# Notes for HackerYou Summer Academy

## GitHub

### Pushing Projects onto a GitHub Repository

In the command line, enter the following commands

**git init**

#initializes the folder you are currently in for git

**git add .**

#adds all the files located in the folder (‘.’ 🡪 selects all, but you can add files one at a time if you want)

**git status**

#not necessary, but checks the status of what has changed

**git commit –m “*your message here”***

#commits the changes you have made

**git log**

#good practice to check what you have done so far

**git remote add origin *<insert link here>***

#used to remotely connect your files offline to your repository online

**git push -u origin master**

#pushes local repo to server

### Forking a Repository

**Click ‘Fork’ on the repository you want on Github**

#forks over the repo

**git clone *<link to your forked repository>***

#creates a copy on your local repository

**cd clone**

#enters folder

**git remote add upstream *<link to your original>***

#assigns the original repository to a remote called “upstream”

**git fetch upstream**

#Pulls in changes not present in your local repository, without modifying your files

## Agile Development Notes

### Models, Views, and Controllers (MVC)

* An architecture used to develop interactive applications
* **Model:** 
  + Responsible for maintaining the state of the application
  + Is the only item that interacts with the database
  + Can also be used to code validation tests
    - Very useful since it is the gatekeeper of the database
    - Thus, before you store anything into the database, the model can check if the data is valid
    - Otherwise, you may end up storing invalid data
  + Rails does not store anything
  + Always saves to the database
  + Models are smart, they do your logic
  + Always have to tell it what type of data
    - Migrations
      * Every migration is run one time, tells the database to change their schema
      * rake db migrate
        + Causes it to update the database
      * Migrations can be run up and down so you can change it
        + Before it was basically like a massive excel file and if you changed one data scheme, you could not undo it (ie have to reinsert it and screws up the order)
      * Ask about AddDescriptionToProducts
    - Rake
      * Runs code within your app without having to run it
      * rake –T
      * db (database)
    - Validations
      * presence: true 🡪 checks if it is there
    - Callbacks
      * Triggers when stuff happens
      * Ex: e-mail a user when an order has been shipped
      * Ex: class < Order::Base
        + send\_email customer.e-mail “Your order has been placed!”
      * Finite set of things you can do with the model
        + You don’t need to create your own
    - Custom Logic/business logic
* **View:**
  + Generating the user interface based on data in the model
  + Can be used with HTML, CSS, and ERB
  + Never handles data, only displays
  + Contain only html and ERb files
  + Rendered by the browser
  + Views do not do ANY logic
    - Only some, like loops and conditionals
  + Output vs. logic
    - <%= Time.now %>
    - <% %>
  + ***Ask question about how the browser sends information regarding Ruby***
* **Controller:**
  + Orchestrates the application
  + All user input goes through the controller
  + It interacts with the model and then displays an updated view for the user
  + Access control
  + Handling errors
  + resources :products
    - Opens up all of the links quickly
    - rake routes 🡪 shows all the possible links your website can open
    - Can also see this in the errors

****

### File Configurations

* Convention over configuration
* You don’t need to tell Rails where to look for shit, it knows as long as you place them in the right folder

### Cool Methods in Ruby

self.send(input)

dynamically sends itself questions? \*\*look this up \*\*

## Lecture / ROR Notes

### Scaffolds

In a ROR app, we need to generate a lot of things to make it work. In particular, we need to create:

* A database table to store data
* Rails model that lets our application use that table
* Views to make up the user interface
* Controllers to orchestrate the application

Luckily, Rails has the ability to create scaffolds, which sets up ALL of these things, along with the appropriate connections and links between them. This is very useful because it saves you a lot of time rather than having to do it manually.

To create with a scaffold, follow the syntax:

*rails generate scaffold Product* \

title:string description:text image\_url:string price:decimal

Using this syntax inputs code into your migration. Looking inside db/migrate/…\_create\_products.rb, it will have the following code.

class CreateProducts < ActiveRecord::Migration

def change

create\_table :products do |t|

t.string :title

t.text :description

t.string :image\_url

t.decimal :price, precision: 8, scale: 2

t.timestamps

end

end

end

In other words, entering title:string… puts the different variables you want into the migration that you want written to the database. This will create a table in the database that looks kind of like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Title | Description | Image\_Url | Price | Timestamp |

### Creating an Application w/o a Scaffold

Sometimes when we are creating applications, it is actually more useful to build ***without*** a scaffold. This is because a scaffold sometimes adds a lot of unnecessary things in your application. Furthermore, it links a whole bunch of things together and as you know, it is much more difficult to debug if you look after the fact.

Consequently, it is useful to know how to build without one sometimes so you can keep track of everything.

**Step 1: Generate a new project and enter the folder**

*rails new depot*

*cd depot*

**Step 2: Generate a controller**

*rails generate controller products*

*important to use “S” at the end*

In Rails, a model is automatically mapped to a database table that is the PLURAL form of the model’s class

**Step 3: Make views and actions for what you need**

Create all the views manually by inserting them in the appropriate folders in your file. You will need to account for the following:

* html.erb files in Views
* Insert all code in it that is needed
* Do not forget the json files for your shopping cart app!

JSON for our purposes:

In our files here, JSON is used to extract data from our database tables. In this case, we use JSON commands to interact with the SQL database in order to get data like :title, :description, and :image\_url.

* JSON stands for Javascript Object Notation. It is a text-based open standard design for reading data
* Scripting language based on Javascript, used to represent simple arrays and data stuctures
* It is actually language-INDEPENDENT, having parses available across many programming languages

**Step 4: Generate your model**

*Rails g model Product title:string description:text…*

*rake db:migrate*

**Step 5: Setup your controller**

Here, you need to setup your controller so that you can get your database information. Thus, you setup commands in your controller to interact with the model, which in turn interacts with the database.

Remember MVC!

User request 🡪 Controller 🡪 Model 🡪 Database 🡪 Model 🡪 Controller 🡪 View

For this example, we setup an index, new, edit, create, update, destroy

**Step 6: Setup routes**

config/routes.rb

*resources :products*

**Step 7: You’re done!**

### Validations

Sometimes you want your program to prevent data from being stored if it is not valid. Validations in ROR are great way of doing just that! We normally place validations in the Model because it is the gateway between the database and the controller.

For the store app, you can validate if each product has the required parameters before entering into the database by using:

*validates :title, :description, :image\_url, :price, presence: true*

We also want to validate that the price is never $0.00. This can be done using the ‘numericality’ method

*validates :price, numericality: { greater\_than\_or\_equal\_to: 0.01 }*

Also want to validate that each title is unique.

*validates :title, uniqueness: true*

Lastly, you want to validate that the image is a valid URL.

*validate :image\_url, allow\_blank: true, format: {*

*with %r{\.(gif|jpg|png)\Z}i,*

*message: ‘must be a URL for GIF, JPG, or PNG image.’*

*}*

Note that allow\_blank is used to avoid multiple errors if the image is left blank

### Testing

Testing is a very important concept in programming because you want to make sure your program works the way it is intended to. You want to make sure you can account for change, modularity, failure, and efficiency. Also, when you make changes to the program, testing is great because it checks whether or not your changes have done anything.

You should always write tests first before you program. That way when you code, you are only coding just enough to do what you want the program to do. This is called Test Driven Development (TDD).

When you create a new project using *rails new <project\_name>*, rails automatically generates a test infrastructure for everything (particularly, all your controllers and models).

When writing tests, you use variations of *assert*. Basically, you are asserting something to be true that you expect to come out of your program. If it fails, then it means your program isn’t doing what you intended it to do.

* Using skip inside the method
  + Test fails come one at a time as to not overwhelm the user
  + Tells you exactly which test line failed

You may also want to write tests that purposely enter in bad user input to make sure that your program can handle it. For example, in the shopping cart app, you can choose to write a test for bad image\_urls.

Good tests read like a story, be as descriptive as possible, but be DRY.

Extra resources: <http://guides.rubyonrails.org/testing.html>

#### Fixtures

In Rails, fixtures is just an area for sample data for testing.

From documentation:

*“Fixtures is a fancy word for sample data. Fixtures allow you to populate your testing database with predefined data before your tests run. Fixtures are database independent written in YAML. There is one file per model.”*

Also, when entering data into fixtures, it is IMPERATIVE that the formatting be exactly the same. This is because when the database needs to be able to read it. Also, make sure that all the required properties are there because a mismatch with the database columns may cause a hard-to-track-down exception.

Make sure to add *fixtures :name\_of\_yml\_file* into your model test. This makes sure that the data in the fixture is loaded every time you run the test.

Note that most of the scaffolding that Rails generates doesn’t contain calls to the fixtures method. That’s because the default for tests is to load all fixtures before running the test. Because that default is generally the one you want, there usually isn’t any need to change it***. Once again, conventions are used to eliminate the need for unnecessary configuration.***

Now the fixture data is being used, what you should actually know is that Rails creates a method for each fixture it loads into a test. Furthermore, the name of the method EXACTLY matches that of the fixture data. You can now use this method to load the fixture data simply by passing its name.

Ex:

test "product is not valid without a unique title - i18n" do

product = Product.new(title: products(:ruby).title, description: "yyy", price: 1, image\_url: "fred.gif")

assert product.invalid?

*assert\_equal [I18n.translate('errors.messages.taken')], product.errors[:title]*

end

Also note that if you do not want to use a hard-coded string, use the above code highlighted.

**Later stuff to cover**

* Check out minitest
* Regular Expressions
  + Used to match a pattern against strings
  + Ex: \d will capture all of the digits
* Testing

### Caching

As it stands, each request made to the website would require a fetch from the database and a render for each product. Since the catalog does not change very often, there is no need to start from scratch each time a request is made.

Enter caching. We can modify our configuration for our development environment (config/environments/development.rb) to turn on caching. This is good because it speeds up the process for future requests.

From Wikipedia:

*In computer science, a cache is a component that transparently stores data so that future requests for that data can be served faster.*

However, this is bad for development environments because any changes we make will not appear right away. Consequently, if you are just developing, make sure to set the caching to *false*.

Also, we only need to re-render if any one product changed. To return the most recently updated product:

app/models/product.rb

**def** self.latest

Product.order(:updated\_at).last

**End**

To update only the ones that changed:

* <% cache [*'store'*, Product.latest] **do** %>

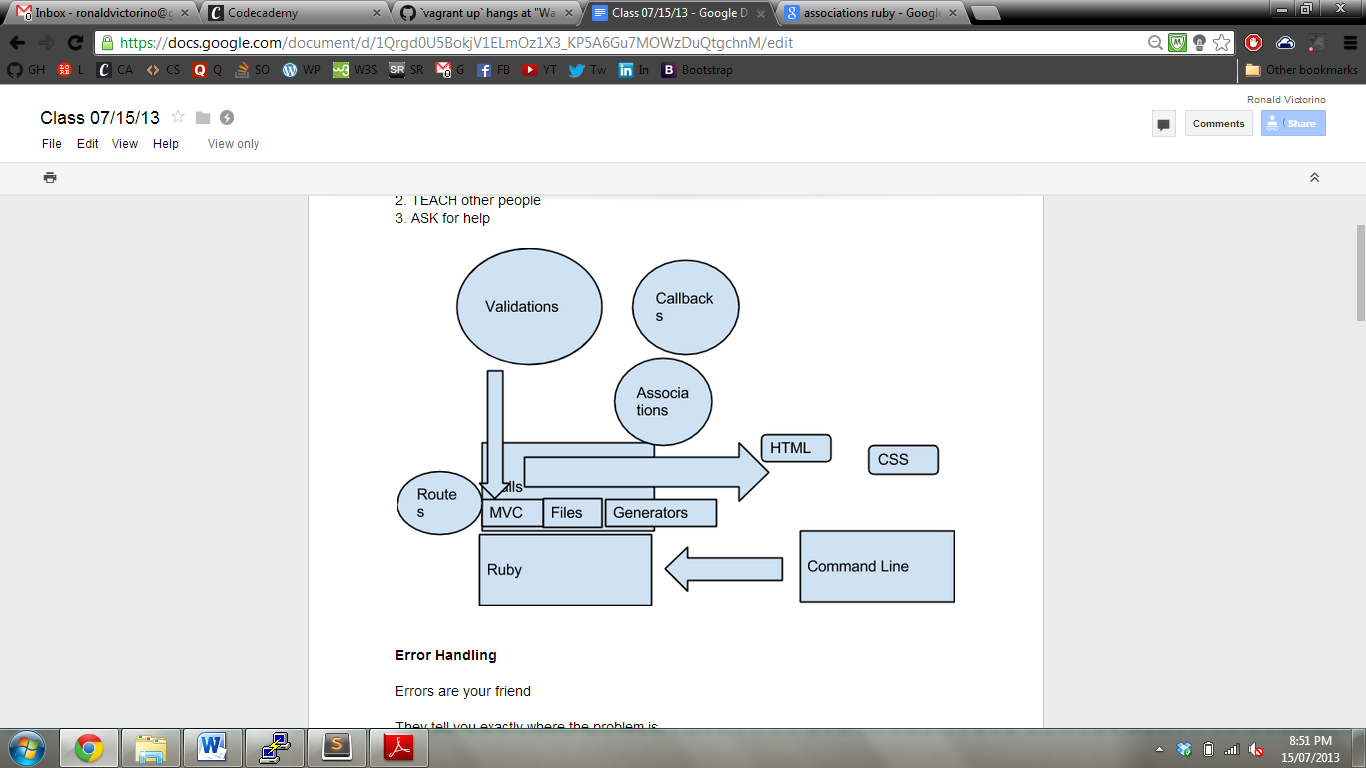
<% @products.each **do** |product| %>

* <% cache [*'entry'*, product] **do** %>

Don’t quite understand caching. Ask questions about how it works!

### Errors

* Errors are your friend because they tell you where the problem is
* Make sure you are familiar with the code written
* No route matches 🡪 not the view
  + A result of no route being identified in routes
* Debugger
  + Enter the word *“debugger”* into any part of your code
  + Will stop when it is first accessed
    - next = go to the next line
    - step = go to the source of this function
    - continue = stop debugging and keep running the program
    - p = print the value of an object or variable
    - e = run some ruby code (put code after) – stands for evaluate



class ProductTest < ActiveSupport::TestCase

def test\_the\_truth

assert true

end

end

These two are the same, just different formatting

class ProductTest < ActiveSupport::TestCase

test "the truth" do

assert true

end

end

* Use rake test:models
  + Use instead of ruby test.rb
  + Using ruby adds a whole bunch of shit that you don’t need
* i18n 🡪 internalization
  + Used to translate things
* Testing takes a long time because it’s constantly clearing a database, reinitializing it, then jamming it into the code

### Associations

* Unique functionality on top of models
* Nothing to do with validations, just another feature
* **Expresses a relationship between two models**
* Ex: blog post vs. comments
* You will want to keep the associations in the database
  + Have the comments point to the blog post, rather than vice versa
  + Easier
* t.references :order is the same as t.integer :order\_id
* Concerns
  + New to rails 4
  + Great for creating code that you are going to be sharing across all things
  + Every user has their own session from when they visit the site
  + Session is a hash
  + File name is important
* Rescue
  + Allows you to deal with an error
  + Ex: rescue ActiveRecord::RecordNotFound
    - @cart = cart.create
    - Session[:cart\_id] [=@cart.id](mailto:=@cart.id)
* Rails make it much easier for everyone to add line items to a database
* 3 steps: look in textbook! t.references, has\_many,

## Design Notes

### Twitter Bootstrap

#### Grid System

* Twitter Bootstrap uses a 960 grid statement
  + Total width of the page is 960px with 20px margins on the left and right
* Automatically has a container that spans the whole webpage
  + The containers have a -20px start already
* Uses 16 columns (for v1.4)
  + span1 all the way up to span16 with 20px between each
  + can actually use span-one-third as well
* With v2.0, they use a 12 column grid
* There is an automatic 60px padding at the top for a toolbar

#### Media Grid

* In Twitter Bootstrap, there exists a class called media-grid used in an unordered list
  + Automatically places images in a container
  + Identified by <li> tags
  + Ex: <ul class = ‘media-grid’><li><img></li>
  + The only input needed is the size of the image
* Placeholder.it is a website that shows you how large the image will be prior to changing the width and height of an image
  + Can type in right in the url (placeholder.it/widthxheight) and it will return you the image

#### Making Custom CSS Sheets

* If you want to add on to the Bootstrap style sheet, make a new file and use import
  + @import(url); 🡪lets our CSS file inherit all of Bootstrap
  + This allows you to add more classes and IDs to the existing bootstrap CSS without changing bootstrap by accident
* With v2.0 you can customize different settings before you downloa

#### Random Notes

* Use pull-right/left to float an element right/left
* To embed videos, go to Youtube file and click that way
  + Can also change width and such there and they will give you the appropriate resizing
  + Youtube videos use iframes
* Changelog is better now
  + Shows differences between versions
* Use the components section! It will be your best friend for design!
  + Contains all of the different things you may need to create

#### Creating a Drop Down List

* In the nav-collapse container, it will collapse as you resize the screen
  + It will also change it into a drop down menu
  + We have to initialize that menu
* To create the dropdown menu and search for your toolbar:
  + Dropdown menu
    - Inside the nav-collapse container, each link is added as a <li>
    - Place a ‘nav’ ul inside it
    - Make a new <li>, with a “dropdown” class
      * Also put a <a href = #></a>
    - Inside the <li>, create a ul called “dropdown-menu”
  + Search
    - Copy code from video
    - Uses “search-query”
* To actually drop down, you will need javascript

#### Tips and Tricks

* Debugging can be done right from the website
  + Click inspect element
  + Sees all the html and css rules for each element
  + Use magnifying glass to look over image
  + Shift + up /down while highlighting the px to grow/shrink the element
  + Click the middle button to check all the pseudo classes (ex: hover, focus, etc.)
  + Can change the elements right on the page to look at it
  + ***Very useful for debugging because you can see EXACTLY what is happening***
  + You can only use the web inspector to look at and change, but not save
    - Have to go into Sublime to change it later