INFO 2310

Topics in Web Programming
Ruby on Rails

Last week on INFO 2310

we...

- Built our first static pages
- Made them slightly more dynamic, using test driven development
- Wrote our first helper
- Played with Ruby!

Week 3

Ruby &&
ActiveRecord &&
The User Model

Lets log in to Amazon and start our EC2 instances

Ruby basics

Hopefully, you went through the material from the end of the last lecture on your own, and started to get a feel for the (very) basics of the Ruby language.

I won't go over the nitty gritty syntax again, but I will touch on the things I would think are most new to Rubyists coming from other languages.

Blocks

Lots (and lots) of objects in Ruby respond to functions which take blocks as arguments.

For example

```
3.times { puts "hello, world" }

user = {:first_name => "matt", :last_name => "goggin"}

user.each do |key, value|
 puts "#{key} => #{value}"
end

user.each { |key, value| puts "#{key} => #{value}" }
```

The convention is to use curly braces "{...}" for blocks that fit on a single line, and "do...end" for blocks that span multiple lines

Blocks...

You can think of the code inside of the block as a mini-method, and the variables between the pipes "|" are the arguments to the method.

```
["hello", "world", "again"].each_with_index do |word, index|
    puts "#{word} is the #{index} word"
end
```

So this block could be thought of as a method which takes two arguments (word and index), and prints out that the position of that word.

For Loops...

are valid ruby

```
for i in [1,2,3,4,5]
  puts "hello, world #{i}"
end
```

But they are not the ruby way.

instead....

```
5.times do |i|
  puts "hello, world #{i}"
end
```

or

```
[1,2,3,4,5].each { |i| puts "hello, world #{i}" }
```

Classes

```
# The Greeter class
class Greeter
  def initialize(name)
    @name = name.capitalize
  end

  def salute
    puts "Hello #{@name}!"
  end
end


g = Greeter.new("world")        # Create a new object
g.salute                        # Output "Hello World!"
```

attr_accessor

attr_accessor adds a field to a Ruby class.

E.G.

```
class MyClass
  attr_accessor :my_field
end
```

```
class MyClass

  def my_field=(value)
    @my_field = value
  end

  def my_field
    @my_field
  end
end
```

```
foo = MyClass.new
puts foo.my_field        # => nil
foo.my_field = "a value"
puts foo.my_field        # => "a value"
```

Take that, Java style setters and getters!!

Open Classes

In Java, you define a class in a single file, and that is that.

In Ruby, you can open and modify any class, any time, anywhere (!!).

A user class

```
class User
	def username
		"matt"
	end
end

class User
	def email
		"mg343@cornell.edu"
	end
end

u = User.new
puts u.username		# "matt"
puts u.email		# "mg343@cornell.edu"
```

Even built in classes can be opened

```
class Fixnum
	def +(b)
		self - b
	end
end

4 + 4
=> 0
```

This is called "monkey patching". It's very rarely a good idea. While this (silly) example is obviously a bad thing to do, in general modifying the behavior of built in classes (String, Fixnum, ...) is a BAD idea.

Back to Rails!

We'll start today with one of the most powerful features of Rails, which is ActiveRecord.

ActiveRecord is an Object Relational Mapper (ORM), which gives us a Ruby API for manipulating records in our database.

It has functions for inserting, updating, destroying, validating, finding... all without us typing any SQL ourselves.

....sweet

Login to PuTTY

- open PuTTY
- On the left panel, navigate to Connection->SSH->Auth
  - Next to "Private key file for authentication:", click "Browse", and select the *.ppk file you created on the previous step.
- Then, navigate to Connection->Data
  - For "Auto-login username", type "ec2-user"
- Navigate to "Session" (the very top)
  - Copy the "Public Domain Name" of your EC2 instance; you can see this on the instances page of the Amazon console, when an instance is selected
  - Paste it into the "Host Name (or IP address)" field
- Click "Open"

Login to WinSCP

- Open WinSCP
  - Paste in your domain to "Host name", as you did in PuTTY
  - Type "ec2-user" for the "User name"
  - Click "..." to select your private key file
  - Click "Login"
- Set NotePad++ as the default editor.
  - Click Options->Preferences
  - Select "Editors" from the left tab
  - Click Add
  - Select "External Editor"
  - Find NotePad++ (C:\Program Files (x86)\Notepad++)
  - Click "Open", then "Okay"
  - Drag it to the top of the editor list

What is an ORM? Why do we want one?

ORM -> Object Relational Mapper

e.g. ActiveRecord in our case

When we program, we are used to working with objects. E.g. a User object, or a Post object.

An ORM gives us a layer of abstraction that maps records in our database to Ruby objects that we can manipulate programmatically.

So if we had this record in the "users" table in our database..

id	username	email
1	goggin13	mg343@cornell.edu

We can do stuff like this...

```
user = User.find_by_id(1)
puts user.username   # => "goggin13"

user.email = "goggin@example.com"
user.save!
```

Instead of writing out the SQL queries ourselves.

First things first

Since we are working on a new feature today, let's start on a feature branch

```
git status                          # should display nothing to commit
git checkout -b user_model              # checkout a new branch
git branch                          # view branches
```

Migrations

Aside from giving us a SQL'less API for interacting with our relational database, ActiveRecord gives us tools for creating and modifying the schema as well.

These are called migrations.

config/database.yml

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000
```

examples for other databases (MySQL, PostgreSQL) here->
https://gist.github.com/961978

Back to the generator

Not surprisingly, Rails will generate the necessary files so we can start modifying them.

rails generate scaffold User name:string email:string

We generate everything we need to support a new model, of class User, and tell Rails it will have 2 fields of type string.

Remember other generator commands we used?

-> rails generate controller StaticPages home new --no-test-framework

-> rails generate integration_test StaticPages

Lots (and lots) of output

What did we get?

- a database migration
- a User class
    - a spec for the User class
- a UsersController
    - a spec for the UsersController
- routes for the UsersController
- views for viewing, editing, creating users
    - specs for these views
- a UserHelper module
    - a spec file for the UserHelper module

db/migrate/[timestamp]_create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :name
      t.string :email

      t.timestamps
    end
  end
end
```

* create_table is a method which takes a symbol and a block
  * other possible function calls inside the block
    * t.integer :age
    * t.text :bio (see guides.rubyonrails.org/migrations.html#supported-types)
* t.timestamps creates two time fields, *created_at* and *updated_at* which Rails will maintain for us automagically.
* what's up with the timestamp?

To execute the migration, we run the migrate command:
bundle exec rake db:migrate

which creates the following table in our sqlite database:

users	
id:	integer
name:	string
email:	string
created_at:	datetime
updated_at:	datetime

What did we get?

Now that our database is ready to accept users, we can take a look around.

lets start our server

rails s

and check out

your_domain.com:3000/users

app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :email, :name
end
```

The class User inherits from ActiveRecord::Base (this is what will give us all the ActiveRecord goodness we're about to see).

attr_accessible dictates which properties can be modified via "mass assignment". More on that soon.

Let's test drive it

Remember "rails console" from last time?

Let's use it to explore the ActiveRecord API

rails console

Creating a user

User.new

=> #<User id: nil, name: nil, email: nil, created_at: nil, updated_at: nil>

user = User.new(name: "Matt", email: "mg343@cornell.edu")
=> #<User id: nil, name: "Matt", email: "mg343@cornell.edu", created_at: nil, updated_at: nil>

user.save

=> true

user

=> #<User id: 1, name: "Matt", email: "mg343@cornell.edu", created_at: "2013-01-30 23:10:53", updated_at: "2013-01-30 23:10:53">

user.name

=> "Matt"

user.email

=> "mg343@cornell.edu"

user.updated_at

=> Wed, 30 Jan 2013 23:10:53 UTC +00:00

Creating...

User.create(name: "A Nother", email: "another@example.org")

=> #<User id: 2, name: "A Nother", email: "another@example.org", created_at: "2013-01-30 23:17:07", updated_at: "2013-01-30 23:17:07">

foo = User.create(name: "Foo", email: "foo@bar.com")

=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2013-01-30 23:19:34", updated_at: "2013-01-30 23:19:34"

User.count

=> 3

... and destroying

foo.destroy

=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2013-01-30 23:19:34", updated_at: "2013-01-30 23:19:34">

foo

=> #<User id: 3, name: "Foo", email: "foo@bar.com", created_at: "2013-01-30 23:19:34", updated_at: "2013-01-30 23:19:34">

User.count

=> 2

Finding

User.find(1)

=> #<User id: 1, name: "Matt", email: "mg343@cornell.edu", created_at: "2013-01-30 23:10:53", updated_at: "2013-01-30 23:10:53">

User.find(3)

=> ActiveRecord::RecordNotFound: Couldn't find User with id=3

User.find_by_id(3)

=> nil

User.find_by_email('mg343@cornell.edu')

=> #<User id: 1, name: "Matt", email: "mg343@cornell.edu", created_at: "2013-01-30 23:10:53", updated_at: "2013-01-30 23:10:53">

User.find_by_name('Matt')

=> #<User id: 1, name: "Matt", email: "mg343@cornell.edu", created_at: "2013-01-30 23:10:53", updated_at: "2013-01-30 23:10:53">

Finding...

User.all
User.first
User.last

```
User.all.each { |u| puts u.name }
```

Updating

user.email
=> "mg343@cornell.edu"

user.email = "goggin@example.com"
=> "goggin@example.com"

user.save
=> true

user.email = "williamson@example.com"
=> "williamson@example.com"

user.reload.email
=> "goggin@example.com"

Updating...

user.update_attributes(name: "The Dude", email: "dude@abides.org")

=> true

The ActiveRecord functions which accept a hash as an argument are the functions which are affected by the "attr_accessible" line in the User model.

Only attributes specified there can be modified in this manner. This allows us to pass incoming web (GET, POST, PUT) requests directly into these function calls without worrying about malicious requests modifying data we do not want modified.

user.update_attributes(created_at: "2003-01-30 23:54:42 +0000")

=> ActiveModel::MassAssignmentSecurity::Error: Can't mass-assign protected attributes: created_at

Validations

User.create(name: "", email: "")

=> #<User id: 2, name: "", email: "", created_at: "2013-01-31 03:09:01", updated_at: "2013-01-31 03:09:01">

Does that seem okay? Do we want records in our database with blank names, and blank emails?

Probably not.

Let's fix this.

Enough console for now

type "exit" to close the console

Time to add some features.

Brief TDD break

Since we are just learning lets play with the validations, then we will test them and continue.

edit app/models/user.rb to

```
class User < ActiveRecord::Base
  attr_accessible :email, :name
  validates :name, presence: true
end
```

And back to the console

rails console --sandbox

user = User.new(name: "", email: "mg343@cornell.edu")
=> #<User id: nil, name: "", email: "mg343@cornell.edu", created_at: nil, updated_at: nil>

user.save
=> false

user.valid?
=> false

user.errors.full_messages
=> ["Name can't be blank"]

And back to TDD

Now that we know a little bit about validations, let's write some tests.

If you open up spec/models/user_spec.rb now, you will see

```
require 'spec_helper'

describe User do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

When RSpec generates specs, it gives us the file, but marks the tests as "pending".

If you run the user_spec you will see output denoting the pending test.

```
bundle exec rspec spec/models/user_spec.rb
```

And now the specs

This is a big snippet, so let's just take it from the lecture_3.txt file and go through it.

If we run them

bundle exec rspec spec/models/user_spec.rb

we see the failure from the email validation

Green

app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :email, :name
  validates :name, presence: true
  validates :email, presence: true
end
```

bundle exec rspec spec/models/user_spec.rb
should now be passing

What else should we validate?

```
class User < ActiveRecord::Base
  attr_accessible :email, :name

  validates :name, presence: true,
                   length: { minimum: 4, maximum: 50 }

  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true,
                   format: { with: VALID_EMAIL_REGEX },
                   uniqueness: { case_sensitive: false }
end
```

Check out all the validators here:

http://guides.rubyonrails.org/active_record_validations_callbacks.html

In the interest of time, we will discuss but not test these. But we still know in our heart of hearts that testing is the path to robust, maintainable code and we feel appropriately guilty for skipping it.

Back to the browser

Now that our tests are passing again, let's start our server and see what the interface looks like when we input invalid data.

rails s

take a brief moment to bask in all the code we didn't have to write to achieve these results

Users probably need passwords

But... passwords are different.

We want a to allow setting a *password* field on our model, but we don't want to actually save it in plain text. We want to save an encrypted version.

How should we go about this?

First, a column for the encrypted password

```
rails generate migration add_hashed_password_to_users hashed_password:string
```

creates this file

```
class AddHashedPasswordToUsers < ActiveRecord::Migration
  def change
    add_column :users, :hashed_password, :string
  end
end
```

```
bundle exec rake db:migrate          # applies the changes
bundle exec rake db:test:prepare   # and to the test database as well
```

A hashed_password test

```
describe "hashed_password" do
	it "should be populated after the user has been saved" do
		@user.save
		@user.hashed_password.should_not be_blank
	end
end

# the full user_spec is "user_spec #2" in lecture_3.txt
```

Recall "attr_accessor :password" adds a field to the user class which we can then populate.

Adding :password to attr_accessible allows us to set that field via the hash style function calls we played with earlier.

e.g. user.update_attributes! password: "new_password"

app/models/user.rb

```
class User < ActiveRecord::Base
    attr_accessor :password
    attr_accessible :email, :name, :password
    validates :password, presence: true
end
```

This allows us to store a password field on a user object in memory, but it will not be persisted to the database

Callbacks

Now we have a password field in memory. Next, we need to use that field to generate and store the hashed password to the database.

To achieve the desired functionality, we'll use something called callbacks.

Callbacks provide us places to hook into the process of saving, updating, destroying an object.

e.g. before_save, after_destroy, before_validation, many more

http://guides.rubyonrails.org/active_record_validations_callbacks.html

before_save

This one seems like it could serve our purpose
lets add this line of code to the user class:

before_save :encrypt_password

as well as these two functions

```
def encrypt_password
    @hashed_password = encrypt(password)
end

def encrypt(string)
    Digest::SHA2.hexdigest(string)
end
```

This should get us to green. The full code for the user class is in lecture_3.txt

Authenticate

Last thing today; looking ahead, we know we are going to want to authenticate users from a login page.

We will be given an email, and a password, and we want to return a user that matches those, or nil.

Authenticate specs

```
describe "authenticate" do

  before do
    @user.save
  end

  it "should return the user with successful credentials" do
    User.authenticate(@user.email, @user.password).should == @user
  end

  it "should return nil if the given email does not exist" do
    User.authenticate("noone@example.com", @user.password).should be_nil
  end

  it "should return nil if the wrong password is provided" do
    User.authenticate(@user.email, "wrong_password").should be_nil
  end
end
```

again, full code for the user spec is "user_spec #3" in lecture_3.txt

Authenticating

Adding these methods to the user class should round out our authentication.

```
def has_password?(plain_text_password)
      @hashed_password == encrypt(plain_text_password)
end

def self.authenticate(email, plain_text_password)
  user = User.find_by_email(email)

  if user && user.has_password?(plain_text_password)
    user
  else
    nil
  end
end
```

Time to commit

```
git status      # see what we modified
git add -A      # add all the changes
git commit -m "User model"

# merge it back into master
git checkout master
git merge user_model

git push origin master      # github
git push heroku master      # heroku
```

Before next class

You try!

Create a **MicroPost** class that has two attributes
user_id:integer
content:string

both of which are required.

Using the user specs we wrote today as a base, write some simple tests that your validations are working as expected.

When you're done, push it to GitHub and Heroku

Today we...

Learned about ActiveRecord
Learned about validations
Learned about callbacks

Used all these to write the User model and authentication logic