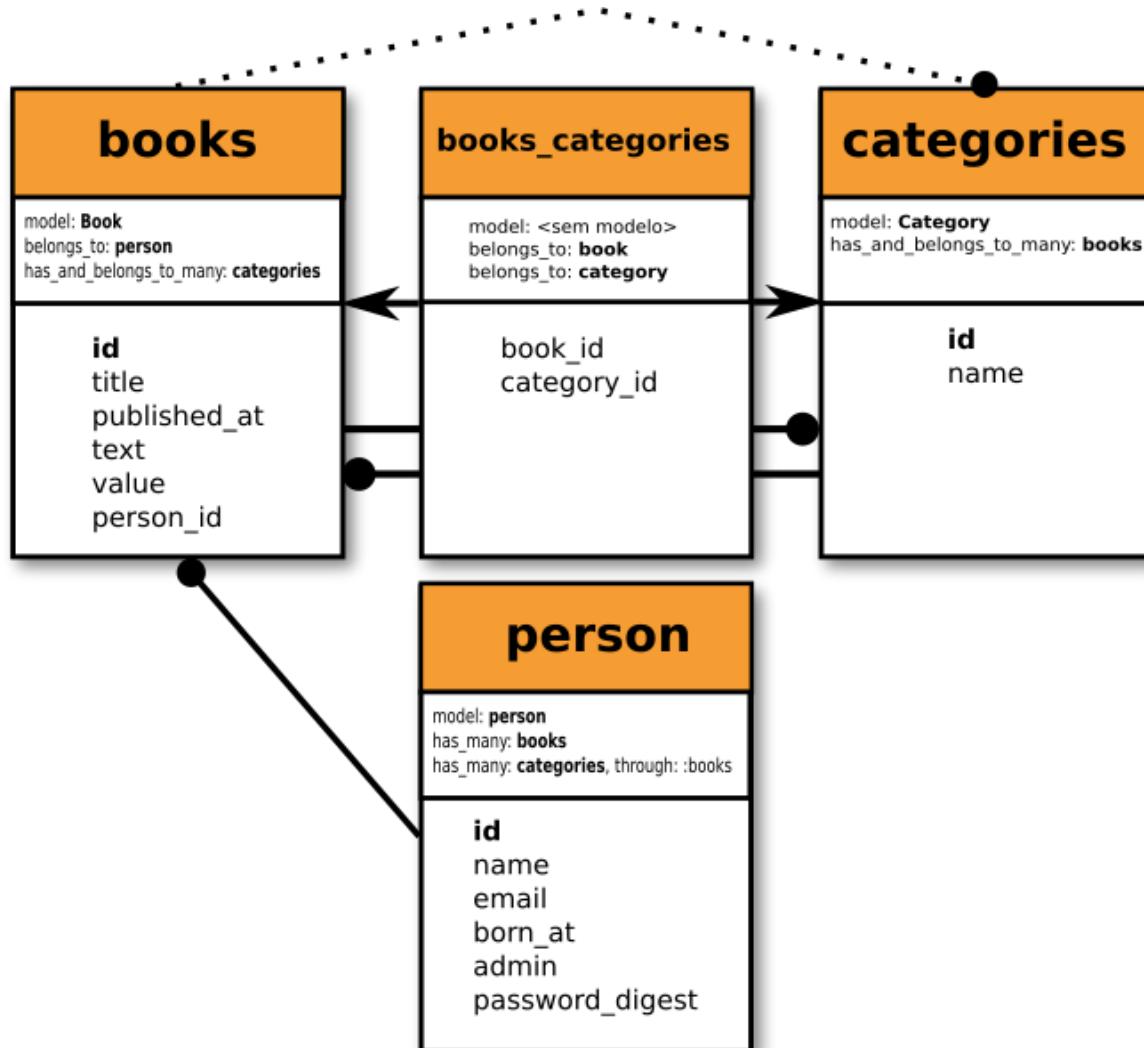
```
1 > Person.last.categories
2   Person Load (0.4ms)  SELECT "people".* FROM "people" ORDER BY "people"."na\
3 me"
4   DESC LIMIT ?  [[{"LIMIT": 1}]] Category Load (0.1ms)  SELECT "categories".* F\
5 ROM
6   "categories" INNER JOIN "books_categories" ON "categories"."id" =
7     "books_categories"."category_id" INNER JOIN "books" ON
8       "books_categories"."book_id" = "books"."id" WHERE "books"."person_id" = ?
9     [{"person_id": 15}]
10  => #<ActiveRecord::Associations::CollectionProxy:0x007fbc4000000000 [
11    #<Category id: 1, name: "Tecnologia", created_at: "2017-03-12 14:02:51", updated_at: "2017-03-12
12    14:02:51">, #<Category id: 2, name: "Programação", created_at: "2017-03-12
13    14:02:51", updated_at: "2017-03-12 14:02:51">, #<Category id: 1, name:
14    "Tecnologia", created_at: "2017-03-12 14:02:51", updated_at: "2017-03-12
15    14:02:51">, #<Category id: 2, name: "Programação", created_at: "2017-03-12
16    14:02:51", updated_at: "2017-03-12 14:02:51">]
```

Só que, como já identificamos no teste, vai trazer as categorias repetidas. Para evitar isso, precisamos pegar os resultados *distintos*, e se alguém conhece SQL por aí, sabe que isso podemos recuperar com uma cláusula DISTINCT, que é o que vamos utilizar enviando através de uma lambda para a associação:

```
1 has_many :categories, -> { distinct }, through: :books
```

Pronto! Agora rodando os testes, vamos constatar que as categorias estão retornando corretamente, sem duplicação.

Essa associação ficou da seguinte forma, representada pelo traço pontilhado:



Pessoa tem várias categorias através de livros

Acelerando as consultas nas associações

Temos alguns jeitos de utilizarmos as associações, sendo que estamos vendo até agora o jeito padrão que já vem com o Rails, onde o ORM cuida de bastante coisas para nós, mas onde também podemos dar uma mãozinha para que as coisas fiquem mais eficientes.

Joins

Vamos imaginar que queremos descobrir as pessoas que tem livros associados. Um jeito de não fazer isso seria assim:

```
1 > Person.select { |person| person.books.count > 0 }
2   Person Load (0.1ms)  SELECT "people".* FROM "people" ORDER BY "people"."name" ASC
3   (0.1ms)  SELECT COUNT(*) FROM "books" WHERE "books"."person_id" = ?  [{"person_id": 4}]
4   (0.1ms)  SELECT COUNT(*) FROM "books" WHERE "books"."person_id" = ?  [{"person_id": 3}]
```

Vejam que eu percorremos toda a tabela de pessoas e estamos fazendo várias outras consultas para pegar cada pessoa e selecionar os livros de cada uma delas. Ficou uma mistura de iteradores Ruby (o método `select`) com código SQL (o `where` dentro de cada bloco). Podemos fazer bem melhor utilizando `joins`:

```
1 Person.joins(:books).distinct
2   Person Load (0.2ms)  SELECT DISTINCT "people".* FROM "people" INNER JOIN
3     "books" ON "books"."person_id" = "people"."id" ORDER BY "people"."name" ASC
4 => #<ActiveRecord::Relation [<Person id: 3, name: "Eustáquio Rangel de
5 Oliveira Jr.", email: "taq@bluefish.com.br", password_digest: ...]
```

Bem melhor, não? Ali o `INNER JOIN` produzido pelo método `joins` já relacionou as duas tabelas, fazendo o filtro de uma vez só no comando SQL, procurando obrigatoriamente pessoas que tem livros. Vejam que utilizamos `distinct` no final, para evitar registros duplicados se a pessoa tiver mais de um livro.



Dica

A partir do Rails 5, também temos o método `left_joins`, que ao invés de um `INNER JOIN`, gera um `LEFT OUTER JOIN`, fazendo o relacionamento se a chave estrangeira existir ou não.

Includes

Temos um problema ali: e se precisarmos saber os títulos dos livros encontrados? O resultado nos trouxe somente uma coleção de objetos tipo `Person`.

Vamos fazer um teste:

```

1 > Person.joins(:books).distinct.map { |person| { person.name => person.books.\
2 pluck(:title) } }
3   Person Load (0.3ms)  SELECT DISTINCT "people".* FROM "people" INNER JOIN
4     "books" ON "books"."person_id" = "people"."id" ORDER BY "people"."name" ASC
5   (0.1ms)  SELECT "books"."title" FROM "books" WHERE "books"."person_id" = ?\n
6   [["person_id", 3]]
7 => [{"Eustáquio Rangel de Oliveira Jr."=>["Conhecendo Ruby", "Conhecendo o G\\
8 it"]}]

```

Reparam que para cada pessoa, foi executada outra consulta para recuperar os livros (e também o nome da pessoa, como chave da Hash) e retornar somente o título. O que ocorre é o método `joins` estabelece o relacionamento, mas não deixa disponível os dados da outra associação, não faz `eager loading`⁹⁵. Para remediar isso, podemos utilizar o método `includes`.

Antes de aplicarmos o método `includes` na nossa consulta, vale mencionar que ele também pode ser utilizado, além do `eager loading`, como um LEFT OUTER JOIN:

```

1 Person.includes(:books).distinct.map { |person| { person.name => person.books.\
2 .pluck(:title) } }
3   Person Load (0.1ms)  SELECT DISTINCT "people".* FROM "people" ORDER BY "peo\\
4 ple"."name" ASC
5   Book Load (0.4ms)  SELECT "books".* FROM "books" WHERE "books"."person_id" \\\
6 IN (4, 3)
7 => [{"Ana Carolina"=>[]}, {"Eustáquio Rangel de Oliveira Jr."=>["Conhecendo \\\\
8 Ruby", "Conhecendo o Git"]}]

```

Podemos ver que para as pessoas que não tinham livros, foi retornado uma coleção vazia e que ao invés de várias consultas aos livros (o que ocorreria se tivéssemos várias pessoas com livros) foi feita uma, especificando os ids de todas as pessoas.

Juntando joins e includes

Mas vamos voltar para resolver nosso problema de performance. Se pensarmos que `joins` faz a associação com a outra tabela e `includes` faz o `eager loading`, é só juntarmos os dois:

⁹⁵ http://guides.rubyonrails.org/active_record_querying.html

```

1 > Person.joins(:books).includes(:books).distinct.map { |person| { person.name \
2 => person.books.pluck(:title) } }
3   SQL (0.2ms)  SELECT DISTINCT "people"."id" AS t0_r0, "people"."name" AS t0_\
4 r1,
5   "people"."email" AS t0_r2, "people"."password_digest" AS t0_r3,
6   "people"."born_at" AS t0_r4, "people"."admin" AS t0_r5, "people"."created_a\
7 t"
8   AS t0_r6, "people"."updated_at" AS t0_r7, "books"."id" AS t1_r0,
9   "books"."title" AS t1_r1, "books"."published_at" AS t1_r2, "books"."text" AS
10  t1_r3, "books"."value" AS t1_r4, "books"."person_id" AS t1_r5,
11  "books"."created_at" AS t1_r6, "books"."updated_at" AS t1_r7, "books"."stoc\
12 k"
13  AS t1_r8, "books"."lock_version" AS t1_r9 FROM "people" INNER JOIN "books" \
14 ON
15  "books"."person_id" = "people"."id" ORDER BY "people"."name" ASC
16 => [{"Eustáquio Rangel de Oliveira Jr."=>["Conhecendo Ruby", "Conhecendo o G\
17 it"]}]

```

Agora sim! Temos uma consulta única de onde são extraídas todas as informações que precisamos. Vejam como o ORM altera o nome dos campos e sabe acessar cada um de maneira transparente.

Pluck versus select

Vimos que utilizamos o método `pluck`. Esse método extrai somente os atributos que indicamos, **sem retornar um objeto da associação**. Assim:

```

1 > Person.pluck(:name)
2   (0.3ms)  SELECT "people"."name" FROM "people" ORDER BY "people"."name" ASC
3 => ["Ana Carolina", "Eustáquio Rangel de Oliveira Jr."]

```

Podemos indicar mais de um atributo:

```

1 > Person.pluck(:name, :email)
2   (0.2ms)  SELECT "people"."name", "people"."email" FROM "people" ORDER BY "\\
3 people"."name" ASC
4 => [["Ana Carolina", "carol@bluefish.com.br"], ["Eustáquio Rangel de Oliveira\
5 a Jr.", "taq@bluefish.com.br"]]

```

Vejam que são retornados **POROS** (Plain Old Ruby Objects), no formato de **Arrays**.

O método `select`, por sua vez, retorna um objeto **com apenas os atributos selecionados preenchidos**, ou seja, não vamos conseguir acessar os outros. Assim, isso funciona:

```
1 > Person.select(:name, :email)
2   Person Load (0.1ms)  SELECT "people"."name", "people"."email" FROM "people"
3   ORDER BY "people"."name" ASC => #<ActiveRecord::Relation [<Person id: nil,
4     name: "Ana Carolina", email: "carol@bluefish.com.br">, <Person id: nil, na\
5     me:
6       "Eustáquio Rangel de Oliveira Jr.", email: "taq@bluefish.com.br">]>
7
8 > Person.select(:name, :email).first.name
9   Person Load (0.3ms)  SELECT "people"."name", "people"."email" FROM "people"
10  ORDER BY "people"."name" ASC LIMIT ?  [{"LIMIT": 1}]
11 => "Ana Carolina"
12
13 > Person.select(:name, :email).first.email
14  Person Load (0.2ms)  SELECT "people"."name", "people"."email" FROM "people"
15  ORDER BY "people"."name" ASC LIMIT ?  [{"LIMIT": 1}]
16 => "carol@bluefish.com.br"
```

Já isso, não:

```
1 > Person.select(:name, :email).first.admin
2   Person Load (0.1ms)  SELECT "people"."name", "people"."email" FROM "people"
3   ORDER BY "people"."name" ASC LIMIT ?  [{"LIMIT": 1}]
4 ActiveModel::MissingAttributeError: missing attribute: admin
```

Criando um outro tipo de associação um-para-um

Podemos utilizar um outro tipo de associação um-para-um, com uma semântica diferente, utilizando `has_one`. Enquanto que em `belongs_to` a *foreign key* estava no modelo que especificava a relação, em `has_one` está no **outro** modelo.

Podemos, por exemplo, dizer que as pessoas da nossa aplicação tem cada uma, uma imagem. Para isso, vamos aprender como fazer *upload* de arquivos.

Upload de arquivos

Existem algumas *gems*, como o Paperclip⁹⁶ e o CarrierWave⁹⁷, que fazem *upload* de arquivos, até com recursos específicos para imagens, mas vamos ver como o *upload* é feito como forma de aprender para o caso de não quisermos utilizar nenhuma *gem* por alguma razão.

Criando o modelo de imagem

Vamos criar um modelo para guardar informações da nossa imagem, tanto em atributos no banco de dados como em atributos virtuais, tais como um título opcional, o nome da imagem, que vai refletir o seu `id`, e o diretório onde é armazenada, que vai ser:

```
1 #Rails.root}/public/images/people/
```



É importante verificar como o seu servidor lida com esse tipo de conteúdo. Em ambiente de desenvolvimento, temos a configuração `config.serve_static_assets`, que lida com a entrega de conteúdo estático armazenado em `/public`. Quando está configurado como `true`, é adicionado um *middleware* para verificar as requisições do navegador, e insere um pouco de *overhead*. Também é interessante o fato de servir conteúdo estático de outro domínio.

Vamos convencionar que nossas imagens estarão no formato JPEG⁹⁸, e sempre salvar/converter para esse formato. As *gems* relacionadas acima tem recursos específicos para armazenar inclusive o formato de imagem enviado, mas novamente, vamos nos virar com os recursos padrões que vem com o Rails.

Gerando agora apenas o nosso **modelo**, chamado `Image`, utilizando um título e uma referência para a pessoa da imagem, reparando que nesse caso utilizei explicitamente `person_id:integer` ao invés de `person:references` como fizemos anteriormente:

```
1 $ rails g model Image title:string person_id:integer
2       invoke  active_record
3       create    db/migrate/20170313222139_create_images.rb
4       create    app/models/image.rb
5       invoke  test_unit
6       ...
```

Rodando a `migration`:

⁹⁶<https://github.com/thoughtbot/paperclip>

⁹⁷<https://github.com/carrierwaveuploader/carrierwave>

⁹⁸http://pt.wikipedia.org/wiki/Joint_Photographic_Experts_Group

```
1 $ rails db:migrate
2 == 20170313222139 CreateImages: migrating =====\\
3 ==
4 -- create_table(:images)
5   -> 0.0011s
6 == 20170313222139 CreateImages: migrated (0.0011s) =====\\
7 ==
```

Agora vamos adaptar o nossos modelos das pessoas e das imagens. No modelo Person:

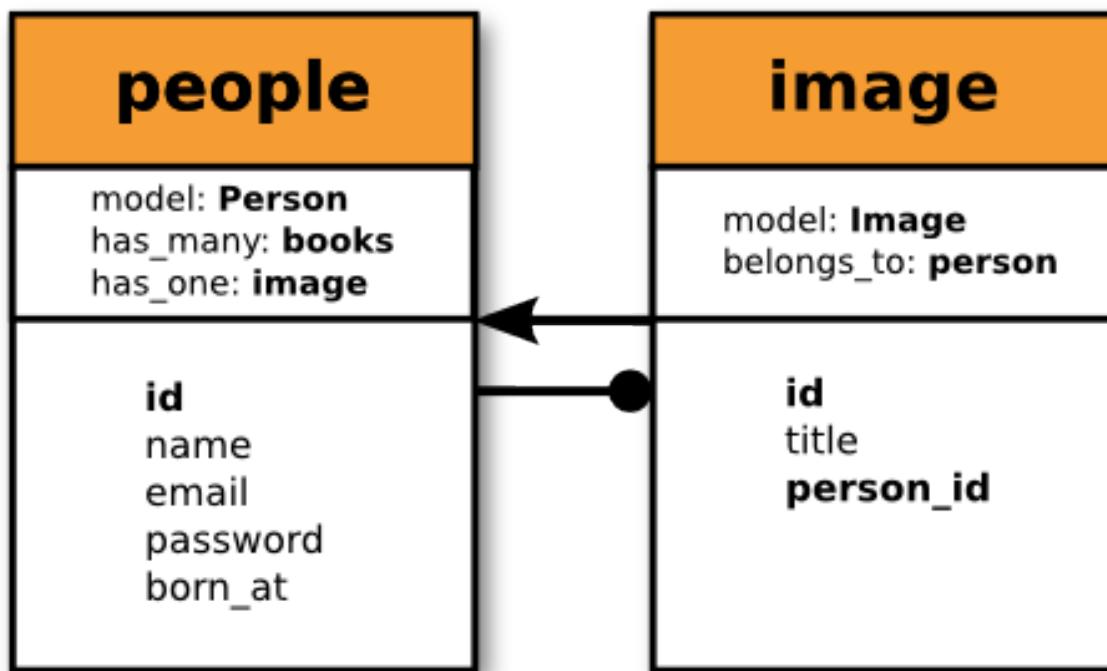
```
1 has_one :image, dependent: :destroy
```

E no modelo Image:

```
1 class Image < ActiveRecord::Base
2   after_save      :update_file
3   after_destroy   :delete_file
4
5   attr_accessor :data_stream
6   belongs_to :person
7
8   def filename
9     "#{id}.jpg"
10  end
11
12  def path
13    '/images/people/'
14  end
15
16  def to_s
17    "#{path}#{filename}"
18  end
19
20  def full_path
21    "#{Rails.root}/public/#{path}/#{filename}"
22  end
23
24  def update_file
25    return false if data_stream.blank?
26
27    data = data_stream.read
28    return false if data.empty?
29
30    delete_file
31    check_path
```

```
32     write_data(data)
33   end
34
35 private
36
37   def delete_file
38     File.unlink(full_path) if File.exist?(full_path)
39   end
40
41   def check_path
42     dir = File.expand_path(File.dirname(full_path))
43     FileUtils.mkpath(dir) unless Dir.exist?(dir)
44   end
45
46   def write_data(data)
47     File.open(full_path, 'wb') do |file|
48       file.write(data)
49     end
50     check_size
51   end
52
53   def check_size
54     File.size(full_path) > 0
55   end
56 end
```

Ficando assim:



Pessoas e imagens

Adaptando as views e controllers, com imagem

Agora temos que adaptar alguma *view* para enviar o arquivo de imagem que queremos, junto com os dados do nosso modelo de pessoa, tanto no momento que é criada como editada.

Vamos personalizar o formulário da *partial _form.html.erb*, para inserir um elemento novo para *upload* da imagem e permitir o envio de dados utilizando *multipart*⁹⁹, sem isso não seria possível enviar os dados binários da imagem pelo formulário:

```

1 <%= form_with(model: person, local: true, html: { multipart: true }) do |form|
2 | %>
3 ...
4 <div class="field">
5   <%= file_field_tag :data_stream %>
6 </div>
7 ...
  
```

E agora o controlador, para receber o fluxo binário de dados do arquivo enviado, gerando uma imagem nova:

⁹⁹<http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.2>

```

1 class PeopleController < ApplicationController
2 ...
3   after_action :save_image, only: [:create, :update]
4 ...
5
6   private
7
8   def save_image
9     return unless params[:data_stream].present?
10    @image = (@person.image || Image.new(title: @person.name, person_id: @per\
11 son.id))
12    @image.data_stream = params[:data_stream]
13    @person.image = @image if @image.save
14  end

```

Agora nosso controlador já sabe como armazenar uma imagem! Se enviarmos uma imagem através do navegador, podemos verificar no console algo como:

```

1 > Image.first
2 Image Load (0.1ms)  SELECT "images".* FROM "images" ORDER BY "images"."id"\\
3 ASC LIMIT ?  [["LIMIT", 1]]
4 => #<Image id: 1, title: "Eustáquio Rangel de Oliveira Jr.", person_id: 3,
5 created_at: "2017-03-13 23:05:01", updated_at: "2017-03-13 23:05:01">

```

E no diretório public/images/people:

```

1 $ ls public/images/people/
2 total 92K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 3 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 83K 1.jpg

```

Redimensionando as imagens

Para redimensionar as imagens, podemos criar atributos virtuais de largura (`width`) e altura (`height`), verificar se não são nulos, e redimensionar utilizando a gem `MiniMagick`¹⁰⁰. Para instalar essa gem, temos que ter certeza que os pacotes do `ImageMagick`¹⁰¹ estão instalados no sistema operacional:

¹⁰⁰ <https://github.com/minimagick/minimagick>

¹⁰¹ <http://www.imagemagick.org/>

```
1 $ sudo apt-get install imagemagick
2 [sudo] password for taq:
3 Lendo listas de pacotes... Pronto
4 Construindo árvore de dependências
5 Lendo informação de estado... Pronto
6 imagemagick já é a versão mais nova.
7 0 pacotes atualizados, 0 pacotes novos instalados, 0 a serem removidos e 18 n\ 
8 ão atualizados.
```

Agora vamos alterar nosso `Gemfile` para incluir a gem `Minimagick` e rodar `bundle install`:

```
1 gem 'mini_magick'
2
3 $ bundle install
4 ...
5 Using mini_magick 4.5.1
6 Your bundle is complete!
```

No computador que estou escrevendo, ambos, tanto a `ImageMagick` como a `mini_magick`, já estavam instaladas. ;-)



Dica

Reiniciem o servidor web após alterar o `Gemfile`!

Agora vamos alterar nosso modelo de imagem para incluir os atributos virtuais, utilizando a `MiniMagick` para redimensionar a imagem, inclusive levando em conta a proporção:

```
1 class Image < ActiveRecord::Base
2   after_save      :update_file
3   after_destroy   :delete_file
4
5   attr_accessor :data_stream, :width, :height
6   ...
7
8   def update_file
9     ...
10    resize
11  end
12
13  ...
14  def resize
15    return unless @width || @height
16    image = MiniMagick::Image.new(full_path)
17    image.resize "#{@width}x#{@height}"
18  end
```

E indicar em nosso controlador o tamanho desejado da imagem, no exemplo abaixo, especificando somente a altura:

```
1 def save_image
2   return unless params[:data_stream].present?
3   @image = (@person.image || Image.new(title: @person.name, person_id: @perso\
4 n.id))
5   @image.data_stream = params[:data_stream]
6   @image.height = 200
7   @person.image = @image if @image.save
8 end
```

Como já temos uma imagem, podemos alterar nosso PersonPresenter para adicionar um método para gerar a tag da imagem, se ela estiver presente:

```
1 def image
2   return '' unless @person.image
3   helpers.image_tag(@person.image.to_s).html_safe
4 end
```

E alterar a view app/views/people/show.html.erb para utilizar o atributo novo:

```
1 <p>
2   <%= @person.image %>
3 </p>
```

Que gera uma tela como essa:

Interface de administração

Pessoas

Nome: Eustáquio Rangel de Oliveira Jr.

E-mail: taq@bluefish.com.br

Senha: *****

Data de nascimento: 10/04/1971

Administrador: Sim



[Editar](#) | [Voltar](#)

Desenvolvido no ebook "Conhecendo Rails"

Presenter com imagem

Criando uma associação polimórfica

Temos um pequeno problema aqui: na relação de pessoa com imagem, especificamos o `id` da pessoa no modelo da imagem, associando com `has_one`, porém agora precisamos cadastrar imagens também para os livros!

Não podemos criar um atributo `book_id` no modelo de imagem ... até podemos, mas ia ser uma modelagem meio estranha. O que precisamos é de uma **associação polimórfica**, onde o modelo da imagem saiba qual outro modelo o qual ele efetua a associação.

A primeira coisa que precisamos fazer é remover o atributo `person_id` do nosso modelo de imagem, pois agora ele não vai mais responder somente para um tipo de associação com apenas um outro modelo, vai ser polimórfico, pertencendo à um ou mais modelos. Logo após removermos o atributo, precisamos criar outros com o nome da nossa associação polimórfica.

Vamos criar o termo `imageable` para referenciar a associação, mas poderia ser qualquer outro (que fizesse sentido, lógico, não vão chamar o negócio de “pepino”, por exemplo). Precisamos incluir os campos `<nome da associação>_id`, como um inteiro, e `<nome da associação>_type`, como `String`, e remover o campo `person_id`, que nessa relação já não faz mais sentido.

Vamos gerar uma `migration` vazia:

```
1 $ rails g migration AddPolymorphismToImage
2       invoke  active_record
3       create    db/migrate/20140416122559_add_polymorphism_to_image.rb
```

Incluir o conteúdo necessário na `migration`, salvando os `ids` das pessoas correntes, removendo a coluna `person_id`, criando as novas coluns `imageable_id` como um inteiro e `imageable_type` como uma `String`, resetando as informações das colunas do modelo e preenchendo as colunas novas com os dados antigos (se não quisermos salvar nada, podemos ignorar toda essa operação salva-e-restaura):

```
1 class AddPolymorphismToImage < ActiveRecord::Migration[5.0]
2   def up
3     old_imgs = Image.all.inject({}) { |memo, img|
4       memo[img.id] = img.person_id
5     }
6   end
7
8   remove_column :images, :person_id
9   add_column :images, :imageable_id, :integer
10  add_column :images, :imageable_type, :string
11  add_index  :images, [:imageable_id, :imageable_type]
12
13  Image.reset_column_information
14
15  old_imgs.each { |id, person_id|
```

```

16      next if person_id.blank?
17      puts "Updating image #{id} with person #{person_id}"
18      Image.find(id).update_attributes(imageable_id: person_id, imageable_type\ 
19 e: 'Person')
20    end
21  end
22
23  def down
24    add_column :images, :person_id
25    remove_column :images, :imageable_id
26    remove_column :images, :imageable_type
27    remove_index :images, column: [:imageable_id, :imageable_type]
28  end
29 end

```

Remover a condição que uma imagem pertence à uma pessoa, removendo no modelo de Image a seguinte linha:

```
1 belongs_to :person
```

E rodar a *migration*:

```

1 $ rails db:migrate
2 == 20170315214710 AddPolymorphismToImage: migrating =====\\
3 ==
4 -- add_column(:images, :imageable_id, :integer)
5   -> 0.0009s
6 -- add_column(:images, :imageable_type, :string)
7   -> 0.0004s
8 Updating image 1 with person 1
9 == 20170315214710 AddPolymorphismToImage: migrated (0.0066s) =====\\
10 ==

```



Dica

Após uma mudança na tabela, se necessário na própria *migration* utilizar as novas colunas, temos que executar

```
1 <Modelo>.reset_column_information
```

para acionar as mudanças.

Agora podemos adaptar os modelos de pessoa e imagem para refletirem a associação polimórfica, primeiro no modelo Person:

```

1 class Person < ActiveRecord::Base
2 ...
3 has_one :image, dependent: :destroy, as: :imageable
4 ...

```

Depois no modelo Image:

```

1 class Image < ActiveRecord::Base
2 ...
3 belongs_to :imageable, polymorphic: true
4 ...

```

E agora em Book, já que foi a causa de criarmos a relação polimórfica:

```

1 class Book < ActiveRecord::Base
2 ...
3 has_one :image, dependent: :destroy, as: :imageable
4 ...

```



Precisamos remover `person_id` das fixtures de `Image`, do contrário nossos testes vão falhar.

Com isso temos os seguintes comportamentos, testando agora no *console*:

```

1 > p = Person.last
2   Person Load (0.3ms)  SELECT  "people".* FROM "people" ORDER BY "people"."name"
3 DESC LIMIT ?  [["LIMIT", 1]]
4 => #<Person id: 11, name: "Eustáquio Rangel de Oliveira Jr.", email:
5 "taq@bluefish.com.br", password_digest:
6 "$2a$10$TXWwkRCaQSPIjZhLxdcVHu3SG820c6BVnZ.cjbm14in...", born_at: "1970-01-01",
7 admin: true, created_at: "2017-03-15 22:04:48", updated_at: "2017-03-15 22:04:48">
8
9
10 > puts p.image
11   Image Load (0.2ms)  SELECT  "images".* FROM "images" WHERE
12 "images"."imageable_id" = ? AND "images"."imageable_type" = ? LIMIT ?
13   [[["imageable_id", 11], ["imageable_type", "Person"], ["LIMIT", 1]]]
14   /images/people/3.jpg
15 => nil
16
17 > p.image.imageable_id
18
19

```

```

20 => 11
21
22 > p.image.imageable_type
23 => "Person"
24
25 > p.image.imageable.name
26   Person Load (0.2ms)  SELECT  "people".* FROM "people" WHERE "people"."id" =\
27 ?
28   ORDER BY "people"."name" ASC LIMIT ?  [["id", 11], ["LIMIT", 1]]
29 => "Eustáquio Rangel de Oliveira Jr."

```

Até agora, exibe quase o comportamento anterior, mas já com relação polimórfica, a ser mostrado de maneira mais clara a seguir, quando vamos alterar o nosso modelo de livro para utilizar associações polimórficas.

Antes de adaptar o modelo e o controlador de livro, devemos alterar o *path* da imagem, que está *hard coded* como `people` no método `path` do modelo `Image`, mudando de:

```

1 def path
2   '/images/people/'
3 end

```

para

```

1 def path
2   "/images/#{imageable_type.pluralize.underscore}/"
3 end

```

E alterar o formulário da *view* de `Book`, de maneira similar ao de `Person`:

```

1   <%= form_with(model: book, local: true, html: { multipart: true }) do |fo\
2 rm| %>
3 ...
4   <div class="field">
5     <%= file_field_tag :data_stream %>
6   </div>

```

E finalmente o controlador de `Book`, para receber as imagens, utilizando o *callback after_action*:

```
1 class BooksController < ApplicationController
2 ...
3   after_action :save_image, only: [:create, :update]
4 ...
5 private
6
7   def save_image
8     return if !params[:data_stream]
9
10    @image = @book.image ? @book.image : Image.new(title: @book.title)
11    @image.data_stream = params[:data_stream]
12    @image.height = 200
13    @book.image = @image if @image.save
14  end
15 end
```

O problema é que aparentemente está ok, mas vamos ter problemas em relação à isso:

1. O atributo `imageable_type` é determinado apenas no momento em que a imagem é associada ao controlador, e não gravada. No momento da gravação esse campo vai estar nulo e vamos ter problemas, com comportamento não esperado. Experimentem para ver o controlador explodindo. :-p
2. São executadas duas operações no banco de dados, uma de `INSERT` (no `save`) e outra de `UPDATE`, no momento em que a imagem é associada ao *container*.

Podemos corrigir e melhorar esse código da seguinte forma:

```
1 def save_image
2   return if !params[:data_stream]
3
4   @image = @book.image ? @book.image : Image.new(title: @book.title, imageable\
5   e_id: @book.id, imageable_type: controller_name.singularize.camelize)
6   @image.data_stream = params[:data_stream]
7   @image.height = 200
8   @image.save
9 end
```

Utilizamos `controller_name` e o `id` da variável de instância `@book` para já gravar somente a imagem, com todas as informações corretas.

Mas espera lá! Vamos ter que fazer a mesma coisa com o controlador das pessoas, e seria outra violação do princípio DRY ... e lá vamos nós consertar isso de novo!

Criando um módulo compartilhado

Vamos criar um módulo compartilhado onde podemos inserir as funcionalidades que queremos compartilhar. Vamos chamar esse módulo de `ImageSaver`, e ele vai ficar em um arquivo chamado `image_saver.rb` (**atenção no nome**: tem que seguir as convenções de sublinhado/*camel case* entre arquivos e classes), utilizando o recurso `concerns` implementados a partir do Rails 4, que nos dá diretórios chamados `concerns` nos diretórios `app/models` e `app/controllers`, onde podemos armazenar esses tipos de módulos para serem utilizados sem precisar fazer alguns procedimentos que tínhamos que fazer antes.

O conteúdo do arquivo do `concern` em `app/controllers/concerns/image_saver.rb` é:

```
1 module ImageSaver
2   extend ActiveSupport::Concern
3
4   included do
5     after_action :save_image, only: [:create, :update]
6   end
7
8   private
9
10  def save_image
11    stream = params[:data_stream]
12    return unless stream.present?
13    save(image, stream)
14  end
15
16  def image
17    name = controller_name.singularize
18    ref = instance_variable_get("@#{name}")
19    img = ref.image
20    return img if img
21
22    Image.new(title: image_title_ref,
23              imageable_id: ref.id,
24              imageable_type: name.camelize)
25  end
26
27  def save(image, stream)
28    image           = image
29    image.data_stream = stream
30    image.height    = 200
31    image.save
32  end
33 end
```

E alteramos nossos controladores de Person e Book, retirando os `after_action` e os métodos privados `save_image`, incluindo o módulo e fornecendo um `payback` em cada modelo indicando o nome do atributo que vai ser utilizado como o título da imagem, que são métodos estáticos e feinhos:

```
1 class PeopleController < ApplicationController
2   include ImageSaver
3   ...
4   def image_title_ref
5     'Foto da pessoa'
6   end
7   ...
```

```
1 class BooksController < ApplicationController
2   include ImageSaver
3   ...
4   def image_title_ref
5     'Capa do livro'
6   end
7   ...
```

Agora ambos os controladores, e quaisquer outros que precisarem, podem compartilhar a funcionalidade de salvar a imagem.

Mas tem um porém.

Movendo a lógica para o modelo

E aquele papo de que controladores devem ser burros, parece que esses ai estão bem espertinhos ... pois é, mas serviram para nos mostrar como fazer módulos compartilhados para dividirmos o que sobrar de inteligência para eles!

Mas agora vamos (não me xinguem, é para efeitos didáticos!) mandar essa inteligência para os modelos, e simplificar mais ainda esse processo.

Vamos apagar o arquivo `image_saver.rb` dos `concerns` dos controladores, o `include` e método de `payback` dos controladores e mudar o nosso módulo para os `concerns` dos modelos, onde vai ficar em `app/models/concerns/image_saver.rb`:

```
1 module ImageSaver
2   attr_accessor :data_stream
3   attr_writer   :image_title
4
5   def self.included(base)
6     base.after_save :save_image
7   end
8
9   def image_title
10    @image_title || (image.present? && image.title.present? ? image.title : '\\
11 ')
12  end
13
14 private
15
16 def save_image
17  return unless data_stream.present?
18  configure_image(find_image).save
19 end
20
21 def find_image
22  return image if image
23  Image.new(imageable_id: id, imageable_type: self.class.to_s)
24 end
25
26 def configure_image(image)
27  image.title      = image_title
28  image.data_stream = data_stream
29  image.height     = 200
30  image
31 end
32 end
```

Após isso, incluímos o módulo nos modelos que desejamos (Person e Book):

```
1 class Person < ActiveRecord::Base
2   include ImageSaver
3   ...
```

```
1 class Book < ActiveRecord::Base
2   include ImageSaver
3   ...
```

Alteramos nossos controladores para permitir o envio dos atributos `image_title` e `data_stream`:

```

1 # people controller
2 def person_params
3   params.require(:person).permit(:name, :email, :plain_password, :born_at, :i\
4 mage_title, :data_stream)
5 end

1 # books controller
2 def book_params
3   params.require(:book).permit(:title, :published_at, :text, :value, :person_\
4 id, :image_title, :data_stream, category_ids: [])
5 end

```

E alteramos as *views* de ambos para:

```

1 <div class="field">
2   <%= form.label :image_title %>
3   <%= form.text_field :image_title %>
4   <%= form.file_field :data_stream %>
5 </div>

```

Como podemos ver, é o mesmo código em ambas as *views* (reparem que trocamos `file_field` para `file_field_tag`), e aplicando DRY, vamos criar uma *partial* para armazenar esse código e inserir em ambas as *views*.

Para isso, vamos criar um diretório de *partials* compartilhadas em `app/views/application` (tem um artigo legal sobre isso [aqui¹⁰²](#)) e salvar o código exibido acima em um arquivo nesse diretório, chamado `_image_fields.html.erb`, e renderizar em nossas *views*, passando a variável `f` (o *handle* do formulário) como uma variável local da *partial*, utilizando:

```
1 <%= render 'image_fields', form: form %>
```

Conteúdo de `app/views/application/_image_fields.html.erb`:

```

1 <div class="field">
2   <%= form.label :image_title %>
3   <%= form.text_field :image_title %>
4   <%= form.file_field :data_stream %>
5 </div>

```



Dica

Arquivos de *partials* devem começar o nome com sublinhado (`_`).

Isso permite que, se precisarmos alterar os formulários para incluir uma frase como “(*deixe em branco se não desejar alterar*)”, podemos alterar somente a *partial*, que refletirá nos outros formulários:

¹⁰² <https://robots.thoughtbot.com/directory-for-shared-partials-in-rails>

```
1 <div class="field">
2   <%= form.label :image_title %>
3   <%= form.text_field :image_title %>
4   <%= form.file_field :data_stream %> <em>(deixe vazio para não alterar)</em>
5 </div>
```

Agora é hora de criar a interface pública da nossa loja.

Interface pública da loja

Já temos nosso controlador pub criado, para onde o visitante da nosso site é direcionado, na página inicial, para a ação index. Vamos dar uma olhada no arquivo de *layout* que foi utilizado:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Bookstore</title>
5          <%= csrf_meta_tags %>
6          <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-\>
7          track': 'reload' %>
8          <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reloa\>
9          d' %>
10     </head>
11     <body>
12         <header>
13             <h1>Bookstore - A maior livraria virtual (depois da Amazon)</h1>
14         </header>
15         <nav>
16             <ul>
17                 <li><%= link_to 'Inicial', '/' %></li>
18             </ul>
19         </nav>
20         <section id="main">
21             <%= yield %>
22         </section>
23         <footer>
24             Desenvolvido no ebook "Conhecendo Rails"
25         </footer>
26     </body>
27 </html>
```

Podemos notar algumas coisas interessantes:

- Utilizamos o *helper* `stylesheet_link_tag` para incluir a referência para a folha de estilo chamada `application.css`, que está no diretório `app/assets/stylesheets` (nas versões anteriores do Rails, ficava em `public/stylesheets/`);
- Utilizamos o *helper* `javascript_include_tag` para incluir a referência para o arquivo com código JavaScript chamado `application.js`, que está no diretório `app/assets/javascripts` (nas versões anteriores do Rails, ficava em `public/javascripts/`);

- Utilizamos `csrf_meta_tags`, que protege a aplicação contra *cross-site request forgery*¹⁰³ e gera `tags` como essas no arquivo HTML:

```
1 <meta content="authenticity_token" name="csrf-param" />
2 <meta content="rmYAhvLd7yaEiz1B4ajjKb+RJ5lc8IH8drPv4/UW3YQ=" name="csrf-token">
3 " />
```

Agora vamos alterar nosso controlador público para carregar nossos livros na ação `index`:

```
1 class PubController < ApplicationController
2   def index
3     @books = Book.all
4   ...
5 end
```

E alterar a view `index.html.erb` para listar nossos livros. Para isso, vamos utilizar o conceito de *partials* que utilizamos para renderizar o campo de envio de arquivo de imagem na seção anterior, mas nesse caso vamos enviar uma coleção de objetos para a *partial*, que vai ser aplicada para cada elemento da coleção.

Para a view `index.html.erb`, vamos ter conteúdo como esse:

```
1 <h1>Nossos produtos</h1>
2
3 <ul id="mini_books">
4   <%= render partial: "mini_book", collection: @books %>
5 </ul>
```

Lembrando que os nomes de *partials* começam com _ (sublinhado), vamos criar o arquivo `_mini_book.html.erb` em `app/views/pub/`, com o seguinte conteúdo:



Dica

Podemos ter um atalho ali: o Rails pode descobrir o nome da *partial* direto pelo modelo da coleção, então se utilizássemos:

```
<%= render @books %>
```

e tivéssemos uma *partial* chamada `_book.html.erb`, iria funcionar. Como vamos ter uma página específica para mostrar cada livro, vamos deixar de maneira explícita como fizemos acima.

¹⁰³ [http://en.wikipedia.org/wiki/Cross-site request forgery](http://en.wikipedia.org/wiki/Cross-site_request_forgery)

```
1 <li>
2   <h1><%= mini_book.title %></h1>
3   <h2><%= mini_book.person.name %></h2>
4   <%= mini_book.image ? image_tag(mini_book.image) : '' %>
5   <%= simple_format(mini_book.text) %>
6   <span class='book_value'><%= number_to_currency(mini_book.value) %></span>
7   <%= link_to 'Mais detalhes', pub_book_path(mini_book), class: 'more_book_in\
8 fo' %>
9 </li>
```

Reparam que eu utilizei o método `pub_book_path` ali acima. Ele é gerado por uma rota que configuramos assim:

```
1 get 'livro/:id' => 'pub#book', as: 'pub_book'
```

Ali estabelecemos que:

1. O método utilizado vai ser GET.
2. Quando a aplicação encontrar um GET apontando para uma URL como `livro/1`, vai redirecionar para o controlador `pub`, na ação `books`, enviando o `id` para os params.
3. Demos um “apelido” para a rota criada, que foi `pub_book`. Isso vai criar os métodos `pub_book_path` (path relativo) e `pub_book_url` (URL completa).

Isso vai gerar algo como na figura abaixo:

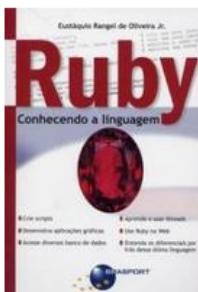
Bookstore - A maior livraria virtual (depois da Amazon)

- [Inicial](#)

Nossos produtos

- **Ruby - Conhecendo a Linguagem**

Eustáquio Rangel

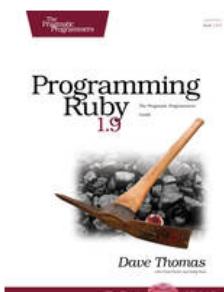


Tinha, mas acabou.

\$40.00 [Mais detalhes](#)

- **Programming Ruby**

Dave Thomas



A Bíblia de Ruby!

\$80.00 [Mais detalhes](#)

Desenvolvido no ebook "Conhecendo Rails"

Imagen da miniatura dos livros na página inicial

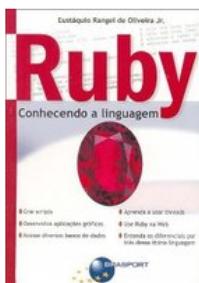
Strings seguras para HTML

Reparam no método `simple_format`: ele converte o conteúdo do campo texto em HTML formatado bem simples, onde troca as quebras de linhas (`\n` gravados no banco de dados) para parágrafos HTML (elementos `p`), o que gera um parágrafo com o texto recebido a cada quebra de linha.

Aí podemos ter alguma situação em que não queremos o elemento `p` com o texto dentro, e sim simples elementos `br`, somente para quebrar a linha. Ao invés de `simple_format`, podemos utilizar:

```
1 <%= mini_book.text.gsub(/\n/, "<br/>") %>
```

Ou seja, isso deve trocar todas as quebras de linha na `String` pelos elementos `br`, mas, olhando no navegador:



Meu livro! Tinha, mas acabou. :-)

Na biblioteca da Fatec tem um monte! **R\$ 40,00**

Desenvolvido no ebook "Conhecendo Rails"

Elementos BR impressos na página?

E olhando o código-fonte no navegador:

```
1 Meu livro! Tinha, mas acabou. :-)
2 &lt;br/&gt;
3 &lt;br/&gt;Na biblioteca da Fatec tem um monte!
```

O que ocorre é que o Rails, por medida de segurança, converte qualquer caractere mais “diferente” dentro da `String` retornada para a sua representação como uma entidade HTML, prevenindo que sejam executados, por exemplo, código HTML dentro do valor mostrado no navegador. Se realmente quisermos mostrar o que tem ali no atributo texto do livro, podemos, como forma de garantir para o Rails que está tudo ok, utilizar o método `html_safe` na `String`:

```
1 <%= mini_book.text.gsub(/\n/, "<br/>").html_safe %>
```

O que deve dar agora o retorno esperado. Mas é melhor ficamos mesmo com o `simple_format` para o que precisamos.

Se inserirmos algum CSS no arquivo `app/assets/stylesheets/pub.scss`, vamos ter algo como na figura abaixo (talvez vá ser necessário reiniciar o servidor em modo de desenvolvimento). O CSS é o utilizado abaixo, só não contem para a moça do *front-end* que eu fiz essa barbaridade (ei, eu fiz rapidinho ;-):

```
1 body {
2     margin: 0;
3     font-size: 14px;
4 }
5
6 header {
7     background: maroon;
8     color: white;
9     padding: 1em;
10}
11
12 nav, footer {
13     background: black;
14     color: white;
15
16     ul {
17         list-style-type: none;
18         margin: auto auto auto 1em;
19         padding: 0;
20
21         li {
22             padding: 1em 1em 1em 0;
23             display: table-cell;
24
25             a, a:visited {
26                 color: white;
27                 text-decoration: none;
28                 font-weight: bold;
29             }
30         }
31     }
32 }
33
34 section#main {
35     margin: auto 2em;
36 }
37
38 ul#mini_books {
39     list-style-type: none;
40     padding: 0;
41
42     li {
43         display: table-cell;
44         padding-right: 7em;
45
46         img {
```

```
47         height: 150px;
48     }
49 }
50
51 h1 {
52     color: maroon;
53     margin-bottom: 0;
54 }
55
56 h2 {
57     margin-top: 0;
58     font-size: 1em;
59 }
60 }
61
62 a.more_book_info, a.buy_book {
63     background: red;
64     color: white;
65     font-weight: bold;
66     border-radius: 15px;
67     padding: 0.5em 1em;
68     margin-left: 3em;
69     text-decoration: none;
70
71     :visited {
72         color: white;
73     }
74 }
75
76 p.sold_out {
77     color: red;
78     font-weight: bold;
79 }
80
81 span.book_value {
82     color: green;
83     font-weight: bold;
84     font-size: 1.5em;
85 }
86
87 a.buy_book {
88     display: block;
89     margin: 0;
90     margin: 1em 0;
91     width: 5em;
92 }
```

```

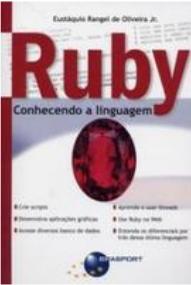
93
94 footer {
95   padding: 1em;
96 }
97
98 section#book_detail {
99   img {
100     height: 200px;
101   }
102 }
```

Bookstore - A maior livraria virtual (depois da Amazon)

Início

Nossos produtos

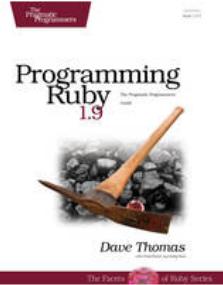
Ruby - Conhecendo a Linguagem
Eustáquio Rangel



Tinha, mas acabou.

\$40.00 [Mais detalhes](#)

Programming Ruby
Dave Thomas



A Bíblia de Ruby!

\$80.00 [Mais detalhes](#)

Desenvolvido no ebook "Conhecendo Rails"

Imagen da miniatura dos livros na página inicial com CSS básico

Agora vamos criar a ação para mostrar os detalhes de um determinado livro, criando um método book no controlador pub, já criamos a rota anteriormente:

```

1 def book
2   @book = Book.find(params[:id])
3 end
```

A view app/views/pub/book.html.erb:

```

1 <section id='book_detail'>
2   <h1><%= @book.title %></h1>
3   <h2><%= link_to @book.person.name, pub_author_path(@book.person) %></h2>
4   <%= image_tag(@book.image) %>
5   <%= simple_format(@book.text) %>
6   <span class='book_value'><%= number_to_currency(@book.value) %></span>
7   <br/>
8   <%= link_to 'Comprar', buy_path(@book), class: 'buy_book' %>
9 </section>

```

E as rotas necessárias, uma para mostrar mais livros do autor, e outra para adicionar o livro no carrinho de compras:

```

1 get 'autor/:id' => 'pub#author', as: 'pub_author'
2 get 'comprar/:id' => 'pub#buy', as: 'buy'

```

Clicando em um livro, vamos ter algo como:

Bookstore - A maior livraria virtual (depois da Amazon)

Início

Ruby - Conhecendo a Linguagem

Eustáquio Rangel

Eustáquio Rangel de Oliveira Jr.
Ruby
Conhecendo a linguagem

Tinha, mas acabou.

\$40.00

Comprar

Desenvolvido no ebook "Conhecendo Rails"

Detalhe de um livro

Podemos reparar que temos um link para ver outros livros do autor. Já definimos a rota acima, agora vamos fazer o método `author` no controlador público, e a `view` correspondente.

Vamos inserir no controlador:

