



GET YOUR WD-40

Cuz we got some CRUD in here.

TODAY

- Get some more reps with the whole CRUD thing so we just grok the crap out of it from now on
 - Edit our models
 - Put some more validations on models
 - Learn just a tiny bit about associating models.
-

RESOURCEFUL ROUTING

- Remember, routing is the beginning of all requests to your rails application.
 - Route to controller, controller to model, controller to view, rendered HTML back to the requesting user's browser.
-

RESOURCEFUL ROUTING

- We've learned:
 - Displaying a list of items from a database
 - Displaying a form and accepting input from it to create items in our database.
 - Displaying a “detail” page.
 - What's left:
 - Displaying an “edit” form and updating the data in the database.
-

REVISITING “CRUD”

- CRUD stands for:
 - Create - Displaying a form and creating data from user input
 - Read - Displaying information about a single or collection of models
 - Update - Update attributes on a particular model in the database.
 - Delete - Delete a single model from the database
-

RAILS ROUTES MIMIC THE CRUD

- When you define a route with “resources :photos” for example, you get these routes:

```
→ resources rake routes
  Prefix Verb   URI Pattern               Controller#Action
  photos GET    /photos(.:format)         photos#index
          POST   /photos(.:format)         photos#create
  new_photo GET    /photos/new(.:format)     photos#new
  edit_photo GET    /photos/:id/edit(.:format) photos#edit
  photo GET    /photos/:id(.:format)     photos#show
          PATCH /photos/:id(.:format)     photos#update
          PUT    /photos/:id(.:format)     photos#update
          DELETE /photos/:id(.:format)     photos#destroy
```

CRUD IN THE ROUTES

Create

POST	/photos(:format)	photos#create
GET	/photos/new(:format)	photos#new

Read

GET	/photos(:format)	photos#index
GET	/photos/:id(:format)	photos#show

Update

GET	/photos/:id/edit(:format)	photos#edit
PATCH	/photos/:id(:format)	photos#update
PUT	/photos/:id(:format)	photos#update

Delete

DELETE	/photos/:id(:format)	photos#destroy
--------	----------------------	----------------

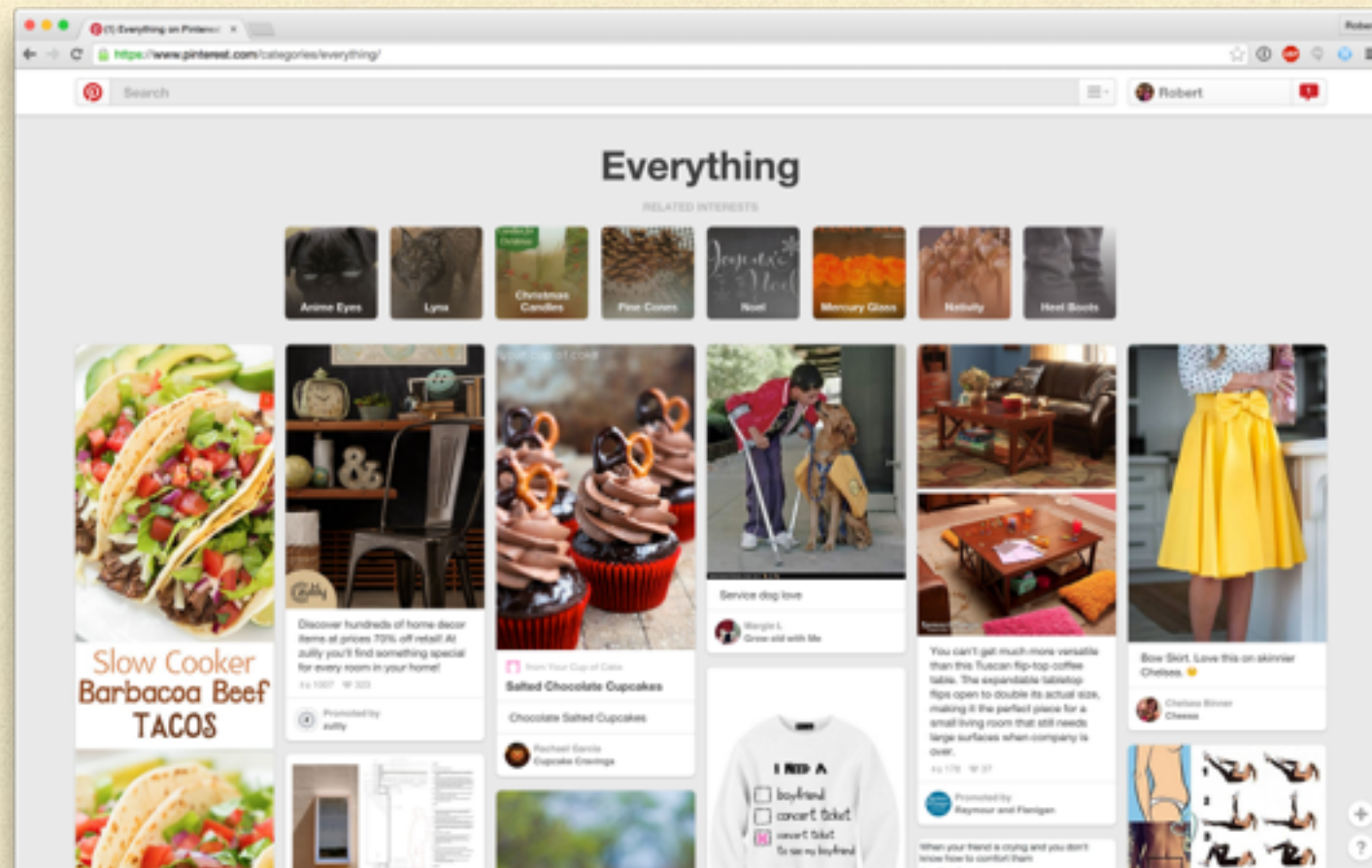
```
→ resources rake routes
  Prefix Verb   URI Pattern          Controller#Action
  photos GET    /photos(:format)     photos#index
             POST   /photos(:format)     photos#create
  new_photo GET    /photos/new(:format) photos#new
  edit_photo GET    /photos/:id/edit(:format) photos#edit
  photo GET    /photos/:id(:format) photos#show
             PATCH  /photos/:id(:format) photos#update
             PUT    /photos/:id(:format) photos#update
             DELETE /photos/:id(:format) photos#destroy
```

