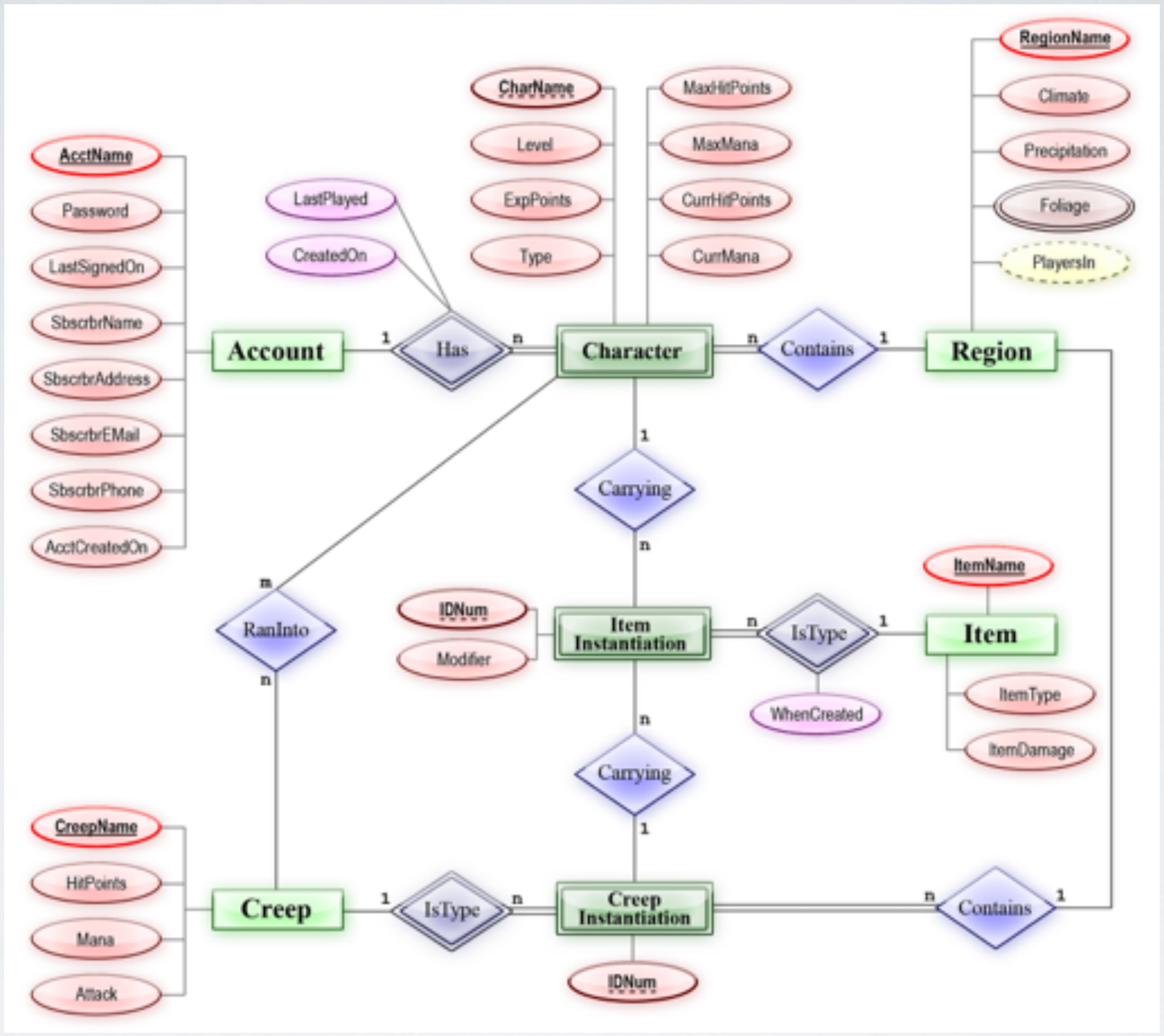


MODEL ASSOCIATIONS

OBJECTIVE

- You will learn the core basics of model validations and associations in Ruby on Rails.
- The goal for today is to understand this conceptually and implement it together. Next week you are on your own.
- This is a complex and sometimes tricky issue. Let’s not rush this.

MODEL VALIDATIONS

- A model validation is to ensure data integrity before it is saved to your database.
- They are defined on the model's class (So app/models/_____.rb)
- Rails comes with many built-in validations, however you can add your own as well. Although, you likely won't need to do this for simple projects initially.

MODEL VALIDATIONS

- Model validations are ran before anything is created, or updated in your database.
- Think of it this way:
  - A user types in an email that is not a valid format when registering an account.
  - When you go to save the new user in your database, your model will validate the format of the email passed in.
  - If the validation fails, ActiveRecord will not allow the record to be persisted (inserted) into the database and will report the errors.

MORE REAL WORLD EXAMPLES

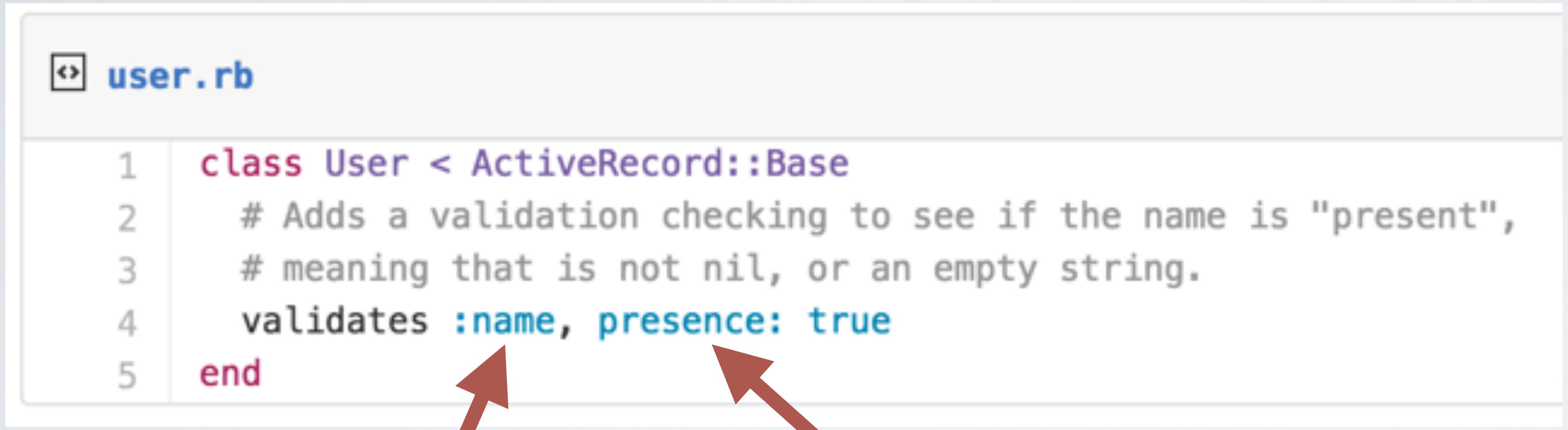
- Checking to see if a value is "numeric"
- Checking the length of something (password for example)
- Confirmation (making sure someone types in the same thing in 2 fields correctly, again… passwords).
- http://guides.rubyonrails.org/active_record_validations.html has all of the built-in validations

SO WHERE DO WE ADD VALIDATIONS?

- Great question!
- Let’s take a look at a model called “User” and add a validation for checking the length of the name!
- In your Photo app, generate a User model with

PRESENCE VALIDATION ON A MODEL EXAMPLE

GIVEN A MODEL “USER” AT APP/MODELS/USER.RB



user.rb

```ruby
class User < ActiveRecord::Base
  # Adds a validation checking to see if the name is "present",
  # meaning that is not nil, or an empty string.
  validates :name, presence: true
end
```