```

1 def author
2   @author = Person.find(params[:id])
3 end

```

E agora a *view* `author.html.erb`:

```

1 <h1>Livros de <%= @author.name %></h1>
2
3 <ul id="mini_books">
4   <%= render partial: 'mini_book', collection: @author.books %>
5 </ul>

```

O que nos resulta em algo como:

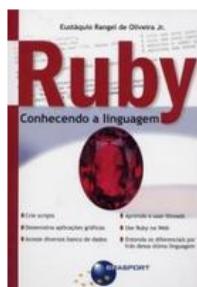
Bookstore - A maior livraria virtual (depois da Amazon)

Início

Livros de Eustáquio Rangel

Ruby - Conhecendo a Linguagem

Eustáquio Rangel



Tinha, mas acabou.

\$40.00

[Mais detalhes](#)

Desenvolvido no ebook "Conhecendo Rails"

Livros de um autor



Reparem em outra coisa: parei de fazer testes. Na parte dos livros não tem praticamente **nada** de testes. :-p

Não, eu repito, **não** parem de testar o seu sistema. Mas para efeitos de melhor didática, agora eu pego um pouco mais leve do que quando estávamos construindo o *scaffold* de `Person`, senão íamos ter páginas e mais páginas de testes aqui, que, espero eu, agora vocês já sabem como faz e entenderam a importância.

Markdown

Se quisermos, podemos renderizar o conteúdo do campo `text` do modelo `Book` como Markdown¹⁰⁴, que é uma linguagem simples de marcação que com poucas regras nos permite fazer algumas formatações bem interessantes. Para utilizar, temos que inserir a gem `redcarpet` no nosso `Gemfile` e rodar o `bundler`:

```
1 gem 'redcarpet'  
2 ...  
3 Installing redcarpet 3.3.4
```

Um jeito legal para utilizar a gem é definir um método nos `helpers`. Vamos chamar o método de `markdown` e inseri-lo em `app/helpers/application_helper.rb`:

```
1 module ApplicationHelper  
2   def markdown(text)  
3     @markdown_renderer ||= Redcarpet::Render::HTML.new(markdown_options)  
4     @redcarpet ||= Redcarpet::Markdown.new(@markdown_renderer, markdown_extensions)  
5   end  
6  
7   @redcarpet.render(text).html_safe  
8  
9   private  
10  
11  def markdown_options  
12    {  
13      filter_html: true,  
14      hard_wrap: true,  
15      link_attributes: { rel: 'nofollow', target: '_blank' },  
16      space_after_headers: true,  
17      fenced_code_blocks: true  
18    }  
19  end  
20  
21  def markdown_extensions  
22    {  
23      autolink: true,  
24      superscript: true,  
25      disable_indented_code_blocks: true  
26    }  
27  end  
28 end
```

Podemos utilizar, por exemplo, na ação `show` de `Book`, dessa forma:

¹⁰⁴<https://daringfireball.net/projects/markdown/>

```

1 <p><strong>Text:</strong></p>
2 <%= markdown(@book.text) %>
```

Onde, se escrevermos no texto do livro algo como:

```

1 Livro sobre a linguagem de programação **Ruby**.
2
3 O livro aborda temas como:
4
5 1. Instalação
6 2. Básico de linguagem
7 3. Tópicos mais avançados
8
9 Para conhecer o *framework* Rails, conheça o outro livro, *"Conhecendo Rails".
```

Vai ser gerado o seguinte código HTML:

```

1 <p>Livro sobre a linguagem de programação <strong>Ruby</strong>.</p>
2
3 <p>O livro aborda temas como:</p>
4
5 <ol>
6 <li>Instalação</li>
7 <li>Básico de linguagem</li>
8 <li>Tópicos mais avançados</li>
9 </ol>
10
11 <p>Para conhecer o <em>framework</em> Rails, conheça o outro livro,
12 <em>"Conhecendo Rails"</em>.</p>
```

Seria uma boa também remover o text da partial app/views/pub/_mini_book.html.erb, senão vamos ter uma miniatura exibindo muito conteúdo, removendo

```
1 <%= simple_format(mini_book.text) %>
```

E alterar a view app/views/pub/book.html.erb para renderizar o Markdown:

```

1 ...
2 <%= image_tag(@book.image) %>
3 <%= markdown(@book.text) %>
4 <span class='book_value'><%= number_to_currency(@book.value) %></span>
5 ...
```

Agora vamos trabalhar no link “*Comprar*”, e para isso, precisamos de um carrinho de compras.

Turbolinks

Vamos aproveitar que montamos a interface pública da loja e verificar um recurso muito legal que começou a ser implementado a partir da versão 4 do Rails, o [Turbolinks¹⁰⁵](#).

Um problema comum hoje em dia quando carregamos uma página no navegador, sendo essa página composta pelo código HTML e geralmente código CSS e JavaScript, onde, apesar que o asset pipeline dá uma boa força juntando tudo e servindo de maneira eficiente, como demonstrado anteriormente, o navegador ainda tem que “parsear” (interpretar, processar) todo esse código recebido de uma forma ou de outra para aplicar no documento corrente.

O que o Turbolinks faz é, para navegadores que suportam o recurso de [pushState¹⁰⁶](#) (adivinhem quem é o único browser que demorou para implementar isso, somente disponível a partir da versão 10?), quando clicado em algum link para alguma página nova, onde o navegador carregaria a página **toda** novamente, é disparada uma requisição com Ajax para pegar o conteúdo da URL requisitada e separado no conteúdo retornado e alterado no documento corrente apenas o conteúdo entre a tag BODY, evitando todo o código JavaScript e CSS que teria que ser interpretado novamente.

Isso diminui consideravelmente o tempo de carregamento do conteúdo requisitado, em algumas estatísticas em até [duas vezes¹⁰⁷](#), e permite algumas experiências interessantes como o uso do *transition cache*, que cria um *cache* de uma determinada URL, e se usuário é redirecionado novamente para a URL, mostra a cópia em *cache* instantaneamente enquanto requisita o conteúdo novamente, atualizando o conteúdo que foi mostrado do *cache*.

Utilização

Essa é uma parte boa: já está habilitado para o uso, automaticamente. O recurso já é incluído no arquivo `app/assets/javascripts/application.js`:

```
1 //= require turbolinks
```

E também no arquivo de *layout* por *default*. No arquivo gerado vem assim:

```
1 <%= stylesheet_link_tag "application", media: "all", "data-turbolinks-track" \ 
2 => true %>
```

Com isso já temos o recurso habilitado.

¹⁰⁵ <https://github.com/rails/turbolinks>

¹⁰⁶ https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Manipulating_the_browser_history

¹⁰⁷ <http://blog.steveklabnik.com/posts/2012-09-27-seriously--numbers--use-them->

Desabilitando

Se **não quisermos utilizar o recurso por completo**, basta remover as linhas mostradas acima do arquivo `application.js` e também dos layouts.

Se **não quisermos utilizar o recurso em um determinado link**, podemos indicar isso no link utilizando o atributo `data-turbolinks`, da seguinte maneira:

```
1 <%= link_to "Mais detalhes", "/livro/#{mini_book.id}", class: "more_book_info"\n2 , data: { turbolinks: false } %>
```

Que vai gerar um link como esse:

```
1 <a class="more_book_info" data-turbolinks="false" href="/livro/1">Mais detalh\n2 es</a>
```



Dica

Reparem como foi utilizado uma *hash* com a chave `data`, que gerou os atributos `data-*` com as chaves e os valores da outra *hash* que foi enviada, inclusive trocando os sublinhados (_) para hífens (-). Isso é uma boa para utilizar JavaScript não-obstrutivo, como já comentado no livro.

Nesse link, será utilizado o comportamento padrão, ou seja, o navegador será redirecionado para a URL.

Verificando o funcionamento

Se utilizarmos uma ferramenta como o [Chrome Developer Tools](#)¹⁰⁸ e clicarmos em um link “Mais detalhes” na página inicial, podemos ver nas requisições de rede algo parecido com isso:

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
 1 /livro	GET	304 Not Modified	text/html	turbolinks.js?body=1:87 Script	440 B 9.1 KB	66 ms 66 ms

Requisições com Turbolinks

Podemos notar ali que o link foi interceptado pelo Turbolinks e feita uma requisição através de Ajax, vendo ali na coluna **Initiator** que mostra `turbolinks.js?body=1:87`.

Podemos também verificar que, mesmo com a requisição Ajax, a URL no navegador foi alterada para `http://localhost:3000/livro/1`, pois o Turbolinks utiliza a [History API](#)¹⁰⁹, convertendo cada clique nos links em uma *visita*, que pode ser de dois tipos.

¹⁰⁸ <https://developer.chrome.com/devtools>

¹⁰⁹ <https://developer.mozilla.org/en-US/docs/Web/API/History>

Visita de aplicação

Uma *visita de aplicação* tem a ação de *avançar* ou *substituir*, onde sempre faz uma requisição de rede, renderizando o HTML e completando a visita quando a resposta chega.

Quando possível, se existir uma cópia da página no *cache*, vai ser mostrada inicialmente essa cópia (mesmo desatualizada) para aumentar a percepção de velocidade do carregamento.

Se a URL contém uma âncora (#), vai ser tentado deslocar a página até essa âncora. Essas visitas sempre vão resultar em uma alteração no histórico do navegador, dependendo da *ação* da visita. Quando for para avançar, será incluída uma nova entrada no histórico do navegador através do método `pushState`¹¹⁰.

Pode ser desejado também o comportamento de substituição, através da ação `replace`, quando desejamos visitar uma página sem incluir ela no histórico do navegador, substituindo a entrada mais recente com o endereço corrente. Para utilizar essa opção nos links gerados pelo Rails, devemos utilizar a opção `data-turbolinks-action` como no exemplo:

```
1 link_to 'Mostrar', person, data: { 'turbolinks-action': 'replace' }
```

Visita de restauração

Esse tipo de visita é acionado sempre que é clicado nos botões de voltar ou avançar. Quando possível, o Turbolinks vai tentar recuperar uma página do *cache* sem fazer uma requisição. Do contrário, a requisição vai ser feita da maneira tradicional.

Também podemos ver na ferramenta que após a requisição, não foram carregados mais nenhum arquivo, diferente de uma requisição normal onde a página é recarregada e são mostrados vários arquivos sendo carregados, reduzindo bem o tempo de resposta.

Interagindo

Para ver como o Turbolinks é acionado, podemos inserir no arquivo `app/assets/javascripts/-bookstore.js` (vamos criar ele):

```
1 document.addEventListener('turbolinks:load', function() {
2   alert('Turbolinks carregado!');
3 });


```

A cada vez que uma página é carregada, é recebido o HTML e começam a ser avaliados os arquivos JavaScript, vamos ver esse alerta. Podemos ver eventos similares para quando a página seja renderizada, utilizando `before-render` e `render`:

¹¹⁰https://developer.mozilla.org/en-US/docs/Web/API/History_API

```
1 document.addEventListener('turbolinks:before-render', function() {  
2     alert('Antes do render!');  
3 });  
4  
5 document.addEventListener('turbolinks:render', function() {  
6     alert('Depois do render!');  
7 });
```

Inclusive, podemos trocar o evento `load` pelo evento `render`, já garantindo que será disparado após a renderização da página. Isso é útil quando é visitada uma página que já conste no `cache`, onde vai ser disparado o `before-render` automaticamente, seguido pelo `render`, **sem** chamar o `load`.

Podemos ver os eventos definidos para o Turbolinks [aqui¹¹¹](#), o que nos permite um controle bem granular da operação.



Dica

Reparam que eu escrevi o código JavaScript de maneira bem “crua”. Foi intencional. A partir do Rais 5.1, o `jQuery` vai ser opcional no Rails, mas nada impendendo o seu uso, vai ser só adicionar a `gem` novamente no `Gemfile`.

¹¹¹<https://github.com/turbolinks/turbolinks#full-list-of-events>

Carrinho de compras

Quantidade em estoque

Antes de mais nada, vamos criar um atributo novo no modelo Book, para armazenar quantos itens de determinado livro temos em nosso estoque:

```
1 $ rails g migration AddStockToBook stock:integer
2       invoke  active_record
3           create    db/migrate/20170321214702_add_stock_to_book.rb
```

Vamos editar o conteúdo da migration:

```
1 class AddStockToBook < ActiveRecord::Migration[5.0]
2   def up
3     add_column :books, :stock, :integer
4     Book.reset_column_information
5     Book.update_all(stock: 1)
6   end
7
8   def down
9     remove_column :books, :stock
10  end
11 end
```



Dica

Reparem como a migration já foi preenchida com add_column pelo que escrevemos no nome dela, AddStockToBook!

Criei uma coluna nova e já utilizei reset_column_information para poder utilizar ela logo abaixo. Agora vamos rodar a migration:

```
1 == 20170321214702 AddStockToBook: migrating =====\\
2 ==
3 -- add_column(:books, :stock, :integer)
4   -> 0.0006s
5 == 20170321214702 AddStockToBook: migrated (0.0006s)
6 =====
```

Alterar o formulário de livros na view do formulário (app/views/books/_form.html.erb):

```

1 <div class="field">
2   <%= form.label :stock %><br />
3   <%= form.number_field :stock %>
4 </div>

```

Alterar a `view show.html.erb` para mostrar o estoque:

```

1 <p>
2   <strong>Estoque:</strong>
3   <%= @book.stock %>
4 </p>

```

Traduzir o atributo:

```

1 book:
2   stock: "Estoque"

```

E finalmente liberar o atributo novo no controlador de livros:

```

1 def book_params
2   params.require(:book).permit(:title, :published_at, :text, :value, :person_\
3 id, :image_title, :data_stream, :stock, category_ids: [])
4 end

```



Dica

Já que estamos mexendo no formulário dos livros, vamos aproveitar e alterar o modo como o autor é selecionado, pois atualmente está mais feio que ir em quermesse bater no vendedor de maçã do amor. Primeiro vamos criar uma variável de instância com a coleção de pessoas no controlador de livros com um método utilizado em `before_action`:

```

1 before_action :load_people, only: [:new, :edit, :create, :update]
2 ...
3 private
4 ...
5 def load_people
6   @people = Person.all
7 end

```

E agora vamos criar um seletor na `view` utilizando `collection_select`:

```

1 <div class="field">
2   <%= form.label :person_id %><br />
3   <%= collection_select :book, :person_id, @people, :id, :name %>
4 </div>

```

Criando um método no modelo para verificar se o livro está fora de estoque:

```
1 def sold_out?  
2   stock < 1  
3 end
```

E agora escondendo o botão de comprar na view do livro na interface pública, (no arquivo app/views/pub/book.html.erb) no caso de livro estiver esgotado:

```
1 <% if @book.sold_out? %>  
2   <p class="sold_out">Esgotado! </p>  
3 <% else %>  
4   <%= link_to 'Comprar', buy_path(@book), class: 'buy_book' %>  
5 <% end %>
```

Configurando as sessões

Vamos alterar o comportamento atual de armazenamento de sessões, fazendo com que os dados de sessões fiquem gravados no banco de dados, e não mais em *cookies* locais e criptografados, onde, dependendo da quantidade de dados na nossa sessão, podem comprometer o tamanho, que no caso de *cookies*, é limitado em 4 Kb.

Para alterar o método de armazenamento de dados de sessão, vamos instalar a gem activerecord-session_store (necessária a partir do Rails 4), como sempre, inserindo no Gemfile:

```
1 gem 'activerecord-session_store'
```

Rodar o bundler:

```
1 $ bundle  
2 ...  
3 Installing activerecord-session_store 1.0.0  
4 ...
```

E criar o arquivo config/initializers/session_store.rb indicando para utilizar sessões usando o ActiveRecord, inserido a seguinte linha:

```
1 Rails.application.config.session_store :active_record_store
```

Agora precisamos criar uma tabela no banco para armazenar as sessões. Vamos gerar a migration que vai criar a tabela no banco de dados para armazenamento dos dados das sessões:

```
1 $ rails g active_record:session_migration
2       create db/migrate/20170321220216_add_sessions_table.rb
```

Dando uma olhada na *migration* gerada:

```
1 class AddSessionsTable < ActiveRecord::Migration[5.1]
2   def change
3     create_table :sessions do |t|
4       t.string :session_id, :null => false
5       t.text :data
6       t.timestamps
7     end
8
9     add_index :sessions, :session_id, :unique => true
10    add_index :sessions, :updated_at
11  end
12 end
```

E agora rodando a *migration*, não esquecendo de reiniciar o servidor depois:



Atenção

Verifiquem se a migration gerada consta a versão no final, após ActiveRecord::Migration, se não constar, a *gem* ainda não foi adaptada, podem inserir conforme o exemplo acima.

```
1 $ rails db:migrate
2 == 20170321220216 AddSessionsTable: migrating =====\\
3 ==
4 -- create_table(:sessions, {})
5   -> 0.0067s
6 -- add_index(:sessions, :session_id, {:unique=>true})
7   -> 0.0009s
8 -- add_index(:sessions, :updated_at)
9   -> 0.0010s
10 == 20170321220216 AddSessionsTable: migrated (0.0089s) =====\\
11 ==
```

As opções para armazenamento de sessões são:

- **cookie_store** - Opção *default*, limitada em tamanho de até 4Kb.
- **p_store** - Os dados são armazenados no formato PStore¹¹², que é um mecanismo de persistência em arquivo, baseado em Hash.

¹¹²<http://ruby-doc.org/stdlib/libdoc/pstore/rdoc/classes/PStore.html>

- **active_record_store** - O que vamos utilizar, onde é necessário criar a *migration* para gerar uma tabela, como demonstrado acima.
- **drb_store** - O DRb¹¹³ é um protocolo que permite que os processos em Ruby compartilhem objetos através de uma conexão de rede. Usando essa opção, os dados da sessão são armazenados em um servidor DRb, gerenciado fora da aplicação.
- **mem_cache_store** - Utiliza o Memcached¹¹⁴ através da interface RMemCache¹¹⁵.
- **memory_store** - Armazena na memória, sem serializar os dados.
- **file_store** - Armazena em arquivos regulares, sem nenhuma formatação específica.

Criando o carrinho

Agora que nossa sessão já está configurada, vamos criar uma classe para agir como um carrinho de compras, criando um novo arquivo chamado `cart.rb` no diretório `app/services` - reparem que não vamos usar nenhum gerador do Rails para isso, pois essa classe **não é um modelo** que reflete no banco de dados, é um simples PORE (Plain Old Ruby Object). **Lembrem-se de verificar o status do Spring após inserir novos diretórios na sua aplicação**. Se necessário, utilizem `spring stop` e reiniciem o servidor. Aqui tem o código do carrinho:

```

1 class Cart
2   def initialize
3     clear
4   end
5
6   def <<(product)
7     @items[product.id] += 1
8     @items
9   end
10
11  def -(product)
12    @items.delete(product.id) if @items.key?(product.id)
13    @items
14  end
15
16  def include?(product)
17    @items.key?(product.id)
18  end
19
20  def total
21    @items.inject(0) do |memo, item|
22      memo += Book.find(item[0]).value * item[1]
23      memo

```

¹¹³http://en.wikipedia.org/wiki/Distributed_Ruby

¹¹⁴<http://memcached.org>

¹¹⁵<http://deveiate.org/projects/RMemCache>

```
24      end
25  end
26
27  def clear
28    @items = Hash.new(0)
29  end
30
31  def items
32    @items.map do |id, qty|
33      {
34        id:  id,
35        item: Book.find(id),
36        qty:  qty
37      }
38  end
39 end
40 end
```



Atenção

Aqui tem uma pegadinha: se após criar um novo diretório abaixo de app e as classes não ficarem acessíveis no *console* mesmo após você usar `reload!`, abrir e fechar, xingar, bufar, ir no banheiro, tentar de novo, etc e tal, batata que é o o Spring. Saíam do *console*, digitem

```
1 bundle exec spring stop
```

e tente novamente. Também pode ser necessário reiniciar o servidor web, interrompendo-o com `CTRL+C` e rodando novamente.

Temos que definir um método para encontrar o objeto do carrinho de compras na sessão corrente, onde se não existir, é criado, no controlador pub:

```
1 def find_cart
2   session[:cart] ||= Cart.new
3 end
```

Comprando um produto

Já temos a nossa URL de compra, a agora vamos definir uma nova rota chamada `/carrinho` para listar os produtos que temos em nosso carrinho:

```
1 get 'carrinho' => 'pub#cart', as: 'cart'
```

Os métodos para inserir um livro no carrinho e listar o carrinho no controlador pub:

```
1 def buy
2   find_cart << Book.find(params[:id])
3   redirect_to action: :cart
4 end
5
6 def cart
7   @cart = find_cart
8 end
```

E finalmente, a *view* e a *partial* do carrinho, primeiro com app/views/pub/cart.html.erb:

```
1 <h1>Seu carrinho de compras</h1>
2
3 <ul id="cart_items">
4   <%= render partial: 'cart_item', collection: @cart.items %>
5 </ul>
6
7 <p>
8   Total: <span class="book_value" id="cart_total"><%= number_to_currency(@car\ 
9 t.total) %></span>
10 </p>
11
12 <%= link_to 'Continuar comprando', root_path %>
```

Agora a *partial* em app/views/pub/_cart_item.html.erb:

```
1 <li id="item_<%= cart_item[:id] %>">
2   <%= image_tag(cart_item[:item].image, height: 100) %>
3   <%= cart_item[:item].title %><br/>
4   <%= text_field_tag :qty, cart_item[:qty], size: 3, readonly: 'readonly' %>
5   <span class="book_value"><%= number_to_currency(cart_item[:item].value) %><\ 
6 /span>
7 </li>
```

Clicando no botão “Comprar” de algum livro, vamos ter uma imagem como a seguinte:

Bookstore - A maior livraria virtual (depois da Amazon)

Início

Seu carrinho de compras

Conhecendo Ruby

Conhecendo Ruby
1 R\$ 1,00

Total: **R\$ 1,00**

[Continuar comprando](#)

Desenvolvido no ebook "Conhecendo Rails"

Carrinho de compras

O CSS básico para essa página foi o seguinte:

```
1 ul#cart_items {  
2     list-style-type: none;  
3     padding: 0;  
4 }  
5  
6 ul#cart_items li {  
7     height: 100px;  
8 }  
9  
10 ul#cart_items li img {  
11     float: left;  
12     margin-right: 1em;  
13 }  
14  
15 ul#cart_items li span.book_value {  
16     margin-top: 1em;  
17     font-size: 1em;  
18 }  
19  
20 a.remove_book {  
21     text-decoration: none;  
22     color: red;  
23     font-weight: bold;  
24 }
```

Também temos que incluir o link para a página do carrinho de compras no layout:

```

1 ...
2 <nav>
3   <ul>
4     <li><%= link_to 'Inicial', '/' %></li>
5     <li><%= link_to 'Carrinho', cart_path %></li>
6   </ul>
7 </nav>
8 ...

```

Removendo um produto do carrinho, com Ajax

Vejam que temos um carrinho bem simples, com a quantidade “travada”, não permitindo alterar, ou mesmo remover um produto. Mesmo assim, é interessante deixar um link para remover o produto do carrinho, ao invés de digitar 0 na quantidade ou ficar clicando até chegar em zero (ei, vamos fazer isso depois!). Para criar uma ação para remover o produto do carrinho, primeiro vamos criar uma rota com o método correto, DELETE, no arquivo de rotas:

```
1 delete 'remover/:id' => 'pub#remove', as: 'remove'
```

Vamos alterar a partial que mostra o produto no carrinho, para incluir o link de remoção:

```

1 <%= link_to 'Remover', remove_path(cart_item[:id]), method: 'delete', data: { \
2   confirm: 'Tem certeza?' } %><br/>
```

Implementar a ação no controlador:

```

1 def remove
2   @book = Book.find(params[:id])
3   @cart = find_cart
4   @cart - @book
5   redirect_to cart_path
6 end
```

Reparam que utilizamos o método - do carrinho para excluir o livro (vocês leram o meu livro sobre Ruby antes desse aqui, correto? se não, corram para lá para entender que feitiçaria que foi feita ali!) e logo após redirecionamos novamente para o carrinho, que vai estar sem o produto em questão. Mas espera aí, onde está o Ajax¹¹⁶ (não, não é aquele produto de limpeza) ali do título? Clicando no link não gerou uma requisição assíncrona, trocamos de página por 2 vezes (primeiro indo para a ação de remover, depois voltando para o carrinho).

Para gerar uma requisição assíncrona, temos que fazer a tarefa hercúlea de inserir remote: true no método link_to utilizado ali. É. Só isso. Vamos ver:

¹¹⁶[http://pt.wikipedia.org/wiki/AJAX_\(programa%C3%A7%C3%A3o\)](http://pt.wikipedia.org/wiki/AJAX_(programa%C3%A7%C3%A3o))

```
1 <%= link_to 'Remover', remove_path(cart_item[:id]), method: 'delete', remote:\
2   true, data: { confirm: 'Tem certeza?' } %><br/>
```

Com isso, já temos a nossa requisição assíncrona! O resultado vai ficar exatamente igual o anterior pois temos um redirecionamento na ação, mas se comentarmos o `redirect_to` do controlador vamos ver que a página não sai do lugar, mas a ação foi disparada e executada. Se recarregarmos a página do carrinho novamente vamos constatar isso.

Mas precisamos dar um indicativo que a ação foi executada, quando a dispararmos de modo assíncrono. Faz sentido remover o item que foi excluído da página, não é mesmo? Como já renderizamos cada livro em um elemento `li` que tem como id a string `item_` seguida pelo id do livro, podemos excluir ela da página. E como fazemos isso?

Simples, podemos na ação renderizar vários tipos de respostas, entre elas, código JavaScript! Esse código vai ser responsável por, além de excluir o elemento `LI`, atualizar o elemento com o id `cart_total` para o valor correto do carrinho. Vamos criar uma `view` para essa ação, determinando que o tipo dela vai ser JavaScript, criando o arquivo `remove.js.erb` no diretório de `views` do controlador `pub` com o seguinte conteúdo:

```
1 $("#item_<%= @book.id %>").fadeOut(1000);
2 $("#cart_total").text(" <%= number_to_currency(@cart.total) %>");


```

O segredo ali é que o que é retornado pela view, que inclusive pode conter código Ruby no formato tradicional do ERb das `views`, é enviado de volta para o cliente, como código JavaScript, e executado no navegador.



Dica

Podemos ver que estamos utilizando o efeito `fadeOut(<milisegundos>)` para fazer com que o elemento `LI` do livro correspondente desapareça da página, mas também podemos utilizar quaisquer dos outros efeitos disponíveis no jQuery¹¹⁷, listados na página com a lista de efeitos padrão¹¹⁸, ou utilizar outros efeitos customizados, através de arquivos JavaScript carregados juntos com a nossa aplicação.

Agora podemos remover o `redirect_to` da ação `remove` e verificar como ficou o nosso carrinho com requisição assíncrona. Porém, a partir do Rails 5.1 tem um pequeno probleminha: ali no arquivo JavaScript utilizamos jQuery, que era o *framework* JavaScript padrão no Rails, mas ele foi removido como dependência. Para que nosso *script* funcione corretamente, temos que inserir a seguinte `gem` no `Gemfile` (daria para colocar como pacotes do Yarn, mas ainda prefiro manter o comportamento anterior) e rodar o `bundler`:

```
1 gem 'jquery-rails'
```

Inserir o jQuery no `app/assets/javascripts/application.js`:

¹¹⁷ <http://jquery.com>

¹¹⁸ <http://api.jquery.com/category/effects/>

```
1 //= require jquery
```

E com isso temos o jQuery de volta e podemos testar a remoção do carrinho, via Ajax.

Alterando a quantidade do produto, com Ajax

Vamos alterar a `partial` que mostra o produto no carrinho, para incluir uma ação de alteração da quantidade do produto, primeiro alterando de um campo texto simples para um campo tipo número (o `type=number` do HTML5), alterando em `app/views/pub/_cart_item.html.erb`:

```
1 <%= number_field_tag :qty, cart_item[:qty], min: 0, max: 999, class: 'cart-qt' \
2 %> 'data-id': cart_item[:id]
```

Vejam que inserimos alguns atributos nos elementos que vão nos permitir identificar e recuperar dados de cada item do carrinho, como a classe dos elementos (`cart-qty`) e o id de cada produto (`data-id`). Precisamos agora de um meio de disparar os eventos de alterações dos produtos. Vamos definir uma rota para fazer a alteração de produto, utilizando o verbo HTTP PATCH, criando no arquivo de rotas:

```
1 patch 'alterar/:id' => 'pub#change', as: 'change'
```

Implementar o método no carrinho para alterar a quantidade de um produto, removendo ele do carrinho caso seja 0 ou menor:

```
1 def change(product, qty)
2   if qty <= 0
3     self - product
4     return
5   end
6   @items[product.id] = qty
7 end
```

E agora o método no controlador:

```
1 def change
2   @book = Book.find(params[:id])
3   @cart = find_cart
4   @cart.change(@book, params[:qty].to_i)
5 end
```

Vamos criar um arquivo JavaScript para disparar nossos eventos. Vejam que todos os arquivos JavaScript definidos nos `assets` quando enviados para produção vão ser todos juntos em um arquivo só, que será minificado. Podemos criar algumas classes (ainda não é ES6 ...) como meio de deixar nosso código mais organizado. Vamos criar um arquivo em `app/assets/javascript/cart.js` com o seguinte conteúdo, que inclusive vai utilizar um `namespace`:

```

1 var Bookstore = typeof Bookstore === "undefined" ? {} : Bookstore;
2
3 Bookstore.Cart = function() {
4     this.fire = function() {
5         $('.cart-qty').on('input', function(evt) {
6             var target = $(evt.target);
7             var id     = target.attr('data-id');
8             var qty    = parseInt(target.val());
9
10            if (isNaN(qty)) {
11                return false;
12            }
13
14            $.ajax({
15                url: "/alterar/" + id,
16                method: 'PATCH',
17                dataType: 'script',
18                data: { qty: qty }
19            });
20        });
21    };
22}

```



Dica

Reparem no `dataType`, é definido como `script`.



Dica

Estudem JavaScript. Por mais que tenhamos alguns *transpilers* disponíveis por aí, JavaScript é um negócio bem universal hoje para a web, e sabendo como programar com ele você pode utilizá-lo em muitas coisas.

Precisamos disparar esses eventos quando a página for carregada. Vamos inserir um pequeno código JavaScript na `view` (não, não é pecado, você não vai queimar no inferno por causa disso e ainda de quebra vai resolver o que precisamos) do carrinho de compras, bem no final:

```

1 <script type='text/javascript'>
2     (new Bookstore.Cart()).fire();
3 </script>

```

Isso vai fazer com que a ação seja disparada a cada vez que o valor seja alterado.

E agora a `view` respectiva dessa ação do controlador, que nesse caso vai ser um arquivo chamado `change.js.erb`, com o seguinte conteúdo:

```

1 <% if @cart.include?(@book) %>
2   $("#" + item_<%= @book.id %>).animate({ opacity: 0.25 }).animate({ opacity: 1 \
3 });
4 <% else %>
5   $("#" + item_<%= @book.id %>).fadeOut(1000);
6 <% end %>
7
8 $("#cart_total").text("<%= number_to_currency(@cart.total) %>");
```

Na *view* verificamos se o livro consta no carrinho, se não constar mais (a quantidade chegou a 0) ele é removido como fizemos anteriormente (com o efeito `fadeOut`), mas se não é criada uma animação para dar uma “piscada” no livro que teve a sua quantidade alterada. O total do carrinho também é atualizado e exibido.

Agora aqui tem uma pegadinha: como utilizamos o método `PATCH` para atualizar o carrinho, a proteção do Rails contra CSRF (*cross site request forgery*) vai bloquear a nossa requisição, que não vai ter o *token* gerado por um formulário regular do *framework* (gerado através dos *helpers form_*), então logo no começo do controlador pub *podemos* desabilitar essa verificação utilizando a seguinte linha:

```
1 skip_before_action :verify_authenticity_token, only: :w(change)
```

Isso fará com que a verificação do *token* não seja disparada para a ação `change`. Como isso talvez não seja uma boa idéia, vamos alterar nosso código JavaScript para enviar o *token* corrente, alterando a linha que envia os dados de

```
1 data: { qty: qty }
```

para

```
1 data: { qty: qty, authenticity_token: $('meta[name=csrf-token]').attr('content') }
```

Fechando o pedido

Agora que temos nosso carrinho de compras, temos que criar uma forma de concluir o pedido, criando um pedido com os itens que foram escolhidos, fazendo baixa no estoque e enviando um *e-mail* para o comprador, indicando que a venda foi bem sucedida.

Vamos alterar nosso modelo `Book` para incluir um método para vender o livro, reduzindo a sua quantidade em estoque. No caso de acontecer algum problema, temos que indicar, além da mensagem de erro, o tipo de erro que foi gerado. Para isso, podemos criar exceções personalizadas, podendo as criar em arquivos separados ou mesmo dentro do próprio modelo de onde elas poderão ser disparadas. Vamos definir, no arquivo do modelo de `Book`, as seguintes exceções, logo no começo do arquivo, antes da declaração da classe do modelo:

```

1 SoldOutException = Class.new(StandardError)
2 NotEnoughException = Class.new(StandardError)

1 def sell(qty = 1)
2   raise SoldOutException, 'Esgotado' if sold_out?
3   raise NotEnoughException, 'Não suficiente' if self.stock - qty < 0
4   self.stock -= qty
5   save!
6 end

```

Agora vamos gerar os modelos de Order (pedido) e OrderItem (item de pedido), sendo que Order pertence à uma Person e tem vários OrderItems e OrderItem pertence à uma Order. Vamos fazer isso através de *migrations*:

```

1 $ rails g model Order person:references
2   invoke active_record
3   create db/migrate/20170322222455_create_orders.rb
4   create app/models/order.rb
5   invoke test_unit
6   ...

```

Vamos preencher a *migration* gerada com:

```

1 class CreateOrders < ActiveRecord::Migration[5.0]
2   def change
3     create_table :orders do |t|
4       t.references :person, foreign_key: true, index: true, null: false
5       t.timestamps
6     end
7   end
8 end

```

E agora o modelo para OrderItem:

```

1 $ rails g model OrderItem order:references book:references quantity:integer \v\
2 alue:decimal
3   invoke active_record
4   create db/migrate/20170322222902_create_order_items.rb
5   create app/models/order_item.rb
6   invoke test_unit
7   ...

```

e preencher a *migration* gerada com o seguinte conteúdo:

```
1 class CreateOrderItems < ActiveRecord::Migration[5.0]
2   def change
3     create_table :order_items do |t|
4       t.references :order, foreign_key: true, index: true, null: false
5       t.references :book, foreign_key: true, index: true, null: false
6       t.integer :quantity, null: false
7       t.decimal :value, precision: 15, scale: 2, null: false
8       t.timestamps
9     end
10   end
11 end
```



Dica

Reparem que utilizei `precision` e `scale` na coluna `value` para armazenar o valor do item, com 2 casas decimais.



Dica

Resistam à tentação de utilizar alguns campos especializados para gravar valores no banco de dados. Exemplo é o campo tipo `MONEY` do PostgreSQL. Mantenham o seu sistema simples e funcional.

Vamos rodar as *migrations*:

```
1 $ rails db:migrate
2 == 20170322222455 CreateOrders: migrating =====\\
3 ==
4 -- create_table(:orders)
5   -> 0.0058s
6 == 20170322222455 CreateOrders: migrated (0.0059s) =====\\
7 ==
8
9 == 20170322222902 CreateOrderItems: migrating =====\\
10 ==
11 -- create_table(:order_items)
12   -> 0.0028s
13 == 20170322222902 CreateOrderItems: migrated (0.0028s) =====\\
14 ==
15 <Paste>
```

Agora temos que adequar as associações nos modelos, primeiro em Order:

```
1 class Order < ApplicationRecord
2   has_many :order_items
3   belongs_to :person
4   validates :person, presence: true
5
6   def total
7     order_items.reduce(0) do |memo, item|
8       memo += item.quantity * item.value
9     memo
10    end
11  end
12 end
```

Agora em OrderItem:

```
1 class OrderItem < ApplicationRecord
2   validates :order, presence: true
3   validates :book, presence: true
4   validates :quantity, presence: true, numericality: { greater_than: 0 }
5   belongs_to :order
6   belongs_to :book
7 end
```

E agora em Person:

```
1 class Person < ApplicationRecord
2 ...
3 has_many :orders
```

Vamos criar métodos, no controlador pub, para converter o carrinho de compras em um pedido, fechar e mostrar pedidos. Como pessoa, já vamos utilizar quem estiver autenticado na aplicação, através da página de login. Se a pessoa não tiver sido autenticada, vamos redirecionar para lá, utilizando aquela Exception que criamos anteriormente para tratar isso.

Antes de mais nada vamos fazer um método no modelo de pedidos (Order) para criar um novo pedido através da pessoa que fez autenticação e do conteúdo do carrinho de compras:

```

1 def self.create_by_cart(person_id, items)
2   order = Order.new(person_id: person_id)
3   items.each do |item|
4     item[:item].sell(item[:qty])
5     order_item = OrderItem.new
6     order_item.book = item[:item]
7     order_item.quantity = item[:qty]
8     order_item.value = item[:item].value
9     order.order_items << order_item
10 end
11 order.save ? order : nil
12 end

```



Atenção

O que acontece se for gerada uma exceção ali, quando for tentado vender o livro, e ele estiver sem estoque? Vamos ver como tratar isso mais para frente.

E agora os métodos no controlador, para fechar e visualizar o pedido:

```

1 def close_order
2   raise NotAuthenticated, 'Faça o login primeiro' if session[:id].blank?
3   @cart = find_cart
4   @order = Order.create_by_cart(session[:id], @cart.items)
5
6   if @order.blank?
7     flash[:notice] = 'Não foi possível criar o seu pedido!'
8     redirect_to root_path
9     return
10 end
11
12 @cart.clear
13 redirect_to order_path(@order)
14 end
15
16 def order
17   @order = Order.find(params[:id])
18 end

```

Criar as rotas para fechar e mostrar o pedido:

```

1 get 'fechar' => 'pub#close_order', as: 'close_order'
2 get 'pedido/:id' => 'pub#order', as: 'order'

```

Criar a view app/views/pub/order.html.erb, para mostrar o pedido:

```
1 <h1>Seu pedido</h1>
2
3 <h2>Número <%= @order.id %></h2>
4
5 Itens:
6
7 <ul id="order_items">
8   <% for item in @order.order_items %>
9     <li>
10       <%= item.book.title %> -
11       <%= item.quantity %> -
12       <%= number_to_currency(item.value) %> -
13       <%= number_to_currency(item.quantity * item.value) %>
14     </li>
15   <% end %>
16 </ul>
17
18 <p>
19   Total: <span class="book_value"><%= number_to_currency(@order.total) %></span>
20   an>
21 </p>
```

E alterar a *view* app/views/pub/cart.html.erb para incluir um link para fechar o pedido:

```
1 <%= link_to 'Fechar pedido', close_order_path, data: { confirm: 'Tem certeza?' } %>
```

Clicando em “Fechar pedido”, vamos ter algo parecido com a seguinte imagem:

Bookstore - A maior livraria virtual (depois da Amazon)

Inicial Carrinho

Seu pedido

Número 4

Itens:

- Conhecendo Ruby - 2 - R\$ 15,00 - R\$ 30,00
- Conhecendo o Git - 1 - R\$ 10,00 - R\$ 10,00

Total: **R\$ 40,00**

Desenvolvido no ebook "Conhecendo Rails"

Pedido feito

Mais um pouco de muitos através

Olhando nossos modelos, vemos que agora, através dos itens do pedido, temos uma associação com os livros, e também com as ordens. Usando `has_many :through`, já visto anteriormente, podemos acessar as ordens diretamente dos livros:

```
1 class Book < ActiveRecord::Base
2 ...
3 has_many   :order_items
4 has_many   :orders, through: :order_items
```

O que nos dá algo como, testando no console, após fazer um pedido:

```
1 > Book.first.orders
2 Book Load (0.1ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" ASC
3 LIMIT ?  [["LIMIT", 1]]
4 Order Load (0.2ms)  SELECT "orders".* FROM "orders" INNER JOIN "order_item
5 ms"
6 ON "orders"."id" = "order_items"."order_id" WHERE "order_items"."book_id"\n
7 =
8 ?  [["book_id", 3]]
9 => #<ActiveRecord::Associations::CollectionProxy:0x007f8000000000>
10 id:
11 3, created_at: "2017-03-26 13:46:33", updated_at: "2017-03-26 13:46:33">\n
12 ]>
```

Também podemos fazer isso em Person:

```

1 class Person < ActiveRecord::Base
2 ...
3 has_many :orders
4 has_many :order_items, through: :orders
5 ...

```

O que nos dá algo como:

```

1 > Person.first.order_items
2   Person Load (0.3ms)  SELECT "people".* FROM "people" ORDER BY "people"."na\
3 me"
4   ASC LIMIT ?  [{"LIMIT": 1}]
5     OrderItem Load (0.2ms)  SELECT "order_items".* FROM "order_items" INNER J\ \
6 OIN
7       "orders" ON "order_items"."order_id" = "orders"."id" WHERE
8       "orders"."person_id" = ?  [{"person_id": 4}]
9     => #<ActiveRecord::Associations::CollectionProxy []>

```

Podemos inclusive ter a associação entre Order e OrderItem descrita de maneira mais simples (para o uso) da seguinte forma:

```

1 class Order < ActiveRecord::Base
2 ...
3 has_many :items, class_name: 'OrderItem', foreign_key: 'order_id'
4 ...

```

Que nos permite acessar items no pedido, que fica a mesma coisa que order_items:

```

1 > Order.first.order_items.map { |item| item.book.title }
2 => ["Conhecendo Ruby", "Conhecendo o Git"]
3
4 > Order.first.items.map { |item| item.book.title }
5 => ["Conhecendo Ruby", "Conhecendo o Git"]

```

É uma questão de ter um pouco mais de código da definição e menos código no uso, além, claro, de uma certa clareza e maior simplicidade.

Se quiséssemos descobrir quem foram as pessoas que compraram determinado livro, também podemos fazer um has_many through em Book, que já tem uma associação que descobre os pedidos através dos itens, apontando direto para as pessoas dos pedidos, dessa forma:

```
1 class Book < ActiveRecord::Base
2 ...
3 has_many :order_items
4 has_many :orders, through: :order_items
5 has_many :people, -> { distinct }, through: :orders
6 ...
```

Isso nos permite visualizar as pessoas que compraram o livro, sem repetir (afinal, uma pessoa pode ter feito vários pedidos com o mesmo livro), pois enviamos a cláusula `distinct` através de uma `lambda`. Inclusive, nessa `lambda` podem ser especificados quaisquer filtros necessários para o uso da associação. Dando uma olhada no resultado e na consulta SQL gerada:

```
1 > Book.first.people
2   Book Load (0.1ms)  SELECT "books".* FROM "books" ORDER BY "books"."id" ASC
3   LIMIT ?  [["LIMIT", 1]]
4   Person Load (0.4ms)  SELECT DISTINCT "people".* FROM "people" INNER JOIN
5     "orders" ON "people"."id" = "orders"."person_id" INNER JOIN "order_items" ON
6     "orders"."id" = "order_items"."order_id" WHERE "order_items"."book_id" = ?
7   ORDER BY "people"."name" ASC  [["book_id", 3]]
8 => #<ActiveRecord::Associations::CollectionProxy [#<Person id: 3, name:
9   "Eustáquio Rangel de Oliveira Jr.", email: "taq@bluefish.com.br",
10  password_digest: "$2a$10$0neZu10i8Bc.Y2HARFpu9e0VYhmCdIwvV2I1JKGTWfW...",
11  born_at: "1970-01-01", admin: true, created_at: "2017-03-26 13:46:32",
12  updated_at: "2017-03-26 13:46:32"]>
```

Aí podemos ver o uso de `SELECT DISTINCT` conforme foi especificado na `lambda`.

Agora temos um pequeno grande problema: o que acontece se, durante o fechamento de um pedido, muitas pessoas estiverem comprando o mesmo produto? Como ele vai responder no método `sell` ao decremento do produto no estoque?

Travando registros na aplicação

Em Rails temos duas metodologias para garantir integridade de transações, permitindo “travar” registros e objetos que se relacionam com o banco de dados, que são:

- **Optimistic Locking** - Esse mecanismo de *locking* permite que múltiplos usuários acessem o mesmo registro para edição, e assume um mínimo de conflitos com os dados. Ele verifica quando outro processo fez alterações em um registro desde que ele foi recuperado. Se há uma tentativa de alteração no registro, uma exceção do tipo `ActiveRecord::StaleObjectError` é disparada e a atualização é ignorada.
- **Pessimistic Locking** - Esse mecanismo de *locking* é feito pelo banco de dados. Usando o método `lock` quando criando uma relação entre objeto e banco de dados obtém uma trava exclusiva nas linhas selecionadas. As relações usando `lock` são costumeiramente envolvidas dentro de uma transação para prevenção de *deadlocks*.

Optimistic Locking

Vamos dar uma olhada primeiro nesse mecanismo. Para funcionar, o Rails só pede uma coluna chamada `lock_version` no modelo, com valor default de `0`. Vamos criar a coluna no modelo dos livros com uma *migration*:

```
1 $ rails g migration AddLockVersionToBooks lock_version:integer
2       invoke  active_record
3       create    db/migrate/20170326145006_add_lock_version_to_books.rb
```

Vamos editar a *migration* para ter o valor `default` como `0`:

```
1 class AddLockVersionToBooks < ActiveRecord::Migration[5.0]
2   def change
3     add_column :books, :lock_version, :integer, default: 0
4   end
5 end
```

E rodar a *migration*:

```
1 rails db:migrate
2 == 20170326145006 AddLockVersionToBooks: migrating =====\ 
3 ==
4 -- add_column(:books, :lock_version, :integer, {:default=>0})
5   -> 0.0067s
6 == 20170326145006 AddLockVersionToBooks: migrated (0.0068s) =====\ 
7 ==
```



Dica

Podemos especificar outro nome para a coluna `lock_version`, utilizando `set_locking_column` no modelo:

```
1 set_locking_column :outro_nome
```

Isso vai nos dar o seguinte comportamento (tenham certeza que existe um número de livros em estoque):

```
1 > b1 = Book.first
2   Book Load (0.2ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" ASC\ 
3   LIMIT ?  [[["LIMIT", 1]]
4 => #<Book id: 3, title: "Conhecendo Ruby", published_at: "2013-06-29", text:
5   "Livro sobre a linguagem de programação **Ruby**.\n\n...", value:
6   #<BigDecimal:397aa10,'0.1E1',9(18)>, person_id : 3, created_at: "2017-03-26
7   13:46:32", updated_at: "2017-03-26 13:46:32", stock: 10, lock_version: 0>
8
9 > b2 = Book.first
10  Book Load (0.1ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" ASC\ 
11  LIMIT ?  [[["LIMIT", 1]]
12 => #<Book id: 3, title: "Conhecendo Ruby", published_at: "2013-06-29", text:
13   "Livro sobre a linguagem de programação **Ruby**.\n\n...", value:
14   #<BigDecimal:395ea90,'0.1E1',9(18)>, person_id : 3, created_at: "2017-03-26
15   13:46:32", updated_at: "2017-03-26 13:46:32", stock: 10, lock_version: 0>
16
17 > b1.sell
18   (0.1ms) begin transaction
19   Person Load (0.2ms)  SELECT  "people".* FROM "people" WHERE "people"."id" =\ 
20   ?
21   ORDER BY "people"."name" ASC LIMIT ?  [[{"id": 3}, {"LIMIT": 1}]] 
22   SQL (0.3ms) UPDATE "books" SET "updated_at" = '2017-03-26 16:28:35.009662',
23   "stock" = 9, "lock_version" = 1 WHERE "books"."id" = ? AND
24   "books"."lock_version" = ?  [[{"id": 3}, {"lock_version": 0}]] 
25   (5.8ms) commit transaction
26 => true
```

```
27
28 > b2.sell
29   (0.1ms) begin transaction
30 Person Load (0.1ms) SELECT "people".* FROM "people" WHERE "people"."id" = \
31 ?
32 ORDER BY "people"."name" ASC LIMIT ? [[{"id": 3}, {"LIMIT": 1}]]  
SQL (0.1ms) UPDATE "books" SET "updated_at" = '2017-03-26 16:28:38.153255',
33 "stock" = 9, "lock_version" = 1 WHERE "books"."id" = ? AND
34 "books"."lock_version" = ? [[{"id": 3}, {"lock_versi
35 on": 0}]]  
(0.0ms) rollback transaction
36 ActiveRecord::StaleObjectError: Attempted to update a stale object: Book.
```

Ou seja, após a primeira vez que o registro foi salvo, a exceção foi disparada, indicando que o objeto (e consequentemente o registro) não se encontrava em um estado atualizado. Podemos pedir para o objeto se atualizar, utilizando `reload`, e tentar salvar novamente:

```
1 > b2.reload
2 Book Load (0.2ms) SELECT "books".* FROM "books" WHERE "books"."id" = ? LI \
3 MIT
4   ? [[{"id": 3}, {"LIMIT": 1}]]  
5 => #<Book id: 3, title: "Conhecendo Ruby", published_at: "2013-06-29", text:
6   "Livro sobre a linguagem de programação **Ruby**.\n\n...", value:
7   #<BigDecimal:3590530,'0.1E1',9(18)>, person_id: 3, created_at: "2017-03-26
8   13:46:32", updated_at: "2017-03-26 16:28:35", stock: 9, lock_version: 1>
9
10 > b2.sell
11   (0.1ms) begin transaction
12 Person Load (0.1ms) SELECT "people".* FROM "people" WHERE "people"."id" = \
13 ?
14 ORDER BY "people"."name" ASC LIMIT ? [[{"id": 3}, {"LIMIT": 1}]]  
SQL (0.2ms) UPDATE "books" SET "updated_at" = '2017-03-26 16:30:15.570154',
15 "stock" = 8, "lock_version" = 2 WHERE "books"."id" = ? AND
16 "books"."lock_version" = ? [[{"id": 3}, {"lock_version": 1}]]  
(5.6ms) commit transaction
17 => true
```



O que ocorre é o seguinte: vamos supor que o valor corrente da `lock_version`, recuperado em ambos `b1` e `b2`, é 3. Quando `b1` é atualizado, esse valor é alterado, junto com `stock`, dentro das condições do comando SQL:

