Ruby

www.railsinstaller.org
DRY: “Don't Repeat Yourself”
“Convention over configuration”

Install using rvm: https://rvm.io/
To test rvm:
source ~/.rvm/scripts/rvm
type rvm | head -n 1

General

Keep it **DRY**, which stands for Don't Repeat Yourself.
Leave 2 whitespaces as indentation
In Ruby, everything except `nil` and `false` is considered true.
By default, a Ruby method returns the value of the last expression it evaluated.
Ruby does this via **short-circuit evaluation**. That means that Ruby doesn't look at both expressions unless it has to
Return = ruby always return the last evaluated statement
String interpolation = “some_string #{some_variable}”

Methods’ convention:
Use :: for describing class methods, # for describing instance methods, and use . for example code. Examples:
Class::new => Class.new
Class#method => Class.new.method = instance_of_class.method

$VERBOSE = nil # This is a global variable and contains a verbose flag
require 'prime' # This is a module. Same as #import in python or #include in c++

```
nil = none
Puts “#{var} something” = print “#{var} something\n”
Gets.chomp = input
```

and (&&), or (||), and not (!)
1..10 = includes 10
1...10 = does not includes 10
1_000_000==1000000
<=> = gives -1,0,1

||= assign a variable if it hasn't already been assigned already (conditional assignment operator)

```
# Comment
=begin
Comment of
multiple lines
=end

alias         new   old # symbol syntax not needed... bewilderingly
alias        :new  :old # comma not needed either... go figure
alias_method :new, :old
```

Variables

Ruby enforces some naming conventions. Use underscores
If an identifier starts with a capital letter, it is a constant.
If it starts with a dollar sign ($), it is a global variable.
If it starts with @, it is an instance variable.
If it starts with @@, it is a class variable. [available through a method]

```
$global_variable
@@class_variable
@instance_variable
CONSTANT
::TOP_LEVEL_CONSTANT
OtherClass::CONSTANT
local_variable
```

Operators and Precedence

(Top to bottom)

```
:: .
[]
**
-(unary) +(unary) [-@ +@] ! ~
*  /  %
+  -
<<  >>
```

&
| ^
> >= < <=
<=> == === != =~ !~
&&
||
.. ...
=(+=, -=...)
not
and or
TODO: add = and others
All of the above are just methods except these:
=, ::, ., .., ..., !, not, &&, and, ||, or, !=, !~
In addition, assignment operators(+= etc.) are not user-definable.

Data types

Numeric:
Boolean:

Basic types are numbers, strings, ranges, regexen, symbols, arrays, and hashes. Also included are files because they are used so often.

'a'..'z'
'a'...'z'
(1..10) === 5 # true
(1..10) === 10 # true
(1...10) === 10 # false
(1..10) === 15 # false

String

string = “This is a string”
string[0...4]

Common methods:
.length

.split(“ ”)
.sub(“search”,”replace”) [done only one time]
.gsub(“search”,”replace”) [done globally]

Array

my_array = [1, 2, 3, 4, 5]
multi_d_array = [[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
my_array = [[1, "a"],[1,2],["a","b"]]
empty_table = Array.new(3) { Array.new(3) }

my_array << 6 == my_array.push(6) = gives [1, 2, 3, 4, 5, 6]

For strings:
%w(foo bar baz #{1+1}) == ["foo", "bar", "baz", "\#{1+1}"]
%W(foo bar baz #{1+1}) == ["foo", "bar", "baz", "2"]

Common methods:
.sort
.sort_by
.each
.collect
.first
.last
.shuffle

Hashes (and symbols)

This are the sets and dictionaries of python.
Use symbols for keys
This is a symbol
:symbol = it is just a reference, always has the same id

{key=>value, 2=>4, 3=>6}

```
produce = {"apples" => 3, "oranges" => 1, "carrots" => 12
puts "#{produce['oranges']}"
```

{ key: val } == { :key => val } # 1.9 only.

```
produce = {apples: 3, oranges: 1, carrots: 12
```

```
puts "#{produce[:oranges]}
```

grades = Hash.new
grades = Hash.new(default_value)
grades.default = default_value

To sort
some_hash.sort_by {|key, val| val }

To iterate hashes:
some_hash.each {|key, value| puts "#{key}: #{value}"}

Common methods:
.keys
.values
.select
.each_key
.each_value

Control Expressions

if bool-expr [then]
  body
elsif bool-expr [then]
  body
else
  body
end

unless bool-expr [then]
  body
else
  body
end

expr if   bool-expr
expr unless bool-expr

```
if  bool-expr then expr1 else expr2 end
unless  bool-expr then expr1 else expr2 end

bool-expr ? true-expr : false-expr

case target-expr
when comparison [, comparison]... [then]
  body
when comparison [, comparison]... [then]
  body
# ...
else # optional else
  body
end
```

Case comparisons may be regexen, classes, whatever. Uses #===.

Loops

Structure:
Loop do … end
Loop {....}

```
loop do
  body
  break
end

while bool-expr [do]
  body
end

until bool-expr [do]
  body
end

begin
```

```
  body
end while bool-expr
do_this_one_line_statement while bool-expr

begin
  body
end until bool-expr
do_this_one_line_statement until bool-expr

for name [, name]... in expr [do]
  body
end

expr.each do | name [, name]... | # preferred form over `for`
  body
end

number_of_times_to_do.times {....}
start.upto(end) {....}
start.downto(end) {...}

break # terminates loop immediately.
redo  # immediately repeats w/o rerunning the condition.
next  # starts the next iteration through the loop.
retry # restarts the loop, rerunning the condition.
```

Invoking a method

The convention in Ruby is that a method which returns true or false should have a name ending in a ?.
Methods: ? ask, ! modifies self

```
answer2 = answer.capitalize
Answer.capitalize!
```

Nearly everything available in a method invocation is optional, consequently the syntax is very difficult to follow. Here are some examples:

```
method
obj.method
Class::method # don't use this
Class.method

method(key1 => val1, key2 => val2) # one hash arg, not 2

method(arg1, *other_arg) == method(arg1, *[arg2, arg3]) == method(arg1, arg2, arg3)

# As ugly as you want it to be:
method(arg1, key1 => val1, key2 => val2, *splat_arg) #{ block }
```

The argument syntax is fairly complex

```
invocation := [receiver ('::' | '.')] name [ parameters ] [ block ]
parameters := ( [param]* [',' hashlist] ['*' array] [&aProc] )
block      := '{' blockbody '}' | 'do' blockbody 'end'
```

Defining a Method

```
def method_name(arg_list, *list_expr, &block_expr)
  expr..
end

# singleton method
def expr.identifier(arg_list, *list_expr, &block_expr)
  expr..
end
```