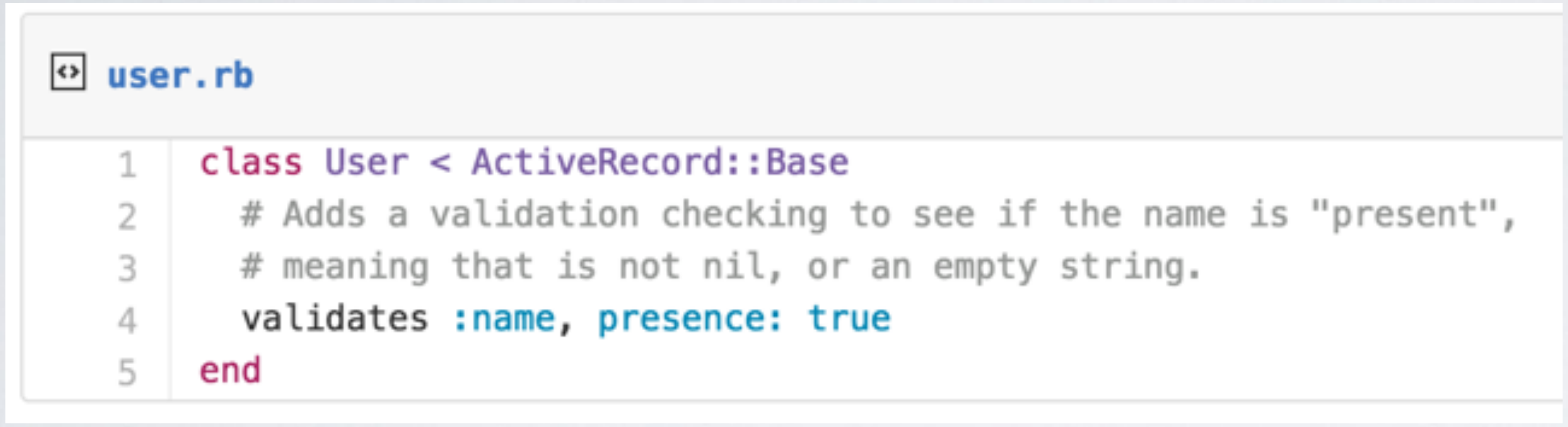
The attribute name
that we want to validate

The validation we want to
perform on this attribute
before saves / updates

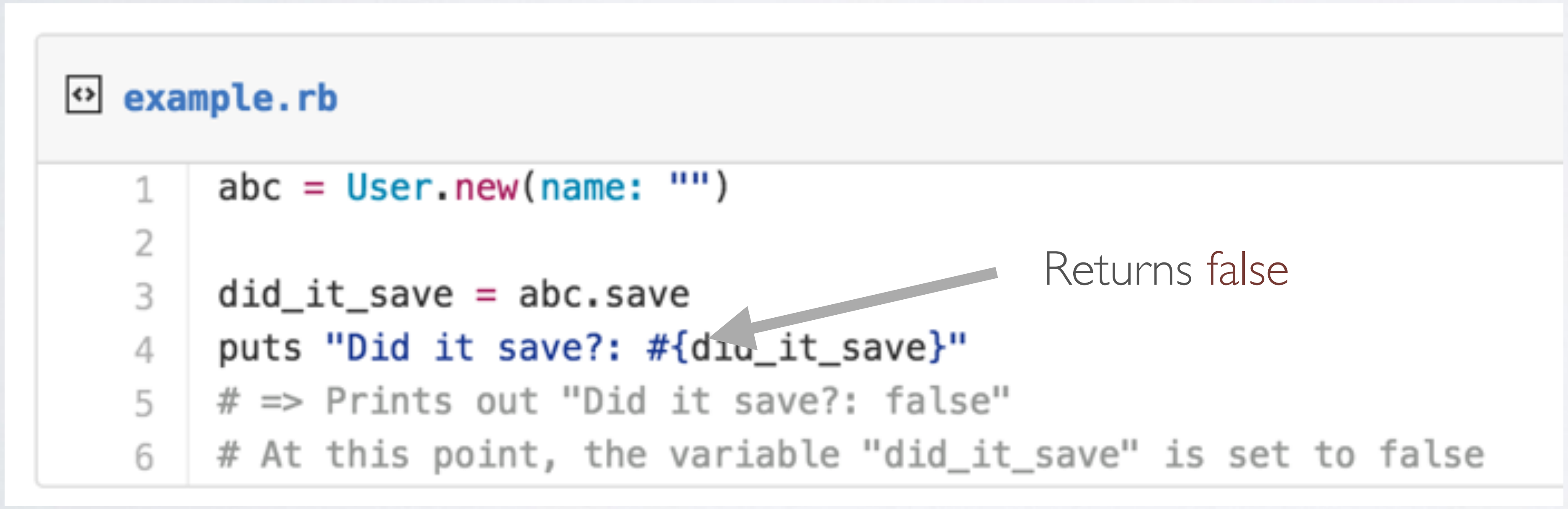
RESULTS

- So when we instantiate a version of this model with a blank name attribute (nil or an empty string “”)
- Attempt to save it with “.save”
- The validations will fail and nothing will be saved to the database.
- Furthermore, “.save” will return *false*

RESULTS


user.rb

```ruby
class User < ActiveRecord::Base
  # Adds a validation checking to see if the name is "present",
  # meaning that is not nil, or an empty string.
  validates :name, presence: true
end
```


example.rb

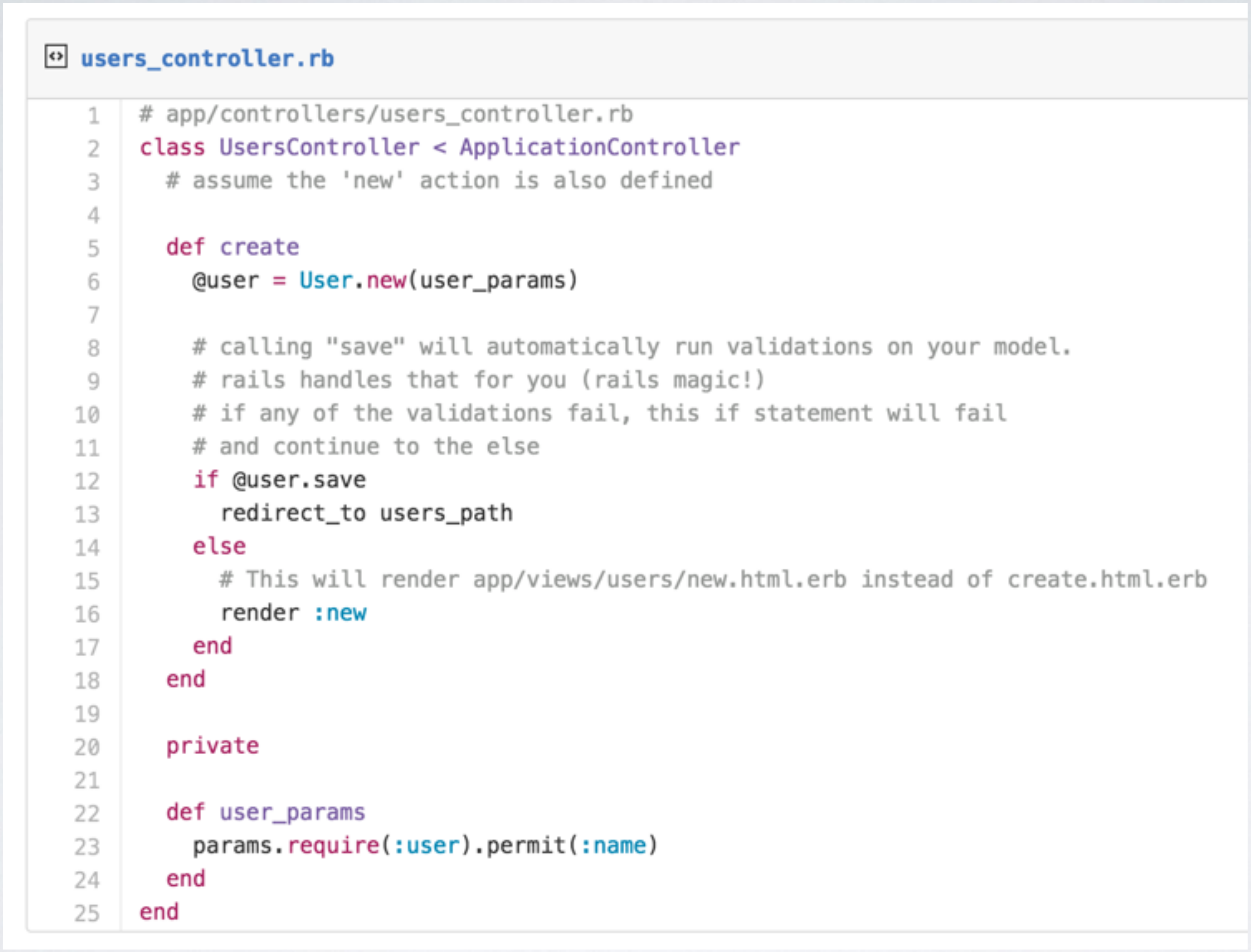
```ruby
abc = User.new(name: "")

did_it_save = abc.save
puts "Did it save?: #{did_it_save}"
# => Prints out "Did it save?: false"
# At this point, the variable "did_it_save" is set to false
```

Returns false

COMMON USAGE IN A CONTROLLER

- Model validations are commonly "used" from the controller.
- Remember, the controller accepts data from forms and saves the data using a model.
- So when you assign the validations on a model, the controller doesn't need to define them, it simply needs to call "save" on the model instance and check to see if it returned true or false.
- Code on next slide…


users_controller.rb

```ruby
# app/controllers/users_controller.rb
class UsersController < ApplicationController
  # assume the 'new' action is also defined

  def create
    @user = User.new(user_params)

    # calling "save" will automatically run validations on your model.
    # rails handles that for you (rails magic!)
    # if any of the validations fail, this if statement will fail
    # and continue to the else
    if @user.save
      redirect_to users_path
    else
      # This will render app/views/users/new.html.erb instead of create.html.erb
      render :new
    end
  end

  private

  def user_params
    params.require(:user).permit(:name)
  end
end
```

LAB

- Go to your Flickr app from Tuesday.
- Skim http://guides.rubyonrails.org/active_record_validations.html to find how you might make sure that you can't create two Users with one email, or two photos with the same url. Also, make sure that captions are no more than 140 characters.
- Provide the appropriate flash message to tell the user whether the create/update worked.
- If you finish, see if you can find make your app ensure that a user's age is a number, and that the number is under 100.