```
1 UPDATE "books" SET "stock" = 997, "updated_at" = '2013-07-14
2 21:44:59.911004', "lock_version" = 6
3 WHERE ("books"."id" = 1 AND "books"."lock_version" = 5)
```

Podemos ver que o valor corrente de `lock_version` é enviado junto na cláusula `WHERE`. Quando tentamos utilizar `sell` em `b2`, esse valor é conferido e impedido de atualizar o registro. Utilizando `reload` o valor corrente da `lock_version` é atualizado e tudo prossegue normalmente.

Agora podemos alterar o código de criação de pedidos para utilizar uma transação, onde vai ser executado `rollback` se alguma exceção for disparada no método `sell!` ou no método `save!` (vejam que aqui eu executei o método `save!`, com o ponto de exclamação no final, para disparar uma exceção e não retornar `true` ou `false` como o método `save` tradicional):

```
1 def self.create_by_cart(person_id, items)
2   Book.transaction do
3     order = Order.new(person_id: person_id)
4     items.each do |item|
5       item[:item].reload.sell(item[:qty])
6       order_item      = OrderItem.new
7       order_item.book = item[:item]
8       order_item.quantity = item[:qty]
9       order_item.value    = item[:item].value
10      order.order_items << order_item
11    end
12    order.save!
13    order
14  end
15 rescue Exception => e
16   logger.error "erro salvando o pedido: #{e}"
17   nil
18 end
```



Dica

Também podemos utilizar um `callback` como fizemos para detectar as exceções do tipo `ActiveRecord::RecordNotFound`, nesse caso interceptando as exceções `ActiveRecord::StaleObjectError` e redirecionar para um determinado método.

No momento em que o livro vai ter seu estoque alterado, ele é recarregado, atualizando seu estoque, e depois é feita a tentativa de venda. Usamos `Book.transaction do ... end` para

garantir que, dentro desse bloco, se algum livro falhar ao ser vendido, os outros não serão alterados.

As transações podem ser aninhadas (talvez tendo que utilizar o método `requires_new: true` dentro de algum aninhamento para garantir o *rollback* do aninhamento) e **não são** distribuídas entre conexões ao banco de dados, agindo em apenas uma conexão. Para mais detalhes sobre aninhamento, pode ser consultada a documentação oficial das transações no ActiveRecord¹¹⁹.

Pessimistic Locking

Para utilizar *Pessimistic Locking*, é só utilizar o método `lock`, dentro de uma transação, como no exemplo:

```
1 Book.transaction do
2   b1 = Book.lock.where(["id = ?", 1]).first
3   b1.stock = 3
4   b1.save!
5 end
```

É importante notar que o banco de dados tem que dar suporte para esse recurso. Para mais informações, podemos consultar a documentação do MySQL¹²⁰ do PostgreSQL¹²¹.

Também podemos especificar o `lock` dessa maneira:

```
1 b = Book.find(1, lock: true)
```

Passando uma mensagem para indicar que o registro está alocado:

```
1 b = Book.find(1, lock: "estou alocado")
```

Utilizando por instância:

```
1 > b = Book.first
2   Book Load (0.2ms)  SELECT "books".* FROM "books" ORDER BY "books"."id" ASC
3   LIMIT ?  [["LIMIT", 1]]
4 => #<Book id: 1, title: "Conhecendo Ruby", published_at: "2013-06-29", text:
5   "Livro sobre a linguagem de programação **Ruby**.\n\n...", value:
6   #<BigDecimal:31f9f98,'0.1E1',9(18)>, person_id: 1, created_at: "2017-03-29
7   21:58:47", updated_at: "2017-03-29 22:11:14", stock: 7, lock_version: 3>
8
9 > b.lock!
```

¹¹⁹<http://api.rubyonrails.org/classes/ActiveRecord/Transactions/ClassMethods.html>

¹²⁰<http://dev.mysql.com/doc/refman/5.1/en/innodb-locking-reads.html>

¹²¹<http://www.postgresql.org/docs/current/interactive/sql-select.html#SQL-FOR-UPDATE-SHARE>

```
10 Book Load (0.2ms)  SELECT  "books".* FROM "books" WHERE "books"."id" = ? LI\
11 MIT
12 ?  [{"id": 1}, {"LIMIT": 1}]
13 => #<Book id: 1, title: "Conhecendo Ruby", published_at: "2013-06-29", text:
14 "Livro sobre a linguagem de programação **Ruby**.\n\n...", value:
15 #<BigDecimal:31c53b0,'0.1E1',9(18)>, person_id: 1, created_at: "2017-03-29
16 21:58:47", updated_at: "2017-03-29 22:11:14", stock: 7, lock_version: 3>
```

Ou utilizar um bloco com `with_lock`:

```
1 > b = Book.first
2 Book Load (0.2ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" ASC
3 LIMIT ?  [{"LIMIT": 1}]
4 => #<Book id: 1, title: "Conhecendo Ruby", published_at: "2013-06-29", text:
5 "Livro sobre a linguagem de programação **Ruby**.\n\n...", value:
6 #<BigDecimal:31f9f98,'0.1E1',9(18)>, person_id: 1, created_at: "2017-03-29
7 21:58:47", updated_at: "2017-03-29 22:11:14", stock: 7, lock_version: 3>
8
9 > b.with_lock { b.sell }
10 (0.1ms)  begin transaction
11 Book Load (0.1ms)  SELECT  "books".* FROM "books" WHERE "books"."id" = ? LIM\
12 IT
13 ?  [{"id": 1}, {"LIMIT": 1}]
14 Person Load (0.1ms)  SELECT  "people".* FROM "people" WHERE "people"."id" = ?
15 ORDER BY "people"."name" ASC LIMIT ?  [{"id": 1}, {"LIMIT": 1}]
16 SQL (0.3ms)  UPDATE "books" SET "stock" = 6, "updated_at" = '2017-03-29
17 22:53:48.781699', "lock_version" = 4 WHERE "books"."id" = ? AND
18 "books"."lock_version" = ?  [{"id": 1}, {"lock_version": 3}] (11.6ms)  commit
19 transaction
20 => true
```

O lock é liberado no final da transação.

E-mail

Agora vamos enviar um e-mail para o cliente, informando sobre o seu pedido que foi fechado.

Para enviarmos os e-mails, precisamos de uma configuração de conta de acordo com o ambiente em que vamos rodar o sistema. No nosso caso, é o ambiente de desenvolvimento (`development`), mas podemos customizar quaisquer dos ambientes. Os arquivos de configurações dos ambientes ficam em `config/environments/`. Vamos configurar o arquivo `development.rb`, para enviar os e-mails através de uma conta do Gmail:

```
1 config.action_mailer.delivery_method = :smtp
2 config.action_mailer.smtp_settings = {
3   address: 'smtp.gmail.com',
4   port: 587,
5   authentication: 'plain',
6   user_name: ENV['MAIL_USER'],
7   password: ENV['MAIL_PWD'],
8   enable_starttls_auto: true
9 }
```

Variáveis de ambiente

Reparam como no arquivo acima carreguei o usuário e senha de variáveis de ambiente através de `ENV`. Não é uma boa deixar dados de autenticação em arquivos, sendo melhor configurar os dados de acesso no servidor. Para configurar essas variáveis, existem vários meios, mas um bem prático é criar um arquivo novo em `/etc/profile.d` com as declarações necessárias, como por exemplo criando o arquivo `bookstore.sh` com o seguinte conteúdo:

```
1 export MAIL_USER=bookstore
2 export MAIL_PWD=senhasecretadoemail
```

Os arquivos ali configurados são executados sempre que um *shell* de *login* é executado. Após configurado, é bom reiniciar alguns serviços como o servidor web e a própria aplicação, para que peguem as variáveis configuradas no arquivo.

Um problema que isso pode gerar é quando ou não temos acesso ao servidor ou temos várias aplicações rodando no mesmo servidor. Lógico que podemos diferenciar os nomes das variáveis de ambiente como `APP1_MAIL_USER` e `APP2_MAIL_USER`, por exemplo, até colocando cada uma em um arquivo separado, mas acaba ficando um tipo de configuração complicada se não tomar cuidado. Podemos utilizar algumas *gems* para gerenciar isso, como por exemplo, a [Figaro](#)¹²². Vamos inserir ela no `Gemfile` e rodar o `bundler`:

¹²²<https://github.com/laserlemon/figaro>

```
1 gem 'figaro'  
2  
3 $ bundle  
4 Installing figaro 1.1.1
```

Agora vamos instalar:

```
1 bundle exec figaro install  
2     create config/application.yml  
3     append .gitignore
```

Reparem que foi criado um arquivo chamado `application.yml` no diretório `config`, e foi automaticamente adicionado no `.gitignore` para que ele não seja controlado pelo `Git`, ou seja, não vamos controlar versões de um arquivo lotado de senhas, enviando ele para o repositório. **Não façam isso!**

Dentro do arquivo podemos configurar dessa forma:

```
1 MAIL_USER: bookstore  
2 MAIL_PWD: senhasecretadoemail
```

Essas variáveis vão ser carregadas na Hash ENV e vão estar disponíveis do mesmo modo que as variáveis normais de ambiente.

Uma opção é até preencher esse arquivo com senhas para os ambientes de desenvolvimento e teste, pois ele aceita as configurações por ambiente, e deixarmos as senhas de **produção** com algo inválido, preenchendo depois com os valores corretos no servidor, e fazendo com que esse arquivo não seja enviado na atualização ou não seja sobreescrito quando a aplicação for atualizada. Podemos deixar ele assim:

```
1 development:  
2     MAIL_USER: user@gmail.com  
3     MAIL_PWD: senhasecretadogmail  
4  
5 test:  
6     MAIL_USER: teste  
7     MAIL_PWD: senhadeteste  
8  
9 production:  
10    MAIL_USER: mepreenchanoservidor!  
11    MAIL_PWD: mepreenchanoservidor!
```

Se você está usando o seu usuário e senha de email para os ambientes de desenvolvimento e teste, pode deixar somente com o ambiente de produção configurado ali e deixar as duas variáveis declaradas, por exemplo, no seu arquivo `~/.bashrc`, afinal, não vai querer que essa informação também vá parar no seu repositório!

Após configurados os ambientes, temos que reiniciar o servidor para carregar as configurações.

Mailers

Agora vamos precisar criar um Mailer, que é um tipo de arquivo que fica em `app/mailers/` e contém métodos que correspondem à *templates* de e-mails, que são renderizados utilizando *views*.

Vamos criar um Mailer chamado `OrderMailer`, com um método chamado `created`, que vai ser acionado quando uma ordem for inserida no sistema:

```
1 $ rails g mailer OrderMailer created
2       create  app/mailers/order_mailer.rb
3       invoke  erb
4       create    app/views/order_mailer
5   identical  app/views/layouts/mailер.text.erb
6   identical  app/views/layouts/mailер.html.erb
7       create    app/views/order_mailer/created.text.erb
8       create    app/views/order_mailer/created.html.erb
9   ...
```

Podemos ver que foram criados os arquivos `order_mailer.rb`, em `app/mailers`, os layouts em `app/views/layouts` e as “*views*” `created.text.erb` e `created.html.erb` em `app/views/order_mailer`.

Prestem atenção na hora de criar os seus Mailers, pois alguns nomes podem conflitar com nomes de modelos, controladores e consequentemente, das *views* utilizadas pelos Mailers e pelos outros. É uma boa sempre terminar um Mailer com ... Mailer. Vamos dar uma olhada e adaptada no arquivo do Mailer:

```
1 class OrderMailer < ApplicationMailer
2   default from: 'rails@bluefish.com.br'
3
4   def created(order)
5     @order = order
6     mail to: @order.person.email, subject: "Pedido #{@order.id} recebido!"
7   end
8 end
```

Temos que fazer a *view* correspondente para renderizar o texto do email, primeiro com a opção de quando o email é enviado em texto puro, em `app/views/order_mailer/created.text.erb`:

```
1 Olá, <%= @order.person.name %>!
2
3 Seu pedido foi recebido em nosso sistema. Anote o número: <%= @order.id %>
4
5 Os items comprados foram:
6
7 <% for item in @order.order_items %>
8 - <%= item.book.title %> - <%= item.quantity %> - <%= number_to_currency(item\
9 .value) %> - <%= number_to_currency(item.quantity * item.value) %>
10 <% end %>
11
12 Obrigado por fazer seu pedido conosco!
```

E agora uma view para enviar o email em formato HTML, no arquivo app/views/order_mailer/created.html.erb. No caso de existirem ambos os arquivos (podemos apagar qual não queremos), o email vai ser enviado com os dois formatos, sendo que o cliente de email seleciona o suportado:

```
1 <h1>Bookstore</h1>
2 <h2>Recebemos o seu pedido!</h2>
3
4 <p>
5   Olá, <%= @order.person.name %> !
6 </p>
7
8 <p>
9   Seu pedido foi recebido em nosso sistema. Anote o número: <strong><%= @orde\
10 r.id %></strong>
11 </p>
12
13 <p>
14   Os items comprados foram:
15 </p>
16
17 <table>
18   <thead>
19     <tr>
20       <th>Título</th>
21       <th>Quantidade</th>
22       <th>Valor</th>
23       <th>Total</th>
24     </tr>
25   </thead>
26   <tbody>
27     <% for item in @order.order_items %>
28       <tr>
```

```
29      <td><%= item.book.title %></td>
30      <td><%= item.quantity %></td>
31      <td><%= number_to_currency(item.value) %></td>
32      <td><%= number_to_currency(item.quantity * item.value) %></td>
33  </tr>
34  <% end %>
35  </tbody>
36  <tfoot>
37    <tr>
38      <td colspan='3'>Total</td>
39      <td><%= number_to_currency(@order.total) %></td>
40    </tr>
41  </tfoot>
42 </table>
```

Agora no controlador, jinserindo o código que vai criar o Mailer (com `created`) e disparar o email (com `deliver_now`):

```
1 def close_order
2   ...
3
4   @cart.clear
5   OrderMailer.created(@order).deliver_now
6   redirect_to order_path(@order)
7 end
```

Quando o pedido é fechado, o email é enviado como no exemplo (reparem os caracteres acentuados já devidamente “escapados”):

```
1 Ol=C3=A1, Eust=C3=A1quio Rangel de Oliveira Jr.!
2
3 Seu pedido foi recebido em nosso sistema. Anote o n=C3=BAmero: 7
4
5 Os items comprados foram:
6
7 - Conhecendo Ruby - 1 - R$ 1,00 - R$ 1,00
8 - Conhecendo o Git - 1 - R$ 10,00 - R$ 10,00
9
10 Obrigado por fazer seu pedido conosco!
```

O resultado do email em formato HTML seria algo como:

Bookstore

Recebemos o seu pedido!

Olá, Eustáquio Rangel de Oliveira Jr.!

Seu pedido foi recebido em nosso sistema. Anote o número: **7**

Os items comprados foram:

Título	Quantidade	Valor	Total
Conhecendo Ruby 1		R\$ 1,00	R\$ 1,00
Conhecendo o Git 1		R\$ 10,00	R\$ 10,00
Total			R\$ 11,00

Obrigado por fazer seu pedido conosco!

[Email em HTML](#)

Anexos

Para anexar arquivos no email, podemos utilizar a Hash attachments do *mailer*, bastando dar identificador e conteúdo para o arquivo enviado em anexo. Por exemplo, podemos enviar o logotipo da livraria em anexo utilizando, no método created do OrderMailer visto acima, indicando o caminho do arquivo:

```
1 class OrderMailer < ApplicationMailer
2   default from: 'carteiro@bluefish.com.br'
3
4   def created(order)
5     attachments['logo.png'] = logo
6     @order = order
7     mail to: @order.person.email, subject: "Pedido #{@order.id} recebido!"
8   end
9
10  def logo
11    File.read("#{Rails.root}/app/assets/images/logo.png")
12  end
13 end
```

Dessa maneira a imagem irá como um anexo separado do corpo da mensagem. Para fazer um anexo *inline*, ou seja, dentro da própria mensagem, podemos fazer da seguinte forma, apenas inserindo, no *mailer*, o método *inline*:

```
1 def created(order)
2   attachments.inline['logo.png'] = logo
3   @order = order
4   mail to: @order.person.email, subject: "Pedido #{@order.id} recebido!"
5 end
```

E na view HTML utilizamos attachments com o valor da chave e o método url como parâmetro para image_tag, que vai renderizar a imagem no local indicado.

```
1 <%= image_tag attachments['logo.png'].url %>
2
3 <h1>Bookstore</h1>
4 <h2>Recebemos o seu pedido!</h2>
5 ...
```

Apesar que, a não ser que você tenha uma razão muito boa para enviar uma imagem anexada *inline* em um email, o procedimento mais costumeiro é enviar um elemento IMG com o atributo SRC apontando para uma URL onde se encontra a imagem em questão.

Utilizando enums

Temos um meio especial de controlar alguns atributos que são inteiros no Rails. São os enums, que podemos definir como utilizando valores inteiros e representados pelos seus nomes, além de ganhar alguns métodos e escopos para lidar com esses atributos.

Vamos imaginar que precisamos de um atributo para controlar o status do pedido, que nesse ponto, após o envio do email, precisa ser alterado. Podemos ter um atributo chamado `status`, que poderá ser representado como `mailed`, `delivered` e `returned`, para indicar alguns estados possíveis para o pedido. Se precisarmos de mais informações a partir desse estado, além de uma representação como inteiro (para o banco de dados) e como String/Symbol (para a interface), como, por exemplo, marcar a data de quando o estado foi alterado, nesse caso seria melhor fazer um outro modelo chamado `OrderState`, indicar que `Order` tem muitos `OrderState`, e armazenar as informações extras lá. Mas vamos partir do princípio aqui que só precisamos saber o estado corrente do pedido.

Para isso, vamos criar uma coluna nova no modelo `Order`, utilizando um número inteiro na migration:

```

1 $ rails g migration AddStatusToOrder status:integer
2   Running via Spring preloader in process 22605
3     invoke  active_record
4       create    db/migrate/20170806153000_add_status_to_order.rb
5 ...
6 $ rails db:migrate
7
8 == 20170806153000 AddStatusToOrder: migrating ===== \\
9 ==
10 -- add_column(:orders, :status, :integer)
11 -> 0.0012s
12 == 20170806153000 AddStatusToOrder: migrated (0.0013s)
13 =====

```

E vamos indicar um enum no modelo, dessa forma:

```

1 class Order < ApplicationRecord
2 ...
3   enum status: [:mailed, :delivered, :canceled]
4 ...

```

Os valores a serem preenchidos no banco de dados **levam em conta a ordem** que foram escritos, baseados em 0. Assim, no exemplo acima, :mailed vale 0, :delivered vale 1 e :canceled vale 2. Se precisarmos especificar valores diferentes dos automáticos, podemos utilizar essa forma:

```
1 enum status: {mailed: 10, delivered: 20, canceled: 30}
```

Mas vamos utilizar na forma padrão mesmo, como demonstrador originalmente. A partir desse ponto, ganhamos alguns métodos e um escopo. Antes de mais nada, temos o método que retorna o atributo:

```

1 > o = Order.first
2 > o.status
3 => nil

```

O escopo, que já filtra nossos pedidos:

```

1 > Order.mailed
2   Order Load (0.3ms)  SELECT "orders".* FROM "orders" WHERE "orders"."status" =
3   " = "
4 ? LIMIT ?  [["status", 0], ["LIMIT", 11]]
5 => #<ActiveRecord::Relation []>

```

Um método para verificar um determinado status:

```

1 > o.mailed?
2 => false

```

E um método para alterar o status para o que quisermos, que podemos inclusive já utilizar quando o job do email enviado for executado:

```

1 > o.mailed!
2   (0.2ms) begin transaction
3 Person Load (0.3ms) SELECT "people".* FROM "people" WHERE "people"."id" = \
4 ?
5 ORDER BY "people"."name" ASC LIMIT ? [[{"id": 1}, {"LIMIT": 1}]] SQL (0.2ms)
6 UPDATE "orders" SET "updated_at" = ?, "status" = ? WHERE "orders"."id" = ?
7 [{"updated_at": "2017-08-06 15:45:01.858267"}, {"status": 0}, {"id": 2}]
8 (3.8ms) commit transaction
9 => true
10
11 > o.mailed?
12 => true
13
14 > Order.mailed
15 Order Load (0.3ms) SELECT "orders".* FROM "orders" WHERE "orders"."status" \
16 " =
17 ? LIMIT ? [{"status": 0}, {"LIMIT": 11}]
18 => #< ActiveRecord::Relation [<Order id: 2, person_id: 1, created_at: \
19 "2017-08-06 15:41:09", updated_at: "2017-08-06 15:45:01", status: "mailed">>]

```

Para comparações, é mais prático utilizar os métodos disponibilizados com o ? no final, pois quando requistamos o atributo do modelo, ele vai ser automaticamente traduzido como uma String:

```

1 > o.status
2 => "mailed"
3
4 > o.status == 0
5 => false
6
7 > o.status == :mailed
8 => false
9
10 > o.status == :mailed.to_s
11 => true

```

Podemos utilizar outro método criado, que é a representação pluralizada dos valores do enum, nesse caso, statuses, para fazer essa comparação:

```

1 > Order.statuses
2 => {"mailed"=>0, "delivered"=>1, "canceled"=>2}
3
4 > o.status == Order.statuses.key(0)
5 => true

```

Isso facilita para criarmos seletores nas nossas *views* (mostrando aqui no exemplo uma que não existe, já que só temos o modelo de *Order*):

```

1 <div class="field">
2   <%= form.label :status %>
3   <%= form.select :status, Order.statuses %>
4 </div>

```

Lembrando que do jeito que utilizamos os `enums` acima, também ganhamos os outros métodos de acordo com os outros símbolos utilizados:

- `Order.delivered`
- `delivered?`
- `delivered!`
- `Order.canceled`
- `canceled?`
- `canceled!`

Criando um job para processamento em background

Um ponto importante a notar é que a aplicação fica “parada” enquanto o método `deliver` acima é executado, pois está conectando no servidor de emails e enviando o email. Precisamos indicar que queremos que esse comportamento seja executado no *background*, de forma a não influenciar na experiência do usuário que está utilizando a aplicação.

A partir do Rails 4.2, temos disponível o `ActiveJob`¹²³, um recurso perfeito para o que precisamos fazer. Primeiro, podemos deixar explícito que vamos criar um *job*, separado do processamento principal, que vai ser utilizado para enviar os emails. Para isso, podemos apenas alterar no controlador o código de

```
1 OrderMailer.created(@order).deliver_now
```

para

¹²³<https://github.com/rails/rails/tree/master/activejob>

```
1 OrderMailer.created(@order).deliver_later
```

Se fechamos um novo pedido agora, vamos ver algo como:

```
1 [ActiveJob] Enqueued ActionMailer::DeliveryJob (Job ID: 36170cd0-16e6-46ea-b2\  
2 e6-d7ad6be51f2f) to Inline(mailers) with arguments: "OrderMailer", "created", \  
3 "deliver_now", gid://bookstore/Order/10  
4 [ActiveJob] Order Load (0.2ms) SELECT "orders".* FROM "orders" WHERE "ord\  
5 ers"."id" = ? LIMIT 1 [["id", 10]]  
6 [ActiveJob] [ActionMailer::DeliveryJob] [36170cd0-16e6-46ea-b2e6-d7ad6be51f2f\  
7 ] Performing ActionMailer::DeliveryJob from Inline(mailers) with arguments: "\\  
8 OrderMailer", "created", "deliver_now", gid://bookstore/Order/10
```

Ali foi criado um *job* automaticamente (onde também poderíamos ter utilizado `deliver_now` para executar o *job* imediatamente, sem utilizar o `ActiveJob`), mas vamos deixar explícito e criar um *job* para processar os emails dos pedidos da seguinte forma:

```
1 $ rails g job OrderCreatedEmail  
2       invoke test_unit  
3       create test/jobs/order_created_email_job_test.rb  
4       create app/jobs/order_created_email_job.rb
```

Dando uma olhada no conteúdo gerado de `order_created_email_job.rb`:

```
1 class OrderCreatedEmailJob < ActiveJob::Base  
2   queue_as :default  
3  
4   def perform(*args)  
5     # Do something later  
6   end  
7 end
```

Podemos adaptá-lo da seguinte forma:

```
1 class OrderCreatedEmailJob < ActiveJob::Base  
2   queue_as :default  
3  
4   def perform(order)  
5     OrderMailer.created(order).deliver_later  
6   end  
7 end
```

E no controlador, inserir o seguinte código, para disparar o *job*, utilizando `OrderCreatedEmailJob.perform_later`:

```
1 def close_order
2   ...
3   OrderCreatedEmailJob.perform_later(@order)
4   redirect_to order_path(@order)
5 end
```

Também podemos utilizar opções para especificar quando o *job* deve ser executado, como por exemplo:

```
1 OrderCreatedEmailJob.set(wait_until: Date.tomorrow.noon).perform_later(@order)
2 OrderCreatedEmailJob.set(wait: 1.week).perform_later(@order)
```

Mas ainda temos a “paradinha” na aplicação, pois criamos um *job*, indicando a *intenção* de o executarmos em *background*, o que efetivamente, ainda não fizemos.

Enviando o processamento para background

DelayedJob

Para evitar o atraso dos emails, podemos utilizar, como uma opção simples (existem outras mais sofisticadas) a gem `DelayedJob`¹²⁴, que envia a execução para o *background*, de maneira bem fácil. Primeiro inserimos a gem específica e a daemons no nosso Gemfile:

```
1 gem 'daemons'  
2 gem 'delayed_job_active_record'
```

Rodar o bundler:

```
1 ...  
2 Installing delayed_job 4.1.2  
3 Installing delayed_job_active_record 4.1.1  
4 ...
```

Rodar o generator e rake db:migrate:

```
1 $ rails g delayed_job:active_record  
2       create  bin/delayed_job  
3       chmod  bin/delayed_job  
4       create  db/migrate/20170330214209_create_delayed_jobs.rb  
5  
6 $ rails db:migrate  
7 == 20170330214209 CreateDelayedJobs: migrating =====\br/>8 ==  
9 -- create_table(:delayed_jobs, { :force=>true })  
10    -> 0.0061s  
11 -- add_index(:delayed_jobs, [ :priority, :run_at ], { :name=>"delayed_jobs_prior"\br/>12   ity })  
13    -> 0.0018s  
14 == 20170330214209 CreateDelayedJobs: migrated (0.0081s) =====\br/>15 ==
```

Para maiores informações, consulte as informações da gem, mas atualmente para enviar o processamento para *background* (disparando o *script* do DelayedJob para executar os processos) é só indicar isso em um *initializer* que vamos criar em `config/initializers/active_job.rb`, com o seguinte conteúdo:

¹²⁴https://github.com/collectiveidea/delayed_job

```
1  ActiveJob::Base.queue_adapter = :delayed_job
```

Uma dica é que o DelayedJob serializa o que foi enviado, então no caso acima seria melhor enviar o id da Order para que ela fosse recuperada no momento em que created for executado, reduzindo o tamanho do que foi serializado.

Já vamos resolver isso, mas primeiro vamos alterar o método `deliver_later` para `deliver_now`, já que estamos com um *backend* já configurado:

```
1  class OrderCreatedEmailJob < ActiveJob::Base
2    queue_as :default
3
4    def perform(order)
5      OrderMailer.created(order).deliver_now
6    end
7  end
```

Nesse ponto, após reiniciarmos o servidor para ter certeza que tudo está ok, podemos notar, após fechar um novo pedido, que a aplicação respondeu mais rápido mas o email ainda não foi disparado. Isso porque precisamos disparar o *daemon* do DelayedJob:

```
1  $ ./bin/delayed_job start
```

Podemos utilizar `run` ao invés de `start` para deixarmos a aplicação visível e rodando no terminal:

```
1  $ ./bin/delayed_job run
2  delayed_job: process with pid 17470 started.
```

E também temos as seguintes opções:

```
1  $ ./bin/delayed_job
2  ERROR: no command given
3
4  Usage: delayed_job <command> <options> -- <application options>
5
6  * where <command> is one of:
7    start          start an instance of the application
8    stop           stop all instances of the application
9    restart        stop all instances and restart them afterwards
10   reload         send a SIGHUP to all instances of the application
11   run            start the application and stay on top
12   zap            set the application to a stopped state
13   status         show status (PID) of application instances
14
15  * and where <options> may contain several of the following:
```

```

16      -t, --ontop           Stay on top (does not daemonize)
17      -s, --shush            Silent mode (no output to the terminal)
18      -f, --force             Force operation
19      -n, --no_wait           Do not wait for processes to stop
20
21
22 Common options:
23     -h, --help              Show this message
24     --version               Show version

```

É importante estar com tudo configurado (acessos ao banco de dados, servidor de email, etc.) configurados onde vai ser executado o processo do `DelayedJob`. Por padrão, a lista de processos vai ser verificada a cada 5 segundos.

Refatorando o processamento no background

Agora vamos resolver o problema que encontramos acima quando utilizamos `OrderMailer.delay.created(@order)` pois estamos enviando para o processamento em *background* uma instância completa de `Order`, que vai ser serializada por completo, e isso não é legal pois podemos fazer de um jeito mais otimizado.

Para resolver isso podemos alterar esse código no *job* para:

```

1 def perform(id)
2   OrderMailer.created(id).deliver_now
3 end

```

Nesse caso, somente o `id` de um pedido (um inteiro, e não mais uma instância do objeto) é enviado para o método `perform` do *job* e também para o método `created` do *mailer*, que também tem que ser alterado nesse caso:

```

1 def created(id)
2   attachments.inline['logo.png'] = logo
3   @order = Order.find(id)
4   ...

```

Vamos precisar alterar também o código que chama o *job*, no controlador `pub`, de:

```
1 OrderCreatedEmailJob.perform_later(@order)
```

para

```
1 OrderCreatedEmailJob.perform_later(@order.id)
```

Agora somente o `id` vai ser enviado para qualquer processo que estiver rodando em *background*.

Resque

Se quisermos uma solução mais potente e parruda para processos em *background*, podemos fazer uso do [Resque](#)^{125 126}, que é baseado no [Redis](#)^{127 128}, que é uma solução de armazenamento de chave-valor. Para instalar o Redis no Ubuntu, podemos utilizar:

```
1 $ sudo apt-get install redis-server
```

E após terminado, verificar se o serviço está rodando:

```
1 $ redis-cli ping
2 PONG
```

Após isso, vamos inserir a gem do Resque no `Gemfile` e rodar o `bundler`:

```
1 gem 'resque'
2 ...
3 $ bundle
4 Installing redis 3.0.7
5 Installing resque 1.25.2
```

Podemos listar os *workers* do Resque pela linha de comando:

```
1 $ resque list
2 None
```

Ou disparar um servidor web que pode ser acessado na porta 5678 do localhost, no navegador:

```
1 $ resque-web
2 [2014-05-13 18:47:29 -0300] Starting 'resque-web'...
3 [2014-05-13 18:47:29 -0300] trying port 5678...
```

Nesse caso, é bom utilizar algum esquema de autenticação para evitar que algum curioso possa ficar xeretando por ali.

Outras opções do comando `resque` são:

¹²⁵ <https://github.com/resque/resque>

¹²⁶ <https://github.com/resque/resque>

¹²⁷ <http://redis.io/>

¹²⁸ <http://redis.io/>

```

1 $ resque -h
2 Usage: resque [options] COMMAND
3
4 Options:
5   -r, --redis [HOST:PORT]      Redis connection string
6   -N, --namespace [NAMESPACE] Redis namespace
7   -h, --help                   Show this message
8
9 Commands:
10  remove WORKER    Removes a worker
11  kill WORKER     Kills a worker
12  list            Lists known workers

```

Vamos adicionar tarefas para o Rake criando o arquivo lib/tasks/resque.rake:

```

1 require 'resque/tasks'
2
3 task "resque:setup" => :environment

```

O que nos dá as seguintes tarefas:

```

1 $ rails -T | grep resque
2 rails resque:failures:sort          # Sort the 'failed' queue for the re\
3 dis_multi_queue failure backend
4 rails resque:work                  # Start a Resque worker
5 rails resque:workers               # Start multiple Resque workers

```

Agora só precisamos alterar o nosso *initializer* em config/initializers/active_job.rb de

```
1 ActiveJob::Base.queue_adapter = :delayed_job
```

para

```
1 ActiveJob::Base.queue_adapter = :resque
```

Para disparar os emails, temos que executar os *workers* do Resque, e para executá-los em qualquer *queue*, podemos utilizar:

```
1 $ rails resque:work QUEUE='*' 
```

A partir desse momento, os emails dos pedidos começam a ser enviados.

É muito importante que você dispare novamente os *workers* sempre que fizer alguma alteração no seu código, pois senão elas não podem ter efeito. O Resque tem várias opções de configurações, *plugins* e outras *gems* que podem ser utilizadas em conjunto, para mais informações, visitem o site oficial ¹²⁹.



Dica

Em versões anteriores do Rails, precisávamos criar explicitamente os *workers*. O *ActiveJob* veio para funcionar como uma camada de abstração entre os vários *backends* que são utilizados, como pudemos ver só trocando de `:delayed_job` para `:resque` no *initializer*, não precisando de mais nada específico de cada um.

Sidekiq

Como código sempre evolui e dá idéias para melhorias ou criação de outros códigos derivados, temos sempre que ficar atentos para ferramentas legais que podem surgir por aí.

O Resque é uma ótima ferramenta, é ainda bastante utilizado, mas hoje também temos uma ferramenta muito boa para processamento em *background* que é o Sidekiq^{130 131}, onde o processamento é mais rápido e feito em várias *threads*, e também utiliza o redis e o conceito de *workers*. Como não vamos mais utilizar, podemos apagar o arquivo da *task* em `lib/tasks/resque.rake`, para não influenciar nos outros processos.

Para instalar o Sidekiq, vamos adicionar a gem no `Gemfile` e rodar o `bundler`:

```
1 ...
2 gem 'sidekiq'
3 ...
4 $ bundle install
5 ...
6 Installing sidekiq 3.2.5
7 ...
```

Como ele também usa o Redis, valem as mesmas instruções para instalação apresentadas acima para o Resque, trocando no *initializer* para:

```
1 ActiveJob::Base.queue_adapter = :sidekiq
```

E agora disparar o Sidekiq no diretório raiz da aplicação:

¹²⁹ <https://github.com/resque/resque>

¹³⁰ <http://sidekiq.org/>

¹³¹ <http://sidekiq.org/>

```

1 $ bundle exec sidekiq
2
3      s
4      ss
5      sss  sss      ss
6      s  sss s  ssss sss  _____ - - - - -
7      s      sssss ssss  / ____|(_)_| | ____| | _(_)_| _ -
8      s      sss      \_ _ \|/_ ^ \|/_ \_ \|/_ / \|/_ ^ |
9      s  sssss s      ____| | | (_| | __/ <| | (_| |
10     ss      s  s    |____/|_| \_,,_|_\_|\_|\_|\_|\_|\_|\_|
11     s      s s
12      s s
13      sss
14      sss
15
16 2014-09-11T13:53:45.054Z 17869 TID-3vozc INFO: Running in ruby 2.1.2p95 (2014\-
17 -05-08 revision 45877) [x86_64-linux]
18 2014-09-11T13:53:45.054Z 17869 TID-3vozc INFO: See LICENSE and the LGPL-3.0 f\
19 or licensing details.
20 2014-09-11T13:53:45.054Z 17869 TID-3vozc INFO: Upgrade to Sidekiq Pro for mor\
21 e features and support: http://sidekiq.org/pro
22 2014-09-11T13:53:45.054Z 17869 TID-3vozc INFO: Starting processing, hit Ctrl-\
23 C to stop
24 2014-09-11T13:53:45.114Z 17869 TID-md9ag INFO: Booting Sidekiq 3.2.5 with red\
25 is options {}
26 2014-09-11T13:53:45.114Z 17869 TID-68fvk INFO: Booting Sidekiq 3.2.5 with red\
27 is options {}
28 ...

```

Quando fechamos um pedido na loja, podemos ver no terminal que ele é processado:

```

1 2014-09-11T14:02:49.923Z 17869 TID-5teaw OrderCreated JID-a6928f8e49807f251fc\-
2 0809b INFO: start
3 2014-09-11T14:02:59.250Z 17869 TID-5teaw OrderCreated JID-a6928f8e49807f251fc\-
4 0809b INFO: done: 9.328 sec

```

E o email chega corretamente. :-)

Podemos verificar o estado atual dos processos utilizando uma app feita em [Sinatra](#)¹³². Para isso, vamos incluir a gem no Gemfile e rodar o bundler (no caso aqui, a gem do Sinatra já estava instalada):

¹³²<http://www.sinatrab.com/>

```

1 gem 'sinatra', '>= 1.3.0', require: false
2 ...
3 $ bundle install
4 ...
5 Using sinatra 1.4.5
6 ...

```

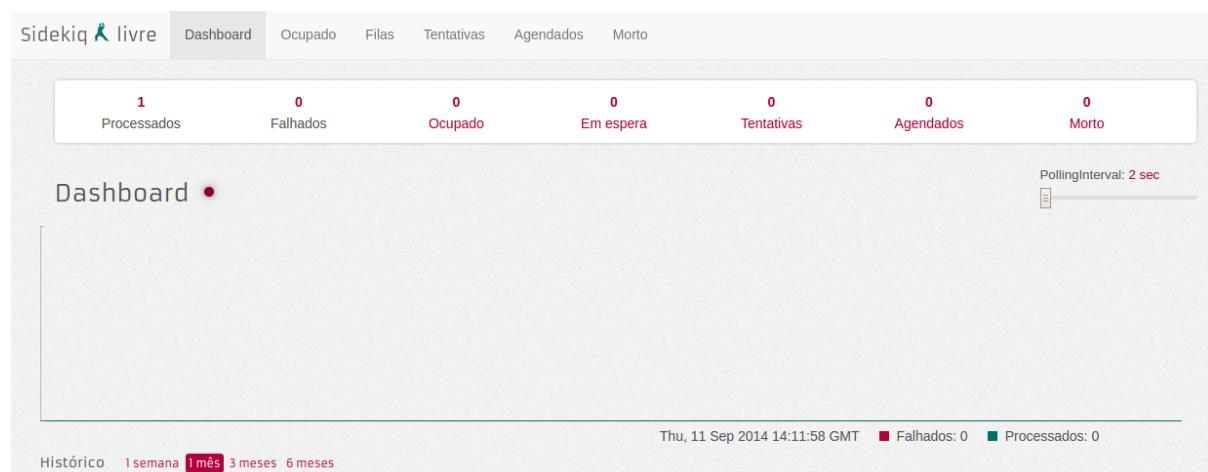
E agora criar uma rota para ela, adicionando o seguinte conteúdo no arquivo de rotas (`config/routes.rb`):

```

1 require 'sidekiq/web'
2
3 Sidekiq::Web.set :sessions, false
4
5 Rails.application.routes.draw do
6   mount Sidekiq::Web => '/sidekiq'
7 ...

```

Reparam que o `require` é **fora** do bloco, e o `mount` é **dentro**. Agora podemos abrir no navegador a URL `http://localhost:3000/sidekiq` que vai nos trazer uma tela como essa:



Monitor do Sidekiq

Aí é bom colocar algumas restrições em ambiente de produção para que curiosos não fiquem olhando as suas estatísticas. Maiores informações sobre isso podem ser vistas na [página da documentação sobre isso¹³³](#) e maiores informações sobre o resto das *features* do Sidekiq podem ser vistas no [índice da documentação do mesmo¹³⁴](#).

¹³³<https://github.com/mperham/sidekiq/wiki/Monitoring>

¹³⁴<https://github.com/mperham/sidekiq/wiki>

ActionCable

O que é

A partir do Rails 5, temos disponível o ActionCable, que é a implementação de websockets¹³⁵ presente no Rails. Pegando a definição da Wikipedia:

WebSocket é uma tecnologia que permite a comunicação bidirecional por canais full-duplex sobre um único soquete Transmission Control Protocol (TCP). Ele é projetado para ser executado em browsers e servidores web que suportem o HTML5, mas pode ser usado por qualquer cliente ou servidor de aplicativos. A API WebSocket está sendo padronizada pelo W3C[3] e o protocolo WebSocket está sendo padronizado pelo IETF.

Isso permite que as aplicações, rodando no navegador, possam fazer “assinaturas” em “canais” disponíveis para comunicação em tempo real quando os dados no servidor forem alterados, evitando aquele velho esquema de ficar recarregando a página ou disparando requisições assíncronas em Ajax de tempos em tempos para atualização.

Atualmente o ActionCable serve bem, mas se por acaso a sua aplicação for um monstro consumidor de mensagens, existem serviços como o Pusher¹³⁶ que fazem esse tipo de serviço de maneira bem eficiente.

Criando o canal

Vamos criar um canal com nome de orders para que o usuário da aplicação seja notificado com quaisquer mensagens que gostaríamos de enviar para ele sobre os seus pedidos, gerando uma notificação via uma DIV HTML no topo do página, que vai desaparecer após alguns segundos. Para criar o canal, vamos utilizar:

```
1 $ rails g channel orders
2 Running via Spring preloader in process 7240
3     create  app/channels/orders_channel.rb
4     identical app/assets/javascripts/cable.js
5     create  app/assets/javascripts/channels/orders.coffee
```

Criando o JavaScript

Vejam que foi gerado um arquivo com código Ruby chamado orders_channel.rb, foi verificado que o cable.js está idêntico, e gerado um com CoffeeScript chamado orders.coffee. Como não vamos programar com CoffeeScript, podemos apagar esse arquivo, vamos o substituir com um arquivo em JavaScript com o seguinte conteúdo:

¹³⁵ <https://pt.wikipedia.org/wiki/WebSocket>

¹³⁶ <https://pusher.com/>

```
1 var Bookstore = typeof Bookstore === "undefined" ? {} : Bookstore;
2
3 // aqui que é o handle do ActionCable, o resto é tudo firula. ;-)
4 Bookstore.Orders = function(order) {
5     this.order = order;
6
7     this.subscribe = function() {
8         App.orders = App.cable.subscriptions.create({channel: 'OrdersChannel', \
9             id: this.order}, {
10             received: function(data) {
11                 Bookstore.Orders.autohide_msg(data.msg);
12             }
13         });
14     };
15 };
16
17 // cria a mensagem
18 Bookstore.Orders.autohide_msg = function(msg) {
19     var div      = Bookstore.Orders.autohide_div();
20     var button   = Bookstore.Orders.autohide_button();
21
22     Bookstore.Orders.autohide_action(div);
23
24     div.append(msg);
25     div.append(button);
26     $("body").prepend(div);
27 };
28
29 // cria a ação de auto esconder
30 Bookstore.Orders.autohide_action = function(div) {
31     window.setTimeout(function() {
32         div.fadeTo(500, 0).slideUp(500, function(){
33             div.remove();
34         });
35     }, 3000);
36 };
37
38 // cria o botão para fechar
39 Bookstore.Orders.autohide_button = function() {
40     var button = $("<button/>");
41     button.addClass("close");
42     button.append("&times;");
43     button.click(function(evt) {
44         $(evt.target).parent().remove();
45     });
46     return button;
```

```
47 };
48
49 // cria a div
50 Bookstore.Orders.autohide_div = function() {
51     var div = $("<div>");
52     div.addClass("alert");
53     div.attr("role", "alert");
54     return div;
55 };
56
57 // retorna o id da URL
58 Bookstore.Orders.idFromURL = function() {
59     var id = parseInt(window.location.href.match(/(\/)(\d+)$/)[2]);
60     return id;
61 }
```

Vejam que a maior parte do código é para criar a DIV que vai mostrar a mensagem, que vai se remover após alguns segundos, com um botão para remover imediatamente. Se quisermos testar o funcionamento, podemos abrir o *console* JavaScript do navegador e digitar código como:

```
1 Bookstore.Orders.autohide_msg("Olá, esse é um teste!")
```

Vai ficar um lance bem feinho, então vamos colocar esse CSS no final de pub.scss que já deve melhorar um pouco (ei, eu disse um pouco!):

```
1 .alert {
2     border: 1px solid #d6e9c6;
3     background: #dff0d8;
4     color: black;
5     font-weight: bold;
6     padding: 1em;
7     z-index: 9999;
8     position: absolute;
9     margin: auto;
10    width: 50%;
11    text-align: center;
12    margin-left: 25%;
13    margin-top: 1em;
14
15    button {
16        margin-left: 1em;
17        border: 0;
18        background: transparent;
19        color: #3c763d;
20        font-weight: bold;
21    }
22 }
```

A partir desse momento, com o uso de

```
1 App.orders = App.cable.subscriptions.create({channel: 'OrdersChannel', id: th\
2 is.order}, {
```

a aplicação já assinou o canal, só falta enviar conteúdo para o canal.

Comunicando com o canal

Foi gerado um arquivo chamado `orders_channel.rb` em `app/channels`, vamos dar uma olhada no seu conteúdo:

```
1 class OrdersChannel < ApplicationCable::Channel
2   def subscribed
3     # stream_from "some_channel"
4   end
5
6   def unsubscribed
7     # Any cleanup needed when channel is unsubscribed
8   end
9 end
```

Existem 2 eventos definidos lá: o `subscribed`, para quando o canal é assinado, e o `unsubscribed`, para quando a assinatura é cancelada. O método (comentado) `stream_from` permite que sejam disparadas mensagens com o nome de um canal (apesar de estarmos dentro da classe `OrdersChannel`), e seria útil no caso de uma comunicação mais geral com os assinantes do canal. Pensem assim: se utilizássemos ali `stream_from 'orders_channel'`, todos os usuários que assinaram o canal iriam receber *quaisquer mensagens que enviássemos para lá*, não importando qual pedido que estariam interessados. Dessa forma, todos receberiam informações dos pedidos de todos, e apesar de podermos limitar a *visualização* dessas informações, fazendo um filtro, não seria algo muito interessante.

Vamos alterar o código indicando que iremos enviar o `id` de um pedido específico para assinar o canal, juntamente com uma mensagem no logger para podermos acompanhar no terminal onde está sendo executado o servidor, utilizando `stream_for` juntamente com o objeto do pedido:

```
1 class OrdersChannel < ApplicationCable::Channel
2   def subscribed
3     Rails.logger.info "Subscribed to the orders channel, order #{params[:id]}"
4     stream_for Order.find(params[:id])
5   end
6
7   def unsubscribed
8     # Any cleanup needed when channel is unsubscribed
9   end
10 end
```

Tudo pronto para transmitir as mensagens, agora temos que indicar a assinatura do canal. Vamos fazer isso na página que mostra o pedido, após concluído, que é a *view* em `app/views/pub/order.html.erb`. Vamos adicionar no final dela:

```
1 <script charset="utf-8" type="text/javascript">
2 $(document).on('turbolinks:load', function() {
3   new Bookstore.Orders(Bookstore.Orders.idFromURL()).subscribe();
4 });
5 </script>
```

Agora o canal está assinado, com o id do pedido que foi gerado, presente na URL da página do pedido. Dando uma olhada no log:

```
1 Started GET "/cable" for 127.0.0.1
2 Started GET "/cable/" [WebSocket] for 127.0.0.1
3 Successfully upgraded to WebSocket (REQUEST_METHOD: GET, HTTP_CONNECTION: Upgr\ade, HTTP_UPGRADE: websocket)
4 Subscribed to the orders channel, order 8
5   Order Load (0.1ms)  SELECT "orders".* FROM "orders" WHERE "orders"."id" = \
6 ? LIMIT ?  [[{"id": 8}, {"LIMIT": 1}]]]
7 OrdersChannel is transmitting the subscription confirmation
8 OrdersChannel is streaming from orders:Z21k0i8vYm9va3N0b3J1L09yZGVyLzg
```

Só precisamos disparar as mensagens para o canal agora. Para isso, vamos alterar o nosso *job* de envio de emails para enviar uma mensagem para o canal criado logo após o email ser enviado, deixando o código em `app/jobs/order_created_email_job.rb` assim:

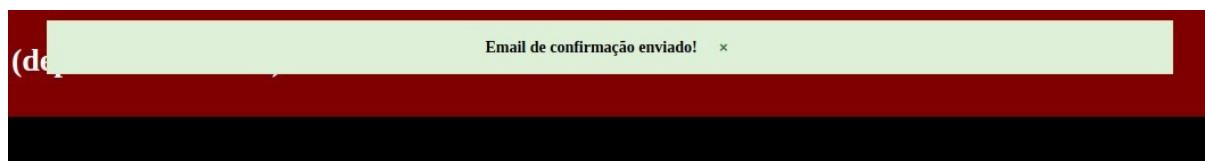
```
1 class OrderCreatedEmailJob < ApplicationJob
2   queue_as :default
3
4   def perform(order)
5     OrderMailer.created(order.id).deliver_now
6     OrdersChannel.broadcast_to(order, msg: 'Email de confirmação enviado!')
7   end
8 end
```

Vejam que utilizamos `OrdersChannel.broadcast_to` com dois valores: o pedido e uma *hash* com a mensagem que queremos consumir lá na função JavaScript `received`. Dessa forma, o ciclo de código se fechou, e já podemos fechar nossos pedidos, ficando na página e recebendo as notificações sem precisar disparar mais requisição alguma ou recarregar a página.

Só tem mais um probleminha. Quase todo esse código que fizemos vai rodar no processo do servidor, mas com certeza o *job* de envio do email vai rodar em outro processo, do *adapter* de envio para *background*, que no caso, se não alterarmos nada feito no último capítulo, vai ser o Sidekiq, e os dois (servidor e Sidekiq), digamos, não vão se comunicar e vamos precisar de uma “ponte” entre eles, tanto aqui em desenvolvimento como em produção. Essa ponte vai ser o Redis, que já foi mencionado como sendo utilizado por várias *gems*. Para configurarmos o uso do Redis, podemos alterar no arquivo `config/cable.yml`:

```
1 redis: &redis
2   adapter: redis
3   url: redis://localhost:6379/1
4
5 development: *redis
6 test: *redis
7 production: *redis
```

Agora sim! Reiniciando o servidor e o Sidekiq, alguns segundos após fechar um pedido, temos algo como:



Aviso de envio de email

Vamos aproveitar que o Sidekiq é *multithreading* e fazer uma pequena baguncinha no *job*. Não deixem esse código lá, é só para mostrar como funciona o envio de várias mensagens:

```

1 class OrderCreatedEmailJob < ApplicationJob
2   queue_as :default
3
4   def perform(order)
5     OrderMailer.created(order.id).deliver_now
6     OrdersChannel.broadcast_to(order, msg: 'Email de confirmação enviado!')
7
8     # baguncinha
9     Kernel.sleep(5)
10
11    1.upto(3) do |n|
12      OrdersChannel.broadcast_to(order, msg: "Mensagem #{n}!")
13      Kernel.sleep(5)
14    end
15  end
16 end

```

Quando fechamos um novo pedido e aguardarmos alguns segundos, podemos ver a mensagem do email enviado, seguida de 3 novas mensagens. Em uma aplicação real, cada uma delas pode mostrar um determinado *status* de processamento do pedido.

Pusher

Vamos aproveitar que estamos com a mão na massa e verificar como funciona esse recurso com o [Pusher](#)¹³⁷. Uma parte importante é que ele pode ser utilizado em aplicações de versões anteriores do Rails, sem suporte ao ActionCable.

Primeiro temos que criar uma conta lá, e logo após criar a conta, já temos disponível a configuração de uma aplicação nova, através de algumas perguntas que são feitas. Vamos escolher o nome da aplicação, o *cluster* onde ela vai ficar (recomendável escolher um mais perto da sua localização), qual vai ser o tipo de *front-end* utilizado (vamos escolher o jQuery) e o *framework* (lógico, vamos escolher Rails). A partir disso já somos instruídos em como adaptar a nossa aplicação para enviar e receber as mensagens. Antes de mais nada, vamos inserir a *gem* pusher no Gemfile:

```

1 gem 'pusher'
2
3 $ bundle install
4 ...
5 Installing pusher 1.3.1
6 ...

```

Criar o arquivo config/initializers/pusher.rb com as configurações especificadas pela aplicação criada:

¹³⁷ <http://pusher.com>

```

1 require 'pusher'
2
3 Pusher.app_id = '<id>'
4 Pusher.key = '<key>'
5 Pusher.secret = '<secret>'
6 Pusher.cluster = '<cluster>'
7 Pusher.logger = Rails.logger
8 Pusher.encrypted = true

```

No *job* de envio de email, vamos alterar para:

```

1 class OrderCreatedEmailJob < ApplicationJob
2   queue_as :default
3
4   def perform(order)
5     OrderMailer.created(order.id).deliver_now
6     Pusher.trigger('orders-channel', 'email-event', {
7       message: 'Notificação de email enviado pelo Pusher!'
8     })
9   end
10 end

```

E agora consumir a mensagem na página de pedidos, primeiro alterando a página de layout pub.html.erb para incluir o JavaScript do Pusher:

```

1 ...
2   <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reloa\
3 d' %>
4   <script src="https://js.pusher.com/4.0/pusher.min.js"></script>
5 </head>
6 ...

```

E na página de pedidos (app/views/pub/order.html.erb) alterar o JavaScript para (trocando a <key> para a correta nas configurações da sua aplicação no Pusher!):

```

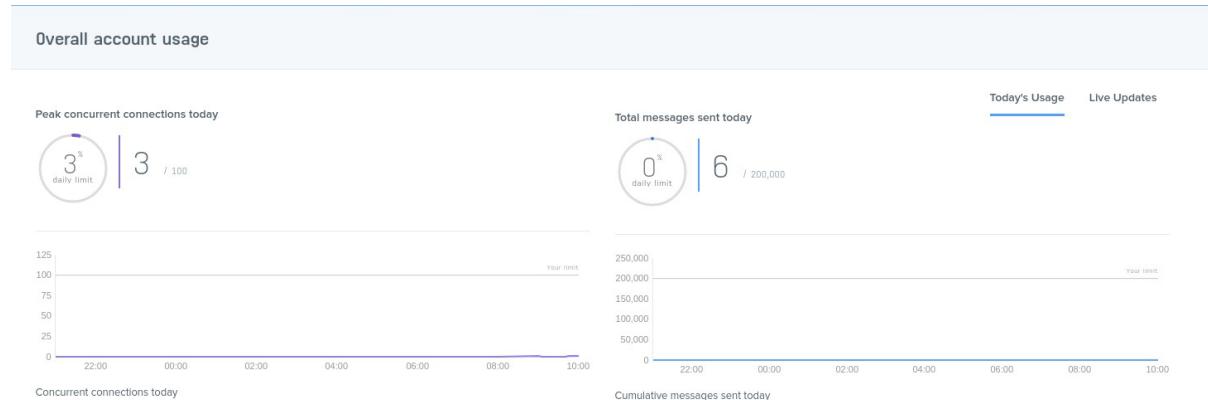
1 <script charset="utf-8" type="text/javascript">
2 $(document).on('turbolinks:load', function() {
3   var pusher = new Pusher('<key>', {
4     cluster: 'us2',
5     encrypted: true
6   });
7
8   var channel = pusher.subscribe('orders-channel');
9   channel.bind('email-event', function(data) {
10     Bookstore.Orders.autohide_msg(data.message)

```

```
11      });
12  });
13 </script>
```

Vejam que já aproveitei a classe que cria as mensagens que desaparecem automaticamente, só não assinei o canal do ActionCable, o que vai nos dar um resultado parecido com o anterior, somente agora com a mensagem diferente.

Após alguns pedidos fechados, podemos acompanhar na tela de estatísticas do Pusher:



Estatísticas do Pusher

Essa conta gratuita do Pusher, a sandbox, no presente momento nos permite enviar até 200.000 mensagens por dia, através de 100 conexões. Mais opções podem ser consultadas na [página de planos deles¹³⁸](#).

¹³⁸ <https://pusher.com/pricing>

Caching

Para melhorar a performance da nossa aplicação, podemos habilitar o recurso de *caching* em nossos controladores, de maneira que requisições que iriam para os controladores, que requisitariam informações para o banco e gerariam o *output* (fechando assim o ciclo MVC) não seriam mais geradas, e os resultados pegos direto de um arquivo gerado estaticamente.

O *caching* fica desativado por padrão em outros modos que não o de produção, mas podemos ativar com o seguinte comando:

```
1 $ rails dev:cache  
2 Development mode is now being cached.
```



Dica

Para desabilitar o caching em modo de desenvolvimento, é só executar o comando novamente.

Para utilizar *caching* do mesmo modo que o Rails 3.x, será necessário utilizar as *gems* `actionpack-page_caching`¹³⁹ e `actionpack-action_caching`¹⁴⁰. O Rails 4 usa o conceito de *Russian Doll Caching*, que pode ser habilitado no Rails 3.x com a *gem* `cache_digests`¹⁴¹. Se você tem curiosidade sobre como era e se pode ter alguma utilidade, continue lendo, senão pule direto para o *russian doll caching* logo abaixo.

Ativando o caching em páginas

Utilizando o comportamento do Rails 3.x, para páginas, temos duas opções de *caching*:

- `caches_page` - Ativa o *cache* da página
- `caches_action` - Ativa o cache da página, executando os filtros do controlador

Utilizando `caches_action` indica que, por exemplo, alguma ação que precise ser restrita através de algum filtro (lembrem-se dos *callbacks* para não deixar que um usuário não-autenticado não possa acessar determinado conteúdo?) é executada antes que o conteúdo do *cache* seja retornado para o usuário.

Para testar o recurso do *caching*, vamos habilitá-lo em modo de desenvolvimento, e especificar que o método `book` do controlador `pub` terá que gerar um arquivo de *cache*:

¹³⁹ https://github.com/rails/actionpack-page_caching

¹⁴⁰ https://github.com/rails/actionpack-action_caching

¹⁴¹ https://github.com/rails/cache_digests

```

1   class PubController < ApplicationController
2     caches_page :book
3     ...

```

Reiniciando o servidor e abrindo a página de algum livro no navegador e olhando no terminal onde roda o servidor, podemos ver algo como:

```

1 Started GET "/livro/1" for 127.0.0.1 at 2014-04-20 13:26:00 -0300
2 Processing by PubController#book as HTML
3   Parameters: {"id"=>"1"}
4
5     Book Load (0.2ms)  SELECT  "books".* FROM "books"  WHERE "books"."id" = ? L \
6   IMIT 1  [["id", 1]]
7     Person Load (0.2ms)  SELECT  "people".* FROM "people"  WHERE "people"."id" \ \
8   = ? ORDER BY "people"."name" ASC LIMIT 1  [["id", 1]]
9
10    Image Load (0.1ms)  SELECT  "images".* FROM "images"  WHERE "images"."image\\ \
11   able_id" = ? AND "images"."imageable_type" = ? LIMIT 1  [{"imageable_id": 1}, \
12   ["imageable_type", "Book"]]
13
14    Rendered pub/book.html.erb within layouts/pub (5.8ms)
15 Write page /home/taq/git/conhecendorails/bookstore/public/livro/1.html (0.2ms)
16 Completed 200 OK in 19ms (Views: 14.1ms | ActiveRecord: 0.8ms)

```

Podemos ver que o servidor indica que vai carregar a página do livro, onde é feita a busca no modelo para o `id` do livro desejado, é renderizada a `view` e criada a página de `cache` do livro em `/public/livro/1.html`. Antes de abrir a página no navegador, tínhamos:

```

1 $ ls public/
2 total 28K
3 drwxr-xr-x 4 taq taq 4,0K .
4 drwxr-xr-x 12 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 1,4K 404.html
6 -rw-r--r-- 1 taq taq 1,4K 422.html
7 -rw-r--r-- 1 taq taq 1,3K 500.html
8 -rw-r--r-- 1 taq taq 0 favicon.ico
9 drwxr-xr-x 4 taq taq 4,0K images
10 -rw-r--r-- 1 taq taq 202 robots.txt

```

Logo após carregada a página no navegador:

```
1 $ ls public/
2 total 32K
3 drwxr-xr-x 4 taq taq 4,0K .
4 drwxr-xr-x 12 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 1,4K 404.html
6 -rw-r--r-- 1 taq taq 1,4K 422.html
7 -rw-r--r-- 1 taq taq 1,3K 500.html
8 -rw-r--r-- 1 taq taq 0 favicon.ico
9 drwxr-xr-x 4 taq taq 4,0K images
10 drwxr-xr-x 2 taq taq 4,0K livro
11 -rw-r--r-- 1 taq taq 202 robots.txt
```

E no diretório public/livro:

```
1 $ ls public/livro/
2 total 12K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 4 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 2,3K 1.html
```

E, quando recarregamos a página, podemos notar que o servidor não executou todo o procedimento de consulta no banco e renderização, dependendo do servidor e da versão do Rails, podemos até ver um alerta do tipo:

```
1 cache: [GET /livro/1]
```

Ou seja, agora quem está sendo servido, é o arquivo HTML estático 1.html!

Limpando o cache

Podemos limpar o *cache* de duas maneiras: explicitamente ou automaticamente.

Limpando o cache explicitamente

Podemos especificar que queremos alterar o *cache* da nossa página de livros quando alterarmos o registro do livro, em seu próprio controlador:

```

1  class BooksController < ApplicationController
2    ...
3    after_action :expire_cache, only: [:update, :destroy]
4    ...
5    def expire_cache
6      expire_page controller: "pub", action: "book", id: @book.id
7    end
8  end

```

Isso deve gerar uma resposta no terminal do servidor como essa:

```

1 Expire page /home/taq/git/conhecendorails/bookstore/public/livro/1.html (0.1m\
2 s)

```

E apaga o arquivo que foi criado um pouco antes:

```

1 $ ls public/livro/
2 total 8,0K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 4 taq taq 4,0K ..

```

Além de `expire_page`, também temos `expire_action`, que funciona para o *cache* que criamos com `cache_action`, que faz o *caching* de uma página após passar pelos *callbacks* necessários do controlador.

Limpando o cache implicitamente

Podemos utilizar um recurso que temos em Rails chamado *sweepers* para fazer a limpeza do *cache* de maneira automática. Esse recurso também foi removido do Rails 4, e pode ser inserido novamente com a `gem rails-observers`¹⁴², inserida no `Gemfile`:

```

1 gem 'rails-observers'
2 ...
3
4 $ bundle
5 ...
6 Installing rails-observers 0.1.2
7 ...

```

Agora vamos comentar o `after_action` que acabamos de inserir no controlador dos livros e inserir o seguinte código em `app/sweepers/book_sweeper.rb`:

¹⁴²<https://github.com/rails/rails-observers>

```

1  class BookSweeper < ActionController::Caching::Sweeper
2    observe Book
3
4    def after_update(book)
5      expire_book_page(book)
6    end
7
8    def after_destroy(book)
9      expire_book_page(book)
10   end
11
12  private
13  def expire_book_page(book)
14    expire_page(controller: "pub", action: "book", id: book.id)
15  end
16 end

```

E, onde antes estava o `after_action` no controlador, expirando a página explicitamente, vamos indicar que desejarmos utilizar o *sweeper*:

```

1  class BooksController < ApplicationController
2    ...
3    #after_action :expire_cache, only: [:update, :destroy]
4    cache_sweeper :book_sweeper, only: [:update, :destroy]
5    ...

```



Dica

Podemos limpar os arquivos dos *caching* direto na linha de comando usando:

```
1 rails tmp:clear
```

Se algum teste funcional onde são utilizados `expire_page` quebrar, pode ser que não tenha configurado o controlador no teste, nesse caso podemos utilizar o seguinte código no teste funcional:

```

1  def expire_book_page(book)
2    @controller = ::ApplicationController.new if !@controller
3    expire_page(controller: "pub", action: "book", id: book.id)
4  end

```

Russian Doll caching

Para testar o jeito que o *caching* é feito no Rails 4, vamos comentar (ou remover) as linhas que criam o *cache* e ativam o *sweeper*, no controlador de livros, se por acaso você testou esses recursos:

```
1 class BooksController < ApplicationController
2 ...
3 # after_action :expire_cache, only: [:update, :destroy]
4 # cache_sweeper :book_sweeper, only: [:update, :destroy]
5 ...
```

e comentar (ou remover) no controlador público o *caches_page*:

```
1 class PubController < ApplicationController
2   # caches_page :book
3   ...
```

apagar o conteúdo do diretório public/livro (façam isso para não ficarem malucos procurando a causa do porque diabos a página do livro não é atualizada).

Agora vamos alterar a view book no controlador pub para utilizar o método *cache*, enviando a referência de um objeto:

```
1 <% cache @book do %>
2   <section id='book_detail'>
3     <h1><%= @book.title %></h1>
4     <h2><%= link_to @book.person.name, pub_author_path(@book.person) %></h2>
5     <%= image_tag(@book.image) %>
6     <%= markdown(@book.text) %>
7     <span class='book_value'><%= number_to_currency(@book.value) %></span>
8     <br/>
9
10    <% if @book.sold_out? %>
11      <p class="sold_out">Esgotado!</p>
12    <% else %>
13      <%= link_to 'Comprar', buy_path(@book), class: 'buy_book' %>
14    <% end %>
15  </section>
16 <% end %>
```

Agora vamos verificar se o *caching* está funcionando. Para verificar isso direto no log do servidor, vamos ter que colocar a seguinte linha nas configurações de desenvolvimento, no arquivo *config/environments/development.rb*:

```
1 config.action_controller.enable_fragment_cache_logging = true
```

Isso vai habilitar o log dos fragmentos, algo que por *default* veio desabilitado a partir do Rails 5.1.

Recarregando a página de um livro, vemos no servidor:

```
1 Started GET "/livro/1" for 127.0.0.1 at 2017-04-05 18:59:48 -0300
2   ActiveRecord::SchemaMigration Load (0.2ms)  SELECT "schema_migrations".* FR\OM "schema_migrations"
3 Processing by PubController#book as HTML
4   Parameters: {"id"=>"1"}
5   Book Load (0.2ms)  SELECT "books".* FROM "books" WHERE "books"."id" = ? LI\MIT ? [[{"id": 1}, {"LIMIT": 1}]]]
6   Rendering pub/book.html.erb within layouts/pub
7 Read fragment views/books/1-20170405215042451457/40ad3161c6ecc55ef972236f44ed\8312 (0.1ms)
8   Person Load (0.1ms)  SELECT "people".* FROM "people" WHERE "people"."id" =\? ORDER BY "people"."name" ASC LIMIT ? [[{"id": 1}, {"LIMIT": 1}]]]
9   Image Load (0.2ms)  SELECT "images".* FROM "images" WHERE "images"."imageable_id" = ? AND "images"."imageable_type" = ? LIMIT ? [[{"imageable_id": 1}, \["imageable_type": "Book"], {"LIMIT": 1}]]]
10 Write fragment views/books/1-20170405215042451457/40ad3161c6ecc55ef972236f44e\8312 (0.1ms)
11   Rendered pub/book.html.erb within layouts/pub (62.2ms)
```

Reparem ali no final, temos uma série de indicações de fragmentos. Recarregando a página novamente, temos o seguinte resultado no log:

```
1 Started GET "/livro/1" for 127.0.0.1 at 2017-04-05 19:01:08 -0300
2 Processing by PubController#book as HTML
3   Parameters: {"id"=>"1"}
4   Book Load (0.0ms)  SELECT "books".* FROM "books" WHERE "books"."id" = ? LI\MIT ? [[{"id": 1}, {"LIMIT": 1}]]]
5   Rendering pub/book.html.erb within layouts/pub
6   Read fragment views/books/1-20170405215042451457/40ad3161c6ecc55ef972236f44ed\8312 (0.1ms)
7   Rendered pub/book.html.erb within layouts/pub (1.9ms)
10 Completed 200 OK in 24ms (Views: 21.9ms | ActiveRecord: 0.0ms)
```

Se alterarmos algum atributo de Book, o atributo `updated_at` vai ser alterado, fazendo com que o *cache* seja invalidado, fazendo com que o fragmento seja gerado novamente:

```
1 > Book.first.update_attribute(:title, 'Conhecendo Ruby - Legal!')
2
3 Started GET "/livro/1" for 127.0.0.1
4 Processing by PubController#book as HTML
5   Parameters: {"id"=>"1"}
6   Book Load (0.2ms)  SELECT  "books".* FROM "books" WHERE "books"."id" = ? LI\
7 MIT ? [[{"id": 1}, {"LIMIT": 1}]]]
8     Rendering pub/book.html.erb within layouts/pub
9   Read fragment views/books/1-2017061923232854526/40ad3161c6ecc55ef972236f44ed\
10 8312 (0.1ms)
11   Person Load (0.8ms)  SELECT  "people".* FROM "people" WHERE "people"."id" = \
12 ? ORDER BY "people"."name" ASC LIMIT ? [[{"id": 1}, {"LIMIT": 1}]]]
13   Image Load (0.5ms)  SELECT  "images".* FROM "images" WHERE "images"."imageable_\
14 ble_id" = ? AND "images"."imageable_type" = ? LIMIT ? [[{"imageable_id": 1}, \
15 {"imageable_type": "Book"}, {"LIMIT": 1}]]]
16 Write fragment views/books/1-2017061923232854526/40ad3161c6ecc55ef972236f44e\
17 d8312 (0.2ms)
```

Agora vamos alterar uma associação, o nome do autor do livro:

```
1 > Book.first.person.update_attribute(:name, 'taq')
```

e recarregar novamente a página no navegador. Se dermos uma olhada lá no log do servidor, vamos ver que continua somente lendo o fragmento, sem alterar, mesmo com a alteração feita acima.

```
1 Started GET "/livro/1" for 127.0.0.1 at 2017-04-05 19:04:33 -0300
2 Processing by PubController#book as HTML
3   Parameters: {"id"=>"1"}
4   Book Load (0.1ms)  SELECT  "books".* FROM "books" WHERE "books"."id" = ? LI\
5 MIT ? [[{"id": 1}, {"LIMIT": 1}]]]
6     Rendering pub/book.html.erb within layouts/pub
7   Read fragment views/books/1-20170405215042451457/40ad3161c6ecc55ef972236f44ed\
8 8312 (0.1ms)
9     Rendered pub/book.html.erb within layouts/pub (1.7ms)
10 Completed 200 OK in 48ms (Views: 45.6ms | ActiveRecord: 0.1ms)
```

Para atualizar o fragmento, vamos precisar de um `after_save` em `Person`, onde vamos utilizar o método `touch`, que atualiza a data de alteração de cada livro da pessoa:

```

1 class Person < ActiveRecord::Base
2 ...
3   after_save :update_associations
4 ...
5   def update_associations
6     books.each(&:touch)
7   end

```

Agora, após atualizamos uma pessoa, os fragmentos em *cache* dos livros são gerados novamente:

```

1 Read fragment views/books/1-20170405221215890496/40ad3161c6ecc55ef972236f44ed\
2 8312 (0.1ms)
3 ...
4 Write fragment views/books/1-20170405221215890496/40ad3161c6ecc55ef972236f44e\
5 d8312 (0.1ms)

```

E se utilizássemos o *caching* na página do autor `app/views/pub/author.html.erb`?

```

1 <% cache @author do %>
2   <h1>Livros de <%= @author.name %></h1>
3
4   <ul id="mini_books">
5     <%= render partial: 'mini_book', collection: @author.books %>
6   </ul>
7 <% end %>

```

Olhando no servidor, quando carregando essa página no navegador:

```

1 Read fragment views/people/1-20170405221859881699/2848518c9a2bd3e8419b1a2fa04\
2 8555d (0.0ms)
3 ...
4 Write fragment views/people/1-20170405221859881699/2848518c9a2bd3e8419b1a2fa0\
5 48555d (0.1ms)

```

Agora, se alterarmos um livro do autor e recarregarmos a página, vamos continuar com o mesmo conteúdo anterior. Nesse caso está mais fácil atualizar a relação, vamos inserir no modelo Book o touch, em `belongs_to`:

```

1 class Book < ActiveRecord::Base
2 ...
3   belongs_to :person, touch: true
4 ...

```

Isso atualiza o registro da pessoa, que renova os fragmentos do *caching*, e a página do autor é gerada novamente, pois o *caching* vai “descendo” para os objetos contidos:

```
1 Write fragment views/people/1-20170405221859881699/2848518c9a2bd3e8419b1a2fa0\\
2 48555d (0.1ms)
```

HTTP caching

Também temos a opção de utilizar o *HTTP caching*, que demanda algumas explicações para que possamos entender o seu funcionamento.

Esse tipo de *caching* funciona no nível do protocolo, e tem a sua base na [ETag¹⁴³](#), ou *entity tag*, que é um dos mecanismos que operam na validação de *cache* do HTTP, e valida se no momento em que requisitamos uma página no navegador, se ele decide se vai utilizar uma cópia que já está no seu *cache* local ou se vai requisitar ao servidor uma nova cópia.

A diferença, como mencionada nos métodos anteriores, é bem aparente: o servidor pode evitar todo o processo de renderizar a página, apenas enviando de volta ao navegador uma resposta que não houve modificação, e a resposta do cliente também é bem mais rápida, afinal, todo o conteúdo necessário está no seu computador local, no *cache*.

Vamos utilizar o [cURL¹⁴⁴](#) para dar uma olhada na nossa aplicação corrente, **removendo quaisquer dos métodos de *caching* apresentados acima**, ou seja, removendo os blocos com o método *cache* e rodando novamente o comando para desabilitar o *caching*:

```
1 $ rails dev:cache
2 Development mode is no longer being cached.
3
4 $ curl -I http://localhost:3000/livro/1
5 HTTP/1.1 200 OK
6 X-Frame-Options: SAMEORIGIN
7 X-Xss-Protection: 1; mode=block
8 X-Content-Type-Options: nosniff
9 Content-Type: text/html; charset=utf-8
10 Etag: "1eb2cd82b87723efaabe54d45ed8186f"
```

A primeira e a última linha ali que nos interessam, onde estão definidos o retorno do HTTP com **200 OK** e o **Etag** com **1eb2cd82b87723efaabe54d45ed8186f**.

Se fizermos outra requisição novamente, podemos ver que continua o status 200 e a Etag foi alterada:

¹⁴³ http://en.wikipedia.org/wiki/HTTP_ETag

¹⁴⁴ <http://curl.haxx.se/>

```
1 $ curl -I http://localhost:3000/livro/1
2 HTTP/1.1 200 OK
3 X-Frame-Options: SAMEORIGIN
4 X-Xss-Protection: 1; mode=block
5 X-Content-Type-Options: nosniff
6 Content-Type: text/html; charset=utf-8
7 Etag: "eee0b16b946863bf58146ae08dd2cb2b"
```

Também podemos fazer uma requisição indicando que desejamos verificar se o conteúdo foi alterado a partir de uma determinada data. Do jeito que está, continua o mesmo tipo de retorno:

```
1 curl -I -H "If-Modified-Since: Wed, 5 Apr 2017 19:24:11 BRT" http://localhost\
2 :3000/livro/1
3 HTTP/1.1 200 OK
4 X-Frame-Options: SAMEORIGIN
5 X-Xss-Protection: 1; mode=block
6 X-Content-Type-Options: nosniff
7 Content-Type: text/html; charset=utf-8
8 Etag: "d0cea110ef01f6d7142fdf0be193778f"
```

Evitando recuperar o conteúdo novamente

Vamos habilitar agora o *HTTP caching* utilizando o método `stale?`, que vai verificar se a `Etag` ou a data especificada em `Last-modified` é menor ou igual a data em que o recurso corrente, ou seja, o livro foi modificado. Se alguma das respostas for positiva, vai ser disparado um `head :not_modified` e retornado o status HTTP 304, “Not modified”, indicando que a cópia que se encontra no *cache* do navegador pode ser utilizada sem problemas.

Para isso, vamos alterar o código do controlador `pub`, no método `book`, para:

```
1 ...
2     stale?(@book) do
3         respond_with @book
4     end
5 end
```

Agora vamos verificar novamente com o cURL, fazendo duas requisições:

```
1 $ curl -I http://localhost:3000/livro/1
2 HTTP/1.1 200 OK
3 X-Frame-Options: SAMEORIGIN
4 X-XSS-Protection: 1; mode=block
5 X-Content-Type-Options: nosniff
6 ETag: W/"dd55b3ed47d60340bb02f298ed8741bf"
7 Last-Modified: Wed, 05 Apr 2017 22:29:36 GMT
8 Content-Type: text/html; charset=utf-8
9 Cache-Control: max-age=0, private, must-revalidate
10 Set-Cookie: _session_id=df52ec0d3392d724c98f93d113f4352f; path=/; HttpOnly
11 X-Request-Id: 56bfd587-9062-46b4-9393-ba87f25868f2
12 X-Runtime: 0.060189
13
14 $ curl -I http://localhost:3000/livro/1
15 HTTP/1.1 200 OK
16 X-Frame-Options: SAMEORIGIN
17 X-XSS-Protection: 1; mode=block
18 X-Content-Type-Options: nosniff
19 ETag: W/"dd55b3ed47d60340bb02f298ed8741bf"
20 Last-Modified: Wed, 05 Apr 2017 22:29:36 GMT
21 Content-Type: text/html; charset=utf-8
22 Cache-Control: max-age=0, private, must-revalidate
23 Set-Cookie: _session_id=df52ec0d3392d724c98f93d113f4352f; path=/; HttpOnly
24 X-Request-Id: 56bfd587-9062-46b4-9393-ba87f25868f2
25 X-Runtime: 0.060189
```

Reparam que agora a Etag continua sempre a mesma, e temos o retorno de `Last-Modified`, que não havia antes. O status HTTP continua o mesmo, 200, mas vamos utilizar a consulta que utilizamos acima, pedindo a verificação a partir de uma data e hora depois que aparece em `Last-Modified`:

```
1 $ curl -I -H "If-Modified-Since: Wed, 5 Apr 2017 22:30:00 GMT" http://localhost:3000/livro/1
2 HTTP/1.1 304 Not Modified
3 X-Frame-Options: SAMEORIGIN
4 X-XSS-Protection: 1; mode=block
5 X-Content-Type-Options: nosniff
6 ETag: W/"dd55b3ed47d60340bb02f298ed8741bf"
7 Last-Modified: Wed, 05 Apr 2017 22:29:36 GMT
8 Cache-Control: max-age=0, private, must-revalidate
9 X-Request-Id: e65a7821-3f81-4012-8e02-0f4290c8884a
10 X-Runtime: 0.009706
```

Ah-há! Agora temos o status 304, indicando que o conteúdo não foi modificado e que o navegador pode utilizar seguramente a sua cópia em *cache*.

O comportamento padrão do método `stale?` é basicamente enviar os valores como agora de forma mais explícita:

```
1   ...
2     stale?(@book, etag: @book, last_modified: @book.updated_at) do
3       respond_with @book
4     end
5   end
```

Vejam que foram enviados `etag` e `last_modified` de forma explícita no método.

Também podemos utilizar para alterar outra configuração importante. Verificando o retorno acima, podemos ver em `Cache-control` que existe a cláusula `private`, que indica que é um conteúdo que deve ser disponibilizado para *caching* em navegadores, mas não em *proxies*.

Como não tem problema algum essa página ficar no *caching* de *proxies* também, podemos deixar isso de forma explícita no método `stale?` utilizando o parâmetro `public`:

```
1   ...
2     stale?(@book, etag: @book, last_modified: @book.updated_at, public: true) \
3   do
4     respond_with @book
5   end
6 end
```

Verificando novamente com cURL:

```
1 $ curl -I -H "If-Modified-Since: Wed, 5 Apr 2017 22:30:00 GMT" http://localho\
2 st:3000/livro/1
3 HTTP/1.1 304 Not Modified
4 X-Frame-Options: SAMEORIGIN
5 X-XSS-Protection: 1; mode=block
6 X-Content-Type-Options: nosniff
7 ETag: W/"dd55b3ed47d60340bb02f298ed8741bf"
8 Last-Modified: Wed, 05 Apr 2017 22:29:36 GMT
9 Cache-Control: public
10 X-Request-Id: 97ecaf41-8384-4894-9a34-09baed34f6fd
11 X-Runtime: 0.007098
```

Podemos ver agora que o `Cache-Control` foi alterado para `public`.

Paginação

Podemos utilizar a `gem kaminari`¹⁴⁵ para fazer paginação em nossa aplicação. Vamos configurar a `gem` em nosso arquivo `Gemfile`, e logo depois executar `bundle install`:

```
1 gem 'kaminari'  
2  
3 $ bundle install  
4 ...  
5 Installing kaminari (0.17.0)  
6 ...
```

Não esquecendo de reiniciar o servidor após instalar alguma `gem` nova! Agora vamos configurar o nosso arquivo de tradução para identificar corretamente os valores das opções de paginação para frente e para trás, criando um novo arquivo chamado `kaminari.pt-BR.yml` em `config/locales/`, inserindo o seguinte conteúdo:

```
1 ---  
2 pt-BR:  
3   views:  
4     pagination:  
5       previous: "&laquo; Anterior"  
6       next: "Próxima &raquo;"  
7       last: "Última"  
8       first: "Primeira"  
9       truncate: "..."
```

Vamos customizar o nosso controlador de `books` para indicar que os resultados serão paginados, levando em conta a página corrente enviada através de `params[:page]`:

```
1 ...  
2 def index  
3   @books = Book.page(params[:page]).per(1)  
4 ...
```

Reparam que utilizamos o método `per`, que permite especificar quantos itens são apresentados em cada página, e indicamos 1 *somente para ver como fica*. Agora vamos customizar a `view index.html.erb` para incluir a paginação:

¹⁴⁵<https://github.com/amatsuda/kaminari>

```
1 ...
2 <%= link_to 'Novo livro', new_book_path %>
3 <%= paginate @books %>
```

E vamos customizar um arquivo CSS para paginação:

```
1 nav.pagination {
2   color: black;
3   background: transparent;
4 }
5
6 nav.pagination span.current {
7   font-size: 1.5em;
8   font-weight: bold;
9 }
10
11 nav.pagination a {
12   text-decoration: none;
13 }
```

Para mais opções de configuração, inclusive com suporte à *caching* com páginação, podemos consultar a documentação da *gem*.

Busca e autocomplete

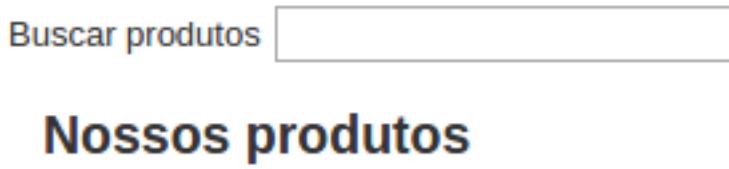
Vamos implementar agora um recurso que vai permitir que os produtos da livraria sejam buscados através de um campo texto na parte superior da aplicação, onde conforme vamos digitando os caracteres vão sendo apresentados os resultados correspondentes. Para isso vamos alterar o arquivo de layout `pub.html.erb` para incluir nosso campo de busca em todas as páginas da interface pública da aplicação:

```
1 ...
2   <section id="search">
3     <%= label_tag :search_term, 'Buscar produtos' %>
4     <%= text_field_tag :search_term, params[:term] %>
5   </section>
```

Vamos dar uma pequena estilizada no CSS com margens:

```
1 section#search {
2   margin: 1em;
3 }
```

E agora temos o nosso campo de busca:



Utilizando o autocomplete do jQuery

O jQuery¹⁴⁶ tem um *plugin* pronto para o recurso de *autocomplete*^{147 148}, disponível através do jQuery UI¹⁴⁹, que pode ser utilizado no Rails, bastando adicionar ao `Gemfile` a seguinte linha (levando em conta que o jQuery já consta no arquivo):

¹⁴⁶<http://jquery.com/>

¹⁴⁷<http://jqueryui.com/autocomplete/>

¹⁴⁸<http://jqueryui.com/autocomplete/>

¹⁴⁹<http://jqueryui.com/>

```
1 gem 'jquery-ui-rails'
```

Como de costume, rodando o `bundle install` após alterado o `Gemfile`:

```
1 $ bundle install
2 ...
3 Installing jquery-ui-rails 6.0.1
4 ...
```

Agora precisamos indicar que queremos utilizar o código JavaScript do *plugin* em `app/assets/-javascript/application.js`, inserindo a seguinte linha:

```
1 //= require jquery-ui
```

E indicar que queremos utilizar os estilos disponibilizados pelo *plugin* inserindo a seguinte linha no arquivo `app/assets/stylesheets/application.css`:

```
1 *= require jquery-ui
```



Dica

Não se esqueçam que são 3 passos necessários para utilizar o jQuery UI no Rails:

1. Habilitar a `gem`
2. Incluir no `application.js`
3. Incluir no `application.css`

Às vezes podemos pensar que é só incluir a `gem`, e quando vamos utilizar ficamos a ver navios sem os métodos estarem disponíveis ou mesmo a estilização deixar o resultado da busca todo fora de formatação.

Já com o *plugin* habilitado, vamos continuar na nossa metodologia não-obstrutiva do JavaScript, e indicar no arquivo `pub.js`, localizado em `app/assets/javascripts` (se não existir, criem, prestando atenção se já não existe criado por padrão um arquivo chamado `pub.coffee`, que deve ser apagado nesse caso), que queremos que o *plugin* seja acionado sempre que forem digitadas no mínimo 3 caracteres no campo de busca identificado com `#search_term`, dessa forma:

```

1 $(document).ready(function() {
2   $('#search_term').autocomplete({
3     source: '/busca',
4     dataType: 'json',
5     minLength: 3,
6     change: function(event, ui) {
7       if (!ui.item) {
8         $('#search_term').val('');
9       }
10    },
11    select: function(event, ui) {
12      if (ui.item) {
13        window.location.href = '/livro/' + ui.item.id;
14      }
15    }
16  });
17 });

```

Com isso já temos a busca habilitada com o *autocomplete* e com Ajax, especificando que os resultados vão ser buscados em uma URL chamada `busca` (`source`), o formato que desejamos que seja enviado seja o JSON (`dataType`), só vai ser disparada a requisição se forem digitados ao menos 3 caracteres (`minLength`), quando houver alguma alteração no resultado em que não seja selecionado nada o conteúdo do campo de busca vai ser esvaziado (`change`) e vamos trocar a URL corrente para a página do livro quando for selecionado um resultado (`select`).

Mas calma, ainda falta bastante coisa, afinal, não criamos nada para retornar os resultados para a busca. Se quisermos testar para ver se as requisições estão sendo feitas, podemos digitar, por exemplo, ‘ruby’ no campo texto, e vamos notar (através de alguma ferramenta como o [Firebug¹⁵⁰](#)¹⁵¹ ou o [Chrome Developer Tools¹⁵² ¹⁵³](#), se você não as utiliza, deve utilizar) que foi feita uma requisição assíncrona no navegador para a seguinte URL:

```
1 http://localhost:3000/busca?term=ruby
```

Reparam que `busca` foi o que foi utilizado no parâmetro `source` no código JavaScript logo acima. Essa é a URL onde o **termo** digitado, que para o *plugin* chama `term`, vai ser enviado para coletar os resultados que correspondem com ele. Como ainda não temos nem uma ação no controlador nem uma rota que corresponda com essa URL, vamos perceber na ferramenta que foi retornado um erro de URL não encontrada, e corrigir esse problema criando a rota e a ação no controlador.

Adaptando a rota e o controlador

Antes de mais nada precisamos indicar que temos uma rota para `busca` no nosso controlador público, inserindo a seguinte linha no arquivo `routes.rb`:

¹⁵⁰<http://getfirebug.com/>

¹⁵¹<http://getfirebug.com/>

¹⁵²<https://developer.chrome.com/devtools>

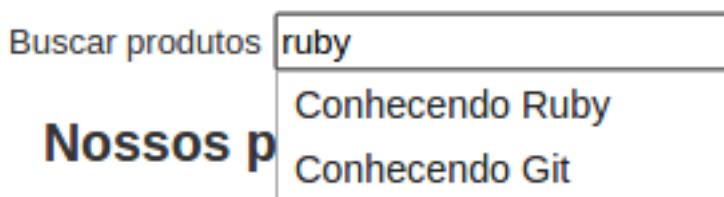
¹⁵³<https://developer.chrome.com/devtools>

```
1 get 'busca' => 'pub#search'
```

Agora vamos criar uma nova ação chamada `search` no controlador `pub`, onde, como exemplo de retorno de resultados, vamos retornar todos os livros disponíveis, só para ver como funciona:

```
1 def search
2   render json: Book.all.pluck(:title)
3 end
```

Uia. Digitando agora alguma coisa no campo de busca, temos o seguinte resultado no navegador:



Plugin funcionando com valores fixos

Legal, mas ali além de retornar todos os livros cadastrados, especificamos “na unha” que o conteúdo e tipo retornados vão ser JSON, utilizando `render: json`.

Sem problemas com isso, mas como uma forma mais limpa, sendo que em várias oportunidades temos que retornar um resultado mais limitado ou transformado, podemos utilizar o [JBuilder](#)¹⁵⁴ ¹⁵⁵, que já vem com o Rails, para formatar o resultado retornado. Isso evita que a lógica de formatação ou filtragem seja feita no controlador.

Para isso, vamos criar uma `view` chamada `app/views/pub/search.json.jbuilder`, com o seguinte conteúdo:

```
1 json.array!(@results) do |result|
2   json.id result.id
3   json.value result.title
4 end
```

Isso fará com que o controlador recupere os resultados (ainda todos os livros, por enquanto) e que automaticamente o Rails detecte que existe uma `view` com o nome do método e o MIME type que especificamos (JSON), renderizando a `view` automaticamente assim como faz com as outras, levando em conta o nome da ação (`search`) o MIME type (`json`) e sabendo como fazer o processamento (`jbuilder`).

Implementando a busca no modelo

O lugar correto para implementar a busca por termos é no modelo. Para isso, vamos criar um método de classe no modelo `Book`, chamado `search`, que vai receber o termo pesquisado e vai fazer uma consulta no banco de dados utilizando a questionável e polêmica cláusula `LIKE`:

¹⁵⁴ <https://github.com/rails/jbuilder>

¹⁵⁵ <https://github.com/rails/jbuilder>

```

1 def self.search(term)
2   where("title like ?", "%#{term}%")
3 end

```

Parece que deu tudo certo:

```

1 > Book.search("ruby").first
2 Book Load (0.4ms)  SELECT  "books".* FROM "books"  WHERE (title like '%ruby\'
3 %')
4   ORDER BY "books"."id" ASC LIMIT 1
5 => #<Book id: 1, title: "Conhecendo Ruby", published_at: "2014-04-16", text:
6   "Livro sobre a linguagem de programação **Ruby**.\r\n...", value:
7   #<BigDecimal:afab1d0,'0.1E2',9(27)>, person_id: 1, created_at: "2014-04-16
8   13:02:39", updated_at: "2014-07-05 21:02:51", stock: 0, lock_version: 45>

```

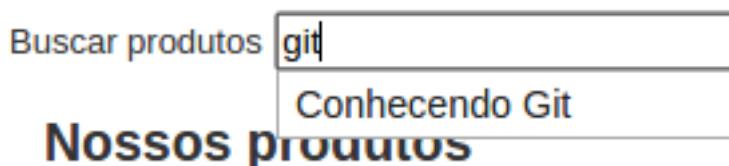
Vamos alterar o controlador, onde antes eram retornados todos os livros:

```

1 def search
2   @results = Book.search(params[:term])
3 end

```

Verificando agora no navegador:



Busca com LIKE

Onde, se selecionado algum resultado com o mouse, já somos redirecionados para a URL correspondente.

Eba, tudo funcionando! Mas ainda existem alguns pequenos e importantes detalhes ...

Buscas com LIKE?

As buscas com LIKE costumam dar arrepios e pesadelos em DBAs, pois geralmente são feitas com **full scan table**¹⁵⁶, onde todas as linhas da tabela são percorridas em busca do resultado procurado. Em uma tabela pequena como a nossa de livros atual não tem problema algum, mas imaginemos que essa tabela cresça para milhares de registros? Aí a coisa complica. Fica lento.

¹⁵⁶http://en.wikipedia.org/wiki/Full_table_scan

Muito lento. Aí vão falar que o problema é com o Rails, que não escala, etc etc etc, e não com o programador que não tomou certos cuidados ao desenvolver a aplicação.

Para complicar mais ainda, podemos ver que o SQLite utilizado por padrão no Rails no ambiente de desenvolvimento que estamos utilizando, é bonzinho e nos retorna resultados sem se importar se estão em minúsculas ou maiúsculas: procurando por ‘conhecendo’, por exemplo, retorna os dois livros que tem ‘Conhecendo’ (com a primeira letra maiúscula), mas isso não é comportamento padrão em vários bancos de dados.

Para alguns bancos, temos até a cláusula `ILIKE`, que é um `LIKE` que não se importa com maiúsculas e minúsculas, mas não são todos os bancos que suportam essa cláusula, e para piorar ainda mais, nós utilizamos (e precisamos utilizar) uma consulta com o caracter coringa % no começo da expressão, senão não conseguiríamos procurar, por exemplo, ‘ruby’ ou ‘git’, e isso também implica em recursos e performance do banco, variando de acordo com o banco utilizado.

Podemos ver que utilizamos “%#{term}#”, onde, se `term` for `ruby`, vai produzir uma consulta como:

```
1 select * from books where title like '%ruby%'
```

Para não utilizarmos mais nada de recursos na parte do código para tentar resolver esse tipo de situação, tentando concentrar o esforço de otimização no banco de dados, é bom uma conversa com o seu DBA (ou, quem sabe, você vai ter que aprender mais algumas coisinhas a mais sobre os conhecimentos necessários de um DBA) para verificar o que pode ser feito no seu banco de dados de produção.

Em um banco PostgreSQL, por exemplo, podemos utilizar os [trigram indexes¹⁵⁷](#) do PostgreSQL para efetuar as buscas, um jeito até mais otimizado e transparente. Vamos ver como utilizar esse recurso.

Dando uma olhada em como a consulta é executada **antes** de criar o índice, já levando em conta que não diferencia entre maiúsculas e minúsculas, podemos ver o custo da consulta no valor final de `cost`:

```
1 explain select title from books where title ilike '%rub%';
2
3          QUERY PLAN
4 -----
5      Seq Scan on books  (cost=0.00..1885.43 rows=10 width=20)
6          Filter: ((title)::text ~~ '%rub%'::text)
```

Agora vamos habilitar a extensão no banco e criar o índice `trigram`:

¹⁵⁷ <https://www.postgresql.org/docs/9.1/static/pgtrgm.html>

```
1 create extension pg_trgm;
2 create index concurrently books_title_trigram_index
3 on books
4 using gin (title gin_trgm_ops)
```

Dando uma olhada em como a consulta é executada **após** criar o índice:

```
1 explain select title from books where title ilike '%rub%';
2                                     QUERY PLAN
3 -----
4 -----
5   Bitmap Heap Scan on books  (cost=12.08..48.44 rows=10 width=20)
6     Recheck Cond: ((title)::text ~~~ '%rub%'::text)
7       -> Bitmap Index Scan on books_title_trigram_index  (cost=0.00..12.07 rows\
8 =10 width=0)
9           Index Cond: ((title)::text ~~~ '%rub%'::text)
```

Uau! O custo caiu de **1885.43** para **48.44**!

Agora podemos utilizar uma consulta utilizando, ao invés de LIKE, ILIKE para não diferenciar entre maiúsculas e minúsculas:

```
1 def self.search(term)
2   where("title ilike ?", "%#{term}%")
3 end
```

Em ambiente de desenvolvimento, esse ILIKE não vai funcionar com o SQLite, então podemos fazer assim:

```
1 def self.search(term)
2   operator = Rails.env.production? ? 'ilike' : 'like'
3   where("title #{operator} ?", "%#{term}%")
4 end
```

Um grande problema com esse tipo de solução é que começamos a criar coisas mais específicas e menos transparentes do banco de dados que estamos utilizando, um recurso que o ActiveRecord implementa tão bem.

Se você tiver certeza que vai utilizar determinado banco de dados para a aplicação em produção por um bom tempo (ou para sempre) não tem problema, mas eu ainda prefiro algumas soluções mais transparentes nesses casos, como o Elasticsearch que vamos ver na parte dos extras aqui no livro, que nos permitem não gastar muito esforço em recursos de determinados bancos e acaba retornando a solução do problema para o desenvolvedor, lembrando que enquanto resolve o problema, pode criar outra dependência com a infraestrutura utilizada.

API

Vamos fazer uma implementação bem básica de uma API, para mostrar alguns conceitos que mais tarde podem ser extendidas de modo mais abrangente e poderoso. Inclusive, o que fizemos no capítulo anterior, com a busca com o *autocomplete*, já é uma API, de uso *interno* da aplicação, para fazer as buscas nos seus produtos. Vamos mover a funcionalidade do controlador público para um controlador específico da API.

Versionando

Não se enganem: em algum momento durante o tempo de vida da sua aplicação, alterações na API vão ser feitas, tanto para satisfazer necessidades internas ou externas. Para organizar de forma mais eficiente, vamos começar a já fazer a nossa API com suporte à *versionamento*, ou seja, podemos implementar a API de forma a implementar simultaneamente várias versões, que podem ou não serem desativadas em algum ponto do futuro.

Para criar a primeira versão da nossa API, vamos criar o novo diretório

```
1 app/controllers/api/v1
```

e dentro desse diretório, o arquivo `books_controller.rb` com o seguinte conteúdo:

```
1 module Api
2   module V1
3     class BooksController < ApplicationController
4       def search
5         @results = Book.search("title: #{params[:term]}").records
6       end
7     end
8   end
9 end
```

Criar o diretório `app/views/api/v1/books/` com o arquivo `app/views/api/v1/books/search.json.jbuilder`, com o conteúdo igual ao que temos no arquivo da busca anterior:

```
1 json.array!(@results) do |result|
2   json.id result.id
3   json.value result.title
4 end
```

Agora precisamos de uma rota, utilizando namespaces para criar a estrutura hierárquica:

```

1 namespace :api do
2   namespace :v1 do
3     get 'books/busca' => 'books#search'
4   end
5 end

```

Vamos testar que está funcionando, primeiro com um teste em `test/controllers/api/v1/books_controller_test.rb`:

```

1 require 'test_helper'
2
3 class Api::V1::BooksControllerTest < ActionDispatch::IntegrationTest
4   setup do
5     @book = books(:one)
6   end
7
8   test 'should return some books' do
9     get api_v1_books_busca_url, params: { term: @book.title.split.last, forma\
10 t: 'json' }
11     assert_equal @book.title, JSON.parse(response.body).first['value']
12   end
13 end

```

e agora acessando diretamente na linha de comando, usando o `cURL`:

```

1 $ curl "http://localhost:3000/api/v1/books/busca?term=ruby&format=json"
2 [{"id":1,"value":"Conhecendo Ruby"}]

```

Tudo funcionando corretamente. Só que o que acontece se tivermos uma API maior, com digamos, umas 25 rotas? Quando fizemos a versão 2 vamos ter que repetir tudo novamente no arquivo de rotas? Se lembrem do `DRY` e para evitar isso, vamos utilizar o conceito de *concerns* também nas rotas, modificando o arquivo de rotas para:

```

1 concern :api_routes do
2   get 'books/busca' => 'books#search'
3 end
4
5 namespace :api do
6   namespace :v1 do
7     concerns :api_routes
8   end
9 end

```

Assim, injetamos as rotas dentro dos *namespaces* que julgarmos necessários, sem precisar repetir tudo.

Apesar de termos agora nossas rotas bem definidas, podemos achar que a rota explícita em /api/v1/books/search fica meio, digamos, deselegante. Particularmente, eu gosto da maneira explícita que fica bem clara ali, mas podemos ter a opção de enviar a versão da API desejada junto com os cabeçalhos HTTP, por exemplo, através de application/vnd.api.v1+json para especificar a versão 1.

Vamos precisar alterar nosso arquivo de rotas estabelecendo, através de uma *constraint*, uma classe (um simples PORSO) que vai escolher, através da versão enviada no cabeçalho, qual o *path* a ser utilizado, primeiro criando a classe em app/services/api_constraints.rb:

