

CAPÍTULO 7

Active Record

"Não se deseja aquilo que não se conhece"

— Ovídio

Nesse capítulo começaremos a desenvolver um sistema utilizando Ruby on Rails com recursos mais avançados.

7.1 – MOTIVAÇÃO

Queremos criar um sistema de qualificação de restaurantes. Esse sistema terá clientes que qualificam os restaurantes visitados com uma nota, além de informar quanto dinheiro gastaram. Os clientes terão a possibilidade de deixar comentários para as qualificações feitas por eles mesmos ou a restaurantes ainda não visitados. Além disso, os restaurantes terão pratos, e cada prato a sua receita.

O site <http://www.tripadvisor.com> possui um sistema similar para viagens, onde cada cliente coloca comentários sobre hotéis e suas visitas feitas no mundo inteiro.

7.2 – EXERCÍCIOS: CONTROLE DE RESTAURANTES

1. Crie um novo projeto chamado `vota_prato`:
 - a. No terminal, garanta que não está no diretoria do projeto anterior.
 - b. Digite o comando `rails new vota_prato -d mysql`
 - c. Observe o log de criação do projeto:

```
File Edit View Terminal Tabs Help
rr71@caelum131-03:~$ rails new vota_prato -d mysql
create
create README
create Rakefile
create config.ru
create .gitignore
create Gemfile
create app
create app/views/layouts/application.html.erb
create app/controllers/application_controller.rb
create app/helpers/application_helper.rb
create app/models
create config
create config/routes.rb
create config/application.rb
create config/environment.rb
create config/environments
create config/environments/production.rb
create config/environments/test.rb
create config/environments/development.rb
create config/initializers
create config/initializers/session_store.rb
create config/initializers/backtrace_silencers.rb
create config/initializers/secret_token.rb
create config/initializers/mime_types.rb
create config/initializers/inflections.rb
create config/locales
create config/locales/en.yml
create config/boot.rb
create config/database.yml
create db
create db/seeds.rb
create doc
create doc/README_FOR_APP
create lib
create lib/tasks
create lib/tasks/.gitkeep
create log
create log/server.log
create log/production.log
create log/development.log
```

Nova editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

7.3 – MODELO – O "M" DO MVC

Models são os modelos que serão usados nos sistemas: são as entidades que serão armazenadas em um banco. No nosso sistema teremos modelos para representar um **Cliente**, um **Restaurante** e uma **Qualificação**, por exemplo.

O componente de Modelo do Rails é um conjunto de classes que usam o ActiveRecord, uma classe ORM que mapeia objetos em tabelas do banco de dados. O ActiveRecord usa convenções de nome para determinar os mapeamentos, utilizando uma série de regras que devem ser seguidas para que a configuração seja a mínima possível.

ORM

ORM (*Object-Relational Mapping*) é um conjunto de técnicas para a transformação entre os modelos orientado a objetos e relacional.

7.4 – ACTIVERECORD

É um framework que implementa o acesso ao banco de dados de forma transparente ao usuário, funcionando como um Wrapper para seu modelo. Utilizando o conceito de *Conventions over Configuration*, o ActiveRecord adiciona aos seus modelos as funções necessárias para acessar o banco.

`ActiveRecord::Base` é a classe que você deve estender para associar seu modelo com a tabela no Banco de Dados.

7.5 – RAKE

Rake é uma ferramenta de build, escrita em Ruby, e semelhante ao **make** e ao **ant**, em escopo e propósito.

Rake tem as seguintes funcionalidades:

- Rakefiles (versão do rake para os Makefiles) são completamente definidas em sintaxe Ruby. Não existem arquivos XML para editar, nem sintaxe rebuscada como

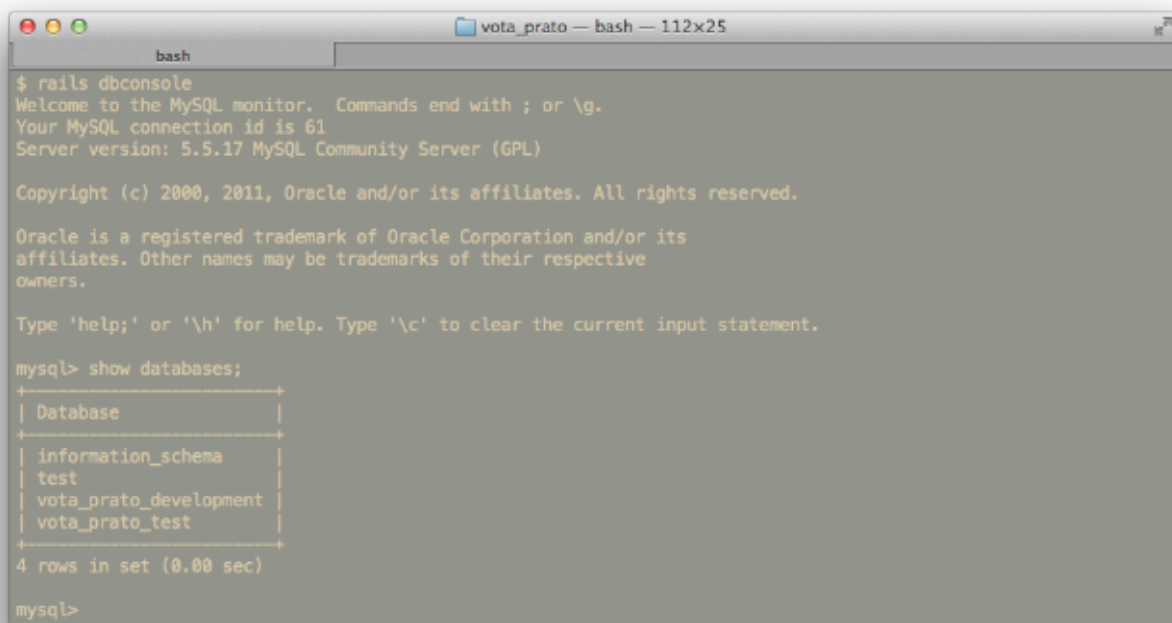
a do Makefile para se preocupar.

- É possível especificar tarefas com pré-requisitos.
- Listas de arquivos flexíveis que agem como arrays, mas sabem como manipular nomes de arquivos e caminhos (paths).
- Uma biblioteca de tarefas pré-compactadas para construir rakefiles mais facilmente.