MODEL ASSOCIATIONS

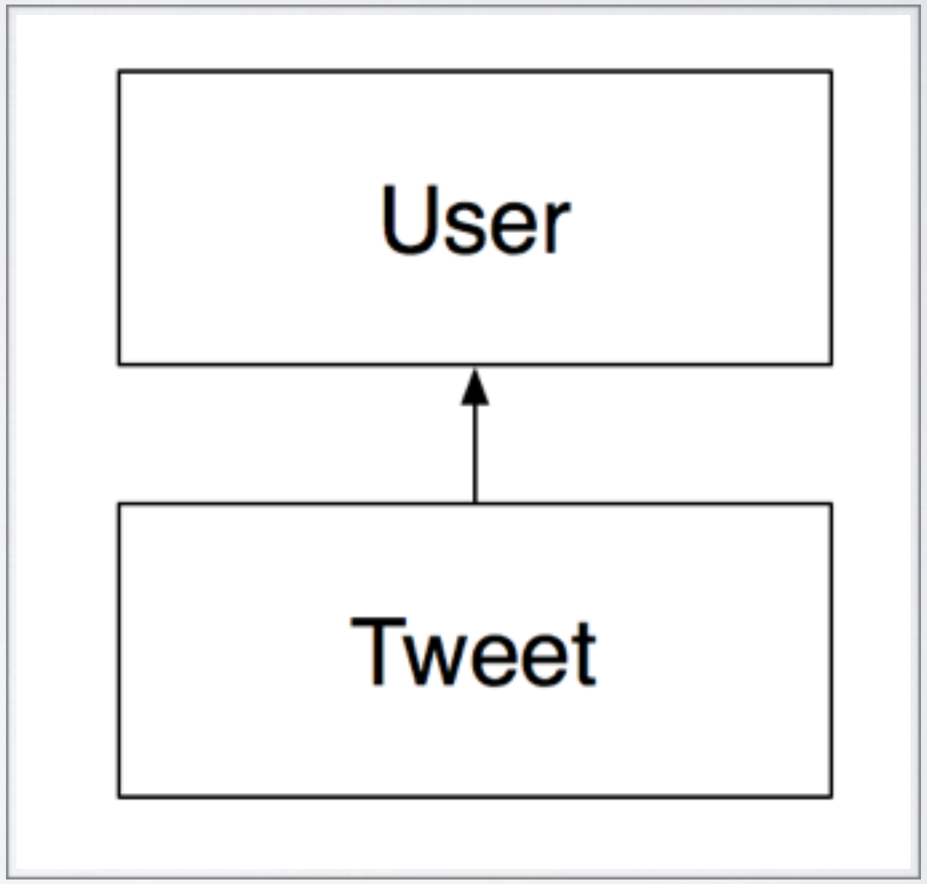
- Model Associations are essential to making robust applications.
- An association is a connection between 2 different models.
- For example, An author has many books that they have written. Inversely, a book belongs to an author.

REAL WORLD EXAMPLES

- When a User "tweets", let's assume they are creating a new Tweet with a Tweet model like so:
  - `Tweet.create(text: "Just saw j.biebs oh em gee!!!")`
- This creates a tweet, but it's "free floating" right now.
- What I mean is, the tweet is created, but we have no idea once it's created *who* tweeted it.
- So how do you associate (or link) data between models together?

REAL WORLD EXAMPLES

- Fictional Twitter Models
  - User
  - Tweet
- A User *has many* Tweets
- A Tweet *belongs to* a User



LINKING DATA TOGETHER W/ ACTIVERECORD

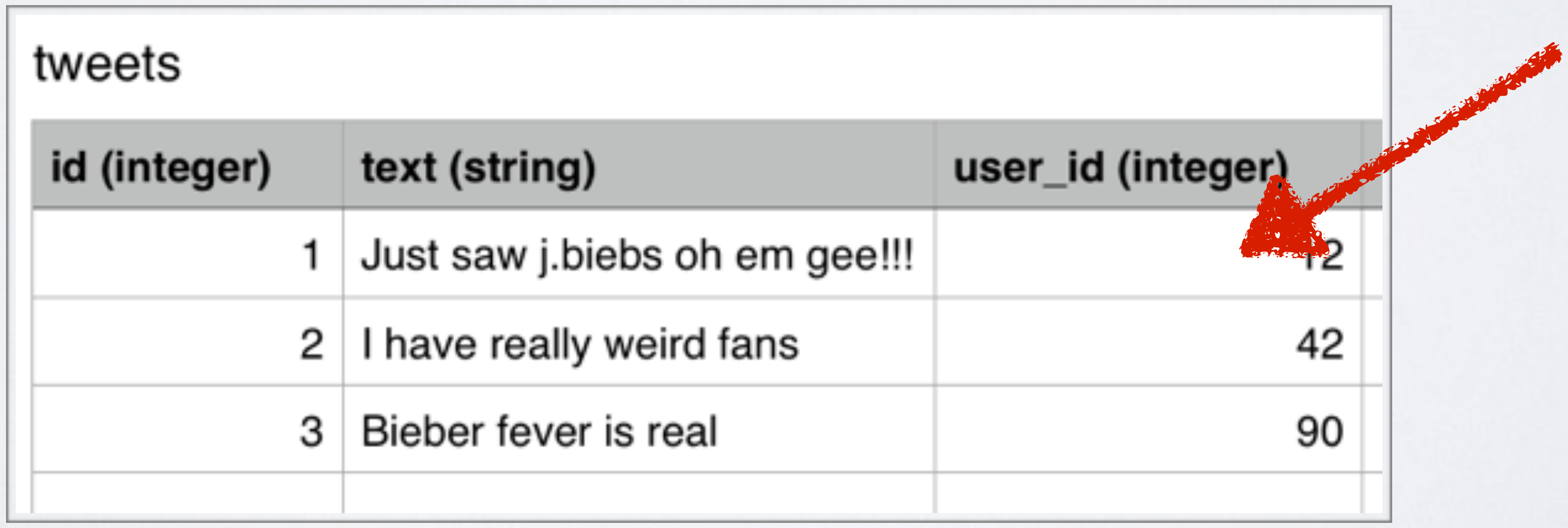
- ActiveRecord (The part of Rails that communicates with the database) allows us to define associations on our models to *other* models.

- This means you must have 2 (or more) models in order to associate them together.

- You must define the associations in the model classes (in app/models/)

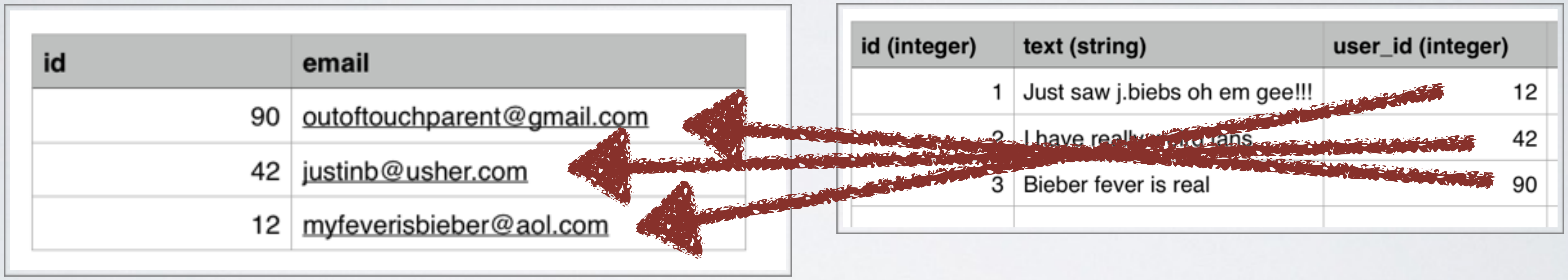
DATABASE CHANGES

- For data to be linked together, you must create a field on model's database table, indicating what it belongs to.
- Has many associations do NOT need an additional column on the model.
- Only the table indicating a "belongs_to" association needs an extra column.


tweets

id (integer)	text (string)	user_id (integer)
1	Just saw j.biebs oh em gee!!!	12
2	I have really weird fans	42
3	Bieber fever is real	90

FIND THE EMAIL!