```
1 class ApiConstraints
2   def initialize(options)
3     @version = options[:version]
4     @default = options[:default]
5   end
6
7   def matches?(req)
8     req.headers['Accept'].include?("application/vnd.api.v#{@version}+json") ||
9   | @default
10  end
11 end
```

e agora alterar o nosso arquivo de rotas para:

```
1 concern :api_routes do
2   get 'books/busca' => 'books#search'
3 end
4
5 namespace :api do
6   scope module: :v1, constraints: ApiConstraints.new(version: 1) do
7     concerns :api_routes
8   end
9
10 namespace :v1 do
11   concerns :api_routes
12 end
13 end
```

Vejam o que isso produz no nosso arquivo de rotas:

```

1 $ rails routes -g api
2           Prefix Verb URI Pattern          Controller#Action
3     api_books_busca GET  /api/books/busca(.:format)    api/v1/books#search
4   api_v1_books_busca GET  /api/v1/books/busca(.:format)  api/v1/books#search

```

Temos a rota explícita apontando para a versão 1, e temos uma rota genérica apontando para uma URL *sem a versão*. Vamos fazer um teste funcional para verificar se ela está ok, enviando o cabeçalho especificando a versão, em `test/controllers/api/books_controller_test.rb`:

```

1 require 'test_helper'
2
3 class BooksControllerTest < ActionDispatch::IntegrationTest
4   setup do
5     @book = books(:one)
6   end
7
8   test 'should return some books on version 1' do
9     get api_books_busca_url, headers: { accept: 'application/vnd.api.v1+json' \
10   }, params: { term: @book.title.split.last, format: 'json' }
11   assert_equal @book.title, JSON.parse(response.body).first['value']
12 end
13 end

```

Rodando o teste, ele roda com sucesso. Se trocarmos de v1 para v2 no teste e o executarmos, vamos ver que ele falha, pois ainda não implementamos a versão 2 da API. Com isso, ficamos com suporte para as rotas explícitas, onde a versão é parte da URL, ou implícitas, onde a versão faz parte dos cabeçalhos HTTP. Podemos manter as duas ou desativarmos alguma que não desejamos mais.

Para verificar através do cURL:

```

1 $ curl -H "Accept: application/vnd.api.v1+json" "http://localhost:3000/api/bo\
2 oks/busca?term=ruby&format=json"
3 [{"id":1,"value":"Conhecendo Ruby"}]

```

Fechando o assunto do versionamento, vamos fazer rapidamente a versão 2, que além das informações da versão 1, vai retornar também o preço do livro. Vamos fazer ela disponível somente através da forma implícita nos cabeçalhos, primeiro criando em `app/controllers/api/v2/books_controller.rb` quase o mesmo código da versão 1, mas trocando aqui de V1 para V2:

```

1 module Api
2   module V2
3     class BooksController < ApplicationController
4       def search
5         @results = Book.search("title: #{params[:term]}").records
6       end
7     end
8   end
9 end

```

Agora vamos configurar a *view* em JSON que vai renderizar as informações, já com o preço do livro em `app/views/api/v2/books/search.json.builder`:

```

1 json.array!(@results) do |result|
2   json.id result.id
3   json.value "#{result.title} - #{number_to_currency(result.value)}"
4 end

```

Alteramos o arquivo de rotas para contemplar a versão 2:

```

1 namespace :api do
2   scope module: :v1, constraints: ApiConstraints.new(version: 1) do
3     concerns :api_routes
4   end
5
6   scope module: :v2, constraints: ApiConstraints.new(version: 2) do
7     concerns :api_routes
8   end
9 ...

```

Fazemos um teste para verificar se a versão 2 está ok, adicionando em `test/controllers/api/-books_controller_test.rb`:

```

1 ...
2 test 'should return some books on version 2' do
3   get api_books_busca_url, headers: { accept: 'application/vnd.api.v2+json' },
4   params: { term: @book.title.split.last, format: 'json' }
5   assert_equal "#{@book.title} - #{ ApplicationController.helpers.number_to_cur\
6 rency(@book.value)}", JSON.parse(response.body).first['value']
7 end
8 ...

```

Rodando o teste ele passa, e podemos verificar também com o cURL, verificando os dois retornos:

```

1 $ curl -H "Accept: application/vnd.api.v1+json" "http://localhost:3000/api/bo\ 
2 oks/busca?term=ruby&format=json"
3 [{"id":1,"value":"Conhecendo Ruby"}]
4
5 $ curl -H "Accept: application/vnd.api.v2+json" "http://localhost:3000/api/bo\ 
6 oks/busca?term=ruby&format=json"
7 [{"id":1,"value":"Conhecendo Ruby - R$ 40,00"}]
```

A cereja do bolo agora em questão do versionamento básico que aprendemos aqui é especificar uma versão *default*, ou seja, a assumida caso o cliente não especifique que versão que ele deseja, enviando nos cabeçalhos HTTP. Para isso é só indicar no arquivo de rotas, por exemplo, utilizando a versão 2 como *default*:

```

1 ...
2 namespace :api do
3   scope module: :v1, constraints: ApiConstraints.new(version: 1) do
4     concerns :api_routes
5   end
6
7   scope module: :v2, constraints: ApiConstraints.new(version: 2, default: true\ 
8 ) do
9     concerns :api_routes
10    end
11 ...
```

Verificando no teste em `test/controllers/api/books_controller.rb`, adicionando:

```

1 ...
2 test 'should return some books on version 2, using as default' do
3   get api_books_busca_url, params: { term: @book.title.split.last, format: 'js\ 
4 on' }
5   assert_equal "#{@book.title} - #{ ApplicationController.helpers.number_to_cur\ 
6 rency(@book.value)}", JSON.parse(response.body).first['value']
7 end
8 ...
```

E utilizando o cURL:

```

1 $ curl "http://localhost:3000/api/books/busca?term=ruby&format=json"
2 [{"id":1,"value":"Conhecendo Ruby - R$ 40,00"}]
```

Agora podemos alterar a requisição Ajax original para a URL da API, indicando qual a versão, se assim desejarmos, enviando o parâmetro nos headers. Se desejarmos utilizar a versão mais atual, é só omitir a informação dos headers e alterar somente a URL, em `app/assets/scripts/pub.js`, que é bem mais fácil:

```
1  $('#search_term').autocomplete({  
2      source: '/api/books/busca',  
3      dataType: 'json',  
4      ...  
5  } );  
6  
7  $('#search_term').autocomplete({  
8      source: '/api/books/busca',  
9      ...  
10 } );
```

Se quisermos enviar a versão, temos que customizar os cabeçalhos HTTP na chamada Ajax. Com o jQuery a partir da versão 1.5, é só enviar o parâmetro `headers`, mas como estamos utilizando o plugin de `autocomplete`, temos que fazer mais algumas adaptações customizando a chamada Ajax antes dela ser acionada, assim, no caso de desejarmos a versão 1:

```
1  $(document).ready(function() {  
2      $.ajaxSetup({  
3          headers: { 'Accept': 'application/json, application/vnd.api.v1+json' }  
4      } );  
5  
6  $('#search_term').autocomplete({  
7      source: '/api/books/busca',  
8      ...  
9  } );  
10 } );  
11  
12  $('#search_term').autocomplete({  
13      source: '/api/books/busca',  
14      ...  
15  } );  
16 } );
```

Permitindo

Se tentarmos fazer a chamada da API de outro computador, através de uma aplicação rodando nesse computador, seja ela outra aplicação Rails, uma aplicação *front-end*, um navegador, o cURL, o que seja, vamos ter uma pequena surpresa. Primeiro, descobrimos o IP do computador onde está rodando a nossa aplicação da livraria (no meu caso, 192.168.0.123), e testamos localmente com o cURL:

```
1  $ curl "http://192.168.0.123:3000/api/books/busca?term=ruby&format=json"  
2  [{"id":1,"value":"Conhecendo Ruby - R$ 40,00"}]
```

Tudo funcionou ok, agora vamos testar no outro computador:

```
1  $ curl "http://192.168.0.123:3000/api/books/busca?term=ruby&format=json"  
2  [{"id":1,"value":"Conhecendo Ruby - R$ 40,00"}]
```

Ok também, mas agora vamos fazer uma chamada **Ajax**, simulando uma aplicação externa fazendo esse chamada. Para isso, podemos utilizar uma aplicação básica rodando com o servidor web Webrick o nosso velho amigo `http://1vh.me`, que vai apontar para a nossa aplicação local, que está rodando no mesmo `localhost`, mas vamos fazer uma requisição como se fosse do subdomínio `test.1vh.me` para `api.1vh.me`, o que já configuraria um *cross domain* (e para ajudar as portas também vão ser diferentes).

Criamos o seguinte código HTML em um arquivo chamado `index.html` em qualquer diretório (de preferência, fora do diretório da aplicação, para não bagunçar por lá):

```

1 <html>
2   <head>
3     <script type="text/javascript" src="https://code.jquery.com/jquery-3.3.\
4 1.min.js"></script>
5   </head>
6   <body>
7     <h1>Testando CORS</h1>
8     <input type='button' value='Testar' />
9   </body>
10  <script type="text/javascript">
11    $(document).ready(function() {
12      $('input').click(function() {
13        $.ajax({
14          method: 'GET',
15          headers: { 'Accept' : 'application/json, application/vnd.api.v\
16 2+json' },
17          url: 'http://api.lvh.me:3000/api/books/busca',
18          data : { term: 'ruby' }
19        }).done(function(data) {
20          alert('Ok!');
21          console.log(data);
22        }).fail(function() {
23          alert('Erro!');
24        });
25      });
26    });
27  </script>
28 </html>

```

Agora fazemos um pequeno script chamado, por exemplo, `server.rb` com o seguinte conteúdo:

```

1 require 'rubygems'
2 require 'webrick'
3
4 s = WEBrick::HTTPServer.new(:Port => 8080, :DocumentRoot => '.')
5 trap('INT') { s.shutdown }
6 s.start

```

Rodamos o servidor, executando no mesmo diretório onde o arquivo `server.rb` está:

```

1 $ ruby server.rb
2 [20:53:30] INFO  WEBrick 1.3.1
3 [20:53:30] INFO  ruby 2.4.1 (2017-03-22) [x86_64-linux]
4 [20:53:30] INFO  WEBrick::HTTPServer#start: pid=29350 port=8080

```

Abrimos o navegador, especificamos a URL `http://other.1vh.me:8080`, clicamos no botão e recebemos uma mensagem de erro.

Isso acontece porque temos a proteção de requisições Ajax entre domínios diferentes e as chamadas CORS (cross-origin HTTP request) são desabilitadas por padrão. Para permiti-las, vamos precisar da gem `rack-cors`, que inclusive é habilitada por padrão quando criamos a nossa aplicação no modo API (indicando `--api` junto com `rails new`). Vamos incluí-la no `Gemfile` e executar o `bundler`:

```

1 ...
2 gem 'rack-cors'
3 ...
4 $ bundle
5 ...
6 Fetching rack-cors 1.0.2
7 Installing rack-cors 1.0.2
8 ...

```

Agora precisamos de um *initializer* para indicar quais os domínios que estão habilitados, junto com algumas outras opções de customização. Vamos criar o arquivo `config/initializers/-cors.rb` com o seguinte conteúdo:

```

1 Rails.application.config.middleware.insert_before 0, Rack::Cors do
2   allow do
3     origins '*'
4     resource '/api/**/*', headers: :any, methods: %i[get]
5   end
6 end

```

Isso vai fazer com que as requisições feitas para a API sejam liberadas para os sites listados em `origins`. Podemos especificar os domínios ali ou, como no nosso caso, especificar `*` para liberar para qualquer domínio.

Parando e disparando o servidor novamente, para carregar as novas configurações, podemos ver que se clicarmos no botão “Testar” novamente, dessa vez recebemos Ok, pois a requisição foi devidamente autorizada.

Para mais configurações, podemos consultar [a documentação da gem¹⁵⁸](#).

Autenticando

Apesar de podemos especificar ali IPs e domínios que estão autorizados a utilizar a nossa API, vale lembrar, de forma muito rápida e eficiente, pois está tudo sendo feito direto pelo Rack, toda vez que tivéssemos uma nova informação esse arquivo teria que ser reconfigurado (e não dá [para

¹⁵⁸ <https://github.com/cyu/rack-cors>

fazer de maneira dinâmica[(<https://github.com/cyu/rack-cors/issues/50>), além do que a proteção do Rack só vai server para o *cross site scripting*, permitindo, como vimos acima, a consulta direta por HTTP.

Vamos criar um meio de permitir o acesso para apenas quem puder, criando uma tabela de integrações, onde cada integração vai ter um nome, domínio e um **token** de acesso. Para isso, vamos criar o seguinte arquivo de modelo em `app/models/integration.rb`, criando e rodando primeiro a `migration`:

```
1 $ rails g model integration name domain token
2 Running via Spring preloader in process 16045
3   invoke  active_record
4     create    db/migrate/create_integrations.rb
5     create    app/models/integration.rb
6     invoke    test_unit
7     create    test/models/integration_test.rb
8     create    test/fixtures/integrations.yml
9
10 $ rails db:migrate
11 == CreateIntegrations: migrating =====
12 -- create_table(:integrations)
13 -> 0.0016s
14 == CreateIntegrations: migrated (0.0017s) =====
```

E agora customizando o código do modelo:

```
1 class Integration < ApplicationRecord
2   before_create :create_token
3
4   private
5
6   def create_token
7     self.token = SecureRandom.uuid.gsub(/[-]/, '')
8   end
9 end
```

Criando um teste para verificar se está tudo ok:

```

1 require 'test_helper'
2
3 class IntegrationTest < ActiveSupport::TestCase
4   setup do
5     @integration = integrations(:one)
6   end
7
8   test 'wont change the token when saving an existing record' do
9     old_token = @integration.token
10    @integration.name = 'New name'
11    assert @integration.save
12    assert_equal old_token, @integration.token
13  end
14
15  test 'must create a new token' do
16    integration = Integration.new
17    integration.name = 'New name'
18    integration.domain = 'localhost'
19    integration.token = nil
20    assert integration.save
21    assert_not_nil integration.token
22  end
23end

```

Ok, agora vamos precisar de um jeito de conferir esse token. A primeira coisa que devemos fazer é enviar ele através dos cabeçalhos HTTP, através de `Authorization: Token token=<token>`, então vamos alterar os nossos testes dos controladores para enviar, de acordo com o que foi encontrado nas *fixtures*. Vou mostrar aqui o teste do envio implícito da versão, mas devem serem alterados também os das versões explícitas, v1 e v2, senão eles vão parar de funcionar:

```

1 require 'test_helper'
2
3 class BooksControllerTest < ActionDispatch::IntegrationTest
4   setup do
5     @book        = books(:one)
6     @integration = integrations(:one)
7   end
8
9   test 'wont return if no token sent' do
10    get api_books_busca_url, headers: { accept: 'application/vnd.api.v1+json' \
11 }, params: { term: @book.title.split.last, format: 'json' }
12    assert_response 401
13  end
14
15  test 'should return some books on version 1' do
16    get api_books_busca_url, headers: { authorization: "Token token=#{@integrat

```

```

17 action.token}", accept: 'application/vnd.api.v1+json' }, params: { term: @book\
18 .title.split.last, format: 'json' }
19     assert_not_nil response.body
20     assert_equal @book.title, JSON.parse(response.body).first['value']
21 end
22
23 test 'should return some books on version 2' do
24     get api_books_busca_url, headers: { authorization: "Token token=#{@integr\
25 ation.token}", accept: 'application/vnd.api.v2+json' }, params: { term: @book\
26 .title.split.last, format: 'json' }
27     assert_not_nil response.body
28     assert_equal "#{@book.title} - #{ ApplicationController.helpers.number_to_\
29 currency(@book.value)}", JSON.parse(response.body).first['value']
30 end
31
32 test 'should return some books on version 2, using as default' do
33     get api_books_busca_url, headers: { authorization: "Token token=#{@integr\
34 ation.token}" }, params: { term: @book.title.split.last, format: 'json' }
35     assert_not_nil response.body
36     assert_equal "#{@book.title} - #{ ApplicationController.helpers.number_to_\
37 currency(@book.value)}", JSON.parse(response.body).first['value']
38 end
39 end

```

Opa, tem um teste quebrando, que é o de justamente que confere que, sem enviar o *token*, vamos ter uma resposta 401, indicando não autorizado. Para corrigir isso, vamos fazer a conferência do *token* em toda chamada de API. Antes de mais nada, vamos fazer um método no modelo `Integration` que vai verificar se existe o *token* enviado, se houver, retorna o registro correspondente, ou se não houver, dispara uma exceção customizada `NotAuthorized`. Poderíamos definir a exceção em `app/controllers/application_controller.rb`, junto com as outras customizadas já definidas lá, mas aquelas são exceções disponíveis apenas nos controladores. Vamos criar um diretório em `app/exceptions/` com o arquivo `not_authorized.rb` com o seguinte conteúdo:

```

1 class NotAuthorized < StandardError
2 end

```

É até uma boa idéia já movermos as nossas outras exceções customizadas para lá, já que nesse caso podemos utilizar em qualquer camada da aplicação.

Agora vamos fazer um teste para verificar se o método de autenticação funciona, nos testes unitários do modelo `Integration`:

```

1 ...
2 test 'must have a method to authenticate' do
3   assert_respond_to Integration, :auth
4 end
5
6 test 'wont authenticate' do
7   assert_raise NotAuthorized do
8     Integration.auth('bla')
9   end
10 end
11
12 test 'must authenticate' do
13   assert_equal @integration, Integration.auth(@integration.token)
14 end
15 ...

```

Temos que alterar a *fixture*, pois existem dois *tokens* lá com `MyString`. Precisamos que sejam diferentes:

```

1 one:
2   name: Integração Um
3   domain: integration1
4   token: token1
5
6 two:
7   name: Integração Dois
8   domain: integration2
9   token: token2

```

Vamos escrever o método `auth` no modelo `Integration`:

```

1 ...
2 def self.auth(token)
3   found = where(token: token).first
4   raise NotAuthorized.new unless found
5   found
6 end
7 ...

```

E agora precisamos verificar no controlador se o *token* está correto. Como já temos 2 controladores que fazem chamadas na API, precisamos concentrar isso em um ponto único. Para isso, vamos criar o controlador `app/controllers/api/base_controller.rb` com o seguinte conteúdo:

```

1 module Api
2   class BaseController < ApplicationController
3     before_action :authenticate
4
5     private
6
7     def authenticate
8       authenticate_or_request_with_http_token do |token, options|
9         Integration.auth(token)
10      end
11    end
12  end
13 end

```

E vamos alterar os arquivos dos controladores em `app/controllers/api/v1/books_controller.rb` para herdar desse controlador:

```

1 module Api
2   module V1
3     class BooksController < BaseController
4     ...

```

Assim como o controlador da versão 2 em `app/controllers/api/v2/books_controller.rb`:

```

1 module Api
2   module V2
3     class BooksController < BaseController
4     ...

```

Rodando os testes, veremos que ficou tudo ok! Só tem um pequeno probleminha ... a própria aplicação parou de consultar a API, pois não está enviando o *token*!

Poderíamos resolver isso criando um *token* para a aplicação, mas teríamos que enviar no arquivo JavaScript e o segredo da coisa iria para o beleléu. Quem visse o *token* ali poderia utilizar ele como se fosse nossa própria aplicação. Para remediar isso, vamos alterar aquele teste que especificamos que se não for enviado o *token*, ele falha:

```

1 ...
2 test 'return if no token sent and localhost' do
3   get api_books_busca_url, headers: { accept: 'application/vnd.api.v1+json' } \
4 , params: { term: @book.title.split.last, format: 'json' }
5   assert_response :success
6 end
7 ...

```

E agora o método de autenticação:

```

1 ...
2 def white_list
3   %w(127.0.0.1)
4 end
5
6 def authenticate
7   return true if white_list.include?(request.ip)
8
9   authenticate_or_request_with_http_token do |token, options|
10    Integration.auth(token)
11  end
12 end
13 ...

```

Agora temos de volta o uso da API localmente. Podemos adaptar o código ali do método `white_list` para indicar quais os IPs liberados de autenticação, utilizar o `host` (com `request.host`), etc.

Limitando

Já que temos agora um método de autenticação, podemos limitar as chamadas de cada cliente em determinado período. Vamos criar um atributo chamado `daily_limit` no nosso modelo `Integration`, com a seguinte *migration*:

```

1 $ rails g migration AddDailyLimitToIntegration daily_limit:integer
2
3 Running via Spring preloader in process 4751
4     invoke  active_record
5       create db/migrate/add_daily_limit_to_integration.rb
6
7 $ rails db:migrate
8
9 == AddDailyLimitToIntegration: migrating =====
10 -- add_column(:integrations, :daily_limit, :integer)
11   -> 0.0010s
12 == AddDailyLimitToIntegration: migrated (0.0011s)
13 =====

```

Vamos criar um registro de `Integration` para testar, lembrando de comentar a verificação da `white_list` no IP 127.0.0.1, pois senão vai passar direto, sem exercitar o limite definido:

```
1 Integration.create(name: 'Teste', domain: 'localhost', daily_limit: 3)
```

Agora vamos implementar a verificação do limite, utilizando o Redis para isso. Vamos criar uma chave com o `token`, válida por 24 horas. Antes de mais nada, precisamos criar um arquivo de inicialização do Redis em `config/initializers/redis.yml` com o seguinte conteúdo:

```
1 host: localhost
2 port: 6379
```

Vamos precisar de uma exceção nova em `app/exceptions/out_of_limit.rb`, para ser disparada quando o limite for excedido:

```
1 class OutOfLimit < StandardError
2 end
```

E agora uma classe para acessarmos as configurações do Redis em `app/services/redis_limit.rb`, especificando quanto tempo cada chave vai durar e o valor inicial da chave quando for criada:

```
1 class RedisLimit
2   include Singleton
3   attr_reader :redis
4
5   LIMIT = 60 * 60 * 24
6   INIT = 1
7
8   def get(token)
9     check_setup
10    return @redis.incr(token).to_i if @redis.get(token)
11    set(token)
12  end
13
14  private
15
16  def set(token)
17    @redis.set(token, INIT)
18    @redis.expire(token, LIMIT)
19    INIT
20  end
21
22  def configuration
23    YAML.load(File.join(Rails.root, "config", "redis.yml"))
24  end
25
26  def check_setup
27    return @redis if @redis
28    config = configuration
29    @redis = Redis.new(host: config['host'], port: config['port'])
30  end
31 end
```

Assim, quando acionarmos o método `get` para retornar a quantidade já utilizada nas últimas 24 horas, vamos já criar a chave se ela não existir, retornando com o valor inicial, que é 1, já que ela foi verificada já nessa vez. Reparem que a classe é *singleton*, ou seja, só existe uma instância dela na VM, e vamos acessá-la utilizando o método `instance`.

Agora vamos inserir uma nova verificação no controlador de API, logo abaixo de `authenticate`:

```

1 module Api
2   class BaseController < ApplicationController
3     before_action :authenticate
4     before_action :check_limit
5   ...

```

Como os filtros obedecem à ordem em que foram especificados, podemos contar com `@integration` já definida e definir o método `check_limit`, deixando uma verificação de segurança ali para o caso de ocorrer uma falha na autenticação, não declarando o objeto de integração:

```

1 def check_limit
2   return true unless @integration
3   raise OutOfLimit if RedisLimit.instance.get(@integration.token) > @integration.daily_limit.to_i
4   true
5 end
6

```

O último `to_i` é para o caso do limite diário da integração estiver nulo.

Agora vamos inserir, em `app/controllers/application_controller.rb`, um método para interceptar a exceção `OutOfLimit` disparada:

```

1 ...
2 rescue_from OutOfLimit, with: :out_of_limit
3 ...
4
5 def out_of_limit
6   render status: 429, json: { message: 'Muitas requisições' }
7 end

```

Isso fará com que uma mensagem em JSON seja retornada, junto com o [status HTTP 429¹⁵⁹](#), “Too Many Requests”, indicando que foram feitos muitas requisições.

Podemos fazer um teste direto no terminal, utilizando o cURL, primeiro entrando no console do Rails, identificando o `token` e apagando a chave correspondente:

¹⁵⁹ <https://developer.mozilla.org/pt-BR/docs/Web/HTTP>Status/429>

```
1 > token = Integration.first.token
2   Integration Load (0.2ms)  SELECT "integrations".* FROM "integrations" ORDER
3   BY "integrations"."id" ASC LIMIT ?  [["LIMIT", 1]] =>
4   "e42341ae287c466fa7f1b81ba91f152e"
5
6 > RedisLimit.instance.get(token)
7 => 7
8
9 > RedisLimit.instance.redis.del(token)
10 => 1
```

Vejam que ali eu já havia feito a chamada 7 vezes, e apaguei a chave correspondente. Agora ficou zerado, e podemos já utilizar o curl no terminal:

```
1 $ curl -H "Authorization: Token token=e42341ae287c466fa7f1b81ba91f152e" "http\
2 ://localhost:3000/api/books/busca?term=ruby"
3 [{"id":1,"value":"Conhecendo Ruby - R$ 40,00"}]
4
5 $ curl -H "Authorization: Token token=e42341ae287c466fa7f1b81ba91f152e" "http\
6 ://localhost:3000/api/books/busca?term=ruby"
7 [{"id":1,"value":"Conhecendo Ruby - R$ 40,00"}]
8
9 $ curl -H "Authorization: Token token=e42341ae287c466fa7f1b81ba91f152e" "http\
10 ://localhost:3000/api/books/busca?term=ruby"
11 [{"id":1,"value":"Conhecendo Ruby - R$ 40,00"}]
12
13 $ curl -H "Authorization: Token token=e42341ae287c466fa7f1b81ba91f152e" "http\
14 ://localhost:3000/api/books/busca?term=ruby"
15 {"message":"Muitas requisições"}
```

Tudo funcionando perfeitamente. Lembrando que comentamos o código da *white list* que libera o host local, ok?

Agora podemos inclusive apagar o código do controlador em books que fazia a busca, já que está sendo feita agora pela API. Se lembrem, toda linha de código é uma linha que você ou alguém vai ter que cuidar depois, então removam o que estiver redundante.

Deploy

Vamos ver como configurar um servidor para produção e como enviar a nossa aplicação para ele. Vamos utilizar o servidor web Nginx¹⁶⁰, o módulo Passenger¹⁶¹, instalados em um sistema com GNU/Linux Ubuntu¹⁶², utilizando a RVM, instalada não no diretório do usuário, e sim como compartilhada pelo sistema, tudo usando o banco de dados PostgreSQL¹⁶³ e sendo enviado para o servidor de produção.

Sistema operacional

Primeiro, instale o sistema operacional, e no caso do Ubuntu, instale a versão **servidor**, sendo recomendada a instalação de uma versão LTS¹⁶⁴ e 64 bits. Aguarde a instalação do sistema operacional.



A partir desse momento vamos presumir que estamos rodando a versão **servidor** do Ubuntu, como usuário **root**. Se estiver rodando como usuário normal, por favor adicione `sudo` na frente dos comandos de atualização/instalação de pacotes do sistema operacional e do *script* da RVM.

Atualize e reinicie o sistema operacional:

```
1 $ apt update  
2 $ apt upgrade  
3 $ apt dist-upgrade  
4 $ reboot
```

SSH

Uma ótima idéia é instalar o acesso via SSH ao servidor. Para isso, podemos utilizar o pacote `openssh-server`:

```
1 $ apt install openssh-server
```

Para ver o endereço IP do computador para que possa conectar com o SSH, é só utilizar o comando `ifconfig`:

¹⁶⁰<http://nginx.net>

¹⁶¹<http://modrails.com>

¹⁶²<http://www.ubuntu.com>

¹⁶³<https://www.postgresql.org>

¹⁶⁴<https://wiki.ubuntu.com/LTS>

```
1 $ ifconfig
2 eth0      Link encap:Ethernet  Endereço de HW 00:24:bf:c9:d8:e0
3           inet end.: 192.168.1.131  Bcast:192.168.1.255  Masc:255.255.255.0
4 ...
```

No exemplo acima, o endereço é 192.168.1.131.

RVM

Agora vamos instalar a RVM de modo com que seja instalada no sistema e acessível através do *path* para todos os usuários do sistema operacional.

Precisamos do curl e do git para instalarmos a RVM:

```
1 $ apt install curl git-core
```

Para instalar a RVM, temos que executar o seguinte comando no terminal:

```
1 $ curl -sSL https://get.rvm.io | bash
```

Adicionar a RVM no *path*, primeiro para uso imediato com

```
1 $ source /etc/profile.d/rvm.sh
```

Instalamos as *libs* requeridas para o sistema corrente:

```
1 $ rvm requirements
```

Instalamos o interpretador Ruby de nossa preferência e indicamos como *default*:

```
1 $ rvm install 2.4.1
2 $ rvm use 2.4.1 --default
```

Não precisamos instalar o Rails logo de cara, pois ele irá ser instalado de acordo com a nossa aplicação, mas temos que instalar o bundler para que ele seja utilizado justamente para isso:

```
1 $ gem install bundler
```

Servidor web e módulo para aplicações Rails

Vamos instalar o Passenger. Primeiro vamos adicionar as chaves necessárias:

```
1 $ apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 561F9B9CA\\
2 C40B2F7
3 $ apt-get install -y apt-transport-https ca-certificates
```

Agora inserir o repositório. Para isso, precisamos saber qual o codinome da versão do Ubuntu instalada. Para isso, podemos utilizar:

```
1 $ lsb_release -cs
```

E utilizar no seguinte comando, trocando `zesty` que foi retornado no momento que executei o comando para o que for retornado quando você rodar:

```
1 $ echo 'deb https://oss-binaries.phusionpassenger.com/apt/passenger xenial ma\\
2 in' > /etc/apt/sources.list.d/passenger.list
```

Rodar o update novamente:

```
1 $ apt update
2 ...
3 Obter:5 https://oss-binaries.phusionpassenger.com/apt/passenger xenial/main a\\
4 md64 Packages [16,3 kB]
5
6 Obter:6 https://oss-binaries.phusionpassenger.com/apt/passenger xenial/main i\\
7 386 Packages [16,4 kB]
```

E instalar o Passenger, junto com o Nginx:

```
1 $ apt install -y nginx-extras passenger
```

Agora precisamos alterar o arquivo de configuração do Nginx para ativar o Passenger. Abra o arquivo `/etc/nginx/nginx.conf` e remova o comentário (que é o `#` no início da linha), ou inclua, se não existir, a seguinte linha:

```
1 # include /etc/nginx/passenger.conf;
```

Podemos verificar se ficou tudo ok utilizando:

```
1 $ /usr/bin/passenger-config validate-install
```

Selezione Passenger e aperte Enter. Deve aparecer algo como:

```
1 * Checking whether this Passenger install is in PATH... ✓
2 * Checking whether there are no other Passenger installations... ✓
3
4 Everything looks good. :-)
```

Usuário no servidor

Vamos criar agora um usuário para rodar a aplicação no servidor. Vamos criar um usuário chamado `rails` (pode ser o nome que desejarmos) com qualquer senha que quisermos escolher, autenticar como esse usuário e criar um diretório chamado `conhecendo-rails` no diretório do usuário (que vai ter como *path* completo `/home/rails/conhecendo-rails`):

```
1 $ useradd rails -m -s /bin/bash
2 $ passwd rails
3 $ su rails
4 $ cd ~
5 $ mkdir conhecendo-rails
```



Dica

Em ambientes Unix, o caracter `~` significa o diretório do usuário.

É bom deixar estabelecido o *shell* que o usuário criado vai utilizar. Para isso, utilizamos ali acima a opção pelo bash. Após criado o usuário, se desejarmos alterar essa opção, podemos editar o arquivo `/etc/passwd` do servidor e alterar o final da linha, onde nesse caso tem `/bin/bash`, como no exemplo abaixo:

```
1 rails:x:1001:1002::/home/rails:/bin/bash
```

Configurando o servidor

Vamos configurar o Nginx para encontrar os arquivos da nossa aplicação. Temos que verificar primeiro se foi criado o diretório onde vão ser armazenadas as configurações das aplicações, que estarão em `/etc/nginx/sites-available`. Se esse diretório não existir, deve ser criado, juntamente com o `sites-enabled`:

```
1 $ mkdir /etc/nginx/sites-available
```

E também verificar se existe a indicação dentro do arquivo `/etc/nginx/conf/nginx.conf`, se não existir, deve ser inserida no bloco `http`:

```
1 include /etc/nginx/sites-enabled/*;
```

e criando o conteúdo em /etc/nginx/sites-available/conhecendo-rails:

```
1 server {
2     listen 80;
3     server_name conhecendo-rails;
4     charset utf-8;
5     location / {
6         root /home/rails/conhecendo-rails/public/;
7         passenger_enabled on;
8     }
9 }
```

Reparam que utilizamos no caminho da aplicação o diretório do usuário rails que criamos há pouco. Se foi criado com outro nome, temos que trocar ali no arquivo de configuração também. E também que indicamos o diretório public no caminho.

Agora temos que criar um link simbólico desse mesmo arquivo em /etc/nginx/sites-enabled/conhecendo-rails:

```
1 $ cd /etc/nginx/sites-enabled
2 $ ln -s ../sites-available/conhecendo-rails .
```

Agora é necessário reiniciar o servidor com

```
1 $ service nginx restart
```

Como colocamos, como forma de evitar conflitos com algum outro servidor, server_name ali no arquivo de configuração configurado para conhecendo-rails, vamos precisar de um jeito de acessar isso pelo navegador, através de `http://conhecendo-rails`. Para isso, pelo menos no GNU/Linux, podemos editar o arquivo /etc/hosts para indicar para onde que esse endereço aponta, no meu caso aqui, para o IP da máquina virtual rodando o servidor, 192.168.1.131, inserindo a seguinte linha no arquivo:

```
1 192.168.1.131 conhecendo-rails
```

Também é uma boa idéia configurar onde o servidor web vai gravar o seu log de erros, **muito útil caso aconteça alguma coisa errada na sua aplicação e você fique a ver navios sem saber o que é**. Para isso, podemos inserir no arquivo /etc/nginx/conf/nginx.conf, logo nas primeiras linhas (geralmente vem comentado com algum outro path):

```
1 error_log /var/log/nginx/error.log
```

Temos também que verificar se o diretório especificado (/var/log/nginx/) existe, e se não existir, criar:

```
1 $ mkdir -p /var/log/nginx
```

Após isso, basta reiniciar o servidor com `service nginx restart` que o *log* de erros vai estar habilitado. Vale lembrar que os *logs* da **aplicação** ficam no diretório `log`, acessível no diretório raiz da aplicação, e assim como qualquer arquivo em um sistema Unix, pode ser monitorado para vermos as alterações que vão sendo feitas utilizando o comando `tail`:

```
1 $ tail -f /var/log/nginx/error.log | less
```

Configurando o banco de dados

Vamos instalar e configurar o servidor do banco de dados PostgreSQL:

```
1 $ apt install postgresql libpq-dev
```

Vamos utilizar o usuário `root` para acessar o banco, mas é **muito boa idéia** que seja criado um novo usuário:

```
1 $ su - postgres
2 $ psql
3 psql (9.5.6)
4 Type "help" for help.
5
6 postgres=# create user rails with createdb login password 'rails';
7 CREATE ROLE
8 postgres=#
9
```

Trocando o arquivo `/etc/postgresql/9.5/main/pg_hba.conf` (ou a versão que estiver disponível) para aceitar autenticação por senha, trocando de

```
1 local    all      all      peer
```

para

```
1 local    all      all      md5
```

e reiniciar o serviço

```
1 $ service postgresql restart
```

Agora temos que alterar o arquivo `config/database.yml` para incluir a configuração do nosso banco de dados de produção. Levando em conta que o servidor fica no IP `192.168.1.131`, e que o usuário é `rails` e a senha é `rails` (que beleza, hein?), temos:

```

1 production:
2   adapter: postgresql
3   encoding: utf8
4   database: bookstore
5   host: localhost
6   username: <%= ENV['DATA_USER'] %>
7   password: <%= ENV['DATA_PWD'] %>
```

Aí utilizamos o mesmo esquema que utilizamos no e-mail: estamos recuperando o usuário e senha do banco de dados de variáveis de ambiente, que vão estar no ambiente ou no arquivo gerado pela gem figaro.

Precisamos instalar a *gem pg*. Vamos abrir o *Gemfile* e inserir a *gem*:

```

1 group :production do
2   gem 'pg'
3 end
```

e rodar o bundle:

```

1 ...
2 Installing pg 0.20.0
3 ...
```

Reparam que inserimos a *gem* no grupo *production*, para que não precisemos dela em ambiente de desenvolvimento.

Configurando o método de login no servidor

Vamos habilitar agora o login do usuário *rails* no servidor. O melhor jeito é como já está configurado, utilizando autenticação sem senha (*passwordless*), trocando chaves do SSH, para que não tenhamos que ficar digitando senhas para lá e para cá, então vamos enviar a chave local para lá. Primeiro verificamos e copiamos o conteúdo local do nosso arquivo *~/.ssh/id_rsa.pub* (se não existir, crie com *ssh-keygen*, seguindo as instruções):

```

1 $ cat ~/.ssh/id_rsa.pub
2 ssh-rsa blablablayaddayaddayaddablablablayaddayadda
3 yaddablablablayaddayaddayaddablablablayaddayaddayad
4 dablablablayaddayaddayaddablablablayaddayaddayaddab
5 lablablablayaddayaddablablablayaddayaddayaddablab
6 lablayaddayaddayaddablablablayaddayaddayaddablab
7 layaddayaddayaddablablablayaddayaddayaddablablay
8 addayaddayaddablablablayaddayaddayaddablablayadd
9 ayaddayaddablablablayaddayaddayaddablablablayadda
10 ddayaddablablablayaddayaddablablablayaddayadda
11 yaddablablablayadd== taq@morpheus
```

E, no servidor, colar esse conteúdo no arquivo `/home/rails/.ssh/authorized_keys`. Após isso já devemos conseguir fazer login via SSH no servidor, sem precisar enviar a senha.

Um belo atalho para isso é utilizar o comando `ssh-copy-id`, que vai fazer o que foi descrito acima de maneira automatizada:

```
1 $ ssh-copy-id rails@192.168.1.131
2 /usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filt\
3 er out any that are already installed
4 /usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are pro\
5 mpted now it is to install the new keys
6 rails@192.168.1.113's password:
7
8 Number of key(s) added: 1
9
10 Now try logging into the machine, with: "ssh 'rails@192.168.1.113'"
11 and check to make sure that only the key(s) you wanted were added.
```

Pronto, agora conseguimos fazer *login* no servidor sem precisar de senha:

```
1 $ ssh rails@192.168.1.131
2 Welcome to Ubuntu 16.04 LTS (GNU/Linux 3.13.0-24-generic i686)
3
4 * Documentation: https://help.ubuntu.com/
5
6 System information as of Wed May 28 20:49:48 BRT 2014
7
8 System load: 0.08           Processes:          85
9 Usage of /:   24.9% of 6.99GB  Users logged in:      1
10 Memory usage: 9%            IP address for eth0: 192.168.1.113
11 Swap usage:  0%
12
13 Graph this data and manage this system at:
14   https://landscape.canonical.com/
15
16 rails@ubuntu-server:~$
```

Configurando o método de deploy

Configurando “na unha”

Temos algumas ferramentas para fazer o deploy da nossa aplicação com apenas uma linha de comando no terminal, porém algumas delas, na minha opinião, estão apresentando um aumento de complexidade (não falando só de código, mas também de curva de aprendizado e manutenção) desnecessário e que complica para quem está chegando agora. Por isso, vamos ver como enviar

a nossa aplicação para produção do jeito mais básico que existe: enviando os arquivos para o servidor e rodando os procedimentos necessários para colocar a aplicação no ar.

Para enviar os arquivos, podemos utilizar o `scp`:

```
1 $ scp -r * rails@192.168.1.131:/home/rails/conhecendo-rails/
```

Se olharmos no diretório do servidor nesse momento, vamos ver que **todos** os arquivos da nossa aplicação estão lá, inclusive alguns desnecessários em um ambiente de produção, como os testes, etc.

Agora conectamos no servidor, vamos para o diretório da nossa aplicação, e **antes de mais nada**, configuramos a variável `RAILS_ENV` para utilizar o ambiente de produção, `production`, trocamos as permissões dos arquivos no diretório `public` e executamos o `bundler`:

```
1 $ cd conhecendo-rails
2 $ export RAILS_ENV=production
3 $ chmod -R 0755 public/*
4 $ bundle
```



Dica

Deixem sempre a variável `RAILS_ENV` como `production` inserindo no final do arquivo `~/.bashrc`:

```
export RAILS_ENV=production
```

Oh-oh. Provavelmente nesse ponto, vamos descobrir que o usuário corrente não está autorizado a instalar as gems necessárias para a aplicação, **no contexto global** do sistema, e é nos dada a opção de instalá-las no diretório `./vendor/bundle`. Vamos fazer isso:

```
1 $ bundle install --path vendor/bundle
```



Não podemos esquecer de um *runtime* JavaScript no servidor! Vamos aproveitar que estamos com o login do `root` e instalar o NodeJS e criar um link para ter um comando `node`:

```
1      $ apt install nodejs
2      $ cd /usr/bin
3      $ ln -s nodejs node
```

Agora já podemos voltar com o login do usuário `rails`, criar nosso banco de dados e rodar nossas `migrations`:

```
1 $ bundle exec rails db:create db:migrate
```

Definindo a chave secreta

Antes de acessar a aplicação, vamos precisar criar uma chave secreta. Para isso, podemos utilizar a seguinte tarefa:

```
1 $ bundle exec rails secret
2 9217d586c8b6a0f85f2b6d2cc9c255656ee6718928caa4d124c4ada1684baf8bdf4e5f9c50176\
3 c3da5b2f25f100b32796d23ecde0c15e4eff78176b457c4f5c9
```

E agora colocar essa chave ou em uma variável de ambiente chamada SECRET_KEY_BASE, ou no arquivo config/application.yml. Recomendável a segunda opção se você vai ter várias aplicações no seu servidor e ambas tem variáveis com o mesmo nome:

```
1 production:
2 ...
3   SECRET_KEY_BASE: 9217d586c8b6a0f85f2b6d2cc9c255656ee6718928caa4d124c4ada168\
4   4ba f8bdf4e5f9c50176c3da5b2f25f100b32796d23ecde0c15e4eff78176b457c4f5c9
```

Compilando os assets

Agora, se acessarmos `http://conhecendo-rails` no navegador, vamos ver a aplicação (vazia) no ar! Uma das coisas que reparamos é que não temos o CSS. Precisamos entregar nossos assets! Para isso, vamos executar no terminal:

```
1 $ bundle exec rails assets:precompile
```

Voltamos no navegador, pedimos para recarregar a página e ... nada. Após fazer alguma mudança significativa em modo de produção (ei, acabamos de gerar os assets!), precisamos reiniciar a aplicação.

Reiniciando a aplicação

Para reiniciar a nossa aplicação em modo de produção vamos precisar da seguinte tarefa hercúlea:

```
1 $ touch tmp/restart.txt
```

Pronto. Sério! Experimentem recarregar a página novamente.

Carregando a aplicação com dados iniciais

Lembram-se do arquivo db/seeds.rb que foi mencionado lá no começo do livro? Podemos utilizá-lo para a carga inicial da nossa aplicação, onde **com certeza** vamos precisar de um usuário para fazer login.

Lógico que podemos ir no terminal no servidor utilizando rails c e criarmos o usuário, mas imaginem se fossemos criar estruturas como estados, cidades etc (falando nisso, tenho uma lista de estados e cidades em [formato YAML, lá no Github^{165 166}](#)), então, convém deixar esse arquivo com conteúdo adequado.

Vamos alterá-lo para:

```
1 person = Person.find_or_create_by(  
2     name: "Eustáquio Rangel",  
3     email: "taq@bluefish.com.br",  
4     born_at: "1970-01-01"  
5 )  
6 person.password = 'teste123'  
7 person.save
```



Dica

O método `find_or_create_by` encontra ou cria um novo registro, baseado nos atributos indicados para ele. Dessa forma, podemos executar o seeds quantas vezes forem necessárias.

Agora vamos executar a linha do scp acima novamente. Tá, precisamos só enviar esse arquivo para lá, e podemos fazer assim:

```
1 $ scp db/seeds.rb rails@192.168.1.131:/home/rails/conhecendo-rails/db/seeds.rb
```

O scp é uma ferramenta muito poderosa e convém aprender alguns truques dele. Agora que já temos nosso arquivo de seeds no servidor, podemos alimentar a aplicação recém-criada:

```
1 bundle exec rails db:seed
```

E pronto! Já podemos fazer login na nossa aplicação com as informações que inserimos no seed e começar a cadastrar as informações.

¹⁶⁵<https://github.com/taq/brstatescities>

¹⁶⁶<https://github.com/taq/brstatescities>



Por uma incompatibilidade da gem `I18n` (que talvez até já foi consertada quando você estiver lendo isso), se quando for editar o registro do usuário criado aparecer uma página de erro, insira a seguinte linha no arquivo `config/application.rb` antes da linha onde indica qual o `locale` utilizado e reinicie a aplicação do modo explicado acima:

```
1 config.i18n.enforce_available_locales = false
```

Configurando deploy com uma ferramenta automatizada

Aqui entramos em um assunto meio delicado pois existem várias ferramentas para *deploy* no Rails, mas apesar da argumentação de simplicidade para colocar a aplicação no ar, algumas delas mudam tanto que fica impossível documentar de forma eficiente sem se prender em alguma determinada versão da ferramenta, o que dependendo de quantos meses demorar para atualizar por aqui, já teria mudado.

Outro problema - ou vantagem - é que várias dessas ferramentas dependem de algum sistema de VCS como o Subversion e na sua maioria, o Git, para fazer o *deploy*. Deixo claro que acho todas ferramentas muito boas que **todo desenvolvedor deve conhecer**, ainda mais o Git, que é fantástico, mas fazer alguém que está começando agora ter que conhecer outra ferramenta apenas para colocar no ar a aplicação que conseguiu fazer é um problema. Em um mundo ideal todos nasceriam sabendo e teriam tempo hábil para estudar tudo o que desse na telha, mas como isso não se aplica na maior parte do tempo, por isso que gosto de mostrar como fazer as coisas com os mínimos recursos disponíveis aqui no Rails. Depois, quando a pessoa ficar boa em Rails e não precisar estudar tanto, pode agregar mais ferramentas no seu leque.

Abaixo segue a lista de algumas das ferramentas de *deploy* disponíveis para o Rails:

- [Capistrano¹⁶⁷ ¹⁶⁸](#), a mais famosa
- [Vlad the Deployer¹⁶⁹ ¹⁷⁰](#)
- [Mina¹⁷¹ ¹⁷²](#), das apresentadas, a opção mais simples

Cada um pode ficar à vontade para escolher a ferramenta que quiser. Como desde mil-novecentos-e-lá-vai-bolinha eu sou um sujeito bem “chucro”, eu usava uma coleção de *shell scripts* para fazer os meus *deploys*, sem *fail safe* nem nada, e de uns tempos para cá resolvi automatizar mais a coisa e construí uma [Traquitana¹⁷³ ¹⁷⁴](#).

¹⁶⁷ <https://github.com/capistrano/capistrano>

¹⁶⁸ <https://github.com/capistrano/capistrano>

¹⁶⁹ http://rubyhitsquad.com/Vlad_the_Deployer.html

¹⁷⁰ http://rubyhitsquad.com/Vlad_the_Deployer.html

¹⁷¹ <http://nadarei.co/mina/>

¹⁷² <http://nadarei.co/mina/>

¹⁷³ <https://github.com/taq/traquitana>

¹⁷⁴ <https://github.com/taq/traquitana>

Conhecendo a Traquitana

Já fica o aviso: essa ferramenta é bem direta e simples, se algo der errado no *deploy*, corra para as colinas. Por enquanto não pensei ainda em colocar alguma coisa de *fail-safe* ali já que envolveria, além de restaurar código, fazer *rollback* em *migrations* etc e tal. O que a Traquitana essencialmente faz é o seguinte (que pode ser verificado no arquivo *proc.sh* que vem junto com a gem):

1. Empacota o código da aplicação, de acordo com algumas máscaras de arquivo, usando compressão *zip*
2. Envia a lista de arquivos alterados e um arquivo *zip* para o servidor.
3. Faz uma cópia dos arquivos correntes (tá, eu menti, tem um *fail-safe* mínimo)
4. Descompacta o arquivo *zip*
5. Roda o *bundle* para verificar se todas as *gems* necessárias estão instaladas.
6. Cria o banco de dados, se necessário
7. Roda as *migrations* pendentes
8. Compila os *assets*
9. Troca as permissões em *public*
10. Reinicia a aplicação

De posse de uma instalação “zerada” como indicado acima, aqui temos uma sessão da Traquitana:

```
1 $ traq
2 Running Traquitana version 0.0.21
3
4 Connecting to 192.168.1.131 using user rails
5 Sending files ...
6 proc.sh : #####
7 server.sh : #####
8 20140423214746947.list : #####
9 20140423214746947.zip : #####
10 All files sent.
11
12 Running remote update commands, please wait ...
13 Log file is /tmp/traq11867.log
14 Moving to /home/rails/conhecendo-rails//traq directory ...
15 Making a safety copy of the old contents on traq/20140423214746947.safe.zip .\
16 ..
17 Unzipping 20140423214746947.zip ...
18 Fixing gems ...
19 Gem dir owner is root
20 Performing a local gem install on vendor/bundle
21 Fetching gem metadata from https://rubygems.org/
```

```
22 Fetching additional metadata from https://rubygems.org/
23 Resolving dependencies...
24 Installing rake 10.3.1
25 Installing i18n 0.6.9
26 ...
27 Installing uglifier 2.5.0
28 Your bundle is complete!
29 It was installed into ./vendor/bundle
30 Creating database if needed ...
31 Running migrations ...
32 == 20140408234833 CreatePeople: migrating =====\\
33 ==
34 -- create_table(:people)
35 -> 0.0192s
36 ...
37 Compiling assets ...
38 Changing file permissions on public to 0755 ...
39 bash: warning: setlocale: LC_ALL: cannot change locale (pt_BR.UTF-8)
40 Restarting Nginx ...
41
42 All done. Have fun.
```

Ou seja, escrevemos `traq` no terminal, e se tudo estiver configurado corretamente, após alguns minutos a aplicação estará rodando. Para utilizarmos o comando `traq`, devemos antes de mais nada configurá-lo, o que criará um arquivo `traq.yml` no diretório `config`. Se tentarmos rodar o `traq` sem o configurar, vamos ter:

```
1 $ traq
2 Running Traquitana version 0.0.21
3
4 No config file (config/traq.yml) found.
5 Did you run traq setup ?
6 Run it and check the configuration before deploying.
```

Para resolver isso, temos que ir no diretório da nossa aplicação e digitar `traq setup`. Após isso, vamos ter um arquivo de configuração padrão, onde podemos configurar para que sejam enviados os arquivos que quisermos. Vamos dar uma olhada no arquivo já configurado para a nossa aplicação:

```
1 # Default configuration
2 directory: /home/rails/conhecendo-rails/
3 user: rails
4 list:
5 - Rakefile
6 - config.ru
7 - Gemfile
8 - - config/application.rb
9 - - config/environment.rb
10 - - config/initializers/**/*
11 - - config/environments/production.rb
12 - - config/locales/**/*
13 - - config/routes.rb
14 - - config/boot.rb
15 - - config/database.yml
16 - - config/extras/**/*
17 - - app/**/*
18 - - db/migrate/**/*
19 - - db/seeds.rb
20 - - public/javascripts/**/*
21 - - public/stylesheets/**/*
22 - - lib/**/*
23 - - public/images/**/*
24 - - public/images/uploads/**/*
25 password:
26 host: 192.168.1.131
27 server: nginx
28 shell: bash -l -c
```

Dando uma explicada no arquivo:

- **directory** - O *path* do servidor para onde a aplicação vai ser enviada
- **user** - O usuário do servidor que vai ser utilizado para a entrega. Nem preciso dizer que esse usuário tem que ter permissão de escrita para onde **directory** aponta, né?
- **list** - A lista de arquivos a serem enviados. Esse conteúdo YAML vai virar um Array quando carregado, e vamos ter conteúdos simples desde o nome do arquivo direto (como em **Rakefile**), como máscaras de arquivos por diretório (como em **app**), conteúdo extra que não vem no arquivo padrão (a linha **config/extras**) e conteúdos como o diretório **public/images**, onde a primeira linha indica que devem ser inseridos **todos** os arquivos e diretórios abaixo desse diretório, e a segunda indica quais as máscaras que devem ser **excluídas** dos arquivos que foram selecionados na primeira (ou seja, exclui do envio tudo o que estiver abaixo de **public/images/uploads/**, não sobreescrevendo algum conteúdo que já esteja na aplicação de produção).
- **password** - Pode ser preenchida, mas é **altamente recomendável** que seja feita a autenticação sem senha, como mostrado anteriormente.

- host - O domínio ou IP para onde os arquivos vão ser enviados.
- server - Qual o servidor web que está sendo utilizado, para que possamos indicar que a aplicação tem que ser reiniciada. Para isso, atualmente, todos os servidores ali utilizam o touch tmp/restart.txt como demonstrado anteriormente.
- shell - Não é recomendável alterar essa linha, mas se necessário, pode ser alterada para indicar qual o shell que deve ser rodado no momento de executar os comandos no servidor. O conteúdo que vem por padrão funciona bem com a RVM.

Após tudo configurado, é só digitar traq para cada vez que for necessário atualizar a aplicação e ficar de olho para ver se não faltou alguma coisa. Tive poucos problemas com o *deploy* utilizando essa ferramenta, e os que tive geralmente era alguma coisa de configuração no servidor, e olha que já a utilizei por vários anos. Assim como as outras, é uma ferramenta livre de código aberto e sem garantias, ou seja, utilize se quiser por sua conta e risco.

Sobre as outras ferramentas, sintam-se livres para dar uma conferida. Tem vários comportamentos interessantes nelas - e menos ... “chucros”, digamos - que pode interessar. Cada caso é um caso. ;-)

Tomando conta dos seus logs

Aí você está todo feliz com a sua aplicação rodando e depois de algum tempo percebe que o espaço em disco está acabando e não sabe porque (além de alguns *gaps* na aplicação). E percebe que tem um arquivo de log de produção de alguns gigabytes no diretório *log*. Um arquivo de log é muito importante para resolvemos alguns problemas que podem acontecer durante a execução da aplicação, nos dando valiosas pistas, por isso não podemos simplesmente apagar o dito cujo ou desabilitá-lo de alguma forma, temos que fazer o gerenciamento dos arquivos de log.

Utilizando o LogRotate

O bom que no mundo do Linux (e outros derivados do Unix) temos uma ferramenta muito boa para fazer isso, o LogRotate¹⁷⁵, que é muito fácil de configurar. Levando em conta que o diretório que instalamos a aplicação no servidor é /home/rails/conhecendo-rails, podemos criar o seguinte arquivo em /etc/logrotate.d/conhecendo-rails:

```
1 /home/rails/conhecendo-rails/log/*.log {
2     daily
3     missingok
4     rotate 7
5     compress
6     delaycompress
7     notifempty
8     copytruncate
9 }
```

¹⁷⁵ http://linuxcommand.org/man_pages/logrotate8.html

Ali foi estabelecido que:

- Rotaciona os logs diariamente (`daily`). Pode ser semanalmente (`weekly`) ou mensalmente (`monthly`).
- Se o arquivo de log não existir, pode ignorar (`missingok`).
- Mantém somente 7 dias de log (`rotate 7`)
- Comprime os arquivos usando compressão `gzip`.
- Comprime o arquivo rotacionado apenas no próximo rotacionamento, deixando o arquivo corrente e um “de brinde” sem comprimir (`delaycompress`).
- Não faz o rotacionamento se o arquivo estiver vazio (`not_ifempty`).
- Copia o arquivo de log e o esvazia. Isso garante que não teremos problemas com a aplicação escrevendo no arquivo, pois ele não some. Se não colocarmos essa opção, temos que reiniciar a aplicação toda vez que o arquivo de log corrente fosse apagado, o que não seria legal (`copytruncate`).

O LogRotate é executado uma vez por dia através do crontab, mas se quisermos ver o resultado logo após configurar o arquivo acima (e também para ver se está tudo certo!), podemos executar:

```
1 $ logrotate -v -f /etc/logrotate.d/conhecendo-rails
```

Após uma semana, teremos algo parecido com isso no diretório log:

```
1 $ ls -lah
2 total 668K
3 drwxr-xr-x  2 rails rails 4,0K .
4 drwxrwxr-x 14 rails rails 4,0K ..
5 -rw-r--r--  1 rails rails  29K production.log
6 -rw-r--r--  1 rails rails 452K production.log.1
7 -rw-r--r--  1 rails rails 3,8K production.log.2.gz
8 -rw-r--r--  1 rails rails 19K production.log.3.gz
9 -rw-r--r--  1 rails rails 71K production.log.4.gz
10 -rw-r--r-- 1 rails rails 33K production.log.5.gz
11 -rw-r--r-- 1 rails rails 34K production.log.6.gz
12 -rw-r--r-- 1 rails rails 4,3K production.log.7.gz
```

Sass

Segundo a descrição no seu site oficial, o Sass é uma metalinguagem que roda sobre o CSS, e é utilizada para descrever o estilo de um documento de maneira limpa e estruturada, com mais poder que o CSS comum permite, criando uma sintaxe simples, mais elegante, que implementa vários recursos úteis para criar folhas de estilo gerenciáveis.

O Rails vai gerar automaticamente as folhas de estilo CSS através de quaisquer arquivos com extensão sass no diretório app/assets/stylesheets, após o comando `rails assets:precompile` ser executado, em modo de produção (em modo de desenvolvimento isso é feito automaticamente).

Podemos fazer um pequeno teste com um arquivo chamado `bookstore.sass`:

```
1 $darkred: #a00000;
2
3 ul#mini_books
4   list-style-type: none
5   padding: 0
6   margin: 0
7
8   li
9     display: block
10    float: left
11    width: 33%
12    margin: 0 25px 25px 0
13    border-bottom: 1px solid gray
14    padding-bottom: 15px
15
16 h1
17   color: $darkred
18   margin-bottom: 0
19
20 h2
21   font-size: 1.2em
22   color: #aaa
23
24 img
25   display: block
```

Esse código Sass vai ser convertido no seguinte código CSS:

```
1 ul#mini_books {
2     list-style-type:none;
3     padding:0;
4     margin:0
5 }
6
7 ul#mini_books li {
8     display:block;
9     float:left;
10    width:33%;
11    margin:0 25px 25px 0;
12    border-bottom:1px solid gray;
13    padding-bottom:15px
14 }
15
16 ul#mini_books li h1{
17     color:#a00000;
18     margin-bottom:0
19 }
20
21 ul#mini_books li h2 {
22     font-size:1.2em;
23     color:#aaaaaa
24 }
25
26 ul#mini_books li img{
27     display:block
28 }
```

Para não ficar tão diferente assim do CSS padrão e não ficar baseado em indentação (o que para uns pode ser uma vantagem, para outros não), podemos ao invés de utilizar o arquivo mostrado aí em cima, criar um chamado `bookstore.css.scss`, já mais parecido com o CSS padrão, suportado à partir da versão 3 do SASS:

```
1 $darkred: #a00000;
2
3 ul#mini_books {
4     list-style-type: none;
5     padding: 0;
6     margin: 0;
7
8     li {
9         display: block;
10        float: left;
11        width: 33%;
12        margin: 0 25px 25px 0;
```

```
13      border-bottom: 1px solid gray;
14      padding-bottom: 15px;
15  }
16
17  h1 {
18      color: $darkred;
19      margin-bottom: 0;
20  }
21
22  h2 {
23      font-size: 1.2em;
24      color: #aaa;
25  }
26
27  img {
28      display: block;
29  }
30 }
```

Reparam que são duas extensões diferentes: **sass**, que vai permitir utilizar o Sass “puro-sangue”, e **css.scss**, que vai ficar mais parecido com o CSS e não precisa levar em conta a indentação.

Para uma descrição mais completa dos recursos do Sass, podemos verificar em seu site oficial¹⁷⁶.

¹⁷⁶<http://sass-lang.com>

CoffeeScript

Podemos definir o CoffeeScript¹⁷⁷ como uma linguagem que compila o seu código em código JavaScript. A regra de ouro do CoffeeScript é que é apenas JavaScript. O código é compilado no equivalente em JavaScript, sem interpretação em tempo de execução. Podemos usar quaisquer em conjunto com qualquer *framework* JavaScript. O código compilado é legível e bem formatado, passa pelo JavaScript Lint¹⁷⁸ sem *warnings*, funciona em qualquer implementação JavaScript e tende a rodar tão ou mais rápido que código equivalente feito manualmente.

De modo similar ao Sass, os arquivos com a extensão coffee são armazenados no diretório app/assets/javascripts e compilados pela task do Rake. Podemos fazer um pequeno teste com um arquivo chamado application.coffee:

```
1 $(document).ready ->
2   $('h1.sayhi').bind 'click', ->
3     alert "Você clicou em '#{$(this).text()}'"
4   alert "Existem #{$('ul#mini_books li').size()} livros nessa página!"
```

Esse arquivo nada mais faz do que inserir alguns *hooks* para quando clicarmos nos títulos dos livros, e exibir uma contagem (chata, é verdade) de quantos livros existem na página corrente. O código em CoffeeScript foi convertido para o seguinte código JavaScript:

```
1 function() {
2   $(document).ready(function(){
3     $("h1.sayhi").bind("click", function() {
4       return alert("Você clicou em '"+$(this).text()+"'")
5     });
6     return alert("Existem "+$("ul#mini_books li").size()+" livros nessa p\'\n7 ágina!")
7   })
8 }
9 }).call(this)
```

Para mais informações e uma descrição completa dos recursos, podemos visitar o site oficial¹⁷⁹.

¹⁷⁷ <http://jashkenas.github.com/coffee-script>

¹⁷⁸ <http://www.javascriptlint.com>

¹⁷⁹ <http://jashkenas.github.com/coffee-script>

Extras

A partir desse ponto, vamos ver algumas coisas interessantes, fora do que vem por padrão no Rails, mas que podemos utilizar em nossa aplicação com resultados muito bons.

MiniTest

A partir da versão 1.9 de Ruby, o [MiniTest](#)¹⁸⁰ ¹⁸¹ foi implementado como a suíte de testes oficial da linguagem, substituindo o `test/unit` padrão. Com o MiniTest podemos executar desde os testes convencionais da linguagem até testes no aspecto de specs, tão em alta com o BDD (Behaviour Driven Development), cuja maior estrela é o RSpec.

Criando a aplicação utilizando o MiniTest como suíte de testes padrão

Para utilizarmos o MiniTest em nossa aplicação, o jeito mais prático é já criá-la indicando que ele vai ser a suíte de testes padrão. Para isso, vamos fazer uma aplicação nova:

```
1 $ rails new mini
2   create
3   create  README.rdoc
4   create  Rakefile
5   create  config.ru
6   create  .gitignore
7   create  Gemfile
8   ...
```

Adicionar a gem necessária no `Gemfile`, no grupo `test` e rodar o `bundler`:

¹⁸⁰ <https://github.com/seattlerb/minitest>

¹⁸¹ <https://github.com/seattlerb/minitest>

```
1 group :test do
2   gem 'minitest-rails'
3 end
4
5 $ bundle
6 ...
7 Installing minitest-rails (3.0.0)
8 ...
```

Agora vamos executar o generator do MiniTest:

```
1 $ rails g minitest:install
2       create test/test_helper.rb
```



Se for perguntado se o arquivo `test/test_helper.rb` deve ser sobreescrito, responda que sim.

Adicionar o seguinte conteúdo em um *initializer* em `config/initializers/minitest.rb`, que vai executar os generators de teste já com os *specs* do MiniTest:

```
1 Rails.application.config.generators.test_framework :minitest, spec: true
```

E, dentro do arquivo `test_helper.rb` gerado acima, inserir, antes do `end` final da classe:

```
1 class ActiveSupport::TestCase
2 ...
3   class << self
4     alias :context :describe
5   end
6 end
```



Dica

Vocês sabem o que está ocorrendo ali com aquele `class << self`? Se não, leiam o meu *ebook* sobre a linguagem Ruby, “Conhecendo Ruby”.

O que vai nos dar um método `context` para ser utilizado em nossos testes.

Testes unitários

Agora vamos gerar um scaffold simples para mostrar o que acontece, destacando onde o MiniTest entrou em ação:

```
1 $ rails g scaffold Person name:string email:string password:string
2     invoke  active_record
3     create    db/migrate/20140424213620_create_people.rb
4     create    app/models/person.rb
5     invoke  minitest
6     create    test/models/person_test.rb
7     create    test/fixtures/people.yml
8     ...
9     invoke  minitest
10    create    test/controllers/people_controller_test.rb
11    ...
12    invoke  minitest
13    create    test/helpers/people_helper_test.rb
14    ...
```

Rodando a migration:

```
1 $ rails db:migrate
2 == 20170502110506 CreatePeople: migrating =====\\
3 ==
4 -- create_table(:people)
5   -> 0.0012s
6 == 20170502110506 CreatePeople: migrated (0.0013s) =====\\
7 ==
```

Dando uma olhada no teste unitário de Person:

```
1 require "test_helper"
2
3 describe Person do
4   let(:person) { Person.new }
5
6   it "must be valid" do
7     expect(person).must_be :valid?
8   end
9 end
```

Uau! Bem diferente hein? Vamos preencher o teste para indicar que não deve aceitar nem o nome nem o email em branco, utilizar a fixture e alterar o modelo:

```
1 require "test_helper"
2
3 describe Person do
4   let(:person) { people :one }
5
6   it "must be valid" do
7     expect(person).must_be :valid?
8   end
9
10  context "name" do
11    it "must not be null" do
12      person.name = nil
13      expect(person).wont_be :valid?
14    end
15
16    it "must not be empty" do
17      person.name = ""
18      expect(person).wont_be :valid?
19    end
20  end
21
22  context "email" do
23    it "must not be null" do
24      person.email = nil
25      expect(person).wont_be :valid?
26    end
27
28    it "must not be empty" do
29      person.email = ""
30      expect(person).wont_be :valid?
31    end
32  end
33 end
```

E agora alterar o modelo para satisfazer o teste:

```
1 class Person < ActiveRecord::Base
2   validates :name, :email, presence: true
3 end
```

Rodando agora o `rails test`, tudo deve funcionar perfeitamente (após também comentado o teste que falha, utilizando `flunk`, em `test/helpers/people_helper_test.rb`).

Testes funcionais

Vamos dar uma olhada agora no teste funcional:

```
1 require "test_helper"
2
3 describe PeopleController do
4   let(:person) { people :one }
5
6   it "gets index" do
7     get people_url
8     expect(response).must_be :success?
9   end
10
11  it "gets new" do
12    get new_person_url
13    expect(response).must_be :success?
14  end
15
16  it "creates person" do
17    expect {
18      post people_url, params: { person: { email: person.email, name: person.\
19        name, password: person.password } }
20      }.must_change "Person.count"
21      must_redirect_to person_path(Person.last)
22    end
23
24  it "shows person" do
25    get person_url(person)
26    expect(response).must_be :success?
27  end
28
29  it "gets edit" do
30    get edit_person_url(person)
31    expect(response).must_be :success?
32  end
33
34  it "updates person" do
35    patch person_url(person), params: { person: { email: person.email, name: \
36      person.name, password: person.password } }
37    must_redirect_to person_path(person)
38  end
39
40  it "destroys person" do
41    expect {
42      delete person_url(person)
43      }.must_change "Person.count", -1
44      must_redirect_to people_path
45    end
46  end
```

Podemos refatorar esse teste utilizando uma situação parecida com a que tivemos antes onde só um administrador poderia acessar o controlador. Nesse ponto ganhamos utilizando o `context`. Mas tem um pequeno detalhe: antes do Rails 5, poderíamos utilizar a variável `session` para “falsificar” os dados das nossas sessões, mas a partir do Rails 5, essa variável não está mais disponível para os testes, fazendo com que tenhamos que executar o código que cria realmente essas variáveis. Com certeza vai dar um pouco mais de processamento, mas também vai garantir que algum tipo de situação não desejada possa acontecer quando ainda havia jeito de dar um “jeitinho” nas sessões usando a `session`.

Vamos fazer um controlador de login com uma ação bem tosca para criar o conteúdo `admin` da variável de sessão. Por favor **não façam isso**, utilizem o seu login “oficial”, só estamos fazendo isso para ilustrar como ficam os procedimentos para variáveis de sessão a partir do Rails 5:

```
1 $ rails g controller login login
2
3 class LoginController < ApplicationController
4   def login
5     session[:admin] = true if params[:user] == 'admin'
6   end
7 end
```

Agora alterar o teste funcional para:

```
1 require "test_helper"
2
3 describe PeopleController do
4   let(:person) { people :one }
5
6   context "when not admin" do
7     it "wont get index" do
8       get people_url
9       must_redirect_to "/"
10    end
11
12    it "wont get new" do
13      get new_person_url
14      must_redirect_to "/"
15    end
16
17    it "wont create person" do
18      assert_no_difference('Person.count') do
19        post people_url, params: { person: { email: person.email, name: perso\
20 n.name, password: person.password } }
21      end
22      must_redirect_to "/"
23    end
24  end
```

```
24
25      it "wont show person" do
26          get person_url(person)
27          must_redirect_to "/"
28      end
29
30      it "wont edit person" do
31          get edit_person_url(person)
32          must_redirect_to "/"
33      end
34
35      it "wont update person" do
36          patch person_url(person), params: { person: { email: person.email, name\
37 : person.name, password: person.password } }
38          must_redirect_to "/"
39      end
40
41      it "destroys person" do
42          assert_no_difference('Person.count') do
43              delete person_url(person)
44          end
45          must_redirect_to "/"
46      end
47  end
48
49  context "when admin" do
50      before do
51          get login_login_path, params: { user: 'admin' }
52      end
53
54      it "gets index" do
55          get people_url
56          expect(response).must_be :success?
57      end
58
59      it "gets new" do
60          get new_person_url
61          expect(response).must_be :success?
62      end
63
64      it "creates person" do
65          expect {
66              post people_url, params: { person: { email: person.email, name: perso\
67 n.name, password: person.password } }
68          }.must_change "Person.count"
69          must_redirect_to person_path(Person.last)
```

```

70      end
71
72      it "shows person" do
73          get person_url(person)
74          expect(response).must_be :success?
75      end
76
77      it "gets edit" do
78          get edit_person_url(person)
79          expect(response).must_be :success?
80      end
81
82      it "updates person" do
83          patch person_url(person), params: { person: { email: person.email, name\
84 : person.name, password: person.password } }
85          must_redirect_to person_path(person)
86      end
87
88      it "destroys person" do
89          expect {
90              delete person_url(person)
91              }.must_change "Person.count", -1
92
93          must_redirect_to people_path
94      end
95  end
96 end

```

E no controlador:

```

1 class PeopleController < ApplicationController
2   before_action :set_person, only: [:show, :edit, :update, :destroy]
3   before_action :admin?
4 ...
5   def admin?
6     redirect_to "/" if !session[:admin]
7   end

```

Deixando os testes mais ... bonitos

Podemos dar uma “embelezada” nos nossos testes utilizando algumas gems, como por exemplo, a `minitest-reporters`¹⁸². Para isso, inserimos ela no `Gemfile` e rodamos o `bundler`:

¹⁸²<https://github.com/kern/minitest-reporters>

```
1 group :test do
2 ...
3   gem 'minitest-reporters'
4 ...
5 end
6
7 $ bundler
```

E escolhemos que tipo de *output* queremos, inserindo no `test_helper.rb`:

```
1 require "minitest/reporters"
2 Minitest::Reporters.use! Minitest::Reporters::SpecReporter.new
```

Agora quando rodamos os testes, temos:

```
1 $ rails test
2
3
4 Started
5
6 Person
7   test_0001_must_be_valid                                     PASS (0.12s)
8
9 Person::name
10  test_0001_must_not_be_null                                 PASS (0.01s)
11  test_0002_must_not_be_empty                               PASS (0.00s)
12
13 Person::email
14  test_0001_must_not_be_null                               PASS (0.00s)
15  test_0002_must_not_be_empty                             PASS (0.00s)
16
17 PeopleController::when_not_admin
18  test_0001_wont_get_index                                PASS (0.01s)
19  test_0002_wont_get_new                                  PASS (0.00s)
20  test_0003_wont_create_person                           PASS (0.01s)
21  test_0004_wont_show_person                            PASS (0.00s)
22  test_0005_gets_edit                                    PASS (0.00s)
23  test_0006_updates_person                            PASS (0.00s)
24  test_0007_destroys_person                           PASS (0.00s)
25
26 PeopleController::when_admin
27  test_0001_gets_index                                    PASS (0.04s)
28  test_0002_gets_new                                     PASS (0.02s)
29  test_0003Creates_person                                PASS (0.01s)
30  test_0004Shows_person                                 PASS (0.01s)
```

```

31   test_0005_gets_edit          PASS (0.01s)
32   test_0006_updates_person    PASS (0.01s)
33   test_0007_destroys_person  PASS (0.01s)
34
35 Finished in 0.28340s
36 19 tests, 34 assertions, 0 failures, 0 errors, 0 skips

```

Além do *reporter* Spec utilizado, temos:

- Minitest::Reporters::DefaultReporter
- Minitest::Reporters::ProgressReporter
- Minitest::Reporters::RubyMateReporter
- Minitest::Reporters::RubyMineReporter
- Minitest::Reporters::JUnitReporter

Expectations

Abaixo temos algumas das [expectations¹⁸³](#) do Minitest. Para testarmos uma condição inversa, na maioria das vezes é só trocar **must** para **wont**, por exemplo, **must_be** por **wont_be**:

- **must_be** - Testa uma condição comparando o valor retornado de um método:

```
1 10.must_be :<, 20
```

- **must_be_empty** - Deve ser vazio:

```
1 [].must_be_empty
```

- **must_be_instance_of** - Deve ser uma instância de uma classe:

```
1 "oi".must_be_instance_of String
```

- **must_be_kind_of** - Deve ser de um determinado tipo:

```
1 1.must_be_kind_of Numeric
```

- **must_be_nil** - Deve ser nulo:

```
1 a = nil
2 a.must_be_nil
```

- **must_be_same_as** - Deve ser o mesmo objeto:

¹⁸³ <http://www.ruby-doc.org/stdlib-1.9.3/libdoc/minitest/spec/rdoc/MiniTest/Expectations.html>

```
1   a = "oi"
2   b = a
3   a.must_be_same_as b
```

- **must_be_silent** - O bloco não pode mandar nada para stdout ou stderr:

```
1   -> {}.must_be_silent
2   => true
3   -> { puts "oi" }.must_be_silent
4   1) Failure:
5     test_0002_should_be_silent(Test) [minitest.rb:10]:
6     In stdout.
```

- **must_be_within_delta(exp,act,delta,msg)** - Compara Floats, verificando se o valor de exp tem uma diferença de no máximo delta de act, comparando se delta é maior que o valor absoluto de exp-act ($\delta > |exp - act|$):

```
1   1.01.must_be_within_delta 1.02, 0.1
2   => true
3   1.01.must_be_within_delta 1.02, 0.1
4   Expected |1.02 - 1.01| (0.01000000000000009) to be < 0.009
```

- **must_be_within_epsilon(exp,act,epsilon,msg)** - Similar ao delta, mas epsilon é uma medida de erro relativa aos pontos flutuantes. Compara utilizando **must_be_within_delta**, calculando delta como o valor mínimo entre exp e act, vezes epsilon ($\delta = \text{min}(exp, act) \cdot \text{epsilon}$).
- **must_equal** - Valores devem ser iguais. Para Floats, use **must_be_within_delta** explicada logo acima.

```
1   a.must_equal b
```

- **must_include** - A coleção deve incluir o objeto:

```
1   (0..10).must_include 5
```

- **must_match** - Deve “casar”:

```
1   "1".must_match /\d/
```

- **must_output(stdout,stderr)** - Deve imprimir determinado o resultado esperado em stdout ou stderr. Para testar somente em stderr, envie nil no primeiro argumento:

```
1 -> { puts "oi" }.must_output "oi\n"
2 => true
3 -> { }.must_output "oi\n"
4 1) Failure:
5 test_0004_should output(Test) [minitest.rb:20]:
6 In stdout.
```

- **must_output** - Deve disparar uma Exception:

```
1 -> { 1+"um" }.must_raise TypeError
2 => true
3 -> { 1+1 }.must_raise TypeError
4 1) Failure:
5 test_0005_should raises an exception(Test) [minitest.rb:25]:
6 TypeError expected but nothing was raised.
```

- **must_respond_to** - Deve responder à um determinado método:

```
1 "oi".must_respond_to :upcase
```

- **must_send** - Deve poder ser enviado determinado método com argumentos:

```
1 must_send ["eustáquio",:slice,3,3]
```

- **must_throw** - Deve disparar um throw:

```
1 ->{ throw :custom_error }.must_throw :custom_error
```

Guard

Nada mais chato do que ficar rodando os testes manualmente após alterarmos algum conteúdo. Para evitar isso, temos algumas ferramentas de testes contínuos/automatizados, que executam os testes necessários assim que alguma parte da aplicação é alterada e nos dando indicações visuais, sejam coloridas no terminal ou na forma de alguma integração com o sistema operacional, como por exemplo utilizando notificações no desktop, agilizando assim as respostas dos testes.

Como ferramenta de teste contínuo, vamos utilizar o [Guard](#)^{184 185}.

Podemos instalar as seguintes *gems* para utilizar Guard e Minitest no Gemfile e rodar o bundle:

```

1 group :test do
2   ...
3   gem 'guard'
4   gem 'guard-minitest'
5   ...
6 end
7
8 $ bundle

```

Após isso, podemos executar:

```

1 $ guard init minitest
2 INFO - Writing new Guardfile to minibookstore/Guardfile
3 INFO - minitest guard added to Guardfile, feel free to edit it

```

Deixar o arquivo Guardfile criado dessa maneira:

```

1 guard :minitest, spring: true do
2   watch(%r{^app/(.+)\.rb$}) { |m| "test/#{$m[1]}"\_
3     _test.rb" }
4   watch(%r{^app/controllers/application_controller\.rb$}) { 'test/controllers'\
5   ' }
6   watch(%r{^app/controllers/(.+)_controller\.rb$}) { |m| "test/integra"\
7   tion/#{$m[1]}_test.rb" }
8   watch(%r{^app/views/(.+)_mailer/.+}) { |m| "test/mailers" \
9   /#{$m[1]}_mailer_test.rb" }
10  watch(%r{^lib/(.+)\.rb$}) { |m| "test/lib/#{$m[1]}_test.rb" }
11  watch(%r{^test/.+_test\.rb$})
12  watch(%r{^test/test_helper\.rb$}) { 'test' }
13
14 end

```

¹⁸⁴<https://github.com/guard/guard>

¹⁸⁵<https://github.com/guard/guard>

E agora abrir um novo terminal e executar o Guard digitando `guard` e dando Enter. Os testes serão executados e o Guard ficará aguardando alguma alteração para executar imediatamente os testes necessários.



Dica

Lembram-se do Spring, lá do começo do livro? O Guard já está pronto para utilizá-lo, utilizando a opção `spring: true` no `Guardfile`, como demonstrado acima.

FactoryBot

Como mencionei quando falamos sobre fixtures, tem gente que acha elas o cão de zorba chupando manga e prefere utilizar o conceito de factories.

Particularmente, eu não penso assim, mas a utilização de ferramentas de factories trazem algumas vantagens bem interessantes, mas não vai ser nenhum pecado continuar a utilizar as boas e velhas fixtures que já vem prontas para o uso. Mas fiquem avisados também que podem trazer certos comportamentos que se não ficarmos atentos podem se tornar complexidades desnecessárias.

A gem que vamos utilizar como factory é a `FactoryBot`¹⁸⁶ ¹⁸⁷, que até pouco tempo atrás se chamava `FactoryGirl`, mas [o politicamente correto fez ela mudar de nome](#)¹⁸⁸.

Antes de mais nada, vamos criar uma nova aplicação, e logo após a aplicação criada, vamos alterar o `Gemfile` para incluir a gem `factory_bot_rails`:

```
1 $ rails new factory
2     create
3     create  README.rdoc
4     create  Rakefile
5     ...
6
7
8
9
10
```



```
1 group :development, :test do
2   ...
3   gem 'factory_bot_rails'
4 end
5
6 $ bundle
7 ...
8 Installing factory_bot 4.8.2
9 Installing factory_bot_rails 4.8.2
10 ...
```

Vamos fazer agora dois scaffolds, bem rapidinhos:

¹⁸⁶ https://github.com/thoughtbot/factory_bot

¹⁸⁷ https://github.com/thoughtbot/factory_bot

¹⁸⁸ https://github.com/thoughtbot/factory_bot/issues/921

```
1 $ rails g scaffold Person name:string email:string born_at:date
2     invoke  active_record
3     create    db/migrate/20140428235817_create_people.rb
4     ...
5     invoke    factory_bot
6     create      test/factories/people.rb
7     ...
8
9 $ rails g scaffold Book title:string person:references
10    invoke  active_record
11    create    db/migrate/20140428235951_create_books.rb
12    ...
13    invoke    factory_bot
14    create      test/factories/books.rb
15    ....
```

Olhem as factories criadas no diretório de teste!

Agora vamos rodar as migrations:

```
1 $ rails db:migrate
2 == 20140428235817 CreatePeople: migrating =====\ \
3 ==
4 -- create_table(:people)
5   -> 0.0011s
6 == 20140428235817 CreatePeople: migrated (0.0012s) =====\ \
7 ==
8
9 == 20140428235951 CreateBooks: migrating =====\ \
10 ==
11 -- create_table(:books)
12   -> 0.0019s
13 == 20140428235951 CreateBooks: migrated (0.0020s) =====\ \
14 ==
```

Temos que criar as associações e validações nos modelos:

```
1 class Person < ActiveRecord::Base
2   validates :name, :email, presence: true
3   has_many :books
4 end
5
6 class Book < ActiveRecord::Base
7   validates :title, presence: true, uniqueness: true
8   belongs_to :person
9 end
```

Vamos dar uma olhada nas factories, já deixando claro que vamos ver algumas opções básicas e que na [documentação da gem](#)^{189 190} existe muito mais coisas interessantes.

Primeiro em test/factories/people.rb:

```
1 FactoryBot.define do
2   factory :person do
3     name "MyString"
4     email "MyString"
5     born_at "2014-04-28"
6   end
7 end
```

E agora em test/factories/books.rb, ligeiramente alterado (em person) do original:

```
1 FactoryBot.define do
2   factory :book do
3     title "MyString"
4     person
5   end
6 end
```

Já podemos ver algumas vantagens das factories no teste unitário, onde não foi criado um registro no banco de testes, como as fixtures fazem, criando o objeto somente em memória:

¹⁸⁹ https://github.com/thoughtbot/factory_bot/blob/master/GETTING_STARTED.md

¹⁹⁰ https://github.com/thoughtbot/factory_bot/blob/master/GETTING_STARTED.md

```
1 require 'test_helper'
2
3 class PersonTest < ActiveSupport::TestCase
4   setup do
5     @person = FactoryBot.build(:person)
6   end
7
8   test "should not have a null name" do
9     @person.name = nil
10    assert !@person.valid?
11  end
12
13  test "should not have an empty name" do
14    @person.name = ""
15    assert !@person.valid?
16  end
17
18  test "should not have a null email" do
19    @person.email = nil
20    assert !@person.valid?
21  end
22
23  test "should not have an empty email" do
24    @person.email = ""
25    assert !@person.valid?
26  end
27 end
```

Associações

O método `build` que é o responsável por isso. Ele constrói um objeto `Person` carregando os dados **sem tocar no banco**, pois as factories não persistem os dados a não ser que pedimos.

No teste unitário de `Book`:

```
1 require 'test_helper'
2
3 class BookTest < ActiveSupport::TestCase
4   setup do
5     @book = FactoryBot.build(:book)
6   end
7
8   test "should not have a null title" do
9     @book.title = nil
10    assert !@book.valid?
11  end
```

```
12
13 test "should not have an empty title" do
14   @book.title = ""
15   assert !@book.valid?
16 end
17
18 test "should belongs to a person" do
19   assert_not_nil @book.person
20   assert_kind_of Person, @book.person
21 end
22 end
```

Reparam que no último teste a associação já foi construída, pois indicamos person na factory, e já que Book belongs_to Person, o registro já foi associado.

Podemos rodar agora somente os testes unitários (os de controladores estão pedindo fixtures ainda e devem ser alterados) para verificar que está tudo ok:

```
1 $ rails test:models
2 Run options: --seed 61657
3
4 # Running:
5
6 .....
7
8 Finished in 0.093021s, 75.2519 runs/s, 86.0022 assertions/s.
9 7 runs, 8 assertions, 0 failures, 0 errors, 0 skips
```

Podemos criar uma factory que define uma pessoa com vários livros, utilizando:

```
1 FactoryBot.define do
2   factory :person do
3     name "MyString"
4     email "MyString"
5   end
6
7   factory :person_with_books, parent: :person do
8     after(:build, :stub, :create) do |person, evaluator|
9       create_list(:book, 3, person: person)
10    end
11  end
12 end
```

e especificar no teste:

```
1 test "should have some books" do
2   @person = FactoryBot::build(:person_with_books)
3   assert_equal 3, @person.books.size
4 end
```

Se rodarmos os testes agora, vamos ter um pequeno probleminha, que vai ser solucionado logo abaixo.

Sequências

Um problema gerando os livros ali em cima pode ocorrer se especificarmos que os títulos deles não podem ser repetidos, utilizando `validates :title, uniqueness: true` no modelo. Podemos resolver isso utilizando `sequences` na factory:

```
1 FactoryBot.define do
2   factory :book do
3     sequence(:title) { |n| "Livro número #{n}" }
4     person
5   end
6 end
```

Agora sempre vai ser gerado um título único. Outro atributo interessante de inserir como uma sequência é o `email`, como vamos ver logo abaixo.

Traits

Traits são agrupamentos de atributos que podem ser aplicados em qualquer factory. Se por exemplo queremos um usuário e uma usuária:

```
1 FactoryBot.define do
2   factory :person do
3     name "MyString"
4     email "MyString"
5   end
6
7   ...
8   trait :male do
9     name "João"
10  end
11
12  trait :female do
13    name "Maria"
14  end
15
```

```
16 trait :bluefish do
17   sequence(:email) { |n| "user#{n}@bluefish.com.br" }
18 end
19
20 factory :male_user, parent: :person, traits: [:male]
21 factory :female_user, parent: :person, traits: [:female]
22
23 factory :bluefish_male_user, parent: :person, traits: [:bluefish, :male]
24 end
```

Verificando no teste unitário:

```
1 test "should be a bluefish's male user" do
2   @person = FactoryBot.build(:bluefish_male_user)
3   assert_equal "João", @person.name
4   assert_match /@bluefish\.com\.\br/, @person.email
5 end
```

Lazy attributes

Podemos calcular um atributo enviando um bloco para ser avaliado. Por exemplo, no caso da pessoa ter que ter mais que 16 anos, na factory:

```
1 FactoryBot.define do
2   factory :person do
3     name "MyString"
4     email "MyString"
5     born_at { 16.years.ago - 1.day }
6     ...
7   end
```

e no teste unitário:

```
1 test "should be more than 16 years old" do
2   assert @person.born_at < 16.years.ago
3 end
```

Aliases

Para ficar mais claro, podemos criar apelidos para as factories, como chamar person de author:

```
1 FactoryBot.define do
2   factory :person, aliases: [:author] do
3     name "MyString"
4     ...
```

Note que se utilizarmos `author` na factory Book também temos que alterar o relacionamento do modelo Book e os testes unitários.

Atributos dependentes

Podem ser criados com os valores de outros atributos:

```
1 FactoryBot.define do
2   factory :person, aliases: [:author] do
3     name "Eustaquio Rangel"
4     email { "#{name.parameterize}@gmail.com" }
5     ...
```

Verificando no teste unitário:

```
1 test "should have an email based on name" do
2   assert_equal "#{@person.name.parameterize}@gmail.com", @person.email
3 end
```

Herança

Como visto em associações, quando utilizamos `person_with_books`, podemos criar novas factories com atributos diferenciados:

```
1 FactoryBot.define do
2   factory :person, aliases: [:author] do
3     name "Eustaquio Rangel"
4     email { "#{name.parameterize}@gmail.com" }
5     born_at { 16.years.ago - 1.day }
6     ...
7
8   factory :yahoo_mail do
9     email { "#{name.parameterize}@yahoo.com" }
10  end
11  ...
```

Isso também poderia ser feito da maneira que estávamos fazendo até agora, que deixa mais separado e explícito:

```
1 ...  
2 end  
3  
4 factory :yahoo_mail, parent: :person do  
5   email { "#{name.parameterize}@yahoo.com" }  
6 end  
7 end
```

Callbacks

Como também vimos em `person_with_books`, podemos utilizar alguns *callbacks* (lá foi utilizado `after`) para disparar blocos de código. Os disponíveis atualmente são:

- `after(:build)` - chamado após uma `factory` ser construída, utilizando `FactoryBot.build` ou `FactoryBot.create`
- `before(:create)` - chamado antes que uma `factory` é salva, utilizando `FactoryBot.create`
- `after(:create)` - chamado depois que uma `factory` é salva, utilizando `FactoryBot.create`
- `after(:stub)` - chamado após que seja feito *stub* em uma `factory`, utilizando `FactoryBot.build_stubbed`

Construindo factories com valores diferentes

Podemos especificar valores diferentes do que estão fixos nas *factories*, direto nos métodos `build`, `build_stubbed` ou `create`. Vamos ver um teste unitário sobre isso, especificando um nome bem diferente ¹⁹¹:

```
1 test "should have a funny name" do  
2   funny_name = "Joey Joe Joe Jr. Shabadoo"  
3   @person = FactoryBot::build(:person, name: funny_name)  
4   assert_equal funny_name, @person.name  
5   assert_equal "joey-joe-joe-jr-shabadoo@gmail.com", @person.email  
6 end
```

Como dito antes, existem muitas outras opções e coisas interessantes na `FactoryBot` (não posso deixar de mencionar o `build_stubbed` como forma de dar uma acelerada na coisa), mas convém ir acostumando com esses conceitos básicos de `factories` expostos e depois ir mais a fundo no excelente material que tem na documentação e em vários outros locais na internet.

¹⁹¹<https://www.youtube.com/watch?v=f2EKaLO82sc>

Capybara

Como já demonstrado anteriormente em testes de sistema, disponíveis a partir do Rails 5.1, é utilizada a *gem* Capybara para fazer o papel do navegador nos testes. Aqui vou apresentar como fazíamos os testes de integração *antes* do Rails 5.1, até porque tem muita aplicação legada por aí (eu tenho várias), então, continue a leitura se você quiser saber como são os testes de integração.

Para instalar a *gem*, com suporte ao Rails, vamos alterar o `Gemfile` e rodar o `bundler`:

```
1 ...
2 group :test do
3   ...
4   gem 'minitest-rails-capybara'
5   ...
6 end
7 ...
8 $ bundle
9 ...
10 Installing minitest-rails-capybara 3.0.1
11 ...
```

Agora adicionar o seguinte conteúdo no arquivo `test/test_helper.rb`:

```
1 require "minitest/rails/capybara"
```

Agora vamos gerar uma `feature` de teste, utilizando `specs`. Lembram-se do nosso teste funcional do controlador `login` onde verificamos se um usuário conseguia se autenticar no sistema? Vamos refazer o controlador `login` novamente, gerando e preenchendo o conteúdo das duas ações e rotas como no exemplo feito anteriormente:

```
1 $ rails g controller login login logout
2 ...
3 <insira o código no controlador e crie as views de acordo com o exemplo anter\ 
4 ior>
```

Agora vamos reescrever o teste como uma `feature`:

```
1 $ rails generate minitest:feature CanLogin
2       create  test/features/can_login_test.rb
3       ...
```

Isso gera o seguinte arquivo `test/features/can_login_test.rb`:

```
1 $ cat test/features/can_login_test.rb
2 require "test_helper"
3
4 feature "CanLogin" do
5   scenario "the test is sound" do
6     visit root_path
7     page.must_have_content "Hello World"
8     page.wont_have_content "Goobye All!"
9   end
10 end
```

Se rodarmos nossos testes agora e verificarmos que a feature gerada não foi incluída nos testes, podemos criar um novo arquivo em lib/tasks/features.rake:

```
1 Rails::TestTask.new("test:features" => "test:prepare") do |t|
2   t.pattern = "test/features/**/*_test.rb"
3 end
4 Rake::Task["test:run"].enhance ["test:features"]
```

Isso vai nos dar uma task nova no Rake (que vai falhar se executada agora):

```
1 $ rake test:features
2 Started
3
4 CanLogin Feature Test
5   test_0001_the test is sound                                     ERROR (0.36s)
```

Para testar a feature, vamos inserir o método auth no model Person como fizemos anteriormente, junto com as fixtures e todo o controlador login exibido anteriormente (pode ser sem encriptar a senha mesmo), onde é feita a autenticação. Depois disso podemos alterar a feature para:

```
1 require "test_helper"
2
3 feature "CanLogin" do
4   scenario "needs header" do
5     visit autenticar_path
6     page.must_have_content "Autenticação no sistema"
7   end
8
9   scenario "needs valid email and password" do
10    visit autenticar_path
11    click_button "Autenticar"
12    page.must_have_content "Informe email e senha"
13 end
```

```
14
15 scenario "needs valid email" do
16   visit autenticar_path
17   fill_in "Senha" , with: "teste"
18   click_button "Autenticar"
19   page.must_have_content "Informe o email"
20 end
21
22 scenario "needs valid password" do
23   visit autenticar_path
24   fill_in "E-mail", with: "taq@bluefish.com.br"
25   click_button "Autenticar"
26   page.must_have_content "Informe a senha"
27 end
28
29 scenario "user can't login with wrong email" do
30   visit autenticar_path
31   fill_in "E-mail", with: "eustaquiorangel@gmail.com"
32   fill_in "Senha" , with: "teste"
33   click_button "Autenticar"
34   page.must_have_content "Falha no login"
35 end
36
37 scenario "user can't login with wrong password" do
38   visit autenticar_path
39   fill_in "E-mail", with: "taq@bluefish.com.br"
40   fill_in "Senha" , with: "testiculo"
41   click_button "Autenticar"
42   page.must_have_content "Falha no login"
43 end
44
45 scenario "user can login" do
46   visit autenticar_path
47   fill_in "E-mail", with: "taq@bluefish.com.br"
48   fill_in "Senha" , with: "teste"
49   click_button "Autenticar"
50   page.must_have_content "Bem-vindo, Eustáquio Rangel"
51 end
52 end
```

Rodando agora:

```

1 $ rake test:features
2 Started
3
4 CanLogin Feature Test
5   test_0001_needs header                                     PASS (0.17s)
6   test_0002_needs valid email and password                 PASS (0.03s)
7   test_0003_needs valid email                             PASS (0.02s)
8   test_0004_needs valid password                         PASS (0.02s)
9   test_0005_user can't login with wrong email          PASS (0.03s)
10  test_0006_user can't login with wrong password        PASS (0.02s)
11  test_0007_user can login                            PASS (0.04s)
12
13 Finished in 0.34653s
14 7 tests, 7 assertions, 0 failures, 0 errors, 0 skips

```

Alguns dos métodos que podemos utilizar com o Capybara:

- `visit` - aciona um GET na URL, e é utilizado com uma String (`visit "/autenticar"`) ou com um *path*, como demonstrado acima.
- `click_link` - clica em um link, utilizado com o *id* ou o texto do elemento.
- `click_button` - clica em um botão, utilizado com o texto do botão
- `click_on` - clica em um link ou botão
- `fill_in` - preenche um elemento (`with: "texto"`)
- `choose` - seleciona um *radio button*
- `check` - ativa uma *check box*
- `uncheck` - desativa uma *check box*
- `attach_file` - anexa um arquivo
- `select` - seleciona uma opção de um elemento *drop down*
- `page.has_selector?` - verifica se existe um seletor (elemento)
- `page.has_css?` - verifica se existe um seletor CSS
- `page.has_content?` - verifica se existe o conteúdo (texto)
- `find_field` - seleciona um campo de formulário
- `find_link` - seleciona um link
- `find_button` - encontra um botão

Para a documentação completa, consulte aqui¹⁹²: <http://rubydoc.info/github/jnicklas/capybara/master/Capybara>

Simulando sessões

Se criarmos uma feature nova para testar o nosso gerenciamento de pessoas:

¹⁹² <http://rubydoc.info/github/jnicklas/capybara/master/Capybara/Node>

¹⁹³ <http://rubydoc.info/github/jnicklas/capybara/master/Capybara/Node>

```
1 $ rails generate minitest:feature ManagePeople
```

e a preencheremos com um simples teste para visitar a página de pessoas cadastradas e validar o valor *default* que o scaffold colocou lá:

```
1 feature "ManagePeople" do
2   scenario "have a list of people" do
3     visit people_path
4     page.must_have_content "Listing People"
5   end
6 end
```

ela vai falhar, porque temos que estar autenticados, com conteúdo na sessão, para que a resposta não seja redirecionada para a página de autenticação. Para simular uma sessão no Capybara, vamos precisar da gem `rack_session_access`:

```
1 group :test do
2 ...
3   gem 'rack_session_access'
4 ...
5 end
6
7 $ bundle
8 ...
9 Installing rack_session_access 0.1.1
10 ...
```

Temos que inserir o seguinte conteúdo no arquivo `test/test_helper.rb`:

```
1 require 'rack_session_access/capybara'
```

e no arquivo do ambiente de testes, `config/environments/test.rb`:

```
1 config.middleware.use RackSessionAccess::Middleware
```

e indicar que queremos simular uma sessão, inserindo na *feature*:

```
1 feature "ManagePeople" do
2   background do
3     @user = people(:one)
4     page.set_rack_session(id: @user.id, admin: true)
5   end
6   ...
```

Agora sim, se rodarmos as features:

```
1 $ rake test:features
2 Started
3
4 CanLogin Feature Test
5   test_0001_needs header                                     PASS (0.15s)
6   test_0002_needs valid email and password                  PASS (0.02s)
7   test_0003_needs valid email                                PASS (0.01s)
8   test_0004_needs valid password                            PASS (0.01s)
9   test_0005_user can't login with wrong email             PASS (0.02s)
10  test_0006_user can't login with wrong password          PASS (0.02s)
11  test_0007_user can login                                PASS (0.03s)
12
13 ManagePeople Feature Test
14 test
15   test_0001_have a list of people                         PASS (0.03s)
16
17 Finished in 0.28890s
18 8 tests, 8 assertions, 0 failures, 0 errors, 0 skips
```

Podemos preencher de acordo com o que precisarmos:

```
1 require "test_helper"
2
3 feature "ManagePeople" do
4   background do
5     @user = people(:one)
6     page.set_rack_session(id: @user.id, admin: true)
7   end
8
9   scenario "have a list of people" do
10    visit people_path
11    page.must_have_content "Listing People"
12  end
13
14 scenario "edit person" do
15   visit people_path
```

```
16  page.click_link "Edit", { href: "/people/#{@user.id}/edit" }
17  page.fill_in "Name" , with: "John Doe"
18  page.fill_in "Email", with: "john@doe.com"
19  page.click_button "Update Person"
20  page.must_have_content "John Doe"
21  page.must_have_content "john@doe.com"
22 end
23 end
```

SimpleCov

Agora temos várias ferramentas para testarmos nossa aplicação, mas quanto da aplicação que foi testada? Existem algumas ferramentas para nos dar estatísticas sobre isso, uma delas é a gem [SimpleCov¹⁹⁴](#), que vamos utilizar agora.

Muito já foi discutido sobre cobertura de testes em uma aplicação, variando das partes essenciais até uma cobertura de 100%. Tem um artigo bem legal do Martin Fowler sobre isso [¹⁹⁵](#), onde ele aponta mais alguns [¹⁹⁶](#) artigos sobre isso [¹⁹⁷](#). Enfim, é uma métrica bem controversa, mas que de uma forma ou de outra ajuda a identificar algumas partes dos testes que devem ser melhoradas ou implementadas. Usem com sabedoria.

Vamos fazer uma nova aplicação para nossos testes:

```
1 $ rails new cov
```

Instalando

Vamos instalar a gem, inserindo no `Gemfile`:

```
1 group :test do
2   ...
3   gem 'simplecov', require: false
4   ...
5 end
```

Rodar o bundler:

```
1 $ bundle
2 ...
3 Fetching simplecov 0.15.1
4 Installing simplecov 0.15.1
5 ...
```

Configurando

No arquivo `test/test_helper.rb`, vamos inserir as instruções para carregar e executar a SimpleCov. Atenção que, segundo a documentação da gem, parece que é importante inserir essas linhas bem no começo do arquivo:

¹⁹⁴<https://github.com/colszowka/simplecov>

¹⁹⁵<https://martinfowler.com/bliki/TestCoverage.html>

¹⁹⁶<http://www.exampler.com/testing-com/writings/coverage.pdf>

¹⁹⁷<http://www.developertesting.com/archives/month200705/20070504-000425.html>

```
1 require 'simplecov'  
2 SimpleCov.start 'rails'  
3 ...
```

Executando

E agora vamos fazer um *scaffold*, que vai gerar testes unitários e funcionais, para verificarmos o retorno, já rodando as *migrations* necessárias:

```
1 $ rails g scaffold person name email  
2  
3 $ rails db:migrate  
4  
5 == 20171118133933 CreatePeople: migrating =====\n6 ==  
7 -- create_table(:people)  
8   -> 0.0007s  
9 == 20171118133933 CreatePeople: migrated (0.0013s) =====\n10 ==
```

Agora estamos prontos para executarmos nossos testes, já com a cobertura ativa:

```
1 $ RAILS_ENV=test rails test  
2 Running via Spring preloader in process 24056  
3 Coverage report generated for MiniTest to /tmp/cov/coverage. 0 / 76 LOC (0.0%)\n4 ) covered.  
5 Run options: --seed 64021  
6  
7 # Running:  
8  
9 .....  
10  
11 Finished in 0.333768s, 20.9727 runs/s, 26.9648 assertions/s.  
12 7 runs, 9 assertions, 0 failures, 0 errors, 0 skips
```

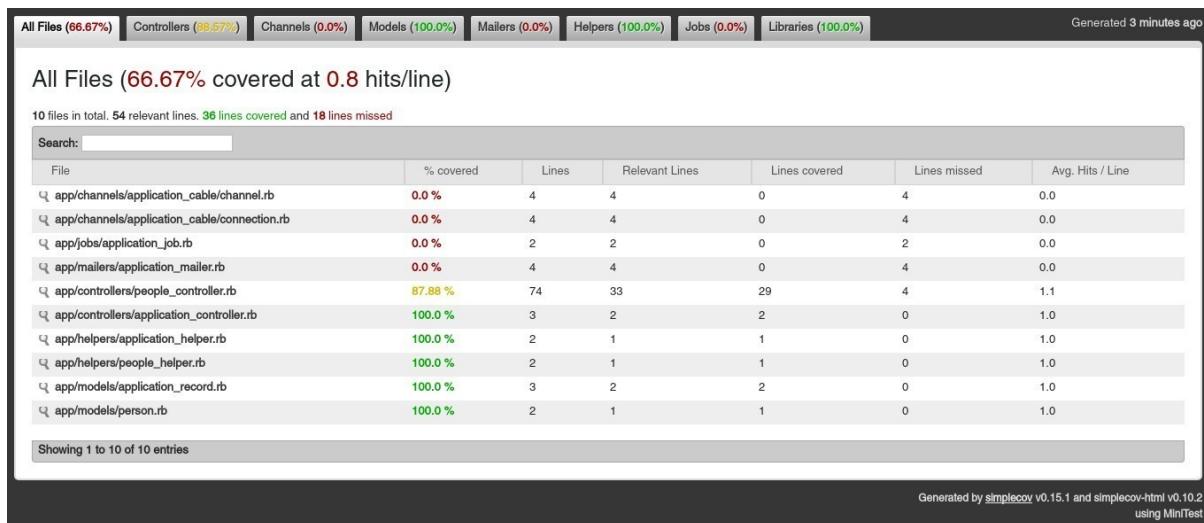
Vejam que os relatórios foram gerados em um diretório chamado `coverage`, na raiz da aplicação. Esse é um diretório bem provável de ser candidato no seu arquivo `.gitignore` (se você estiver utilizando o Git, lógico), para não entrar no seu controle de versão. Vamos dar uma olhada no que tem nesse diretório:

```

1 $ ls coverage/
2 total 24K
3 drwxrwxr-x 3 taq taq 4,0K .
4 drwxrwxr-x 14 taq taq 4,0K ..
5 drwxrwxr-x 3 taq taq 4,0K assets
6 -rw-rw-r-- 1 taq taq 4,0K index.html
7 -rw-rw-r-- 1 taq taq 51 .last_run.json
8 -rw-rw-r-- 1 taq taq 223 .resultset.json
9 -rw-rw-r-- 1 taq taq 0 .resultset.json.lock

```

O arquivo que estamos interessados é o `index.html`. Abrindo ele no navegador, vamos ter algo como:



Tela inicial do SimpleCov

Ali temos as informações pertinentes à cobertura dos testes, onde podemos ver:

1. O nome do arquivo
2. O percentual de linhas cobertas por testes
3. Quantas linhas o arquivo tem
4. Quantas linhas *relevantes* para a cobertura de testes o arquivo tem
5. Quantas linhas foram cobertas por testes
6. Quantas linhas faltaram
7. Média de quantas linhas foram atingidas por teste

Na configuração padrão do `scaffold`, podemos ver que o controlador de pessoas está com cobertura de 87,8% de testes. Vamos clicar no nome do arquivo para o navegador abrir a cobertura desse arquivo específico, e destacando o método `create`, vamos encontrar algo como:

```

26. def create
27.   @person = Person.new(person_params)
28.
29.   respond_to do |format|
30.     if @person.save
31.       format.html { redirect_to @person, notice: 'Person was successfully created.' }
32.       format.json { render :show, status: :created, location: @person }
33.     else
34.       format.html { render :new }
35.       format.json { render json: @person.errors, status: :unprocessable_entity }
36.     end
37.   end
38. end
39.

```

Cobertura do controlador de pessoas

Ali podemos ver:

- Linhas em **cinza e branco**: não são linhas pertinentes para a cobertura
- Linhas em **verde**: são linhas que foram exercitadas pelos testes
- Linhas em **vermelho**: são linhas que **não foram** exercitadas pelos testes

Podemos ver que as linhas que estão em vermelho são as que não foram exercitadas pois nos testes foi verificado o comportamento de conseguir criar uma pessoa, mas o que acontece se não conseguirmos? Temos que fazer um teste verificando isso, primeiro, fazendo testes unitários para garantir que nem nome nem email serão permitidos vazios, em `test/models/person_test.rb`:

```

1 require 'test_helper'
2
3 class PersonTest < ActiveSupport::TestCase
4   setup do
5     @person = people(:one)
6   end
7
8   test 'nome não pode ser vazio' do
9     @person.name = ''
10    assert !@person.valid?
11  end
12
13  test 'email não pode ser vazio' do
14    @person.email = ''
15    assert !@person.valid?
16  end
17 end

```

Agora no modelo em `app/models/person.rb`:

```

1 class Person < ApplicationRecord
2   validates :name, presence: true
3   validates :email, presence: true
4 end

```

E finalmente no teste funcional:

```

1 test 'não cria pessoa' do
2   assert_no_difference('Person.count') do
3     post people_url, params: { person: { email: '', name: '' } }
4   end
5 end

```

Rodando os testes e dando uma olhada no arquivo `index.html` novamente, veremos que agora ele está com 93,94% de cobertura, com as linhas do método `create` devidamente cobertas por testes:

```

26. def create
27.   @person = Person.new(person_params)
28.
29.   respond_to do |format|
30.     if @person.save
31.       format.html { redirect_to @person, notice: "Person was successfully created." }
32.       format.json { render :show, status: :created, location: @person }
33.     else
34.       format.html { render :new }
35.       format.json { render json: @person.errors, status: :unprocessable_entity }
36.     end
37.   end
38. end

```

Cobertura do controlador de pessoas

Agora só faltam as linhas do método `update`, onde vamos testar o que acontece se tentarmos atualizar a pessoa com nome e email vazios. Mais um teste no teste funcional:

```

1 test "não atualiza pessoa" do
2   patch person_url(@person), params: { person: { email: '', name: '' } }
3   @person.reload
4   assert @person.name.size > 0
5   assert @person.email.size > 0
6 end

```

E pronto, todo nosso controlador está 100% coberto por testes! Se olharmos nossa suíte completa novamente, vamos verificar que ela alcançou 75% de cobertura de testes, sendo que não alcançou 100% ainda por causa da falta de cobertura do ActionCable, dos *jobs* e dos *mailers*. Vamos supor que nossa aplicação não vai ter utilizar nenhum desses recursos, podemos configurar o SimpleCov para ignorar esses recursos dessa forma, alterando no `test/test_helper.rb`:

```
1 require 'simplecov'  
2 SimpleCov.start 'rails' do  
3   add_filter 'app/channels'  
4   add_filter 'app/jobs'  
5   add_filter 'app/mailers'  
6 end
```

Utilizem os filtros com sabedoria, não os utilizando para ignorar partes que realmente devem ser testadas, só para “ficar bem na foto”! Rodando novamente os testes, podemos ver que agora nossa aplicação está com cobertura de 100% dos testes:



Cobertura completa

Guard

E o que acontece se estivermos rodando uma ferramenta como o Guard, que vimos anteriormente, para executar nossos testes de forma contínua? Será que somente o teste atual vai ser reavaliado (e consequentemente os resultados gerais também), será que vai ser disparado o procedimento de cobertura de testes para a aplicação inteira quando apenas um arquivo for alterado, deixando a suíte um pouco mais lenta (e lembrem-se: suíte de testes lenta não é legal), será que as diversas *gems* vão se comportar para orquestrar toda essa operação?

Como existem vários fatores envolvidos, uma opção interessante é desabilitar a cobertura de testes quando estivermos rodando o Guard, e como vamos ver, qualquer outro procedimento onde não seja interessante disparar a cobertura de testes. Para isso, vamos alterar o arquivo `test/test_helper.rb` para:

```
1 require 'simplecov'  
2  
3 unless ARGV.any? { |env| env =~ /guard/ }  
4   SimpleCov.start 'rails' do  
5     add_filter 'app/channels'  
6     add_filter 'app/jobs'  
7     add_filter 'app/mailers'  
8   end  
9 end
```

Nada mais ali do que verificar a linha de comando que está disparando a suíte de testes, e não disparando a coberta se houver guard presente por lá (quando rodamos o Guard, vai existir --guard). Dessa maneira, podemos deixar nosso aplicativo de teste contínuo **sem** a cobertura de testes, e rodar ela manualmente, afinal, não é toda hora que queremos e precisamos verificar a cobertura de testes. Assim, quando rodarmos os testes de forma manual, a cobertura é gerada:

```
1 $ RAILS_ENV=test be rails test  
2 Running via Spring preloader in process 16889  
3 Run options: --seed 36189  
4  
5 # Running:  
6  
7 .....  
8  
9 Finished in 0.352726s, 31.1857 runs/s, 39.6909 assertions/s.  
10 11 runs, 14 assertions, 0 failures, 0 errors, 0 skips  
11 Coverage report generated for MiniTest to /tmp/cov/coverage. 42 / 42 LOC (100%\  
12 .%) covered.
```

Mas quando rodamos o Guard, não:

```
1 RAILS_ENV=test be guard  
2 10:26:12 - INFO - Guard::Minitest 2.4.6 is running, with Minitest::Unit 5.10.\  
3 !  
4 10:26:12 - INFO - Running: all tests  
5 Run options: --guard --seed 84  
6  
7 # Running:  
8  
9 .....  
10  
11 Finished in 0.329904s, 33.3430 runs/s, 42.4366 assertions/s.  
12 11 runs, 14 assertions, 0 failures, 0 errors, 0 skips  
13 10:26:14 - INFO - Guard is now watching at '/tmp/cov'
```

Brakeman

O Brakeman é uma ferramenta que nos permite verificar nossa aplicação por vulnerabilidades de segurança. Por mais que o Rails nos dê proteção para várias vulnerabilidades relativas à uma aplicação web, muitas vezes os desenvolvedores relaxam por causa das facilidades já entregues, e ou não tentam entender tudo o que já está sendo entregue e escrevem código que anulam essas proteções, e pior, aumentam elas de forma explícita.

Lembrem-se que no final das contas o desenvolvedor é quem manda na ferramenta, e se por acaso alguém escrever código ruim em termos de segurança, muitas vezes vai receber um alerta do *framework*, que não vai se recusar o código, e em algumas, nem alertas são recebidos. Então, procurem entender o que já tem de proteção no Rails, como forma de aprendizado a não tentar anular esses procedimentos.

Instalando

Vamos fazer uma aplicação nova e instalar o Brakeman no Gemfile:

```
1 $ rails new brake
```

```
1 ...
2 group :development do
3   gem 'brakeman', :require => false
4 end
5 ...
```

Utilizando

Vamos fazer o *scaffold* tradicional, rodando as *migrations*:

```
1 $ rails g scaffold person name email
2
3 $ rails db:migrate
```

E agora rodar o Brakeman:

```

1 $ brakeman
2 == Brakeman Report ==
3
4 Application Path: /tmp/brake
5 Rails Version: 5.1.4
6 Brakeman Version: 4.0.1
7 ...
8 == Warning Types ==
9
10
11 No warnings found

```

Como é um código padrão gerado pelo Rails, não tem nenhum alerta de segurança. Vamos modificar o controlador de pessoas para introduzir uma má-prática:

```

1 def set_person
2   @person = Person.where("id = #{params[:id]}").first
3 end

```

Vejam que ali foi inserida uma consulta forçando o id recebido nos parâmetros direto na consulta. **Não façam isso.** O Brakeman já nos mostra a razão:

```

1 $ brakeman
2 == Brakeman Report ==
3 ...
4 == Warning Types ==
5
6 SQL Injection: 1
7
8 == Warnings ==
9
10 Confidence: High
11 Category: SQL Injection
12 Check: SQL
13 Message: Possible SQL injection
14 Code: Person.where("id = #{params[:id]}")
15 File: app/controllers/people_controller.rb
16 Line: 68

```

A alteração feita gerou uma falha de SQL Injection¹⁹⁸, deixando uma grande brecha para ser explorada na aplicação, onde podemos ver pelo *confidence level*, que está em *high*, ou seja, alto. O Brakeman utiliza 3 níveis de confiança:

1. **High** (alto): Um simples aviso ou o *input* do usuário está sendo utilizado de forma insegura.

¹⁹⁸https://pt.wikipedia.org/wiki/Inje%C3%A7%C3%A3o_de_SQL

2. **Medium** (médio): Geralmente indica o uso inseguro de uma variável, mas a variável pode ou não ser *input* do usuário.
3. **Weak** (fraco): Geralmente indica que o *input* do usuário foi indiretamente utilizado de forma insegura.

Para ver somente os alertas de um determinado nível *mínimo*, podemos utilizar a chave `-w`, indicando o nível desejado:

```
1 $ brakeman -w2
```

Guard

Existe uma *gem* para executar o Brakeman para o arquivo sendo testado pelo Guard. Vamos instalar o Guard e o guard-brakeman no Gemfile e rodar o bundler:

```
1 ...
2 group :test do
3   gem 'guard'
4   gem 'guard-minitest'
5   gem 'guard-brakeman'
6 end
7 ...
8
9 $ bundle
```

Criar (e depois configurar corretamente) o Guardfile com o Minitest e o Brakeman:

```
1 $ guard init minitest
2 $ guard init brakeman
```

Vamos deixar o arquivo dessa maneira:

```
1 guard :minitest do
2   watch(%r{^app/(.+)\.rb$}) { |m| "test/#{$m[1]}\\"
3     _test.rb" }
4   watch(%r{^app/controllers/application_controller\.rb$}) { 'test/controllers\\
5   ' }
6   watch(%r{^app/controllers/(.+)_controller\.rb$}) { |m| "test/integra\\
7   tion/#{$m[1]}_test.rb" }
8   watch(%r{^app/views/(.+)_mailer/.+}) { |m| "test/mailers\\
9     /#{$m[1]}_mailer_test.rb" }
10  watch(%r{^lib/(.+)\.rb$}) { |m| "test/lib/#{$m\\
11    [1]}_test.rb" }
12  watch(%r{^test/.+_test\.rb$})
```

```
13   watch(%r{^test/test_helper\.rb$}) { 'test' }
14 end
15
16 guard 'brakeman', run_on_start: true, quiet: true do
17   watch(%r{^app/.+\.(erb|haml|rhtml|rb)$})
18   watch(%r{^config/.+\.(rb)$})
19   watch(%r{^lib/.+\.(rb)$})
20   watch('Gemfile')
21 end
```

Rodando o Guard, temos:

```
1 $ RAILS_ENV=test bundle exec guard
2 12:06:49 - INFO - Guard::Minitest 2.4.6 is running, with Minitest::Unit 5.10. \
3 !
4 12:06:49 - INFO - Running: all tests
5 Run options: --guard --seed 31111
6
7 # Running:
8
9 .....
10
11 Finished in 0.385492s, 18.1586 runs/s, 23.3468 assertions/s.
12 7 runs, 9 assertions, 0 failures, 0 errors, 0 skips
13 12:06:52 - INFO - rocessed
14 > [#eac38865f3d8] ----- brakeman warnings -----
15 > [#eac38865f3d8]
16 12:06:52 - INFO - 1 brakeman findings
17 12:06:52 - INFO - High - SQL Injection - Possible SQL injection near line 68 \
18 in
19 /tmp/brake/app/controllers/people_controller.rb: Person.where("id = "
20 "#{params[:id]}")
21 12:06:52 - INFO - Guard is now watching at '/tmp/brake'
22 [1] guard(main)>
```

E agora, revertendo a alteração que fizemos de propósito no controlador de pessoas, o arquivo é avaliado de imediato:

```
1 12:08:44 - INFO - Running: test/controllers/people_controller_test.rb
2 Run options: --guard --seed 48313
3
4 # Running:
5
6 .....
7
8 Finished in 0.350290s, 19.9834 runs/s, 25.6930 assertions/s.
9 7 runs, 9 assertions, 0 failures, 0 errors, 0 skips
10 12:08:47 - INFO -
11 > [#]
12 > [#] rescanning ["app/controllers/people_controller.rb"], running all checks
13 12:08:47 - INFO -
14 > [#] ----- brakeman warnings -----
15 > [#]
```

O que nos mostra que está tudo ok.

Chartkick

Vamos retornar para a nossa aplicação da livraria e implementar alguns gráficos na nossa interface administrativa, para irmos acompanhando os pedidos na última semana. Para implementar os gráficos, vamos precisar da gem Chartkick.

Instalando

Vamos instalar a gem específica e mais uma auxiliar, que faz agrupamentos dos registros por data (essa é uma interessante de dar uma olhada mais tarde), inserindo no Gemfile:

```
1 gem 'chartkick'
2 gem 'groupdate'
```

Rodar o bundler:

```
1 $ bundle
2 ...
3 Fetching chartkick 2.2.5
4 Installing chartkick 2.2.5
5 ...
6 Fetching groupdate 3.2.0
7 Installing groupdate 3.2.0
8 ...
```

Agora precisamos selecionar uma *lib* para renderizar os gráficos. As opções são:

1. [Chart.js¹⁹⁹](#)
2. [Google Charts²⁰⁰](#)
3. [High Charts²⁰¹](#)

Vamos usar a primeira opção, e inserir no nosso arquivo `application.js`:

```
1 //= require Chart.bundle
2 //= require chartkick
```

E por último, como estamos utilizando o `SQLite` para desenvolvimento, a `gem Groupdate` tem suporte limitado para ele, sem suporte de *time zones*. Vamos desabilitar as *time zones* criando o arquivo `groupdate.rb` nos *initializers*, com o seguinte conteúdo:

```
1 Groupdate.time_zone = false unless Rails.env.production?
```

Utilizando

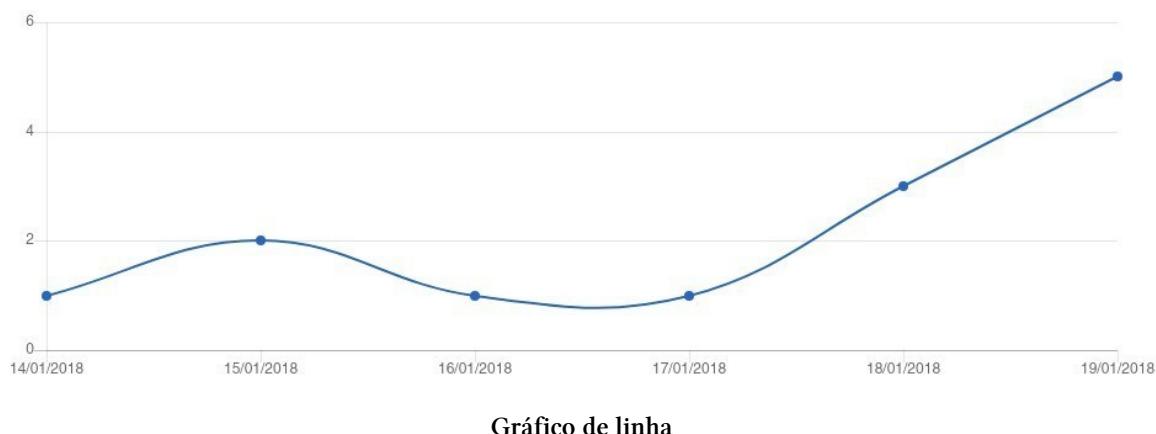
Para gerar os gráficos, vamos criar alguns pedidos com o atributo `created_at` variando entre a data corrente e alguns dias atrás. Podemos fazer isso utilizando algo como:

```
1 Order.create(person: Person.first, created_at: Time.current - 5.days)
```

Agora já temos dados para gerar os gráficos. Vamos escolher um lugar para renderizar um gráfico, por exemplo, podemos por no `layout` em `application.html.erb` para sempre serem apresentados no topo das páginas da interface de administração, fazendo com que sempre vamos ver como estão os pedidos a cada vez que recarregarmos ou irmos para outra página da administração. Para renderizar o gráfico, podemos inserir, onde quisermos que ele apareça:

```
1 <%= line_chart Order.group_by_day(:created_at).count %>
```

Isso vai produzir uma imagem como:



¹⁹⁹ <http://www.chartjs.org/>

²⁰⁰ <https://developers.google.com/chart/>

²⁰¹ <https://code.highcharts.com/highcharts.js>

Sério, é só inserir uma linha para gerar o gráfico!

Se quisermos um gráfico de pizza:

```
1 <%= pie_chart Order.group_by_day(:created_at, format: "%d/%m/%Y").count %>
```

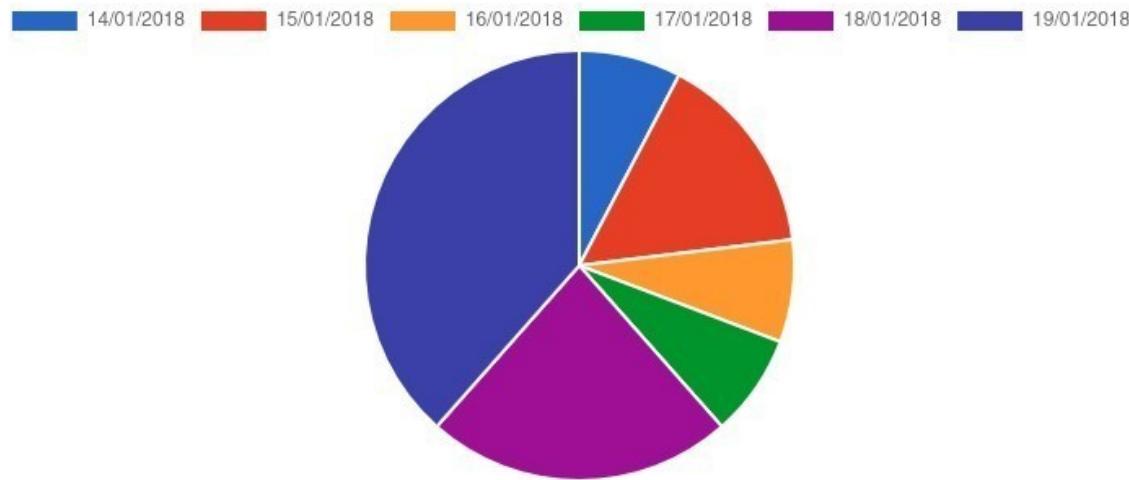


Gráfico de pizza

De coluna:

```
1 <%= column_chart Order.group_by_day(:created_at, format: "%d/%m/%Y").count %>
```

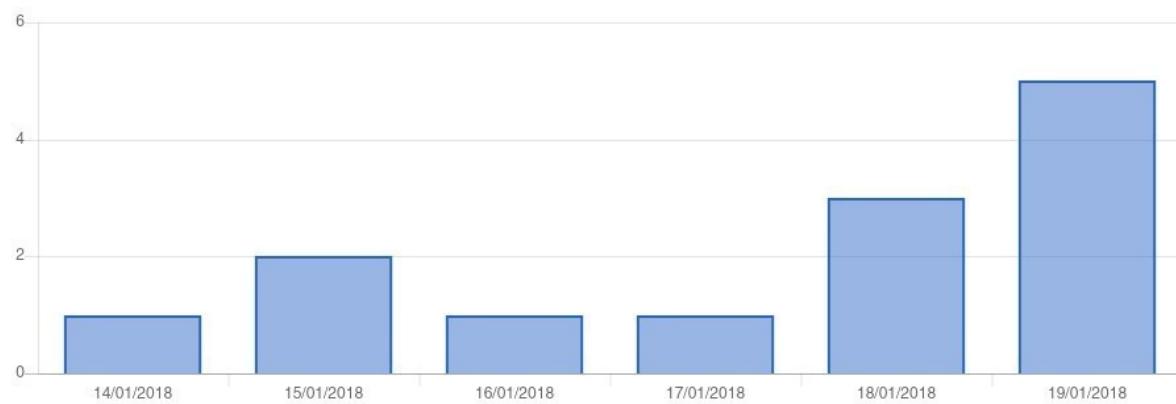


Gráfico de coluna

De barra:

```
1 <%= bar_chart Order.group_by_day(:created_at, format: "%d/%m/%Y").count %>
```

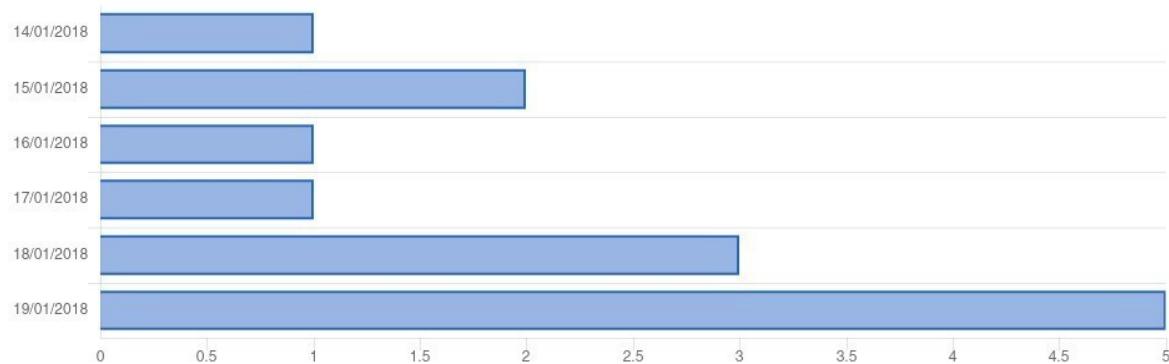


Gráfico de barra

E por aí vai. Existe documentação farta sobre os formatos de gráficos e customizações na [página do Chartkick²⁰²](#).

²⁰²<https://github.com/ankane/chartkick>

PaperTrail

Vimos o Active Record Dirty no livro, onde podemos verificar as alterações que foram feitas em um determinado objeto. Esse comportamento é utilizado pela gem PaperTrail²⁰³, que armazena as alterações que são feitas. Vamos fazer uma nova aplicação com um modelo simples para verificar como isso é feito.

Instalando

Vamos gerar uma pequena aplicação para testar:

```
1 $ rails new paper
2     create
3     create README.rdoc
4     create Rakefile
5     create config.ru
6     ...
7
```

Agora adicionar a gem no Gemfile, rodar o bundler, o generator da gem e a migration para criar a tabela necessária para armazenar as versões:

```
1 gem 'paper_trail'
2 ...
3 $ bundle
4 ...
5 Installing paper_trail 7.0.2
6 ...
7
8 $ rails g paper_trail:install
9     create db/migrate/20140506224444_create_versions.rb
10
11 $ rails db:migrate
12 == 20140506224444 CreateVersions: migrating =====\n13 ==
14 -- create_table(:versions)
15   -> 0.0019s
16 -- add_index(:versions, [:item_type, :item_id])
17   -> 0.0005s
18 == 20140506224444 CreateVersions: migrated (0.0026s) =====\n19 ==
```

Vamos fazer um modelo simples para verificar as alterações:

²⁰³https://github.com/airblade/paper_trail

```

1 $ rails g model Person name:string email:string
2   invoke  active_record
3   create    db/migrate/20140506224833_create_people.rb
4   create    app/models/person.rb
5   ...
6
7 $ rails db:migrate
8 == 20140506224833 CreatePeople: migrating ===== \\
9 ==
10 -- create_table(:people)
11   -> 0.0036s
12 == 20140506224833 CreatePeople: migrated (0.0037s) ===== \\
13 ==

```

Temos que indicar que o modelo tem suporte ao PaperTrail, utilizando o método `has_paper_trail`:

```

1 class Person < ActiveRecord::Base
2   has_paper_trail
3 end

```

Utilizando

Podemos brincar um pouco no console:

```

1 $ rails c
2 > p = Person.create(name: "Eustáquio Rangel", email: "taq@bluefish.com.br")
3
4 > p.versions.size
5   (0.3ms)  SELECT COUNT(*) FROM "versions" WHERE "versions"."item_id" = ? AND
6   "versions"."item_type" = ?  [[{"item_id": 1}, {"item_type": "Person"}]]
7 => 1

```

Podemos ver que o método `versions` nos retorna as versões que esse registro teve (no caso ali, usando `size` para contar quantas foram) e como isso é gerenciado internamente pelo PaperTrail, usando uma tabela com estrutura parecida com que a utilizamos anteriormente para o polimorfismo.

Podemos utilizar o método `live?` para verificar se a versão que temos é a versão corrente:

```

1 > p.paper_trail.live?
2 => true

```

Vamos fazer uma alteração e salvar, para ver outros métodos que podemos utilizar:

```
1 > p.update_attribute(:email, "eustaquiorangel@gmail.com")
2
3 > p.versions.size
4 => 2
5
6 > p.paper_trail.previous_version
7 => #<Person id: 1, name: "Eustaquio Rangel", email: "taq@bluefish.com.br",
8 created_at: "2014-05-06 23:34:00", updated_at: "2014-05-06 23:34:00">
```

Recuperando versões anteriores

```
1 > p.paper_trail.previous_version.save
2 => true
3
4 > p = Person.first
5 => "taq@bluefish.com.br"
```

ou

```
1 > p.paper_trail.version_at(30.minutes.ago).email
2 => "taq@bluefish.com.br"
```

Recuperando um registro apagado

```
1 > p.destroy
2 => #<Person id: 1, name: "Eustaquio Rangel", email: "taq@bluefish.com.br",
3 created_at: "2014-05-06 23:34:00", updated_at: "2014-05-06 23:34:00">
4
5 > Person.find(1) rescue nil
6 => nil
7
8 > PaperTrail::Version.where(item_type: 'Person', item_id: 1).last.reify.save
9 => true
10
11 p = Person.find(1)
12 => #<Person id: 1, name: "Eustaquio Rangel", email: "taq@bluefish.com.br",
13 created_at: "2014-05-06 23:34:00", updated_at: "2014-05-06 23:34:00">
```

Verificando quem fez as alterações

Para isso devemos criar um método `current_user` no `ApplicationController`, cujo valor retornado vai ser armazenado na coluna `whodunnit`, ou o método (também no `ApplicationController`) `user_for_paper_trail`, que vai ter a mesma finalidade:

```
1 def user_for_paper_trail
2   Person.first.name
3 end
```

Vejam que está sendo retornada uma String, mas podemos retornar qualquer tipo que seja representável como uma String, lembrando que no momento em que consumirmos essa informação, ela também retornará como uma String, ou seja, se gravarmos o id do usuário ao invés do nome, vamos ter que converter em um Fixnum antes de utilizar, por exemplo, o método `find` para encontrar a pessoa responsável pela alteração.

Para testarmos isso no console, podemos indicar usando a seguinte String que vai pegar o usuário do sistema operacional:

```
1 > PaperTrail.whodunnit = "#{`whoami`.strip}: console"
2
3 > p = Person.first
4
5 > p.update_attribute(:name, "Eustaquio Taq Rangel")
6 => true
7
8 > p.versions.last.whodunnit
9 => "taq: console"
```

Testando

Para os testes do sistema, é bom desabilitar o comportamento do `PaperTrail`, inserindo em `config/environments/test.rb`:

```
1 config.after_initialize do
2   PaperTrail.enabled = false
3 end
```

PaperClip

Agora vamos utilizar uma gem para fazer *upload* de arquivos, mais especificamente, imagens. Vamos utilizar duas como exemplo, e a primeira que vamos ver é o PaperClip²⁰⁴.

Instalação

O PaperClip precisa que o pacote do ImageMagick²⁰⁵ esteja instalado, então não se esqueça de verificar isso. No Ubuntu, podemos instalar com

```
1 $ sudo apt install imagemagick
```

Vamos novamente fazer uma aplicação vazia para testar essa gem, e já instalar ela no Gemfile e executar o bundler:

```
1 $ rails new clip
2     create
3     create  README.rdoc
4     create  Rakefile
5     ...
6 ...
7 gem 'paperclip'
8 ...
9 $ bundle install
10 ...
11 Installing cocaine 0.5.8
12 Installing paperclip 5.1.0
13 ...
```



Não se assustem com o nome da gem instalada acima, não é nada ilícito. Sabe-se lá a razão que deram esse nome. Tem algumas outras chamadas *lolita*, *tranny*, *clit*, *tit*, *texticle*, vai entender.

Vamos criar nosso scaffold customeiro, e rodar a migration:

²⁰⁴<https://github.com/thoughtbot/paperclip>

²⁰⁵<http://www.imagemagick.org/>

```
1 $ rails g scaffold Person name:string email:string
2     invoke  active_record
3     create    db/migrate/20140507230947_create_people.rb
4     ...
5
6 $ rails db:migrate
7 == 20140507230947 CreatePeople: migrating =====\\
8 ==
9 -- create_table(:people)
10    -> 0.0009s
11 == 20140507230947 CreatePeople: migrated (0.0010s) =====\\
12 ==
```

Associando uma imagem com o modelo

Vamos inserir no modelo a informação que ele tem um arquivo anexo no atributo chamado `image`, que tem o tamanho médio de 300x300 *pixels* e um *thumbnail* de 100x100 *pixels*, validando que o *content type* do arquivo enviado deve ser uma imagem:

```
1 class Person < ActiveRecord::Base
2   has_attached_file :image,
3     styles: { medium: "300x300>", thumb: "100x100>" },
4     default_url: "/images/:style/missing.png"
5
6   validates_attachment_content_type :image, content_type: /\Aimage\/.*\Z/
7 end
```

Precisamos de uma coluna no modelo para armazenar a imagem. Para isso, vamos fazer a seguinte `migration`:

```
1 $ rails g migration AddImageToPeople
2     invoke  active_record
3     create    db/migrate/20140507234243_add_image_to_people.rb
```

Conteúdo da `migration`:

```

1 class AddImageToPeople < ActiveRecord::Migration[5.0]
2   def up
3     add_attachment :people, :image
4   end
5
6   def down
7     remove_attachment :people, :image
8   end
9 end

```

Rodando:

```

1 $ rails db:migrate
2 == 20140507234243 AddImageToPeople: migrating =====\\
3 ==
4 -- add_attachment(:people, :image)
5   -> 0.0015s
6 == 20140507234243 AddImageToPeople: migrated (0.0016s) =====\\
7 ==

```

Adaptando o formulário para envio da imagem

Do mesmo modo que utilizamos quando alteramos o formulário quando fizemos o *upload* “na unha”, vamos indicar agora que o formulário aceita o envio de arquivos, utilizando `multipart`, e inserir o elemento HTML para fazer upload:

```

1 <%= form_with(model: person, local: true, html: { multipart: true }) do |form|
2 | %>
3   <% if @person.errors.any? %>
4     ...
5   <div class="field">
6     <%= form.file_field :image %>
7   </div>
8   <div class="actions">
9     <%= form.submit %>
10  </div>
11 <% end %>

```

Alterar os *strong parameters* do controlador de pessoas para aceitar o envio da imagem:

```

1 def person_params
2   params.require(:person).permit(:name, :email, :image)
3 end

```

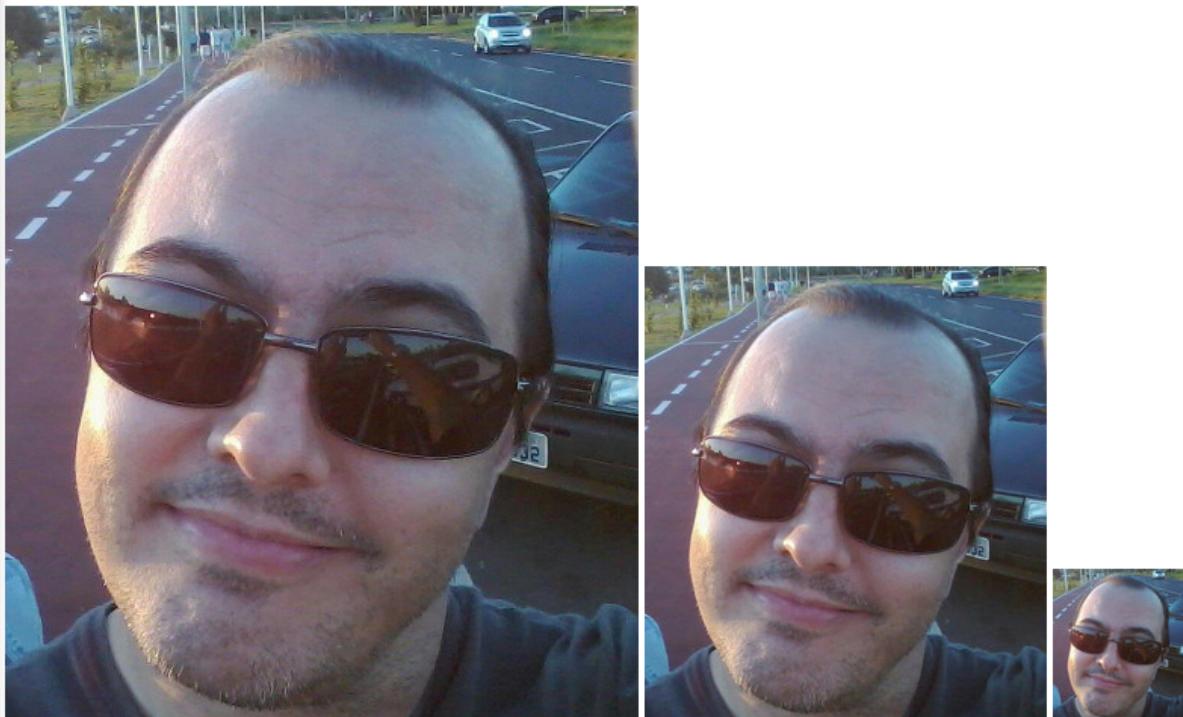
E modificar a view show para exibir a imagem:

```
1 <p id="notice"><%= notice %></p>
2
3 <p>
4   <strong>Nome:</strong>
5   <%= @person.name %>
6 </p>
7
8 <p>
9   <strong>Email:</strong>
10  <%= @person.email %>
11 </p>
12
13 <%= image_tag @person.image.url %>
14 <%= image_tag @person.image.url(:medium) %>
15 <%= image_tag @person.image.url(:thumb) %>
16
17 <br style="clear:both;" />
18 <%= link_to 'Editar', edit_person_path(@person) %> |
19 <%= link_to 'Voltar', people_path %>
```

Que deve resultar em algo como a seguinte imagem (ei, eu não sou metido não, é que usando imagens minhas não preciso pagar direito autoral para ninguém ;-):

Nome: Eustáquio Rangel

Email: taq@bluefish.com.br



[Editar](#) | [Voltar](#)

Upload com Paperclip

Essas imagens foram gravadas no seguinte diretório, que tem 3 sub-diretórios dentro dele, de acordo com o tamanho da imagem:

```
1 $ ls public/system/people/images/000/000/001/
2 total 20K
3 drwxrwxr-x 5 taq taq 4,0K Mai  7 21:16 .
4 drwxrwxr-x 3 taq taq 4,0K Mai  7 21:16 ..
5 drwxrwxr-x 2 taq taq 4,0K Mai  7 21:16 medium
6 drwxrwxr-x 2 taq taq 4,0K Mai  7 21:16 original
7 drwxrwxr-x 2 taq taq 4,0K Mai  7 21:16 thumb
```

Esse caminho é baseado em configurações de *deploy* do Capistrano, e segue o formato :rails_-root/public/system/:class/:attachment/:id_partition/:style/:filename.

Validações

Podemos fazer validações como:

```
1 validates_attachment :image, presence: true,  
2   content_type: { content_type: "image/jpeg" },  
3   size: { in: 5..10.kilobytes }
```

Que especificam:

1. Deve ter uma imagem presente
2. O tipo da imagem tem que ser jpeg
3. O tamanho da imagem deve ser entre 5 e 10 *kilobytes*

Mais opções podem ser encontradas no site do PaperClip.

Apagando um anexo

É simples, é só deixar com nulo (`nil`) e salvar o modelo:

```
1 person.image = nil  
2 person.save
```

Armazenamento

O PaperClip vem com alguns métodos de *storage*:

1. Armazenamento em arquivos locais (*file storage*, que acabamos de utilizar)
2. S3 Storage, enviando os arquivos para a Amazon S3 ²⁰⁶, utilizando a gem `aws-sdk` ²⁰⁷
3. Fog Storage, enviando os arquivos para o Fog ²⁰⁸
4. Dropbox, enviando os arquivos para a Dropbox ²⁰⁹, através da gem `paperclip-dropbox` ²¹⁰
5. Windows Azure, através da gem `paperclip-azure-storage`^{211 212}

A utilização é bem fácil. Como exemplo, podemos utilizar o S3 e configurá-lo no arquivo do ambiente desejado, como por exemplo, `config/environments/production.rb`:

²⁰⁶<http://aws.amazon.com/pt/s3/>

²⁰⁷<https://github.com/aws/aws-sdk-ruby>

²⁰⁸<http://fog.io/>

²⁰⁹<http://dropbox.com>

²¹⁰<https://github.com/janko-m/paperclip-dropbox>

²¹¹<https://github.com/gmontard/paperclip-azure-storage>

²¹²<https://github.com/gmontard/paperclip-azure-storage>

```
1 config.paperclip_defaults = {  
2   storage: :s3,  
3   s3_region: 'us-east',  
4   s3_credentials: {  
5     bucket: 'bucket_name',  
6     access_key_id: 'access_key_id',  
7     secret_access_key: 'secret_access_key'  
8   }  
9 }
```

Informações mais precisas e detalhadas podem ser encontradas nos sites de cada *storage*.

CarrierWave

Como alternativa ao PaperClip, podemos utilizar a gem CarrierWave²¹³, que possui algumas opções de customizações interessantes.

Instalando

Para instalar, vamos repetir o mesmo procedimento do que foi feito para o PaperClip, desde criar uma nova aplicação (chamei aqui de carrier) até o scaffold, apenas substituindo o nome da gem para carrierwave, não esquecendo de inserir também a `gem mini_magick`.

Configurando

Após tudo instalado, vamos precisar criar um uploader, que é um conceito do CarrierWave para indicar uma classe que vai armazenar arquivos em uma determinada *storage*.

Vamos criar um uploader chamado Image:

```
1 $ rails g uploader Image
2       create  app/uploaders/image_uploader.rb
```

Que, como vemos, gerou um arquivo novo em `app/uploaders/image_uploader.rb`, com conteúdo como esse, especificando para utilizar o CarrierWave com o MiniMagick, que já especificamos no Gemfile acima:

```
1 class ImageUploader < CarrierWave::Uploader::Base
2
3   # Include RMagick or MiniMagick support:
4   # include CarrierWave::RMagick
5   include CarrierWave::MiniMagick
6
7   # Choose what kind of storage to use for this uploader:
8   storage :file
9   ...
```

Vamos alterar o modelo Person (vocês criaram o scaffold, correto?) para adicionar a imagem:

²¹³<https://github.com/carrierwaveuploader/carrierwave>

```
1 $ rails g migration AddImageToPeople image:string
2       invoke  active_record
3       create    db/migrate/20140508135059_add_image_to_people.rb
4
5 $ rails db:migrate
6 == 20140508135059 AddImageToPeople: migrating =====\\
7 ==
8 -- add_column(:people, :image, :string)
9   -> 0.0070s
10 == 20140508135059 AddImageToPeople: migrated (0.0080s) =====\\
11 ==
```

Alterar o modelo para utilizar o uploader:

```
1 class Person < ActiveRecord::Base
2   mount_uploader :image, ImageUploader
3 end
```

Agora podemos utilizar o console para criar um registro com uma imagem:

```
1 > p = Person.create(name: "Eustáquio Rangel", email: "taq@bluefish.com.br")
2 => #<Person id: 1, name: "Eustáquio Rangel", email: "taq@bluefish.com.br",
3 created_at: "2014-05-08 13:54:01", updated_at: "2014-05-08 13:54:01", image:
4 nil>
5
6 > p.image = File.open("/home/taq/Imagens/taq.jpg")
7 => #<File:/home/taq/Imagens/taq.jpg>
8
9 > p.save
10 => true
11
12 > p.image.url
13 => "/uploads/person/image/1/taq.jpg"
14
15 > p.image.current_path
16 => "/home/taq/git/conhecendorails/carrier/public/uploads/person/image/1/taq.\
17 jpg"
18
19 > p.image.identifier
20 => "taq.jpg"
```

Como pudemos ver, o arquivo foi salvo em /uploads/person/image/1/, que é onde o método storage_dir do uploader indica:

```
1 def store_dir
2   "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
3 end
```

Criando versões

Podemos especificar o redimensionamento da imagem e versões diferentes do arquivo, como por exemplo, os *thumbnails*. Para isso, especificamos as versões no uploader. Para isso, vamos precisar também do ImageMagick instalado no sistema operacional e o MiniMagick (já inserido no Gemfile):

```
1 class ImageUploader < CarrierWave::Uploader::Base
2   include CarrierWave::MiniMagick
3   storage :file
4
5   def store_dir
6     "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
7   end
8
9   process resize_to_fit: [500, 500]
10  version :thumb do
11    process resize_to_fill: [200,200]
12  end
13  ...
```

O que nos dá, salvando a imagem novamente:

```
1 p = Person.first
2 => #<Person id: 1, name: "Eustaquio Rangel", email: "taq@bluefish.com.br",
3 created_at: "2014-05-08 13:54:01", updated_at: "2014-05-08 13:54:26", image:
4 "taq.jpg">
5
6 p.image = File.open("/home/taq/Imagens/taq.jpg")
7 => #<File:/home/taq/Imagens/taq.jpg>
8
9 p.save
10 => true
11
12 p.image.thumb.url
13 => "/uploads/person/image/1/thumb_taq.jpg"
```

Mostrando os resultados:

```

1 $ ls public/uploads/person/image/1/
2 total 60K
3 drwxr-xr-x 2 taq taq 4,0K .
4 drwxr-xr-x 3 taq taq 4,0K ..
5 -rw-r--r-- 1 taq taq 38K taq.jpg
6 -rw-r--r-- 1 taq taq 8,8K thumb_taq.jpg

```

Formulários e views

Já que já criamos um usuário e algumas imagens, vamos alterar a view show primeiro:

```

1 <p id="notice"><%= notice %></p>
2
3 <p>
4   <strong>Nome:</strong>
5   <%= @person.name %>
6 </p>
7
8 <p>
9   <strong>Email:</strong>
10  <%= @person.email %>
11 </p>
12
13 <%= image_tag(@person.image.url) %>
14 <%= image_tag(@person.image.thumb.url) %>
15
16 <%= link_to 'Editar', edit_person_path(@person) %> |
17 <%= link_to 'Voltar', people_path %>

```

O que nos dá uma página similar a mesma página exibida no PaperClip (a atual só tem 2 imagens, a imagem original e o *thumbnail*).

Agora vamos alterar o formulário:

```

1 <%= form_with(model: person, local: true, html: { multipart: true }) do |form|
2 | %>
3 ...
4   <div class="field">
5     <%= form.file_field :image %>
6     <%= form.hidden_field :image_cache %>
7     <%= form.check_box :remove_image %> Remover imagem
8   </div>
9 ...

```

Não esquecendo de liberar os atributos nos *strong parameters* do controlador:

```
1 def person_params
2   params.require(:person).permit(:name, :email, :image, :image_cache, :remove\
3   _image)
4 end
```

Alguns atributos interessantes ali são o elemento escondido (`hidden_field`) `image_cache`, que mantém o arquivo selecionado no caso de dar algum problema nas validações do modelo, e a checkbox para remover a imagem.

Temos também várias opções de *storages* com o CarrierWave. Para mais detalhes, consultar o site da gem.

Bootstrap

O Bootstrap é, como definido no seu site ²¹⁴, “uma estrutura de front-end e elegante, intuitiva e poderosa para o desenvolvimento web mais rápido e fácil, criado por Mark Otto e Jacob Thornton, e mantida pela equipe principal com o apoio maciço e envolvimento da comunidade”. Ele é um *framework* para o *front-end*, sendo uma coleção de estilos CSS e código JavaScript (através do jQuery, já disponível com o Rails) utilizados para agilizar o desenvolvimento da interface de uma página web.

Instalando

Vamos fazer uma nova aplicação chamada bstrap:

```
1 $ rails new bstrap
2   create
3   create  README.rdoc
4   create  Rakefile
5   create  config.ru
6   ...
7
```

Alterar o Gemfile para inserir a gem bootstrap-sass e rodar o bundler:

```
1 gem 'bootstrap-sass'
2 ...
3 $ bundle
4 Fetching gem metadata from https://rubygems.org/.....
5 Fetching additional metadata from https://rubygems.org/..
6 Resolving dependencies...
7 ...
8 Installing bootstrap-sass 3.3.7
9 ...
```

Importar o bootstrap no arquivo app/assets/stylesheets/bstrap.scss (**não** chamem esse arquivo de bootstrap.scss senão vai ocorrer um *loop*, chamem de qualquer outra coisa!):

```
1 @import "bootstrap-sprockets";
2 @import "bootstrap";
```

e no arquivo app/assets/javascripts/application.js:

²¹⁴<http://getbootstrap.com>

```
1 //= require bootstrap-sprockets
```

Utilizando

Agora podemos criar o nosso scaffold costumeiro para ver como a aplicação vai ficar:

```
1 $ rails g scaffold Person name:string email:string
2     invoke  active_record
3     create    db/migrate/20140521002222_create_people.rb
4     create    app/models/person.rb
5     ...
6
7 $ rails db:migrate
8 == 20140521002222 CreatePeople: migrating ===== \\
9 ==
10 -- create_table(:people)
11   -> 0.0010s
12 == 20140521002222 CreatePeople: migrated (0.0011s) ===== \\
13 ==
```

E **remover** o arquivo gerado scaffold.scss para não conflitar com os estilos do bootstrap:

```
1 $ rm app/assets/stylesheets/scaffolds.scss
```

E agora alterar o *layout* da aplicação em app/views/layouts/application.html.erb para utilizar as classes do bootstrap:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Bootstrap test</title>
5   <%= csrf_meta_tags %>
6   <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track': 'reload' %>
7   <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
8 </head>
9 <body>
10 <header>
11   <nav>
12     <ul class="nav nav-tabs navbar-inverse">
13       <li class="<%=' active' if request.original_url =~ '/people/' %>">
14         <%= link_to "Pessoas", people_path %>
15       </li>
```

```
18      </ul>
19      </nav>
20  </header>
21  <section class="container">
22    <%= yield %>
23  </section>
24</body>
25</html>
```

E o index do controlador people em app/views/people/index.html.erb:

```
1 <p id="notice"><%= notice %></p>
2
3 <h1>Listando pessoas</h1>
4
5 <table class='table table-striped'>
6   <thead>
7     <tr>
8       <th>Nome</th>
9       <th>Email</th>
10      <th colspan="3"></th>
11    </tr>
12  </thead>
13
14  <tbody>
15    <% @people.each do |person| %>
16      <tr>
17        <td><%= person.name %></td>
18        <td><%= person.email %></td>
19        <td><%= link_to 'Mostrar', person, class: 'btn' %></td>
20        <td><%= link_to 'Editar', edit_person_path(person), class: 'btn' %><%
21      td>
22          <td><%= link_to 'Apagar', person, method: :delete, data: { confirm: '\
23 Tem certeza?' }, class: 'btn btn-danger' %></td>
24      </tr>
25    <% end %>
26  </tbody>
27</table>
28
29 <br>
30 <%= link_to 'Nova pessoa', new_person_path, class: 'btn btn-primary' %>
```

E o formulário:

```
1 <%= form_with(model: person, local: true) do |form| %>
2   <% if person.errors.any? %>
3     <div id="error_explanation">
4       <h2><%= pluralize(person.errors.count, "error") %> prohibited this pers\on from being saved:</h2>
5
6       <ul>
7         <% person.errors.full_messages.each do |message| %>
8           <li><%= message %></li>
9         <% end %>
10      </ul>
11    </div>
12  <% end %>
13
14  <div class="field">
15    <%= form.label :name %>
16    <%= form.text_field :name, class: 'form-control' %>
17  </div>
18
19
20  <div class="field">
21    <%= form.label :email %>
22    <%= form.text_field :email, class: 'form-control' %>
23  </div>
24
25  <div class="actions">
26    <%= form.submit class: 'btn btn-primary' %>
27  </div>
28 <% end %>
```

Isso vai nos dar uma imagem como essa:

The screenshot shows a web application interface. At the top left, there is a navigation bar with a 'Pessoas' button. Below the navigation bar, the title 'Listando pessoas' is displayed. A table follows, listing two individuals: Eustáquio Rangel and Ana Carolina, along with their respective emails and actions (Mostrar, Editar, Apagar). At the bottom left, a blue button labeled 'Nova pessoa' is visible.