THE POINT OF ACTIONS

- Here is a list of controller actions and their respective part in “CRUD”
 - new, create - Create
 - show, index - Read
 - edit, update - Update
 - delete - Delete (or destroy)
-

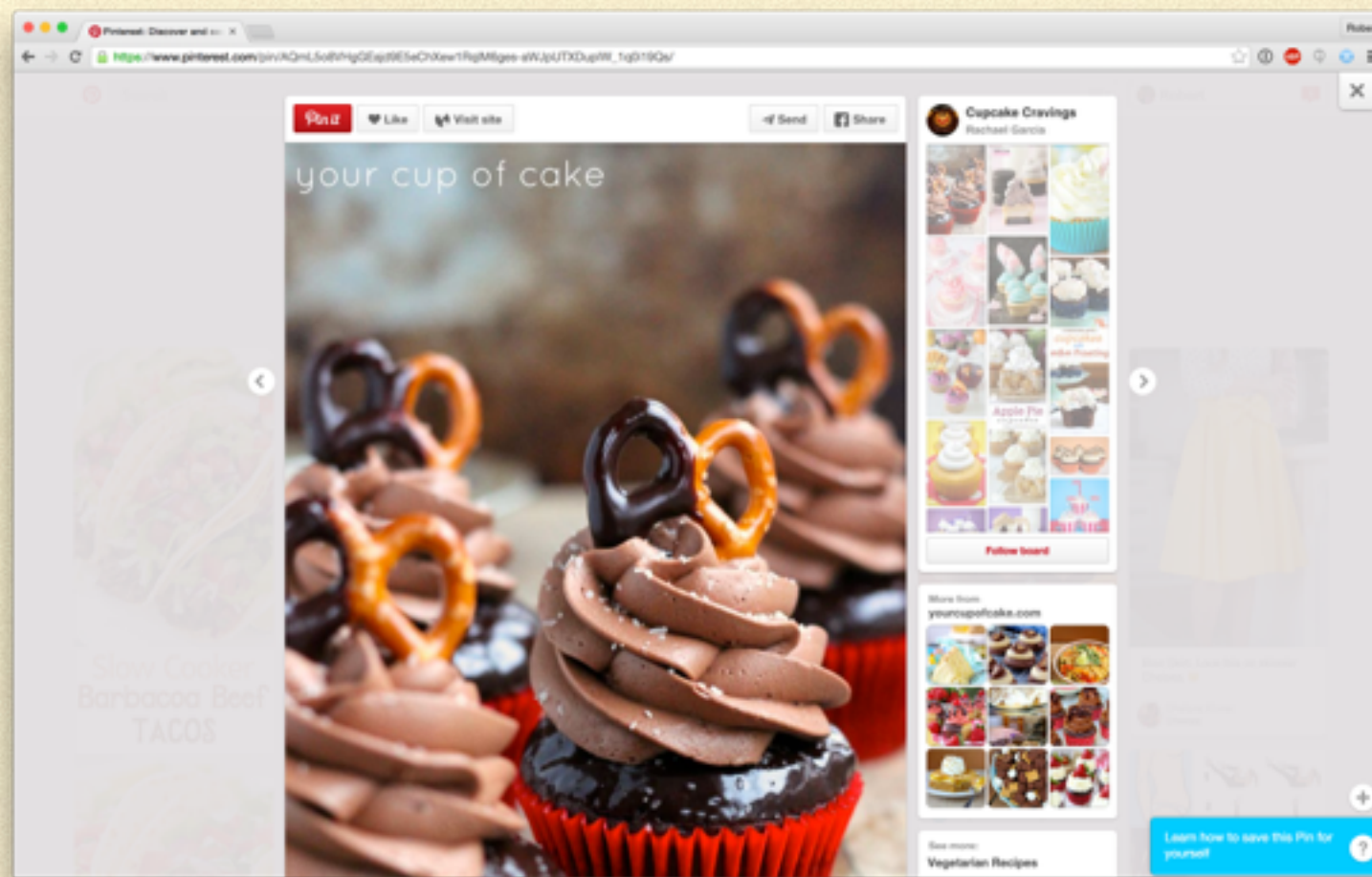
THE CRUD IN ACTION

- The “index” action for reading, should typically display a list of something. Or “starting point”