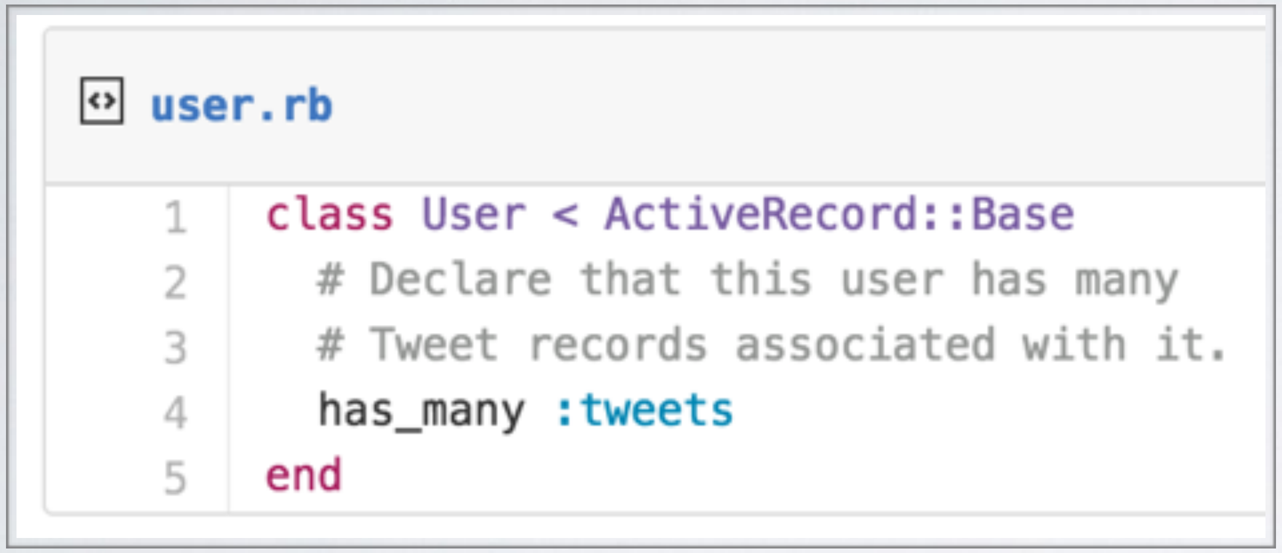
User's

id	email
90	outoftouchparent@gmail.com
42	justinb@usher.com
12	myfeverisbieber@aol.com

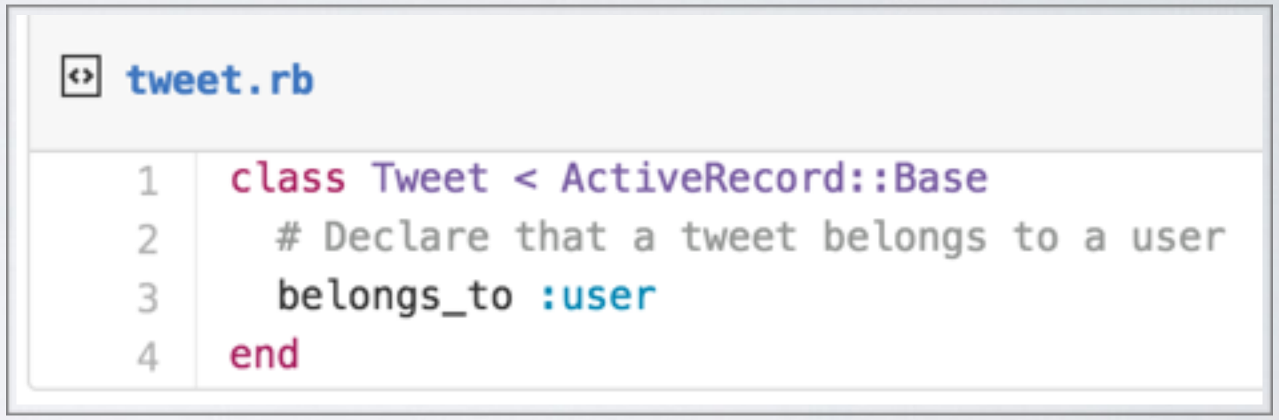
Tweet's

id (integer)	text (string)	user_id (integer)
1	Just saw j.biebs oh em gee!!!	12
2	I have really weird fans	42
3	Bieber fever is real	90

CODE!


user.rb

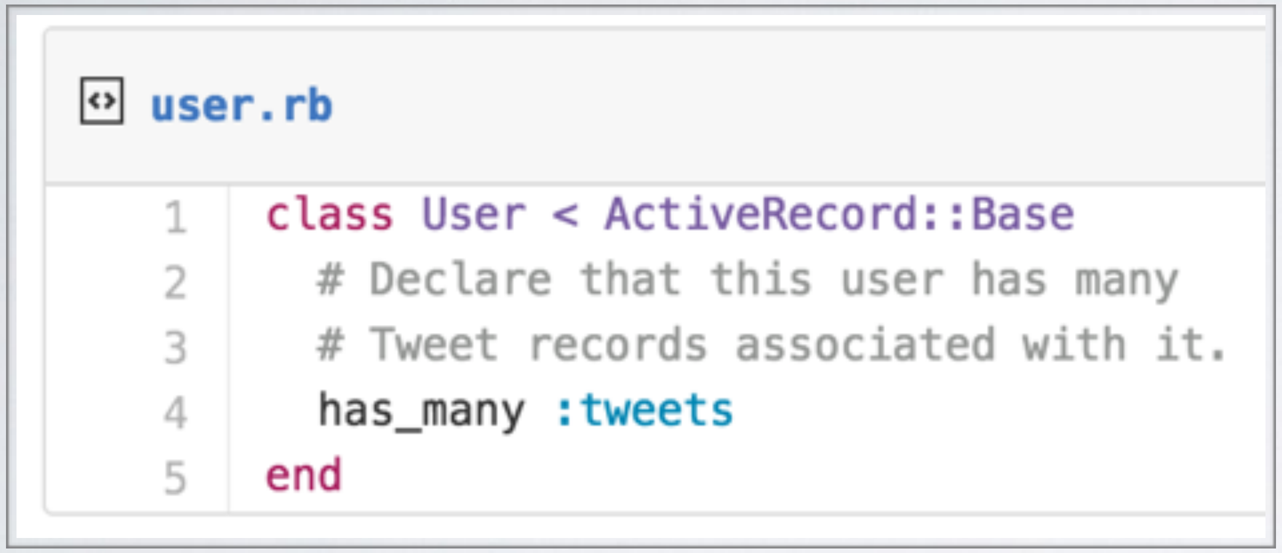
```ruby
class User < ActiveRecord::Base
  # Declare that this user has many
  # Tweet records associated with it.
  has_many :tweets
end
```


tweet.rb

```ruby
class Tweet < ActiveRecord::Base
  # Declare that a tweet belongs to a user
  belongs_to :user
end
```

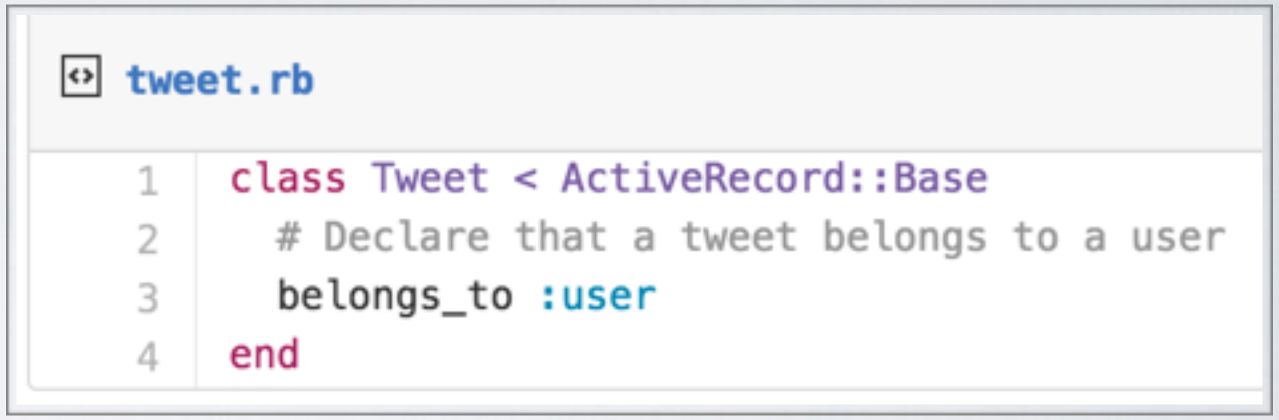
- The User model is declaring that a single user has many tweets in another model call "Tweet".
- The association must be plural, Rails will figure out ":tweets" is the Tweet model automatically for you. (magic!)