Nome	Email	Mostrar	Editar	Apagar
Eustáquio Rangel	taq@bluefish.com.br	Mostrar	Editar	Apagar
Ana Carolina	carol@bluefish.com.br	Mostrar	Editar	Apagar

Já deu uma cara melhor, não é mesmo? E apenas utilizamos as classes `nav`, `nav-tabs`, `navbar-inverse`, `container`, `table`, `table-striped`, `btn` e `btn-danger`. Para um guia completo das classes que podem ser utilizadas com o bootstrap (e são muitas opções!), podemos consultar o “Getting Started” do mesmo.²¹⁵ Como dica inicial e rápida, não esqueçam de inserir a classe `form-control` nos elementos dos formulários para deixar com uma cara mais bonita, levando em conta que o tamanho deles a partir do Bootstrap 3 sempre vai ser 100% do tamanho do `container`.

²¹⁵<http://getbootstrap.com/2.3.2/getting-started.html>

Tmux

Enquanto não é relacionado diretamente à programação, o Tmux²¹⁶ é uma ferramenta bem essencial no meu dia-a-dia e acredito em no dia-a-dia de muitos outros desenvolvedores. Antes dele, eu utilizava o GNU Screen²¹⁷, mas o Tmux se provou mais potente, flexível e atualizado mais frequentemente.

O Tmux é um multiplexador de terminais, e utilizei basicamente para algumas finalidades como:

1. Trabalhar em um servidor remoto, via SSH, mantendo todo o ambiente que estava trabalhando quando eu desconecto, permitindo que eu volte no mesmo ponto quando conectar novamente.
2. Rodar alguns serviços que não sejam *daemons* no servidor, deixando-os ativos quando eu desconectar.
3. Criar um ambiente onde eu possa desenvolver (ah-há!) utilizando o meu editor de texto de preferência (Vim, sempre!), rodar o servidor web, rodar os testes (seja utilizando a opção padrão que vem com o Rails ou o Minitest), rodar um terminal etc.

É na terceira opção que fica a dica aqui. Antes de abrir trecentas janelas diferentes de um terminal e ficar perdido sem saber onde está cada uma, ou abrir várias abas em um mesmo terminal, a dica é abrir uma sessão nova do Tmux onde dá para abrir várias “abas” e inclusive dividir a tela na horizontal ou vertical, se precisar. Algo mais ou menos assim:

The screenshot shows a Tmux session with two panes. The left pane displays a file tree of a Ruby on Rails application structure. The right pane shows a Vim editor with some Ruby code for a 'Order' model. The bottom status bar shows the command history, current window, date and time.

```

taq@neo:/home/taq/git/conhecendorails
▶ controllers/
▶ helpers/
▶ mailers/
▼ models/
  ▶ concerns/
    book.rb
    book_category.rb
    cart.rb
    category.rb
    image.rb
    order.rb
    order_item.rb
    person.rb
  ▶ presenters/
  ▶ sweepers/
  ▶ views/
  ▶ workers/
▶ bin/
▶ config/
▶ db/
▶ lib/
NERD
1 alteração; antes de #1 2 segundos atrás
neo rails vim 1 vim 2 server 3 tests 4 console Dom Jul 6 10:45:25 BRT 2014 taq

```

```

0 class Order < ActiveRecord::Base
1   belongs_to :person
2   validates :person_id, presence: true
3   has_many :order_items
4   has_many :items, class_name: "OrderItem", foreign_key>
5
6   def total
7     order_items.inject(0) { |m,i| m += i.value}
8   end
9
10  def self.create(person_id, cart, locking, item_ref)
11    order = Order.new(person_id: person_id)
12    locking.transaction do
13      for item in cart.items
14        order.order_items << item_ref.new(book_id: item>
15        item.reload.sell
16      end
17      order.save ? order : nil
18    end
19  end
20 end

```

NORMAL > <els/order.rb Order < ruby 4% 1: 2

Tmux

²¹⁶<http://tmux.sourceforge.net/>

²¹⁷<http://www.gnu.org/software/screen/>

Reparam no rodapé, onde são mostrados os nomes das “abas” que tenho abertas dentro da minha sessão do Tmux. Cada nova janela pode ser aberta digitando `ctrl-b c`, dá para navegar para cada janela específica utilizando `ctrl-b <número>`, ou para frente utilizando `ctrl-b n` e para trás utilizando `ctrl-b p`.

Falar do Tmux de uma maneira mais abrangente requer um livro à parte (inclusive, ele já existe²¹⁸) mas dá para ter uma boa introdução ao mesmo em alguns excelentes artigos que existem na web^{219 220 221}.

Tmuxinator

Para automatizar a configuração de uma sessão do Tmux, temos a ótima gem tmuxinator. Para instalar, o método tradicional de sempre para instalar uma gem:

```
1 $ gem install tmuxinator
```

Podemos criar projeto novo, chamado, por exemplo, `sample`, da seguinte forma:

```
1 $ tmuxinator new sample
```

Isso vai gerar um arquivo chamado `~/.tmuxinator/sample.yml` (onde o nome do arquivo é `sample.yml` e fica no diretório “escondido” `.tmuxinator`, abaixo do `home`) com o seguinte conteúdo:

```
1 name: sample
2 root: ~/
3 windows:
4   - editor:
5     layout: main-vertical
6     panes:
7       - vim
8       - guard
9     - server: bundle exec rails s
10    - logs: tail -f log/development.log
```

Se executarmos agora:

```
1 $ tmuxinator sample
```

²¹⁸<http://pragprog.com/book/bhtmx/tmux>

²¹⁹<http://robots.thoughtbot.com/a-tmux-crash-course>

²²⁰<http://blog.hawkhost.com/2010/06/28/tmux-the-terminal-multiplexer/>

²²¹<http://www.sitepoint.com/tmux-a-simple-start/>

Vai gerar uma sessão parecida com essa:

The screenshot shows a terminal window titled "taq@neo:/home/taq". It has a vertical split pane on the left and a single pane on the right. The right pane displays the command "\$ guard" and the timestamp "[taq@neo ~] 11:16:25". The bottom of the window features a tab bar with several tabs: "NORMAL > [Sem nome][+]", "100% n_L 1: 0", "neo", "sample", "editor", "1 editor", "2 server", "3 logs", and the date/time "Dom Jul 6 11:16:39 BRT 2014 taq". A green vertical bar is visible on the right side of the terminal window.

Tmuxinator

Que já nos economiza bastante digitação.

ElasticSearch

Lembram do problema que tivemos com o *autocomplete* e a famigerada cláusula LIKE? Um dos jeitos que podemos resolver o problema de uma forma mais padronizada, e melhor, otimizada, é utilizar o [ElasticSearch](#)²²² ²²³.

O que é o ElasticSearch

O ElasticSearch é, segundo a descrição em seu próprio site, “um *engine* de código aberto flexível, poderoso e distribuído de análise e busca em tempo real, que dá a habilidade de mover facilmente além da simples busca de texto”. Ele é orientado a documentos estruturados no formato JSON, provê uma API RESTful rodando sob o protocolo HTTP e é escrito em Java, não significando que temos que conhecer alguma coisa de Java para que possamos o utilizar.

Realmente, utilizar simples buscas de texto é utilizar somente o recurso mais básico (mas que também é o mais utilizado) do ElasticSearch, sendo que existem livros inteiros dedicados em como utilizar os seus outros variados e poderosos recursos. Mas vamos focar aqui em como procurar as palavras que precisamos para o *autocomplete* e utilizar somente mesmo essa camada superficial. Se vocês gostarem, podem procurar mais sobre o ElasticSearch na web ou nos livros especializados.

Lembrando que a escolha do ElasticSearch é mais um serviço a ser aprendido e gerenciado. Convém analisar se a criação de índices como os trigrams do PostgreSQL, apresentados anteriormente, não seriam uma melhor opção, levando em conta utilizar um recurso específico do banco (e que outros bancos, de forma mais fácil ou difícil, podem ter) ou a implementação de uma nova “camada” na aplicação como o ElasticSearch. Essa é outra área em que as pessoas tomam decisões rápidas sem analisar tudo o que a ferramenta atual oferece e por muitas vezes começam a complicar desnecessariamente a estrutura. Algumas pessoas pregam a utilização de bancos para escrita e leitura (usando o ElasticSearch nesse último caso) sem nem ter ao menos tentado otimizar a performance das consultas e do banco de dados que pode servir perfeitamente (como sempre serviu) para fazer todo o trabalho necessário.

Instalando o ElasticSearch

Os pacotes do ElasticSearch são fornecidos de algumas formas:

- Arquivos .zip
- Arquivos .tar.gz
- Arquivos .deb, para distribuições Linux como o Debian e o Ubuntu
- Arquivos .rpm, para distribuições Linux como o Fedora e o CentOS
- Reppositórios compatíveis com o APT, para distribuições como o Debian e o Ubuntu
- Reppositórios compatíveis com o YUM, para distribuições Linux como o Fedora e o CentOS
- Pacotes do sistema operacional

²²²<http://www.elasticsearch.org/>

²²³<http://www.elasticsearch.org/>

Particulamente eu prefiro a instalação por repositórios, e vamos utilizar aqui os repositórios do Ubuntu. Para isso, primeiro temos que fazer *download* e instalar a chave pública do repositório utilizando

```
1 $ wget -qO - https://packages.elastic.co/GPG-KEY-elasticsearch | sudo apt-key\\
2 add -
```

Vamos adicionar o repositório na lista de fontes, inserindo a seguinte linha no final do arquivo `/etc/apt/sources.list.d/elasticsearch-2.x.list`:

```
1 echo "deb http://packages.elastic.co/elasticsearch/2.x/debian stable main" | \\
2 sudo tee -a /etc/apt/sources.list.d/elasticsearch-2.x.list
```

E agora atualizar a lista de pacotes e instalar o ElasticSearch:

```
1 $ sudo apt update
2 $ sudo apt install elasticsearch
```

Vamos abrir o arquivo `/etc/elasticsearch/elasticsearch.yml` e indicar, removendo o comentário da linha ou inserindo a linha, que queremos rodar o serviço no localhost:

```
1 network.host: localhost
```

Isso já deve disponibilizar o ElasticSearch como um serviço do sistema operacional:

```
1 $ sudo service elasticsearch status
2   * elasticsearch is not running
3
4 $ sudo service elasticsearch start
5   * Starting Elasticsearch Server
6   [ OK ]
7
8 $ sudo service elasticsearch status
9   * elasticsearch is running
10
11 $ sudo service elasticsearch stop
12   * Stopping Elasticsearch Server
13   [ OK ]
14
15 $ sudo service elasticsearch status
16   * elasticsearch is not running
```

Vamos inicializar o serviço e verificar se está tudo ok fazendo uma requisição HTTP utilizando o cURL^{224 225}:

²²⁴<http://curl.haxx.se/>

²²⁵<http://curl.haxx.se/>

```
1 $ sudo service elasticsearch start
2 * Starting Elasticsearch Server [ OK ]
3 ]
4
5 $ curl -X GET http://localhost:9200/
6 {
7   "name" : "Blizzard",
8   "cluster_name" : "elasticsearch",
9   "cluster_uuid" : "8S3ZNY5ATxKmvvQZ62zOfw",
10  "version" : {
11    "number" : "2.4.5",
12    "build_hash" : "c849dd13904f53e63e88efc33b2ceeda0b6a1276",
13    "build_timestamp" : "2017-04-24T16:18:17Z",
14    "build_snapshot" : false,
15    "lucene_version" : "5.5.4"
16  },
17  "tagline" : "You Know, for Search"
18 }
```

Tudo ok!

Utilizando o ElasticSearch com o Rails

Para utilizar o ElasticSearch com o Rails, vamos retornar ao nosso projeto da livraria (vocês não apagaram ele, apagaram?) e vamos precisar inserir duas gems no Gemfile dele:

```
1 gem 'elasticsearch-model'
2 gem 'elasticsearch-rails'
3
4 $ bundle
5 ...
6 Installing elasticsearch 5.0.4
7 Installing elasticsearch-model 5.0.1
8 Installing elasticsearch-rails 5.0.1
9 ...
```

Agora vamos indicar no nosso modelo Book que queremos utilizar o ElasticSearch nele, devendo remover o método `search` que criamos lá pois o ‘ElasticSearch’ vai nos dar um método com o mesmo nome:

```

1 require 'elasticsearch/model'
2 ...
3
4 class Book < ActiveRecord::Base
5   include Elasticsearch::Model
6   include Elasticsearch::Model::Callbacks
7 ...

```

Vamos testar no console do Rails para ver se deu certo, **comentando o método search já existente em Book:**

```

1 books = Book.search('git')
2 => #<Elasticsearch::Model::Response::Response:0x000000048b13f8 @klass=[PROXY]
3 Book (call 'Book.connection' to establish a connection),
4 @search=#<Elasticsearch::Model::Searching::SearchRequest:0x000000048b30e0
5 @klass=[PROXY] Book (call 'Book.connection' to establish a connection),
6 @definition={:index=>"books", :type=>"book", :q=>"git"}>>

```

Ops, parece que o método `search`, que agora é provido pelo `ElasticSearch` está lá, mas ainda está faltando alguma coisa. O que vamos precisar aqui, já que já existiam dados no banco e acabamos de instalar o `ElasticSearch` no modelo, é o método `import`:

```

1 Book.import(force: true)
2 Book Load (0.1ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" ASC\ 
3 LIMIT 1000

```

Testando novamente:

```

1 books = Book.search('git')
2 => #<Elasticsearch::Model::Response::Response:0x000000031475c8 @klass=[PROXY]
3 Book(id: integer, title: string, published_at: date, text: text, value: 
4 decimal, person_id: integer, created_at: datetime, updated_at: datetime, sto\ 
5 ck: 
6 integer, lock_version: integer),
7 @search=#<Elasticsearch::Model::Searching::SearchRequest:0x000000031476e0
8 @klass=[PROXY] Book(id: integer, title: string, published_at: date, text: te\ 
9 xt,
10 value: decimal, person_id: integer, created_at: datetime, updated_at: dateti\ 
11 me,
12 stock: integer, lock_version: integer), @definition={:index=>"books", 
13 :type=>"book", :q=>"git"}>>

```

Opa, agora aparentemente deu certo! É uma boa utilizar o método `import` em uma *migration* após colocar o `ElasticSearch` no projeto, para que quando for para produção, já seja acionado nos modelos que são necessários.

Uma pequena alteração que podemos fazer no controlador é, logo após o método `search`, utilizar o método `records` do `ElasticSearch` para que retorne uma coleção de instâncias do modelo, ao invés de retornar documentos. Vamos verificar isso no console:

```
1 Book.search('git').to_a
2 => [#<Elasticsearch::Model::Response::Result:0x00000002ef9b18
3 @result=#<Hashie::Mash _id="2" _index="books" _score=0.081366636
4 _source=#<Hashie::Mash created_at="2014-04-16T22:22:29.740-03:00" id=2
5 lock_version=18 person_id=1 published_at="2014-04-16" stock=5 text="Livro
6 prático sobre Git." title="Conhecendo Git"
7 updated_at="2014-07-05T18:04:00.817-03:00" value="15.0"> _type="book">>]
8
9 Book.search('git').records.to_a
10 Book Load (0.3ms)  SELECT "books".* FROM "books" WHERE "books"."id" IN (2)\n=>
12 [#<Book id: 2, title: "Conhecendo Git", published_at: "2014-04-16", text:
13 "Livro prático sobre Git.", value: #<BigDecimal:2e5c1d8,'0.15E2',9(27)>,
14 person_id: 1, created_at: "2014-04-17 01:22:29", updated_at: "2014-07-05
15 21:04:00", stock: 5, lock_version: 18>]
```

Então vamos trocar o código no controlador pub de

```
1 def search
2   @results = Book.search(params[:term])
3 end
```

para

```
1 def search
2   @results = Book.search(params[:term]).records
3 end
```

Selecionando os atributos para indexação

Utilizando o Elasticsearch da forma simples que utilizamos até aqui, **todos** os atributos do modelo são indexados. No meu exemplo eu tenho cadastradas as seguintes descrições dos livros:

Conhecendo Ruby

Livro sobre a linguagem de programação Ruby.

O livro aborda temas como:

1. Instalação
2. Básico de linguagem
3. Tópicos mais avançados

Para conhecer o framework Rails, conheça o outro livro, “Conhecendo Rails”. Legal!

Conhecendo Git

Livro prático sobre Git.

Se no campo de busca no navegador eu digitar as seguintes palavras da esquerda, vou obter os resultados da direita:

Digitado	Resultado
git	Conhecendo Git
ruby	Conhecendo Ruby
framework	Conhecendo Ruby
prático	Conhecendo Git
sobre	Conhecendo Git, Conhecendo Ruby

Isso pode ajudar ou atrapalhar, dependendo do caso, pois se os outros atributos não forem úteis na consulta, ficam redundantes, ocupando espaço e processamento de forma inútil. Para evitarmos isso e especificarmos que desejamos que seja indexado e feita a consulta apenas no atributo `title`, podemos definir no modelo o método `as_indexed_json` para retornar apenas o que queremos indexar:

```
1 require 'elasticsearch/model'  
2  
3 class Book < ActiveRecord::Base  
4   ...  
5   def as_indexed_json(options = {})  
6     as_json(only: :title)  
7   end  
8   ...
```

Executando o `import` novamente e efetuando as consultas, limitando o retorno para apenas o número de livros que foram encontrados, vamos ter:

```
1 > Book.import(force: true)  
2   Book Load (0.5ms)  SELECT  "books".* FROM "books"    ORDER BY "books"."id" A\\  
3 SC LIMIT 1000  
4 => 0  
5  
6 > Book.search("conhecendo").size  
7 => 2  
8  
9 > Book.search("ruby").size  
10  
11 => 1  
12  
13 > Book.search("prático").size  
14 => 0
```

Como podemos ver, agora somente o atributo `title` foi utilizado para indexação e busca, sendo que “prático”, que fica no atributo `text`, não foi mais retornado.



Dica

Reparem que utilizamos o método `as_json` e não o `to_json`. Existe uma diferença entre eles:

```
1 > Book.first.as_json(only: :title)
2     Book Load (0.4ms)  SELECT "books".* FROM "books" ORDER BY "books"."id" \
3 " ASC LIMIT 1
4 => {"title"=>"Conhecendo Ruby"}
5 > Book.first.to_json(only: :title)
6     Book Load (0.5ms)  SELECT "books".* FROM "books" ORDER BY "books"."id" \
7 " ASC LIMIT 1
8 => "{\"title\":\"Conhecendo Ruby\"}"
```

Como podemos ver, o `to_json` retorna uma `String`, que não serve ali para o `ElasticSearch`, que usa a Hash retornada pelo `as_json`.

Um problema que vamos ter após alterar algum índice é que ele já foi criado e está armazenando as propriedades anteriores. Vamos verificar o estado do índice utilizando no terminal:

```
1 $ curl 'http://localhost:9200/books/_mapping?pretty=true'
2 {
3   "books" : {
4     "mappings" : {
5       "book" : {
6         "properties" : {
7           "created_at" : {
8             "type" : "date",
9             "format" : "strict_date_optional_time||epoch_millis"
10          },
11           "id" : {
12             "type" : "long"
13           },
14           "lock_version" : {
15             "type" : "long"
16           },
17           "person_id" : {
18             "type" : "long"
19           },
20           "published_at" : {
21             "type" : "date",
22             "format" : "strict_date_optional_time||epoch_millis"
23           },
24         }
25       }
26     }
27   }
28 }
```

```
24     "stock" : {
25         "type" : "long"
26     },
27     "text" : {
28         "type" : "string"
29     },
30     "title" : {
31         "type" : "string",
32         "analyzer" : "ngram_analyzer"
33     },
34     "updated_at" : {
35         "type" : "date",
36         "format" : "strict_date_optional_time||epoch_millis"
37     },
38     "value" : {
39         "type" : "string"
40     }
41   }
42 }
43 }
44 }
45 }
```

Podemos ver que estão todos os atributos no índice, que temos que reconstruir utilizando apenas o atributo `title`, como indicado no método `as_indexed_json`. Para isso, vamos apagar o índice utilizando no terminal do Rails:

```
1 > Book.__elasticsearch__.client.indices.delete index: Book.index_name rescue \
2 nil
3 => {"acknowledged"=>true}
```

Consultando novamente no terminal, constatamos que o índice não existe mais:

```
1 $ curl 'http://localhost:9200/books/_mapping?pretty=true'
2 {
3   "error" : {
4     "root_cause" : [ {
5       "type" : "index_not_found_exception",
6       "reason" : "no such index",
7       "resource.type" : "index_or_alias",
8       "resource.id" : "books",
9       "index" : "books"
10    } ],
11     "type" : "index_not_found_exception",
12     "reason" : "no such index",
```

```
13     "resource.type" : "index_or_alias",
14     "resource.id" : "books",
15     "index" : "books"
16   },
17   "status" : 404
18 }
```

Agora temos que recriar o índice, o que pode ser feito com o método `import`, mas vamos forçar a criação do índice antes, utilizando `create_index` com `force: true`:

```
1 > Book.__elasticsearch__.create_index! force: true
2 => {"acknowledged"=>true}
3
4 > Book.import(force: true)
5   Book Load (0.5ms)  SELECT  "books".* FROM "books"    ORDER BY "books"."id" A\SC LIMIT 1000
6 => 0
```

Verificando novamente no terminal:

```
1 $ curl 'http://localhost:9200/books/_mapping?pretty=true'
2 {
3   "books" : {
4     "mappings" : {
5       "book" : {
6         "properties" : {
7           "title" : {
8             "type" : "string"
9           }
10          }
11        }
12      }
13    }
14 }
```

Agora o índice foi criado utilizando somente o atributo `title`.

Para mais opções de customizações da busca, podemos verificar a documentação da gem, na *library* `ElasticSearch::Model`^{226 227}.

²²⁶ <https://github.com/elasticsearch/elasticsearch-rails/tree/master/elasticsearch-model>

²²⁷ <https://github.com/elasticsearch/elasticsearch-rails/tree/master/elasticsearch-model>

Pesquisando substrings

Quando eu mencionei as consultas utilizando a cláusula `LIKE`, disse que poderíamos utilizar o Elasticsearch para fazer as consultas de maneira mais inteligente, sem o *overhead* e as particularidades que o `LIKE` traz. Como vimos acima, a solução aqui apresentada funciona bem, mas se quisermos pesquisar parcialmente, como por exemplo, digitando “conhec” na consulta?

Do jeito que está, não vai ser encontrado nada, pois a consulta está configurada para **palavras inteiras**.

Para fazer a consulta por partes de palavras, como utilizando a cláusula `LIKE` no banco de dados, o Elasticsearch nos dá algumas opções como consultas do estilo `prefix`, `wildcard` e `regexp`, mas podemos nos deparar com problemas similares de performance quando utilizando partes que se situam no **meio** dos termos, utilizando um *wildcard* no começo da consulta. Para resolver esse problema, vamos utilizar **n-grams**²²⁸.

Os n-grams são sequências contínuas de caracteres em uma determinada sequência de texto, agrupados por tamanho mínimo e tamanho máximo, que gera várias sequências de caracteres, como por exemplo, com a palavra ruby:

```
1 [r] [u] [b] [y]
2 [ru] [ub] [by]
3 [rub] [uby]
4 [ruby]
```

Os n-grams que tem tamanho 1 (sendo que tamanho pode não necessariamente ser um caracter, podendo ser uma unidade, como por exemplo, uma palavra) são chamados de *unigrams*, os de tamanho 2 são *bigrams*, os de tamamho 3 são *trigrams* e os com tamanhos maiores são chamados de acordo com o número em Inglês, como por exemplo, “four-grams”, “five-grams” e por ai vai.

Para utilizarmos os n-grams na nossa aplicação com Elasticsearch, temos que indicar algumas configurações no modelo:

```
1 require 'elasticsearch/model'
2
3 class Book < ActiveRecord::Base
4   include Elasticsearch::Model
5   include Elasticsearch::Model::Callbacks
6   ...
7   settings analysis: {
8     filter: {
9       ngram_filter: {
10         type: "nGram",
11         min_gram: 3,
12         max_gram: 20
13     }
14 }
```

²²⁸<http://en.wikipedia.org/wiki/N-gram>

```
14     },
15     analyzer: {
16       ngram_analyzer: {
17         tokenizer: "lowercase",
18         filter: ["ngram_filter"],
19         type: "custom"
20       }
21     }
22   } do
23   mapping do
24     indexes :title, type: 'string', analyzer: 'ngram_analyzer'
25   end
26 end
27 ...
```

Convém notar que como tamanho mínimo, foi utilizado 3, que é o mesmo tamanho mínimo utilizado no *plugin* de *autocomplete*. Após configurar o modelo, vamos apagar o índice no *console* do Rails e criá-lo novamente (não esquecendo de recarregar com `reload!` caso ele já esteja aberto):

```
1 reload!
2 Reloading ...
3 => true
4
5 > Book.__elasticsearch__.client.indices.delete index: Book.index_name rescue \
6 nil
7 => {"acknowledged"=>true}
8
9 > Book.__elasticsearch__.create_index! force: true
10 => {"acknowledged"=>true}
11
12 > Book.import(force: true)
13 Book Load (0.2ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" A\
14 SC LIMIT 1000
15 => 0
```

Dando uma olhada novamente no terminal:

```
1 $ curl 'http://localhost:9200/books/_mapping?pretty=true'
2 {
3     "books" : {
4         "mappings" : {
5             "book" : {
6                 "properties" : {
7                     "title" : {
8                         "type" : "string",
9                         "analyzer" : "ngram_analyzer"
10                }
11            }
12        }
13    }
14 }
15 }
```

Aparentemente está ok, o analisador customizado foi ativado! Vamos testar com algumas buscas:

```
1 > Book.search("conhecendo").size
2 => 2
3
4 > Book.search("conhec").size
5 => 0
6
7 > Book.search("endo").size
8 => 0
```

Epa! Mas foi retornado 0 nas duas últimas consultas? O que fizemos não funcionou?

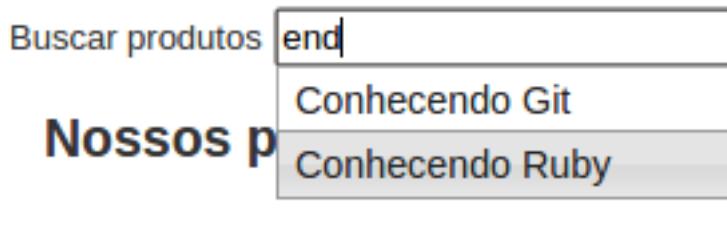
O detalhe aqui é que utilizando analisadores customizados temos que ser mais específicos nas nossas consultas para o Elasticsearch. Se as fizermos dessa maneira, especificando o atributo a ser consultado, tudo funcionará de acordo:

```
1 > Book.search("title: conhecendo").size
2 => 2
3
4 > Book.search("title: conhec").size
5 => 2
6
7 > Book.search("title: endo").size
8 => 2
```

Agora temos que alterar o código no controlador para fazer esse tipo de consulta:

```
1 def search
2   @results = Book.search("title: #{params[:term]}").records
3 end
```

E temos tudo funcionando como um LIKE, mas melhor:



Consultas com n-grams

Restringindo o acesso

É uma boa idéia, para o ambiente de produção, evitar que o serviço do Elasticsearch seja acessado de fora do servidor corrente. Para isso, vamos alterar algumas configurações no arquivo `/etc/elasticsearch/elasticsearch.yml`, inserindo as seguintes linhas, sendo que a primeira já foi configurada anteriormente:

```
1 network.bind_host: localhost
2 script.disable_dynamic: true
```

Após alteradas as configurações, devemos restartar o serviço:

```
1 $ sudo service elasticsearch restart
2 * Stopping Elasticsearch Server [ OK ]
3 * Starting Elasticsearch Server [ OK ]
```

Tudo deve estar funcional como antes, porém, agora só com acesso no `localhost`.

Apenas API

Temos um modo no Rails de criarmos uma aplicação para ser utilizada apenas em “modo API”, ou seja, só servindo funcionalidades para outras aplicações, sem precisar ter as próprias *views*, toda a parte dos *assets*, etc. Vamos criar uma pequena aplicação desse modo utilizando o parâmetro `--api`:

```
1 $ rails new apionly --api
```

Se dermos uma olhada no `Gemfile` criado, podemos notar algumas coisas interessantes:

```
1 # Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
2 # gem 'jbuilder', '~> 2.5'
3
4 # Use Rack CORS for handling Cross-Origin Resource Sharing (CORS), making cro\
5 ss-origin AJAX possible
6 # gem 'rack-cors'
```

Temos, comentados, as *gems* do `jbuilder` e `rack-cors`, que vimos no nosso capítulo sobre APIs. Em `app/controllers/application_controller.rb`, temos uma outra alteração:

```
1 class ApplicationController < ActionController::API
2 end
```

O controlador herda de `ActionController::API`, ao invés do tradicional `ActionController::Base`, fazendo com que vários recursos orientados especificamente para navegadores sejam removidos. Também temos uma alteração discreta mas bem importante em `config/application.rb`, especificando que a aplicação está em modo API:

```
1 # Only loads a smaller set of middleware suitable for API only apps.
2 # Middleware like session, flash, cookies can be added back manually.
3 # Skip views, helpers and assets when generating a new resource.
4 config.api_only = true
```

Agora vamos criar um modelo básico de `Book` e preencher ele com apenas dois registros, para fazer um teste rápido da API:

```

1 $ rails g model Book title:string value:decimal
2   invoke  active_record
3   create    db/migrate/create_books.rb
4   create    app/models/book.rb
5   invoke  test_unit
6   create    test/models/book_test.rb
7   create    test/fixtures/books.yml
8
9 $ rails db:migrate
10 == CreateBooks: migrating =====
11 -- create_table(:books)
12 -> 0.0013s
13 == CreateBooks: migrated (0.0014s) =====
14
15 > Book.create(title: 'Conhecendo Ruby', value: 10.00)
16 > Book.create(title: 'Conhecendo Git', value: 20.00)

```

Agora um controlador para fazer a busca por livros:

```

1 $ rails g controller books search
2   create  app/controllers/books_controller.rb
3   route   get 'books/search'
4   invoke  test_unit
5   create    test/controllers/books_controller_test.rb

```

Inserindo esse código (ineficiente, como vimos anteriormente, mas é só para demonstração) lá:

```

1 class BooksController < ApplicationController
2   def search
3     render json: Book.where('title like ?', "%#{params[:term]}%")
4   end
5 end

```

Agora precisamos decidir como renderizar isso para retorno. Ali já estamos especificando que vai ser retornado um JSON com *todos* os atributos do modelo:

```

1 [{"id":1, "title":"Conhecendo Ruby", "value":"10.0",
2  "created_at":"2018-01-30T11:20:04.613Z", "updated_at":"2018-01-30T11:20:04.6\"
3  13Z"}
4 }]

```

Mas e se quisermos apenas alguns atributos, atributos processados ou até mais alguma informação? Para isso vamos dar uma olhada nos *serializers*.

Serializers

Jbuilder

Um *serializer* que já conhecemos é o `jbuilder`, que inclusive está comentado no `Gemfile` e já utilizamos anteriormente. O bom do `jbuilder` é que ele praticamente dá controle total do retorno e é bem simples. Vamos remover o comentário do `Gemfile` e rodar o `bundler`:

```

1 gem 'jbuilder', '~> 2.5'
2
3 ...
4
5 $ bundle
6 ...
7 Using jbuilder 2.6.4
8 ...

```

Agora vamos fazer uma *view* praticamente igual a que utilizamos anteriormente, em `app/views/books/search.json.jbuilder`:

```

1 json.array!(@books) do |book|
2   json.id book.id
3   json.value book.title
4 end

```

E alterar o controlador para retornar a variável `@books`:

```

1 class BooksController < ApplicationController
2   def search
3     @books = Book.where('title like ?', "%#{params[:term]}%")
4   end
5 end

```

Agora temos retornados somente os atributos `id` e `title`:

```

1 $ curl "http://localhost:3000/books/search?term=Ruby"
2 [{"id":1,"value":"Conhecendo Ruby"}]

```

ActiveModel::Serializers

Outra opção é o `ActiveModel::Serializers`²²⁹. Apesar de uma opção prática, está havendo um burburinho²³⁰ sobre o desenvolvimento dessa *gem*, com os autores inclusive listando algumas alternativas (vamos ver uma logo em seguida). Mas vamos ver como funciona essa *gem* primeiro, utilizando para fazer a busca da mesma maneira que feita anteriormente, só com os atributos `id` e `title`.

Vamos inserir a *gem* no `Gemfile` e rodar o `bundler`:

²²⁹ https://github.com/rails-api/active_model_serializers

²³⁰ https://github.com/rails-api/active_model_serializers#status-of-ams

```
1 gem 'active_model_serializers'  
2  
3 ...  
4  
5 $ bundle  
6 ...  
7 Fetching active_model_serializers 0.10.7  
8 Installing active_model_serializers 0.10.7  
9 ...
```

Agora vamos criar um *serializer* do modelo Book, que vai especificar como ele deve ser transformado em JSON e vai gerar o arquivo app/serializers/book_serializer.rb:

```
1 $ rails g serializer book  
2       create  app/serializers/book_serializer.rb
```

Vamos preencher o arquivo com o seguinte conteúdo:

```
1 class BookSerializer < ActiveModel::Serializer  
2   attributes :id, :title  
3 end
```

E alterar o controlador novamente para:

```
1 class BooksController < ApplicationController  
2   def search  
3     render json: Book.where('title like ?', "%#{params[:term]}%")  
4   end  
5 end
```

Verificando:

```
1 $ curl "http://localhost:3000/books/search?term=Ruby"  
2 [{"id":1,"title":"Conhecendo Ruby"}]
```

JSONAPI-RB

A última alternativa a ser vista aqui (existem também outras) é a **JSONAPI-RB**²³¹, que é inclusive listada como alternativa à ActiveModel::Serializers acima. Vamos inclui-la no Gemfile, através da gem jsonapi-rails (específica, lógico, para o Rails) e rodar o bundler:

²³¹<http://jsonapi-rb.org/>

```
1 gem 'jsonapi-rails'  
2  
3 ...  
4  
5 $ bundle  
6 ...  
7 Fetching jsonapi-rails 0.3.1  
8 Installing jsonapi-rails 0.3.1  
9 ...
```

Agora vamos criar o arquivo `app/serializers/serializable_book.rb` com:

```
1 class SerializableBook < JSONAPI::Serializable::Resource  
2   type 'books'  
3   attributes :title  
4 end
```

E alterar o controlador para:

```
1 class BooksController < ApplicationController  
2   def search  
3     render jsonapi: Book.where('title like ?', "%#{params[:term]}%")  
4   end  
5 end
```

Verificando:

```
1 $ curl "http://localhost:3000/books/search?term=Ruby"  
2 { "data":  
3   [  
4     {"id": "1", "type": "books", "attributes": {"title": "Conhecendo Ruby"} }  
5   ], "jsonapi": {"version": "1.0"}  
6 }
```

Vejam que o retorno segue o formato da [JSON-API](#)²³², sendo que essa *gem* é bem flexível para customizar os atributos e retornar também associações. Como demonstração, vamos inserir o valor e criar um *slug* no retorno:

²³²<http://jsonapi.org/>

```
1 class SerializableBook < JSONAPI::Serializable::Resource
2   type 'books'
3   attributes :title, :value
4
5   attribute :slug do
6     @object.title.parameterize
7   end
8 end
```

Verificando:

```
1 $ curl "http://localhost:3000/books/search?term=Ruby"
2 { "data": [
3   [
4     {"id": "1", "type": "books", "attributes": {"title": "Conhecendo Ruby", "value": "10.0", "slug": "conhecendo-ruby"}}
5   ], "jsonapi": {"version": "1.0"}}
6 }
7 }
```

Gerando um diagrama do banco de dados

Até agora fizemos a estrutura do banco de dados diretamente no código, sem (ou com muito pouco) planejamento das estruturas em uma ferramenta (que pode ser perfeitamente uma descrição, uma lousa, papel e lápis, etc., não necessariamente um *software*) ERD²³³, que significa *entity relationship model*, ou *modelo entidade-relacionamento*. Um diagrama desse tipo é muito bom para visualizarmos com o nosso banco de dados está estruturado, nos fornecendo uma visão gráfica e clara de como estão nossos modelos.

Geralmente isso é feito antes do início do projeto, e com o modelo MVC e convenções adotados pelo Rails, na minha opinião deixa até mais rápido de fazer e com uma visão mais limpa. Eu costumo fazer esse diagrama primeiro em uma lousa, para depois fazer uma revisão mais detalhada em outra ferramenta (geralmente papel e lápis, rabiscar é barato), depois redesenhando a estrutura somente das partes que vamos implementar imediatamente novamente na lousa, fazendo um revezamento para dar até mais foco em cada parte específica.

Em cada uma dessas fases, eu costumo tirar fotos, armazenando em alguma ferramenta como o Evernote²³⁴ ou o Pivotal Tracker²³⁵, de onde podemos recuperar mais tarde para conferência de cada parte ou para o desenho do diagrama total.

De qualquer maneira, tendo o desenho total do diagrama ou não, podemos utilizar algumas ferramentas para extrair esse diagrama do código em nossos modelos da aplicação e gerar uma imagem para que tenhamos uma idéia de como estão, para efeito de comparação com a idéia original da implementação ou mesmo para que vejamos como estão os relacionamentos se não tivermos nada desenhado e só tenhamos eles na cabeça. Para isso, vamos utilizar a gem rails-erd²³⁶.

Para instalar, do modo costumeiro, instando no grupo development (não vamos precisar dela em produção) e rodando o bundler. Mas antes precisamos instalar no sistema operacional o pacote graphviz, e levando em conta que estamos no Ubuntu, podemos fazer da seguinte maneira:

```
1 $ sudo apt install graphviz
```

E agora instalar a gem:

²³³ http://pt.wikipedia.org/wiki/Modelo_entidade_relacionamento

²³⁴ <https://www.evernote.com>

²³⁵ <http://www.pivotaltracker.com/>

²³⁶ <https://github.com/voormedia/rails-erd>

```

1 group :development do
2   gem 'rails-erd'
3 ...
4 $ bundle
5 ...
6 Installing ruby-graphviz 1.2.3
7 Installing rails-erd 1.5.0
8 ...

```

Agora é só utilizar o comando erd para gerar o diagrama:

```

1 $ erd
2 Loading application in 'bookstore'...
3 Generating entity-relationship diagram for 9 models...
4 Diagram saved to 'erd.pdf'.

```

Que nos gera um arquivo chamado erd.pdf, como esse:

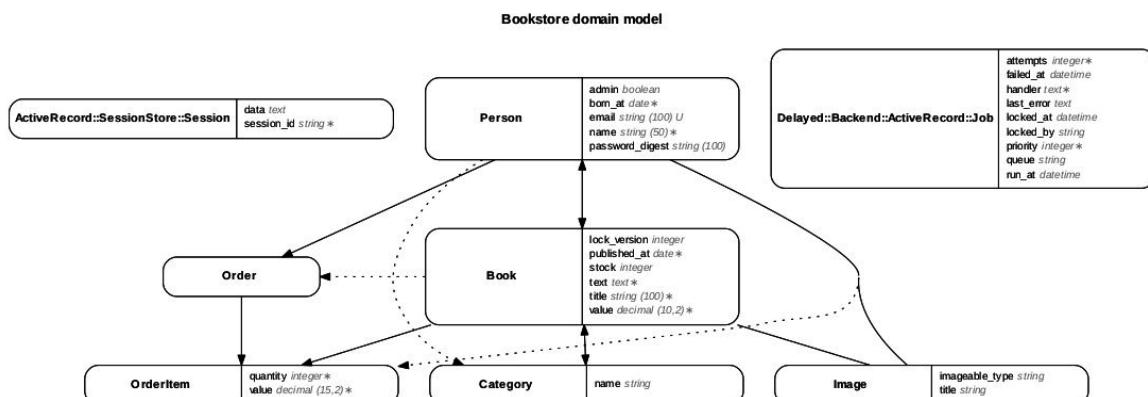


Diagrama do banco de dados

O comando erd nos dá algumas opções:

```

1 Usage: erd [-v]
2
3 Diagram options:
4     --title=TITLE          Replace default diagram title with a cus\
5 tom one.
6     --notation=STYLE        Diagram notation style, one of simple, b\
7 achman, uml or crowsfoot.
8     --attributes=TYPE,...   Attribute groups to display: false, cont\
9 ent, primary_keys, foreign_keys, timestamps and/or inheritance.
10    --orientation=ORIENTATION    Orientation of diagram, either horizonta\
11 l (default) or vertical.
12    --inheritance          Display (single table) inheritance relat\

```

```

13  ionships.
14      --polymorphism          Display polymorphic and abstract entitie\
15  s.
16      --direct                Omit indirect relationships (through oth\
17 er entities).
18      --connected             Omit entities without relationships.
19      --only                  Filter to only include listed models in \
20 diagram.
21      --only_recursion_depth=INTEGER
22                                Recurses into relations specified by --o\
23 nly upto a depth N.
24      --exclude               Filter to exclude listed models in diagr\
25 am.
26      --sort=BOOLEAN          Sort attribute list alphabetically
27      --prepend_primary=BOOLEAN
28 ute list
29      --cluster               Display models in subgraphs based on the\
30 ir namespace.
31
32 Output options:
33      --filename=FILENAME      Basename of the output diagram.
34      --filetype=TYPE          Output file type. Available types depend\
35 on the diagram renderer.
36      --no-markup             Disable markup for enhanced compatibilit\
37 y of .dot output with other applications.
38      --open                  Open the output file after it has been s\
39 aved.
40
41 Common options:
42      --help                  Display this help message.
43      --debug                 Show stack traces when an error occurs.
44      -v, --version            Show version and quit.

```

Vamos testar algumas:

```

1 $ erd --title='ERD da livraria' --inheritance --polymorphism --connected \
2 --filename=bookstore-erd --filetype=png --notation=bachman \
3
4
5 Loading application in 'bookstore'...
6 Generating entity-relationship diagram for 9 models...
7 Diagram saved to 'bookstore-erd.png'.

```

E agora temos esse resultado:

ERD da livraria

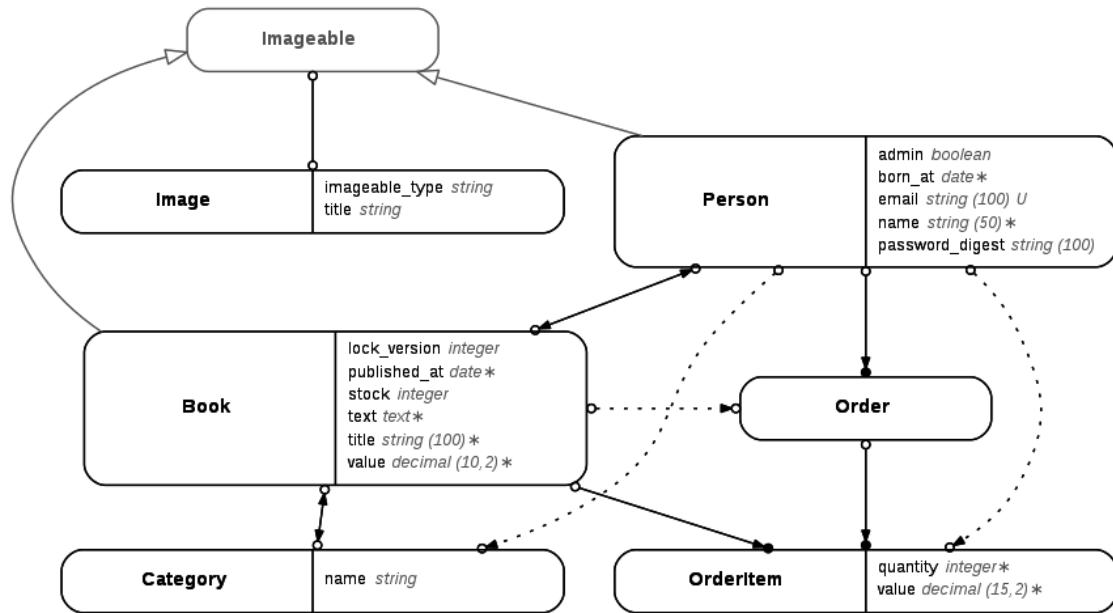


Diagrama do banco de dados

Fim!

Ok, terminamos por aqui. Espero que você tenha curtido o livro e tenha valido a pena ter adquirido (ei, obrigado!) e mais do que isso que tenha aprendido como programar utilizando o *framework* para fazer a aplicação básica voltada à didática exposta aqui, desde as suas opções mais básicas até utilizando alguns dos “extras” apontados ali atrás.

Também espero que se você tenha ficado um pouco bravo quando viu que algumas das coisas que fizemos poderia ter sido feita de um jeito mais fácil (como as senhas criptografadas, o *upload* de arquivos, etc) que também tenha percebido que juntamente com o jeito “mais difícil” eu tentei demonstrar alguns tipos de técnicas que podem ser úteis quando não for desejado por alguma razão utilizar alguma gem já pronta com a funcionalidade (de novo: não entupam os projetos de gems a torto e direito, se não tiver necessidade) ou se quiser fazer alguma coisa diferente do que já tem pronto.

Agora é continuar a estudar. A comunidade Ruby e Rails (são duas, sim, mas que acabam gerando uma terceira) são vibrantes e tem muita, mas muita coisa interessante para estudar de modo que possa complementar o que foi aprendido até agora. Eu também vou continuar a atualizar o livro, e você, como um cara legal que me ajudou a pagar o leitinho da criança aqui de casa, vai continuar a receber as atualizações. Novamente, obrigado.