Associations - code-along - c. 1hr+

Check out rails api docs

Associations between models make common operations simpler and easier in your code.

Consider a simple Rails application that includes a model for customers and a model for orders. Each customer can have lots of (many) orders, and a order belongs to a customer.

Write the one-to-many relationship the board and the 3 Customers we're creating.

```
rails new associations
cd associations

git init
git add .
git commit -m "Initial commit."

rails g model Customer name:string
rails g model Order customer_id:integer value:float
rake db:create db:migrate

git add .
git commit -m "Added Customer and Order models."

rails c

marc = Customer.create name: "marc"
mike = Customer.create name: "mike"
gerry = Customer.create name: "gerry"

Order.create customer_id: marc.id, value: 10.32
Order.create customer_id: marc.id, value: 15.73
Order.create customer_id: mike.id, value: 1.29
Order.create customer_id: mike.id, value: 99.99
Order.create customer_id: marc.id, value: 110.33
Order.create customer_id: marc.id, value: 21.15
Order.create customer_id: mike.id, value: 201.91
Order.create customer_id: gerry.id, value: 98.89
```

How do I find all the orders for Gerry?

`Order.where customer_id: gerry.id`

All easy enough... easier than manual SQL. But there are some clunky bits (like needing to know the customer's id to create an order. And I'm lazy so I'd still prefer it easier....

What happens if I delete "mike"? His orders are left in the system... or I have to check and delete them first.

With Active Record associations, we tell Rails that there is a connection between the two models.

We add `dependent: :destroy` to a class so that if we delete a Customer his/her Orders are destroyed also.

```
  class Customer < ActiveRecord::Base
    has_many :orders, dependent: :destroy
  end

  class Order < ActiveRecord::Base
    belongs_to :customer
  end
```

```
  git add .
  git commit -m "Added association between Customer and Order"

  // in console
  reload!

  Customer.first["name"] #=> "marc"
  Customer.first.orders

  Order.count
  Customer.first.destroy # remember I said to use 'destroy' not 'delete'? this is why
 :-)
  Order.count

  Customer.first = "mike" #=> mike is now first, but his id is still 2

  Customer.first.orders.create(value: 100.00)

  Order.first.customer.orders #=> grab all the orders for the customer who owns the f
irst Order

  Order.where(customer_id: 3) #=> find all orders for the Customer with the customer_
id of 3
```

Types of association

has_one

Sets up a one-to-one connection - essentially the same as belongs_to, but implies the foreign-key is on the associated table. It's used when there's only one Thing connected to one OtherThing

- For instance, a Car `has_one` SteeringWheel, and a SteeringWheel `belongs_to` a Car.
- Which side `has_one` and which `belongs_to` can be contencious

```
class SteeringWheel < ActiveRecord::Base
    belongs_to :car
end

class Car < ActiveRecord::Base
    has_one :steering_wheel
end
```

belongs_to

Sets up a connection between one model and another - it's the table upon which the foreign key resides.

- We set up our Order model to `belongs_to` Customer

has_many

Defines a one-to-many relationship. It would generally imply zero-or-more Things associated (but the scope could be restricted).

- We said our Customer `has_many` Orders.

has*and*belongs*to*many

Sets up a many-to-many association. For instance, a Person can drive many Cars, and a Car can be driven by many People. It's a "traditional" join-table design - with two foreign keys (reservations). If you need any other information about the relationship between drives and cars (like the date the driving occurred, or how far the trip was), then another association type might be better....

```
class Driver < ActiveRecord::Base
    has_and_belongs_to_many :cars
end

class Car < ActiveRecord::Base
    has_and_belongs_to_many :drivers
end
```

has_many :through

This sets up a many-to-many relationship "through" another associated model.