CODE!



user.rb

```ruby
class User < ActiveRecord::Base
  # Declare that this user has many
  # Tweet records associated with it.
  has_many :tweets
end
```



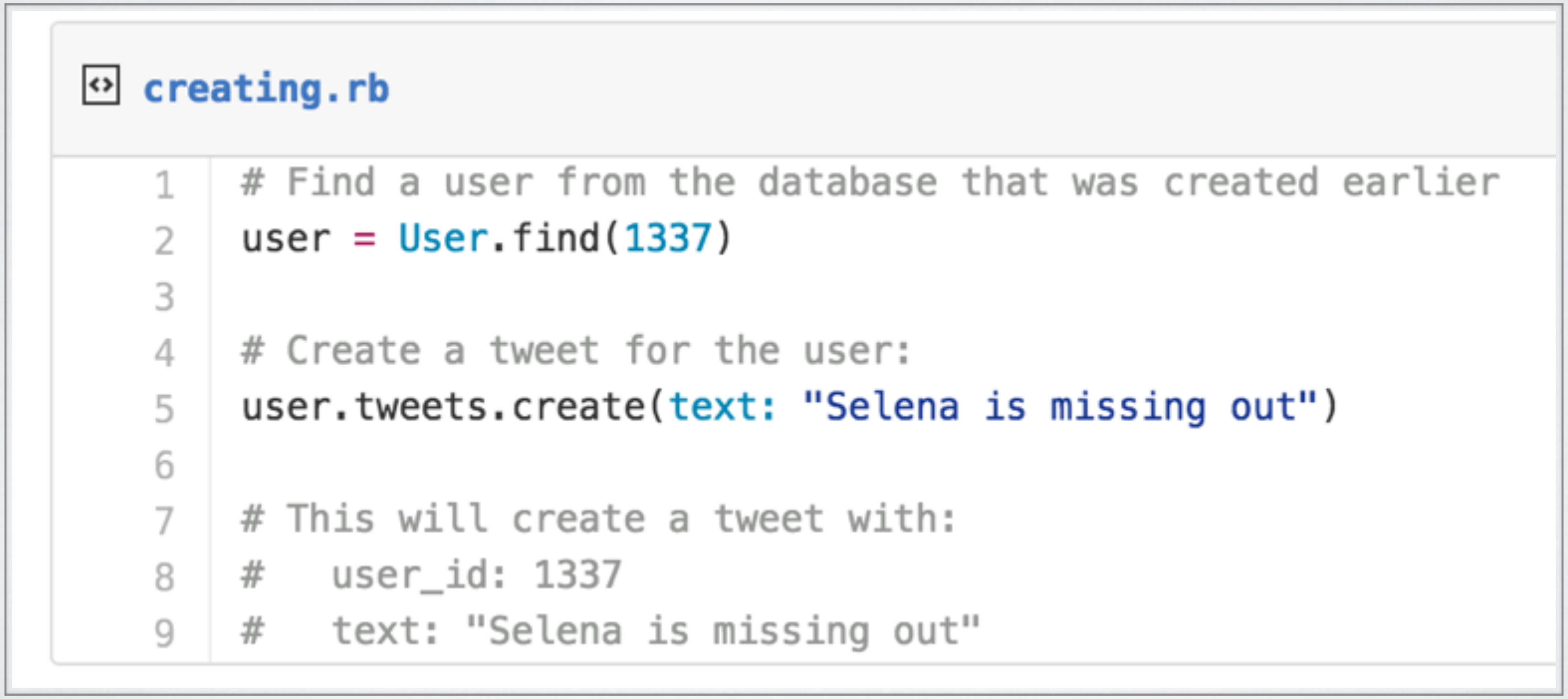
tweet.rb

```ruby
class Tweet < ActiveRecord::Base
  # Declare that a tweet belongs to a user
  belongs_to :user
end
```

- The User model is declaring that a single user has many tweets in anot
- The association must ut ":tweets" is the Twee u.

CREATING A TWEET FOR A USER

- Instead of "Tweet.create()", we'd now do something like this: (most of the time)



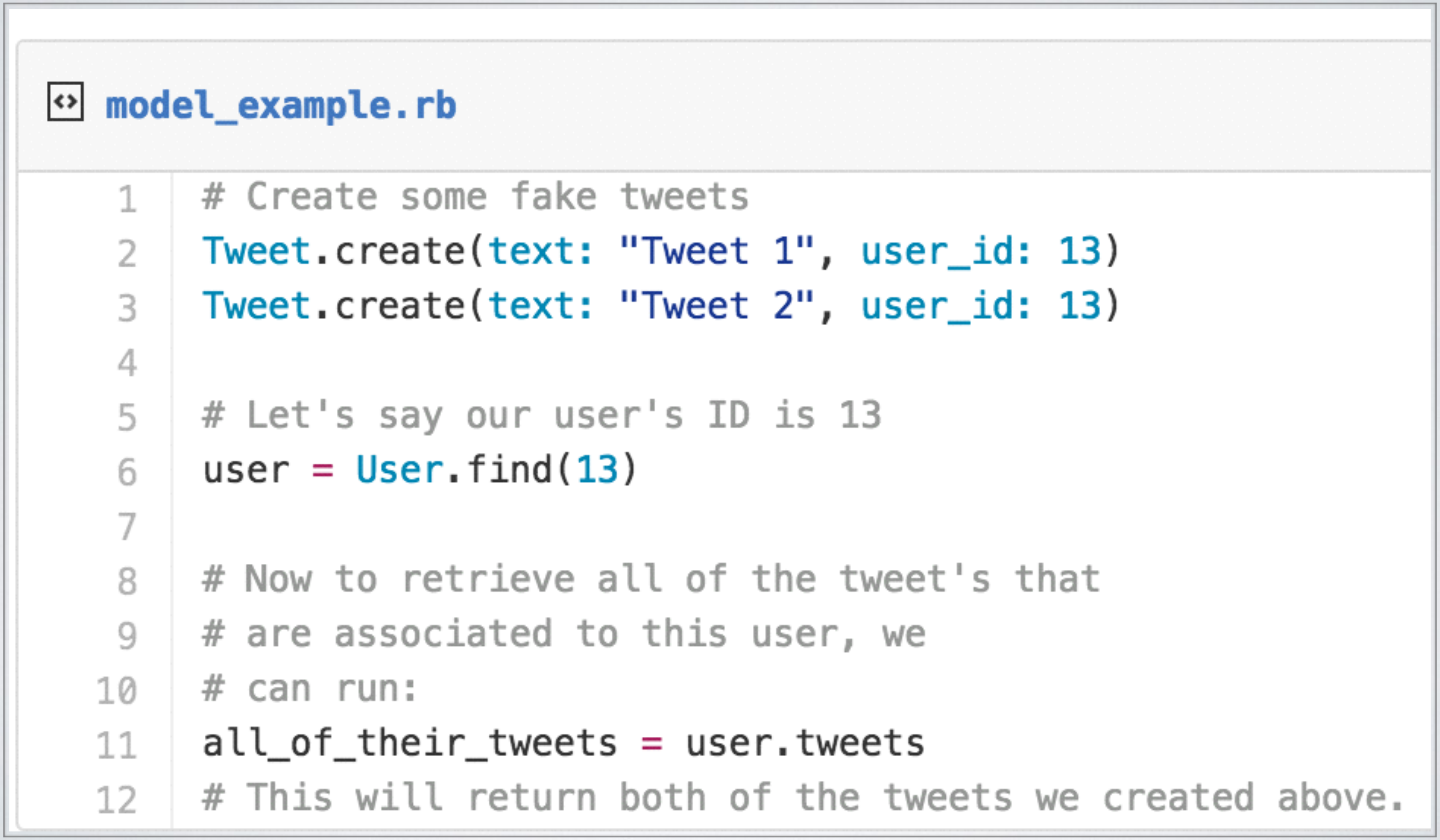
creating.rb

```ruby
# Find a user from the database that was created earlier
user = User.find(1337)

# Create a tweet for the user:
user.tweets.create(text: "Selena is missing out")

# This will create a tweet with:
#   user_id: 1337
#   text: "Selena is missing out"
```

WHAT DOES THIS GET YOU?