All items of the arg list, including parens, are optional.
Arguments may have default values (name=expr).
Method_name may be operators (see above).
The method definitions can not be nested.
Methods may override operators: |, ^, &, <=>, ==, ===, =~, >, >=, <, <=, +, -, *, /, %, **, <<, >>,

~, +@, -@, [], []= (2 args)

Yield to blocks

Method + block:

```
def block_test
  puts "We're in the method!"
  puts "Yielding to the block..."
  yield
  puts "We're back in the method!"
end

block_test { puts ">>> We're in the block!" }
```

Method + block(with_argument)

```
def yield_name(name)
  puts "In the method! Let's yield."
  yield("Kim")
  puts "In between the yields!"
  yield(name)
  puts "Block complete! Back in the method."
end

yield_name("Eric") { |n| puts "My name is #{n}." }
```

Blocks/Closures

blocks must follow a method invocation (it is not an object):

```
invocation do ... end
invocation { ... }
"this is a sentence".gsub("e"){ puts "Found an E!"}
"this is a sentence".gsub("e"){|letter| puts "Found an #{letter.upcase}!"}
```

Blocks remember their variable context, and are full closures.
Blocks are invoked via yield and may be passed arguments.
Brace form has higher precedence and will bind to the last parameter if invocation made

w/o parens.
do/end form has lower precedence and will bind to the invocation even without parens.

Example:
ints = some_numbers.select {|n| n.is_a? Integer}

Proc Objects

-This does not check the number of parameters it gets
-This does not return control back to the calling method
-A block created with Proc.new behaves like it's a part of the calling method when return is used within it, and returns from both the block itself as well as the calling method.

Definition:
proc_name = Proc.new {|var| expr of var}
Call:
some.method(**&**proc_name) # & converts the proc to an executable block

strings = ["1", "2", "3"]
nums = strings.map(&:to_i) # is this equal to "nums = strings.map.to_i" ?

```
proc      { |args| ... } # {} or do/end
Proc.new  { |args| ... }
lambda    { |args| ... }
-> (args) { ... }       # 1.9+ only
&:method_name           # calls Symbol#to_proc creating: proc { |o| o.method_name }

# or by invoking a method w/ a block argument and catching it on the
# calling side with a &block_arg:

def my_method &block
  block.call 42
end

obj.my_method { |o| ... }
# in 1.9+, Proc aliases #=== to #call so you can use them as case conditions:
```

```
case []
when :empty?.to_proc then
  # ...
when -> (o) { o > 42 && o.prime? } then
  # ...
end

This_is_the_same = proc { |a, b| a + b }
Than_this = Proc.new { |a, b| a + b }

Example:
proc_name = Proc.new {|n| n.is_a? Integer}
ints = some_numbers.select(&proc_name)
```

Lambda objects

-This checks the number of parameters it gets
-This returns control back to the calling method
-A block created with lambda behaves like a method when you use return and simply exits the block, handing control back to the calling method.

Definition:

```
some_var = lambda {|n| expr of var }
```

Call:

```
some.method(&lambda_name)   # & converts the lambda to an executable block

This_is_the_same = ->(a, b) { a + b }
Than_this = lambda { |a, b| a + b }

Example:
lambda_name = lambda {|n| n.is_a? Integer}
ints = some_numbers.select(&lambda_name)
```

Classes

```
# This is the same as
# class MyClass
```

```
#   attr_accessor :instance_var
# end
MyClass = Class.new do
  attr_accessor :instance_var
  def initialize(param)
    something
  end
end
```

Access restrictions in method
public: totally accessible.
protected: accessible only by instances of class and direct descendants. Even through hasA relationships.
private: accessible only by instances of class (must be called nekkid no “self.” or anything else) [inside class, accessible by functions of the class].

Accessors
We can use attr_reader to access a variable and attr_writer to change it, attr_accessor does both. This gives me the possibility to access the variable from outside the class definition
Define attributes for instances of a class

Class Module provides the following utility methods:

```
attr_reader   :attribute [, :attribute]...  # Creates reader methods
attr_writer   :attribute [, :attribute]...  # Creates setter methods
attr_accessor :attribute [, :attribute]...  # Creates both readers and writers
```

Inheritance

```
class ChildClass < ParentClass
  something
end
```

This support override, if parent class’ method is needed use **super**

```
def method_with_same_name_as_parent_class(param)
    do something
    super(param)
end
```

Class methods

This are methods that only work on the Class and not the instances of the class

```
def ClassName.method_name(param)
   do something
end

def self.method_name(param)
   do something
end
```

Modules

[Module are much like libraries]
The syntax is pretty much like the class syntax but it does not inherit, nor create instances, nor have variables (it should not)
When used inside classes they are called mixin

```
require 'module'    # to include modules as import in python or include in C++
include 'module'   # to include modules at the instance level as from XXX import * in python
extend 'module'    # to include modules at the class level SomeClass.something_from_module
```

Namespacing = this is where to look for a constant or a function
Math::PI
:: is the **scope resolution operator** [equivalent to . for classes]

MODULES

Comparable
Enumerable
Errno
FileTest
GC
Kernel
Marshal
Math
ObjectSpace

Precision
Process

Exception

-NoMemoryError
-ScriptError
LoadError
NotImplementedError
SyntaxError
-SecurityError (1.9: move!)
-SignalException
Interrupt
-StandardError [default for plain rescue]
ArgumentError
EncodingError (1.9)
-Encoding::CompatibilityError (1.9)
-Encoding::ConverterNotFoundError (1.9)
-Encoding::InvalidByteSequenceError (1.9)
-Encoding::UndefinedConversionError (1.9)
FiberError (1.9)
IOError
-EOFError
IndexError
-KeyError (1.9)
-StopIteration
LocalJumpError
Math::DomainError (1.9)
NameError
-NoMethodError
RangeError
-FloatDomainError
RegexpError
RuntimeError [default for plain raise]
SecurityError (1.8: move!)
SystemCallError

-Errno::*
    SystemStackError (1.8: move!)
    ThreadError
    TypeError
    ZeroDivisionError
-SystemExit
-SystemStackError (1.9: move!)
-fatal

Raising and Rescuing

```
raise ExceptionClass[, "message"]

begin
  expr...
[rescue [error_type [=> var],...]
  expr...] ...
[else
  expr...]
[ensure
  expr...]
end
```

Catch and Throw

```
catch :label do
  do_stuff
  throw :label if condition?
  do_other_stuff
end
```

throw :label jumps back to matching catch and terminates the block.
can be external to catch, but has to be reached via calling scope.
Hardly ever needed.

Ruby On Rails

Ruby on Rails is a web framework that makes it easy to build powerful web apps in a short amount of time.
Ruby on Rails is written in the Ruby programming language.
Rails uses MVC, it is short for Model, View, and Controller. MVC is a popular way of organizing your code:

- **Model**: Model code typically reflects real-world things. This code can hold raw data, or it will define the essential components of your app.
- **View**: View code is made up of all the functions that directly interact with the user.
- **Controller**: Controller code acts as a liaison between the Model and the View, receiving user input and deciding what to do with it.

Create an app

1. **rails new MyApp**: command created a new Rails app named MyApp. It generated a number of files and folders that we will use to build the app. The rails new command is the starting point of every Rails project.
2. Change to MyApp: cd MyApp
3. **bundle install**: command installed all the software packages needed by the new Rails app. These software packages are called gems and they are listed in the file **Gemfile**.
4. **rails server**: command started the Rails development server so that we could preview the app in the browser by visiting http://localhost:8000. This development server is called WEBrick.
5. http://localhost:8000: go to this url on the browser

Parts of an app (Controller, route, view, and [later] model)

Request/response cycle

This is called the **request/response cycle**. It's a useful way to see how a Rails app's files and folders fit together (https://www.codecademy.com/articles/request-response-cycle-static).

1. The browser makes a request for the URL http://localhost:8000.
2. The request hits the Rails router in **config/routes.rb**. The router recognizes the URL and sends the request to the controller.
3. The controller receives the request and processes it.
4. The controller passes the request to the view.
5. The view renders the page as HTML.

6. The controller sends the HTML back to the browser for you to see.

Workflow

Using the request/response cycle as a guide, this has been our workflow when making a Rails app.

1. Generate a new Rails app (Create an app).
2. Generate a controller and add an action.
3. Create a route that maps a URL to the controller action.
4. Create a view with HTML and CSS.
5. Run the local web server and preview the app in the browser.

Controller

1. **rails generate controller Pages:** command generated a new controller named **Pages**. This created a file named **app/controllers/pages_controller.rb**.
2. **Inside** the new **Pages controller [file]**, we added a **method** called **home [empty method]**. Methods in Rails controllers are also referred to as controller actions, so here we added the home action to the Pages controller.

Route

1. Open **config/routes.rb** and underneath line 1, type: **get 'welcome' => 'pages#home'**. Now when a user visits **http://localhost:8000/welcome**, the route get **'welcome'** => **'pages#home'** will tell Rails to send this request to the **Pages controller home action**.

View

1. Open **app/views/pages/home.html.erb**, and type in the following **HTML**. Fill in your own name.

```
<div class="main">
  <div class="container">
    <h1>Hello my name is my_name</h1>
    <p>I make Rails apps.</p>
  </div>
</div>
```

2. We've provided **CSS** in the file **app/assets/stylesheets/pages.css.scss**
3. Now it is possible to go to **http://localhost:8000/welcome**

App with database (here comes the model)

Request/response cycle

Here's how a database fits into the request-response cycle. Check out the diagram in the browser (https://www.codecademy.com/articles/request-response-cycle-dynamic).

1. When you type http://localhost:8000/welcome, the browser makes a request for the URL /welcome.
2. The request hits the Rails router.
3. The router maps the URL to a controller action to handle the request.
4. The controller action receives the request, and asks the model to fetch data from the database.
5. The model returns data to the controller action.
6. The controller action passes the data on to the view.
7. The view renders the page as HTML.
8. The controller sends the HTML back to the browser.

Workflow

Using the request/response cycle as a guide, this has been our workflow when making a Rails app.

1. Generate a new Rails app (Create an app).
2. Generate a model and migrate and seed the database.
3. Generate a controller and add an action.
4. Create a route that maps a URL to the controller action.
5. Create a view with HTML and CSS.
6. Run the local web server and preview the app in the browser.

we need four parts to build a Rails app - a model, a route, a controller, and a view.

Model

1. **rails generate model Message**: command created a new model named **Message**. In doing so, Rails created two files:
    a. A model file in **app/models/message.rb**. The model represents a table in the database.
    b. A migration file in **db/migrate/timestamp_name**. Migrations are a way to update the database [creating a new table or changing an existing one, the table, the schema itself and not the contenent].

2. Open the migration file in **db/migrate/timestamp_name**. The migration file contains a few things:

```
class CreateMessages < ActiveRecord::Migration
  def change
    create_table :messages do |t|
      t.text :content
      t.timestamps
    end
  end
end
```

   a. The **change** method tells Rails what change to make to the database. Here it uses the **create_table** method to create a new table in the database for storing messages.
   b. Inside **create_table**, we added **t.text :content**. This will create a **text column (this are SQL types)** called **content** in the messages tables.
   c. The final line **t.timestamps** is a Rails command that creates two more columns in the messages table called **created_at** and **updated_at**. These columns are automatically set when a message is created and updated.
3. **rake db:migrate**: command updates the database with the new **messages data model** (also call Run the migration to update the database).
4. **rake db:seed**: command seeds the database with sample data from **db/seeds.rb**.

Controller

1. **rails generate controller Messages:** command generated a new controller named **Messages**. This created a file named **app/controllers/messages_controller.rb**.
2. **Inside** the new **Messages controller [file]**, we added a **method** called **index**

```
def index
  @messages = Message.all
end
```

   Rails provides seven standard controller actions for doing common operations with data. Here we want display a list of all messages, so we used the **index** action. The **index** action retrieves all messages from the database and stores them in variable **@messages**. The **@messages** variable is passed on to the view.

Route

1. Open **config/routes.rb** and underneath line 1, type: **get '/messages' => 'messages#index'**. Now when a user visits **http://localhost:8000/messages**, the routes file maps this request to the Messages controller's **index** action.
2. Other option in general (not this example):

View

1. Open **app/views/messages/index.html.erb**, and type in the following **HTML**. Fill in your own name.

```
<div class="header">
  <div class="container">
    <img
src="http://s3.amazonaws.com/codecademy-content/courses/learn-rails/img/logo-1m.svg
">
    <h1>Messenger</h1>
  </div>
</div>

<div class="messages">
  <div class="container">

    <!-- Your code here -->
    <% @messages.each do |message| %>
      <div class="message">
        <p class="content"><%= message.content %></p>
        <p class="time"><%= message.created_at %></p>
      </div>
    <% end %>

  </div>
</div>
```

2. We've provided **CSS** in the file **app/assets/stylesheets/messages.css.scss**
3. Now it is possible to go to **http://localhost:8000/messages**

Explication of index.html.erb

The file **index.html.erb is a web template**. **Web templates are HTML files that contain variables and control flow statements**. Rather than write the same HTML over and over again for each message, we can use web templates to loop through and display data from the database. The default web templating language in Rails is embedded Ruby, or ERB. In this case:

1. <% @messages.each do |message| %> iterates through each message in @messages array. We created @messages in the Messages controller's index action.
2. For each message, we use **<%= message.content %>** and **<%= message.created_at %>** to display its content and the time when it was created.

UNTIL HERE THIS EXAMPLE ONLY SHOW MESSAGES

To create and save new messages on the database

So far we've been loading messages from the database and displaying them in the view. How can we create new messages and save them to the database? Looking at the seven standard Rails actions, we need to use the **new** and create **actions**. Let's set them up now

1. Create a **route** that maps requests to **messages/new** to the Message controller's **new** action.
2. Then in the **Messages controller** below the **index** action, add the **new** action:
   def new
   @message = Message.new # This line, I believe, creates the variable for **form_for**, without this, the **link_to** (step 7) does not work
   end
3. In the **routes** file, add this route to map requests to the Message controller's **create** action: **post 'messages' => 'messages#create'**
4. Then in the **Messages controller** below the new action, add a **private method named message_params**. Type:
   private
   def message_params
   params.require(:message).permit(:content)
   end
5. Between the **new** action and the **private method**, add the create action. Type:
   def create
   @message = Message.new(message_params)
   if @message.save
   redirect_to '/messages'
   else
   render 'new'
   end
   end
6. Next, in **app/views/messages/new.html.erb** under line 11, type in the contents as you see here:
   <%= **form_for**(@message) do |f| %>

   <%= f.label :message %>
   <%= f.text_area :content %>

   <%= f.submit "Create" %>

   <% end %>

7. Finally in **app/views/messages/index.html.erb** below the <% @messages.each do |message| %>...<% end %> block, add **<%= link_to 'New Message', "messages/new" %>**
8. Visit **http://localhost:8000/messages** in the browser. Click on **New Message** to add a message of your own

How does it work?

When you visit http://localhost:8000/messages/new to create a **new message**, it triggers the first turn of the request/response cycle:

1. The browser makes a **HTTP GET** request for the URL **/messages/new**.
2. The **Rails router maps this URL** to the **Messages controller's new action**. The new action **creates** a new Message object **@message** and passes it on to the **view** in **app/views/messages/new.html.erb.**
3. In the view, **form_for** creates a form with the fields of the **@message** object.

Then when you fill out the form and press **Create**, it triggers the second turn of the request/response cycle:

4. The **browser sends the data** to the Rails app via an **HTTP POST** request to the URL **/messages**.
5. This time, the Rails **router maps** this URL to the **create** action.
6. The **create** action uses the **message_params method** to safely collect data from the form and update the database.
7. Here we used link_to to create a link to **/messages/new**. Instead of hardcoding **<a>** elements, we can use **link_to** to generate links:
    a. The first parameter is the **link text**
    b. The second parameter is the **URL**

Summary

- A **model** represents a **table in the database**.
- A **migration** is a way to **update the database** with a new table, or changes to an existing table.
- **Rails provides seven standard controller actions for doing common** things such as display and create data
- **Data** can be **displayed in the view** using **ERB web templating** [Html with variables and control flow statements, in this case is embedded ruby].
- **Data** can be **saved into the database** using a **web form**.

Associations (multiple models) [Relational database?]

Workflow

Using the request/response cycle as a guide, this has been our workflow when making a Rails app.

1. Generate a new Rails app (Create an app).
2. Generate a model **with an association between Tag and Destination** and migrate and seed the database.
3. Generate a controller and add an action.
4. Create a route that maps a URL to the controller action.
5. Create a view with HTML and CSS.
6. Run the local web server and preview the app in the browser.

Development

1. Create the app (see create an app)
2. Create model Tag with **rails generate model Tag**
3. Create model Destination with **rails generate model Destination**
4. In **app/models/tag.rb** add a **has_many** method, like this:
   class Tag < ActiveRecord::Base
   has_many :destinations
   end
5. In **app/models/destination.rb**, add a **belongs_to** method:
   class Tag < ActiveRecord::Base #ActiveRecord::Base maps to a “tags” table in database
   belongs_to :tag
   end

In the model files, we used the methods **has_many** and **belongs_to** define an association between **Tag** and **Destination**:

- **has_many** :destinations denotes that a single Tag can have multiple Destinations.
- **belongs_to** :tag denotes that each Destination belongs to a single Tag.

The **has_many** / **belongs_to** pair is frequently used to define one-to-many relationships. A few examples are:

- A Library has many Books; a Book belongs to a Library
- An Album has many Photos; a Photo belongs to an Album
- A Store has many Products; a Product belongs to a Store

6. Now that there's an **association between Tag and Destination**, let's continue and add columns to the migration files. Open the migration file in **db/migrate/ for the tags** table, and add the following columns **(the columns types are SQL types)**:
    - a string column called title: **t.string :title**
    - a string column called image: **t.string :image**
7. Next in the **migration file for the destinations table**, add the following columns:
    - a string column called name: **t.string :name**
    - a string column called image: **t.string :image**
    - a string column called description **t.string :description**
    - the line **t.references :tag [This must be a foreign key for tag or some link to it, they said "This adds a foreign key pointing to the tags table"]**
8. **rake db:migrate**: command updates the database with the new **Tag** and **Destination data models** (also call Run the migration to update the database with **Tag** and **Destination**).
9. **rake db:seed**: command seeds the database with sample data from **db/seeds.rb** (see the file db/seeds.rb sample for Associations on drive).
10. Generate the controller for Tags: **rails generate controller Tags**
11. Add the route from /tags to index method: **get '/tags' => 'tags#index'**
12. Create the index action (method) in the controller, which collect all the tags from the database:

```
def index
  @tags = Tag.all
end
```

13. Create the view, in the index.html.erb create and index for all the tags
14. Go to the browser to see the results
15. Add route to the show action with :id and as: :tag: **get '/tags/:id' => 'tags#show', as: :tag**
16. Add the show action in the Tags controller
17. Create the view in show.html.erb
18. Add a link to this view in index.html.erb with **<%= link_to "Learn more", tag_path(t) %>**
19. Go to the browser to see the results
20. Repeat previous steps to show each destination

To add edit and update

1. Routes:
    - get '/destinations/:id/edit' => 'destinations#edit', as: :edit_destination
    - patch '/destinations/:id' => 'destinations#update'
2. Controller:
    - Edit:

```
def edit
  @destination = Destination.find(params[:id])
```

```
end
```
- Destination_params

```
private
def destination_params
  params.require(:destination).permit(:name, :description)
end
```
- Update

```
def update
  @destination = Destination.find(params[:id])
  if @destination.update_attributes(destination_params)
    redirect_to(:action => 'show', :id => @destination.id)
  else
    render 'edit'
  end
end
```

3. Modify the edit view from the edit action

```
<div class="destination">
  <div class="container">
    <%= image_tag @destination.image %>

    <!-- Your code here -->
    <%= form_for(@destination) do |f| %>
    <div class="field">
      <%= f.label :destination %> <br>
      <%= f.text_area :description %>
    </div>
    <div class="actions">
      <%= f.submit "Update" %>
    </div>
    <% end %>

  </div>
</div>
```

4. Connection from destination to edit_destination in the view: ** <%= link_to "Edit", edit_destination_path(@destination) %> **

How does it work?

When you visit http://localhost:8000/destinations/1/edit to edit a destination, it triggers the first turn of the request/response cycle:

1. The browser makes a HTTP GET request for the URL **/destinations/1/edit.**

2. The Rails **router** maps this URL to the Destinations controller's **edit action**. The edit action finds the destination with **id 1**, stores it in **@destination**, and passes it on to the view **app/views/destinations/edit.html.erb**.
3. In the view, **form_for** creates a form **with the fields of the @destinations** object.

Then when you fill out the form and submit it, it triggers the second turn of the request/response cycle:

4. The browser sends the data to the Rails app via an HTTP POST request to the URL **/destinations/update**. **Patch** verb
5. This time, the Rails router maps this URL to the **update action**.
6. The update uses the **destination_params method** to safely collect data from the form. **It finds the destination in the database, updates its attributes, and redirects to the destination's show page**.

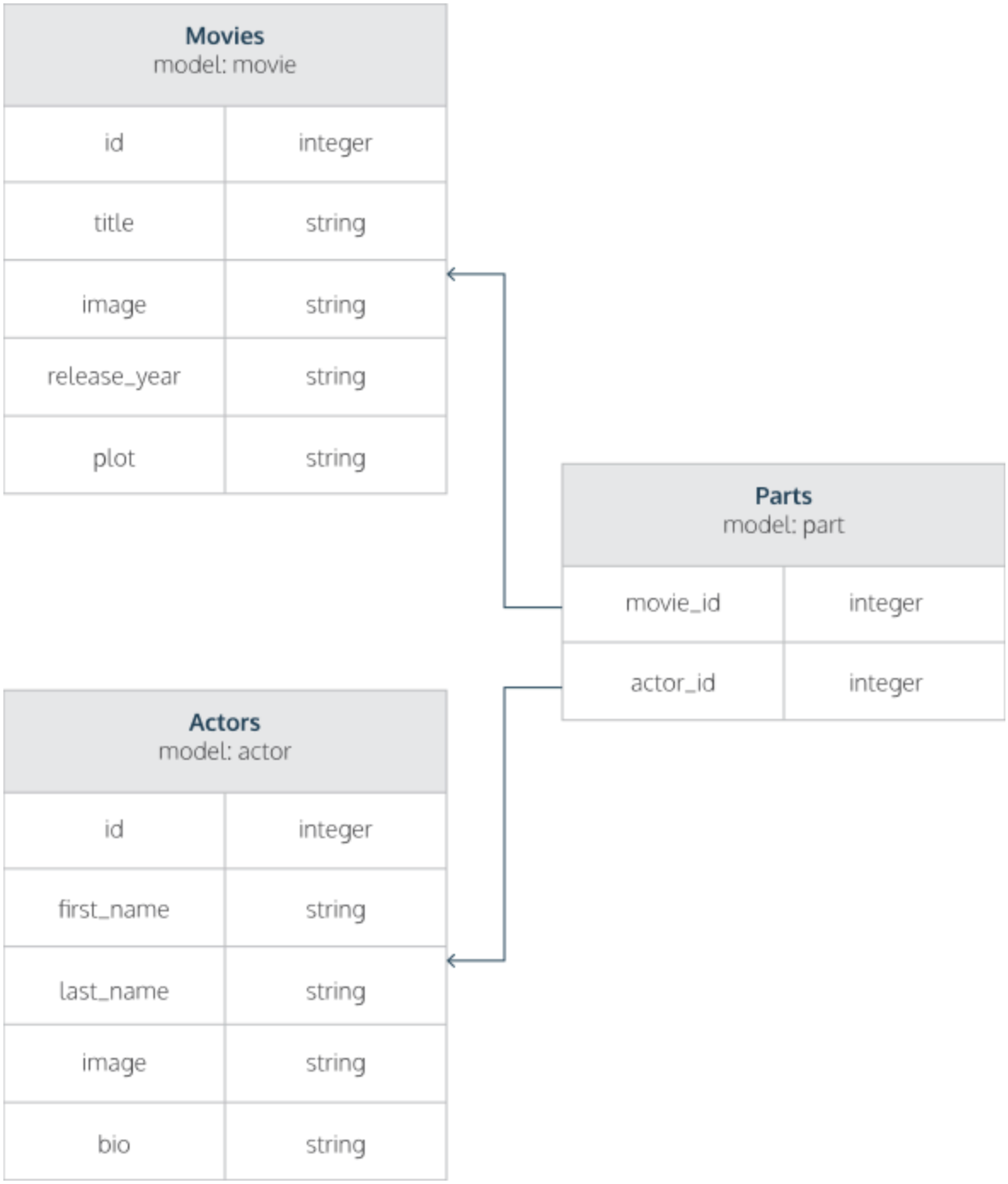
Associations (many to many)

To represent other types of associations use:

- belongs_to
- has_one
- has_many
- has_many :through (has_many :movies, through: :parts)
- has_one :through
- has_and_belongs_to_many

To set dependency in migration use: **t.belongs_to :movie, index: true**

For many to many relationships a intermediate model is needed, a model a belongs to the other models that have it. No controller is needed


Movies
model: movie
id
integer
title
string
image
string
release_year
string
plot
string
Parts
model: part
movie_id
integer
actor_id
integer
Actors
model: actor
id
integer
first_name
string
last_name
string
image
string
bio
string

General Comments:

Model: Zombie
Table: zombies
Controller: Zombies
Route: zombies
View: Methods from controller

Database

Badabase_name.method (allows chaining):
.new
.find # needs id
.find_by(attribute: ‘some’) # returns the first field with attribute
.save
.create # auto save
.attributes
.update # auto save
.destroy
.destroy_all
.count
.order(discriminant)
.limit(X) # how many tweets
.where(key: value) #search for condition, returns a list []

Inside models:

- ActiveRecord::Base = inherence to new class, it maps the Class_name to the class_name’s table
- validates_presence_of :symbol # validates that a new item has a :symbol before saving it, you can validate: presence, numericality, uniqueness, confirmation, acceptance, length, format, inclusion, exclusion. **validates_presence_of :name**
- validates :what_name_var, against_what_type_of_validation: value. **validates :name, presence: true**

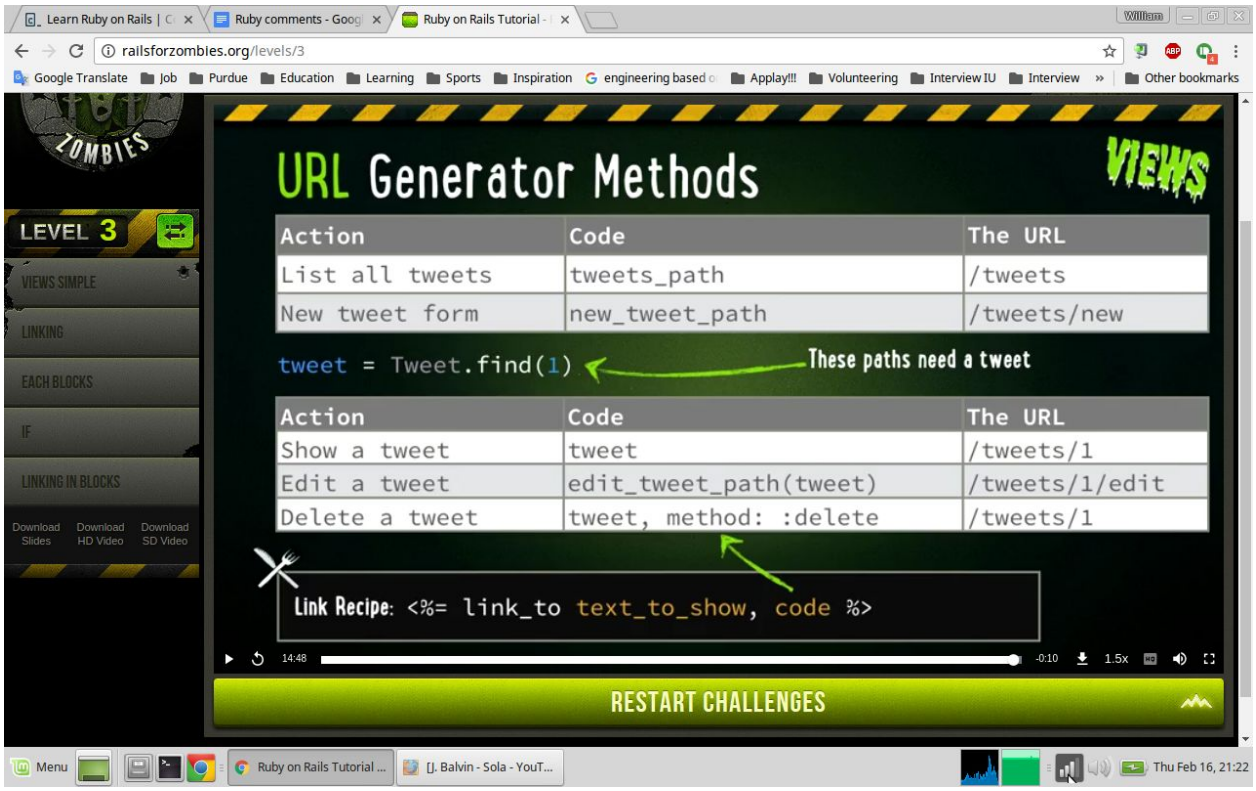
Relational databases

- One element (controlled inside a model) can “**has_many**” other elements of other models
- One element of a model can “**belongs_to**” an element of other model. **belongs_to :zombie**

HTML

<% evaluate %>
<%= evaluate and print %>
To display images use: **<%= image_tag var_name.image %>**

app/view/model_name/some_page.html.erb # has the ruby code per view
app/view/layouts/application.html.erb # has the template use <%= yield %> to execute the models
link_to name_for_the_link, element_for_url, **link_to zombie.name, zombie**, **<%= link_to zombie.name, edit_zombie_path(zombie) %>**
For URLs:



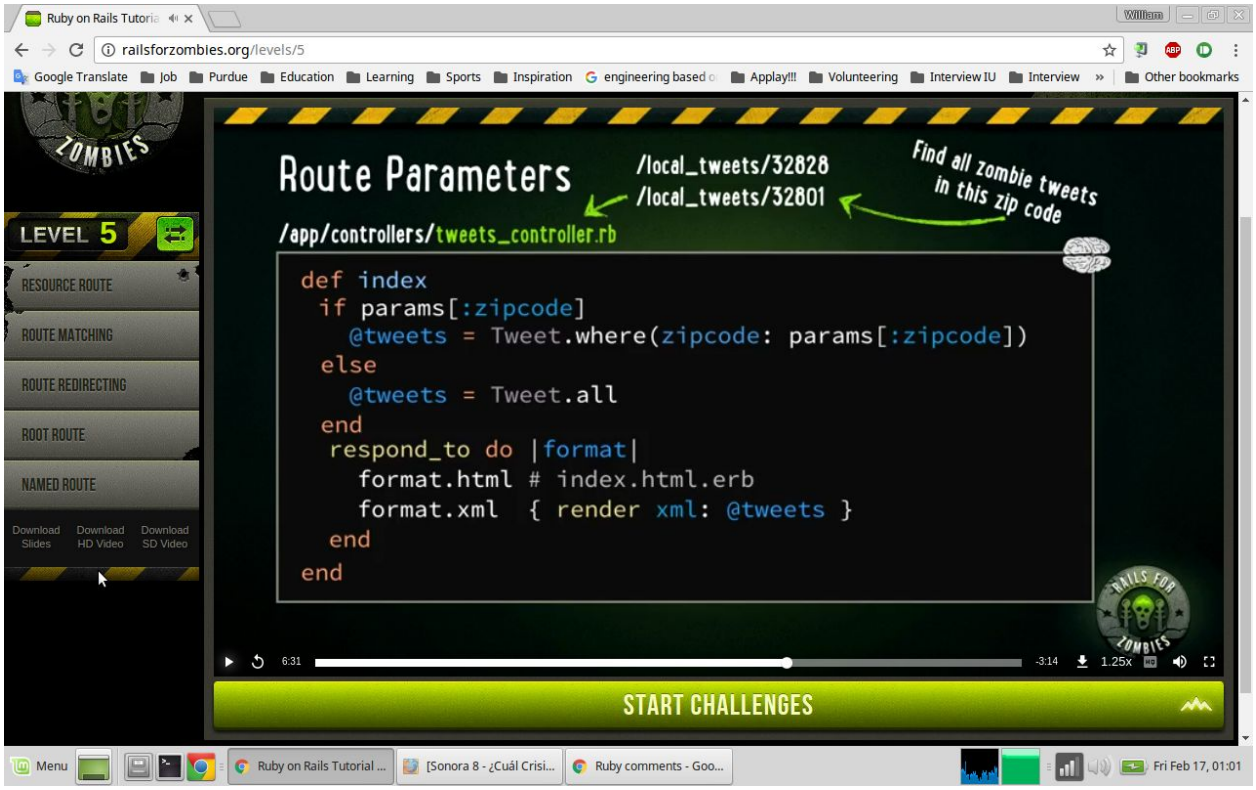
Controllers

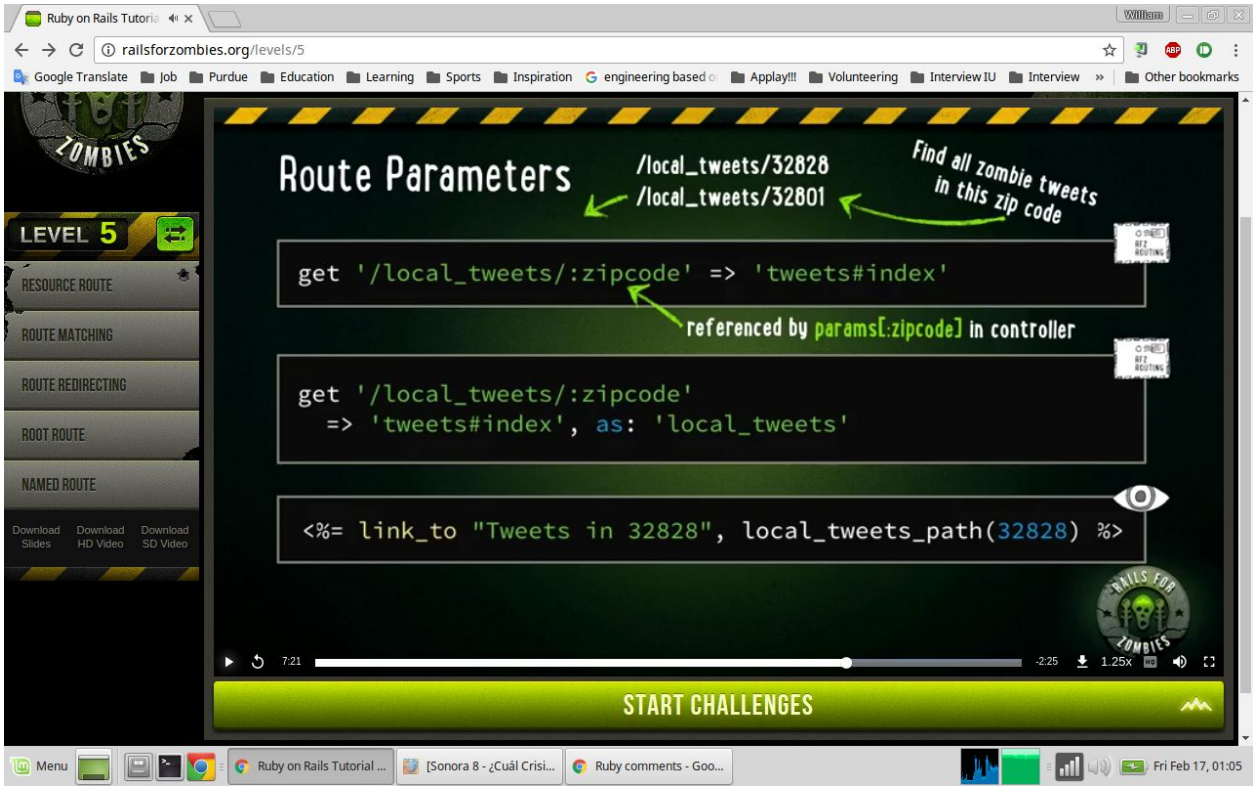
- The methods (also called actions) maps to views.
- To make them render to other view: **render action: 'view_name'**
- To pass parameters to controllers of batabase ids use name_controller.find.(**params[:id]**)
- There are parameters and strong parameters [??] (require(:tweet))

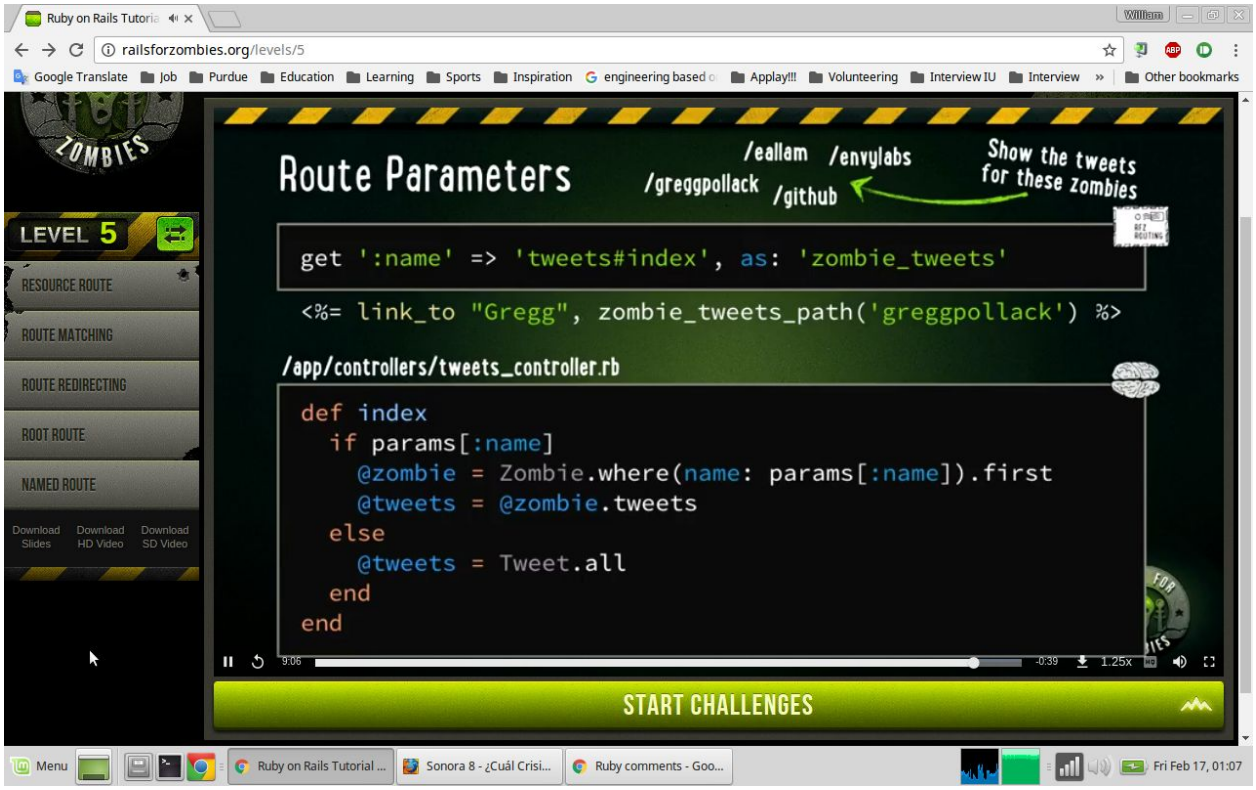
- To create new instances use **@tweet = Tweet.create(params[:tweet])**, for strong parameters use **@tweet=Tweet.create(params.require(:tweet).permit(:status))**
- Json and XML:
  respond_to do |format|
  format.html # show.html.erb
  format.json { render json: @tweet }
  format.xml { render xml: @tweet }
  end
- Session: per user hash, use **session[:zombie_id]**
- Send messages to the user with **flash[:notice] = "Message"**. In the html layouts
  <% if flash[:notice] %>
  <%= flash[:notice] %>
  <% end %>
- Redirect page with **redirect_to(@url_code_path)**, **redirect_to(url_code_path, notice: "Message")**
- To simplify code that different controller's method or actions do use:
  before_action :some_method, only: [:edit, :update, :other_controller_method]
  def some_method
  @some_var = ControllerClass.find(params[:id])
  end