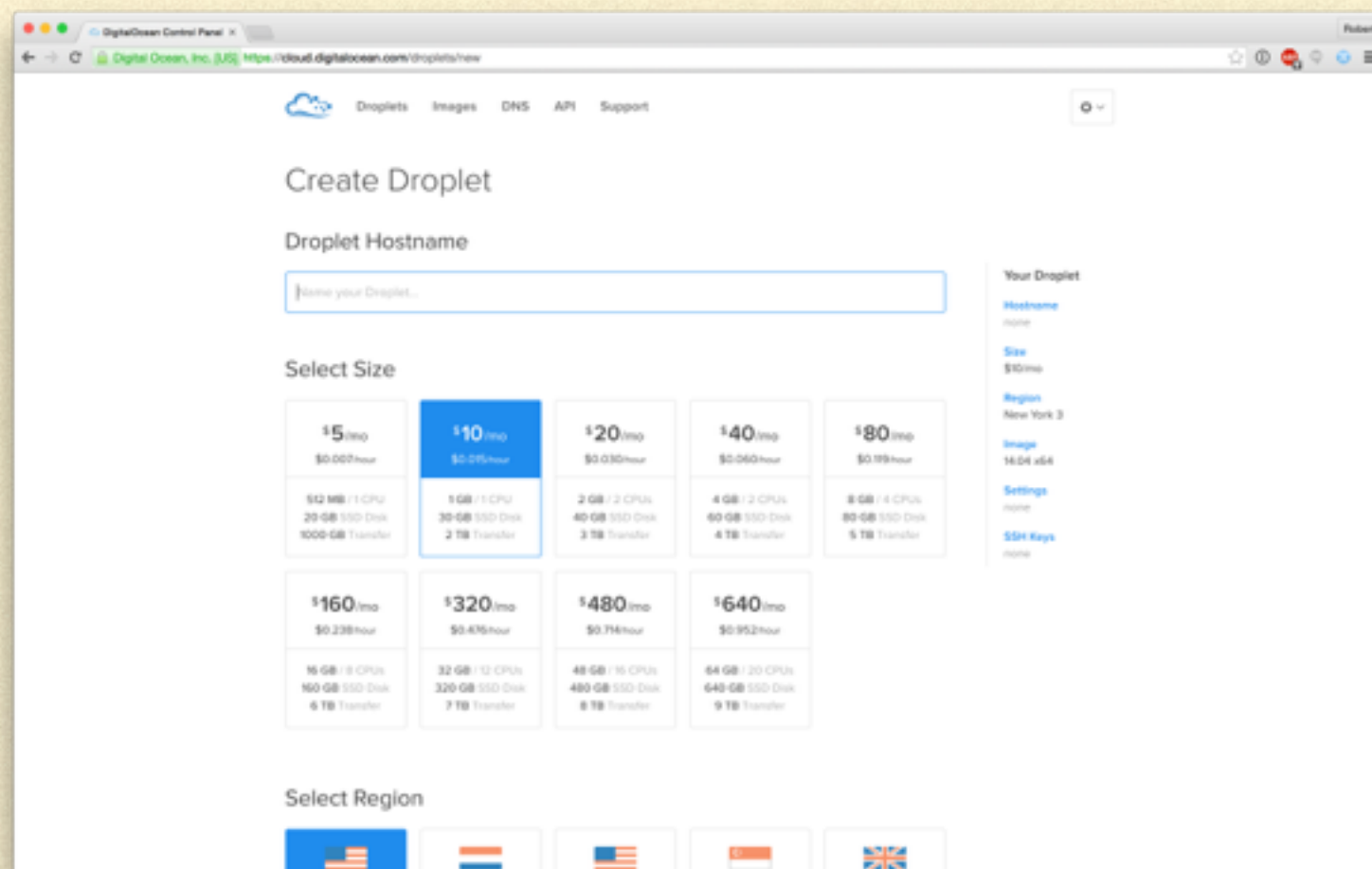
THE CRUD IN ACTION

- The “show” action for reading, should typically display a singular thing as the main attraction.



THE CRUD IN ACTION

- The “new” action for create, should typically display a form to create an object in the database. It will also usually have a “create” action.



The screenshot shows the DigitalOcean 'Create Droplet' form. It includes a 'Droplet Hostname' input field, a 'Select Size' grid with 10 options (ranging from \$5 to \$640/month), and a 'Select Region' section with five flags. A sidebar on the right lists configuration options like Hostname, Size, Region, Image, Settings, and SSH Keys.

Create Droplet

Droplet Hostname

name your Droplet...