model_example.rb

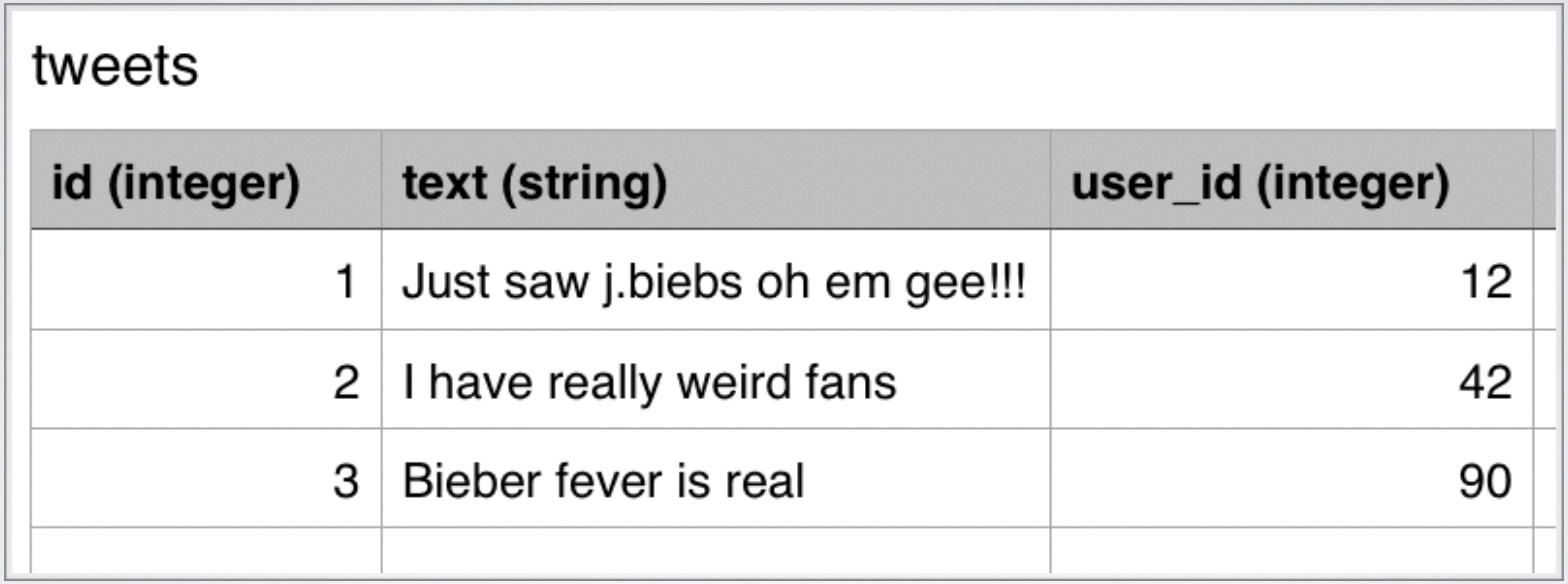
```ruby
# Create some fake tweets
Tweet.create(text: "Tweet 1", user_id: 13)
Tweet.create(text: "Tweet 2", user_id: 13)

# Let's say our user's ID is 13
user = User.find(13)

# Now to retrieve all of the tweet's that
# are associated to this user, we
# can run:
all_of_their_tweets = user.tweets
# This will return both of the tweets we created above.
```

LOGICALLY SPEAKING…

- If we know a User's ID (which we may or may not get from params — spoiler alert), we can fetch all of their tweets easily!
- We can tell our Tweet model, "Give me all of the tweets where the user_id column is equal to X" Where **X** is the User ID.
- On the opposite side, where we only have a tweet, with no user, we can say "Give me the user with the ID of X" where X is the user_id on the Tweet.



tweets

id (integer)	text (string)	user_id (integer)
1	Just saw j.biebs oh em gee!!!	12
2	I have really weird fans	42
3	Bieber fever is real	90

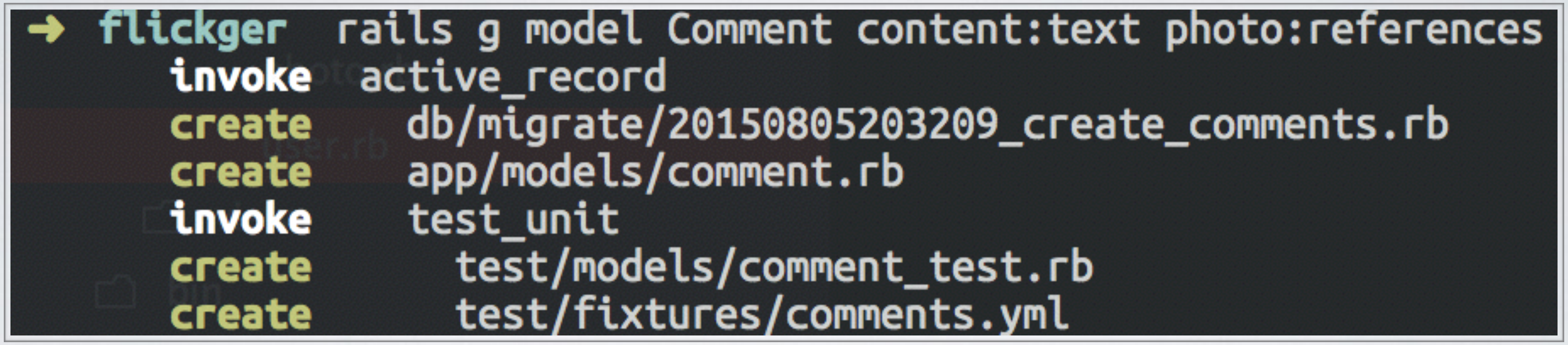
GENERATING A MODEL W/ ASSOCIATIONS

- It is possible to define an association in a rails model while generating it. And it’s easy!
- Just like defining the columns type (name:string), you can define a relationship with the underscore version of the associated model.
  - So, if we’re associating to a model “State”, we can do:
  - `rails g model City state:references`

GENERATING A MODEL W/ ASSOCIATIONS

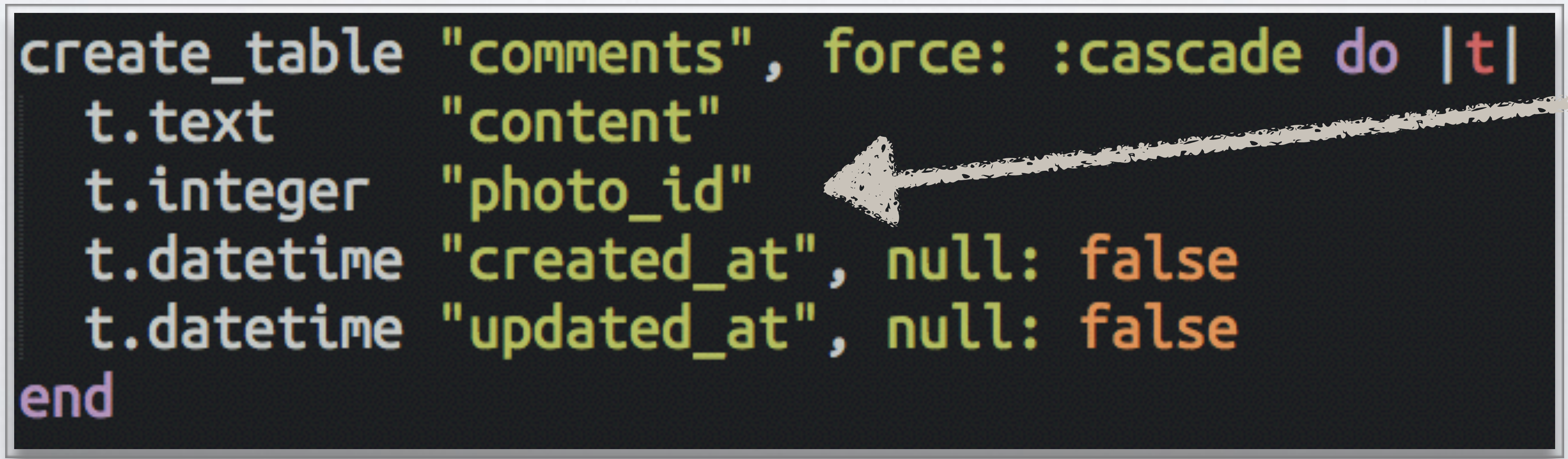
- `rails g model City state:references`
- This will generate a new model, City, with a column in our database "state_id".
- The "state_id" column is how a City will be linked to a State record in another database table.
- Remember, *all* things have ID's in our database by default. Rails / Your database handles the uniqueness of them automatically. MAGIC.
- If you delete a record, that ID is *gone.* IDs are not just simple ordinal numbers.