Para criar nossas bases de dados, podemos utilizar a rake task **db:create**. Para isso, vá ao terminal e dentro do diretório do projeto digite:

```
$ rake db:create
```

O *Rails* criará dois databases no mysql: `vota_prato_development`, `vota_prato_test`.



```
bash
$ rails dbconsole
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 61
Server version: 5.5.17 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| test |
| vota_prato_development |
| vota_prato_test |
+-----+
4 rows in set (0.00 sec)

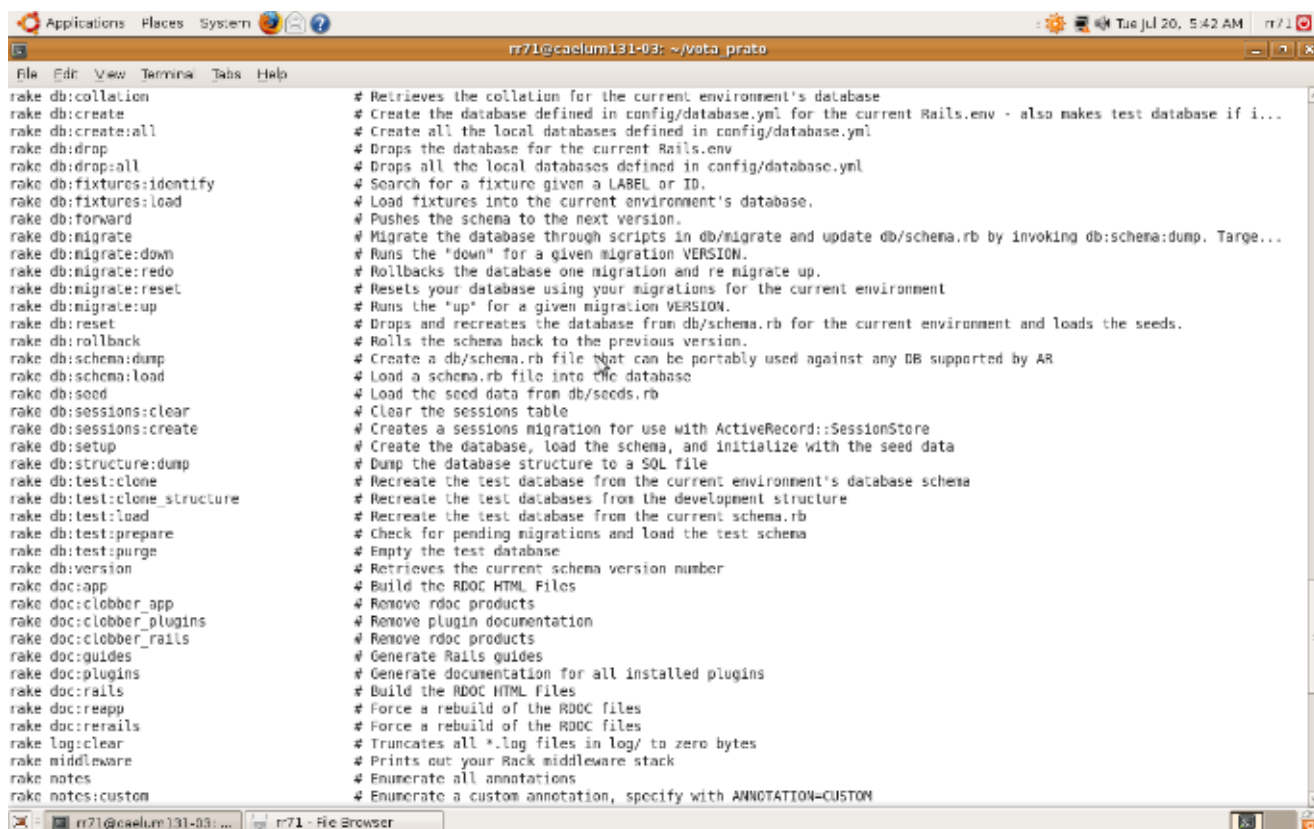
mysql>
```

rake db:create:all

Uma outra tarefa disponível em uma aplicação *Rails* e a rake `db:create:all`. Além de criar os dois databases já citados, ela também é responsável por criar o database `vota_prato_production`. No ambiente de desenvolvimento, é bem comum trabalharmos apenas com os databases de teste e desenvolvimento.

Para ver todas as tasks rake disponíveis no seu projeto podemos usar o comando (na raiz do seu projeto):

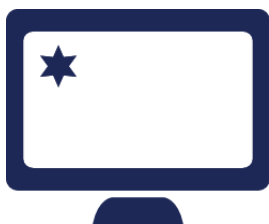
```
$ rake -T
```



```
File Edit View Terminal Tabs Help
n71@caelum131-03: ~/vota-prato

rake db:collation # Retrieves the collation for the current environment's database
rake db:create # Create the database defined in config/database.yml for the current Rails.env - also makes test database if i...
rake db:create:all # Create all the local databases defined in config/database.yml
rake db:drop # Drops the database for the current Rails.env
rake db:drop:all # Drops all the local databases defined in config/database.yml
rake db:fixtures:identify # Search for a fixture given a LABEL or ID.
rake db:fixtures:load # Load fixtures into the current environment's database.
rake db:forward # Pushes the schema to the next version.
rake db:migrate # Migrate the database through scripts in db/migrate and update db/schema.rb by invoking db:schema:dump. Targe...
rake db:migrate:down # Runs the 'down' for a given migration VERSION.
rake db:migrate:redo # Rolls back the database one migration and re migrate up.
rake db:migrate:reset # Resets your database using your migrations for the current environment
rake db:migrate:up # Runs the 'up' for a given migration VERSION.
rake db:reset # Drops and recreates the database from db/schema.rb for the current environment and loads the seeds.
rake db:rollback # Rolls the schema back to the previous version.
rake db:schema:dump # Create a db/schema.rb file that can be portably used against any DB supported by AR
rake db:schema:load # Load a schema.rb file into the database
rake db:seed # Load the seed data from db/seeds.rb
rake db:sessions:clear # Clear the sessions table
rake db:sessions:create # Creates a sessions migration for use with ActiveRecord::SessionStore
rake db:setup # Create the database, load the schema, and initialize with the seed data
rake db:structure:dump # Dump the database structure to a SQL file
rake db:test:clone # Recreate the test database from the current environment's database schema
rake db:test:clone:structure # Recreate the test databases from the development structure
rake db:test:load # Recreate the test database from the current schema.rb
rake db:test:prepare # Check for pending migrations and load the test schema
rake db:test:purge # Empty the test database
rake db:version # Retrieves the current schema version number
rake doc:app # Build the RDOC HTML files
rake doc:lobber_app # Remove rdoc products
rake doc:lobber_plugins # Remove plugin documentation
rake doc:lobber_rails # Remove rdoc products
rake doc:guides # Generate Rails guides
rake doc:plugins # Generate documentation for all installed plugins
rake doc:rails # Build the RDOC HTML files
rake doc:reapp # Force a rebuild of the RDOC files
rake doc:rerails # Force a rebuild of the RDOC files
rake log:clear # Truncates all *.log files in log/ to zero bytes
rake middleware # Prints out your Rack middleware stack
rake notes # Enumerate all annotations
rake notes:custom # Enumerate a custom annotation, specify with ANNOTATION=CUSTOM
```

Já conhece os cursos online Alura?



A **Alura** oferece dezenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Java, Ruby, Web, Mobile, .NET e outros, com uma **assinatura** que dá acesso a todos os cursos.

[Conheça os cursos online Alura.](#)

7.6 – CRIANDO MODELOS

Agora vamos criar o modelo do Restaurante. Para isso, temos um gerador específico para **model** através do comando *rails generate model restaurante*.

Repare que o Rails gerou uma série de arquivos para nós.

```

rr71@caelum131-03:~/vota_prato$ rails generate model restaurante
  invoke  active_record
  create  db/migrate/20100720084332_create_restaurantes.rb
  create  app/models/restaurante.rb
  invoke  test_unit
  create  test/unit/restaurante_test.rb
  create  test/fixtures/restaurantes.yml
rr71@caelum131-03:~/vota_prato$ rake db:create:all
(in /home/rr71/vota_prato)
vota_prato_development already exists
vota_prato_test already exists
vota_prato_production already exists
rr71@caelum131-03:~/vota_prato$

```

7.7 – MIGRATIONS

Migrations ajudam a gerenciar a evolução de um esquema utilizado por diversos bancos de dados. Foi a solução encontrada para o problema de como adicionar uma coluna no banco de dados local e propagar essa mudança para os demais desenvolvedores de um projeto e para o servidor de produção.