Select Size

Size	Price (1mo)	RAM	CPU	Disk	Transfer
512 MB / 1 CPU	\$5	512 MB	1 CPU	20 GB SSD Disk	1000 GB Transfer
1 GB / 1 CPU	\$10	1 GB	1 CPU	30 GB SSD Disk	2 TB Transfer
2 GB / 2 CPU	\$20	2 GB	2 CPU	40 GB SSD Disk	3 TB Transfer
4 GB / 2 CPU	\$40	4 GB	2 CPU	60 GB SSD Disk	4 TB Transfer
8 GB / 4 CPU	\$80	8 GB	4 CPU	80 GB SSD Disk	5 TB Transfer
16 GB / 8 CPU	\$160	16 GB	8 CPU	160 GB SSD Disk	6 TB Transfer
32 GB / 12 CPU	\$320	32 GB	12 CPU	320 GB SSD Disk	7 TB Transfer
48 GB / 16 CPU	\$480	48 GB	16 CPU	480 GB SSD Disk	8 TB Transfer
64 GB / 20 CPU	\$640	64 GB	20 CPU	640 GB SSD Disk	9 TB Transfer

Select Region

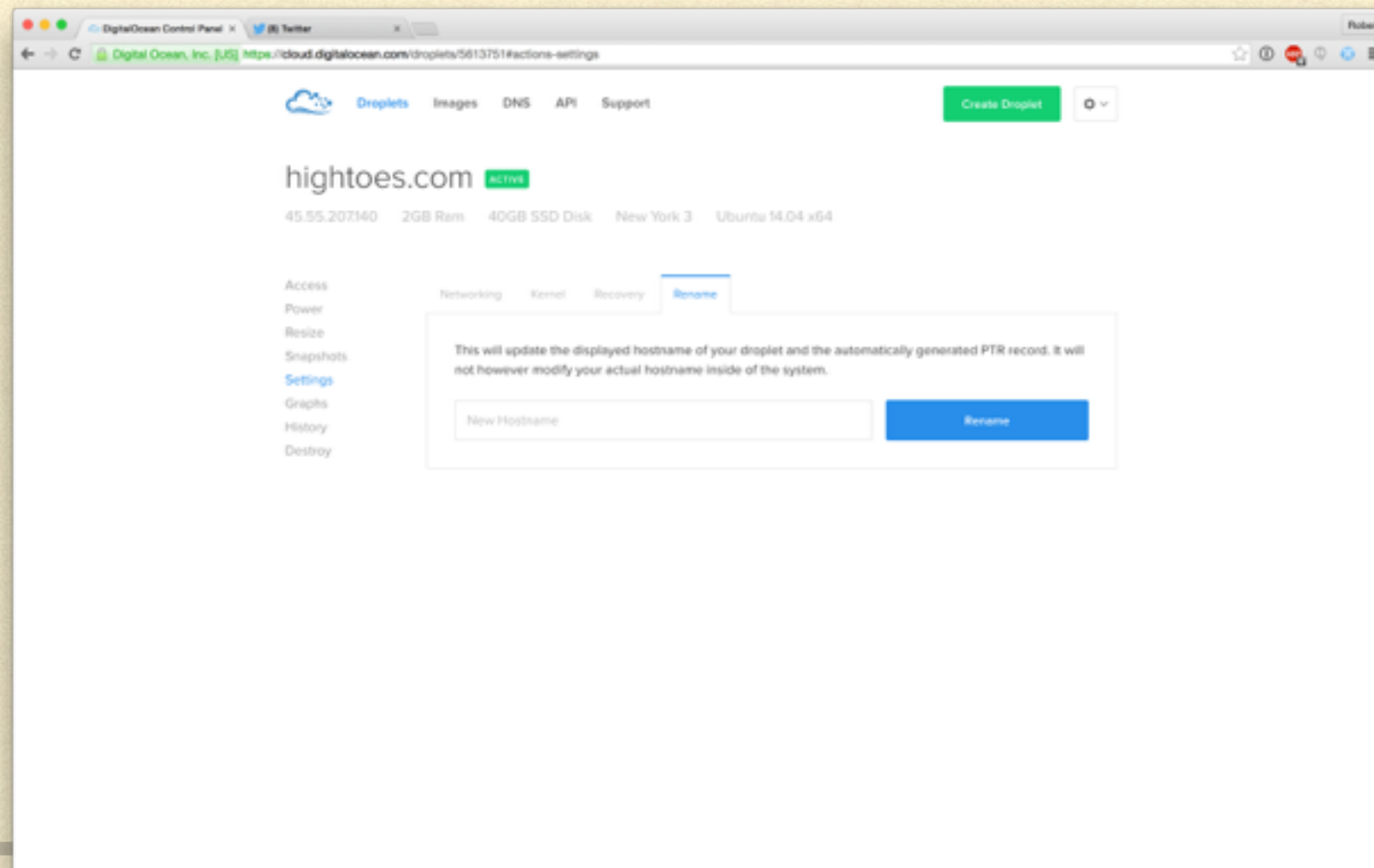
Flags: United States, Netherlands, Germany, Singapore, United Kingdom

Your Droplet

- Hostname: none
- Size: \$10/mo
- Region: New York 3
- Image: 16.04 x64
- Settings: none
- SSH Keys: none

THE CRUD IN ACTION

- The “edit” action should display a form pre-populated with information about the item being edited. It should send the user input to the “update” action on the same controller.



