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

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

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 Bluefish [1], 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 Rails [2]. 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 conhecerem 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://eustaquiorangel.com[4], meu Twitter é @taq[5] e meu Github é http://github.com/taq[6].

Convenções utilizadas

Durante o livro temos algumas convenções:

- Texto em *itálico* 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 (. . .), signifjca 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://eustaquiorangel.com
[5] http://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

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*:

```
$ rails g scaffold Book title:string published_at:date text:text value:decima\
l person:references
      invoke  active_record
      create    db/migrate/20170311144612_create_books.rb
      create    app/models/book.rb
      invoke    test_unit
      create      test/models/book_test.rb
      create      test/fixtures/books.yml
      ...
```

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:

```ruby
class CreateBooks < ActiveRecord::Migration[5.0]
  def change
    create_table :books do |t|
      t.string :title, limit: 100, null: false
      t.date :published_at, null: false
      t.text :text, null: false
      t.decimal :value, precision: 10, scale: 2, null: false
      t.references :person, foreign_key: true
      t.timestamps
    end
  end
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 digitos (`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 estrangeira[7]. 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
```

Reparem 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:

[7] https://pt.wikipedia.org/wiki/Chave_estrangeira

```ruby
require 'test_helper'

class BookTest < ActiveSupport::TestCase
  setup do
    @book = books(:one)
  end

  # título
  test 'deve ter um título' do
    @book.title = nil
    assert !@book.valid?
  end

  test 'não pode ter mais que 100 caracteres' do
    @book.title = '*' * 101
    assert !@book.valid?
  end

  # data de publicação
  test 'deve ter data de publicação' do
    @book.title = nil
    assert !@book.valid?
  end

  # texto
  test 'deve ter texto' do
    @book.text = nil
    assert !@book.valid?
  end

  # valor
  test 'deve ter valor' do
    @book.value = nil
    assert !@book.valid?
  end

  test 'deve ser um número' do
    @book.value = 'bla'
    assert !@book.valid?
  end

  test 'não deve ser maior que o permitido' do
    @book.value = 100000000.00
    assert !@book.valid?
  end

```

```
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`:

```
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:

```
class Book < ApplicationRecord
  validates :title, presence: true, length: { maximum: 100 }
  validates :published_at, presence: true
  validates :text, presence: true
  validates :value, presence: true, numericality: { less_than_or_equal_to: 99\
999999.99 }
  validates :person, presence: true

  belongs_to :person
end
```

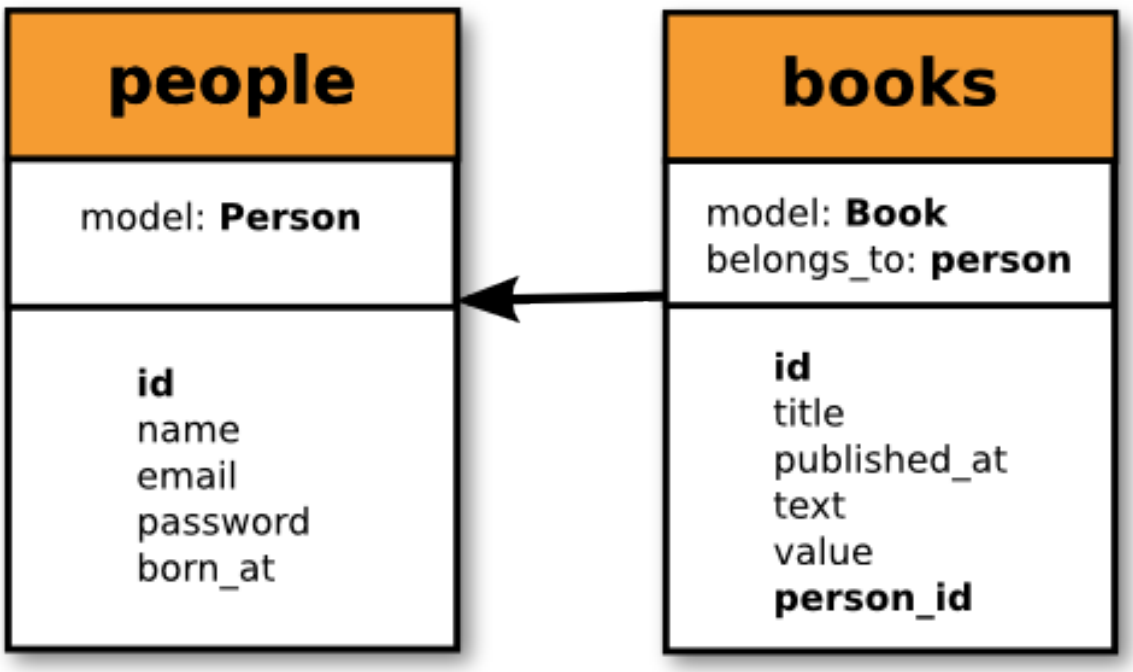
Vamos ver como funcionam essas validações logo mais no livro, mas tem algumas ali que até já são auto-explicativas. Se rodarmos nossos testes agora, todos devem passar.