Com as migrations, podemos descrever essas transformações em classes que podem ser controladas por sistemas de controle de versão (por exemplo, git) e executá-las em diversos bancos de dados.

Sempre que executarmos a tarefa *Generator -> model*, o Rails se encarrega de criar uma migration inicial, localizado em **db/migrate**.

`ActiveRecord::Migration` é a classe que você deve estender ao criar uma migration.

Quando geramos nosso modelo na seção anterior, Rails gerou para nós uma migration (**db/migrate/<timestamp>_create_restaurantes.rb**). Vamos agora editar nossa migration com as informações que queremos no banco de dados.

Queremos que nosso restaurante tenha um nome e um endereço. Para isso, devemos acrescentar as chamadas de método abaixo:

```

t.string :nome, limit: 80
t.string :endereco

```

Faça isso dentro do método `change` da classe `CreateRestaurantes`. Sua migration deverá ficar como a seguir:

```

class CreateRestaurantes < ActiveRecord::Migration
  def change
    create_table :restaurantes do |t|

```

```

    t.string :nome, limit: 80
    t.string :endereco
    t.timestamps
  end
end
end

```

Supondo que agora lembramos de adicionar a especialidade do restaurante. Como fazer? Basta usar o outro gerador (*Generator*) do rails que cria migration. Por exemplo:

```
$ rails generate migration add_column_especialidade_to_restaurante
especialidade
```

Um novo arquivo chamado <timestamp>_add_column_especialidade_to_restaurante.rb será criado pelo *Rails*, e o código da migração gerada será como a seguir:

```

class AddColumnEspecialidadeToRestaurante < ActiveRecord::Migration
  def change
    add_column :restaurantes, :especialidade, :string
  end
end

```

Note que através do nome da migração e do parâmetro que passamos no gerador, o nome da coluna que queremos adicionar, nesse caso **especialidade**, o *Rails* deduziu que a migração é para adicionar a coluna e já fez o trabalho braçal para você!

Vamos apenas adicionar mais alguns detalhes a essa migração, vamos limitar o tamanho da string que será armazenada nesse campo para 40 caracteres:

```

class AddColumnEspecialidadeToRestaurante < ActiveRecord::Migration
  def change
    add_column :restaurantes, :especialidade, :string, limit: 40
  end
end

```

7.8 – EXERCÍCIOS: CRIANDO OS MODELOS

1. Crie nosso banco de dados

a. Entre no terminal, no diretório do projeto

b. execute o comando:

```
$ rake db:create
```

No terminal, acesse o mysql e verifique que os databases foram criados:

```
$ rails dbconsole
mysql> show databases;
mysql> quit
```

2. Crie o modelo do Restaurante

a. Novamente no Terminal, no diretório do projeto

b. execute **rails generate model restaurante**

3. Edite seu script de migração do modelo "restaurante" para criar os campos nome e endereço:

a. Abra o arquivo db/migrate/<timestamp>_create_restaurantes.rb

b. Adicione as seguintes linhas no bloco do método create_table:

```
t.string :nome, limit: 80
t.string :endereco
```

O código final de sua migration deverá ser como o que segue:

```
class CreateRestaurantes < ActiveRecord::Migration
  def change
    create_table :restaurantes do |t|
      t.string :nome, limit: 80
      t.string :endereco
      t.timestamps
    end
  end
end
```

4. Migre as tabelas para o banco de dados:

a. Vá ao Terminal

b. Execute a tarefa db:migrate:

```
$ rake db:migrate
```

Verifique no banco de dados se as tabelas foram criadas:

```
$ rails dbconsole
mysql> use vota_prato_development;
mysql> desc restaurantes;
mysql> quit
```

5. Faltou adiciona-r a coluna **especialidade** ao modelo "restaurante". Podemos adiciona-la criando uma nova migration:

a. Vá novamente ao Terminal e digite:

```
$ rails generate migration add_column_especialidade_to_restaurante
especialidade
```

b. Abra o arquivo db/migrate/<timestamp>_add_column_especialidade_to_restaurante.rb

c. Altere o limite de caracteres da coluna especialidade como a seguir:

```
add_column :restaurantes, :especialidade, :string, limit: 40
```

Seu arquivo ficará assim:

```
class AddColumnEspecialidadeToRestaurante < ActiveRecord::Migration
  def change
    add_column :restaurantes, :especialidade, :string, limit: 40
  end
end
```

d. Para efetivar a mudança no banco de dados execute a seguinte tarefa:

```
rake db:migrate
```

A saída do comando será semelhante a seguinte:

```
== CreateRestaurantes: migrating
=====
-- create_table(:restaurantes)
   -> 0.7023s
== CreateRestaurantes: migrated (0.7025s)
=====

== AddColumnEspecialidadeToRestaurante: migrating
=====
-- add_column(:restaurantes, :especialidade, :string, {:limit=>40})
   -> 0.9402s
== AddColumnEspecialidadeToRestaurante: migrated (0.9404s)
=====
```

e. Olhe novamente no banco de dados, veja que as tabelas foram realmente criadas.

```
$ rails dbconsole
mysql> use vota_prato_development;
mysql> show tables;
```

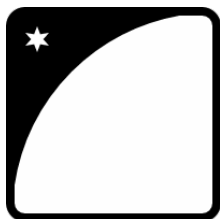
E o resultado será semelhante ao seguinte:

```
| Tables_in_vota_prato_development |
+-----+
| restaurantes                      |
| schema_migrations                 |
+-----+

2 rows in set (0.00 sec)
```

6. (Opcional) Utilizamos o método `add_column` na nossa *migration* para adicionar uma nova coluna. O que mais poderia ser feito? Abra a documentação e procure pelo módulo `ActiveRecord::ConnectionAdapters::SchemaStatements`.

Você não está nessa página a toa



Você chegou aqui porque a Caelum é referência nacional em cursos de Java, Ruby, Agile, Mobile, Web e .NET.

Faça curso com **quem escreveu essa apostila**.

[Consulte as vantagens do curso *Desenv. Ágil para Web com Ruby on Rails*.](#)

7.9 – MANIPULANDO NOSSOS MODELOS PELO CONSOLE

Podemos utilizar o console para escrever comandos Ruby, e testar nosso modelo. A grande vantagem disso, é que não precisamos de controladores ou de uma view para testar se nosso modelo funciona de acordo com o esperado e se nossas regras de validação estão funcionando. Outra grande vantagem está no fato de que se precisarmos manipular nosso banco de dados, ao invés de termos de conhecer a sintaxe sql e digitar a query manualmente, podemos utilizar código ruby e manipular através do nosso console.

Para criar um novo restaurante, podemos utilizar qualquer um dos jeitos abaixo:

```
r = Restaurante.new
r.nome = "Fasano"
r.endereco = "Av. dos Restaurantes, 126"
r.especialidade = "Comida Italiana"
r.save

r = Restaurante.new do |r|
  r.nome = "Fasano"
  r.endereco = "Av. dos Restaurantes, 126"
  r.especialidade = "Comida Italiana"
end
r.save
```

Uma outra forma possível para a criação de objetos do Active Record é usando um *hash* como parâmetro no construtor. As chaves do hash precisam coincidir com nomes das propriedades as quais queremos atribuir os valores. Veja o

exemplo a seguir:

```
r = Restaurante.new nome: "Fasano",  
                    endereco: "Av. dos Restaurantes, 126",  
                    especialidade: "Comida Italiana"  
r.save
```

Note que o comando `save` efetua a seguinte ação: se o registro não existe no banco de dados, cria um novo registro; se já existe, atualiza o registro existente.