EXAMPLE



```
➜ flickger  rails g model Comment content:text photo:references
      invoke  active_record
      create    db/migrate/20150805203209_create_comments.rb
      create    app/models/comment.rb
      invoke    test_unit
      create      test/models/comment_test.rb
      create      test/fixtures/comments.yml
```

After rake db:migrate, this would create a table that looks like:



```
create_table "comments", force: :cascade do |t|
  t.text     "content"
  t.integer  "photo_id"
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
end
```

HAS ONE

- Sometimes you need to associate one record to one record.
- On some records, rather than has_many, you use has_one for this sort of relationship.

CODE ALONG

- Creating a new rails app called “general_assembly”
- Create a model Course
- Create a model called Student that belongs to a Course
- Play around with it in Rails Console.
- If you finish early, create an Instructor model. What sort of relationship to the other models will that have?
- We will reconvene and do this together in 15 minutes.

BREAK

CODE ALONG

- Let’s perfect that GA app.

GA CODE ALONG

- Logically, the instructor has students, and students have an instructor
- We don't need much more on the model to make this work.

GA CODE ALONG

- Active Record models can be related *through* another model.
- Students and Instructors already have something in common: the course.

GA CODE ALONG

- Add this to Student:
  - `has_one :instructor, through: :course`
- Add this to Instructor:
  - `has_many :students, through: :course`

DEPENDENT DESTROY

- Let’s just pretend that GA flushes your records the minute you finish the class. It would be a pain to do this one at a time.
- First destroy the course.
- Then destroy the instructor.
- Then loop over each student and delete them.
- BORING! We can structure this so students and instructors are dependent on course.

DEPENDENT DESTROY

- has_many and has_one relationships have an option called **`dependent: :destroy`**
- Put this after the has_many relationship on Course.
- Now delete the course in the terminal. See what happens? Keep in mind that this doesn't always make much sense.

LAB (Let's A take this to the Browser)

- So far we have been doing this from the comfort of the Rails Console. This isn't how people want to interact with a web app.

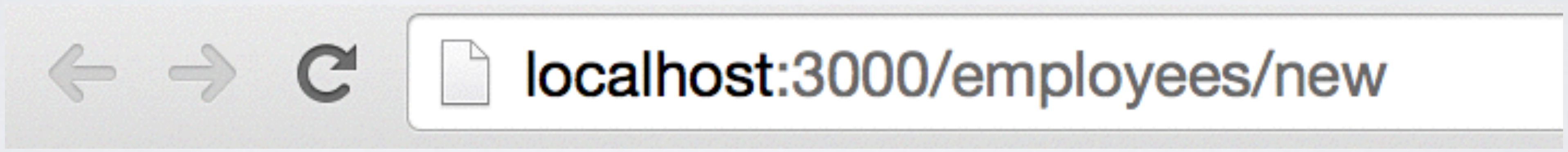
- First, we need routes. Go ahead and define resourceful routes for courses.

- Then let's code up an *index, and show* action for courses. We can just use the Course object we already made.

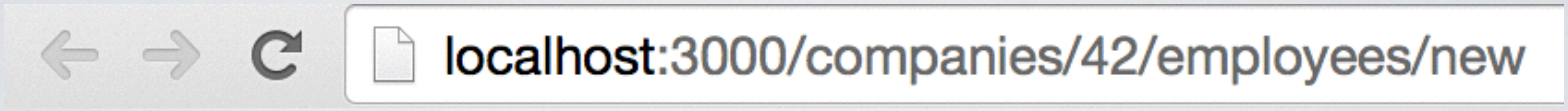
NESTED RESOURCES AND ASSOCIATIONS

- Our routes commonly will indicate what our associations are.
- They don't *define* our associations, but they can give a glimpse of what they are under the hood.

- Let’s assume a Course has many Employees schema.
- To add a company, our route commonly will look like: `resources :companies`
- But what about our employees routes? It’s normal to think, well “resources :employees”
- However, when we do this, our route can’t indicate what company a employee is being added to.

- “our route can’t indicate what company a employee is being added to.”
- Let’s assume:
  - 
    localhost:3000/employees/new
- When we click “submit” how will our application know what company this employee belongs to?
- Hint: It can’t

THE SOLUTION

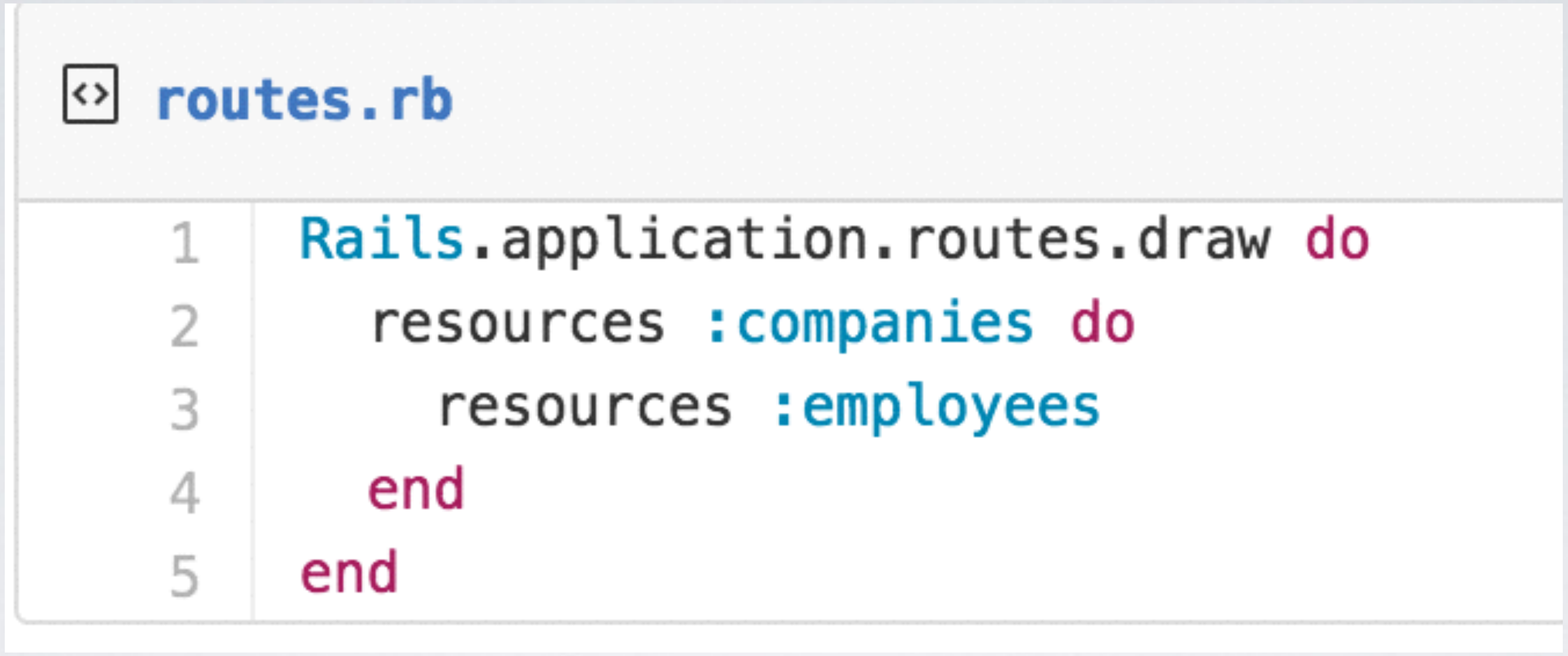

localhost:3000/companies/42/employees/new

- This URL contains the company ID that we'd like to add an employee for.

- This is accomplished by using a "nested resource"

- It will still route to our employees controller and 'new' action, but it will include a parameter in our "params" hash indicating which company we'd like to create the employee for.