Como temos a associação com pessoa, em nossos objetos de livros, ela já pode ser testada no *console* ou no navegador:

```
book = Book.new
=> #<Book id: nil, title: nil, published_at: nil, text: nil, value: nil, pers\
on_id: nil, created_at: nil, updated_at: nil>

book.title = "Ruby - Conhecendo a Linguagem"
=> "Ruby - Conhecendo a Linguagem"

book.published_at = "2006-03-01"
=> "2006-03-01"

book.text = "Tinha, mas acabou."
=> "Tinha, mas acabou."

book.value = 40.00
=> 40.0

book.person = Person.last
Person Load (0.3ms)  SELECT "people".* FROM "people" ORDER BY name DESC LIMIT\
 1
=> #<Person id: 2, name: "Eustáquio Rangel", email: "taq@bluefish.com.br",
password: "9733340c840c719779f234407ee0bac26ae8904b", born_at: "1971-04-06",
admin: true, created_at: "2013-07-06 22:51:34", updated_at: "2013-07-07
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:

```
test "deve ter uma coleção de livros" do
  person = people(:admin)
  assert_respond_to person, :books
  assert_kind_of Book, person.books.first
end
```

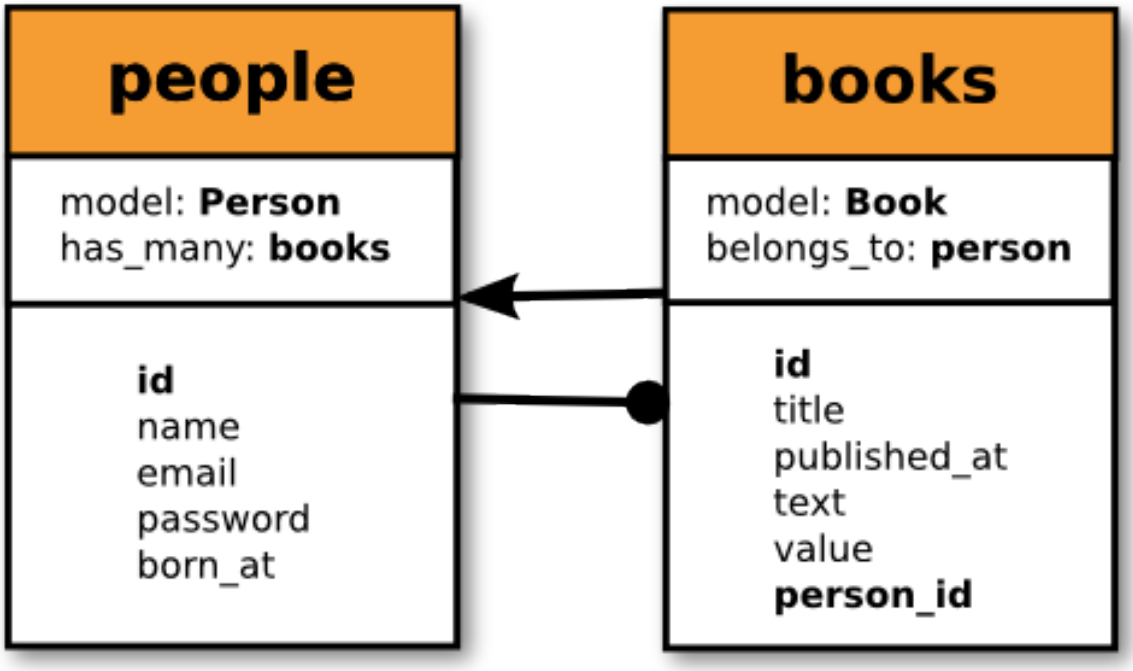
E no modelo, estabelecendo a associação, usando `has_many`:

```
    has_many :books
```

Através do `has_many` temos uma coleção de livros no objeto da pessoa, como podemos verificar pelo *console*:

```
p = Person.last
Person Load (0.3ms)  SELECT "people".* FROM "people" ORDER BY name DESC LIMIT\
 1
=> #<Person id: 2, name: "Eustáquio Rangel", email: "taq@bluefish.com.br",
password: "9733340c840c719779f234407ee0bac26ae8904b", born_at: "1971-04-06",
admin: true, created_at: "2013-07-06 22:51:34", updated_at: "2013-07-07
22:48:20">

p.books
Book Load (0.1ms)  SELECT "books".* FROM "books" WHERE "books"."person_id" = 2
=> [#<Book id: 1, title: "Ruby - Conhecendo a Linguagem", published_at:
"2006-03-01", text: "Tinha, mas acabou.", value:
#<BigDecimal:aaae9c0,'0.4E2',9(36)>, person_id: 2, created_at: "2013-07-09
15:15:46", updated_at: "2013-07-09 15:15:46">]

p.book_ids
=> [1]

p.books.last.title
=> "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 funciona[8] 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:

[8] https://github.com/rails/rails/issues/3458

```
test "deve apagar os livros quando apagar a pessoa" do
  person = people(:admin)
  assert person.books.size > 0

  assert_difference('Person.count', -1) do
    assert_difference('Book.count',person.books.size * -1) do
      assert person.destroy, "deveria apagar a pessoa"
    end
  end
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` anter 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`:

```
$ rails g scaffold Category name:string
      invoke  active_record
      create    db/migrate/20170312135948_create_categories.rb
      create    app/models/category.rb
      invoke    test_unit
$ 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:

```
tech:
  name: Tecnologia

dev:
  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`:

```
class BooksController < ApplicationController
  before_action :set_book, only: [:show, :edit, :update, :destroy]
  before_action :load_categories, only: [:new, :edit, :create, :update]
  ...

  private
  def load_categories
     @categories = Category.all
  end
  ...
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*:

```
$ rails g migration CreateJoinTableBookCategory book category
      invoke  active_record
      create    db/migrate/20170312145600_create_join_table_book_category.rb
```

Ela vai ter um conteúdo como esse, já removidos os comentários:

```
class CreateJoinTableBookCategory < ActiveRecord::Migration[5.0]
  def change
    create_join_table :books, :categories do |t|
      t.index [:book_id, :category_id]
      t.index [:category_id, :book_id]
    end
  end
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:

```
$ rails db:migrate
== 20170312145600 CreateJoinTableBookCategory: migrating ====================\
==
-- create_join_table(:books, :categories)
   -> 0.0076s
   == 20170312145600 CreateJoinTableBookCategory: migrated (0.0076s)
   =============
```

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:

```
# categorias
test 'deve ter categorias' do
  assert_respond_to @book, :categories
end

test 'deve ter categorias do tipo correto' do
  @book.categories << categories(:one)
  assert_kind_of Category, @book.categories.first
end
```

E agora sim podemos alterar o modelo de livro indicando que ele pertence à várias categorias:

```
class Book < ApplicationRecord
  ...
  has_and_belongs_to_many :categories
  ...
end
```

Isso nos dá os seguintes métodos em `Book`:

```
> b = Book.first
  Book Load (0.1ms)  SELECT  "books".* FROM "books" ORDER BY "books"."id" ASC
  LIMIT ?  [["LIMIT", 1]] => #<Book id: 14, title: "Conhecendo Ruby",
  published_at: "2013-06-29", text: "Livro prático sobre a linguagem Ruby",
  value: #<BigDecimal:1701a10,'0.1E1',9(18)>, person_id: 15, created_at:
  "2017-03-12 15:22:07", updated_at: "2017-03-12 15:22:07">

> b.categories
  Category Load (0.2ms)  SELECT "categories".* FROM "categories" INNER JOIN
  "books_categories" ON "categories"."id" = "books_categories"."category_id"
  WHERE "books_categories"."book_id" = ?  [["book_id", 14]] =>
  #<ActiveRecord::Associations::CollectionProxy [#<Category id: 1, name:
  "Tecnologia", created_at: "2017-03-12 14:02:51", updated_at: "2017-03-12
  14:02:51">, #<Category id: 2, name: "Programação", created_at: "2017-03-12
  14:02:51", updated_at: "2017-03-12 14:02:51">]>

> b.category_ids
 => [1, 2]
```

E agora podemos fazer a mesma coisa no modelo de categoria:

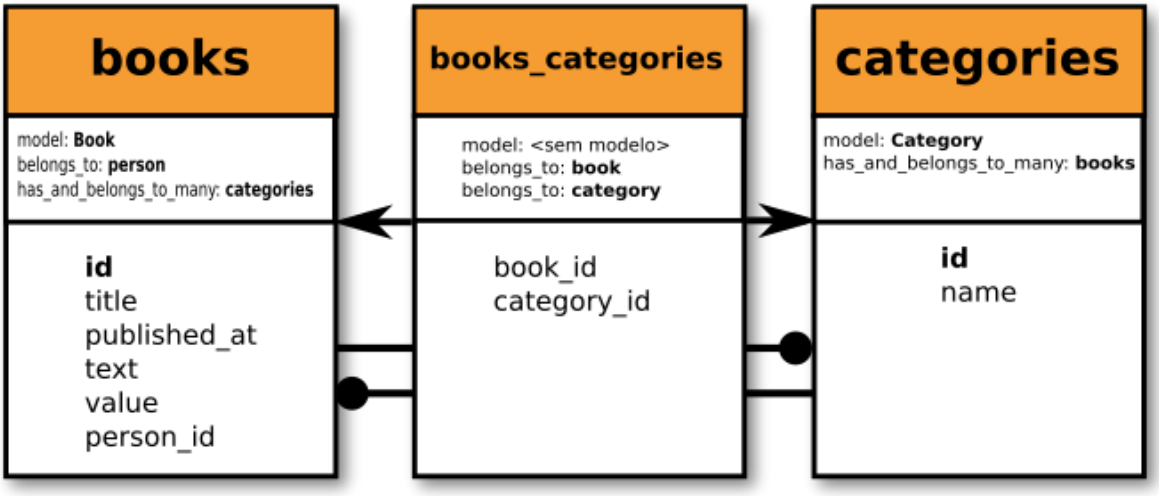
```
class Category < ApplicationRecord
  has_and_belongs_to_many :books
end
```

Que nos dá os métodos:

```
> c = Category.first
Category Load (0.4ms)  SELECT  "categories".* FROM "categories" ORDER BY
"categories"."id" ASC LIMIT ?  [["LIMIT", 1]]
=> #<Category id: 1, name: "Tecnologia", created_at: "2017-03-12 14:02:51",
updated_at: "2017-03-12 14:02:51">

> c.books
  Book Load (0.3ms)  SELECT "books".* FROM "books" INNER JOIN "books_categori\
es"
  ON "books"."id" = "books_categories"."book_id" WHERE
  "books_categories"."category_id" = ?  [["category_id", 1]] =>
  #<ActiveRecord::Associations::CollectionProxy [#<Book id: 14, title:
  "Conhecendo Ruby", published_at: "2013-06-29", text: "Livro prático sobre a
  linguagem Ruby", value: #<BigDecimal:3799958,'0.1E1',9(18)>, person_id: 15,
  created_at: "2017-03-12 15:22:07", updated_at: "2017-03-12 15:22:07">, #<Bo\
ok
  id: 15, title: "Conhecendo o Git", published_at: "2013-06-24", text: "Quer
  aprender Git de forma rápida e prática?", value:
```

```
  #<BigDecimal:378fef8,'0.1E2',9(18)>, person_id: 15, created_at: "2017-03-12
  15:22:07", updated_at: "2017-03-12 15:22:07">]>

> c.book_ids
 => [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:

```
def book_params
  params.require(:book).permit(:title, :published_at, :text, :value, :person_\
id, category_ids: [])
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`:

```
  <div class="field">
    <h2>Categorias</h2>
    <%= collection_check_boxes :book, :category_ids, @categories, :id, :name \
%>
  </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:

```
<ul>
<%= collection_check_boxes :book, :category_ids, @categories, :id, :name do |\
builder| %>
  <li>
    <%= builder.label { builder.check_box + builder.text } %>
  </li>
<% end %>
</ul>
```

Agora sim ficou legal. Vamos alterar a `view app/views/books/show.html.erb` para listar as categorias cadastradas do livro:

```
<h2>Categorias</h2>
<ul>
  <% for category in @book.categories %>
    <li><%= category.name %></li>
  <% end %>
</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:

```
one:
  title: Conhecendo Ruby
  published_at: 2013-06-29
  text: Livro prático sobre a linguagem Ruby
  value: 1.00
  person: admin
  categories: [ tech, dev ]

two:
  title: Conhecendo o Git
  published_at: 2013-06-24
  text: Quer aprender Git de forma rápida e prática?
  value: 10.00
  person: admin
  categories: [ tech, dev ]
```

Agora um teste de pessoa:

```
# categorias
test 'deve ter várias categorias, através de livros' do
  assert_respond_to @person, :categories
end

test 'deve ter uma categoria do tipo correto' do
  assert_kind_of Category, @person.categories.first
end

test 'deve ter apenas duas categorias' do
  assert_equal 2, @person.categories.count
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:

```
...
has_many :categories, through: :books
...
```

Novamente, uma configuração bem simples, que nos dá as categorias da pessoa:

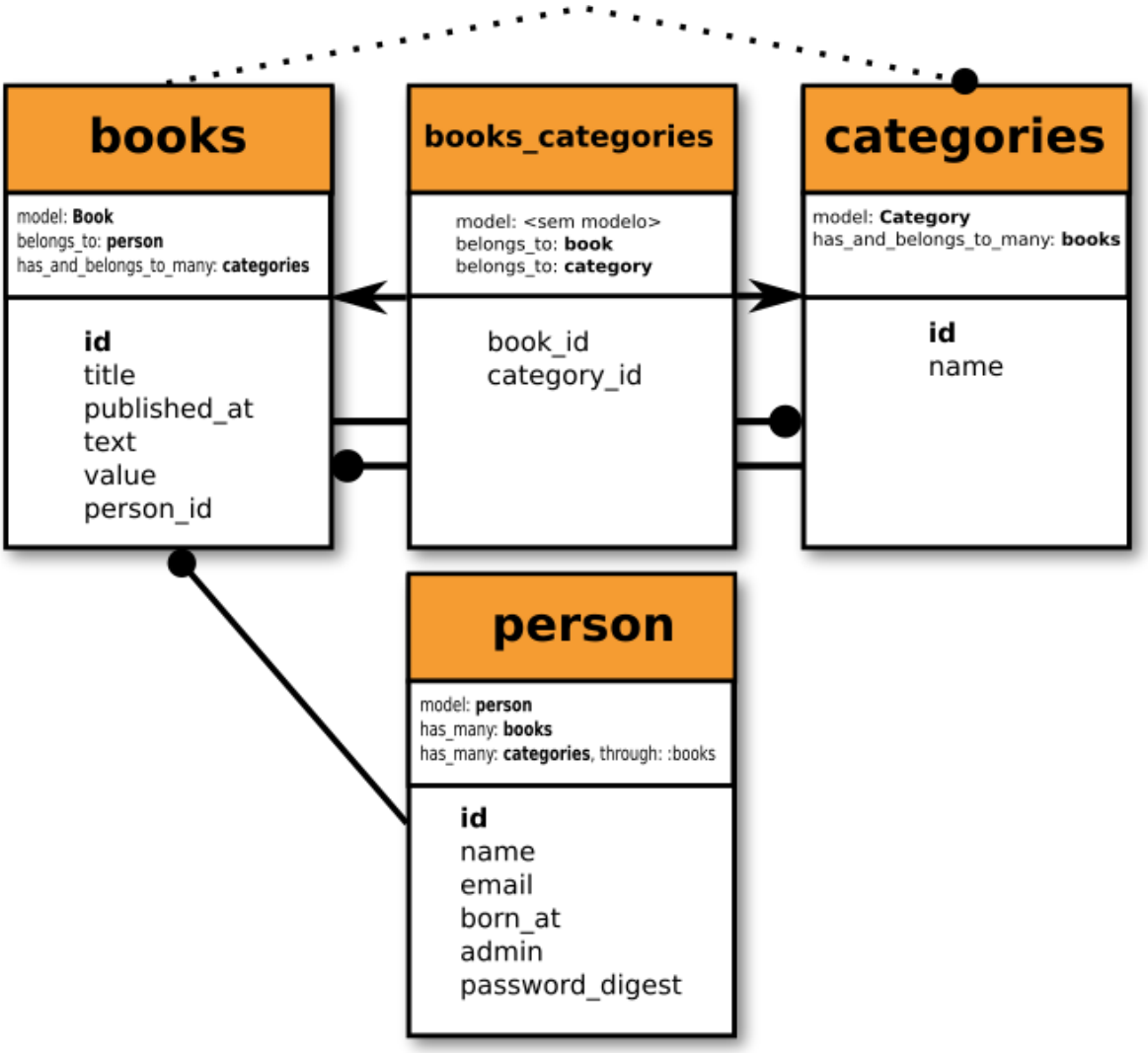
```
> Person.last.categories
  Person Load (0.4ms)  SELECT  "people".* FROM "people" ORDER BY "people"."na\
me"
  DESC LIMIT ?  [["LIMIT", 1]] Category Load (0.1ms)  SELECT "categories".* F\
ROM
  "categories" INNER JOIN "books_categories" ON "categories"."id" =
  "books_categories"."category_id" INNER JOIN "books" ON
  "books_categories"."book_id" = "books"."id" WHERE "books"."person_id" = ?
  [["person_id", 15]]
 => #<ActiveRecord::Associations::CollectionProxy [#<Category id: 1, name:
 "Tecnologia", created_at: "2017-03-12 14:02:51", updated_at: "2017-03-12
 14:02:51">, #<Category id: 2, name: "Programação", created_at: "2017-03-12
 14:02:51", updated_at: "2017-03-12 14:02:51">, #<Category id: 1, name:
 "Tecnologia", created_at: "2017-03-12 14:02:51", updated_at: "2017-03-12
 14:02:51">, #<Category id: 2, name: "Programação", created_at: "2017-03-12
 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:

```
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:

```
> Person.select { |person| person.books.count > 0 }
  Person Load (0.1ms)  SELECT "people".* FROM "people" ORDER BY "people"."nam\
e" ASC
   (0.1ms)  SELECT COUNT(*) FROM "books" WHERE "books"."person_id" = ?  [["pe\
rson_id", 4]]
   (0.1ms)  SELECT COUNT(*) FROM "books" WHERE "books"."person_id" = ?  [["pe\
rson_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`:

```
Person.joins(:books).distinct
  Person Load (0.2ms)  SELECT DISTINCT "people".* FROM "people" INNER JOIN
  "books" ON "books"."person_id" = "people"."id" ORDER BY "people"."name" ASC
 => #<ActiveRecord::Relation [#<Person id: 3, name: "Eustáquio Rangel de
 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:

```
> Person.joins(:books).distinct.map { |person| { person.name => person.books.\
pluck(:title) } }
  Person Load (0.3ms)  SELECT DISTINCT "people".* FROM "people" INNER JOIN
  "books" ON "books"."person_id" = "people"."id" ORDER BY "people"."name" ASC
   (0.1ms)  SELECT "books"."title" FROM "books" WHERE "books"."person_id" = ?\
  [["person_id", 3]]
 => [{"Eustáquio Rangel de Oliveira Jr."=>["Conhecendo Ruby", "Conhecendo o G\
it"]}]
```

Reparem 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 disponivel os dados da outra associação, não faz eager loading[9]. 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`:

```
Person.includes(:books).distinct.map { |person| { person.name => person.books\
.pluck(:title) } }
  Person Load (0.1ms)  SELECT DISTINCT "people".* FROM "people" ORDER BY "peo\
ple"."name" ASC
  Book Load (0.4ms)  SELECT "books".* FROM "books" WHERE "books"."person_id" \
IN (4, 3)
 => [{"Ana Carolina"=>[]}, {"Eustáquio Rangel de Oliveira Jr."=>["Conhecendo \
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:

[9] http://guides.rubyonrails.org/active_record_querying.html

```
> Person.joins(:books).includes(:books).distinct.map { |person| { person.name\
 => person.books.pluck(:title) } }
  SQL (0.2ms)  SELECT DISTINCT "people"."id" AS t0_r0, "people"."name" AS t0_\
r1,
  "people"."email" AS t0_r2, "people"."password_digest" AS t0_r3,
  "people"."born_at" AS t0_r4, "people"."admin" AS t0_r5, "people"."created_a\
t"
  AS t0_r6, "people"."updated_at" AS t0_r7, "books"."id" AS t1_r0,
  "books"."title" AS t1_r1, "books"."published_at" AS t1_r2, "books"."text" AS
  t1_r3, "books"."value" AS t1_r4, "books"."person_id" AS t1_r5,
  "books"."created_at" AS t1_r6, "books"."updated_at" AS t1_r7, "books"."stoc\
k"
  AS t1_r8, "books"."lock_version" AS t1_r9 FROM "people" INNER JOIN "books" \
ON
  "books"."person_id" = "people"."id" ORDER BY "people"."name" ASC
 => [{"Eustáquio Rangel de Oliveira Jr."=>["Conhecendo Ruby", "Conhecendo o G\
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:

```
> Person.pluck(:name)
   (0.3ms)  SELECT "people"."name" FROM "people" ORDER BY "people"."name" ASC
 => ["Ana Carolina", "Eustáquio Rangel de Oliveira Jr."]
```

Podemos indicar mais de um atributo:

```
> Person.pluck(:name, :email)
   (0.2ms)  SELECT "people"."name", "people"."email" FROM "people" ORDER BY "\
people"."name" ASC
 => [["Ana Carolina", "carol@bluefish.com.br"], ["Eustáquio Rangel de Oliveir\
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:

```
> Person.select(:name, :email)
  Person Load (0.1ms)  SELECT "people"."name", "people"."email" FROM "people"
  ORDER BY "people"."name" ASC => #<ActiveRecord::Relation [#<Person id: nil,
  name: "Ana Carolina", email: "carol@bluefish.com.br">, #<Person id: nil, na\
me:
  "Eustáquio Rangel de Oliveira Jr.", email: "taq@bluefish.com.br">]>

> Person.select(:name, :email).first.name
  Person Load (0.3ms)  SELECT  "people"."name", "people"."email" FROM "people"
  ORDER BY "people"."name" ASC LIMIT ?  [["LIMIT", 1]]
 => "Ana Carolina"

> Person.select(:name, :email).first.email
  Person Load (0.2ms)  SELECT  "people"."name", "people"."email" FROM "people"
  ORDER BY "people"."name" ASC LIMIT ?  [["LIMIT", 1]]
 => "carol@bluefish.com.br"
```

Já isso, não:

```
> Person.select(:name, :email).first.admin
  Person Load (0.1ms)  SELECT  "people"."name", "people"."email" FROM "people"
  ORDER BY "people"."name" ASC LIMIT ?  [["LIMIT", 1]]
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.