Existe também o comando `save!`, que tenta salvar o registro, mas ao invés de apenas retornar **"false"** se não conseguir, lança a exceção `RecordNotSaved`.

Para atualizar um registro diretamente no banco de dados, podemos fazer:

```
Restaurante.update(1, {nome: "1900"})
```

Para atualizar múltiplos registros no banco de dados:

```
Restaurante.update_all("especialidade = 'Massas'")
```

Ou, dado um objeto `r` do tipo `Restaurante`, podemos utilizar:

```
r.update_attribute(:nome, "1900")
```

```
r.update_attributes nome: "1900", especialidade: "Pizzas"
```

Existe ainda o comando `update_attributes!`, que chama o comando `save!` ao invés do comando `save` na hora de salvar a alteração.

Para remover um restaurante, também existem algumas opções. Todo `ActiveRecord` possui o método `destroy`:

```
restaurante = Restaurante.first  
restaurante.destroy
```

Para remover o restaurante de id **1**:

```
Restaurante.destroy(1)
```

Para remover os restaurantes de ids **1, 2 e 3**:

```
restaurantes = [1,2,3]  
Restaurante.destroy(restaurantes)
```

Para remover **todos** os restaurantes:

```
Restaurante.destroy_all
```

Podemos ainda remover todos os restaurantes que obedecem determinada condição, por exemplo:

```
Restaurante.destroy_all(especialidade: "italiana")
```

Os métodos `destroy` sempre fazem primeiro o `find(id)` para depois fazer o `destroy()`. Se for necessário evitar o `SELECT` antes do `DELETE`, podemos usar o método `delete()`:

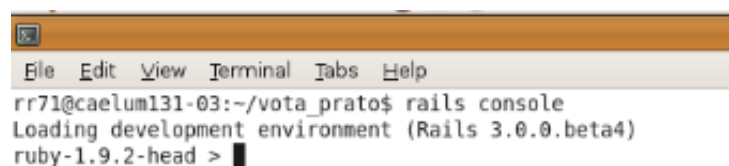
```
Restaurante.delete(1)
```

7.10 – EXERCÍCIOS: MANIPULANDO REGISTROS

Teste a manipulação de registros pelo console.

1. Insira um novo restaurante

a. Para ter acesso ao Console, basta digitar **rails console** no Terminal



b. Digite:

```
r = Restaurante.new nome: "Fasano",  
                    endereco: "Av. dos Restaurantes, 126",  
                    especialidade: "Comida Italiana"
```

c. Olhe seu banco de dados:

```
$ rails dbconsole  
mysql> select * from restaurantes;
```

d. Volte para o Console e digite:

```
r.save
```

e. Olhe seu banco de dados novamente:

```
$ rails dbconsole  
mysql> select * from restaurantes;
```

2. Atualize seu restaurante

a. Digite:

```
r.update_attributes nome: "1900"
```

b. Olhe seu banco de dados novamente:

```
$ rails dbconsole  
mysql> select * from restaurantes;
```

3. Vamos remover o restaurante criado:

a. Digite

```
Restaurante.destroy(1)
```

b. Olhe seu banco de dados e veja que o restaurante foi removido

```
$ rails dbconsole  
mysql> select * from restaurantes;
```

7.11 – EXERCÍCIOS OPCIONAIS

1. Teste outras maneiras de efetuar as operações do exercício anterior.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Conheça os títulos e a nova proposta, você vai gostar.

[Casa do Código, livros para o programador.](#)

7.12 – FINDERS

O ActiveRecord possui o método "**find**" para realizar buscas. Esse método, aceita os seguintes parâmetros:

```
Restaurante.all    # retorna todos os registros
Restaurante.first  # retorna o primeiro registro
Restaurante.last   # retorna o último registro
```

Ainda podemos passar para o método find uma lista com os id's dos registros que desejamos:

```
r = Restaurante.find(1)
varios = Restaurante.find(1,2,3)
```

Além desses, podemos definir condições para nossa busca (como o SELECT do MySQL). Existem diversas formas de declararmos essas condições:

```
Restaurante.where("nome = 'Fasano' and especialidade = 'massa'")
Restaurante.where(["nome = ? and especialidade = ?",
  'Fasano', 'massa'])
Restaurante.where(["nome = :nome and especialidade = :especialidade", {
  nome: "Fasano", especialidade: "Massa"}])
Restaurante.where({nome: "Fasano", especialidade: "massa" })
```

Essas quatro formas fazem a mesma coisa, ou seja, procuram por registros com o campo nome = "Fasano" e o campo especialidade = "massa".

Para finalizar, podemos chamar outros métodos encadeados para fazer queries mais complexas:

- .order – define a ordenação. Ex: "created_at DESC, nome".
- .group – nome do atributo pelo qual os resultados serão agrupados. Efeito idêntico ao do comando SQL GROUP BY.
- .limit – determina o limite do número de registros que devem ser retornados
- .offset – determina o ponto de início da busca. Ex: para offset = 5, iria pular os registros de 0 a 4.
- .include – permite carregar relacionamentos na mesma consulta usando LEFT OUTER JOINS.

Exemplo mais completo:

```
Restaurante.where('nome like :nome', {nome: '%teste%'}).
  order('nome DESC').limit(20)
```

Para Sabe Mais – Outras opções para os finders

Existem mais opções, como o ":lock", que podem ser utilizadas, mas não

serão abordadas nesse curso. Você pode consultá-las na documentação da API do Ruby on Rails.

7.13 – EXERCÍCIOS: BUSCAS DINÂMICAS

1. Vamos testar os métodos de busca:

a. Abra o console (`rails console` no Terminal)

b. Digite:

```
Restaurante.first
```

c. Aperte **enter**

d. Digite:

```
Restaurante.all
```

e. Aperte **enter**

f. Digite:

```
Restaurante.find(1)
```

g. Aperte **enter**

h. Digite:

```
Restaurante.where(["nome = ? AND especialidade = ?",  
                  "Fasano", "Comida Italiana"])
```

i. Aperte **enter**

j. Digite:

```
Restaurante.order("especialidade DESC").limit(1)
```

k. Aperte **enter**

7.14 – VALIDAÇÕES

Ao inserir um registro no banco de dados é bem comum a entrada de dados inválidos.

Existem alguns campos de preenchimento obrigatório, outros que só aceitem números, que não podem conter dados já existentes, tamanho máximo e mínimo etc.

Para ter certeza que um campo foi preenchido antes de salvar no banco de dados, é necessário pensar em três coisas: "como validar a entrada?", "qual o campo a ser validado?" e "o que acontece ao tentar salvar uma entrada inválida?".

Para validar esses registros, podemos implementar o método `validate` em qualquer `ActiveRecord`, porém o Rails disponibiliza alguns comandos prontos para as validações mais comuns. São eles:

- `validates_presence_of`: verifica se um campo está preenchido;
- `validates_size_of`: verifica o comprimento do texto do campo;
- `validates_uniqueness_of`: verifica se não existe outro registro no banco de dados que tenha a mesma informação num determinado campo;
- `validates_numericality_of`: verifica se o preenchimento do campo é numérico;
- `validates_associated`: verifica se o relacionamento foi feito corretamente;
- etc...