DYNAMIC ROUTES

- Now that we know a little bit more about actions and their individual purposes, let's talk about dynamic routing in rails.
 - The gist of it is, a route can define a controller and action, but it can also pass in more information from the URL requested.
-

DYNAMIC ROUTES

- *“but it can also pass in more information from the URL requested.”*
 - Let’s look at a few different URL paths:
 - /users/12345123
 - /users/15060
 - Both of these URL’s are different, but point to the same controller and action.
 - The numbers in these URL’s are the ID of the user we want detail for.
-

DYNAMIC ROUTES

- We can't create every single route for every single ID in our database (especially since it could grow while we're asleep).
 - What I mean is:
 - `get "/users/12345123" => "users#user_12345123"`
 - Simply makes no sense at all.
-

DYNAMIC ROUTES

- Instead we have what are called “placeholders” for routing.
 - A placeholder starts with a colon, and is followed by a word(s) separated by underscores. It is used within the path portion of the route.
 - For example:
 - get “/users/:id” => “users#show”
 - This will match any URL that has the format:
 - /users/{number}
 - And still go to the users controller, and show action.
-

DYNAMIC ROUTES

- get “/users/:id” => “users#show”
- When assigning a route this way, the :id portion becomes available in the params hash within our controller. For example:

↔ `users_controller.rb`

```
1 class UsersController < ApplicationController
2   def show
3     user_id = params[:id]
4   end
5 end
```

DYNAMIC ROUTES

- get “/users/:id” => “users#show”
- So in a browser, if we make a request to “/users/12345”, then:

 **users_controller.rb**

```
1 class UsersController < ApplicationController
2   def show
3     user_id = params[:id]
4   end
5 end
```

user_id would be assigned as “12345” (a string)

DYNAMIC ROUTES NOTES

- Keep in mind that the route helper, “resources” will define 7 routes for you. Including the routes for specific models (show, edit, etc...)

Create		
POST	/photos(:format)	photos#create
GET	/photos/new(:format)	photos#new
Read		
GET	/photos(:format)	photos#index
GET	/photos/:id(:format)	photos#show
Update		
GET	/photos/:id/edit(:format)	photos#edit
PATCH	/photos/:id(:format)	photos#update
PUT	/photos/:id(:format)	photos#update
Delete		
DELETE	/photos/:id(:format)	photos#destroy

LAB - FLEX THEM RAILS MUSCLES

- Another day, another rails app. Getting your reps in. Work with your seat mate.
-
- Please create a rails app called “flickr” in your class 12 folder.
 - In the rails application, please add *resourceful* routes for “photos”. Hint: 7 routes.
 - Generate a Photo model with a url:string and caption:string attributes
 - Add controller actions and views for index, show, new, create, and delete. Seed with a few photos from the web. In the index and show views, display the photo with HTML
 - Damn, look at all the stuff you already know!
 - Try to do as much as possible without peeking at the last app, but it's cool if you do.
-

Editing

LETS ADD
EDITING!!!!!!



THE ROUTE

- You'll notice there's a route for the edit action in our routes for photos:
 - `GET /photos/:id/edit(.:format) photos#edit`
 - This points to the edit action on our photos controller.
 - The edit action is similar to the “new” action, in that it only displays a form. But slightly more.
-

THE CONTROLLER

- The edit action typically will fetch the item we're editing from the database using our model, and assign it an instance variable. Like so:

 **photos_controller.rb**

```
1  class PhotosController < ApplicationController
2    def edit
3      @photo = Photo.find(params[:id])
4    end
5
6    def update
7      # ...
8    end
9  end
```

WHY

- Just like the “new” action our controllers have had, we need an instance of a photo to pass to our view, to render a form correctly.
 - By finding a pre-existing instance of a photo from our database using the “.find()” method on our model, the form will automatically be pre-populated with the values.
 - Remember, our edit action is solely to display a form. Which is pre-populated.
-

UPDATING A MODEL OBJECT

- Models can be updated several ways, but the easiest way is use the “update” method on an instance of a model that has already been saved.
 - For example:
 - `photo = Photo.find(123)`
 - `photo.update(url: “flickr.com/newfile.png”, caption: “whut”)`
 - Notice that we are passing in something like a hash?
-

UPDATING A MODEL OBJECT

- We can reuse `safe_photo_params`

```
def update

  @product = Product.find(params[:id])
  @product.update(safe_product_params)

  flash[:notice] = "Product updated!"
  redirect_to products_path

end
```


THE FORM

- The form is basically identical to the “new” form we learned a week ago. With one change: the submit button text that we don’t really need to begin with.

```
edit.html.erb
1  <%= form_for @photo do |form| %>
2    <p>
3      Filename:
4      <%= form.text_field :filename %>
5    </p>
6
7    <p>
8      Caption:
9      <%= form.text_field :caption %>
10   </p>
11
12   <%= form.submit "Update Photo!" %>
13 <% end %>
```

NOTICE HOW THAT FORM IS EXACTLY
THE SAME AS THE FORM IN NEW?