For instance, a Doctor and Patient could join through an Appointment model. This would allow a Doctor to have_many Patients, and for a Patient to have many Doctors. The Appointment model will contain foreign keys for both Patients and Doctors.

```
class Doctor < ActiveRecord::Base
  has_many :appointments
  has_many :patients, through: :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :doctor
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, through: :appointments
end
```

EXAMPLE

- If we add a DeliveryAddress model

```
  rails g model DeliveryAddress directions:text
```

```
  class DeliveryAddress < ActiveRecord::Base
    has_one :order
  end
```

- and change Order to belongs_to a DeliveryAddress

```
rails g migration AddDeliveryAddressIdToOrders delivery_address_id:integer
```

```
class Order < ActiveRecord::Base
  belongs_to :customer
  belongs_to :delivery_address
end
```

- we can set our Customer to have_many DeliveryAddresses through Orders

```
class Customer < ActiveRecord::Base
  has_many :orders, dependent: :destroy
  has_many :delivery_addresses, through: :orders
end
```

- migrate and test in the console

```
rake db:migrate
rails c

Order.all.each do |order|
  order.build_delivery_address directions: rand
  order.save
end

Customer.first.delivery_addresses
```

We can now add delivery addresses to the Customer through Order.

```
Customer.first.delivery_addresses.create(directions: "HOME: take a right at the tree")
```

has_one :through

Gives a single-association to another model, but through a third model.

- If we try this, it should fail: `DeliveryAddress.first.customer.name`
- we can reciprocate from DeliveryAddresses to set it to be able to access the Customer through its Order

```
class DeliveryAddress < ActiveRecord::Base
  has_one :order
  has_one :customer, through: :order
end
```

Run `rake db:migrate`. Now you can find Customer using DeliveryAddress:

`DeliveryAddress.first.customer.name`

or

`DeliveryAddress.find_by(id: 1).customer.name`

Conversely, you can now find a DeliveryAddress for a given Customer:

`Customer.find_by(name: "gerry").delivery_addresses`

Then choose a specific one:

`Customer.find_by(name: "gerry").delivery_addresses.find_by(id: 3).directions`

You can also associate a model to itself

```
rails g model Employee name:string manager:references
rake db:migrate
rails c
Employee

Employee.create name: 'Bob'
Employee.create name: 'John'
```

- Rails has set up our foreign-key field, and also an association in Employee.
- In our `schema.rb` we now have an `add_index` which is a data structure that improves the speed of queries and operations in a table.

We don't have a Manager model (though maybe in our OO world we might think to use inheritance to differentiate), but in this case, our managers are already in our DB as Employees themselves.

```
class Employee < ActiveRecord::Base
  belongs_to :manager, class_name: "Employee"
  has_many :subordinates, class_name: "Employee", foreign_key: "manager_id"
end

reload!

bob = Employee.first
john = Employee.last
bob.manager = john
bob.save
Employee.first.manager #=> 'John'
Employee.last.manager #=> nil

bob #=> Should now have a manager_id foreign key for john
john #=> manager_id is nil
bob.manager.name #=> 'John'
```

Be careful of circular references - don't associate people (even indirectly) to themselves! :-)

Query Caching - benefits of associations

```
c = Customer.first
c.orders # note the query that's run
c.orders # what query runs this time?
```

- http://guides.rubyonrails.org/v3.2.19/association_basics.html
- Now we're associating properly, it increase the power of our AR queries... we'll revisit this next week

Bonus - adding index to the schema with migrations

Let's pretend we have Songwriter, Song and Artist models. A songwriter `has_many` songs and a song `belongs_to` one songwriter. A song `has_many` many artists and an artist `belongs_to` one song.

I know a song can have more than one writer, but let's pretend for now.

What kind of relationships are these? Two 'belongs to and has many' relationships.

Let's build the Models:

```
rails g model Artist name
rails g model Song title length:float
rails g model Songwriter name

rake db:migrate
```

Let's set up the relationships in the models:

```


class Songwriter < ActiveRecord::Base
  has_many :songs
end

class Artist < ActiveRecord::Base
  belongs_to :song
end

class Song < ActiveRecord::Base
  belongs_to :songwriter
  has_many :artists
end
```