Todos estes métodos disponibilizam uma opção (**`:message`**) para personalizar a mensagem de erro que será exibida caso a regra não seja cumprida. Caso essa opção não seja utilizada, será exibida uma mensagem padrão.

Toda mensagem de erro é gravada num hash chamado `errors`, presente em todo `ActiveRecord`.

Além dos validadores disponibilizados pelo rails, podemos utilizar um validador próprio:

```
validate :garante_alguma_coisa
```

```
def garante_alguma_coisa
  errors.add_to_base("Deve respeitar nossa regra") unless campo_valido?
end
```

Repare que aqui, temos que incluir manualmente a mensagem de erro padrão do nosso validador.

Se quisermos que o nome do nosso restaurante comece com letra maiúscula, poderíamos fazer:


```
validate :primeira_letra_deve_ser_maiuscula
```

```
private
def primeira_letra_deve_ser_maiuscula
  errors.add("nome",
    "primeira letra deve ser maiúscula") unless nome =~ /[A-Z].*/
end
```

Existe ainda uma outra maneira de criar validações que facilita os casos onde temos vários validadores para o mesmo campo, como por exemplo:

```
validates :nome, presence: true, uniqueness: true, length: {maximum: 50}
```

equivalente a:

```
validates_presence_of :nome
validates_uniqueness_of :nome
validates_length_of :nome, :maximum => 50
```

Modificadores de acesso

Utilizamos aqui, o modificador de acesso **private**. A partir do ponto que ele é declarado, todos os métodos daquela classe serão privados, a menos que tenha um outro modificador de acesso que modifique o acesso a outros métodos.

Validadores prontos

Esse exemplo poderia ter sido reescrito utilizando o validador "validates_format_of", que verifica se um atributo confere com uma determinada expressão regular.

Agora é a melhor hora de aprender algo novo



Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos novos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

[Conheça a Alura.](#)

7.15 – EXERCÍCIOS: VALIDAÇÕES

1. Para nosso restaurante implementaremos a validação para que o campo nome, endereço e especialidade não possam ficar vazios, nem que o sistema aceite dois restaurantes com o mesmo nome e endereço.

a. Abra o modelo do restaurante (**app/models/restaurante.rb**)

b. inclua as validações:

```
validates_presence_of :nome, message: "deve ser preenchido"
validates_presence_of :endereco, message: "deve ser preenchido"
validates_presence_of :especialidade, message: "deve ser preenchido"

validates_uniqueness_of :nome, message: "nome já cadastrado"
validates_uniqueness_of :endereco, message: "endereço já cadastrado"
```

2. Inclua a validação da primeira letra maiúscula:

```
validate :primeira_letra_deve_ser_maiuscula
```

```
private
def primeira_letra_deve_ser_maiuscula
  errors.add(:nome,
    "primeira letra deve ser maiúscula") unless nome =~ /[A-Z].*/
end
```

3. Agora vamos testar nossos validadores:

a. Abra o Terminal

b. Entre no Console (rails console)

c. Digite:

```
r = Restaurante.new nome: "fasano",
  endereco: "Av. dos Restaurantes, 126"
r.save
```

d. Verifique a lista de erros, digitando:

```
r.valid?           # verifica se objeto passa nas validações
r.errors.empty?    # retorna true/false indicando se há erros ou não
r.errors.count     # retorna o número de erros
r.errors[:nome]    # retorna apenas o erro do atributo nome

r.errors.each {|field, msg| puts "#{field} - #{msg}"}
```

7.16 – PLURALIZAÇÃO COM INFLECTIONS

Repare que o Rails pluralizou o termo **restaurante** automaticamente. Por exemplo, o nome da tabela é **restaurantes**. O Rails se baseia nas regras gramaticais da língua inglesa para fazer essa pluralização automática.

Apesar de **restaurante** ter sido plurificado corretamente, outros termos como **qualificacao** por exemplo, será plurificado para **qualificacaos**, ao invés de **qualificacoes**. Podemos fazer o teste no console do Rails, a String tem os métodos `pluralize` e `singularize` para fazer, respectivamente, pluralização e singularização do termo:

```
"qualificacao".pluralize # => "qualificacaos"  
"qualificacoes".singularize # => "qualificacao"
```

Outro termo onde encontraremos problema é em "receita":

```
"receita".pluralize # => "receita"  
"receitas".singularize # => "receita"
```

Apesar da singularização estar correta, a pluralização não ocorre, isso acontece por que de acordo com as regras gramaticais do inglês, a palavra **receita** seria plural de **receitum**. Testando no console obteremos o seguinte resultado:

```
"receitum".pluralize # => "receita"
```

Ou seja, teremos que mudar as regras padrões utilizadas pelo Rails. Para isso, iremos editar o arquivo **config/initializers/inflections.rb**, adicionando uma linha indicando que a pluralização das palavras **qualificacao** e **receita** é, respectivamente, **qualificacoes** e **receitas**:

```
ActiveSupport::Inflector.inflections do |inflect|  
  inflect.irregular 'qualificacao', 'qualificacoes'  
  inflect.irregular 'receita', 'receitas'  
end
```

Feito isso, podemos testar nossas novas regras utilizando o console:

```
"qualificacao".pluralize # => "qualificacoes"  
"qualificacoes".singularize # => "qualificacao"  
"receita".pluralize # => "receitas"  
"receitas".singularize # => "receita"
```

7.17 – EXERCÍCIOS – COMPLETANDO NOSSO MODELO

1. Vamos corrigir a pluralização da palavra 'receita'

a. Abra o arquivo **"config/initializers/inflections.rb"**

b. Adicione as seguintes linhas ao final do arquivo:

```
ActiveSupport::Inflector.inflections do |inflect|  
  inflect.irregular 'receita', 'receitas'  
end
```

2. a. Execute o console do Rails:

```
$ rails console
```

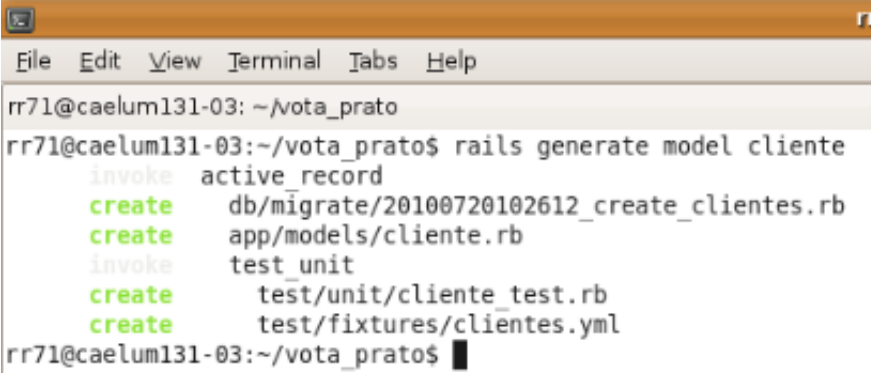
b. No Console do Rails digite:

```
"receita".pluralize
```