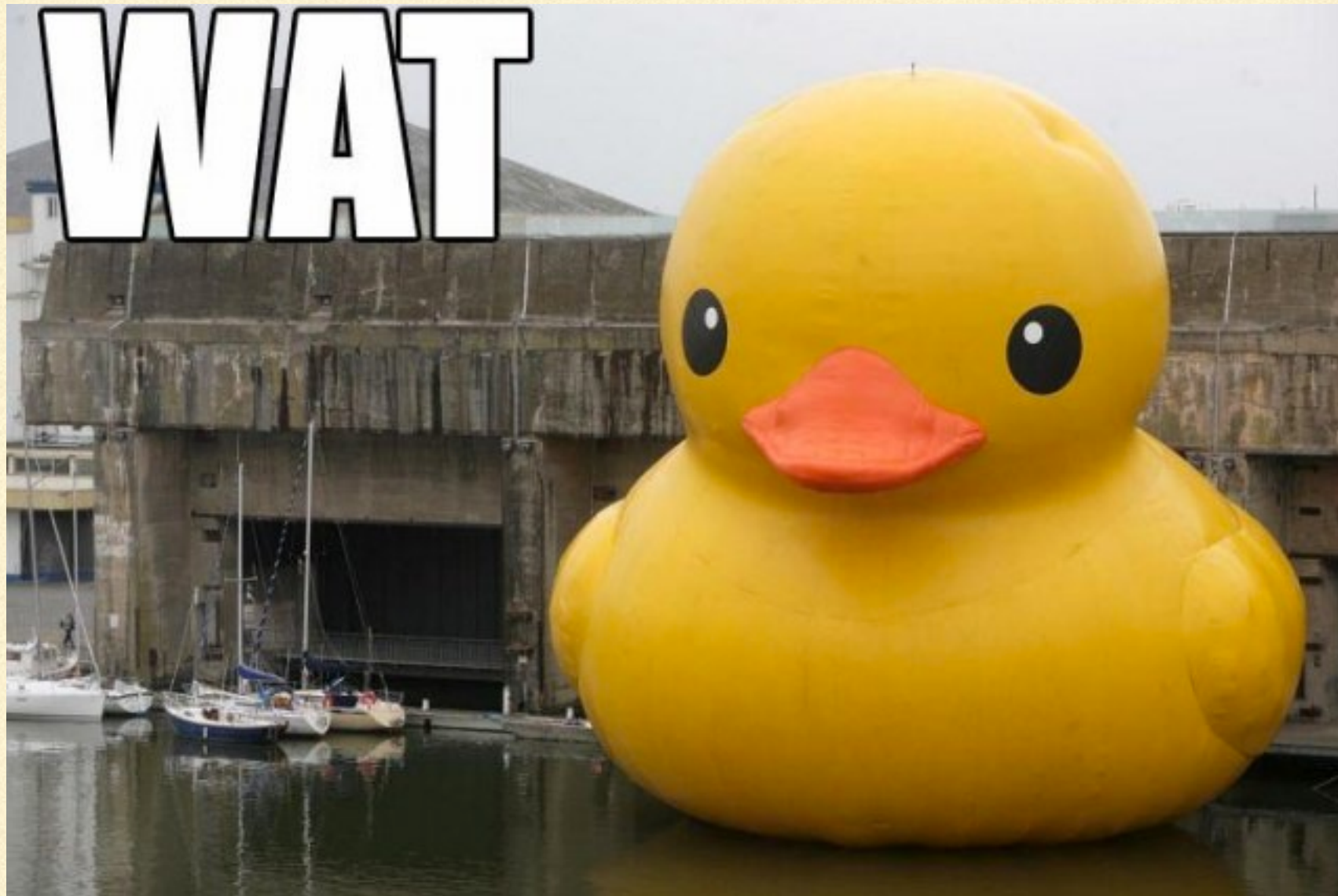
DRY (DON'T REPEAT YOURSELF)

- Programmers hate repeating themselves.
 - Not-DRY code is bad, error prone and hard to maintain. Whenever you see repetition, consider refactoring.
 - The form in edit and new is the same damn thing. When you have views like this, you can extract it into a *partial*.
-

PARTIALS

- You can render a view inside a view.
 - Prepend an underscore(_) to a view to make it a partial.
 - Paste the form code into `app/views/photos/_form.html.erb`
 - In `edit` and `new.html.erb`, just put `<%= render 'form' %>`
-

BREAK



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.
-

LAB

- In the flickger application, we're going to add a new model: User
 - Generate a model called User
 - On the model, add the follow attributes: email, name
 - 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.
 - Add the all dat CRUD for the user model
 - This means, you should be able to create, view a list of users, and edit a user.
-

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

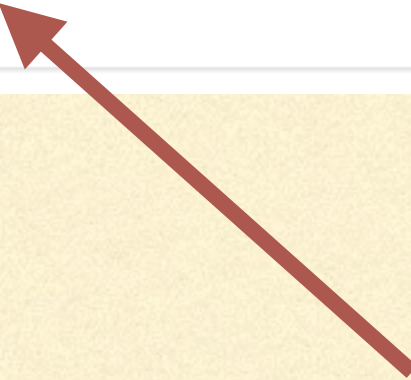
GIVEN A MODEL "USER" AT APP/MODELS/USER.RB

 user.rb

```
1 class User < ActiveRecord::Base
2   # Adds a validation checking to see if the name is "present",
3   # meaning that is not nil, or an empty string.
4   validates :name, presence: true
5 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

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

example.rb

```
1 abc = User.new(name: "")
2
3 did_it_save = abc.save
4 puts "Did it save?: #{did_it_save}"
5 # => Prints out "Did it save?: false"
6 # 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


```
1 # app/controllers/users_controller.rb
2 class UsersController < ApplicationController
3   # assume the 'new' action is also defined
4
5   def create
6     @user = User.new(user_params)
7
8     # calling "save" will automatically run validations on your model.
9     # rails handles that for you (rails magic!)
10    # if any of the validations fail, this if statement will fail
11    # and continue to the else
12    if @user.save
13      redirect_to users_path
14    else
15      # This will render app/views/users/new.html.erb instead of create.html.erb
16      render :new
17    end
18  end
19
20  private
21
22  def user_params
23    params.require(:user).permit(:name)
24  end
25 end
```

ASSOCIATIONS

IMAGINE YOU ARE THIS GUY



IT'S 2006, YOU ARE ABOUT TO CREATE THIS



Use twtr to stay in touch with your friends all the time. If you have a cell and can txt, you'll never be bored again...EVER!

What your friends are

Kevin Symon baby-sitting
Jeremy Home - going to bed
Kate: it's my birthday! :))

txt
{or}
What are you doing?

Send

.timeline
follow along with what your friends are doing throughout the day

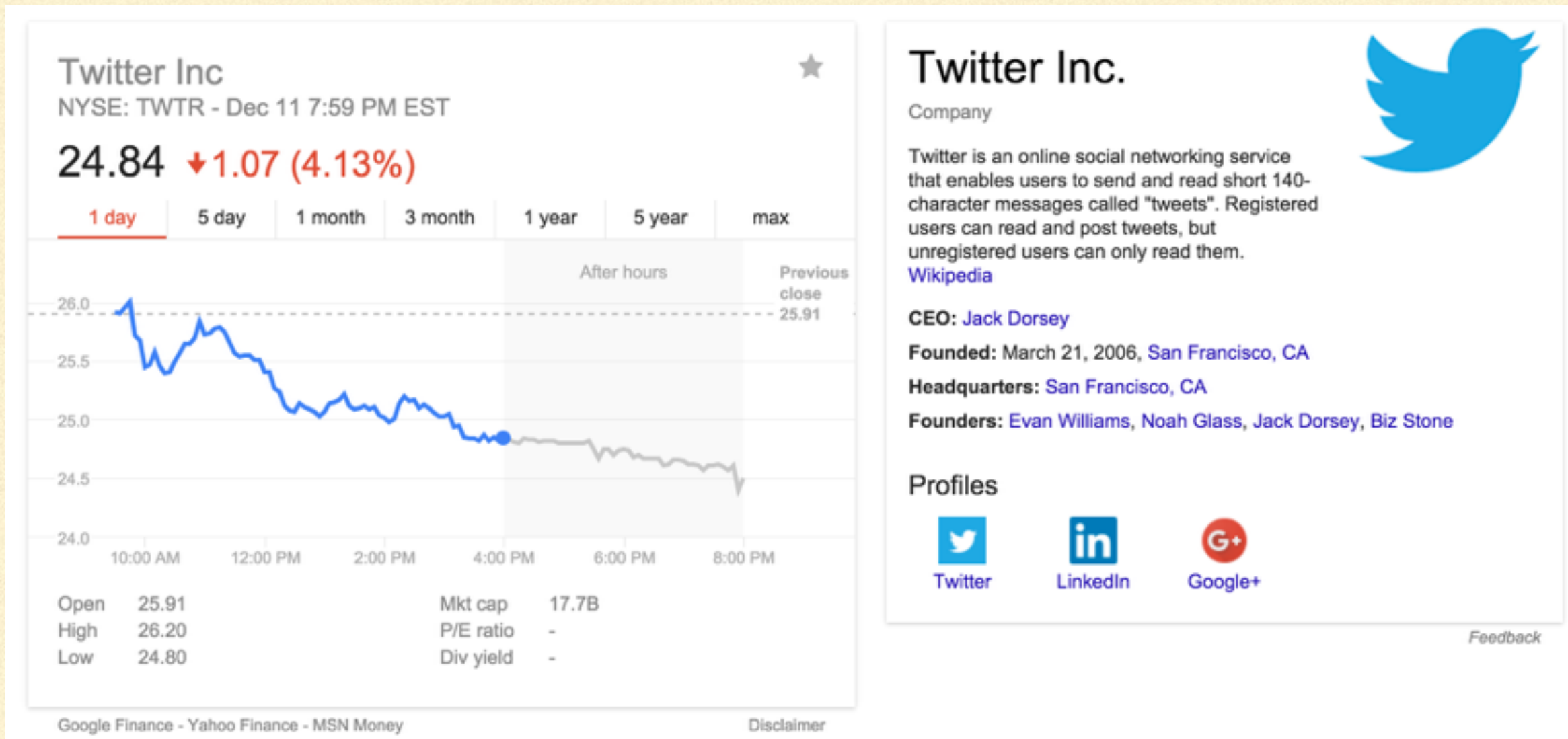
★ Florian x {x4}
★ Gareth x {x4}
★ goldman x {x4}