You nest routes inside of others by using a block (do .. end)

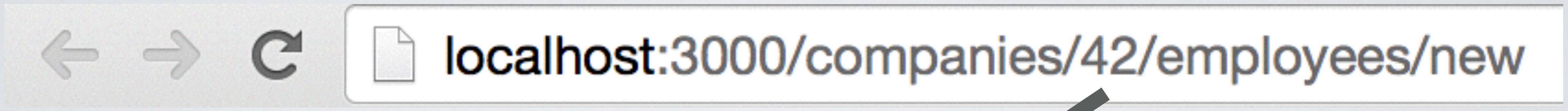

routes.rb

```ruby
Rails.application.routes.draw do
  resources :companies do
    resources :employees
  end
end
```

This produces routes that include the company ID in the URL

```
GET    /companies/:company_id/employees(.:format)          employees#index
POST   /companies/:company_id/employees(.:format)          employees#create
GET    /companies/:company_id/employees/new(.:format)      employees#new
GET    /companies/:company_id/employees/:id/edit(.:format) employees#edit
GET    /companies/:company_id/employees/:id(.:format)      employees#show
PATCH  /companies/:company_id/employees/:id(.:format)      employees#update
PUT    /companies/:company_id/employees/:id(.:format)      employees#update
DELETE /companies/:company_id/employees/:id(.:format)      employees#destroy
GET    /companies(.:format)                                companies#index
... plus all of the other resources :companies routes
```

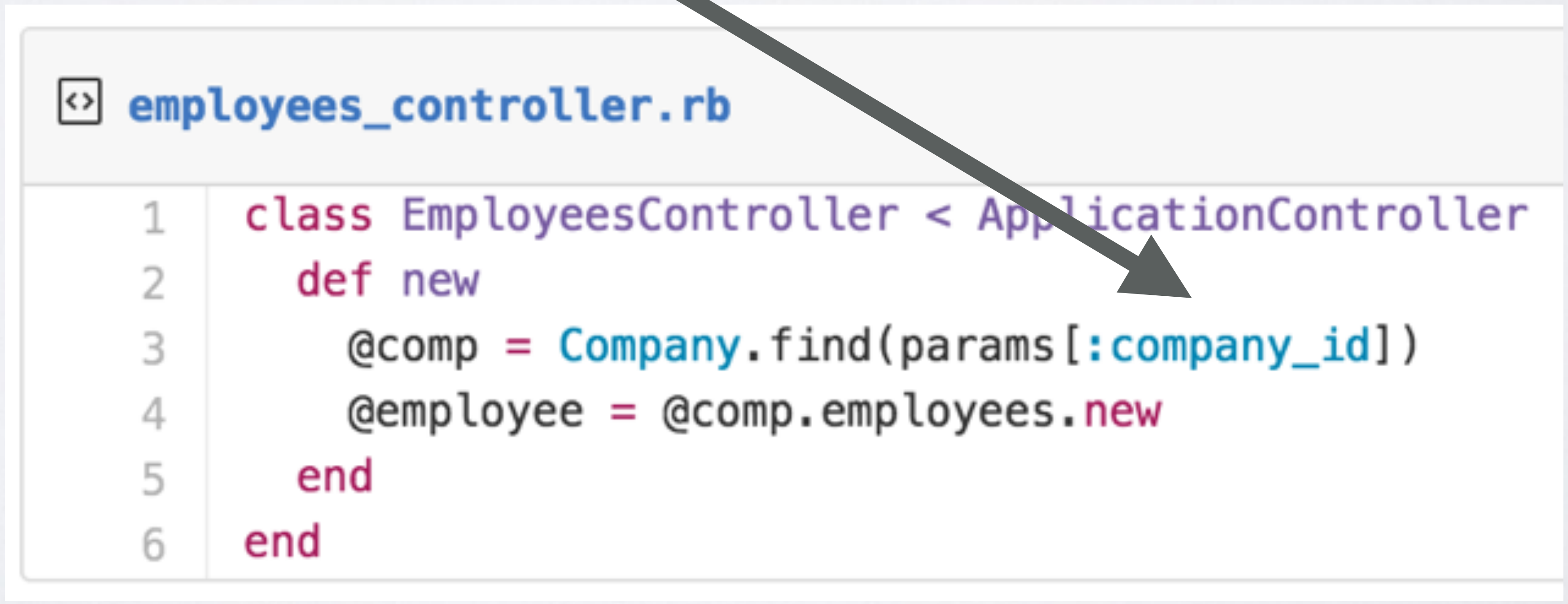
SO IN OUR CONTROLLER



localhost:3000/companies/42/employees/new

rake routes

```
GET    /companies/:company_id/employees/new(.:format)      employees#new
```

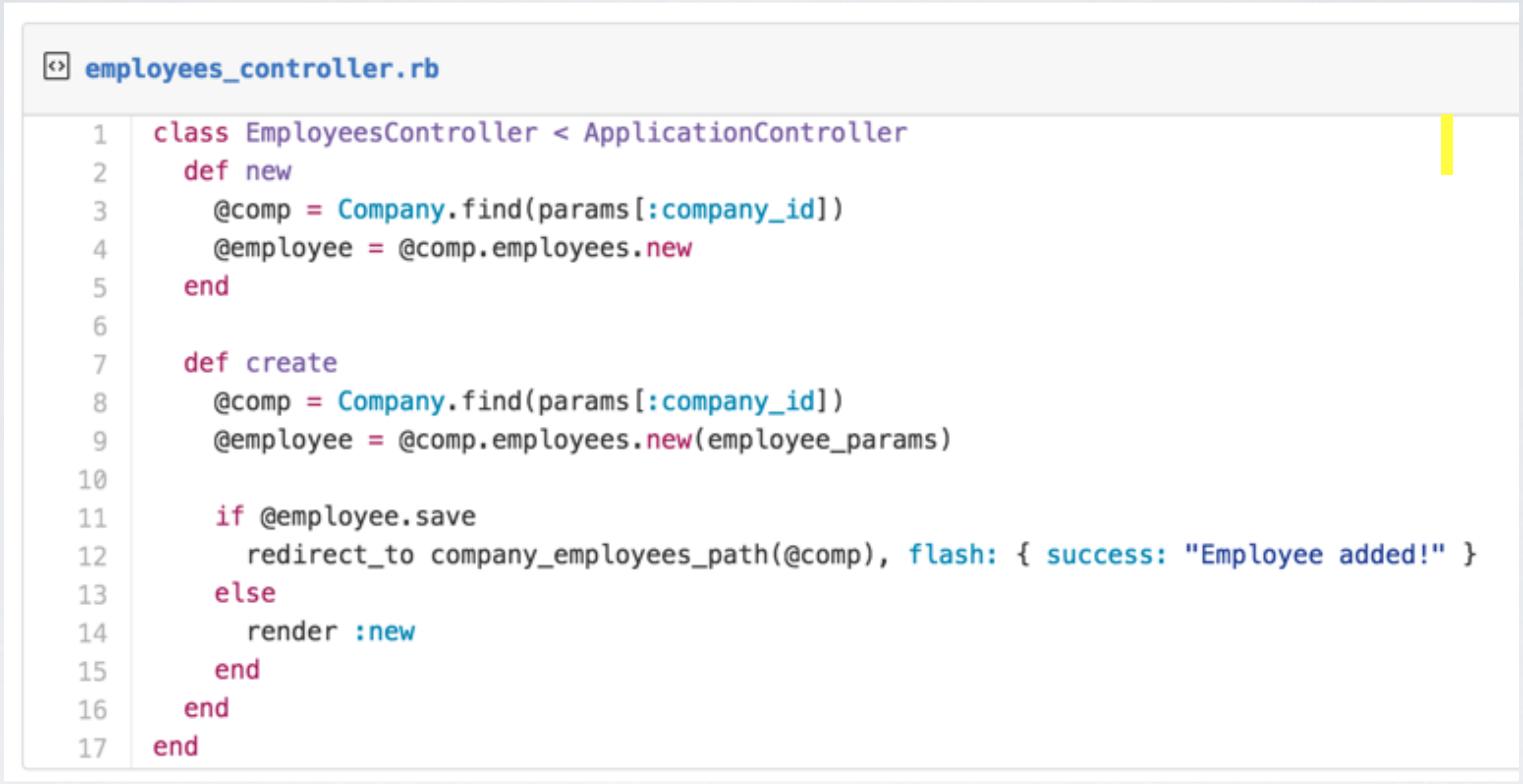


employees_controller.rb

```ruby
class EmployeesController < ApplicationController
  def new
    @comp = Company.find(params[:company_id])
    @employee = @comp.employees.new
  end
end
```

THIS IS AWESOME BECAUSE…

- Now we can create an employee with a company id, effectively associating them to a company with that same ID.
- We do the same sort of thing when creating our employee in our database…



employees_controller.rb

```ruby
class EmployeesController < ApplicationController
  def new
    @comp = Company.find(params[:company_id])
    @employee = @comp.employees.new
  end

  def create
    @comp = Company.find(params[:company_id])
    @employee = @comp.employees.new(employee_params)

    if @employee.save
      redirect_to company_employees_path(@comp), flash: { success: "Employee added!" }
    else
      render :new
    end
  end
end
```

CODE ALONG

- Adding a nested resource controller
- We should be able to add a student for a course, and view a list of students for the course.

REST OF CLASS

- (Mostly) unguided reps
- I want you all to practice ~~struggling~~ debugging. We are here to answer questions, but I want you to avail yourselves to the wealth of information on the web and in this classroom.

YOUR SPEC

- Root route should display all courses, clickable to show individual course

- Course show should have a link to a form to create a new student. The form should notify whether it worked.

- Students must have a name and the grade must exist and be numeric.

- I should be able to view all students in a course with the course's instructor and name at the top. Each student name should link to a show page which includes a delete student and edit student link. Provide the necessary controller actions and views.

- If you finish, flesh out the CRUD actions and routes for the three models we have in this app.