3. Vamos criar o nosso modelo Cliente, bem como sua migration:

a. No terminal, digite:

```
$ rails generate model cliente
```



```
rr71@caelum131-03: ~/vota_prato  
rr71@caelum131-03:~/vota_prato$ rails generate model cliente  
  invoke  active_record  
    create  db/migrate/20100720102612_create_clientes.rb  
    create  app/models/cliente.rb  
  invoke  test_unit  
    create  test/unit/cliente_test.rb  
    create  test/fixtures/clientes.yml  
rr71@caelum131-03:~/vota_prato$
```

b. Abra o arquivo "db/migrate/<timestamp>_create_clientes.rb"

c. Edite o método change para que ele fique como a seguir:

```
create_table :clientes do |t|  
  t.string :nome, limit: 80  
  t.integer :idade  
  t.timestamps  
end
```

d. Vamos efetivar a migration usando o comando:

```
$ rake db:migrate
```

e. Olhe no console o que foi feito

```
rr71@caelum131-03: ~/vota_prato
File Edit View Terminal Tabs Help
rr71@caelum131-03: ~/vota_prato$ rake db:migrate
(in /home/rr71/vota_prato)
== CreateClientes: migrating =====
-- create_table(:clientes)
   -> 0.0039s
== CreateClientes: migrated (0.0041s) =====
rr71@caelum131-03:~/vota_prato$
```

f. Olhe no banco de dados!

```
mysql> show tables;
+-----+
| Tables_in_vota_prato_development |
+-----+
| clientes                          |
| restaurantes                      |
| schema_migrations                |
+-----+
3 rows in set (0.01 sec)
```

4. Vamos agora fazer as validações no modelo "cliente":
- a. Abra o arquivo "app/models/cliente.rb"
 - b. Adicione as seguintes linhas:

validates_presence_of :nome, message: " - deve ser preenchido"

validates_uniqueness_of :nome, message: " - nome já cadastrado"

validates_numericality_of :idade, greater_than: 0,
less_than: 100,
message: " - deve ser um número entre 0 e 100"
5. Vamos criar o modelo Prato, bem como sua migration:
- a. Vamos executar o generator de model do rails novamente.
 - b. rails generate model prato no Terminal
 - c. Abra o arquivo "db/migrate/<timestamp>_create_pratos.rb"
 - d. Adicione as linhas:

t.string :nome, limit: 80
 - e. Volte para o Terminal

f. execute `rake db:migrate`

g. Olhe no console o que foi feito

h. Olhe no banco de dados!

6. Vamos agora fazer as validações no modelo **"prato"**. Ainda no arquivo `app/models/prato.rb` adicione o seguinte código:

```
validates_presence_of :nome, message: " - deve ser preenchido"
```

```
validates_uniqueness_of :nome, message: " - nome já cadastrado"
```

7. Vamos criar o modelo Receita, bem como sua migration:

a. Vá ao Terminal

b. Execute `rails generate model receita`

c. Abra o arquivo **"db/migrate/<timestamp>_create_receitas.rb"**

d. Adicione as linhas:

```
t.text :conteudo
```

e. Volte ao Terminal

f. Execute `rake db:migrate`

g. Olhe no console o que foi feito

h. Olhe no banco de dados!

8. Vamos agora fazer as validações no modelo **"receita"**:

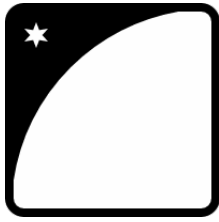
a. Abra o arquivo **"app/models/receita.rb"**

b. Adicione as seguintes linhas:

```
validates_presence_of :conteudo, message: " - deve ser preenchido"
```

Você pode também fazer o curso RR-71 dessa apostila na Caelum

Querendo aprender ainda mais sobre a linguagem Ruby e o framework Ruby on Rails? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?



A Caelum oferece o **curso RR-71** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

[Consulte as vantagens do curso *Desenv. Ágil para Web com Ruby on Rails*.](#)

7.18 – EXERCÍCIOS – CRIANDO O MODELO DE QUALIFICAÇÃO

1. Vamos corrigir a pluralização da palavra 'qualificacao'

a. Abra o arquivo "**config/initializers/inflections.rb**"

b. Adicione a seguinte inflection:

```
inflect.irregular 'qualificacao', 'qualificacoes'
```

2. a. Execute o console do Rails:

```
$ rails console
```

b. No Console do Rails digite:

```
"qualificacao".pluralize
```

3. Vamos continuar com a criação do nosso modelo Qualificacao e sua migration.

a. No Terminal digite

b. rails generate model qualificacao

```
rr71@caelum131-03: ~/vota_prato
rr71@caelum131-03:~/vota_prato$ rails generate model qualificacao
  invoke  active_record
  create   db/migrate/20100720110602_create_qualificacoes.rb
  create   app/models/qualificacao.rb
  invoke  test_unit
  create   test/unit/qualificacao_test.rb
  create   test/fixtures/qualificacoes.yml
rr71@caelum131-03:~/vota_prato$
```

c. Abra o arquivo "**db/migrate/<timestamp>_create_qualificacoes.rb**"

d. Adicione as linhas:

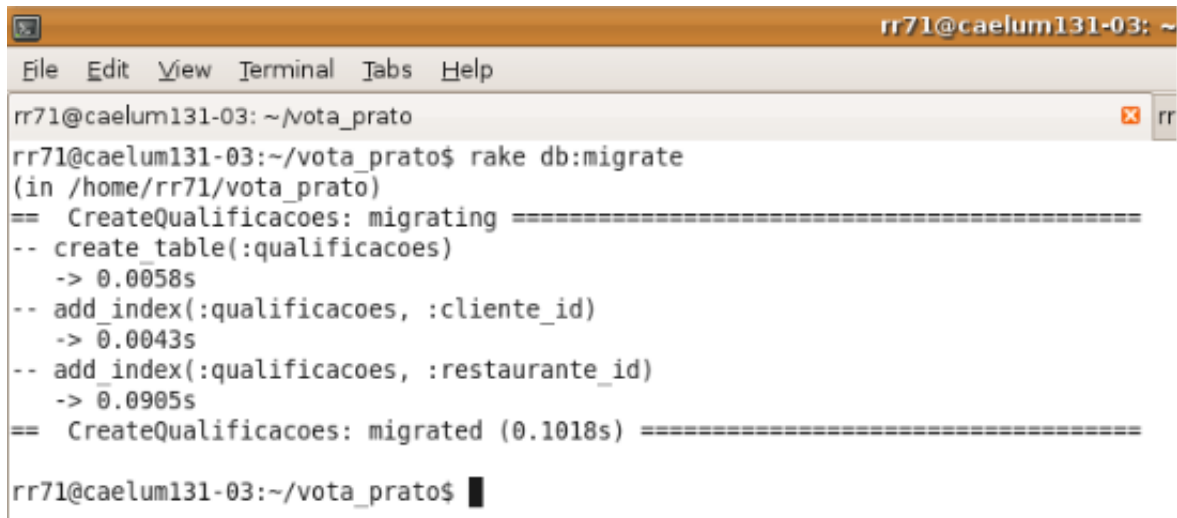
```
t.float :nota  
t.float :valor_gasto
```

e. Volte ao Terminal

f. Execute a task para rodar as migrações pendentes:

```
$ rake db:migrate
```

g. Olhe no console o que foi feito



```
rr71@caelum131-03: ~/vota_prato  
rr71@caelum131-03:~/vota_prato$ rake db:migrate  
(in /home/rr71/vota_prato)  
== CreateQualificacoes: migrating =====  
-- create table(:qualificacoes)  
   -> 0.0058s  
-- add_index(:qualificacoes, :cliente_id)  
   -> 0.0043s  
-- add_index(:qualificacoes, :restaurante_id)  
   -> 0.0905s  
== CreateQualificacoes: migrated (0.1018s) =====  
rr71@caelum131-03:~/vota_prato$
```

h. Olhe no banco de dados

```
mysql> show tables;  
+-----+  
| Tables_in_vota_prato_development |  
+-----+  
| clientes  
| pratos  
| qualificacoes  
| receitas  
| restaurantes  
| schema_migrations  
+-----+  
6 rows in set (0.00 sec)
```