.what up?
send updates from your cell or from the web about what's in yr head

Sign in.
Mobile number (or email)
Password (or PIN)
☐ Remember me
Sign in

New? Sign up!
twtr works best when updated from your mobile phone. To verify you are you, we'll need your number.
Mobile number
Get started

WHICH EVENTUALLY BECOMES THIS



GRAB SOME MARKERS

WITH A PARTNER

- Draw on your desk or paper or whatever.
 - Think of the models that you would need to make Twitter.
 - What are some attributes of those models? Validations?
 - Think of some controller actions for those objects.
 - Think about how those objects relate to one another.
-

ASSOCIATION BETWEEN MODELS

TWITTER, FOR EXAMPLE

- Users have many tweets
 - Users have many favorites
 - Tweets have many Users who favorited them.
 - Tweets belong to individual users
 - Users are associated to other users — following
-

ASSOCIATIONS

- When we make resources, sometimes they need to interact.
- For example, if we want to store the user who wrote a tweet, we could start by storing the `user_name` as an attribute on a story.
- Then we'd add `user_email`, `user_nickname`, etc... and it would quickly get messy

```
create_table "tweets", force: true do |t|  
  t.string :user_name  
  t.string :user_email  
  t.string :user_nickname  
  ...  
end
```

```
create_table "tweets", force: true do |t|
  t.string :user_name
  t.string :user_email
  t.string :user_nickname
  ...
end
```

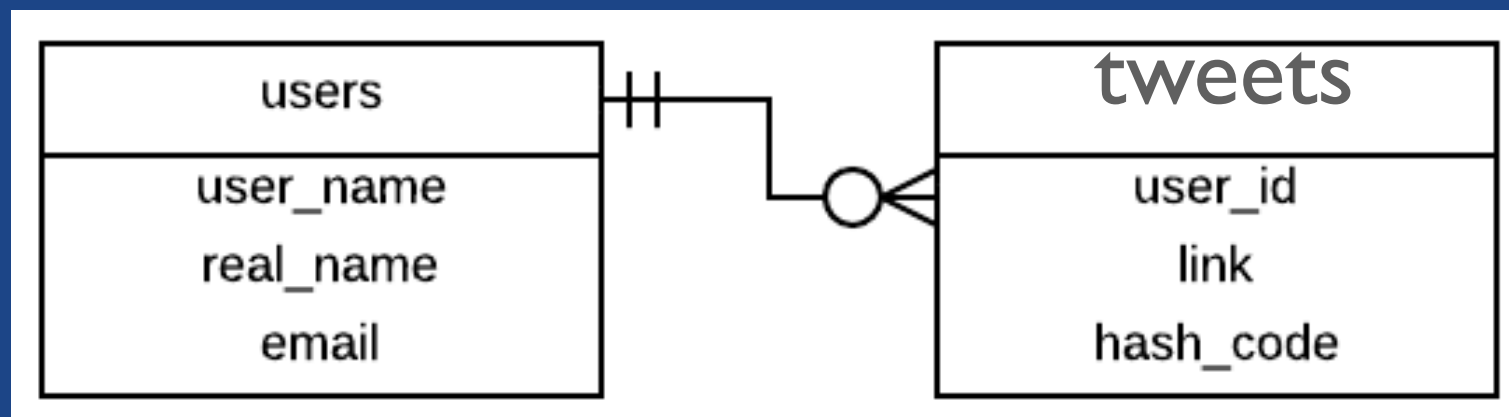
Let's not do that ^^

We have a User model for that, so let's leave it be.

```
create_table "users", force: true do |t|
  t.string :user_name
  t.string :real_name
  t.string :email
  ...
end
```

ASSOCIATIONS: CONCEPT & CODE

AN ASSOCIATION USING AN ENTITY-RELATIONSHIP DIAGRAM



```
create_table "tweets", force: true do |t|
  t.text      "link"
  t.string    "hash_code"
  t.integer   "user_id"
end

class Tweet < ActiveRecord::Base
  belongs_to :user
end

class User < ActiveRecord::Base
  has_many :tweets
end
```

The primary key is the field that uniquely identifies each row of a table. in rails, this is always the ':id'

The foreign key identifies which row of another table it is associate with ':user_id'

FLICKR APP

- We can go back to the photo model that we already have and add the foreign key for user.
 - In your terminal:
 - `rails g migration AddUserToPhotos user:references`
 - Look at the migration, then `rake db:migrate`
-

ASSOCIATIONS: SIX TYPES

- **belongs_to**
- **has_many**
- **has_one**
- **has_and_belongs_to_many**
- **has_many :through**
- **has_one :through**

ASSOCIATING THE MODELS

- We can now associate the models. Let's tell user it has many photos
 - And photos belong to users.
 - This means a photo can have a user
 - And users have photos.
 - Let's practice in the rails console.
-

GOING FORWARD

- We will soon be able to show only a particular user's photos, let signed in users create their own photos, and lots of other stuff.

LAB / HOMEWORK

- Spend a few minutes thinking about how Models might be related in your final project app.
 - Run it by your favorite co-student.
 - Challenge each others' assumptions
 - Add this to your project proposal.
-

LAB / HOMEWORK

- Spend a few minutes thinking about how Models might be related in your final project app.
 - Run it by your favorite co-student.
 - Challenge each others' assumptions
 - Add this to your project proposal.
-