Routes:

- To have the url generator used use **resources :tweets**
- New route **get '/all' => 'tweets#index', as 'name_for_route'**, the path would be **name_for_route_path**
- Redirect route **get '/all' => redirect('/tweets')**
- Root route (initial route) **root to: "controller#action"** ("tweets#index"), the path would be **root_path**. **root to: "zombies#index"**


railsforzombies.org/levels/5
LEVEL 5
RESOURCE ROUTE
ROUTE MATCHING
ROUTE REDIRECTING
ROOT ROUTE
NAMED ROUTE
Download Slides
Download HD Video
Download SD Video
Route Parameters
/local_tweets/32828
/local_tweets/32801
Find all zombie tweets in this zip code
/app/controllers/tweets_controller.rb
def index
if params[:zipcode]
@tweets = Tweet.where(zipcode: params[:zipcode])
else
@tweets = Tweet.all
end
respond_to do |format|
format.html # index.html.erb
format.xml { render xml: @tweets }
end
end
START CHALLENGES


railsforzombies.org/levels/5
LEVEL 5
RESOURCE ROUTE
ROUTE MATCHING
ROUTE REDIRECTING
ROOT ROUTE
NAMED ROUTE
Download Slides
Download HD Video
Download SD Video
Route Parameters
/local_tweets/32828
/local_tweets/32801
Find all zombie tweets in this zip code
get '/local_tweets/:zipcode' => 'tweets#index'
referenced by params[:zipcode] in controller
get '/local_tweets/:zipcode'
=> 'tweets#index', as: 'local_tweets'
<%= link_to "Tweets in 32828", local_tweets_path(32828) %>
START CHALLENGES



Other

To see what is it in the database:
rails console
Books.connection
Now all the methods work: Book.first

To use gems:
Go to https://rubygems.org/gems/devise
Add gem to gemfile
Bundle install
Install gem following instructions from the github Homepage of the gem

rails generate migration add_catagoty_id_to_books category_id:integer

This might be used with nested forms
#@book.build_author #use this for to_one relationship
@book.authors.build() #use this for to_many relationship

BooksRentalApp

Start

- rails new BooksRentalApp
- git init
- git remote
- git add
- git commit
- git fetch
- git merge
- git push

Books

- git branch books_and_authors
- git checkout books_and_authors
- rails generate model Book name:string isbn:string genre:string author:string editorial:string
- rake db:migrate
- rails generate controller Books
- rake routes
- # use gem simple_form
- # use gem bootstrap
- bundle install
- rails generate simple_form:install --bootstrap
- # configure bootstrap
- -------
- # configure controller
- # configure index
- # configure new and the rest of the views
- rake db:seed
- # use rake db:reset in case is needed
- git ….

Authors

- rails generate model Author name:string country:string dob:datetime
- rake db:migrate
- rails generate controller Authors

- # with gem simple_form use: gem 'country_select'
- bundle install
- -------
- # add routes resources :authors
- # configure controller
- # configure index
- # configure new and the rest of the views
- # configure country and dob
- rake db:seed
- # use rake db:reset in case is needed
- git ….

Associations

- Create branch books_belongs_to_authors on github
- git branch books_belongs_to_authors
- git checkout books_belongs_to_authors
- rails generate migration CreateJoinTableAuthorsBooks author book
- # add associations has_and_belongs_to_many
- rails generate migration RemoveAuthorFromBooks author:string # Removes the author attribute from Books, now they have the has_and_belongs_to_many
- rails db:migrate
- # Modified seed to fit this model
- rails de:reset
- # reconfigure views index, show and link between authors and book
- # reconfigure _form for books (impacts new and edit)
- # In Book model change params, add an :author_id =>[ ]
- git add .
- git commit -m “comment”
- git fetch
- git merge origin/books_belongs_to_authors
- git push origin books_belongs_to_authors

Reading list

- Create branch reading_lists on github
- git branch reading_lists
- git checkout reading_lists
- rails generate model ReadingList name:string
- # check the migration file and the model
- rake db:migrate
- rails generate controller ReadingLists

- # Add routes resources :reading_lists
- # configure controller (all views, normally one add the action and then the view)
- # configure associations:
    - # Association with books: has_and_belongs_to_many :books
    - # Association with reading_list: has_and_belongs_to_many :reading_list
    - # In ReadingList model add params for book association :book_ids => []
- rails generate migration CreateJoinTableBooksReadingLists book reading_list
- rails db:migrate
- # Modified seed to fit this model
- rails de:reset
- # configure index
- # configure show
- # configure new
- # create _form
- # configure edit
- # configure delete
- git add .
- git commit -m “comment”
- git fetch
- git merge origin/reading_lists
- git push origin reading_lists

reading_lists was merged with master:

- On GitHub, to merge reading_lists with master:
    - Compare and create a pull request
    - Merge (solve the pull request)
- git checkout master
- git fetch
- git merge origin/master