4. Vamos agora fazer as validações no modelo "qualificacao":

a. Abra o arquivo "app/models/qualificacao.rb"

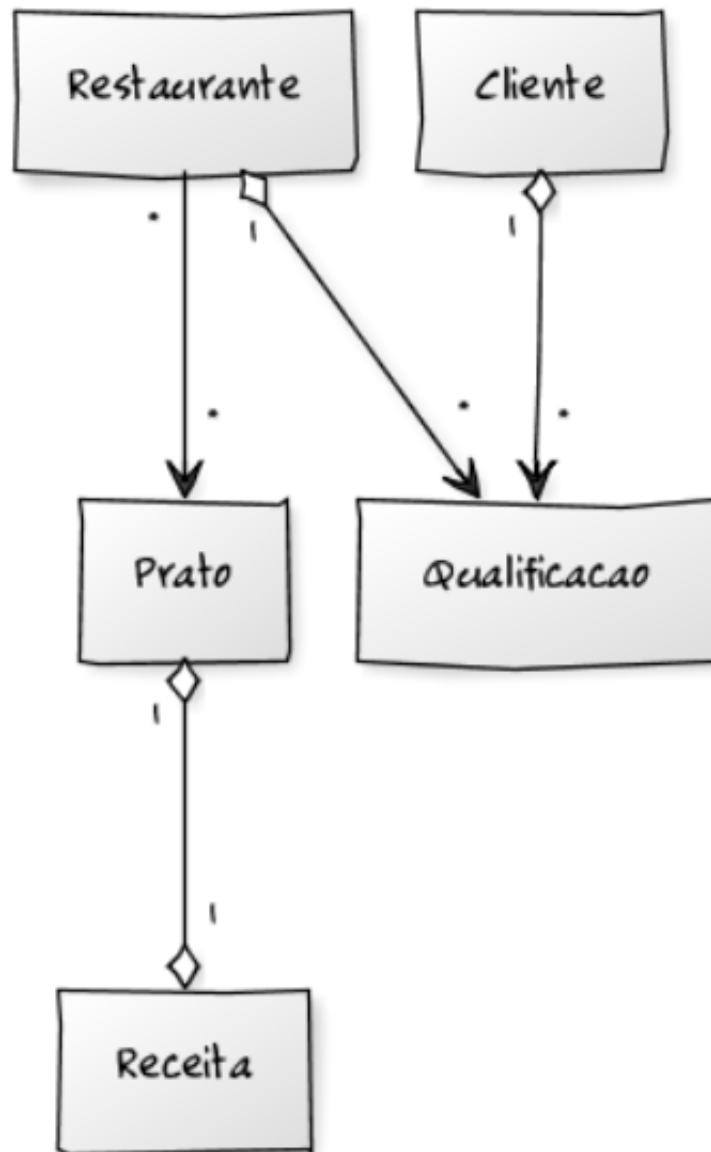
b. Adicione as seguintes linhas:

```
validates_presence_of :nota, message: " - deve ser preenchido"  
validates_presence_of :valor_gasto, message: " - deve ser preenchido"  
  
validates_numericality_of :nota, greater_than_or_equal_to: 0,  
                           less_than_or_equal_to: 10,  
                           message: " - deve ser um número entre 0 e  
10"  
  
validates_numericality_of :valor_gasto, greater_than: 0,
```


maior que 0"

message: " - deve ser um número

7.19 - RELACIONAMENTOS



Para relacionar diversos modelos, precisamos informar ao *Rails* o tipo de relacionamento entre eles. Quando isso é feito, alguns métodos são criados para podermos manipular os elementos envolvidos nesse relacionamento. Os relacionamentos que Rails disponibiliza são os seguintes:

- `belongs_to` - usado quando um modelo tem como um de seus atributos o id de outro modelo (many-to-one ou one-to-one). Quando dissermos que uma qualificação **belongs_to** um restaurante, ainda ganharemos os seguintes métodos:

- `Qualificacao.restaurante` (similar ao `Restaurante.find(restaurante_id)`)

- `Qualificacao.restaurante=(restaurante)` (similar ao `qualificacao.restaurante_id = restaurante.id`)
- `Qualificacao.restaurante?` (similar ao `qualificacao.restaurante == algum_restaurante`)
- **has_many** – associação que provê uma maneira de mapear uma relação one-to-many entre duas entidades. Quando dissermos que um restaurante **has_many** qualificações, ganharemos os seguintes métodos:
 - `Restaurante.qualificacoes` (semelhante ao `Qualificacao.find :all, conditions: ["restaurante_id = ?", id]`)
 - `Restaurante.qualificacoes<<`
 - `Restaurante.qualificacoes.delete`
 - `Restaurante.qualificacoes=`
- **has_and_belongs_to_many** – associação muitos-para-muitos, que é feita usando uma tabela de mapeamento. Quando dissermos que um prato **has_and_belongs_to_many** restaurantes, ganharemos os seguintes métodos:
 - `Prato.restaurantes`
 - `Prato.restaurantes<<`
 - `Prato.restaurantes.delete`
 - `Prato.restaurantes=`

Além disso, precisaremos criar a tabela **pratos_restaurantes**, com as colunas **prato_id** e **restaurante_id**. Por convenção, o nome da tabela é a concatenação do nome das duas outras tabelas, seguindo a ordem alfabética.

- **has_one** – lado bidirecional de uma relação um-para-um. Quando dissermos que um prato **has_one** receita, ganharemos os seguintes métodos:
 - `Prato.receita`, (semelhante ao `Receita.find(:first, conditions: "prato_id = id")`)
 - `Prato.receita=`

7.20 – PARA SABER MAIS: AUTO-RELACIONAMENTO

Os relacionamentos vistos até agora foram sempre entre dois objetos de classes

diferentes. Porém existem relacionamentos entre classes do mesmo tipo, por exemplo, uma Categoria de um site pode conter outras categorias, onde cada uma contém ainda suas categorias. Esse relacionamento é chamado *auto-relacionamento*.

No ActiveRecord temos uma forma simples para fazer essa associação.

```
class Category < ActiveRecord::Base
  has_many :children, class_name: "Category",
                    foreign_key: "father_id"
  belongs_to :father, class_name: "Category"
end
```

Tire suas dúvidas no novo GUJ Respostas



O GUJ é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

[Faça sua pergunta.](#)

7.21 – PARA SABER MAIS: CACHE

Todos os resultados de métodos acessados de um relacionamento são obtidos de um cache e não novamente do banco de dados. Após carregar as informações do banco, o ActiveRecord só volta se ocorrer um pedido explícito. Exemplos:

restaurante.qualificacoes	# busca no banco de dados
restaurante.qualificacoes.size	# usa o cache
restaurante.qualificacoes.empty?	# usa o cache
restaurante.qualificacoes(true).size	# força a busca no banco de dados
restaurante.qualificacoes	# usa o cache

7.22 – EXERCÍCIOS – RELACIONAMENTOS

1. Vamos incluir os relacionamentos necessários para nossos modelos:

a. Abra o arquivo "**app/models/cliente.rb**"

b. Adicione a seguinte linha:

has_many :qualificacoes

c. Abra o arquivo "app/models/restaurante.rb"

d. Adicione a seguinte linha:

has_many :qualificacoes
has_and_belongs_to_many :pratos

e. Abra o arquivo "app/models/qualificacao.rb"

f. Adicione as seguintes linhas:

belongs_to :cliente
belongs_to :restaurante

g. Abra o arquivo "app/models/prato.rb"

h. Adicione as seguintes linhas:

has_and_belongs_to_many :restaurantes
has_one :receita

i. Abra o arquivo "app/models/receita"

j. Adicione a seguinte linha:

belongs_to :prato

2. Para persistirmos cada relacionamento, é necessário criarmos colunas com os ids dos modelos relacionados em nossas tabelas:

a. Vamos adicionar uma coluna com o id do prato em cada receita. Para isso, digite **rails generate migration addColumnPratoIdToReceitas**
prato_id:integer

b. Abra o arquivo **db/migrate/<timestamp>_add_column_prato_id_to_receitas.rb**. O código deverá estar da seguinte forma:

add_column :receitas, :prato_id, :integer

c. Vá ao terminal e execute as migrações:

\$ rake db:migrate

3. Vamos fazer o mesmo para os relacionamentos do modelo de qualificação

a. Vá ao terminal e execute o seguinte código:

\$ rails generate migration addClienteIdAndRestauranteIdToQualificacoes

```
cliente_id:integer restaurante_id:integer
```

b. Abra o arquivo

db/migrate/<timestamp>_add_cliente_id_and_restaurante_id_to_qualificacoes.rb. O código deverá apresentar as seguintes linhas:

```
add_column :qualificacoes, :cliente_id, :integer
add_column :qualificacoes, :restaurante_id, :integer
```

c. Execute as migrações:

```
$ rake db:migrate
```

4. Vamos criar a tabela para nosso relacionamento **has_and_belongs_to_many**:

a. Vá para o Terminal

b. Digite: **rails generate migration createPratosRestaurantesJoinTable**

c. Abra o arquivo

"db/migrate/<timestamp>_create_pratos_restaurantes_join_table.rb"

d. O rails não permite que uma tabela de ligação para um relacionamento **has_and_belongs_to_many** possua chave primária auto incremento. Por isso, vamos fazer com que o rails não gere essa chave. Basta passar o parâmetro `id: false` para o método `create_table`. Dessa forma, teremos que digitar o seguinte no método `change`:

```
create_table :pratos_restaurantes, id: false do |t|
  t.integer :prato_id
  t.integer :restaurante_id
end
```

e. Volte ao Terminal e execute as migrações:

```
$ rake db:migrate
```

f. Olhe no console o que foi feito

g. Olhe no banco de dados

5. Vamos incluir as validações que garantam que os relacionamentos foram feitos corretamente:

a. Abra o arquivo **"app/models/qualificacao.rb"**

b. Adicione as seguintes linhas:

```
validates_presence_of :cliente_id, :restaurante_id
validates_associated :cliente, :restaurante
```

c. Abra o arquivo **"app/models/receita.rb"**

d. Adicione as seguintes linhas:

```
validates_presence_of :prato_id  
validates_associated :prato
```

e. Abra o arquivo **"app/models/prato.rb"**

f. Adicione as seguintes linhas:

```
validate :validate_presence_of_more_than_one_restaurante  
  
private  
def validate_presence_of_more_than_one_restaurante  
  errors.add("restaurantes",  
    "deve haver ao menos um restaurante") if restaurantes.empty?  
end
```

6. Faça alguns testes no terminal para testar essas novas validações.

7.23 – PARA SABER MAIS – EAGER LOADING

Podemos cair em um problema grave caso o número de qualificações para um restaurante seja muito grande. Imagine que um determinado restaurante do nosso sistema possua com 100 qualificações. Queremos mostrar todas as qualificações desse restaurante, então fazemos um for:

```
for qualificacao in Qualificacao.all  
  puts "restaurante: " + qualificacao.restaurante.nome  
  puts "cliente:      " + qualificacao.cliente.nome  
  puts "qualificacao: " + qualificacao.nota  
end
```

Para iterar sobre as 100 qualificações do banco de dados, seriam geradas 201 buscas! Uma busca para todas as qualificações, 100 buscas para cada restaurante mais 100 buscas para cada cliente! Podemos melhorar um pouco essa busca. Podemos pedir ao ActiveRecord que inclua o restaurante quando fizer a busca:

```
Qualificacao.find(:all, include: :restaurante)
```

Bem melhor! Agora a quantidade de buscas diminuiu para 102! Uma busca para as qualificações, outra para todos os restaurantes e mais 100 para os clientes. Podemos utilizar a mesma estratégia para otimizar a busca de clientes:

```
Qualificacao.includes(:restaurante, :cliente)
```

Com essa estratégia, teremos o número de buscas muito reduzido. A quantidade total agora será de 1 + o número de associações necessárias. Poderíamos ir mais além, e trazer uma associação de uma das associações existentes em qualificação:

```
Qualificacao.includes(:cliente, restaurante: {pratos: :receita})
```

Nova editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

7.24 – PARA SABER MAIS – NAMED SCOPES

Para consultas muito comuns, podemos usar o recurso de **Named Scopes** oferecido pelo ActiveRecord, que permite deixarmos alguns tipos de consultas comuns "preparadas".

Imagine que a consulta de restaurantes de especialidade "massa" seja muito comum no nosso sistema. Podemos facilitá-la criando um named scope na classe Restaurante:

```
class Restaurante < ActiveRecord::Base
  scope :massas, where(especialidade: 'massas')
end
```

As opções mais comuns do método find também estão disponíveis para *named scopes*, como :conditions, :order, :select e :include.

Com o *named scope* definido, a classe ActiveRecord ganha um método de mesmo nome, através do qual podemos recuperar os restaurantes de especialidade "massas" de forma simples:

```
Restaurante.massas
Restaurante.massas.first
Restaurante.massas.last
```

```
Restaurante.massas.where(["nome like ?", '%x%'])
```

O método associado ao *named scope* criado retorna um objeto da classe `ActiveRecord::NamedScope`, que age como um `Array`, mas aceita a chamada de alguns métodos das classes `ActiveRecord`, como o `find` para filtrar ainda mais a consulta.

Podemos ainda definir diversos *named scopes* e combiná-los de qualquer forma:

```
class Restaurante < ActiveRecord::Base
  scope :massas, where(especialidade: 'massas')
  scope :recentes, where(["created_at > ?", 3.months.ago])
  scope :pelo_nome, order('nome')
end
```

```
Restaurante.massas # todos de especialidade = 'massas'
Restaurante.recentes # todos de created_at > 3 meses atras
```

```
# especialidade = 'massas' e created_at > 3 meses atras
Restaurante.massas.recentes
Restaurante.recentes.massas
```

```
Restaurante.massas.pelo_nome.recentes
```

7.25 - PARA SABER MAIS - MODULES

As associações procuram por relacionamentos entre classes que estejam no mesmo módulo. Caso precise de relacionamentos entre classes em módulos distintos, é necessário informar o nome completo da classe no relacionamento:

```
module Restaurante
  module RH
    class Pessoa < ActiveRecord::Base
    end
  end

  module Financeiro
    class Pagamento < ActiveRecord::Base
      belongs_to :pessoa, class_name: "Restaurante::RH::Pessoa"
    end
  end
end
```

CAPÍTULO ANTERIOR:

[Ruby on Rails](#)

PRÓXIMO CAPÍTULO:

[Rotas](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter