

Table of Contents

Introduction & Syllabus	0
Welcome	0.1
Student Holidays	0.2
Homework Submission	0.3
Class Library	0.4
Required Software	0.5
Course Outline	0.6
All Videos/Source Code	0.7
Front End Fundamentals	1
Introduction	1.1
Mac Cheat Sheet	1.1.1
Terminal Cheat Sheet	1.1.2
Workshop: Star Wars Command Line	1.1.3
Hello World	1.2
Variables	1.2.1
Data Types	1.2.2
Control Flow	1.2.3
String Concatenation	1.2.4
Arrays	1.2.5
Loops	1.2.6
Syntax	1.2.7
Homework: Markdown and Readmes	1.2.8
Functions, Loops, Objects	1.3
Homework	1.3.1
Drawing on the Web	1.4
HTML and CSS	1.4.1
The DOM	1.4.2

Converting Pixels to EM	1.4.3
Canvas	1.4.4
Second Pass Friday	1.5
Weekend Practice	1.5.1
Bonus Weekend Practice	1.5.2
Object Oriented JS & jQuery	2
Objects and CSS	2.1
Morning Review	2.1.1
Object Methods	2.1.2
this, hasOwnProperty, .keys	2.1.3
HTML & CSS Layouts	2.1.4
Homework: Layout Challenges	2.1.5
Wireframe to Production	2.2
Morning Exercise	2.2.1
Sass & CSS Variables	2.2.2
Colour Theory 101	2.2.3
Fonts	2.2.4
Style Guides	2.2.5
Homework	2.2.6
Creating User Interface Components	2.3
Selectors	2.3.1
DOM Events	2.3.2
Components 101	2.3.3
Constructors	2.3.4
Component Constructor Boilerplate	2.3.5
Videos	2.3.6
Homework	2.3.7
Write Less & Do More avec jQuery	2.4
Component Recap	2.4.1
jQuery	2.4.2

jQuery Boilerplate	2.4.3
document.ready	2.4.4
Homework	2.4.5
Second Pass Friday	2.5
Advanced Front End	3
Ajax	3.1
Ajax in Depth	3.2
Fellowship Solution	3.3
Project #1: The Game	4
Requesting Feedback	4.1
Ruby & Sinatra	5
Introduction to Ruby	5.1
Front-End Recap Quiz	5.1.1
Dev Environment	5.1.2
Strings, Arrays, & Hashes	5.1.3
Detecting Ruby Types	5.1.4
Video: Pry	5.1.5
Conditionals	5.1.6
Loops	5.1.7
Enumeration	5.1.8
Methods	5.1.9
Single vs Double Quotes	5.1.10
Practice: Rups!	5.1.11
Homework: Rups! 2.0	5.1.12
Everything is an Object	5.2
Rups Recap/Examples	5.2.1
.each	5.2.2
Movie Object	5.2.3
Methods, Again	5.2.4
Classes	5.2.5

Stephen's Class Notes	5.2.6
Inheritance	5.2.7
Homework: Classes	5.2.8
Introduction to Sinatra	5.3
Our First Server	5.3.1
Lab: Driving with Sinatra	5.3.2
Lab: Sinatra On Your Own	5.3.3
ERB (Embedded Ruby) Views	5.3.4
Homework: Animals!	5.3.5
Bonus Homework	5.3.6
Second Pass Friday	5.4
JSON	5.4.1
Ruby Q&A	5.4.2
Sinatra Second Pass	5.4.3
Deploying to Heroku	5.4.4
Homework	5.4.5
MVC with Sinatra	6
Databases	6.1
Introduction to Databases	6.1.1
Entity Relationship Diagrams	6.1.2
Postgres Data Types	6.1.3
Lab: ERDs!	6.1.4
Structured Query Language	6.1.5
Postgres Cheatsheet	6.1.6
SQL Examples	6.1.7
Lab: Twitter Migrations	6.1.8
Homework	6.1.9
'Physical' Servers	6.2
Building a Web Server	6.2.1
Digital Ocean	6.2.2

Server Building in a Nutshell	6.2.3
Your App on a server	6.2.4
Homework: FileReader	6.2.5
Bonus Homework: Dragon Latin	6.2.6
Models, ActiveRecord, & CRUD	6.3
Shopping List Wireframes	6.3.1
Models with ActiveRecord	6.3.2
CRUD with ActiveRecord	6.3.3
ERB in Depth	6.3.4
Rake & Migrations	6.3.5
Lab: Pair Programming	6.3.6
Homework	6.3.7
Models and Controllers	6.4
Models and SQL Q&A	6.4.1
Bundling it Together	6.4.2
Guide: CRUD from Scratch	6.4.3
CRUD Controllers	6.4.4
Relationship Advice	6.5
ActiveRecord Cheatsheet	6.5.1
Homework	6.5.2
Security 101	6.6
Server Side Authentication	6.6.1
Account Models & BCrypt	6.6.2
Tux: Console for Sinatra	6.6.3
Account Controller	6.6.4
Rake	6.6.5
Second Pass Friday	6.7
Morning Exercise w/Divvy	6.7.1
Mapping Routes	6.7.2
Classes, Revisited	6.7.3

A Class in Java	6.7.4
JSONReader Class	6.7.5
BCrypt Salt/Hashing	6.7.6
Application Controller Architecture	6.7.7
Digital Ocean Production Server Videos	6.7.8
Bootstrap Self-Practice	6.7.9
Many:Many & 1:Many Models with ActiveRecord	6.7.10
Sending Emails with Mandrill	6.7.11
 Project #2: Full Stack	 7
Week Schedule	7.1
Project Teams	7.2
Project Scope	7.3
Initiation Survey	7.4
Scope Survey	7.5
 Full Stack Node	 8
From Ruby to Javascript	8.1
Intro to Test Driven Development	8.1.1
Ruby	8.1.2
Sinatra	8.1.3
Javascript	8.1.4
Lab: Javascript Recap, Reps, & Resilience	8.1.5
Homework: Setup Node Environment	8.1.6
 Introduction to Node	 8.2
Introduction to Jasmine	8.2.1
Intro to Node.js	8.2.2
Node File Structure	8.2.3
Node Event Loop	8.2.4
Node Server Examples	8.2.5
Sinatra vs Node	8.2.6
Node Modules	8.2.7

Exercise: Modules	8.2.8
Homework: Requesting with Node	8.2.9
MongoDB & the Wonders of NoSQL	8.3
npm Installation Notes	8.3.1
Installing MongoDB	8.3.2
Mongo 101 & Importing JSON	8.3.3
Mongoose Models, Schemas, & CRUD	8.3.4
Model Example	8.3.5
Mongoose + Models with Node	8.3.6
Express.js	8.4
Morning Q&A	8.4.1
Getting Started w/Express	8.4.2
Express Guide	8.4.3
RESTful API Example	8.4.4
Lab: Express Generate	8.4.5
Lab: Express Reps	8.4.6
OOJS Recap & 2nd Pass	8.5
Prototype	8.5.1
Full Stack from Scratch	8.5.2
ES6 Classes	8.5.3
Ajax on a View	8.5.4
This Week's Videos	8.5.5
Homework	8.5.6
Backbone.js	9
Introduction to Backbone.js	9.1
CDN Quick-Start	9.1.1
JSON Recap	9.1.2
Models	9.1.3
Views	9.1.4
Underscore Templates	9.1.5

Binding Backbone to an API	9.1.6
Videos	9.1.7
Homework	9.1.8
Backbone Second Pass	9.2
Models	9.2.1
Collections	9.2.2
Events	9.2.3
Views	9.2.4
Underscore Templates	9.2.5
Ruby + Backbone	9.2.6
Authentication with Passport	9.2.7
Swimlan Guide to Backbone	9.2.8
Homework	9.2.9
Project #3	10
Project Scope	10.1
Scope Survey	10.2
Initiation Survey	10.3
Teams	10.4
Week Schedule & Deliverables	10.5
Ruby on Rails	11
Getting Started	11.1
Rails Cheat Sheet	11.2
Intro to Rails	11.3
Video	11.3.1
Controllers and Routes	11.3.2
Views and Layouts	11.3.3
Validations and API's	11.4
Validations	11.4.1
Making Rails API	11.4.2
CRUD and Auth	11.5

CRUD	11.5.1
Authentication	11.5.2
Angular & React.js	12
Angular.js 1.x	12.1
Introduction to Angular	12.1.1
Controllers	12.1.2
Binding to a REST API	12.1.3
Capstone Project	13

WDI Chicago #3: *Dragons*



Welcome!

Please use the navigation bar to the left to browse our cohort's *living syllabus/textbook*.

Your Instructors



James Traver
(code-for-coffee)



Adriana Castaneda
(alcastaneda)



Jim Haff (jimbojones1)



Welcome from your instructors

Hi there! You've no doubt stumbled upon our class Gitbook. This book will be our **living syllabus**. It will grow and become a true study guide throughout the duration of your cohort.

Pre-Work

WDI requires for you to complete <https://fundamentals.generalassemb.ly> prior to the first day of WDI. **If you have not completed this by day one, speak to your producer or instructor immediately.**

Recommend Reading

Prior to starting the course, we highly encourage students to take a look at Shaye Howe's (product manager at Belly) Learn HTML & CSS book (it is available for [free here](#)). Try to read the first three chapters prior to starting class; however read as much as you have time for.

Development Environment

WDI has a strict class policy of only supporting Mac OS. Because we teach Ruby and Node, we cannot allow Windows machines into the class as many Ruby libraries will **not work** on Windows. Linux machines may be considered on a case-by-case basis if a student has a strong knowledge of the Linux command line however the instructional staff **will not** support those machines.

Student Holidays & Days Off

- **Thanksgiving Observed:** Wed, Thurs, Fri Nov. 25th through 27th
- **Holiday Break:** Wed, Thurs, Fri Dec. 23rd through 25th
- **Holiday Break Continued:** Mon through Friday, Dec. 28th through January 1st
- **Martin Luther King Day:** Monday Jan. 18th

Ferris Bueller's Day Off

During the WDI Cohort we have two spontaneous days to take off! These will be decided by the instructors to take off (typically during long stretches with no time off). These will be used at our discretion to take days off and students will be notified ahead of time.

Homework Submission

1. Browse to `dragons/wdi_chi_dragons`
2. `git add .` all of your changes to the Git purgatory.
3. `git commit -m "your commit message"` to commit your changes from the Git purgatory into your branch.
4. `git pull origin master` to gather any changes from the base repository
5. `git push upstream master` to push your changes to your own fork!
6. Inside of your fork, create a new **pull request**.
7. Submit the pull request for your instructors to check!

That's all!

Homework Submission Deadline

Homework must be submitted via pull request no later than Midnight CST. Any work submitted past this deadline will *not* be counted.

Class Library

As a student and alumni of GA Chicago, you will have access to our campus library. Please read the onboarding documentation that you were provided with on orientation for more information.

Check books in/out here using Github: <http://ga-chicago-library.herokuapp.com>

Required Software

WDI requires specific software to use in class. We do this so each student is using the same development environment as each other. Please install the following software during orientation (or prior to class start).

If you run into any problems installing this software, please alert an instructor or your producer.

XCode

- **What:** A set of development tools created by Apple that allow for users to write applications.
- **Why:** WDI requires that we
- **Instructions:** On newer versions of OS X, search the Mac App Store for **XCode** and download/install it (it is free). On older versions that do not have the Mac App Store, you will need to create a *free* Apple account and download it at <https://developer.apple.com/xcode/downloads/>. Once you install XCode, please open it and accept the EULA (feel free to close the app once you complete this).

Atom Text Editor

- **What:** A modern day, open source text editor.
- **Why:** It was created by the team at Github, is open source, it is free, and is a fantastic text editor for multiple languages.
- **Instructions:** Visit <https://atom.io/> to download Atom. Drag/drop the application to your Mac application folder (and pin it to your dock). We use this every single day in WDI.

Google Chrome

- **What:** A web browser.
- **Why:** We rely on specific Google Chrome extensions in WDI and it provides the best toolset for developers in a web browser given the learning curve. The tools are stable and consistent between Windows, Mac, and Linux so your core environment never changes.
- **Instructions:** Use Safari to download Google Chrome at

<https://www.google.com/chrome/browser/desktop/>

Homebrew

- **What:** A package manager for command line tools on OS X.
- **Why:** It is the easiest package manager to use on OS X for newbies to get started with.
- **Instructions:** Visit <http://brew.sh/> and follow the instructions on the website. It will ask you to copy/paste a command to paste and run in **Terminal.app**. That's it!

Slack Chat

- **What:** A modern day IRC-like chat client.
- **Why:** GA as a company uses Slack to communicate with staff and students.
- **Instructions:** Browse to <https://slack.com/apps> and download/install Slack. Sign in to the **ga-students** team that you have been invited to.

Required Chrome Extensions

Postman

- **What:** Software that allows you to communicate directly with servers.
- **Why:** It is arguably the best app that handles this with a GUI.
- **Instructions:** Browse to <https://www.getpostman.com/> and download the Chrome Extension.

MDN Search

- **What:** An extension that allows you to search the Mozilla Developer Network through the Chrome Omnibox.
- **Why:** It is a great resource. Also, W3Schools sucks (and any professional in this industry will poke fun at you if you use it).
- **Instructions:** Visit <https://chrome.google.com/webstore/detail/mdn-search/ffpifaemeofjmncjdbegmbpcdaemkeoc> and install the extension.

Tape CSS

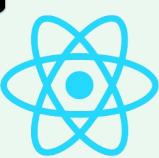
- **What:** A graphical extension that helps measure HTML elements on the page.

- **Why:** You don't want your design and websites to look bad, do you?
- **Instructions:** Visit
<https://chrome.google.com/webstore/detail/tape/jmfleijdbicilomppnomcbkcgidbefb>
and install the extension.

JSON View

- **What:** An extension that makes JSON look pretty.
- **Why:** To easily navigate JSON files using a tree-view.
- **Instructions:** Visit
<https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc?hl=en> and install the extension.

Course Outline

Foundations of the Web	Ruby, MVC, and Sinatra	APIs with Node.js	Advanced Frameworks
   	   	   	  

- **Section 1: Foundations of the Web**
 - HTML
 - CSS
 - Javascript
- **Section 2: Ruby, MVC, and Sinatra**
 - Ruby
 - MVC
 - Sinatra
 - SQL
 - Servers
- **Section 3: APIs with Node.js**
 - Node.js
 - Express.js
 - MongoDB
 - Backbone.js
- **Section 4: Advanced Frameworks**
 - Ruby on Rails
 - React.js

Weekly Schedule Breakdown

Our weekly schedule is available for viewing here. It is broken down to display what day-to-day looks like in WDI.

	Monday	Tuesday	Wednesday	Thursday	Friday
9am	9 - 9:40am: Morning Recap	9 - 9:40am: Morning Recap	9 - 9:40am: Morning Recap	9 - 9:40am: Morning Recap	9 - 9:40am: Morning Recap
10am	9:40am - 9:50: Standup	9:40am - 9:50: Standup	9:40am - 9:50: Standup	9:40am - 9:50: Standup	9:40am - 9:50: Standup
11am	9:50am -10: Break 10am - 11:20am Block 1	9:50am -10: Break 10am - 11:20am Block 1	9:50am -10: Break 10am - 11:20am Block 1	9:50am -10: Break 10am - 11:20am Block 1	9:50am -10: Break 10am - 11:20am Block 1
12pm	11:20am - 11:30: Break 11:30am - 12:50 Block 2	11:20am - 11:30: Break 11:30am - 12:50 Block 2	11:20am - 11:30: Break 11:30am - 12:50 Block 2	11:20am - 11:30: Break 11:30am - 12:50 Block 2	11:20am - 11:30: Break 11:30am - 12:50 Block 2
1pm					
2pm	12:50pm - 2pm Lunch	12:50pm - 2pm Lunch	12:50pm - 2pm Lunch	12:50pm - 2pm Lunch	12:50pm - 2pm Lunch
3pm	2pm - 3:20pm Block 3	2pm - 3:20pm Block 3	2pm - 3:20pm Block 3	2pm - 3:20pm Block 3	
4pm	3:20pm - 3:30pm: Break	3:20pm - 3:30pm: Break	3:20pm - 3:30pm: Break	3:20pm - 3:30pm: Break	
5pm	3:30pm - 6:00pm Workshop	3:30pm - 6:00pm Workshop	3:30pm - 4:30pm - Outcomes 4:30pm to 6:00pm - Workshop	3:30pm - 6:00pm Workshop	2pm - 5:00pm Workshop & Weekend Introduction
6pm					

Course Videos

It is not uncommon for the instructional staff to record screencasts in the middle of the class. All of those screencasts will be linked here along with sample code links.

Videos

- Javascript 101 Recap: <https://www.youtube.com/watch?v=24dV8xpEljU>
- Selector Recap: <https://youtu.be/dlRXcwgcigQ>
- User Interface Component Recap: <https://youtu.be/bp6HzzRtkjc>
- Component Constructors: <https://www.youtube.com/watch?v=CpVz9Z7mqQA>

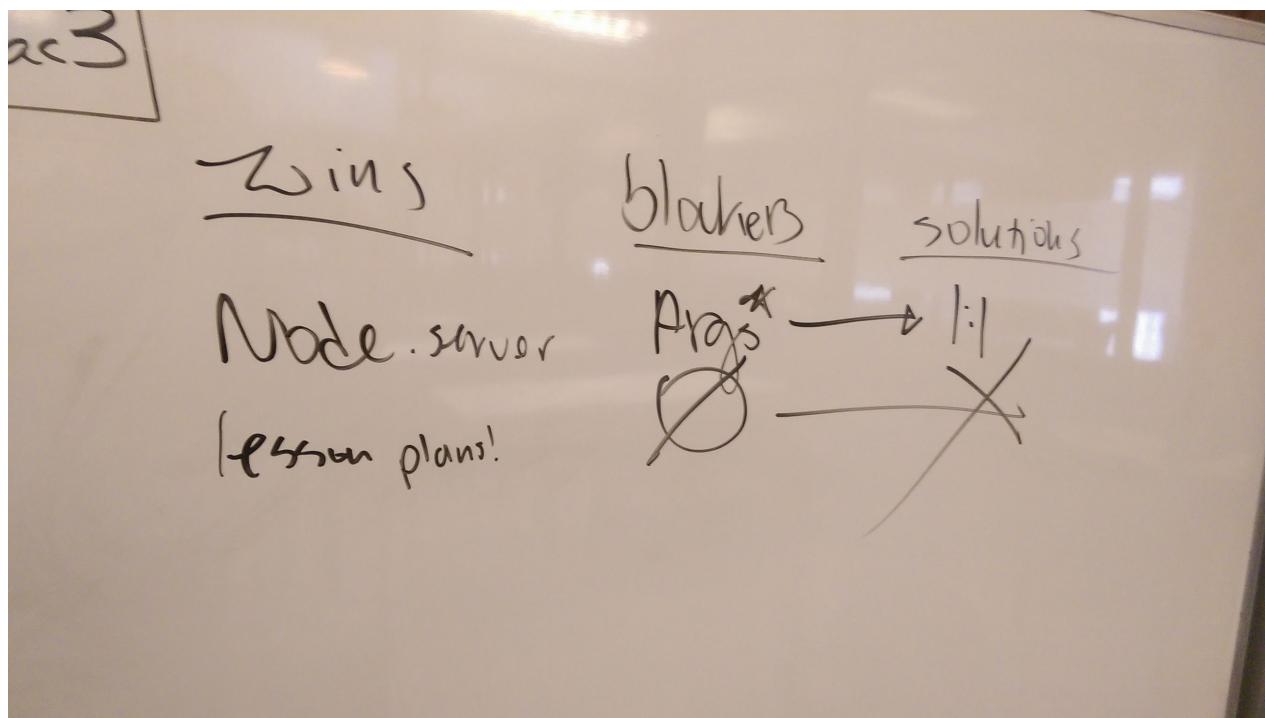
Source Code

- HTML & CSS Boilerplate: https://github.com/code-for-coffee/html_css_boilerplategit
- Component Constructors: <https://jsfiddle.net/qf8tsy3y/1/>

Part 1: Front End Fundamentals

1.1 Introduction to WDI / Web Development

Example Standup



Culture

- Help each other; you're in this together!
- Build relationships! You never know who may help you land a job.

Lab: Transforming your Laptop!

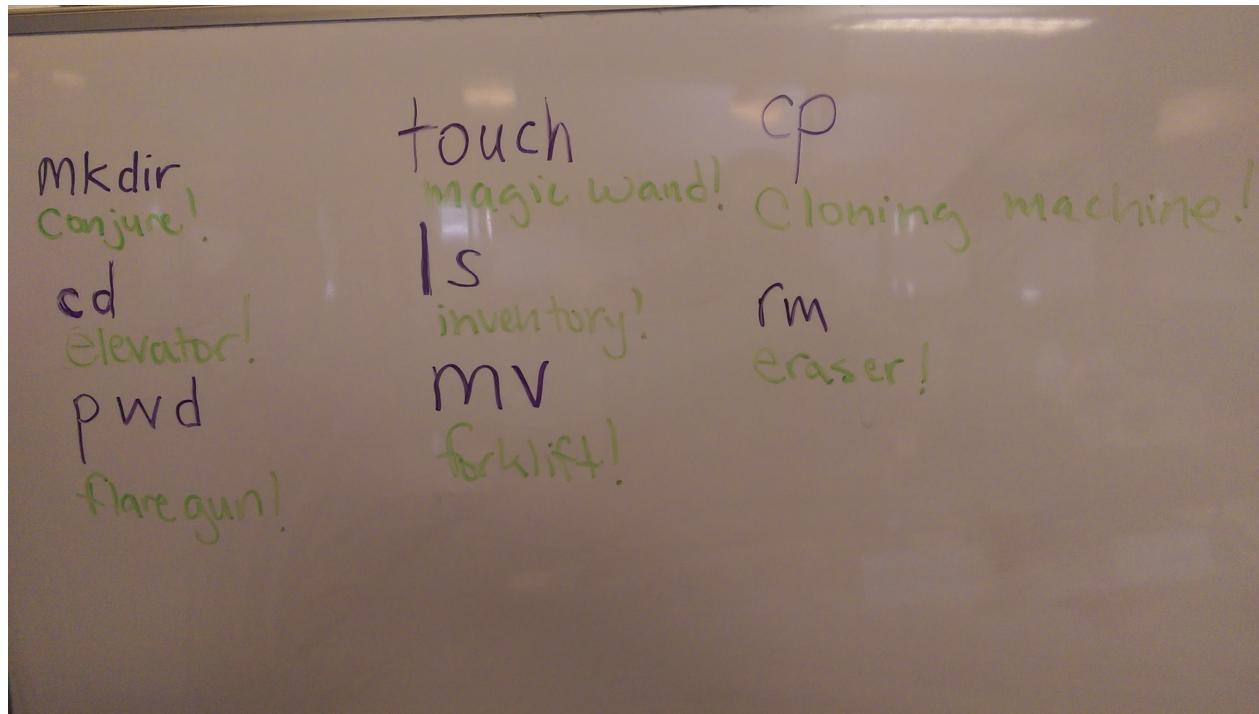
- Create a Github account
- Create a Heroku account
- Turn your laptop into a developer machine!
- Verify that your computer is ready for WDI!

Lesson: Culture of WDI

- Build a relationship and form community together
- Set reasonable expectations and ground rules
- Understand the curriculum arc and how to succeed in an immersive course
- Establish class values and collectively commit to owning the WDI experience

Lesson: Basic Terminal + Navigating the File System

- Summarize a basic file system structure
- Use commands to navigate and modify files and directories
- Differentiate between the GUI and CLI
- Here is a [link to the slide deck from class](#) - use the **left** and **right** arrows to navigate forward and backward.



Workshop Time:

- Star Wars, the Command Line, and The Battle for the Fate of the Universe

Homework

Tonight's homework is to read *chapters 1 & 2* from **Eloquent Javascript** (<http://eloquentjavascript.net/>). These chapters should introduce the concepts of *variables* and *types* as well as *programming structure* in Javascript. This book is available for free entirely online (though you may purchase a physical copy).

OS X Keyboard Shortcuts

Mac OSX Shortcuts

Searching with Spotlight

- In OS X, `cmd-space` allows you to open **Spotlight** and search for files on your computer.

Applications and Windows

- `cmd-space` open up spotlight and type in the name of an app
- `cmd-shift-[` or `cmd-shift-]` cycle through tabs
- `cmd-w` close tab
- `cmd-shift-w` close a window
- `cmd-q` quit a program
- `cmd-o` open a file
- `cmd-n` make a new file
- `cmd-s` save a file
- `cmd-tab` will let you cycle through windows and choose one to bring to the top of your GUI (also known as switching applications)

Text Editing

- `cmd-left_arrow` OR `cmd-right_arrow` move cursor to the beginning or end of a line
- `cmd-up_arrow` OR `cmd-down_arrow` move cursor to the top or bottom of a document
- `option-left_arrow` OR `option-right_arrow` move cursor word by word
- `cmd-backspace` delete entire line before the cursor
- `option-backspace` delete entire word at once

Selecting/Moving Text

- `cmd-a` select all
- `cmd-c` copy selected item to the clipboard
- `cmd-x` remove selected item and copy it to the clipboard
- `cmd-v` paste copied item

Undo/Redo

- `cmd-z` undo
- `cmd-y` redo

Terminal Cheatsheet

Navigation

- `pwd` print working directory (where am I?)
- `ls` list files in current directory (what is here?)
- `cd` go to home directory.
- `cd ..` goes up a level in the directory hierarchy.
- `cd ../../` go up two directories
- `cd folder` go into that folder

Making/Removing Files and Folders

- `mkdir folder` create a folder
- `rmdir folder` deletes an empty directory.
- `rm -rf folder` deletes folder and all files in that folder
- `rm file-name` deletes a file
- `touch file-name` creates a file

Moving and Copying

- `mv file-name new-file-name` rename a file
- `mv file-name folder-name/file-name` put a file into a subfolder
- `cp file-name new-file-name` copy a file
- `cp -R folder new-folder` copy a folder

Traversing the Filesystem using `..`

- You can traverse the filesystem using `..` to move to a parent folder.
- If I'm in `~/Documents`, I can move to `~/Downloads` using the following command: `cd .. ~/Downloads`.
- Use tab-completion to feel your way around the terminal when using that.
- The `/` helps denote that you mean you're moving into a parent folder.

Editing

- `atom .` in the folder that has the files you wish to edit

Get Out of There

- `exit` exit the terminal

Star Wars, the Command Line, and The Battle for the Fate of the Universe

Working in the command line is a key skill to develop as a programmer. It's a big break from what you're used to, and practice makes (eventually) perfect. Let's explore the Star Wars narrative using the command line.

Note: Each file (not directory/folder) you create should be a *text file*. This means the file will end with the extension of `*.txt`. For example: `jimbo_jones.txt`

"A New Hope"

Act I

- In your `dragons` directory, create a directory called `star_wars`
- In your `star_wars` folder, create two new directories: `empire` and `rebellion`.
- Inside the `empire` directory, create a .txt file called `darth_vader`
- Use the force(or your knowledge of the command line) to add the text "...heavy breathing..." to the `darth_vader` file. (Don't remember how to do this? Use the 'other force', known as Google or Stack Overflow)
- Inside the `empire` directory, create a .txt file called `emperor_palpatine`.
- Inside the `empire` directory, create a directory called `death_star`
- Move `darth_vader` into the `death_star`

Act II

- Move back to your `star_wars` directory, and enter the `rebellion` directory
- Create a file called `princess_leia` and add the text 'Help me, Obi-Wan...You're my only hope.'
- Create a file called `obi_wan`
- Create a file called `luke_skywalker`
- Create a directory called `millenium_falcon`
- Inside the `millenium_falcon`, create two files: `han_solo` and `chewy`
- Move `luke_skywalker`, `obi_wan`, and `princess_leia` into the `millenium_falcon`
- Move the `millenium_falcon` into the `death_star`

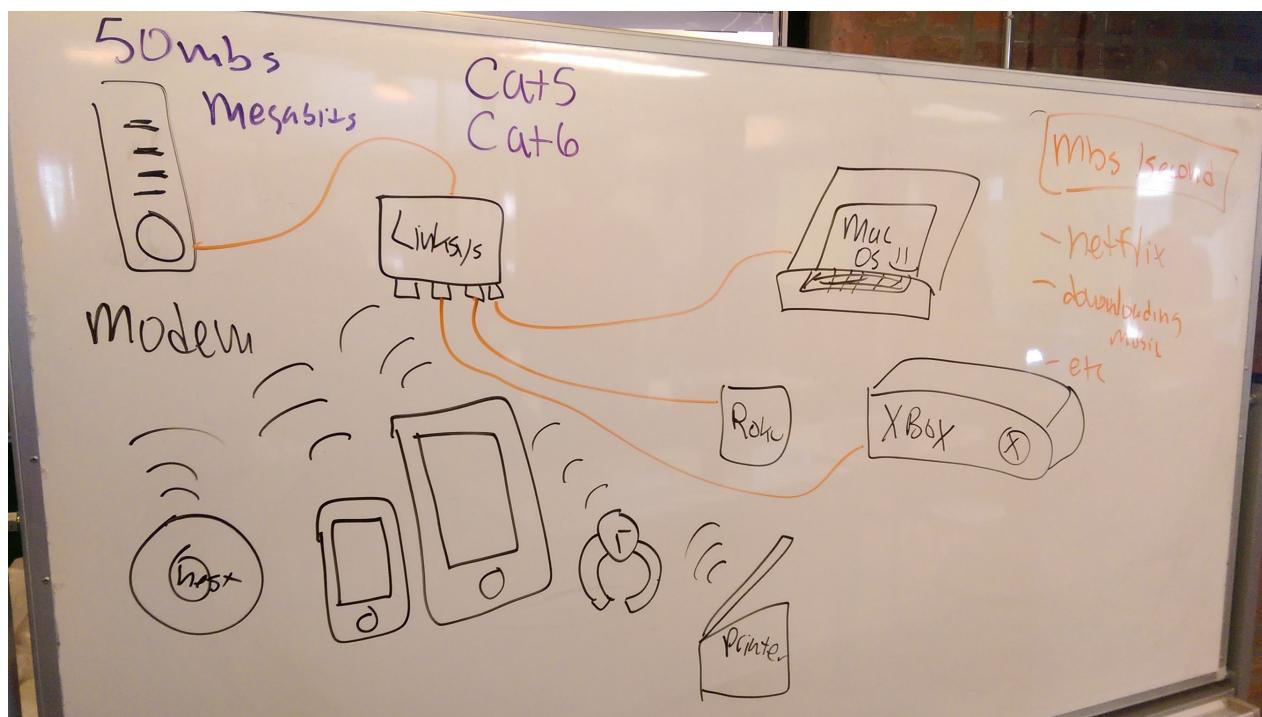
Act III

- Unload the Millenium Falcon! Move the whole crew from the `millenium_falcon` directory into the `death_star` directory
- Darth_vader has defeated obi_wan! Delete poor `obi_wan`
- Our heroes have disabled the tractor beam! Move the whole crew back into the `millenium_falcon`!
- Move the `millenium_falcon` back into the `rebellion` directory
- darth_vader leaves the death_star to pursue Luke! Move him from the `death_star` into the `empire` directory!
- Thanks to his practice back home at Beggar's Canyon, Luke blew up the death star! Remove it from the galaxy!
- You win!

1.2 Hello, World

Morning Exercise

- Understand network topology
- Visualize a network
- Draw a network diagram

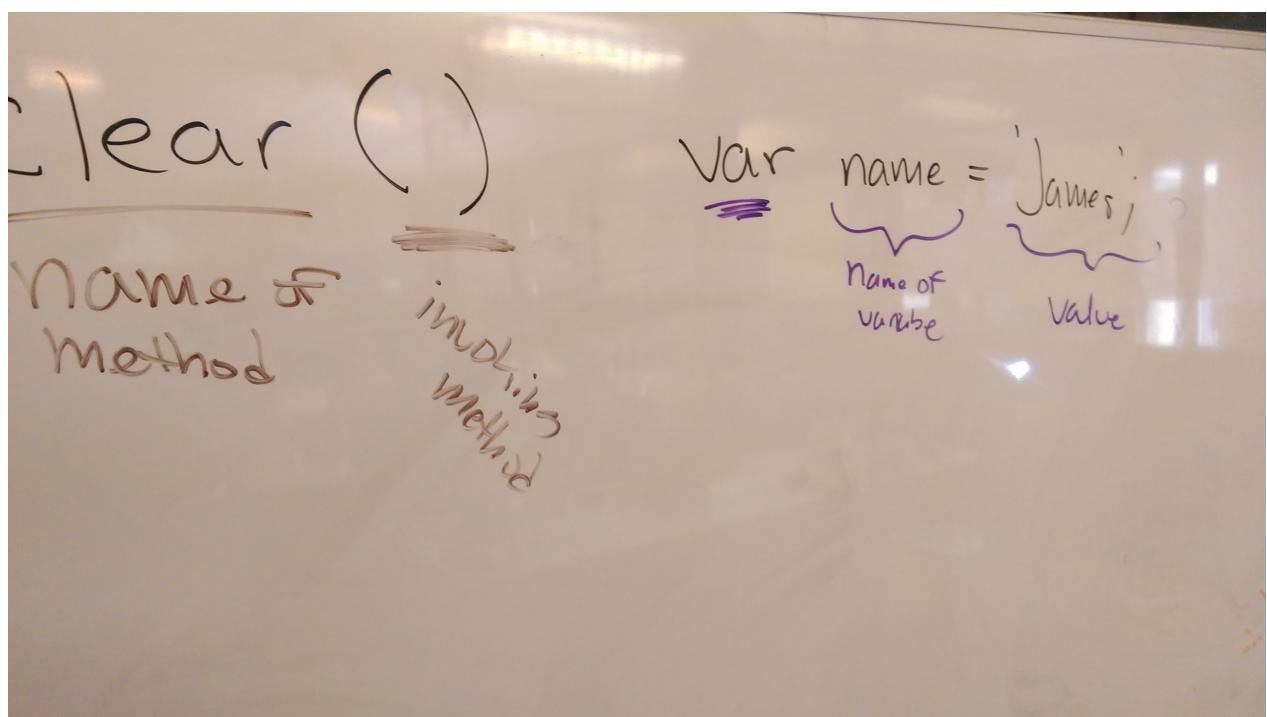
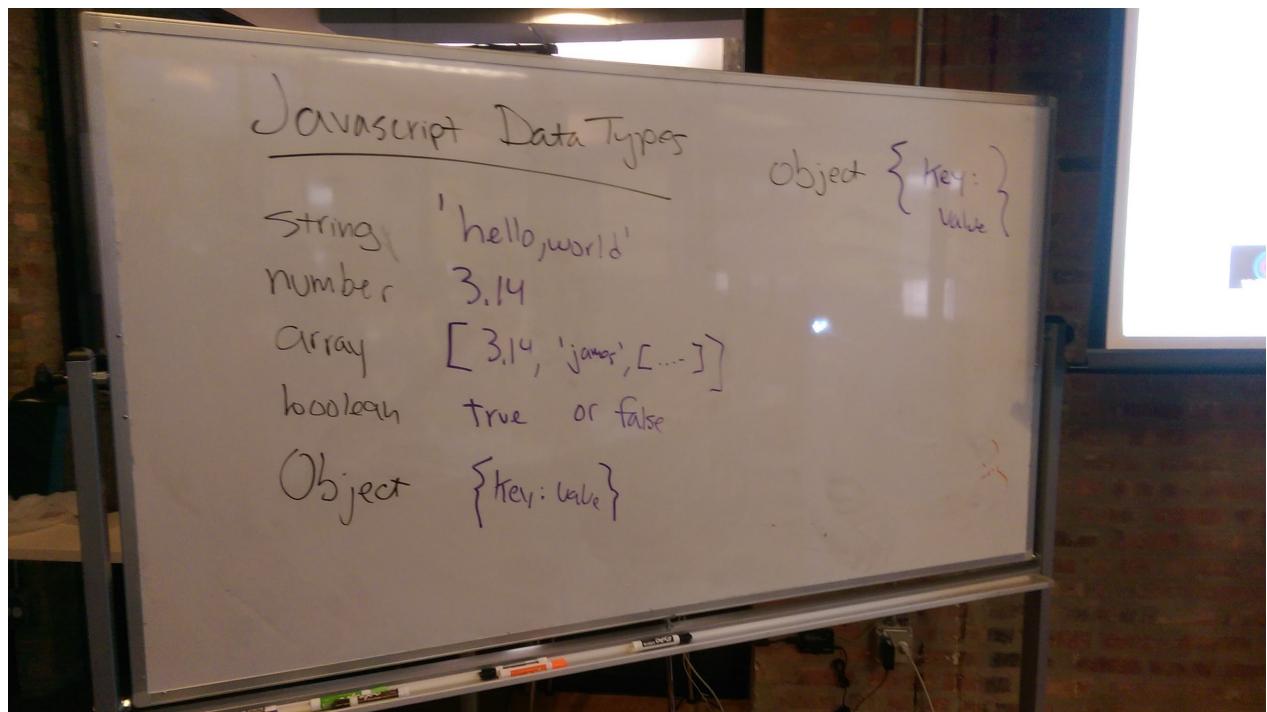


Data Types, Variables, and Arrays

Objectives

- Describe the concept of a "data type" and how it relates to variables
- Describe use cases of different "data types"
- Declare, assign to, and manipulate data stored in a variable
- Explore and use a programming or markup language's standard library and built-in functions (iterators, datatype/array methods)
- Iterate over and and manipulate values in an array
- Describe how arrays are used to store data

Class Examples



<u>Javascript</u>	vs	<u>Java</u>
var place = "SPACE";		String place = "Space";
var x = 42;		Integer x = 42;
var list = [....];		ArrayList list = [....];

Arrays... Arrays, Everywhere ⚡

- Let's build an array
- Then manipulate it using METHODS!

- pop();
- reverse();
- shift();
- push();
- unshift();

```
var cocktails = ['gin & tonic', 'white russian', 'mojito', 'sangria', 'grape ape'];

cocktails.pop(); // pop() removes the last item in an array

// this didn't work...
//var lengthOfArray = cocktails.length;

// this will!
for (var i = 0; i <= cocktails.length; i++) {

    console.log('i = ' + i);
    //console.log('array length = ' + lengthOfArray);
    console.log(cocktails[i]);

    console.log("I could use a... " + cocktails[i]);

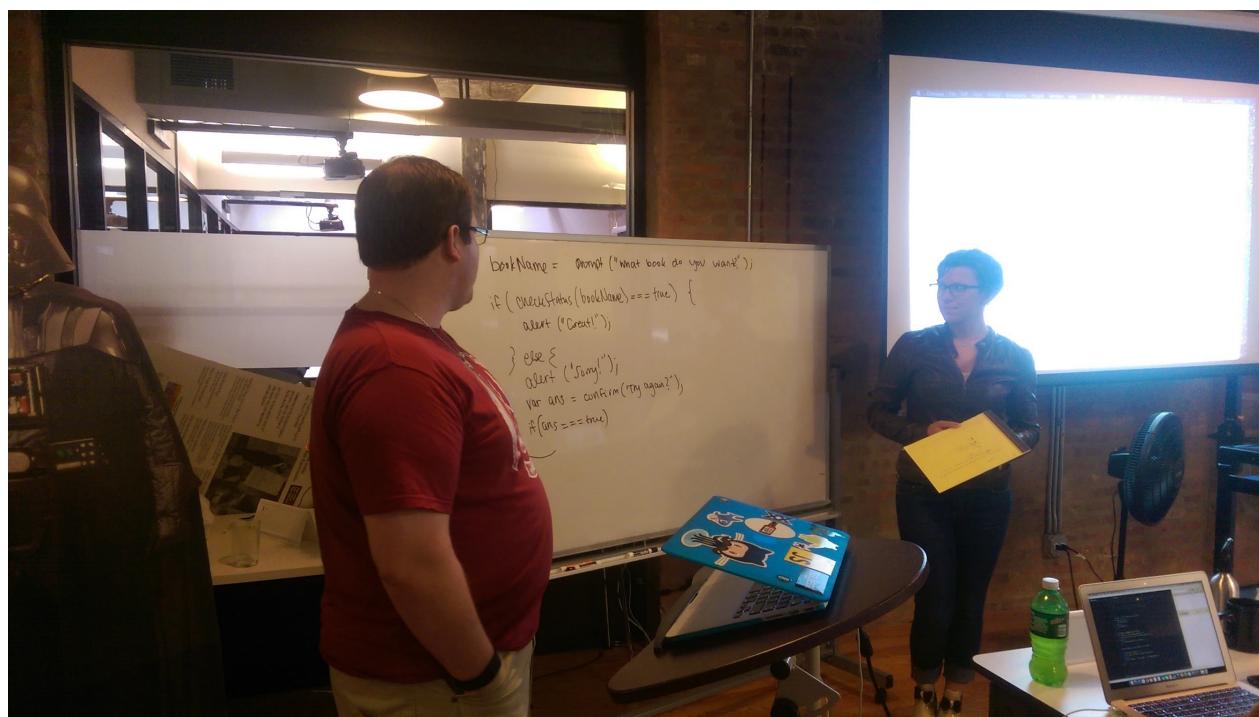
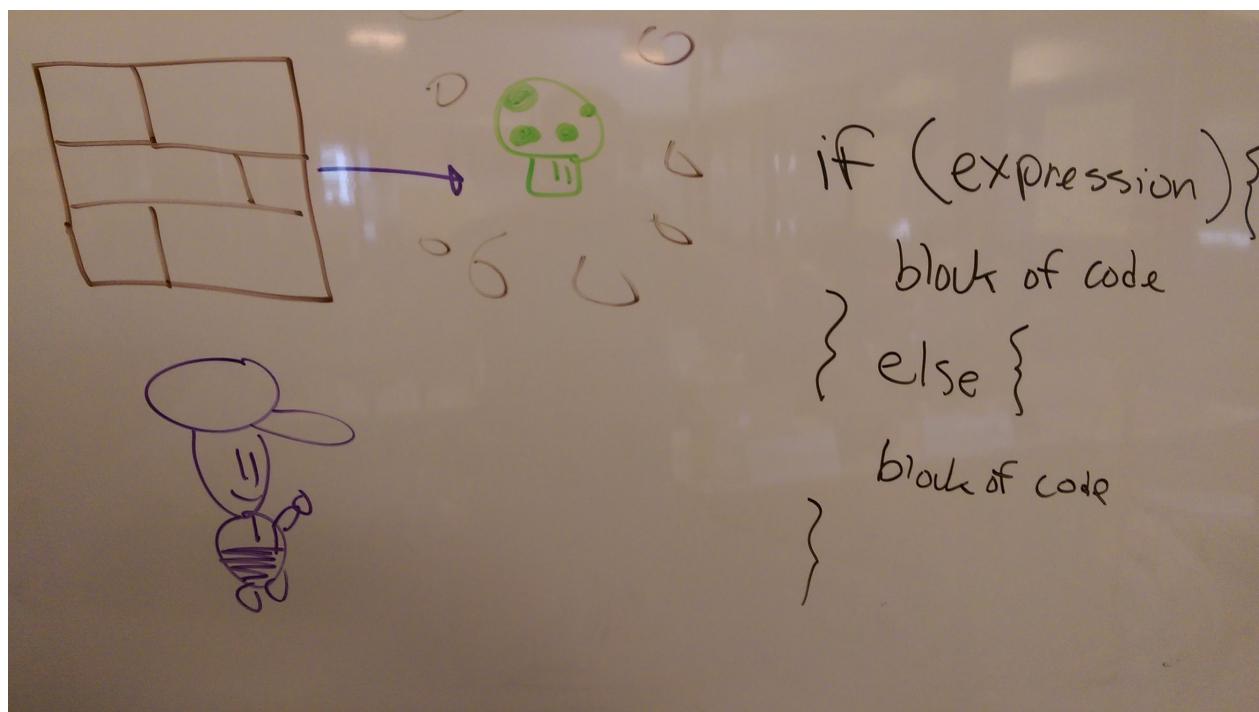
    cocktails.pop();
}
```

Mastering Control Flow

Objectives

- Differentiate between true, false, 'truth-y', and 'false-y'
- Use if/else if/else conditionals to control program flow based on boolean conditions
- Use switch conditionals to control program flow based on explicit conditions
- Use comparison operators to evaluate and compare statements
- Use boolean logic (!, &&, ||) to combine and manipulate conditionals

Class Examples



```

// block of code
{
    console.log('hello, world');
}

var age = confirm('Are you over 21?');
if (age === true) {
    alert('huzzah');
} else {
    alert('soon.jpg');
}

```

```
// 'location' is a reserved word
var somewhere = window.location.host;
if (somewhere != 'ga-chicago.github.io') {
  alert('oh no! this is not the right website.');
} else {
  alert('aww yeah u aint hackd');
}

var userInput = prompt('what is your name?');

if (userInput.length < 1) {
  alert('hey, you didn\'t give us your name');
} else {
  alert('thanks, ' + userInput + '!!!!@121211212');
}

// and operator &&

if (age >= 21 && hasMoney == true) {
  // you can buy booze
}

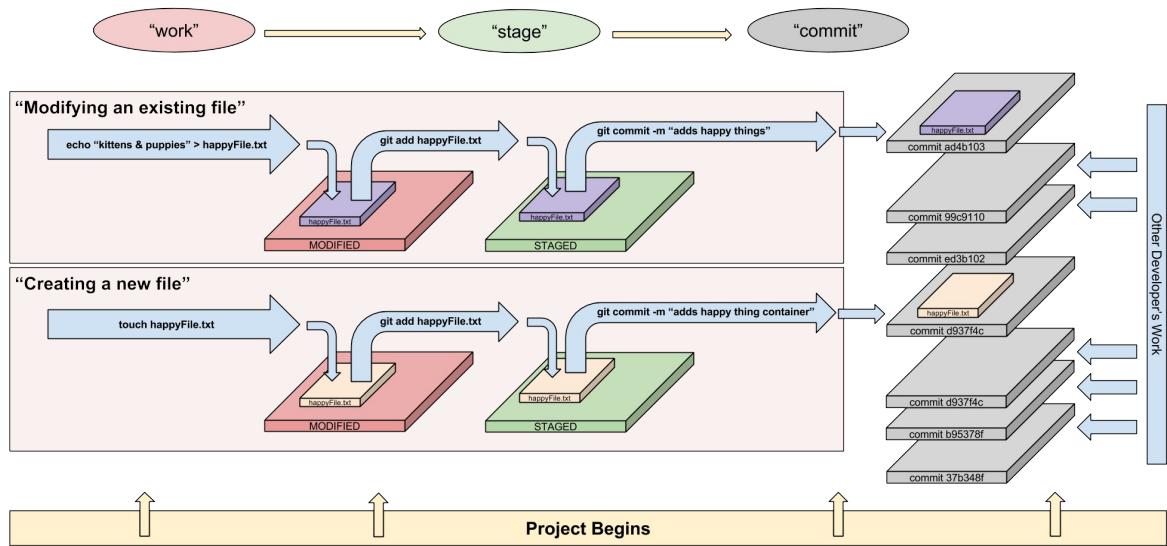
if (21 >= 21 && true == true) {
  console.log('you can haz booze');
}

// OR operator ||
if (true || false) {
  console.log('boolean party!!@!212');
}

var name = 'james';
if (name === 'james' || name === 'jim') {
  console.log('j names rule');
}

var car = 'ford';
switch (car.toLocaleLowerCase()) {
  case "mazda": alert('mazdas zoom zoom');
    break;
  case "nissan": alert('the leaf is green');
    break;
  default:
    alert('your car is not here..?!@');
    break;
}
```

Lab: Introduction to Git & Github



- Slide Deck: <https://presentations.generalassemb.ly/559430358d25e96187c4#/>

Homework

- Read a *Re-introduction to Javascript*: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript
- Complete the **Day 2 Homework: Markdown and READMEs**
-

:)

Javascript

Variables

Summary

This page shows examples of how to declare and use variables in Javascript.

Discussion

- **Declaring variables:** `var myVariable = ["item", 2, 4, "stuff"];`
- **Data types:**
 - string: `"stuff"`
 - integer: `42`
 - float: `3.14`
 - bool: `false`
 - array: `["things", "go", "here"]`
- Access items in an array through **indices**: `myArray[0];`

Examples

- String: `var myString = 'hello';`
- integer: `var myInt = 42;`
- float: `var myFloat = 3.14;`
- bool: `var myBool = false;`
- array: `var myArray = ["things", "go", "here"];`

References

- [MDN: Variables](#)

1.2 Lesson Examples: Data Types

typeof()

To get an idea of the type of data we're working with, we can use `typeof()`. Let's try it out in the console with the following:

```
typeof(37) === 'number';
=> true

typeof({}) === 'object';
=> true

typeof('hi there') === 'string';
=> true
```

`typeof()` returns a string with the type of the operand, or expression of the object you're looking at.

Numbers

In more low-level languages, numbers are divided into two classes or objects:

- Integers

```
..., -1, 0, 2, 3, 4, 5, ...
```

- Floats (or Decimal numbers)

```
2.718, 3.14, .5, .25, etc
```

All numbers in JavaScript are "**double-precision 64-bit format IEEE 754 values**" - read this as "There's really no such thing as an integer in JavaScript." You have to be a little careful with your arithmetic if you're used to math in other programming languages. Let's take a look at what happens when we do this:

```
0.1 + 0.2
=> 0.3000000000000004
```

In JavaScript, these data points are the same **type** of object, which it calls *Numbers*, so if you know floats and integers do not go looking for them.

Arithmetic Operators

Operators are used to work with data in JavaScript. The standard [arithmetic operators](#) - that you've been learning since grade school - are supported, including addition, subtraction, modulus (or remainder) arithmetic and so forth. Check it out:

```
1 + 2  
=> 3
```

```
2 - 5  
=> -3
```

```
5 / 2  
=> 2.5
```

```
6 * 2  
=> 12
```

Special Number Operators

JavaScript can be a little cheap with the number of operations it allows you to do. For example, how is someone supposed to square a number or cube a number easily?

Luckily there is a special `Math` object with some very useful methods.

- Taking a number to some `power`? Then just use `Math.pow`

```
// 3^2 becomes  
Math.pow(3, 2)  
=> 9  
// 2^4 becomes  
Math.pow(2, 4)  
=> 16
```

- Taking a square root

```
// √(4) becomes  
Math.sqrt(4)  
=> 2
```

- Need a `random` number? Then use `Math.random`.

```
// The following only returns a random decimal
Math.random()
=> .229375290430
/**
The following will return a
random number between 0 and 10
*/
Math.random()*10
```

- Since Numbers can be **Floats** or **Integers** we often want to get rid of remaining decimal places, which can be done using `Math.floor`.

```
// Remove the decimal
Math.floor(3.14)
=> 3
Math.floor(3.9999)
=> 3
```

Strings

Strings are collections of letters and symbols known as *characters*, and we use them to deal with words and text in JavaScript. Strings are just another type of **value** in Javascript.

```
"John"
"Jane"
"123"
```

String helper methods

To find the length of a string, access its `length` property:

```
"hello".length;
=> 5
```

There's our first brush with JavaScript objects! Did I mention that you can use strings like objects, too?

Strings have other `methods` as well that allow you to manipulate the string and access information about the string:

```
"hello".charAt(0);
=> "h"

"hello, world".replace("hello", "goodbye");
=> "goodbye, world"

"hello".toUpperCase();
=> "HELLO"
```

Types of values like `Number` or `String` are not very useful without being able to form **Expressions or Combinations**.

Try your favorite number operators as expressions:

```
1 + 1
=> 2
2 - 1
=> 1
```

You can also create expressions with strings using addition:

```
"Hello, " + "world!"
=> "Hello, world!"
```

This is called **String Concatenation**.

Converting Strings to Integers with `parseInt()` and `parseFloat()`

You can convert a string to an integer using the built-in `parseInt()` function. This takes the base for the conversion as an optional second argument, which you should always provide:

```
parseInt("123", 10);
=> 123

parseInt("010", 10);
=> 10
```

This will be important later when we're taking user input from the web and using it on our server or in our browser to do some type of numeric calculation.

Similarly, you can parse floating point numbers using the built-in `parseFloat()` function which uses base 10 always unlike its `parseInt()` cousin.

```
parseFloat("11.2");
=> 11.2
```

You can also use the unary `+` operator to convert values to numbers:

```
+ "42";
=> 42
```

NaN

The `parseInt()` and `parseFloat()` functions parse a string until they reach a character that isn't valid for the specified number format, then return the number parsed up to that point. However the `+` operator simply converts the string to `NaN` if there is any invalid character in it.

A special value called `NaN` (short for "Not a Number") is returned if the string is non-numeric:

```
parseInt("hello", 10);
=> NaN
```

`NaN` is **toxic**: if you provide it as an input to any mathematical operation the result will also be `NaN`:

```
NaN + 5;
=> NaN
```

You can test for `NaN` using the built-in `isNaN()` function:

```
isNaN(NaN);
=> true
```

JavaScript's numeric operators are `+`, `-`, `*`, `/` and `%` and all work as you expect and should have practiced during your prework.

Null and Undefined

JavaScript distinguishes between:

- `null` a value that indicates a deliberate non-value
- `undefined` that indicates an uninitialized value — that is, a value hasn't even been assigned yet

The main note to make here is that these variables should always have the `var` keyword and use `camelCase`

Assignment Operators

Values are assigned using `=`, and there are also compound assignment statements such as `+=` and `-=`:

```
var x = 1;
=> 1

x += 5
=> 6
```

You can use `++` and `--` to increment and decrement, respectively. These can be used as prefix or postfix operators.

In Javascript we just discussed two types of values we can use. We call these values objects, which for now just means that in addition to storing some data you also get to use some helpful methods when you are working with them.

- If you want to turn a number into a string you can use a helpful method called `toString`.

```
(1).toString()
=> "1"
/**
  be careful though,
  since numbers can be floats
  javascript might
  misunderstand you.
*/
1.toString()
=> Float Error
// but the following works
1..toString()
```

Arrays

Unfortunately, strings and numbers are not enough for most programming purposes. What is needed are collections of data that we can use efficiently, Arrays.

Arrays are great for:

- Storing data
- Enumerating data, i.e. using an index to find them
- Quickly reordering data

Arrays, ultimately, are a data structure that is similar in concept to a list. Each item in an array is called an element, and the collection can contain data of the same or different types. In JavaScript, they can dynamically grow and shrink in size.

```
var friends = ['Moe', 'Larry', 'Curly'];
=> ['Moe', 'Larry', 'Curly']
```

Items in an array are stored in sequential order, and indexed starting at `0` and ending at `length - 1`.

```
// First friend
var firstFriend = friends[0];
=> 'Moe'
// Get the last friend
var lastFriend = friends[2]
=> 'Curly'
```

We can even use strings like arrays:

```
var friend = "bobby bottleservice";
// pick out first character
friend[0]
//=> 'b'
friend.length
```

Working with Arrays

Using the JavaScript Keyword `new`, is one way of creating arrays:

```

var a = new Array();
=> undefined

a[0] = "dog";
=> "dog"

a[1] = "cat";
=> "cat"

a[2] = "hen";
=> "hen"

a
=> ["dog", "cat", "hen"]

a.length;
=> 3

```

A more convenient notation is to use an array literal:

```

var a = ["dog", "cat", "hen"];

a.length;
=> 3

```

Length method

The `length` method works in an interesting way in Javascript. It is always one more than the highest index in the array.

So `array.length` isn't necessarily the number of items in the array. Consider the following:

```

var a = ["dog", "cat", "hen"];
a[100] = "fox";
a.length; // 101

```

Remember: the length of the array is one more than the highest index.

Getting data from an array

If you query a non-existent array index, you get `undefined`:

```
var a = ["dog", "cat", "hen"];
=> undefined

typeof a[90];
=> undefined
```

Array helper methods

Arrays come with a number of methods. Here's a list of some popular helpers:

Note: You might want to demonstrate a few of these.

- `a.toString()` - Returns a string with the `toString()` of each element separated by commas.
- `a.pop()` - Removes and returns the last item.
- `a.push(item1, ..., itemN)` - `Push` adds one or more items to the end.
- `a.reverse()` - Reverse the array.
- `a.shift()` - Removes and returns the first item.
- `a.unshift([item])` - Prepends items to the start of the array.

Remember, though, you'll never remember *every* method. Explore the [the full documentation for array methods](#) and other helper methods given to you for particular objects.

Iterating through an array

Iterating through the elements of an array, one at a time, is a very common practice in programming.

We can use a `for` loop to iterate over the elements of an array like this:

```
var teams = ['Bruins', 'Cal Bears', 'Ravens', 'Ducks'];
for (var i = 0; i < teams.length; i++) {
    console.log(teams[i]);
}
```

JavaScript arrays have several advanced *iterator methods*.

Several of these methods require a function be supplied as an argument, and the code you write in the function will be applied to *each* item in the array, individually.

As an example, lets look at the `forEach` method that we can use instead of a `for` loop to iterate the elements:

```
var teams = ['Bruins', 'Cal Bears', 'Ravens', 'Ducks'];
teams.forEach(function(el) {
    console.log(el);
});
```

This function would return:

```
Bruins
Cal Bears
Ravens
Ducks
undefined
```

Notice how much clearer this syntax is than that of the `for` loop?

Here are some other iterator methods for you to research and practice with:

- `Array.every()`
- `Array.some()`
- `Array.filter()`
- `Array.map()`

1.2 Lesson Examples: Logical operators and control flow

JavaScript supports a compact set of statements, specifically control flow statements, that you can use to incorporate a great deal of interactivity in your application.

Block Statements

Statements meant to be executed after a control flow operation will be grouped into what is called a **block statement**. These statements are wrapped into a pair of curly braces:

```
{  
  console.log("hello");  
  console.log("roar");  
}
```

Block scope

We've seen that the scope in JavaScript changes often. In the case of **block statements**, there is no scope created.

```
var name = "james";  
{  
  var name = "adriana";  
}  
console.log(x); // outputs gerry
```

Only functions introduce scope in Javascript.

Conditional statements

Conditional statements are a way of essentially skipping over a block of code if it does not pass a boolean expression. JavaScript supports two conditional statements:

```
if ... else and switch .
```

if...else statement

```
if(expr) { code }
```

... means run the `code` block if `expr` is `true`

```
if (1 > 0) { console.log("hi") }
//=> hi
```

When you need to test more than one case, you may use `else if`:

```
var name = "kittens";
if (name == "puppies") {
  name += "!";
} else if (name == "kittens") {
  name += "!!";
} else {
  name = "!" + name;
}
name == "kittens!!"
//=> true
```

Note: It is recommended **to not** assign variables in the conditional expression, because the assignment of a value to a variable, like this:

```
student = "Jason";
//=> "Jason"
```

The expression above will return the value (as shown on the second line), so if you assign a truthy value inside a conditional statement, then this condition will always be true, or if you assign something undefined, it will make the conditional statement false (because undefined is falsy). Another potential issue with this is that it can be confused with equality(`==`). The example below is the illustration of **WHAT NOT TO DO**, in general:

```
if (x = 3) {
  console.log("boo");
}
```

Truthy & Falsy

All of the following become false when converted to a Boolean

- `false`
- `0`
- `""` (empty string)
- `NaN`
- `null`
- `undefined`

All other values become true when converted to a Boolean

Do not confuse the primitive boolean values `true` and `false` with the true and false values of the Boolean object. For example:

```
var b = new Boolean(false);
if (b) { console.log("true") }
//=> true
```

There is a simple way of verifying the truthyness or falsiness of a value. When you add `!` in front of a value, the returned value will be the inverse of the value in a boolean. So if you add two `!` then you'll get the boolean value of the original one:

```
!!1
//=> true

!!0
//=> false

!!-1
//=> true

!![]
//=> true

!!{}
//=> true

!!null
//=> false

!!"
//=> false
```

Boolean/Logical Operators

Logical operators

Logical operators will always return a boolean value `true` or `false`.

There are two "binary" operators that require two values:

- **AND**, denoted `&&`
- **OR**, denoted `||`

A third "unary" operator requires only one value:

- **NOT**, denoted `!`

&& (AND)

The `&&` operator requires both left and right values to be `true` in order to return `true`:

```
true && true
//=> true
```

Any other combination is false.

```
true && false
//=> false

false && false
//=> false
```

|| (OR)

The `||` operator requires just one of the left or right values to be `true` in order to return true.

```
true || false
//=> true

false || true
//=> true

false || false
//=> false
```

Only `false || false` will return `false`

The `!` takes a value and returns the opposite boolean value, i.e.

```
!(true)
//=> false
```

The `&&` and `||` operators use short-circuit logic, which means whether they will execute their second operand is dependent on the first. This is useful for checking for null objects before accessing their attributes:

```
var name = o && o.getName();
```

In this case, if the first operand `o` is false, then the second operand `o.getName()` will not be evaluated. The expression is basically saying "we already know the whole `&&` expression is false, because `o` is falsey. Why bother dealing with the second operand?"

Or for setting default values:

```
var name = otherName || o.getName();
```

In this case, if the first operand `otherName` is `false`, then we'll see that `"my name"` will be returned. If `othername` is truthy (e.g. it contains a value), it will get returned, and the second expression won't even be evaluated. The expression is basically saying "we already know the whole `||` expression is true, because `o` is truthy. Why bother dealing with the second operand?"

Comparison Operators

[Comparisons](#) in JavaScript can be made using `<`, `>`, `<=` and `>=`. These work for both strings and numbers. This is both useful, and can be the source of frustration for some developers, since most languages do not implicitly convert strings to numbers the way that JavaScript does.

```
"A" > "a"  
//=> false  
  
"b" > "a"  
//=> true  
  
12 > "12"  
//=> false  
  
12 >= "12"  
//=> true
```

Equality Operator `==`

Equality is a bit more complex. There are 2 ways in JavaScript to verify equality.

When verifying equality using double equal `==`, JavaScript does a lot of the "type coercion" in the background. Like we mentioned above, if the operands have a different type (ie: the number `1` and the string `"1"`), JavaScript will try to change the type of both operands to check whether they are equal. This means that a lot of times, expressions will return equal more easily than if we were stricter about what things were equivalent. Some examples:

```
"dog" == "dog";  
//=> true  
  
1 == true;  
//=> true
```

Equality Operator `==`

To avoid type coercion and measure equality more strictly, **use the triple-equals operator**. Because `==` more truly measures actual equality, we'll use this far more often when checking whether two things are, in fact, the same thing.

Note: "Sameness" and "equality" have various definitions and can be somewhat "fuzzy". They can also differ by programming language. Because you'll often be measuring whether two things are equal, you should investigate the way this works carefully.

Some examples:

```
1 === true;  
//=> false  
  
true === true;  
//=> true  
  
"hello" === "hello"  
//=> true
```

However, there are some incidents when it does not do what we expect, for example when working with empty objects or arrays:

```
{ } === {}  
//=> Uncaught SyntaxError: Unexpected token ===  
  
[ ] === []  
//=> false  
  
[1, 7] === [1, 7]  
//=> false
```

Switch Statement

The switch statement can be used for multiple branches based on a number or string:

```

var food = "apple";

switch(food) {
  case 'pear':
    console.log("I like pears");
    break;
  case 'apple':
    console.log("I like apples");
    break;
  default:
    console.log("No favourite");
}
//=> I like apples

```

In this case the `switch` statement compares `food` to each of the cases (`pear` and `apple`), and evaluates the expressions beneath them if there is a match. It uses `==` to evaluate equality.

The default clause is optional.

Iteration

Iterating is a way of incrementally repeating a task.

for

You can iterate over an array with:

```

var a = [1, 2, 3, 4, 5];
for (var i = 0; i < a.length; i++) {
  console.log(i);
}

```

This is slightly inefficient as you are looking up the `length` property once every loop. An improvement is to chain the `var` assignment:

```

var a = [1, 2, 3, 4, 5];
for (var i = 0, len = a.length; i < len; i++) {
  console.log(i);
}

```

Notice the placement of the comma and semi-colons.

Further Reading

- [Control Flow](#)
- [While](#)

Javascript

String Concatenation

Example

```
// if you need to create a long string, you can:  
var myStory = "hey guys, how is it going? I hope " +  
    "you are enjoying today's WDI!" +  
    " Cheers, all.";  
  
console.log(myStory);
```

Javascript

Arrays

Summary

We can access an array's items using **indices**. Arrays have a variety of methods to utilize.

Discussion

```
var myArray = ["Alex", "Andrew", "James"];
myArray[0]; // this will return "Alex"
// let's assign a new value to "James"
myArray[2] = "Dragons";
```

We also covered a lot of **methods** that each array has. Here are the methods we used.

- **pop()** - removes and returns the the *last* item in the array.
- **push()** - adds an item to the end of the array.
- **shift()** - removes the *first* item in an array.
- **unshift()** - adds an item to the *start* of the array.
- **reverse()** - reverses the order of an array.
- **sort()** - sorts an array by alpha-numerics, based on the first character read.

References

Your references may be placed here. Please place them in an unordered list and add a quick summary of each.

- [MDN: Arrays](#)

Javascript

Loops

Summary

We covered three types of loops: `for`, `while`, and `for-in`. The following example loops through a **kitties** array and then logs each item.

```
var kitties = [
  "magda",
  "grumpy cat",
  "pi",
  "roscoe",
  "adventure kitty"
];
```

Each loop below does nearly the exact same thing.

- A `for` loop should be used when you **know how you want to control** your loop. You have complete control over the condition.

```
for (var i = 0; i < kitties.length; i++) {
  console.log(kitties[i]);
}
```

- A `for-in` loop will allow you to iterate and access every item in your array but may or may not return everything in the appropriate order. If you need to access all items without concern for order, you should use a `for-in` loop.

```
for (var cat in kitties) {
  console.log(kitties[cat]);
}
```

- A `while` loop is designed to allow you to **do** something without knowing how long it could occur. Remember the *b* button for Mario running example?

```

var i = 0;
while (i < kitties.length) {
  console.log(kitties[i]);
  i++;
}

```

Examples

Javascript

```

// FIRST, let us create our object
var myObj = {
  name: "James",
  students: [
    'Jason', 'Paul', 'Ruth', 'Katie'],
  food: 'soylent',
  favouriteMusic: 'indie pop',
  favouriteAnimal: 'cat',
  colour: 'purple',
  phone: 'HTC',
  money: 'none, I am broke',
  isTired: false,
  isAwake: false
};

// for-in loop on an object
// the goal is to log out all values in keys

console.log("Logging keys then values using for-in loop:");
console.log("-----");

// var key = index
// myObj = object
for (var key in myObj) {

  // log the key only
  // ie: myObj.name
  // "name" is the key

  console.log(key);

  // only logging my values per each
  // myObj.key
  // or myObj[key]

  console.log(myObj[key]);
}

```

```
}

console.log("Only logging keys using for loop:");
console.log("-----");
// create an array of my keys
// Object.keys() returns an array of all
// keys in an Object
var myKeys = Object.keys(myObj);

// now, we will just do a for loop
// for each key to be printed out
for (var counter = 0; counter < myKeys.length; counter++) {

    console.log(myKeys[counter]);

}

// for-in loop on an object
// log the key : value on same line

console.log("Logging keys : values using for-in loop:");
console.log("-----");

// var key = index
// myObj = object
for (var key in myObj) {

    console.log("Key: " + key + ", Value: " + myObj[key]);

}
```

References

- [MDN: Loops and Iteration](#)

Javascript

Syntax

Syntax Overview

```
// This quick Javascript file is designed to walk you through
// the basic syntax of Javascript!
// Let us get started!

// we use 'var' to declare a variable.
// always, always, always use 'var'.
var myVariable;

// we've just created a brand new variable with an 'undefined' value.
// we want to 'assign' a value to a variable. You use '=' to assign values.
var myVariable = "I have a value!";

// let's break that down:
// var declares our variable called 'myVariable'. we then use '=' to *assign* it
// a *string* value of "I have a value!". Finally, we 'end' that statement with a ';'
// always, always, always end your statements with a ';'.
// the ';' tells Javascript "HEY! WE'RE DONE, MOVE ON TO THE NEXT STATEMENT!"
// let's do this again! We'll make a number:
// I am *declaring a variable* called 'myNumber' and *assigning* it a value of 42. en
var myNumber = 42;

// let's make a few more!
var myArray = [myVariable, myNumber, "some text"];
var myFloat = 3.14;

// you should practice creating variables. Remember how to "build" your statements.
// So what about 'conditionals' and 'loops'? These braces ('{} * '}') are confusing,
// The braces are designed to let you *organize* your code into 'blocks'.
// They are basically the walls of your code. They keep things contained - neat and t
// Let us inspect a basic 'if' statement:
if (myNumber == 42) { // <-- this brace starts a new block of code!
    // you could have some code in here
} // this brace ends this block of code!

// Notice how we encased that 'condition' inside of ()?
// That allows us to define a condition.
```

```
// We're going to now build a new condition.  
// How would we write out 'if myAge > 65 then I am retired'?  
if (myAge > 65) {  
    alert("retired!");  
}  
  
// notice how we put our 'condition' inside of the ()?  
// They should always be wrapped inside of () when using if/else/loops/etc.  
// And we can run code inside of that block.  
// But what if we aren't retired? How do I use the braces appropriately?  
if (myAge > 65) {  
    alert("retired");  
} else { // notice how 'else' is between different braces? this allows us to divide the code  
    alert("not retired");  
} // end second block  
  
// if you want to add another condition (if else), it would be built the same way:  
if (myAge > 65) {  
    alert("retired!");  
} else if (myAge < 18) { // see how we () to "wrap" another condition? and add a new brace  
    alert("not an adult yet");  
} else { // new 'block' of code  
    alert("not retired");  
}  
  
// So braces allow us to organize our code.  
// Keep in mind - 'conditions' must be inside of (). If they're not, your code will error.  
// This has been a quick and dirty intro to JS Syntax. If you have any questions, please ask!
```

Markdown and READMEs

Popular repositories on Github often have a jazzy introduction to their project that shows right on their Github project's main page.

They accomplish this by including a `Readme.md` file in their repo written in a language called Markdown.

Check out some examples:

- [browserify](#)
- [phaser](#)
- [ponysay](#)

Adding a Readme to a repo

1. On your terminal, navigate to your `yourusername.github.io` folder that you created earlier
2. Create a new file `README.md`
3. Edit `README.md` in Atom and add a short description of the repo (e.g. "Here are some facts about me...")

Previewing Markdown in Atom

1. In the Atom sidebar ctrl-click (or right-click) on `README.md` and select `Markdown Preview` 
2. You should see on the right hand side of Atom the rendered markdown 

Pushing/Viewing on Github

1. Save all your changes to the git repo and push it up to Github
2. Now if you go to your project's page on Github, you should see the text that you put into the README showing below the list of files in the folder 

Jazzing it up

You probably don't yet know Markdown, so this will be one of your first assignments in digging into documentation.

Use these resources as your guides to Markdown:

- [Markdown Basics](#)
- [Mastering Markdown](#)
- [Markdown Cheatsheet](#)

1.3 Functions, Loops, and Objects

Morning Recap

- This morning we will revisit Markdown.

Functions

Learning Objectives

- Describe a functions
- Define using a function
- Create and call a function with and without parameters
- Differentiate an anonymous and a named function
- Pass a function... as an argument

Loops (again)

Learning Objectives

- Describe how a loop works
- Use a standard **for** loop with an **iterator**
- Use a **for-in** loop with collections
- Describe the difference between these types of loops

Objects

Learning Objectives

- Describe **key-value** pair storage
- Utilize **key-value** storage
- Access and assign values in an object
- Iterate through an object

Outcomes

- Today you will work with Amy Hayes
- She will introduce the Outcomes program

- She's here to help you get a job! <3

Homework: Mid-Week Recap

Let's take a few moments this evening to reflect upon what we have studied. Below are a few problems to solve this evening that will work in what you've learned. Solve these problems and then test them in Google Chrome. Place all of these in `01_front_end_fundamentals/your_name/day3_homework.js`. We will create pull requests tomorrow in class to handle the submission.

1. Detecting Types

- Create a re-usable construct in Javascript (..you know, keeping things *DRY*..).
- It should accept **one** argument of any variable.
- This construct should **return** the *type of* the variable that is the argument.

2. Carousel

- Create an array that represents people hopping off a carousel.
- Loop through the carousel using a **for** loop. Every other cycle through the loop (odd), someone will hop off.
- Repeat until the carousel is empty.

3. You are an object...

- Create an object that represents yourself.
- Assign ten attributes about yourself to this object.
- Use a **foreach (for-in)** loop to list all of these *attributes* (and the *keys* that they are associated with).

4. Elementary School

- In elementary school, when you divide you typically use division and have to report the remainder.
- Create a function that returns a string.
- It will accept two arguments: a number, and a number to divide that number against.
- The returned string should state what the result is **and** the remainder.
- You will need to use the `/` and the `%` operators.

1.4 Drawing on the Web

Morning Exercise

1. Define a function max() that takes two numbers as arguments and returns the largest of them. Use the if-then-else construct available in Javascript.
2. Write a function that takes a character (i.e. a string of length 1) and returns true if it is a vowel, false otherwise.
3. Define a function sum() and a function multiply() that sums and multiplies (respectively) all the numbers in an array of numbers. For example, sum([1,2,3,4]) should return 10, and multiply([1,2,3,4]) should return 24. NEW MESSAGES

Re-Introduction to HTML & CSS

Objectives

- Re-introduce HTML and CSS
- Understand what the DOM is
- Define the role of CSS and how selectors work
- Create a web page and style it

A link to the slide deck is available here:

<https://presentations.generalassemb.ly/68229ab5327ec68f5a2d#/>

Selectors and Canvas

Objectives

- Understand how to create a Canvas element
- Create and use Javascript selectors
- Use the coordinate system (and Chrome DevTools)
- Draw on the web using Canvas
- Create a random number generator

Javascript Scope

Objectives

By the end of this, students should be able to:

- Draw a diagram representing variable scope.
- Create a list of operations that explain runtime behavior.
- Create a program that hoists variables.

Lab: Creating a Boilerplate

- Soon

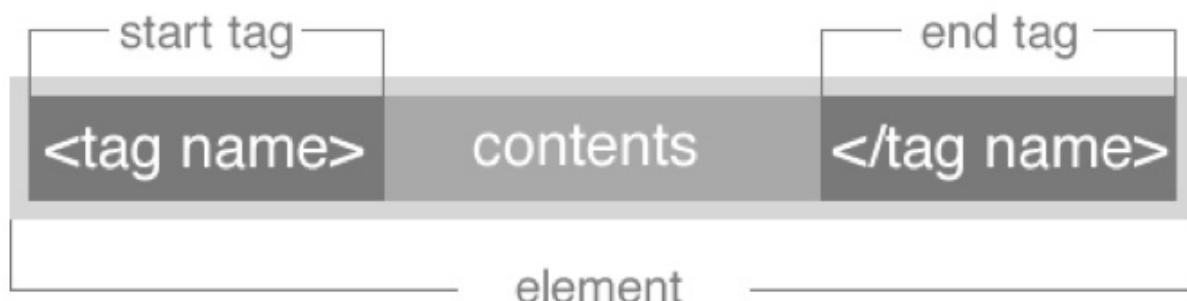
Homework

- Work on and complete the WDI Fundamentals Pre-work
- Review the content that we have covered this week
- Create an issue on our Github repository for any question(s) you may have regarding the content we covered this week

HTML & CSS

Basic HTML

Tag Structure



Base HTML Page:

```
<html>
  <head>
    <title>My webpage</title>
    <link rel="stylesheet" href="css/style.css">
  </head>
  <body>
    <header>
      Insert some sort of logo.
    </header>
    <section>
      <div id="column1">
      </div>
      <div id="column2">
      </div>
    </section>
    <footer>
      &copy; 2015 you!
    </footer>
  </body>
</html>
```

Base CSS Stylesheet:

```
header {
}
section {
}
div {
  position: relative;
}
#column1 {
  position: relative;
  float: left;
}
#column2 {
  position: relative;
  float: left;
}
footer {
}
```

HTML5 Tags

- <header></header>

- Contains header information, logo, other important top of page details.
- <footer></footer>
 - Opposite of header; copyright information, contact, legal, etc.
- <section></section>
 - Specialized div tag
 - Container and is a part of a page
 - contains a single or multiple <article></article> or <div></div> tags.
- <nav></nav>
 - Contains navigation links, menus, and elements

HTML & CSS

DOM

Introduction

The Document Object Model, or DOM, is the interface that allows you to programmatically access and manipulate the contents of a web page (or document). It provides a structured, object-oriented representation of the individual elements and content in a page with methods for retrieving and setting the properties of those objects. It also provides methods for adding and removing such objects, allowing you to create dynamic content.

The DOM also provides an interface for dealing with events, allowing you to capture and respond to user or browser actions. This feature is briefly covered here but the details are saved for another article. For this one, the discussion will be on the DOM representation of a document and the methods it provides to access those objects.

Example

You may access the DOM in Javascript via:

```
// vanilla Javascript  
document  
  
// jQuery  
$( 'document' )
```

Equation for Converting Pixels to EMs

- $1\text{em} = 10\text{px}$
- OR
- $\text{target} / \text{context} = \text{result}$ (Target = target font size, context = font size of containing element)

Example

Here is an `h1` using pixels:

```
h1 {  
  font-size: 30px;  
  font-weight: bold;  
}
```

We can convert that `px` value to `em`s:

$30\text{px} / 10\text{px}$ (standard font size) = 3em

Result:

```
h1 {  
  font-size: 3em; /*  $30\text{px}/10\text{px}$  */  
  font-weight: bold;  
}
```

Javascript

Canvas

Summary

Below is a sample HTML page with a canvas element that will draw a rectangle.

```
<body>
  <canvas id='myCanvas' width='400' height='400'></canvas>

  <script>
    var canvas = document.getElementById('myCanvas');
    var ctx = canvas.getContext('2d');

    ctx.fillRect(0, 200, 20, 10);

  </script>
</body>
</html>
```

Discussion

When you begin drawing an item, remember that you need to `beginPath(x, y);` and then `ctx.fill();` finish filling in the drawing. Any code you want to use to draw should go inbetween these functions.

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');
ctx.beginPath(0, 0);
// draw some other paths and such
// then end it
ctx.fill();
```

Examples

Here is how you draw a circle

```
//ctx.arc(xposition, yposition, radius, 0, Math.PI*2, true);
//if you don't know what radius means, just know that a bigger
//radius will make a bigger circle, and 50 is a good starting point
ctx.arc(75,75,50,0,Math.PI*2,true);
```

References

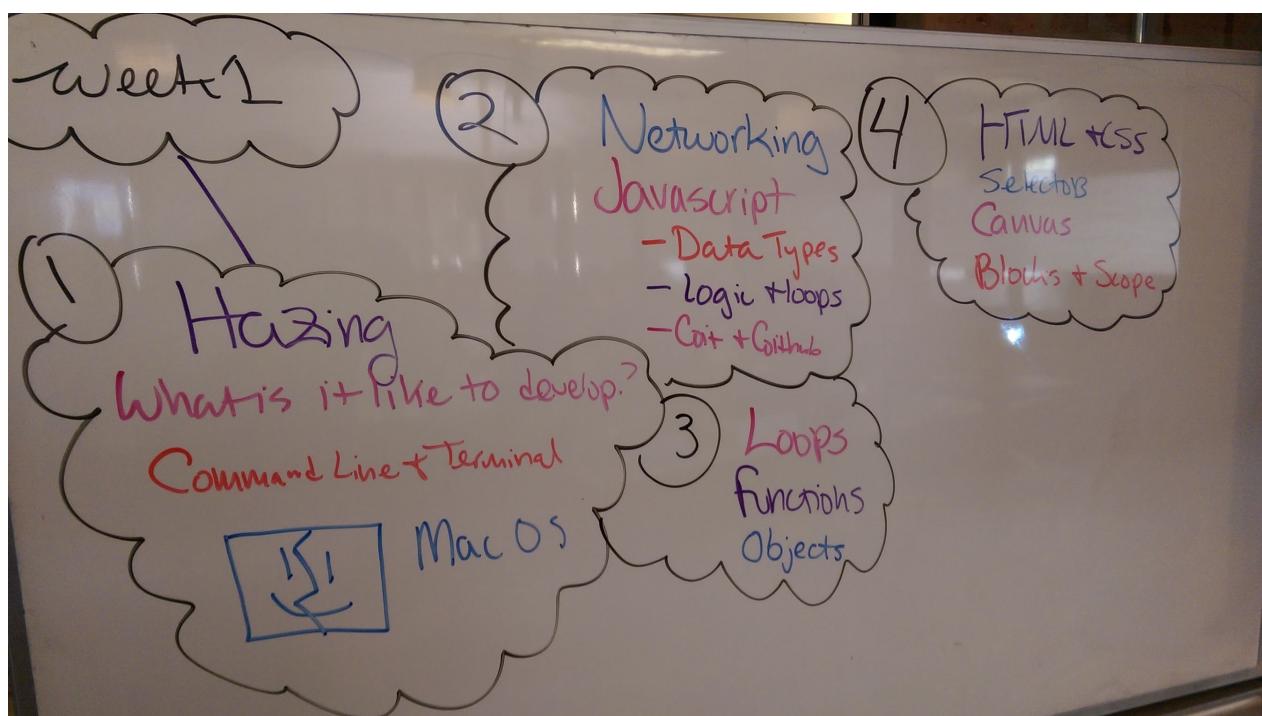
Your references may be placed here. Please place them in an unordered list and add a quick summary of each.

- [MDN Canvas Tutorial](#)

Further Resources

- https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Drawing_shapes
- <http://www.effectgames.com/demos/canvascycle/>

1.5 Second Pass Friday



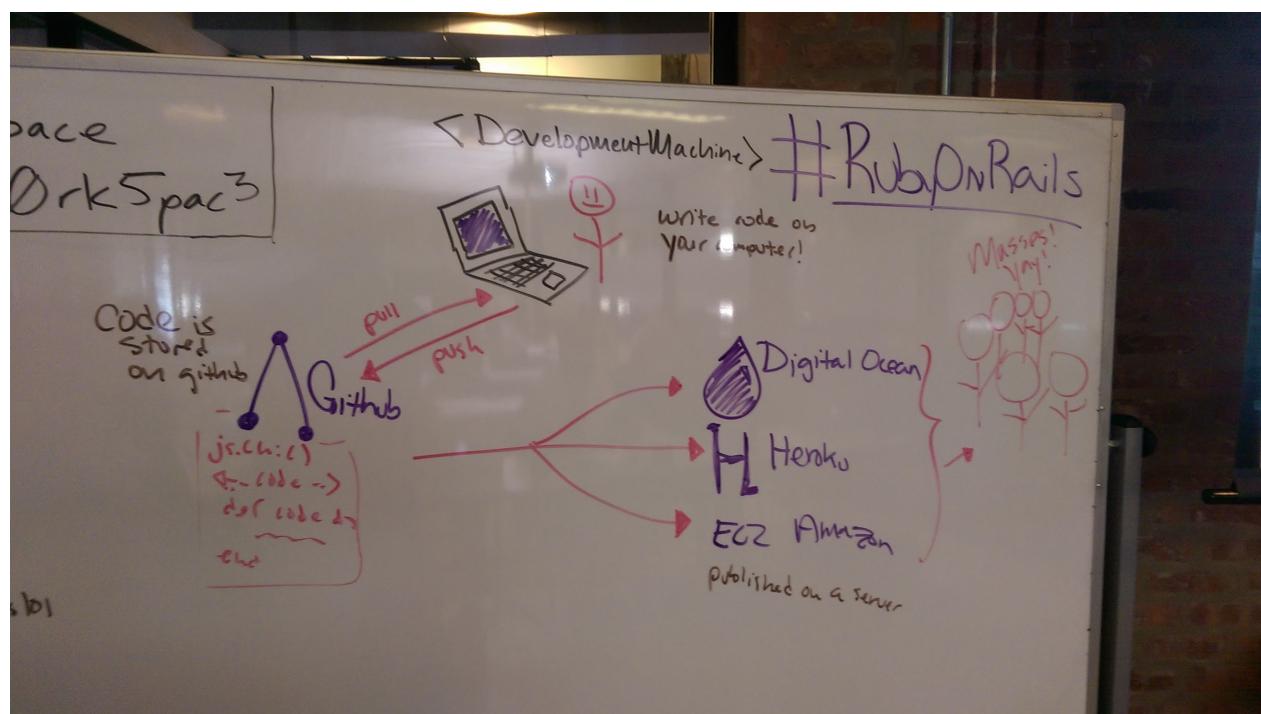
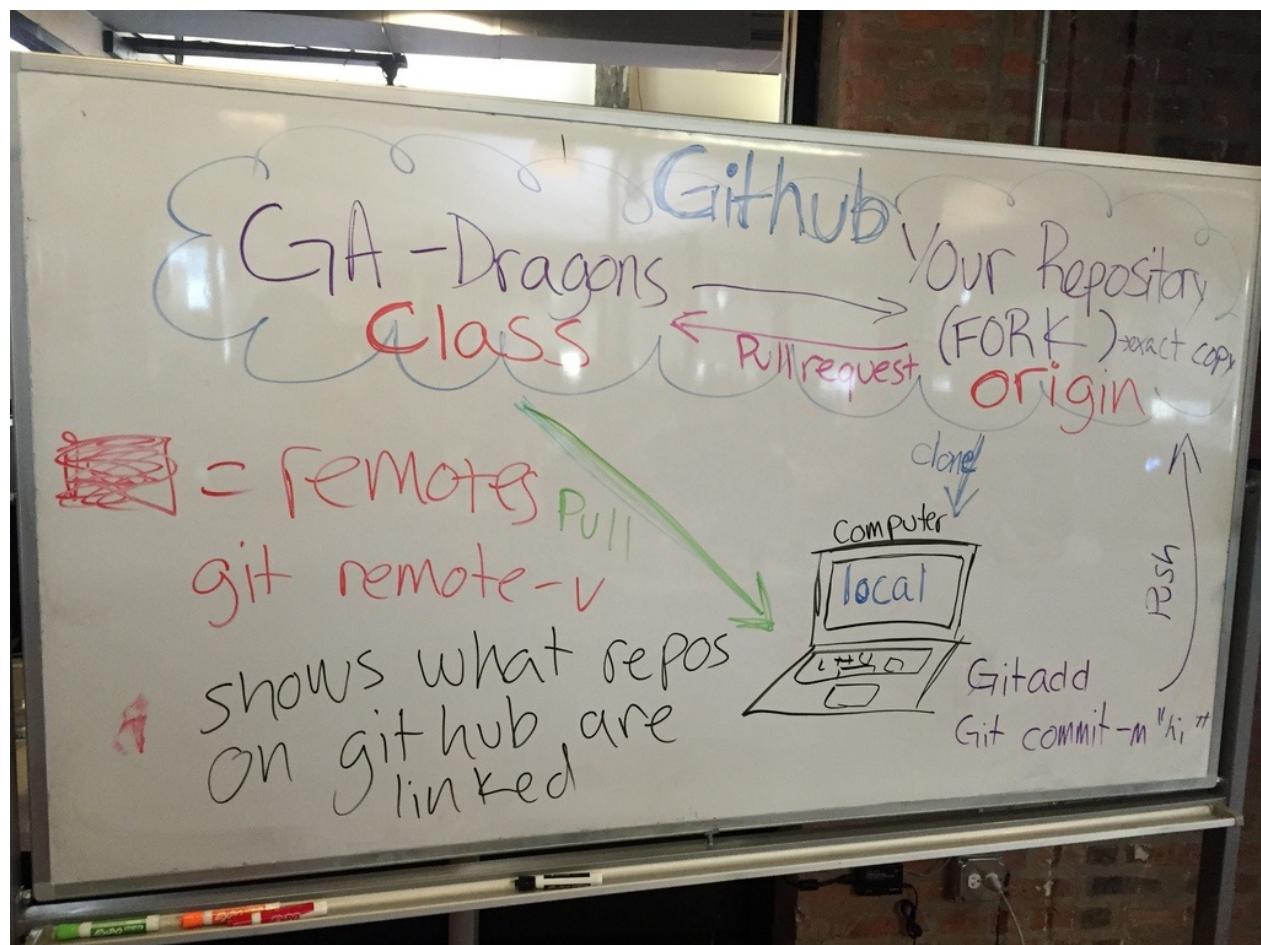
WAT

- <https://www.youtube.com/watch?v=FqhZZNUyVFM>

Homework Submission Recap

1. Navigate to `/your_name` in our class repository. `git add`, `git commit`, `git push origin master` to your fork.
2. Browse to https://github.com/your-username/wdi_chi_dragons
3. Select the **green** pull request icon to create a new pull request.
4. Fill out the comments with your level of comfort, completeness, and any comments.
5. Submit!
6. We'll provide feedback to you via Github so check your email!

Git & Github



Javascript 101 Recap

- Video: <https://www.youtube.com/watch?v=24dV8xpEljU>

Carousel Example

```

var circus = ['johny', 'jenny', 'james', 'jennifer', 'jack', 'jim'];

console.log(circus);

for (var cycles = 0; circus.length > 0; cycles++) {

    console.log(cycles);
    if (cycles % 2 == 1) {
        console.log('odd');
        var popped = circus.pop();
        console.log(popped + ' is off the carousel');
    } else {
        console.log('even');
    }
}

console.log(circus);

```

Canvas

```

<html>
<body>

<canvas id="myCanvas" width="400" height="400"></canvas>

<script type="text/javascript">
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.fillStyle = "rgb(100, 200, 160)";
ctx.fillRect (0, 350, 100, 50);
</script>
</body>
</html>

```

Draw a square in each corner

Change the javascript in the boiler plate so there is a 50x50 square in each corner of the canvas.

```
ctx.fillStyle = "red";
ctx.fillRect (0, 0, 50, 50);

ctx.fillStyle = "blue";
ctx.fillRect (350, 0, 50, 50);

ctx.fillStyle = "red";
ctx.fillRect (350, 350, 50, 50);

ctx.fillStyle = "blue";
ctx.fillRect (0, 350, 50, 50);
```

Week 1 Weekend Practice - REPS!

Round 1

Write a function `lengths` that accepts a single parameter as an argument, namely an array of strings. The function should return an array of numbers. Each number in the array should be the length of the corresponding string. To get you started, you'll need to **loop** through each *string* in the array and get the length of each one. Those lengths should be stored in a different array that you will return.

Remember that when building a function, you want to use **declare** a function with a **name** that accepts **arguments**. So for building our *function* called *lengths* that accepts an array of strings, it might look something like:

```
// declare function named "lengths"
// that accepts a arguments named "arrayOfStrings"
function lengths(arrayOfStrings) {

    // we can log out our "arrayOfStrings"
    console.log(arrayOfStrings);

    // now, we want to "return" something... but what?

    return whateverVariableYouWantToReturnHere;

}
```

- Reference: [MDN: String.length](#)

```
var words = ["hello", "what", "is", "up", "dude"]
lengths(words) # => [5, 4, 2, 2, 4]
```

Round 2

Write a Javascript function called `transmogrifier`. This function should accept three arguments, which you can assume will be numbers. Your function should return the "transmogrified" result

The transmogrified result of three numbers is the product (numbers multiplied together) of the first two numbers, raised to the power (exponentially) of the third number.

For example, the transmogrified result of 5, 3, and 2 is `(5 times 3) to the power of 2` is 225.

Use your function to find the following answers.

- Reference: [MDN: Math.pow\(\) for Exponential numbers](#)

```
transmogrifier(5, 4, 3)
transmogrifier(13, 12, 5)
transmogrifier(42, 13, 7)
```

Round 3

Write a function called `toonify` that takes two parameters, `accent` and `sentence`.

- If `accent` is the string `"daffy"`, return a modified version of `sentence` with all "s" replaced with "th".
- If the accent is `"elmer"`, replace all "r" with "w".
- Feel free to add your own accents as well!
- If the accent is not recognized, just return the sentence as-is.
- Reference: [MDN: String.replace\(\)](#)

```
toonify("daffy", "so you smell like sausage")
#=> "tho you thmell like thauthage"
```

Round 4

Write a function `wordReverse` that accepts a single argument, a string. The method should return a string with the order of the words reversed. Don't worry about punctuation. You'll need to use `String.split()` to create an array of words (splitting them with a space or " "). Then you'll need to reverse the order of that array using `array.reverse()`. Finally, you'll loop through them to create a new string).

- References:

- [MDN: String.split\(\)](#)
- [MDN: Array.reverse\(\)](#)

```
wordReverse("Now I know what a TV dinner feels like")
# => "like feels dinner TV a what know I Now"
```

Round 5

Write a function `letterReverse` that accepts a single argument, a string. The function should maintain the order of words in the string but reverse the letters in each word. Don't worry about punctuation. This will be very similar to round 4 except you won't need to split them with a space.

- References:

- [MDN: String.split\(\)](#)
- [MDN: Array.reverse\(\)](#)

```
letterReverse("Now I know what a TV dinner feels like")
# => "woN I wonk tahw a VT rennid sleef ekil"
letterReverse("Put Hans back on the line")
# => "tuP snaH kcab no eht enil"
```

Round 6

Write a function `longest` that accepts a single argument, an array of strings. The method should return the longest word in the array. In case of a tie, the method should return the word that appears first in the array.

- Reference: [MDN: String.length](#)

```
longest(["oh", "good", "grief"]) # => "grief"
longest(["Nothing", "takes", "the", "taste", "out", "of", "peanut", "butter", "quite"]
# => "unrequited"
```

Final Round

Write a function, called `repMaster`, that accepts two arguments, input and a function. Input should be able to be used with the function. The function used as an argument must return a string. `repMaster` should take the result of the string, passed as an argument to the argument function, and return this result with '`' proves that I am the rep MASTER!'` concatenated to it.

```
repMaster("Never give your heart to a blockhead", wordReverse) # =>
"blockhead a to heart your give never proves that I am the rep MASTER!"
repMaster("I finished this practice", toUpperCase);
"I FINISHED THIS PRACTICE proves that I am the rep MASTER!"
```

Note that a function can be used as an argument for a function! Inside of the function, you just need to **call** it. Example:

```
function logSomething(string) {
  console.log(string);
}

function doSomethingWithFunctions(yourFunction) {
  // you 'call' yourFunction
  var someString = "hey, you're going to log me";
  return yourFunction(someString);
}

doSomethingWithFunctions(logSomething);
```

Submitting this Assignment

- To submit this assignment create a pull request when you have completed this assignment.

Bonus Practice: Javascript Functions

```
// Question 1
// write a function square(x) that returns the argument passed multiplied by itself
// then write the functions cube(x), x times x times x,
// and quad(x), x times x times x times x, using the function square(x)
// verify that square(2) === 4, cube(3) === 27, and that quad(4) === 64

/* your code starts here */

/* your code ends here */

//Question 1 check
if (square(2) !== 4 || cube(3) !== 27 || quad(4) !== 256) {
  console.log("check question 1");
}

// Question 2
// write a function sum(numbers, twiceOrHalf).
// It should expect an array of numbers as the first argument
// and an optional flag, twiceOrHalf, as the second argument.
// If the flag is undefined, sum should return the total of the numbers in the array
// If the flag is truthy, it should return twice that sum
// If the flag is falsy but not undefined, it should return half that sum

/* your code starts here */

/* your code ends here */

//Question 2 check
var nums = [2, 4, 6];
if (sum(nums) !== 12 || sum(nums, false) !== 6 || sum(nums, !undefined) !== 24) {
  console.log("check question 2");
}

// Question 3
// write a function, max() that loops through zero or more arguments
// and returns the largest number in that list

/* your code starts here */

/* your code ends here */

//Question 3 check
if (max() !== undefined || max(15) !== 15 ||
  max(-1, 0) !== 0 || max(-1, -7, -4) !== -1) {
  console.log("check question 3");
}
```


Part 2: Objected Oriented JS & jQuery

2.1 Objects and CSS

Morning Exercise

- We're going to recap the REPS from the weekend.
- Source code is available in a section by itself.

Revisiting Objects

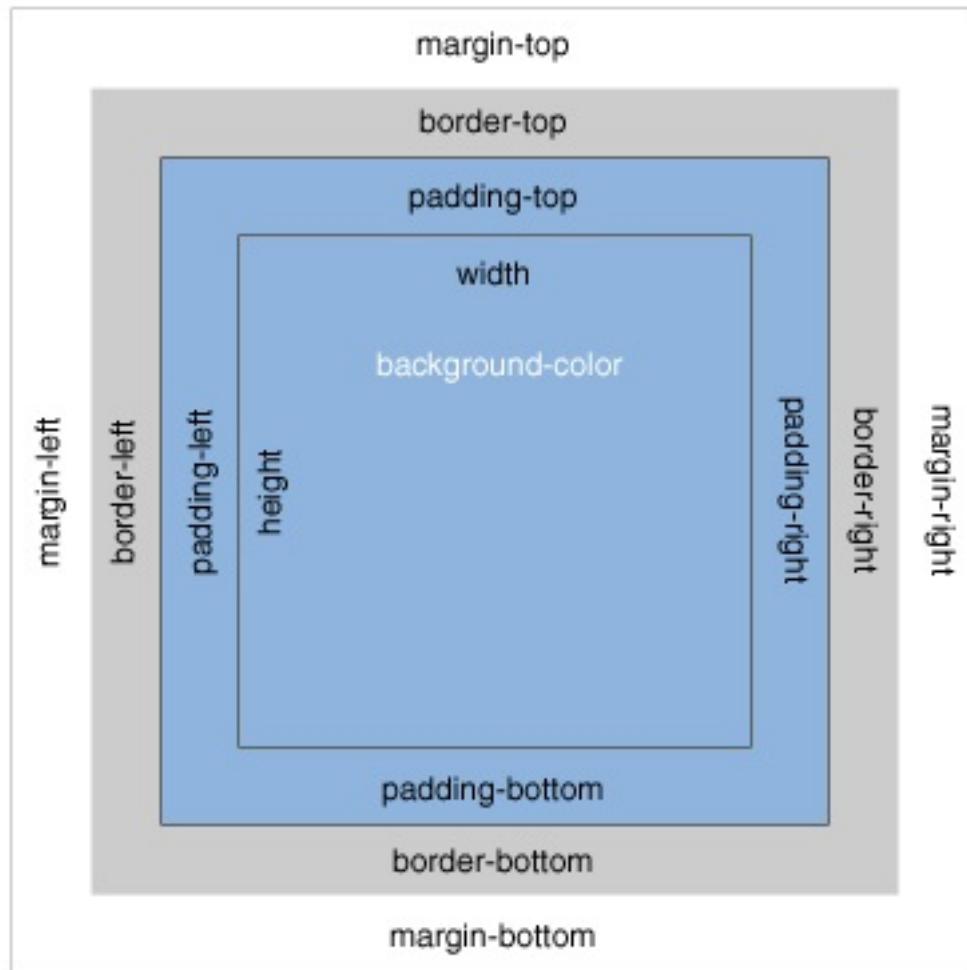
Objectives

- Revisit Object syntax and how to access properties
- Practice creating multiple Objects
- Model real world data as Objects
- Use Object methods

CSS Box Model

Objectives

- Explain the CSS Box Model
- Understand the difference between *margin*, *border*, and *padding*
- Describe the difference between absolute and relative positioning
- Build a wireframe outline using HTML and CSS



Agile Development, Wireframes, and User Stories

Objectives

- Explain the basics of Agile methodologies and why it's used
- Describe user stories and how they are different than to*dos
- Practice creating user stories for an example app

Slide Deck

- <https://presentations.generalassembly.ly/678c653e54647bc83a43#/>

Homework: Layout Challenges

You are provided with a PDF of "grey box" drawings. You must create HTML and CSS layouts for **each** of the layouts. Place each layout in a folder inside of `02..../your_name/day1_hw/`. You will need an HTML and CSS file for each layout (keep

things organized).

The PDF included "gray box" drawings of page layouts that are increasingly difficult to achieve using CSS. These are intended to be used after introducing students to the basics of `float` and `clear`. These exercises are meant to help you explore the basics and practice to develop a better intuitive sense for how `float` and `clear` work.

If you'd like, use the first two layouts as sample layouts to work through in a demonstration to explain the basics to the class. If used this way, I still recommend asking the students to repeat those two when they start their own work, as this will validate to them that they are building an understanding and still be useful practice before they work on the harder examples.

Please also note that we are not providing finished HTML and CSS for these exercises. We've found that HTML and CSS style can vary considerably from person to person, so please feel free to take your own approach to these quick layout wireframes.

2.1 Morning Exercise

We recapped the weekend practice homework.

Source Code

Below is one set of solutions for the weekend practice.

```
// declare function named "lengths"
// that accepts a arguments named "arrayOfStrings"
function lengths(arrayOfStrings) {

    // we can log out our "arrayOfStrings"
    console.log(arrayOfStrings);

    var strLengthArray = [];

    for (var inc in arrayOfStrings) {
        var tempLength = arrayOfStrings[inc].length;
        strLengthArray.push(tempLength);
    }

    // now, we want to "return" something... but what?
    return strLengthArray;

}

lengths(['kepler 22b', 'misa-et 980', 'pluto']);

// 2.

function transmogrifier(num1, num2, num3) {
    return Math.pow((num1 * num2), num3);
}
transmogrifier(42, 100, 9000);
transmogrifier(2, 4, 10);

// 3. toonify

function toonify(accent, sentence) {

    if (accent == 'daffy') {
        return sentence.replace(/s/g, 'th');
    } else if (accent == 'elmer') {
        return sentence.replace(/r/g, 'www');
    } else {
        return sentence;
```

```
}

}

toonify('elmer', 'ehh what\'s up doc? I smell rascally rabbits!');

// 4. word reverse

var topic = 'today we will cover CSS';
function wordReverse(words) {

    var tempArray = words.split(' ');
    console.log(tempArray);
    var reversed = tempArray.reverse();
    console.log(reversed)
    return reversed.join(" ");

}
wordReverse(topic);

// 5. letterReverse

function letterReverse(someStuff) {

    var tempArray = someStuff.split(' ');
    var finalSentence = '';

    for (var inc in tempArray) {

        var word = tempArray[inc];
        console.log(word);
        var splitWord = word.split('');
        console.log(splitWord);
        splitWord.reverse();
        word = splitWord.join('');
        console.log(word);
        finalSentence = finalSentence + ' ' + word;
        console.log(finalSentence);

    }

    return finalSentence;

}

// 6. longest
function longest (stringArray) {
    var compare = 0;
    for (var inc = 0; inc < stringArray.length; inc++) {
        if (stringArray[inc].length > compare) {
            compare = stringArray[inc].length;
            var longestWord = stringArray[inc];
        }
    }
}
```

```
        }
    }
    return longestWord;
}

// 7. rep master
function repMaster (input, aFunction) {
    var storedReturnValue;
    storedReturnValue = aFunction(input);
    if (typeof storedReturnValue === "string") {
        console.log(storedReturnValue + " proves I am the rep master!")
    } else {
        console.log("Give me a string input!")
    }
}
```

Building Object Methods

- We can add `abilities` (functions) to Objects
- When we assign a function to an Object, it becomes a **method**

```
var err = {
  name: 'Error',
  sayMyName: function() {
    return 'I am ' + this.name;
  },
  makeFunOfGreenClothes: function() {
    return "Your clothes look silly, little elf-man";
  },
  changeName: function(newName) {
    if (typeof(newName) == 'string') {
      var oldName = this.name;
      this.name = newName;
      return 'Name has been changed to: ' + newName + ' and our old name was ' + oldName;
    } else {
      return 'That name is not a valid string';
    }
  }
};
err.sayName();
err.changeName('Solution');
```

Object Methods & 'this'

hasOwnProperty

- `Object.hasOwnProperty('name-of-key');`
- Allows you to check an object for a specific key
- Returns true/false

Example:

```
var obj = {
  kittens: 'aww';
};
obj.hasOwnProperty('kittens'); // true
```

Object.keys()

- `Object.keys(nameOfObject);`
- Returns an array of every key that an Object has

Example:

```
var obj = {
  kittens: 'aww',
  puppies: 'adorable'
};
Object.keys(obj); // ['kittens', 'puppies']
```

'this'

- 'this' allows us to reference a base object
- Think of `this` as being able to jump up a root level in an object
- Further reading: <http://javascriptissexy.com/understand-javascripts-this-with-clarity-and-master-it/>

Sample HTML & CSS Layout

This layout uses floats to create a two column (content and sidebar) layout that may be used for any future project!

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Box Model Intro</title>
    <link rel='stylesheet' href='style.css'>
  </head>
  <body>

    <div id='wrapper'>
      <header id='logo'>
        Random Website Goes Here
      </header>
      <nav id='navbar'>
        Links to places will go here!
      </nav>
      <section id='content'>
        Lorem Ipsum Text
      </section>
      <aside id='sidebar'>
        <ul>
          <li>things</li>
          <li>to</li>
          <li>do</li>
        </ul>
      </aside>
      <footer id='goodbye'>
        Copyright nobody 2015...
      </footer>
    </div>

  </body>
</html>
```

style.css

```
/* ids are identifiable using a #name */
#wrapper {
  width: 90%;
  margin-left: auto;
  margin-right: auto;
  position: relative;
  /*margin: auto;*/
}

#logo {
  font-size: 3em;
  font-weight: bold;
  border: 3px dashed orange;
  margin: 3px;
}

#navbar {
  font-size: 1.5rem;
  font-weight: bold;
  border: 3px dashed purple;
  margin: 3px;
}

#sidebar {
  float: right;
  border: 3px dashed green;
  margin: 3px;
  width: 25%;
}

#content {
  float: left;
  border: 3px dashed navy;
  margin: 3px;
  width: 65%;
}

#goodbye {
  margin: 3px;
  border: 3px dashed red;
  clear: both;
}
```

2.1 Homework: Layout Challenges

You are provided with a PDF of "grey box" drawings. You must create HTML and CSS layouts for **each** of the layouts. Place each layout in a folder inside of `02.../your_name/day1_hw/`. You will need an HTML and CSS file for each layout (keep things organized).

[The PDF may be viewed here](#)

The PDF included "gray box" drawings of page layouts that are increasingly difficult to achieve using CSS. These are intended to be used after introducing students to the basics of `float` and `clear`. These exercises are meant to help you explore the basics and practice to develop a better intuitive sense for how `float` and `clear` work.

If you'd like, use the first two layouts as sample layouts to work through in a demonstration to explain the basics to the class. If used this way, I still recommend asking the students to repeat those two when they start their own work, as this will validate to them that they are building an understanding and still be useful practice before they work on the harder examples.

Please also note that we are not providing finished HTML and CSS for these exercises. We've found that HTML and CSS style can vary considerably from person to person, so please feel free to take your own approach to these quick layout wireframes.

Submission

Please submit your work via a pull request on Github. Submit as much as you have completed by midnight.

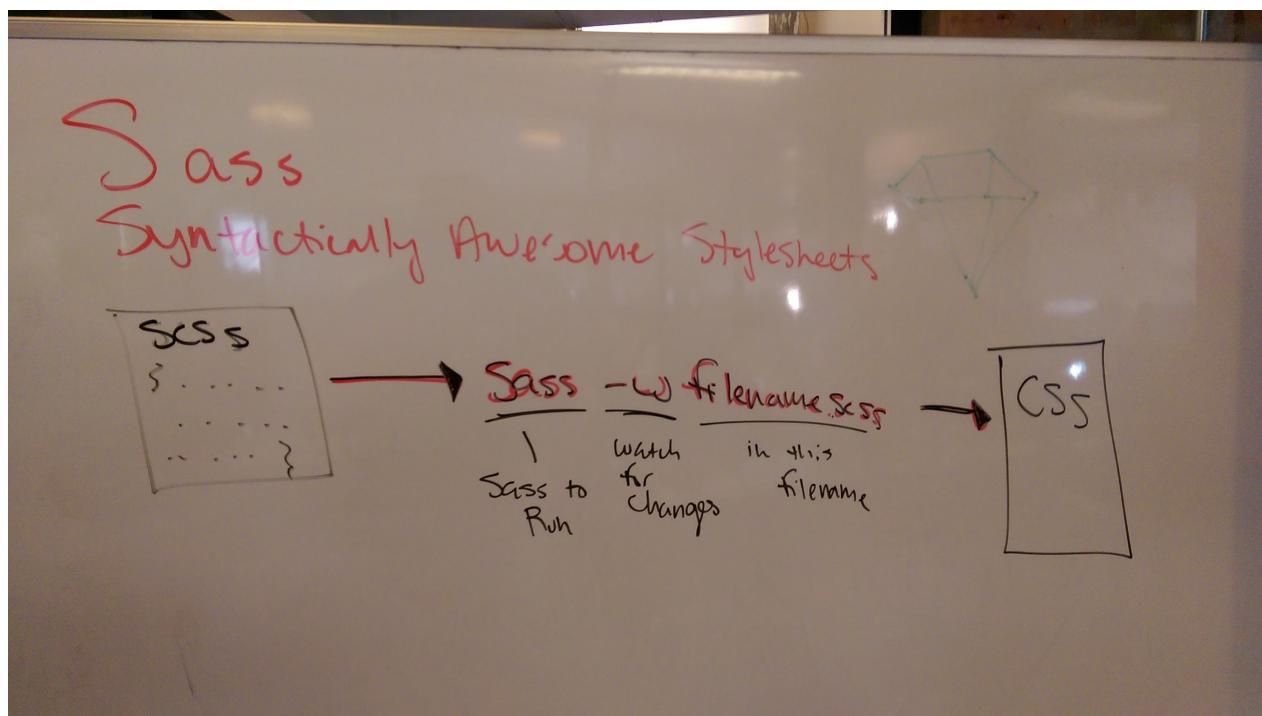
2.2 From Wireframe to Production

The theme of today's class will be learning how to compose a website from scratch. We'll discuss a variety of standards to help build a layout. We'll then explore how to build wireframes, what to look out for, and consider a few examples from live websites. Finally, we'll build our own wireframe and by the evening we'll build out a basic layout.

Sass

Learning Objectives

- Understand what a CSS pre-processor is
- Understand a strong use case for a CSS pre-processor
- Use Sass to include variables in your CSS
- Watch for changes in your Sass files and compile them to CSS

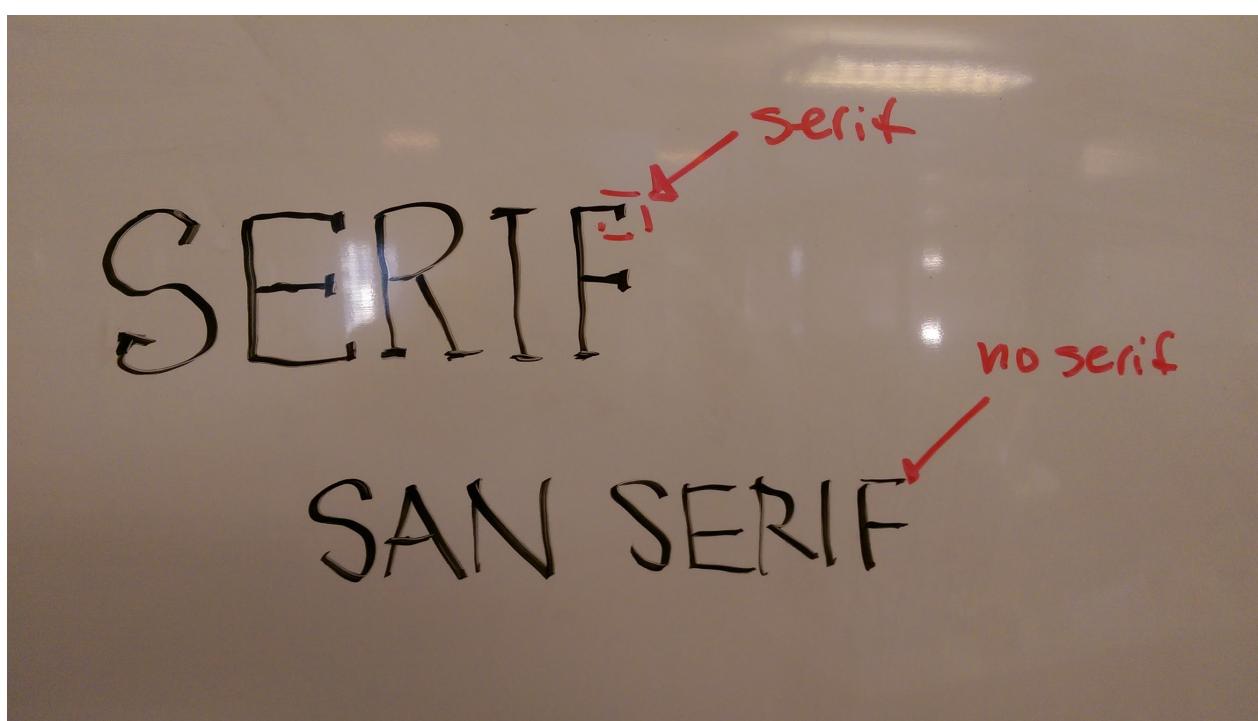
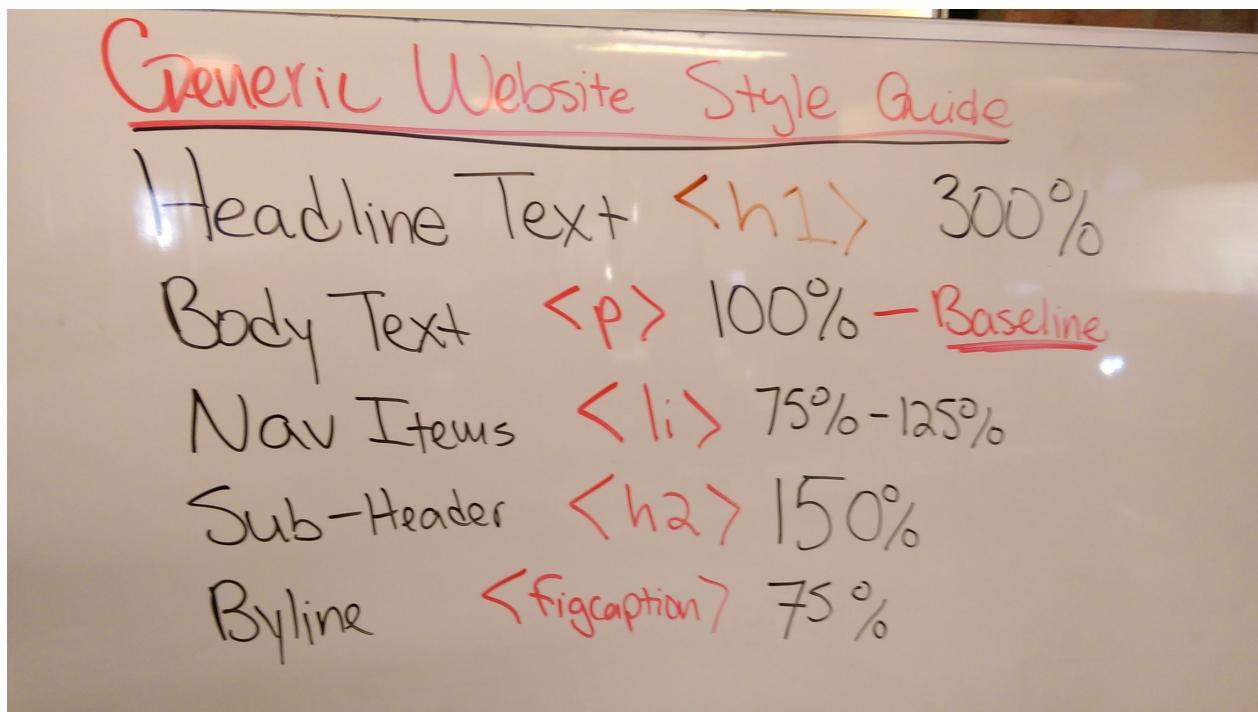


Web Typography

Learning Objectives

- know the differences between webfonts & desktop fonts
- find good webfonts to use in their projects
- incorporate webfonts into their own designs

- Understand how to Build a style guide



Advanced CSS

Learning Objectives

- Use `classes` in CSS
- Understand how to position elements
- Utilize pseudo-classes in CSS to create a hover effect
- Leverage `:before` and `:after`

Wireframes and Mockups

Learning Objectives

- Understand the need for wireframes, mockups and MVPs
- Create your own wireframe for a two-column layout
- Polish a mockup after wireframe is completed
- Describe which wireframe elements will be DOM elements

Lab/Homework: Create Website from Wireframe

- Create a new Github repository
- Take your wireframe concept and turn it into HTML & CSS
- Store all changes in this Github repository
- Take a screenshot and include it in your `Readme.md`. The code for this is `![Alt text](/path/to/img.jpg)`
- Post on Slack with a link to your repository by midnight.

2.2 Morning Exercise

- write out an array of five foods
- loop through the array and console.log each item with a message
- create an object with your name with 3 to 4 attributes including one array
- create a method for your object that accepts an argument that is an array and loops through the array and console.logs the array with a message!
- create a button in html
- style your button give it a border, color, radius, etc....(Maybee put it inside a div and center the dive on your screen)
- using javascript console.log my button is working when it is clicked (hint.... onclick method)
- Invoke your objects method using your button by passing in the object property that includes its array

HTML & CSS

Sass with SCSS

Installing Sass

To globally install Sass on your computer, run `gem install sass`.

Watching a File

We can watch our Sass files so they are automatically converted to usable CSS. To do so, we run the following command in the same folder as our SCSS files:

```
sass -w your_file.scss or sass --watch your_file.scss:compiled_css_file.css
```

Where we say **sass** please listen for changes in **your_file.scss** and output the changes to ***compiled_css_file.css**

Variables

To create and re-use variables:

```
$my_variable: 0px;
$my_other_variable: black;

body {
  font-size: $my_variable;
  color: $my_other_variable;
}
```

Importing other SCSS files

We can import other SCSS files for code organization!

variables.scss

```
$my_variable: 0px;  
$my_other_variable: black;
```

style.scss

```
@import 'variables.scss';  
  
body {  
  font-size: $my_variable;  
  color: $my_other_variable;  
}
```

Comments

```
// sass allows us to use JS style comments... these won't be compiled into the CSS fi
```

References

- Sass Guide: <http://sass-lang.com/guide>
- Sass Meister (validator): <http://sassmeister.com/>

Colour Theory

- Two types of colour - colours you can **touch** and color you **can't touch**.
- Touched colours are called *subtractive*; *additive* is colour you can't touch.
- Subtractive is measured in CMYK (cyan, magenta, yellow)
- Additive is measured by RGB (red, green, blue)
- Humans process color via HSL - hue, saturation, lightness

Emotions evoked from Colour

- Red - heat, passion, excitement.
- Orange- warmth, vitality.
- Yellow - optimism, creativity.
- Green - serenity, health.
- Blue - security, truth, stability.
- Purple - spirituality, intelligence, wealth.
- Pink - youthful, intensity.
- Brown - durability, class.
- Black - power, drama.
- White - simplicity, cleanliness.

Web Fonts

Typeface vs. Fonts

A **typeface** is the design of the letters. A **font** is that design written with code, wrapped up in what's considered a software package. So legally, a font is considered a piece of software. Which is why when you "buy" a font, you don't actually own it – you're just buying a license to install and use that software in a certain context.

Licensing

Licensing often doesn't work exactly the way you'd expect. You usually have to buy a license for the context you're using it in – *web*, *app/ebook*, or *desktop*.

Desktop fonts are meant to be installed locally on your computer, and are only allowed to be used in rasterized images. You're not allowed to embed the font in any way, you can only use it to make images of fonts.

App/ebook involves giving away a copy of the font to whomever needs to run your program, so often prices are ridiculously high.

Webfonts are meant to only be used on a website – they come in weird formats that are necessary for each browser to render them. You often have to estimate how much usage you need – like XXX,XXX monthly views on a page.

How to use webfonts

Assuming you've got all the necessary formats you need for multiple browsers (EOT, SVG, TTF, WOFF, WOFF2), all it takes is referencing them using a special CSS rule.

```

@font-face {
    font-family: 'League Spartan';
    src: url('leaguespartan-bold.eot');
    src: url('leaguespartan-bold.eot?#iefix') format('embedded-opentype'),
         url('leaguespartan-bold.woff2') format('woff2'),
         url('leaguespartan-bold.woff') format('woff'),
         url('leaguespartan-bold.ttf') format('truetype'),
         url('leaguespartan-bold.svg#league_spartanbold') format('svg');
    font-weight: bold;
    font-style: normal;
}

h1 { font-family:"League Spartan"; }

```

The **font-family** is whatever want to call it, though it makes sense to have it be the actual font family name. The src order matters, and the example giving is the current suggested order that is most compatible with modern browsers, to make sure each gets the file they need.

The **font-weight** & **font-style** are allowed to be set by you, according to the font. If you have multiple, say a regular weight & a bold weight, as long as you reference the same family name, you'll be to just change the weight or style in your future CSS calls and it'll work like any other font.

Where to find good webfonts

A few options – commercial, free, and open-source.

- [MyFonts](#) is a great marketplace for type designers to sell their stuff, so you see a wide variety of prices, from free to super expensive. They're good at organizing, and make it obvious & searchable to find fonts you can use on the web.
- [Typekit](#), which was recently bought by Adobe, is a subscription service that provides a library of awesome typefaces for use on the web.
- [FontSquirrel](#) is a great list of free fonts – some open-source, some just free. They've also got a webfont generator, where you can upload a font (assuming you're legally allowed to use it), and they'll convert it to all the formats you need and give you a ready-to-use kit.
- [Google Fonts](#) is a large collection of free fonts, some open-source, which they host and let you reference. It's very easy to use and include if you want your fonts hosted somewhere.

- **The League of Moveable Type**, started by yours truly, is the first open-source type foundry, whose fonts are not only free to use, but free to dissect and learn from, too.

Style Guide & Fonts

Rule of Thumb for Style Guides

Note that every style guide is unique. This is just a good starting point.

- Headline Text: 300%
 - B-Head (sub) text: 150%
 - Nav Item: 100%
 - Body copy (text): 100% - starting point could be `body { font-size: 16px }`
- Byline: 75%

Examples with Sass variables

- `1rem` = 100%
- `rem` is responsive and related to the `root` of the page.

```
// style guide variables

$headline-text: 3rem; // 300%
$body-text: 1rem; // 100%
$nav-item: 1.25rem; // 125%
$sub-header: 1.50rem; // 150%
$byline: 0.75rem; // 75%

body {
  font-size: 16px;
}
```

2.2 Homework

Website Repository

- Create a new Github repository
- Take your wireframe concept and turn it into HTML & CSS
- Store all changes in this Github repository
- Take a screenshot and include it in your `Readme.md`. The code for this is `![Alt text](/path/to/img.jpg)`
- Post on Slack with a link to your repository by midnight.

Optional (but valuable)

- Shay Howe, one of the product managers at Belly in Chicago wrote a book on HTML/CSS.
- It is available for free here: <http://learn.shayhowe.com/>
- We also have **3 copies** in our library if you like physical books.
- Tonight you should read **chapters 3 through 5**.

2.3 Creating User Interface Components

We are going to continue working on the website that you wireframed and created today. We're going to look at creating front end elements that can be interactive. To accomplish this, we need to use Object-Oriented Javascript.

Morning Exersize

- CSS Layouts (Redux)
- Classwide Q & A

DOM Manipulation and Events

We are going to dive right in to modifying the DOM based on user input.

Learning Objectives

- Select DOM elements using `Document.querySelector()`
 - Listen for user input with events
 - Mouse Input
 - Touch Input
 - Keyboard Input
 - Modify DOM elements as a reaction to events
- ```
> document.getElementsByTagName('body');
< [►<body>]
 ...</body>
> document.getElementsByTagName('body')[0];
< ►<body>
 ...</body>
> document.getElementsByTagName('li');
< [We are the champions (my friend),
 We are the champions (my friend),
 We are the champions (my friend)]
>
```

## UI Components

*We are going to build a component that is a form that accepts user input and monitors it.*

### Learning Objectives

- Create a UI component using an Object
- Teach the component how to know about itself using `state`
- Modify `state` based on events

## Blueprinting Objects with Constructors

*We will now dive into the heart of Object-Oriented programming.*

### Learning Objectives

- Understand that you can create a blueprint of an Object
- Describe the word `instantiate`
- Create new `instances` of Objects
- Describe how Objects can communicate with other Objects.

## Outcomes

### Goals

- Introduction to the local job market and industry
- Create a job hunt workflow for outcomes in Trello

## Lab/Homework: Add user input to your website

- 'Take what you have learned' - Yoda
- Tonight you need to add some form of user input to your website. This can be a game element, an interactive menu bar, etc.
- You should accomplish this by creating a component
- Use your newfound understanding of Constructors to build a blueprint for this component
- `instantiate` that component and verify that all input works
- In your `readme.md`, add a description of your UI Component's constructor and how it works
- `add` , `commit` , `push` to your repository and share the link to it in class



# Javascript Selectors

- **HTML** is the *skeleton* of a webpage; **Javascript** is the *muscle!* Javascript selectors allow us to create references to DOM Objects. Because DOM Objects are accessed you may modify the properties of these HTML elements directly.

## Creating Selectors

- `var selector = document.getElementById("news");`
- Getting by **Ids** (unique) and **Classes** (meant for re-use):
  - `document.getElementById("my-id");`
    - returns a single item.
  - `document.getElementsByClassName("navigation-item");`
    - returns array of items.
- use the `innerHTML` property to get or set values into a **selector**. For example:
  - `selector.innerHTML;` will return the innerHTML of a selector.
  - using `selector.innerHTML = "your text";` will replace the innerHTML.

## Assignment

You can assign the results of functions to a variable!

- `var answer = prompt("Did you sleep at all?");`
- `var selector = document.getElementById("main");`

## Types of Selectors

- `getElementById('string-id-name');` - Returns a single object
- `getElementsByTagName('ul');` - returns an array of all `ul` tags
- `getElementsByClassName('string-class-name');` - returns an array of all `.string-class-name`
- `querySelector('css-selector');` - returns a single object using CSS selector syntax
- `querySelectorAll('.my-class');` - returns an array of objects that use the CSS selector syntax

## Creating DOM Elements / Manipulating them

```
// basic selectors
// declare a selector named container
// access that container via document.getElementById('name-of'id)
var container = document.getElementById('container');
console.log(container);
var monsters = ['Wreck-it Ralph', 'The giraffe from Lion King SNES', 'Ganon'];

for (var baddie in monsters) {
 // create a new dom element using document.createElement('name-of-tag');
 var li = document.createElement('li');
 console.log(li);
 // access and assign a property to my dom element
 li.innerHTML = monsters[baddie];
 // append it to a container using selector.appendChild(domElement)
 container.appendChild(li);
}

// now, we need to create an image!
var kittenImage = document.createElement('img');
// alt text (alt) - ADA compliancy text for the blind
kittenImage.alt = 'A cute random kitten';
kittenImage.id = 'kitten';
// src = image source
kittenImage.src = 'http://vignette3.wikia.nocookie.net/clubpenguinpookie/images/d/d0/I
// append my element as a child to a selector
container.appendChild(kittenImage);
```

# Binding Events to DOM Elements

## Declare a DOM Element to bind to (using a selector)

```
var body = document.getElementsByTagName('body')[0];
```

## We need to add a listener for events to an element

### Mouse events

```
body.addEventListener('click', function(event) {
 console.log(event);
 console.log('ow, y u click me bro?');
});
```

- <https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent>

### Touch events

```
body.addEventListener('touchstart', function(event) {
 // console.log(event);
 // touchstart
 // touchmove
 // touchend
 console.log('yo yo dude y u pokin me? wtf man');
});
```

- [https://developer.mozilla.org/en-US/docs/Web/API/Touch\\_events](https://developer.mozilla.org/en-US/docs/Web/API/Touch_events)

### Keyboard events

```
body.addEventListener('keyup', function(event) {
 // look for specific keys to be pressed
 if (event.keyCode == 13) {
 console.log('y u press enter so much yo?');
 }
 console.log(event.keyCode);
});
```

- <https://developer.mozilla.org/en-US/docs/Web/Events/keyup>

- <http://www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes>
- [http://www.kirupa.com/html5/keyboard\\_events\\_in\\_javascript.htm](http://www.kirupa.com/html5/keyboard_events_in_javascript.htm)

# Components 101

- Each component has a DOM element that contains information to be shown to the user
- Each component has an `initialize` function to set up the component
- Each component has a `render` function that is used to update/display content to the user
- A component should be contained and be able to know about itself (by checking `attributes`)

```
var component = {
 domElement: null,
 initialize: function(selector) {
 // create a dom element
 this.domElement = document.createElement('div');
 // attach it
 selector.appendChild(this.domElement);
 },
 render: function(statusText) {
 // update the dom element
 this.domElement.innerHTML = statusText;
 }
};
```

## User Component Example

```
// create a user interface component!
// the goal here is to create an Object
// that can update itself
// and visually show that if needed

// ex #1: user component
var user = {
 name: null,
 score: 0,
 domElement: null,
 // elementToAppendTo: document.selector for an individual element
 initialize: function(elementToAppendTo) {
 if (this.name == null) {
 this.name = prompt('What is your name?');
 }
 this.domElement = document.createElement('div');
 elementToAppendTo.appendChild(this.domElement);
 console.log('initialize: complete');
 },
 // innerHTML: valid html to place in our domElement
 render: function(innerHTML) {
 if (typeof(innerHTML) == 'string') {
 this.domElement.innerHTML = innerHTML;
 }
 },
 buildPlayerStatusString: function() {
 return this.name + ': ' + this.score;
 },
 getName: function() {
 return this.name;
 },
 saveName: function(newName) {
 if (typeof(newName) == 'string' && newName.length > 0) {
 this.name = newName;
 } else {
 alert('You entered an incorrect or empty name');
 }
 },
 getScore: function() {
 return this.score;
 },
 updateScoreByOnePoint: function() {
 this.score = this.score + 1;
 var status = this.buildPlayerStatusString();
 this.render(status);
 return this.score;
 }
};
```

## Another Example

```
var comp = {
 domElement: null,

 initialize: function(domSelector) {
 console.log('initializing component');
 this.domElement = document.createElement('img'); //
 domSelector.appendChild(this.domElement);
 },

 render: function(imageSrc) {
 this.domElement.src = imageSrc; //
 }

};

var body = document.getElementsByTagName('body')[0];
console.log(body);
body.innerHTML = '';
var ponyImg = 'http://www.animalsbase.com/wp-content/uploads/2015/06/Pony.jpg';
comp.initialize(body);
comp.render(ponyImg);
```

# Constructors

## Description

Constructors are blueprints to create objects. We define a function that accepts arguments. We then create a new **instance** of an object (through **instantiation**).

## In-Class Examples

```
// the giver (todo: read)

function person(name, age, fact) {

 // assign properties to an object
 this.name = name;
 this.age = parseInt(age);
 this.fact = fact;

}

// bike constructor
// it creates an object
// a constructor is a blueprint to construct an object
// speeds, colour, size, price, brand
function bike(speeds, colour, size, price, brand) {

 // attributes
 this.speeds = speeds;
 this.colour = colour;
 this.size = size;
 this.price = price;
 this.brand = brand;

 // abilities
 this.toString = function() {
 return 'This bike has ' + this.speeds + ' and is ' + this.colour;
 }
}

// declare a variable called annasBike
// create a 'new' INSTANCE of 'bike'
// create a new copy of bike
var annasBike = new bike(21, 'teal', 'small', 350, 'diamondback');
annasBike.toString();
var jamesBike = new bike(6, 'white', 'medium', 200, 'biria');
jamesBike.toString();
```

# Component Constructor Boilerplate

```
function component(domElement) {

 this.domElement = domElement;
 this.initialize = function() {
 console.log('init');
 };
 this.render = function() {
 console.log('render');
 }
}
}
```

## 2.3 Videos

- Selector Recap: <https://youtu.be/dlRXcwgcigQ>
- User Interface Component Recap: <https://youtu.be/bp6HzzRtkjc>

## 2.3 Homework

### Lab/Homework: Add user input to your website

- 'Take what you have learned' - Yoda
- Tonight you need to add some form of user input to your website. This can be a game element, an interactive menu bar, etc.
- You should accomplish this by creating a component
- Use your newfound understanding of Constructors to build a blueprint for this component
- `instantiate` that component and verify that all input works
- In your `readme.md`, add a description of your UI Component's constructor and how it works
- `add` , `commit` , `push` to your repository and share the link to it in class

## 2.4 Write Less & Do More avec jQuery

*Yesterday we spent a lot of time using vanilla Javascript to modify web pages. We created selectors that allowed us to access content on DOM elements and modify them. Today, we're going to show you an easier way to write selectors. By the end of the day, you will write less... and do more.*

### Morning Exercise

- Selector and Component Recap
- Tips and tips to succeeding in WDI

### Intro to jQuery

#### Objectives

- Describe jQuery and the context to use it
- Include jQuery in your projects
- Practice using jQuery selectors

### jQuery Effects and Animations

#### Objectives

- Animate and move DOM elements using jQuery
- Fade elements in and out
- Use `.toggleDisplay` and `.css` to transform elements

### Intermediate jQuery

- Use jQuery API to look up methods
- Use methods to manipulate selectors
- Modify attributes of dom elements

#### Objectives

### Lab: Stepping through Code

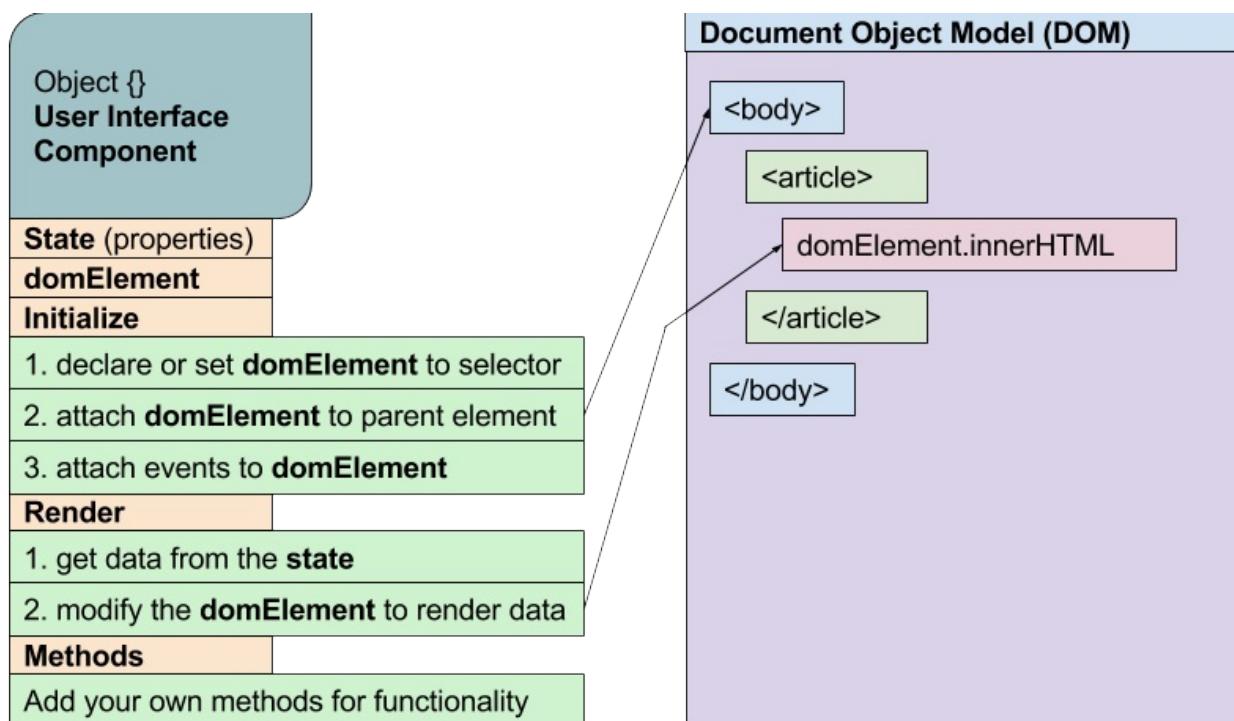
- Use Google Chrome to debug like a pro!
- Understand what breakpoints are and how to use them
- Step through source code examples

## Objectives

## Homework

- We are assigning a two-night assignment.
- We do not expect everyone to finish tonight but protip: stay during after hours ad lab time to work with others on this.
- You are to work on Fellowship tonight and over the weekend for jQuery practice.
- jQuery Fellowship

# Components Recap



- Video: <https://www.youtube.com/watch?v=CpVz9Z7mqQA>
- Source Code: <https://jsfiddle.net/qf8tsy3y/1/>

# jQuery

- jQuery's motto is *write less, do more*
- jQuery was originally written by John Rezig
- It is now maintained by a strong community
- There is even a yearly jQuery conference in Toronto
- <http://jquery.com>

## Define what a library vs framework is

- **Library** is set of methods to be utilized
- **framework** is a combination of multiple libraries and tools.
- Various libraries include:
  - jQuery: <http://jquery.com>
  - Zepto: <http://zeptojs.com/>
  - Prototype: <http://prototypejs.org/>
  - Modernizr: <http://modernizr.com/>

## List primary components of jQuery

- DOM traversal
- Event Handling
- Ajax interactions
- Event Handling

## How to include jQuery into an app

- Include using CDN or local file
- See the **Installing jQuery with NPM** section below
- CDNS:
  - <http://cdnjs.com>
  - <http://googleapis.com>

## Select existing DOM nodes

| Example          | ... in jQuery                                           |
|------------------|---------------------------------------------------------|
| Elements         | <code>\$(“element”), `\$(“#id”) or `\$(“.class”)</code> |
| Descendants      | <code>\$(“#id descendant”)</code>                       |
| Children         | <code>\$(“.class &gt; child”)</code>                    |
| Multiple         | <code>\$(“#id1, #id2, #id3”)</code>                     |
| Pseudo-selectors | <code>\$(“li :first”)</code>                            |

## Build and append custom DOM nodes

- `.append()`
- `.appendTo()`
- `.text()`
- `.prepend()`
- `.prependTo()`
- `.html()`

## Trigger and listen to events

```
$(selector).click(function() {
 // your code
});
```

- <https://api.jquery.com/category/events/>

## Using jQuery in your html head element

- In your index.html file, add: `<script type='text/javascript' src='scripts/jquery.js'></script>`

## Resources & Further Reading

- [jQuery Selector Reference](#)
- [jQuery Cheat Sheet Reference](#)
- [jQuery Succinctly](#)
- [jQuery Basics](#)
- [Try jQuery \(Interactive\)](#)



# jQuery Boilerplate

```
<!DOCTYPE html>
<html>
<head>
 <title></title>
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></sc
 <script src="app.js"></script>
</head>
<body>

</body>
</html>
```

## **\$(document).ready()**

```
$(document).ready(function(event){ ... }) is similar to window.onload = function() {
... }
```

### **Example**

```
$(document).ready(function(){
 // things to do when the page has fully loaded
});
```

# Trip To MorDOMr using jQuery

## Learning Objectives

- Practice using **jQuery** to manipulate the DOM

We are going to take a trip from the Shire, through Rivendell, across Middle Earth, and into the heart of Mordor itself, Mount Doom. Pack up, because we're going on an adventure.

Your goal is to use jQuery to complete this adventure! Instead of using traditional Javascript like the weekend's prompt, you should use jQuery selectors to manipulate the DOM. This second pass on Fellowship should give you a greater appreciation of the ease and simplicity of jQuery compared to standard Javascript.

You will likely need to research methods using the jQuery API documentation. Please check the resources in this Gitbook as well as at the end of this assignment for places to look for resources.

## Directions

1. We have placed a homework project in our class repository. `git pull class master` .
2. Check out `02_oojs_jquery/your_name/fellowship` . This is where you will work. Starter data has been provided. There is an `index.html` and a few other files to work with.
3. Define and call the functions outlined below.
4. When you complete the assignment, please `add` , `commit` , `push origin master` and create a Pull Request on Github.

## Resources

- [jQuery Selector Reference](#)
- [jQuery Cheat Sheet Reference](#)
- [jQuery Succinctly](#)
- [jQuery Basics](#)
- [Try jQuery \(Interactive\)](#)

====

## Part 1

```
var makeMiddleEarth = function () {
 // create a section tag with an id of `middle-earth`
 // add each land as an `article` tag
 // inside each `article` tag include an `h1` with the name of the land
 // append `middle-earth` to your document `body`
};

makeMiddleEarth();
```

## Part 2

```
var makeHobbits = function () {
 // display an `unordered list` of hobbits in the shire
 // (which is the second article tag on the page)
 // give each hobbit a class of `hobbit`
};

makeHobbits();
```

## Part 3

```
var keepItSecretKeepItSafe = function () {
 // create a div with an id of `the-ring`
 // give the div a class of `magic-imbued-jewelry`
 // add the ring as a child of `Frodo`
};

keepItSecretKeepItSafe();
```

## Part 4

```
var makeBuddies = function () {
 // create an `aside` tag
 // attach an `unordered list` of the `buddies` in the aside
 // insert your aside as a child element of `rivendell`
};

makeBuddies();
```

## Part 5

```
var beautifulStranger = function () {
 // change the `Strider` text to `Aragorn`
};

beautifulStranger();
```

## Part 6

```
var leaveTheShire = function () {
 // assemble the `hobbits` and move them to `rivendell`
};
```

## Part 7

```
var forgeTheFellowShip = function () {
 // create a new div called `the-fellowship` within `rivendell`
 // add each `hobbit` and `buddy` one at a time to `the-fellowship`
 // after each character is added make an alert that they // have joined your party
};
```

## Part 8

```
var theBalrog = function () {
 // change the `Gandalf` text to `Gandalf the White`
 // apply the following style to the element, make the // background 'white', add a
};
```

## Part 9

```
var hornOfGondor = function () {
 // pop up an alert that the horn of gondor has been blown
 // Boromir's been killed by the Uruk-hai!
 // Remove `Boromir` from the Fellowship
};
```

## Part 10

```
var itsDangerousToGoAlone = function () {
 // take `Frodo` and `Sam` out of the fellowship and move // them to `Mordor`
 // add a div with an id of `mount-doom` to `Mordor`
};
```

## Part 11

```
var weWantsIt = function () {
 // Create a div with an id of `'gollum'` and add it to Mordor
 // Remove `the ring` from `Frodo` and give it to `Gollum`
 // Move Gollum into Mount Doom
};
```

## Part 12

```
var thereAndBackAgain = function () {
 // remove `Gollum` and `the Ring` from the document
 // Move all the `hobbits` back to `the shire`
};
```

## Second Pass Friday

### UX Collaboration

- 9:00am to 10:30am
- We will meet in classroom #3
- Our User Experience Design Immersive (UXDI) will teach us...
- **Ideation!**

### Second Pass Friday

- 10:45am until 12:50pm
- Second pass Friday as usual
- Please create issues on Github for questions you'd like answered

### Weekend Practice

- ...soon...

## Part 3: Advanced Front-End

Over the past two weeks we have covered a lot of content. Some of the things we have covered are:

- Javascript and programming fundamentals
- Object-oriented programming techniques
- Basic HTML and CSS
- Fundamental design layout and structure for websites
- How to manipulate the DOM using Javascript **and** jQuery
- To listen for events and do things when they are triggered
- How to include resources on your page (such as CSS and Javascript)
- How to use variables in CSS using Sass to precompile CSS

## Javascript

### jQuery \$.ajax()

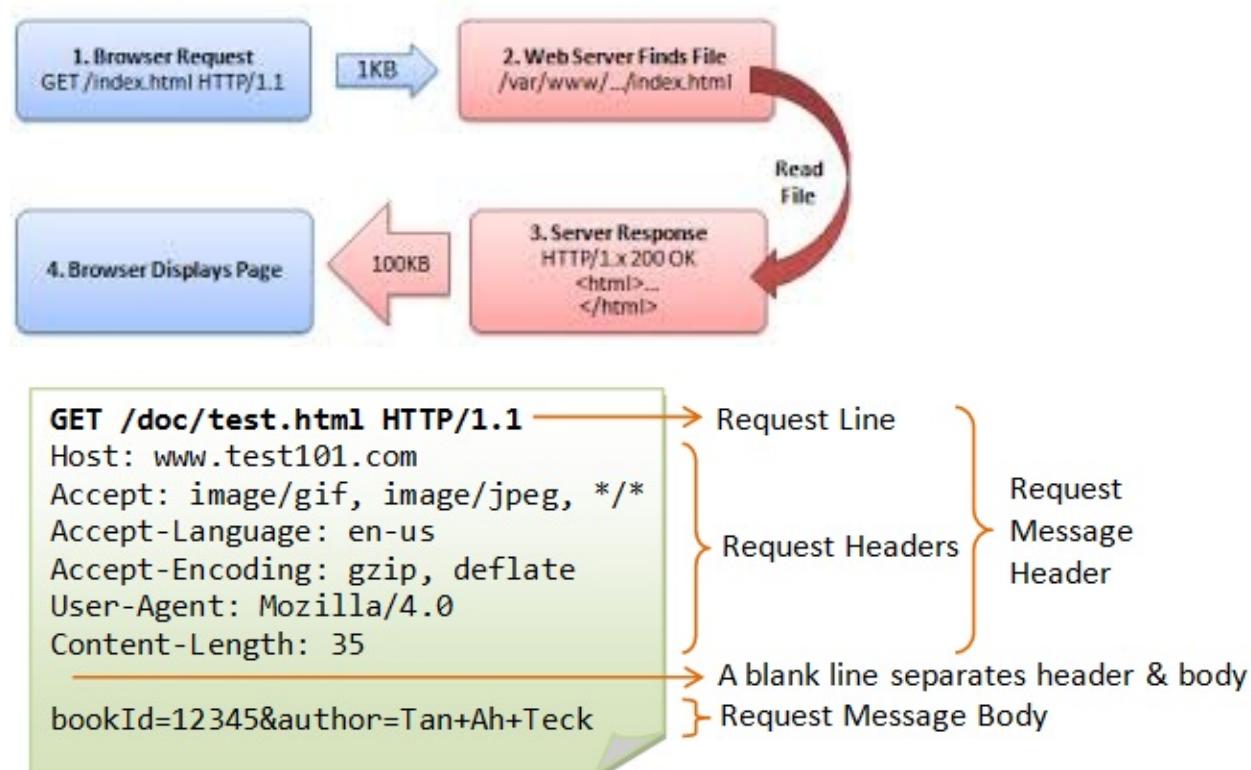
## Summary

"Ajax" refers to the ability of JavaScript to make HTTP requests asynchronously (a.k.a. "in the background") and act on the results directly without having the browser load a new page. Just like DOM manipulation and event handling, jQuery gives us a high-level interface to the browser's Ajax capabilities through the `$.ajax` function.

Most Ajax interactions use the JSON data format for requests and responses.

## Request

### HTTP Request and Response



## Standard Example

```
$.ajax({
 url: 'http://localhost:8000/animals',
 type: 'POST',
 dataType: 'json',
 data: {"animal": {
 "name": animal,
 "sound": sound}
 }})
});
```

## OMDBI Example

```
$.ajax({
 url: 'http://www.omdbapi.com/?t=Star+Wars&y=&plot=short&r=json',
 type: "GET",
 dataType: 'json',
 success: function(data) {
 $('body').append(data.Title + " was released in " + data.Year + "<hr>
");
 }
});
```

## Shake-it-speare

```
$.ajax({
 url: 'http://shakeitspeare.com/api/poem',
 type: "GET",
 dataType: 'json',
 success: function(data) {
 var data = data;
 //console.log(data);
 $('body').append(data.poem);
 //alert("huzzah! we did it guys!");
 },
 fail: function(error) {
 console.log(error);
 }
});
```

## POST examples

```
var animal = $("#animalName").val();
var sound = $("#animalSound").val();

// alert(animal);

$.ajax({
 url: 'http://localhost:8000/animals',
 type: 'POST',
 dataType: 'json',
 data: {"animal":
 {"name": animal,
 "sound": sound}
 })
});
```

# Ajax in Depth

We can **GET** and **POST** resources to the web using *Ajax!*

```
// $.ajax
// GET: get resources from a web
// POST: send /submit data somewhere (POST a tweet or status)
```

Remember... Ajax is just a method of jQuery!

```
$.ajax(); // ajax is a method of jQuery
```

And that accepts an **arguments** object!

```
// arguments {} for jQuery
{
 type: 'get', // OR 'post'
 url: 'http://somewhere.com/api/stuff', // url
 dataType: 'json', // also could be 'xml', etc..
 // POST only...
 data: {} // send
}
```

Let's add the object as the argument to the jQuery method...

```
// add arguments - argument for Ajax is a JS {}
$.ajax({
 type: 'get',
 url: 'http://somewhere.com/api/stuff',
 dataType: 'json'
});
```

I can refer to that argument as a variable for ease of use

```
var ajaxArgument = {
 type: 'get',
 url: 'http://api.openweathermap.org/data/2.5/forecast/city?id=524901&APPID=11111111111111111111111111111111',
 dataType: 'json',
 success: function(data) {
 console.log("success");
 console.log(data);
 },
 error: function(error) {
 console.log("error")
 console.log(error);
 }
};
// make the ajax call
$.ajax.ajaxArgument;
```

I can also call .done and .fail if I prefer to chain methods...

```
// or...
$.ajax.ajaxArgument).done(function(data) {
 console.log(data);
});
$.ajax.ajaxArgument).fail(function(error) {
 console.log(error);
});
```

**Example:** Add a loading spinner icon when a request is made and remove it when the request is done.

```
// how do I add a spinner or some sort of icon to show loading?

// do some code to show a spinner...
$('.spinner').show(); // <div class="spinner">.....
$.ajax({
 type: 'get',
 url: 'http://imperialholonet.herokuapp.com',
 dataType: 'json',
 success: function(data) { // data is the data from our server
 console.log(data);
 // some code to success message!
 $('.success').show(); // <div class="success">.....
 $('.spinner').hide(); // <div class="spinner">.....
 },
 error: function(error) {
 console.log(error);
 // some error code...
 $('.wompwomp').show(); // <div class="wompwomp">.....
 $('.spinner').hide(); // <div class="spinner">.....
 },
});
});
```

## What is this `.done()` anyways?

```
// using .done()
// only should be used when you own the server for the API
// not if you rely on someone else for data!
var ajaxArgument = {
 type: 'get',
 url: 'http://api.openweathermap.org/data/2.5/forecast/city?id=524901&APPID=11111111111111111111111111111111',
 dataType: 'json'
};
$.ajax(ajaxArgument).done(function(data) {
 console.log('whoa now, we\'re done..');
 console.log(data);
});
```

## We can also shorthand Ajax **GET** requests...

```
// what about... .getJSON?
// shorthand 'GET' request method
$.getJSON("url", function(data) {
 // do stuff with your data
 console.log(data);
});
```

Now, let's take a look at what a *closure* is...

```
// closure is a way to access data inside of a
// scope that no longer exists
var ajaxArgument = {
 type: 'get',
 url: 'http://api.openweathermap.org/data/2.5/forecast/city?id=524901&APPID=11111111111111111111111111111111',
 dataType: 'json',
 success: function(data) {
 console.log("success");
 console.log(data);
 },
 error: function(error) {
 console.log("error")
 console.log(error);
 }
};
var oldAjax = $.ajax.ajax(ajaxArgument); // assign ajax to
console.log(oldAjax.responseJSON); // closure data
```

# Fellowship (Journey to MorDOMr) Solution

Below is one of *many* solutions to this weekend's homework.

```
$(function() {

 console.log("Linked.");

 // Dramatis Personae
 var hobbits = [
 'Frodo Baggins',
 'Samwise \'Sam\' Gamgee',
 'Meriadoc \'Merry\' Brandybuck',
 'Peregrin \'Pippin\' Took'
];

 var buddies = [
 'Gandalf the Grey',
 'Legolas',
 'Gimli',
 'Strider',
 'Boromir'
];

 var lands = ['The Shire', 'Rivendell', 'Mordor'];

 // we don't need a var to hold body with jQuery.
 // var body = document.querySelector('body');

 // Chapter 1
 function makeMiddleEarth() {
 // create a section tag with an id of middle-earth
 var middleEarth = $('<section>');
 for(var i = 0; i < lands.length; i++) {
 // add each land as an article tag
 var land = $('<article>');
 // inside each article tag include an h1 with the name of the land
 var landName = $('<h1>');
 landName.text(lands[i]);
 land.append(landName);
 middleEarth.append(land);
 }
 // append middle-earth to your document body
 $('body').append(middleEarth);
 }

 makeMiddleEarth();
});
```

```

var theShire = $('article').eq(0);
var rivendell = $('article').eq(1);
var mordor = $('article').eq(2);

// Chapter 2
function makeHobbits() {
 // display an unordered list of hobbits in the shire (which is the first article
 var hobbitList = $('

');
 for(var i = 0; i < hobbits.length; i++) {
 // give each hobbit a class of hobbit
 var hobbit = $('- ');
 hobbit.addClass('hobbit');
 hobbit.text(hobbits[i]);
 hobbitList.append(hobbit);
 }
 theShire.append(hobbitList);
}

makeHobbits();

var frodo = $('li').eq(0);

// Chapter 3
function keepItSecretKeepItSafe() {
 // create a div with an id of 'the-ring'
 var theRing = $('

');
 theRing.attr('id', 'the-ring');
 // give the div a class of 'magic-imbued-jewelry'
 theRing.addClass('magic-imbued-jewelry');
 // add an event listener so that when a user clicks on the ring, the nazgulScreech
 // theRing.addEventListener('click', nazgulScreech);
 // add the ring as a child of Frodo
 frodo.append(theRing);
}

keepItSecretKeepItSafe();

// Chapter 4
function makeBuddies() {
 // create an aside tag
 var aside = $('');
 var buddyList = $('
');
 for(var i = 0; i < buddies.length; i++) {
 // attach an unordered list of the 'buddies' in the aside
 var buddy = $(' - ');
 buddy.text(buddies[i]);
 buddyList.append(buddy);
 }
 // insert your aside as a child element of rivendell
 aside.append(buddyList);
 rivendell.append(aside);
}

```

```
}

makeBuddies();

var strider = rivendell.find('li').eq(3);

// Chapter 5
function beautifulStranger() {
 // change the 'Strider' text to 'Aragorn'
 strider.text('Aragorn');
}

beautifulStranger();

var hobbits = theShire.find('ul').eq(0);

// Chapter 6
function leaveTheShire() {
 // assemble the hobbits and move them to Rivendell
 rivendell.append(hobbits);
}
leaveTheShire();

var fellowshipMembers = rivendell.find('li');

// Chapter 7
function forgeTheFellowShip() {
 // create a new div called 'the-fellowship' within rivendell
 var theFellowship = $('

');
 theFellowship.attr('id', 'the-fellowship');
 for(var i = 0; i < fellowshipMembers.length; i++) {
 theFellowship.append(fellowshipMembers.eq(i));
 alert(fellowshipMembers.eq(i).text() + ' has joined the fellowship!');
 }
 // add each hobbit and buddy one at a time to 'the-fellowship'
 // after each character is added make an alert that they have joined your party
 rivendell.append(theFellowship);
}

forgeTheFellowShip();

var gandalf = fellowshipMembers.eq(0);

// Chapter 8
function theBalrog() {
 // change the 'Gandalf' textNode to 'Gandalf the White'
 gandalf.text('Gandalf the White');
 // apply style to the element
 gandalf.css('border', '3px solid slategrey');
 // make the background 'white', add a grey border
 gandalf.css('backgroundColor', 'white');


```

```
}

theBalrog();

var boromir = fellowshipMembers.eq(4);

// Chapter 9
function hornOfGondor() {
 alert('the horn of gondor has blown');
 // pop up an alert that the horn of gondor has been blown
 // put a linethrough on boromir's name
 boromir.css('text-decoration', 'line-through');
 alert('Boromir\'s been killed by the Uruk-hai!');
 // Remove Boromir from the Fellowship
 rivendell.append(boromir);
}

hornOfGondor();

var sam = fellowshipMembers.eq(6);

// Chapter 10
function itsDangerousToGoAlone() {
 // take Frodo and Sam out of the fellowship and move them to Mordor
 mordor.append(frodo);
 mordor.append(sam);
 // add a div with an id of 'mount-doom' to Mordor
 var mountDoom = $('

');
 mountDoom.attr('id', 'mount-doom');
 mordor.append(mountDoom);
}

itsDangerousToGoAlone();

var gollum, theRing;

// Chapter 11
function weWantsIt() {
 // Create a div with an id of 'gollum' and add it to Mordor
 gollum =($('

Fellowship Solution

157


```

```
function thereAndBackAgain() {
 gollum.remove();
 // remove Gollum and the Ring from the document
 var hobbitUL = $('');
 var hobbits = $('.hobbit');
 for(var i = 0; i < hobbits.length; i++){
 hobbitUL.append(hobbits.eq(i));
 }
 theShire.append(hobbitUL);
 // Move all the hobbits back to the shire

}

thereAndBackAgain();

function golemGotCash() {
 var bling = $('<div>');
 bling.attr('id', 'lord-of-the-bling');
 $('body').append(bling);
}

golemGotCash();

});
```



# Project #1: The Game

## Overview

Let's start out with something fun - **a game!**

Everyone will get a chance to **be creative**, and work through some really **tough programming challenges** – since you've already gotten your feet wet, it's up to you to come up with a fun and interesting game to build.

**You will be working individually for this project**, but we'll be guiding you along the process and helping as you go. Show us what you've got!

---

## Technical Requirements

Your app must:

- **Render a game in the browser**
  - **Switch turns** between two players
  - **Design logic for winning & visually display which player won**
  - **Include separate HTML / CSS / JavaScript files**
  - Stick with **KISS (Keep It Simple Stupid)** and **DRY (Don't Repeat Yourself)** principles
  - Use **Javascript or jQuery for DOM manipulation**
  - **Deploy your game online**, where the rest of the world can access it
  - Use **semantic markup** for HTML and CSS (adhere to best practices)
- 

## Necessary Deliverables

- A **working game, built by you**, hosted somewhere on the internet
- A **link to your hosted working game** in the URL section of your Github repo. This must be placed inside of `your_username.github.io` in a folder called `project1`. It will then be accessible via [http://your\\_username.github.io/project1/](http://your_username.github.io/project1/)
- A **git repository hosted on Github**, with a link to your hosted game, and frequent

commits dating back to the very beginning of the project. We strongly encourage you to work out of the repository `your_username.github.io`

- A `readme.md` file with explanations of the technologies used, the approach taken, installation instructions, unsolved problems, etc.
- 

## Suggested Ways to Get Started

- **Break the project down into different components** (data, presentation, views, style, DOM manipulation) and brainstorm each component individually. Use whiteboards!
  - **Use your Development Tools** (`console.log`, inspector, alert statements, etc) to debug and solve problems
  - Work through the lessons in class & ask questions when you need to! Think about adding relevant code to your game each night, instead of, you know... *procrastinating*.
  - **Commit early, commit often.** Don't be afraid to break something because you can always go back in time to a previous version.
  - **Consult documentation resources** (MDN, jQuery, etc.) at home to better understand what you'll be getting into.
  - **Don't be afraid to write code that you know you will have to remove later.** Create temporary elements (buttons, links, etc) that trigger events if real data is not available. For example, if you're trying to figure out how to change some text when the game is over but you haven't solved the win/lose game logic, you can create a button to simulate that until then.
- 

## Potential Project Ideas

### Blackjack

Make a one player game where people down on their luck can lose all their money by guessing which card the computer will deal next!

### Concentration

Sometimes just called "Memory", it's a card game in which all of the cards are laid face down on a surface and two cards are flipped face up over each turn. If you get all the matching cards, you've won!

## Self-scoring Trivia

Test your wits & knowledge with whatever-the-heck you know about (so you can actually win). Guess answers, have the computer tell you how right you are!

---

## Useful Resources

- [MDN Javascript Docs](#) (*a great reference for all things Vanilla Javascript*)
  - [jQuery Docs](#) (*if you're using jQuery*)
  - [Github Pages](#) (*for hosting your game*)
- 

## Project Feedback + Evaluation

- **Project Workflow:** Did you complete the user stories, wireframes, task tracking, and/or ERDs, as specified above? Did you use source control as expected for the phase of the program you're in (detailed above)?
- **Technical Requirements:** Did you deliver a project that met all the technical requirements? Given what the class has covered so far, did you build something that was reasonably complex?
- **Creativity:** Did you add a personal spin or creative element into your project submission? Did you deliver something of value to the end user (not just a login button and an index page)?
- **Code Quality:** Did you follow code style guidance and best practices covered in class, such as spacing, modularity, and semantic naming? Did you comment your code as your instructors have in class?
- **Deployment:** Did you deploy your application to a public url using GitHub Pages?
- **Total:** Your instructors will give you a total score on your project between:

Score | Expectations ----- | ----- **0** | *Incomplete. 1 | Does not meet expectations.*  
**2** | *Meets expectations, good job!* **3** | *Exceeds expectations, you wonderful creature, you!*

This will serve as a helpful overall gauge of whether you met the project goals, but **the more important scores are the individual ones** above, which can help you identify where to focus your efforts for the next project!



# Requesting Help + Feedback

If/when you need help on your project...

- Create an **Issue** in your Github repository that describes **what** you aim to do and **what** went wrong. **What have you tried?**
- Come let an instructor know you have a problem!
- Instructional staff will only be assisting inside of the classroom
- Please use us for support!

## Part 4: Ruby 101

This week we are going to dive deep into the Ruby programming language!



- Ruby Language: <https://www.ruby-lang.org/en/>
- Ruby API Documentation: <http://ruby-doc.org/core-2.2.2/>
- OverAPI: <http://overapi.com/ruby/>

### pry

We will be using the **pry** environment for learning Ruby. Install it using:

```
gem install pry
```

## 4.0 Ruby Reading

Below is a list of Ruby resources that we recommend checking out if you want to prior to next week. Next week, we'll cover Ruby (not Ruby on Rails) and introduce building servers. If you'd like to get a bit of a head start, here are some of your instructor's favourite reads (order of least in depth to most in depth):

- **Try Ruby** - <http://tryruby.org> A brief, in-browser way to start trying Ruby!
- **Learn to Program Ruby by Chris Pine** - <https://pine.fm/LearnToProgram/> This is a quick, no frills introduction to Ruby.
- **Ruby Monk: Ruby Primer** - <https://rubymonk.com/learning/books/1-ruby-primer> An extensive set of Ruby tutorials.
- **Learn Ruby the Hard Way** - <http://learnrubythehardway.org/book/> An extensive and free Ruby book.

Finally, we highly recommend checking out Sandi Metz's **Practical Object-Oriented Ruby**. It is a short book (compared to other programming books) fantastic examples. We have two copies on our local library. Unfortunately, it is not free but it may be purchased as a digital or physical copy.

Finally: *Ruby on Rails != Ruby*. Rails is a framework written in Ruby. So no need to read Rails books (we'll cover Rails later in the course).

## 4.1 Introduction to Ruby

### Data Types, Control Flow, and Logic

#### Objectives

Understand the following in Ruby:

- Variables
  - Numbers
  - Strings
  - Booleans
  - Arrays
  - Hashes
- Method Basics (how to build a method)
- Conditional Logic

#### Resources

**Slide Presentation:** <https://presentations.generalassemb.ly/9a7806889f8585009ccd#/>

#### Optional Reads for the Week

- Read Chris Pine's *Learn to Program* in Ruby - <https://pine.fm/LearnToProgram/>
- Read RubyMonk's *Ruby Primer* - <https://rubymonk.com/learning/books/1-ruby-primer>

## 4.1 Front End Recap Quiz

1. List all traditional data types in Javascript.
2. Objects are what type of storage? They use :. Explain how this works.
3. What is a function called when it is owned by an Object?
4. Describe what CSS is used for.
5. Explain the difference between a class and ID in CSS.
6. What is an attribute in an HTML tag? Provide an example.
7. Using Vanilla JS, what is the event that is triggered when all content on a page has loaded?
8. What is jQuery and how is it useful?
9. Describe a scenario where using jQuery makes no sense.
10. How would you select a DOM element using both vanilla JS and jQuery if it were an article with an ID of 'win'.
11. How do you organize a project folder to separate scripts & styles?
12. Explain block scope. This can also be referred to as lexical scoping.
13. What is a boilerplate? Why would you use one?
14. When using a constructor to instantiate something, what is the data type that is created?
15. Describe Git in your own words.

## 4.1 Setting Up a Development Environment

Before we get started, we need to all be on the same page! That means we need to have the same version of Ruby as well as the same tools. Let's do that!

1. Install `rbenv`, a Ruby environment manager: `brew install rbenv`
2. Now, have `rbenv` install Ruby version 2.2.0: `rbenv install 2.2.0`
3. Choose the `global` version of Ruby (for your entire Mac) to version 2.2.0: `rbenv global 2.2.0`
4. Finally, install `pry` - the Ruby REPL console that we'll use: `gem install pry`
5. PS: `gems` are Ruby applications. You can install them using `gem install gem_name`.  
For a complete listing: <https://rubygems.org/>

## 4.1 Strings, Arrays, & Hashes

### String

```
'happy'.object_id

'happy'.gsub('ha', 'HAHAHA')
'happy'.split('')
'happy'.chars()

lichy = 'Lichard DeGray'
"My name is not #{lichy}" # String interpolation

number = 99
 "#{ number } bottle#{ 's' unless number==1 } of beer on the wall... #{ number } bottle
```

### Array

```
nums = Array(1..2) # auto-populate
letters = Array('a'..'z') # auto-populate
my_things = ['apartment', 'laptop', 'cat', 'wii u'] # manual declaration
```

Let's take a look at that again...

```
arrays!
pirates = ['Blackbeard', "Blue beard", "James Cook"]
pirates[0]
pirates[0] = 'Blackbeard is DEAD'
nums = Array(1..10)
letters = Array('a'..'z')
```

- Grab a random **sample** from an array...

```
my_things.sample
```

- Access a specific item at..

```
my_things.at(1)
```

- First & last...

```
my_things.first
my_things.last
```

- Adding items to an Array

```
arr = [1, 2, 3, 4]
arr.push(5) #=> [1, 2, 3, 4, 5]
arr << 6 #=> [1, 2, 3, 4, 5, 6]
```

- Removing items from an array

```
arr = [1, 2, 3, 4, 5, 6]
arr.pop #=> 6
arr #=> [1, 2, 3, 4, 5]

arr.shift #=> 1
arr #=> [2, 3, 4, 5]

arr.delete_at(2) #=> 4
arr #=> [2, 3, 5]
```

## Hash

```
lich = { :name => 'Lichard', :age => 3}
kat = { :name => 'Kathew', :age => 3}
om = { :name => 'Omily', :age => 3}
```

- Using the hash rocket syntax

```
sample_hash = {'one'=>1, 'two'=>2}
sample_hash_2 = { :one=>1, :two=>2}
```

- Using colons to create a hash will create the keys as symbols

```
sample_hash_3 = {one:1, two:2}
```

- How to access a value (**have** to use bracket notation)

```
sample_hash_4 = {'one'=>1, 'two'=>2, 'three'=>3, 'four'=>4, 'five'=>5}
sample_hash_4["one"] #=> Returns 1
sample_hash_4["five"] #=> Returns 5
```

- How to change the value of an element

```
sample_hash_4['one'] = 10
```

## 4.1 Detecting Ruby Types

```
#!/usr/bin/ruby

h = { :name => "Lichard", :age => 28 }

p true.class, false.class
p "Ruby".class
p 1.class
p 4.5.class
p 3_463_456_457.class
p :age.class
p [1, 2, 3].class
p h.class
```

## 4.1 Video

### Introduction to Ruby with pry

- <https://www.youtube.com/watch?v=ThbqQeYC0V8&feature=youtu.be>

## 4.1 Conditionals

- if/elsif/else/end
- unless
- Unless you're using a one-liner then you need to end your if statement with and 'end' statement.

```
x = 2

if x < 3
 puts 'less than 3'
end

* Writing the same thing as a one-liner

if x < 3 then puts 'less than 3' end
puts 'less than 3' if x < 3

* Using elsif

if x < 2
 puts "less than 2"
elsif x == 2
 puts "It's two!"
else
 puts 'greater than two!'
end
```

- Unless
- My own personal preference is to avoid the 'not' operator and state conditions in the positive. Using the 'unless' keyword is great for this.

```
x = true

puts "it's true!" if x != false

unless x == false
 puts "it's true!"
end
```

- Unless can also be used as a one-liner

```
puts "it's true!" unless x == false
```

### You cannot use `elsif` with `unless`, only `else`

```
unless x == false
 puts "it's true!"
else
 puts "it's false!"
end
```

```
unless x == false
 puts "it's true!"
elsif
 puts "it's false!"
end
```

```
#returns an error
```

## More Examples

```
num_of_pizzas= 7

def num_of_slices(pizza_count)
 pizza_count * 8
end

slices_per_person = 3
total_num_of_students =14

def totalslices
 num_of_slices * num_of_pizzas
end

if totalslices / 3 > 14
 puts we have enough slices!
else
 puts we dont have enough slices!
end
```

## 4.1 Loops

### While Loop

```
n = 1
while n < 11
 puts n
 n += 1
end
```

### Until Loop

```
n = 1
until n > 10
 puts n
 n += 1
end
```

```
sample_array = Array(1..5)
sample_array.each{|elem| puts elem}
```

```
sample_array.each do |elem|
 puts elem
 puts "The long way!!!"
end
```

```
each_example = [2,3,4].each{|elem| elem ** 2}
map_example = [2,3,4].map{|elem| elem ** 2}

p each_example
p map_example

each_example_2 = {two:2,three:3,four:4}.each{|key,value| elem ** 2}
map_example_2 = {two:2,three:3,four:4}.map{|key,value| elem ** 2}

p each_example_2
p map_example_2
```



## 4.1 Enumeration

### .each

- Used to perform task on each element... but I don't need a new array

```
names = ['Andy', 'Pandy', 'Dandy']

names.each do |specific_name|
 puts specific_name.upcase # name has the value of a specific name
end

names.each { |specific_name| puts specific_name.upcase }
```

{ is replacing the do

} is replacing the end

### .each

Returns the original array

```
same_as_original = names.each do |specific_name|
 puts specific_name.upcase # name has the value of a specific name
end

same_as_original = names.each { |specific_name| puts specific_name.upcase }
```

{ is replacing the do

} is replacing the end

### .map

Used to perform task on each element, and I want a new array of modified data

```

names = ['Andy', 'Pandy', 'Dandy']

modified_data = names.map do |specific_name|
 specific_name.upcase ## builds new array... of last values in the block
end

names = ['Andy', 'Pandy', 'Dandy']
initials = names.map { |name| name[0] }

```

## Examples

```

three_woodstocks = 3.times.map{ |num| "Woodstock #{num}" }
five_hun_woodstocks = 500.times.map{ |num| "Woodstock#{num}" }

['Hi', 'There'].map{ |word| "Woodstock #{word}" }

10.times.map{ |taco| "How many tacos! #{taco}" }

['Hi', 'There'].reverse.map{ |num| "Woodstock #{num}" }

funky_fresh = ['Hi', 'There'].reverse.map do |num|
 "Woodstock #{num}"
end.join(' - ').chars.reverse.join(':)')

```

## Getting Weird

```

names = ['Andy', 'Pandy', 'Dandy']

names.each do |name|
 puts name
end.each do |name|
 puts "#{name} is great!"
end.map do |name|
 "#{name} is awesome!"
end.push("you are awesomest :)").join(' and ').upcase << "!"

```

## 4.1 Methods

```
def yolo
 puts 'yolo swag'
end

now, let's see an example with an argument
def smarty_pants(name_of_person)
 puts name_of_person.to_s + ' is awesome'
end

variable and method names
should NOT start with upper-case
no reason to m use upper-case
ruby also uses snake_case...
notCamelCase :)

smarty_pants('Adriana')
```

## 4.1 Single Quotes vs Double Quotes

### String Interpolation

The usage of double quotes allows you to use **string interpolation**. This is essentially *templating for strings*. Example:

```
world = 'Saturn'
"Hello, #{world}"
```

This example **will not** work with single quotes.

### Escaping Characters

The usage of double quotes allows you to **escape** any/all characters in a string.

Example:

```
puts 'a\nb' # just print a\nb
puts "a\nb" # print a, then b at newline
```

Using single quotes will only allow you to escape **other** single quotes. Example:

```
puts 'this is james\'s favourite subject'
```

## 4.1 Ruby Rups!

### Round 1: Prime Time

- Write a method called `prime?` that takes a single parameter called `number` and returns `true` if the parameter is a prime number, or `false` otherwise.
  - Use the `Math.sqrt` ...
- 

### Round 2: Fardingworth Falls

- Let's generate some random town names for a Tycoon-style video-game. We can do this by combining the following generic name fragments:
- **Starts:** Bed, Brunn, Dun, Far, Glen, Tarn
- **Middles:** ding, fing, ly, ston
- **Ends:** borough, burg, ditch, hall, pool, ville, way, worth

### Step 1

- Write a method called `town_names` that randomly generates a number of town names by combining one Start, one Middle, and one End. Calling `town_names(5)` should give an array of 5 town names. If just `town_names` is called, generate 3 names.

### Step 2

- Modify the method so that calling `town_names(3, 'near_water')` will randomly add either "-on-sea" or " Falls" to each of the names. Optionally, think of another value that the second argument could have, and add appropriate random suffixes or prefixes when it is provided.

### Step 3

- Modify the method so that calling `town_names(3, 'short_name')` will always generate names without a Middle.



## 4.1 Ruby Practice - REPS (RUPS 2.0)!

### Learning Objectives... or rather... REPS!!!!

- ...REPS with creating methods
  - ...REPS with iteration
  - ...REPS with functions on numbers, strings, arrays
- 

## Round 1

Write a function `lengths` that accepts a single parameter as an argument, namely an array of strings. The function should return an array of numbers. Each number in the array should be the length of the corresponding string.

```
words = ["hello", "what", "is", "up", "dude"]
lengths(words) # => [5, 4, 2, 2, 4]
```

---

## Round 2

Write a Ruby function called `transmogrifier`. This function should accept three arguments, which you can assume will be numbers. Your function should return the "transmogrified" result

The transmogrified result of three numbers is the product (numbers multiplied together) of the first two numbers, raised to the power (exponentially) of the third number.

For example, the transmogrified result of 5, 3, and 2 is `(5 times 3) to the power of 2` is 225.

Use your function to find the following answers.

```
transmogrifier(5, 4, 3)
transmogrifier(13, 12, 5)
transmogrifier(42, 13, 7)
```

## Round 3

Write a function called `toonify` that takes two parameters, `accent` and `sentence`.

- If `accent` is the string `"daffy"`, return a modified version of `sentence` with all "s" replaced with "th".
- If the accent is `"elmer"`, replace all "r" with "w".
- Feel free to add your own accents as well!
- If the accent is not recognized, just return the sentence as-is.

```
toonify("daffy", "so you smell like sausage")
#=> "tho you thmell like thauthage"
```

---

## Round 4

Write a function `wordReverse` that accepts a single argument, a string. The method should return a string with the order of the words reversed. Don't worry about punctuation.

```
wordReverse("Now I know what a TV dinner feels like")
=> "like feels dinner TV a what know I Now"
```

---

## Round 5

Write a function `letterReverse` that accepts a single argument, a string. The function should maintain the order of words in the string but reverse the letters in each word. Don't worry about punctuation. This will be very similar to round 4 except you won't need to split them with a space.

```
letterReverse("Now I know what a TV dinner feels like")
=> "woN I wonk tahn a VT rennid sleef ekil"
letterReverse("Put Hans back on the line")
=> "tuP snaH kcab no eht enil"
```

---

## Round 6

Write a function `longest` that accepts a single argument, an array of strings. The method should return the longest word in the array. In case of a tie, the method should return either.

```
longest(["oh", "good", "grief"]) # => "grief"
longest(["Nothing", "takes", "the", "taste", "out", "of", "peanut", "butter", "quite",
=> "unrequited"
```

## 4.2 Everything is an Object

### Homework Recap

- Recap last night's homework (the Ruby Rups!)

## Objects

### Objectives

- Describe what an Object is and how it differs from a JS Objects
- Define what object properties and methods are
- Write a getter to retrieve a property's value
- Write a setter to set a property's value
- Understand that everything is an object – including abstract things, basic data types, and objects we make up ourselves
- Understand that an object's properties are only accessible if there's a getter method – an object can have information inside we aren't able to access
- Demonstrate & explain instantiation

### Slide Deck

- <https://presentations.generalassembly.ly/73e032a06421abf77789#/>

## Methods in Depth

### Objectives

- Write a method that takes no parameters
- Write a method that takes multiple necessary parameters
- Write a method that takes optional parameters

## Blueprinting Objects with Classes and Inheritance

### Objectives

- Understand how classes inherit from other classes
- Describe how inheritance works with Object-oriented programming
- Describe what a base class is

- Describe what a child class is

### Interactive Class example

- <http://ga-chicago.github.io/ruby-class/>

## 4.2 Rups Recap

Below are various examples of how to solve the Rups homework from last night.

```
#####
Question 1
#Write a function lengths that accepts a single parameter as an argument, namely an array
#####

#example 1
def lengths(array)
 p array.map{|word| word.length}
end

lengths(words)

#example 2

def stringLengths (arrayOfStrings)
 i = 0
 secondArray = []
 while i < arrayOfStrings.length
 secondArray.push(arrayOfStrings[i].length)
 i += 1
 end
 return secondArray
end

p words = ["hello", "what", "is", "up", "dude"]
p stringLengths(words)

example 3
words = ["hello", "what", "is", "up", "dude"]
def lengths(arrayInput)
 wordlength = Array.new
 arrayInput.each{|word| wordlength<<word.length}
 p wordlength
end

#####

Question 2
#Write a Ruby function called transmogrifier This function should accept three arguments
The transmogrified result of three numbers is the product (numbers multiplied together)
For example, the transmogrified result of 5, 3, and 2 is (5 times 3) to the power of 2
#####
```

```

#example 1
def transmogrifier(num1, num2, num3)
 p (num1 * num2) ** num3
end

transmogrifier(2,5,2)

#example 2
def transmogrifier a, b, c
 p (a * b) ** c
end

#####
Question 3
Write a function called toonify that takes two parameters, accent and sentence.
#
If accent is the string "daffy", return a modified version of sentence with all "s"
If the accent is "elmer", replace all "r" with "w".
Feel free to add your own accents as well!
If the accent is not recognized, just return the sentence as-is.
#####

#example 1

def toonify(accent, sentence)
 accent = accent.downcase
 if accent == "daffy"
 p sentence.gsub("s", "th")
 elsif accent == "elmer"
 p sentence.gsub("r", "w")
 elsif accent == "liljohn"
 p sentence.gsub("./", "WHAT ")
 else
 p sentence
 end
end

toonify('daffy', 'so you smell like sausage')
toonify('elmer', "I'm gonna catch that rabbit!")
toonify('lilJohn', "Nice day out today, don't you think?")

#example 2
def toonify (accent,sentence)
 if accent.downcase == 'daffy'
 return p sentence.gsub 's', 'th'
 elsif accent.downcase == 'elmer'
 return p sentence.gsub 'r', 'w'
 else
 return sentence
 end
end

```

```

toonify("daffy", "so you smell like sausage")

Question 4
Write a function wordReverse that accepts a single argument, a string. The method s
#####

#example 1
def word_reverse(string)
 p string.split.reverse.join(' ')
end
word_reverse("Now I know what a TV dinner feels like")

#example 2
def word_reverse(string)
 words = string.split(' ')
 return words.reverse
end

p word_reverse("Now I know what a TV dinner feels like")

Question 5
Write a function letterReverse that accepts a single argument, a string. The functio
#####

#example 1
def letter_reverse(string)
 p string.split.map{|word| word.reverse}.join(" ")
end
letter_reverse("Put Hans back on the line")

#example 2
def sentenceReverse (string)
 someArray = string.split(' ')
 sentenceResult = ''
 i = 0
 while i < someArray.length
 sentenceResult = sentenceResult + someArray[i].reverse + ' '
 i += 1
 end
 return sentenceResult
end

p sentenceReverse("Now I know what a TV dinner feels like")

Question 6
#Write a function longest that accepts a single argument, an array of strings. The me
#####

#example 1
def longest(arr)
 longest = ""

```

```
arr.each do |i|
 if i.length > longest.length
 longest = i
 elsif i.length== longest.length
 longest = longest +" "+ i
 end
end
p longest
end

longest(["oh", "good", "grief", "fives"])

#example 2

def longest ls
 p ls.max{|i, j| i.length <=> j.length }
end

#example 3
def longest (arr)
 return arr.max{|a,b| a.length <=> b.length}
end

#example 4
def longest(words)
 return words[words.map{|word| word.length}.sort[0]]
end
p longest(["oh", "good", "grief"])

#example 5
def longest(arrStr)
 output = ""
 arrStr.map{|word| output = word if (word.length == arrStr.map{|word| word.length}.max)}
 return output
end
```

## 4.2 Enumeration, Again

Below are the examples used in class:

```
who_was_sleepy = ['anna', 'nick', 'ruth', 'lidia', 'everyone']

who_was_sleepy.each do |person|
 puts person
end

ninja_turtles = ['donatello', 'leonardo', 'michaelangelo', 'raphael']

+ ' says \'cowabunga dude\''
ninja_turtles.each do |turtle|
 puts turtle + ' says \'cowabunga dude\'!'
end

#introducing hash rockets
and introducing symbols!
good_news = {
 :happiness => 'EDM Festivals',
 :favourite_thing => 'ginger tea',
 :something_nice => 'kittens are adorable'
}

good_news.each do |item|
 puts item
end

#turn a symbol into a string
:happiness.to_s
```

## 4.2 The Movie Object

Below is the Movie object we built today in class prior to learning how to use Classes.

```
require some libraries
require 'httparty'
require 'json'

instantiate a new object called 'fetcher'
fetcher = Object.new

http://www.omdbapi.com/?t=Interstellar&y=&plot=short&r=json
def fetcher.get_favourite_movie
 puts HTTParty.get('http://www.omdbapi.com/?t=Interstellar&y=&plot=short&r=json')

 return HTTParty.get('http://www.omdbapi.com/?t=Interstellar&y=&plot=short&r=json').
end

fetcher.get_favourite_movie

def fetcher.convert_hash_to_json(hash)
 # convert a hash to json
 # return the json version
 return hash.to_json
end

test_hash = {
 :something => 'is awesome',
 :everything => 'is great',
 :enjoy => 'every day'
}

my_json = fetcher.convert_hash_to_json(test_hash)
puts my_json

def fetcher.set_url(url_to_api)
 # this.url = url_to_api
 @url = url_to_api
 return @url
end

def fetcher.get_url
 return @url
end

def fetcher.get_data
 return HTTParty.get(@url)
end

fetcher.set_url('http://www.omdbapi.com/?t=Die+Hard&y=&plot=short&r=json')
fetcher.get_data
```



## 4.2 Methods (and the \*SPLAT! argument)

Below are the examples built in class:

```
taking a look at SPLAT

def all_the_things(*things)

 things.each do |thing|
 puts thing
 end

end

all_the_things('random', 90, 'hooloovoo', 'ocean at the end of the lane', :stuff)

def name_builder(first_name, last_name, *misc)
 puts '----'
 puts misc

 misc.each do |item|
 puts item
 end
 return "Welcome to the world, " + first_name + ' ' + last_name
end

name_of_child = name_builder('lichard', 'slacken', [42, 'lol', :lol], 'meow', 'haters')
puts name_of_child

cereals = ['count chocula', 'booberry', 'frankberry', 'MONSTERRRRRRR AHCCCC', 'fruity']

def list_breakfast(food, *appetizers)

 puts food
 puts appetizers
 return appetizers

end

yum = list_breakfast(cereals, 'eggs', 'bacon', 'bloody mary')
puts yum

define a singleton method on an object

steve_ballmer = Object.new

steve_ballmer.define_singleton_method(:get_excited) do
```

```
puts 'DEVELOPERS DEVELOPERS DEVELOPERS WOOO YEAAHHH'
end

steve_ballmer.get_excited

puts '---- objectsssss with methods ----'

server = Object.new

def server.output_data
 puts '1010011010101001011111101010110101010101'
end

def server.say_hello(name)
 puts 'Hello, ' + name
end

server.output_data
server.say_hello('jimbo jones 1 @ github')
```

## 4.2 Classes in Ruby

### Scope

1. Defining a class
2. Creating an instance of a class
3. Define what an instance variable is
4. Define what a class variable is
5. Implementing a setter (mutator) and getter

**Class:** A class is instructions (a blueprint) for building an object. As a class, I can inherit abilities from another class. A class is just an Object.

To create a class, we need to build a skeleton out:

```
class MyClassName
 #stuff goes here
end
```

To create a new instance of `MyClassName`, I would:

```
my_awesome_class = MyClassName.new()
```

Each class may have a constructor. That constructor's method name is a *reserved* word in Ruby called `initialize`. This constructor can expect arguments passed into the `.new()` method.

```
class MyClassName

 def initialize(message)
 puts message
 end

end

my_awesome_class = MyClassName.new("Hello, world!")
```

If we wanted to create an **instance variable** for message, we could do so. An instance variable is just a variable assigned to whatever *new* object created based on each new class. In the following example, we will create an instance variable for message.

Instance variables are preceded with `@`.

```
class MyClassName

 def initialize(message)
 @message = message #instance variable
 end

end

my_awesome_class = MyClassName.new("Hello, world!")
my_other_class = MyClassName.new("oh, hello friends")
```

`my_awesome_class` and `my_other_class` have two *different* values for their `@message` instance variables.

We can also create **class variables**, variables that apply to *all* instances of a class. These are not used anywhere near as much as instance variables but may be declared using `@@variable_name`.

```
class MyClassName

 def initialize(message)
 @@all_classes_have_me = message #class variable
 end

end

my_awesome_class = MyClassName.new("Hello, world!")
my_other_class = MyClassName.new("oh, hello friends")
```

Finally, to get/set data in our instance variables, we need to create **getter** and **setter** methods. Mutator is another word for setter.

```
class MyClassName

 def initialize(message)
 @message = message #instance variable
 end

 #getter - calling myClass.message returns @message
 def message
 @message
 end

 #setter - calling myClass.message = "stuff" changes @message
 def message=(new_message)
 @message = new_message
 end

end

my_awesome_class = MyClassName.new("Hello, world!")
my_awesome_class.message # => "Hello, world!"
my_awesome_class.message = "new message"
my_awesome_class.message # => "new message"
```

## 4.2 Stephen's Class Notes

### Accessors

- **Accessors**: create getters and setters for instance variables
- **attr\_reader** —> creates getter
- **attr\_writer** —> creates setter
- **attr\_accessor** —> creates both getter & setter

### Variables

- **local** —> lives within the space it is declared in
- **\$global** —> can be accessed from anywhere
- **@instance** —> can be accessed by all methods in INSTANCE of that class
- **@@class** —> can be accessed by class & all instances of that class

### Methods

- Class methods (can only be called on class)

```
def self.method_name
end
```

- Instance methods (can only be called on instance of class)

```
def method_name
end
def initialize(arguments)
end
```

(populates new instance with whatever info you set here)

## 4.2 Inheritance via Classes

```
#inheritance through space ships!

class SaturnV
 def initialize
 @name = 'Saturn V'
 @fuel = 'liquid hydrogen'
 @mission = 'go to the moon'
 end

 def take_off
 return "We're taking off! YAY!"
 end

 def to_s
 return @name + " has a mission to " + @mission
 end
end

class SLS < SaturnV
 def initialize
 @name = 'SLS'
 @fuel = 'liquid hydrogen'
 @mission = 'to make it to Mars and Europa'
 end
end

class Enterprise < SLS
 def initialize
 @name = 'Starship enterprise'
 @mission = 'to boldly go where no man has gone before'
 @fuel = 'antimatter'
 end
end
```

## 4.2 Homework - Class Reps

Tonight we're going to have you repeatedly build classes over and over. The primary goal is for you to become familiar with building classes. We're going to take what you have learned to build a few amazing utilities that you can use once we build servers tomorrow!

### 1. Movie Class

- Build a class called `Movie`.
- You are **not** allowed to use `attr_accessor` with this class.
- This means that you need to build manual getters/setters!
- This class has the following **attributes** as `@instance_variables`:
  - `@movie_title`
  - `@movie_description`
  - `@omdb_url`
- This class has the following **abilities**:
  - `set_omdb_url` with a single argument of `url` that will set `@omdb_url`
  - `get_omdb_url` that returns `@omdb_url`
  - `fetch_movie` that uses `HTTParty.get` with the `@url` to receive data. Next, it will update `@movie_title` and `@movie_description` with
  - `to_s` that returns the `@movie_title & @movie_description` as a string.
- Test these methods to verify they work in your code.
- Note: Don't forget to `require 'HTTParty'` in your code!
- Save this in a file called `movie_class.rb`

### 2. Dictionary Class

- Build a class called `Dictionary`
- You are **not** allowed to use `attr_accessor` with this class.
- This means that you need to build manual getters/setters!
- This class has the following **attributes** as `@instance_variables`:
  - `@internal_hash`
- This class has the following **abilities**:
  - `get_dictionary` that returns `@internal_hash`
  - `add` that accepts two arguments: `:key` and `value`. You will then add these to your `@internal_hash`

- `to_s` that returns the `@internal_hash` as a string (consider using `.each` here)
- `to_json` that returns the `@internal_hash` as a JSON object.
- Note: Don't forget to `require 'json'` in your code!
- Test these methods to verify they work in your code.
  - Add a `404` key with a value of `womp womp - the page doesn't exist`
  - Add a `403` key with a value of `DENIED ACCESS`
  - Add a `500` key with a value of `SERVER ERROR OH NOES`
  - Output the Dictionary as JSON using the method you built.
- Save this in a file called `dictionary_class.rb`

### 3. RUPS Class

- Build a class called `Rups`
- You are allowed to use `attr_accessor` in this class!
- Turn all of your homeworks' methods last night into individual methods that your `Rups` class owns.
- Add instance variables as needed here (you may or may not need them depending on how you build your class)
- Test and verify all of your methods work.
- Save this in a file called `rups_class.rb`

To get you started...

```
class Rups

 def lengths(argument)
 # some code...
 end

end
```

## 4.3 - Introduction to Sinatra

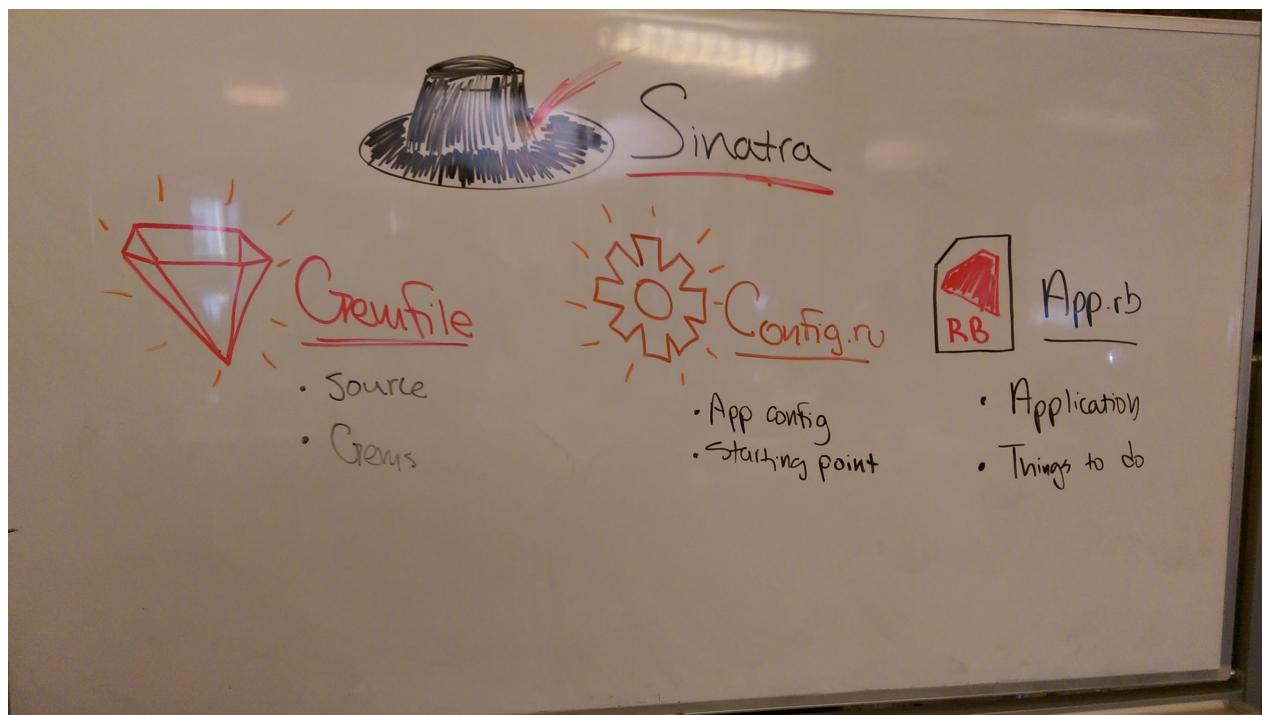
### Sinatra 101

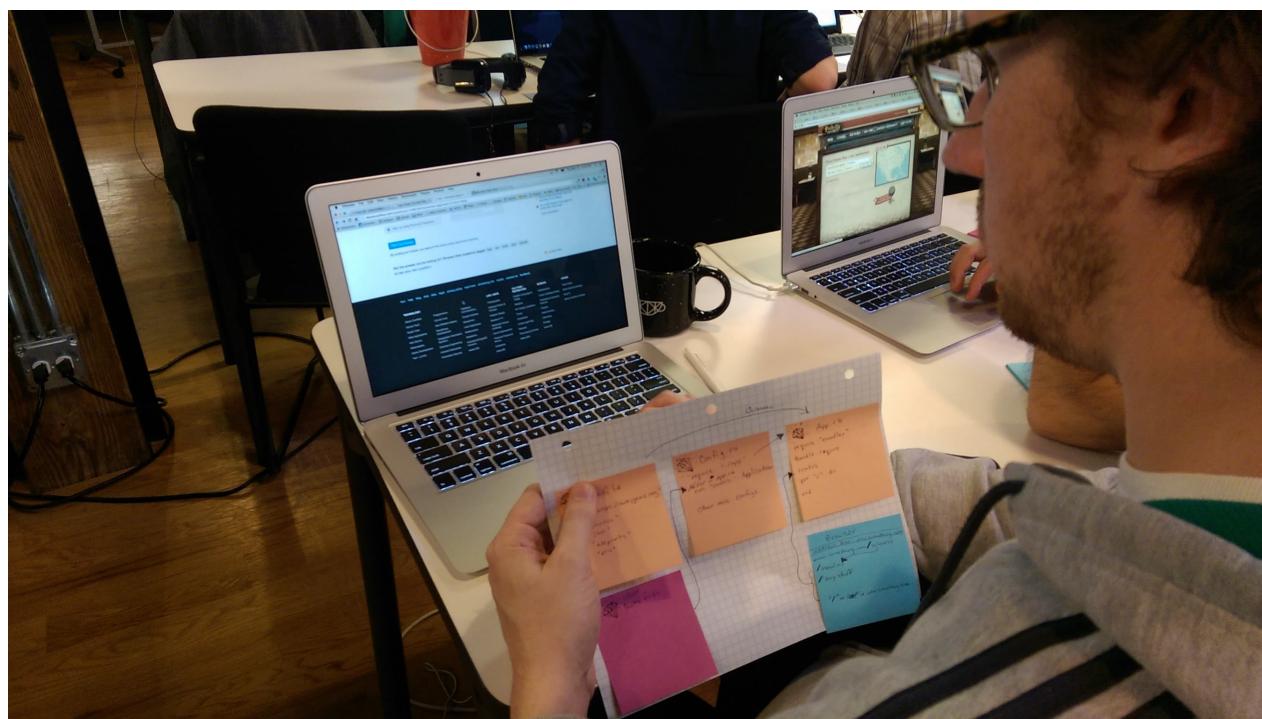
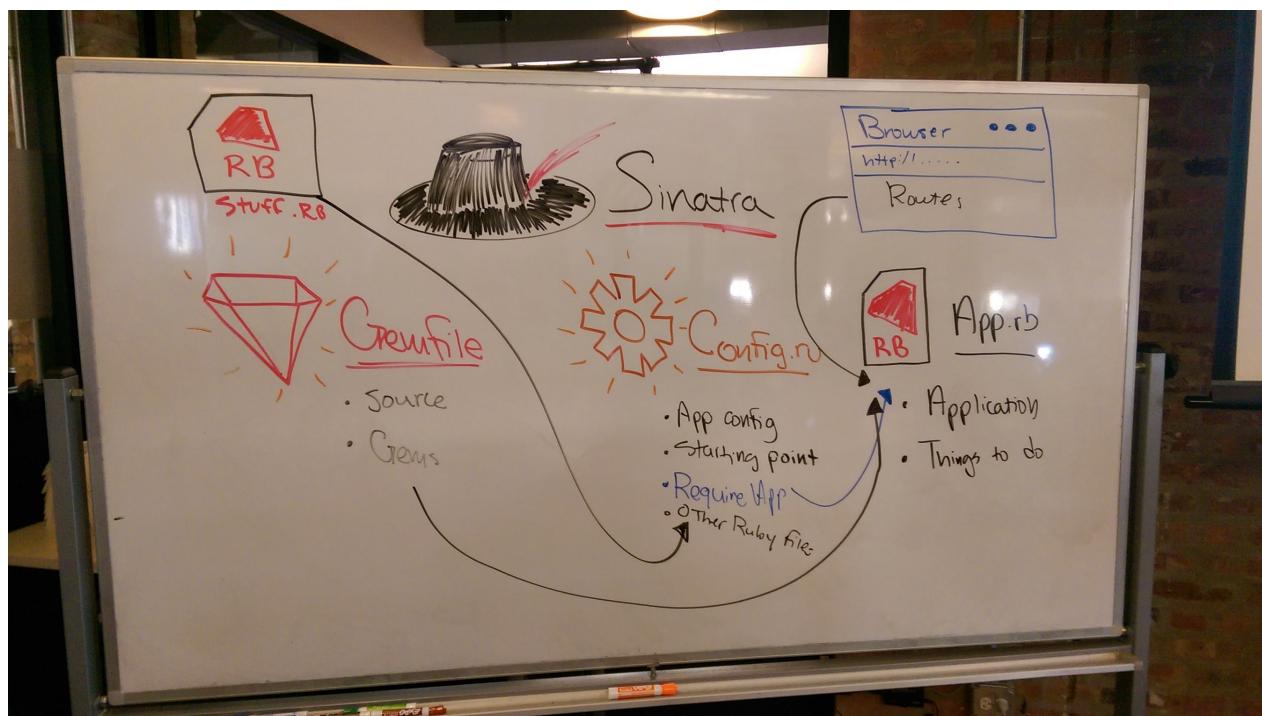
#### Objectives

- Build your first web server
- Serve JSON to anyone who requests it
- Use Sinatra to do the above

#### Notes

- When you add a gem to your `Gemfile`, you need to `bundle`
- `bundle exec rackup` allows you to run your server locally





## Resources

- Slide Deck
- Driving with Sinatra
- Sinatra in 2 minutes: <https://www.youtube.com/watch?v=eEwXaedKnW4&feature=youtu.be>

## 4.3 Ruby: Our First Web server with Sinatra

### Agenda

- Quiz
- Introducing Sinatra
- Your first web server
  - Part 1: Gemfile
  - Part 2: Config.ru
  - Part 3: App.rb
- Let's run our server
- Viewing our resource

### QUIZ

1. What is a popular way to iterate through collections such as Arrays and Hashes?
2. What is used to blueprint Objects in Ruby?
3. Name a use case for using \*args with a method!
4. Classes are objects and therefore have \_ and \_\_\_\_.
5. To modify these, we create \_ and \_ methods (hint: mutator).

### We need a few Gems...

- We'll need Sinatra
- Let's install it! `gem install sinatra`
- Finally, we want Bundler - `gem install bundler`
- This gem allows us to bundle all the things together to make an app

### Introducing Sinatra

- <http://www.sinatrarb.com/>
- Lightweight, designed to get to the point
- Highly modular and built to scale

- The easiest way to write a server in Ruby

# Your First Webserver

**Let's get started by building a web server in Ruby**

## 1. Gemfile

First, we need somewhere to store all of the Gems that our application needs...

- Gemfiles are where we store information about our application.
- They allow us to specify what gems we want our application to use.
- ...and where to get them from.

*Gemfile*

```
source 'https://rubygems.org'

gem 'sinatra'
gem 'json'
```

## Oh wait... did you hear that?

*The Gemfile had new Gems added to it!*

- `bundle`
- Yes, `bundle`
- From terminal in the folder where your app lives...
- Run the `bundle` command
- Bundler will then grab all the gems and bundle them together for your app

## 2. Config.ru

- We now need a file to tell our application how it should be configured.
- This tells us what we need to do and what settings our app should use.
- Since we're using Sinatra, this will be pretty simple

```
require './app'
run Sinatra::Application
```

## 2. Config.ru

What was going on in that file?

- We required the `app.rb` file by using Ruby's **Require** statement
- This forces a file to be loaded once into our application, making all of its methods available
- We then `run Sinatra::Application` - or tell Sinatra to start.
- That's it!

## 3. App.rb

*Our final app.rb will look like this.. but we're going to build it!*

```
require 'bundler'
Bundler.require()

get '/' do
 { :name => 'test' }.to_json
end
```

## 3. App.rb

- In our `app.rb` file, we now need to require Bundler.
- It is what bundles our gems.
- We need Gems in our app - you'll see why when we want to send JSON results.

*app.rb*

```
require 'bundler'
Bundler.require()
```

## 3. App.rb

- Now, we need to let users access resources on our server.
- But how will we tell them where to go?
- We provide **Routers**. These are similar to *Controllers*.
- These route the user to <http://somedomain.com/route/>
- We'll define a root route first to access <http://localhost/>
- If this were on a live server, it could be <http://somedomain.com/>

*app.rb*

```
require 'bundler'
Bundler.require()

get '/' do
 # some code goes here
end
```

## App.rb

- Now that we can have a user access a resource via a route...
- Let's expose a resource to them!
- We'll use the JSON gem here to return a Hash back to a user as JSON

*app.rb*

```
require 'bundler'
Bundler.require()

get '/' do
 {:message => 'hello, world!'}).to_json
end
```

## Save EVERYTHING

- Add
- Commit
- Pull
- Push

## Let's run our server

- In terminal, let's run our app.

- Did you add any new gems? If so, `bundle!`
- If not, you heard nothing `shifty_eyes`
- Now, let's start our server

```
bundle exec rackup
```

- This tells the **Rackup** middleware to run our server

## Viewing our resource

- Sinatra listens for requests on port **9292**
- Let's browse to our resource!
- <http://localhost:9292/>
- ...what do you see!

## Conclusion

- You just build a webserver!
- Is this awesome?
- Yes?
- HECK YEAH

## 4.3 Driving with Sinatra

Pair Programming by building a Sinatra Server

### Instructions

- Create an app.rb, config.ru, and Gemfile
- For each file, switch turns with your partner
- One of you will navigate: 'explain what to do'
- The other will drive: 'write the code'
- In your app.rb, create a '/' route
- Return a Hash.to\_json of data about your team
- Create two additional routes
- '/first\_person' and '/second\_person'
- These routes should return a hash to json about each person
- Upon completion, test and verify all 3 routes work!
- Git add, commit, push!

### Example

- [http://github.com/code-for-coffee/driving\\_with\\_sinatra/](http://github.com/code-for-coffee/driving_with_sinatra/)

## 4.3: Sinatra (on your own)

- It is time to create another Sinatra application. This time, you're going to create a new **git repository** on Github.
- Clone it down to `~/dragons`.
- Create a basic boilerplate for Sinatra.
- This should include an `app.rb`, `config.ru`, and `Gemfile`.
- Make sure you set your `.gitignore` to Ruby.
- All you need is a base `get '/'` route.
- You do not need to **implement anything in your route(s)**
- However, you need to test your code to verify the server works (remember that Sinatra requires you to `return` strings).
- Add, commit, push to this repository.

## 4.3 ERB

ERB stands for **embedded Ruby**.

You may call Ruby methods and variables using the `<%= ruby_here %>` syntax. To create an ERB view, create a file in the `/views/` directory. It must be called `filename.erb`. This is standard HTML with Ruby built in.

### Rendering a view

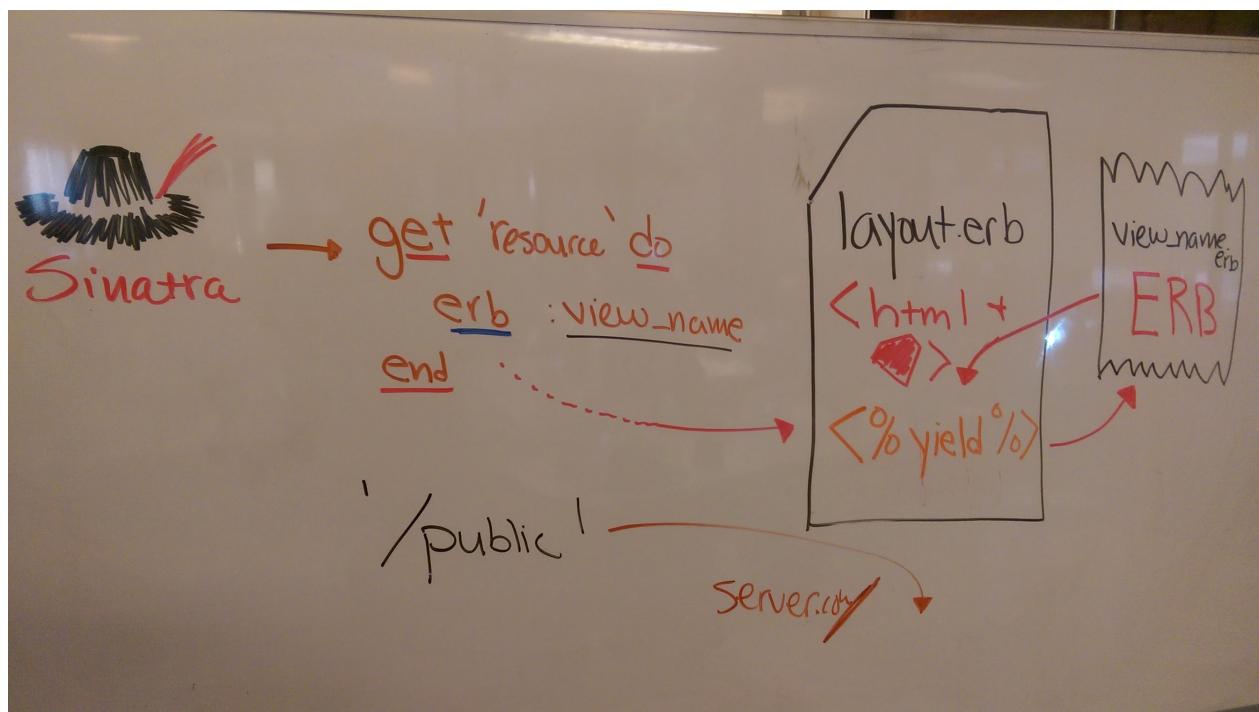
To render a view (instead of JSON), call:

```
get '/' do
 erb :filename # this would point to /views/filename.erb
end
```

Notice the use of a **symbol** to call the filename.

### Yield + Master Layout

You can include a master layout file that will apply to **every** page you access. This must be named `layout.erb`. Inside, you will need to create a *yield* statement. This statement yields to the view you request and places that specific HTML/Ruby where yield is located.



## Example

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>Box Model Intro</title>
 <link rel='stylesheet' href='style.css'>
 </head>
 <body>

 <%= yield %>

 </body>
</html>
```

## 4.3 Homework: MOAR SERVERS!@!@!@1212

### MOAR SERVERS

1. It is time to build the cutest server ever - the `small_animals` server! Place this in a new folder in `05_ruby101/your_name/` called `small_animals`
2. Create a brand new, small Sinatra app. Check out the *Simplest Sinatra app, ever* above for guidance.
3. Write a **simple** Sinatra app with a `Gemfile`, `config.ru`, and `app.rb`
4. Inside of your `app.rb`, you need to create a few routes...
  - o `'/'` - This should return a Hash turned into JSON with the following keys: `:name` and `:message`
  - o *Five* different routes mapped to the names of small animals - ie `/kitten` and `/puppy`
  - o These five different routes should return a hash turned JSON with the following keys: `:name`, `:cuteness`, `:habitat`, `:picture_url`, and `:description`
5. Test and verify these routes render JSON in the browser.
6. **BONUS:** Add *ERB* views and then create a new `/views/json_test.erb`. Map it to a `router` and call each API call using `$.getJSON` or `$.ajax` to render the content using `console.log` or whatever else you choose.
7. **BONUS:** You can use each animal route's `:picture_url` to render images as a double bonus!

## 4.3 Bonus Homework

### Servers! Servers, everywhere!

1. Create a new directory in `05_ruby101/your_name/` called `sinatra_reps`
2. Create a brand new, small Sinatra app.
3. Write a **simple** Sinatra app with a `Gemfile`, `config.ru`, and `app.rb`
4. Inside of your `app.rb`, you need to create a few routes...
  - o `'/'` - This should return a Hash turned into JSON with the following keys: `:name` and `:message`
  - o `'/about'` - this should return a Hash turned into JSON with the following keys: `:about`, `:age`, `:favourites`
  - o `'/contact'` - this should return a Hash turned into a JSON with the following keys: `:name`, `:email`, and `:thanks`
5. Test and verify these routes render JSON in the browser.
6. **BONUS** Add *ERB* views and then create a new `/views/json_test.erb`. Map it to a `router` and call each API call using `$.getJSON` or `$.ajax` to render the content using `console.log` or whatever else you choose.

## 4.4 Second Pass Friday

### Topics

- JSON
- Ruby Q&A / Practice Session
- Sinatra Second Pass
- Deploying Sinatra apps to Heroku

## 4.4 JSON

JSON stands for **Javascript Object Notation**. It is a string version of a Javascript Object. Hashes in Ruby can easily be converted to JSON. In turn, this JSON can be used by an AJAX call on the client (Javascript) side.

### JSON Example

```
{
 "sleep": "haz sleep last night yay",
 "awake": true,
 "times": "can be had in jamaica",
 "ice_cream": "dreamsicle",
 "people": "this class"
}
```

### Lecture Video

- <https://youtu.be/rBWK3kschJg>

## 4.4 Ruby Q&A

### Recap Video

- <https://www.youtube.com/watch?v=GHsJbm3m1fQ>

### Classes

```
Classes - getting and setting
class User
 attr_accessor :name, :email, :password
initialize optional
 def initialize(name, email, password, array)
 @name = name
 @email = email
 @password = password
 @array = array
 end
WAT
 def change_array
 @a = @array
 @b = @array
 p @a.each{|word| p word.upcase!}
 @a += ["anything"]
 p @b
 p @a
 p @array
 end
end
adriana= User.new("adriana", "adri@gmail.com", "password123", ["hello"])
p adriana.name
p adriana.name= "adrian"
adriana.change_array
```

## 4.4 - Sinatra, Again

We built a Sinatra API that can be viewed online!

### Source

- [https://github.com/code-for-coffee/happy\\_things](https://github.com/code-for-coffee/happy_things)

### Video

- <https://www.youtube.com/watch?v=l6bl7O3ELck>

### JSONReader Class

```
class JSONReader

 def initialize(filename)
 @json = String.new
 File.foreach(filename) do |line|
 @json = @json + line
 end
 #binding.pry
 end

 def to_hash
 return JSON.parse(@json)
 end

end
```

## 4.4 Deploying to Heroku

1. Log in to <http://heroku.com>
2. In the top-right corner, select the +
3. Create a new app. Give it a name
4. Select the tie-in for **Github**
5. Scroll down and select the repository you wish to use.
6. Scroll down and select **Automatic Deployment**
7. Scroll down and select **Manual Deployment** to push live
8. From now on, every time you push to your Git repository, Heroku will automatically push those changes live!

## Video

- [Heroku Video on Youtube](#)

## 4.4 Weekend Homework

### Read, Read, Read

Research and read Ruby books. [You can view our recommendations here](#). We highly recommend either **Learn Ruby the Hard Way** or **Ruby Monk** if you do not purchase Sandi Metz's *Practical Object Oriented Ruby*. You know what content is giving you problems... so go out and research them!

### Project 1... Live!

- Create a brand new Github repository for your first project (the game).
- Create a Sinatra server.
- Host your game using **ERB View(s)** and placing your scripts/styles in the **/public/** folder.
- Deploy it to Heroku!
- To submit for completion, in the `05_ruby101/your_name` folder create a file called `weekend-hw.md`. Inside of it, link to your repository and live website URL.
- If you run into Heroku issues please at minimum link to your repository.

## Part 5: Fullstack Sinatra

# 5.1 Behind the Models: Databases & SQL

## Introduction to Databases

Wi-Fi: Space  
Word: w0rk5pac3

Daily Agenda

- Morning Exercise
- Unit 1 - Intro to Databases
- Unit 2 - ERD Diagrams
- Up - Lunch
- Unit 3 - SQL
- Unit 4 - Migrations
- 4:00 PM - After work

#SQLDay

User-name	user_password	email
Drosef	TR3	drosef@yandex.ru
Omily	SD2	omily@outlook.com
Tom	(top)	tom@meowcity.com

## Objectives

- Understand why we use databases?
- Understand what Tables, Rows, & Columns represent.
- Know what **primary keys** and **foreign keys** are.
- Know the difference between database relationship types.
- Practice drawing Entity Relationship Diagrams (ERD).

## Resources

- [Slide Deck](#) | [Markdown Version](#)

## SQL and Migrations

## Objectives

- Create databases.
- Connect to databases.
- Create tables.
- Add / query / delete rows from tables.

## Resources

- [Postgres.app](#)
- [Slide Deck | Markdown Version](#)

## 5.1 Introduction to Databases

### Agenda

- Quiz
- Why databases?
- But... why databases?
- Understand what Tables, Rows, & Columns represent
- What is a primary key?
- Understand how tables relate to each other using foreign keys
- Know the difference between relationship types
- Draw Entity Relationship Diagrams (ERD)
- Bringing it together
- ERD Lab

### QUIZ

1. What Sinatra method lets you render Views?
2. What is a Gemfile good for?
3. If I didn't use `Bundler.require`, what would I have to do to load libraries (gems)?
4. Sinatra has two built in ways to handle `404` errors - they are \_\_ and \_\_.
5. Routers/Controllers dictate that to do when the user requests a \_\_.

### Why databases?

- Why indeed.
- We tried to store data in files.
- However, text files are unique to an operating system.
- If Lichard in development made changes to the text file...
- ...Kathew in accounting would need to manually update his.
- Entire jobs were dedicated to keeping files updated!
- Network sharing helped a little bit here but there was still one problem...
- If someone edited the file, someone else's changes could be lost!

### But... why databases?

1. They are structured!

2. Data is stored into neat, organized containers that make logical sense.
3. Data can be accessed via a dedicated *query language*
4. They have their own protocol, too.
5. Databases can even be standardized and normalized.

## Databases are Tables of Objects

- We use databases to store our objects as tables.
- These objects have attributes; in very special cases there can be abilities as well.
- Databases are great for keeping things neat and organized.

## Tables, Rows, & Columns

- **Tables** represent a list of objects as data.
- **Rows** represent *one unique item\** from this list.
- **Columns** represent **the attributes** of the object being listed.

Now, I'm going to demonstrate on the whiteboard what tables, rows, and columns represent.

## What is a primary key?

- A **primary key** is a unique identifier.
- Each table will contain one column that is unique.
- This means that no two values in this column can ever be the same.
- These are usually the object's **id**.
- They typically *auto increment*.
- So `Lichard.id` is 1, `Kathew.id` is 2, and so forth.

## Foreign keys connect databases

- We can also specify **foreign keys** on columns.
- These are used to link two tables together.
- Humans sometimes have cats, right?
- So the human's `cat_id` would have a foreign key to the cat's `id`!

I'm going to draw this out on the board to demonstrate it.

## Let's talk relationships

- No, not your Facebook status.
- We've talked about so far about linking things together.
- Let's think about ownership or... sharing.
- People can have one thing... such as a home, or a car.
- People can also have many things... such as pets, friends, or books.
- Things can be owned by people as well, right?
- Let's remove the word 'owned' and now call think of it as a relationship.

## Sound familiar?

- Foreign keys are used to connect the dots here.
- They link different objects together by relationship.
- In SQL, there are a few proper ways to show relationships.
- Let's check them out

## Relationship Statuses

*"Oh we just split up... he preferred JSON over me..."*

- One to One
- One to Many
- Many to Many

**Can you think of some examples for each type?**

## ERD - Entity Reference Diagram

- Let's think of our tables of objects as **entities**
- When planning out a new website or system, we need to think about it.
- Who starts building a skyscraper without an architectural layout?
- We need to **plan** out our databases.
- We use **ERDs** to do this.

**Problem:** Before we can make diagrams, we need to know what type of data each column is.

## Data Types

We'll focus on a few basic data types. They are:

- CHAR(number of characters) - `CHAR(5)`

- DATETIME is a UTC timestamp. - DATETIME
- DATE is a UTC date - DATE
- TEXT is a blob of text - TEXT
- BOOLEAN is **true** or **false** - BOOLEAN
- INTEGER .. you know what this is - INTEGER
- DECIMAL is basically a float - DECIMAL

A handout will be provided that covers all accepted data types for additional study.

## Bringing it Together

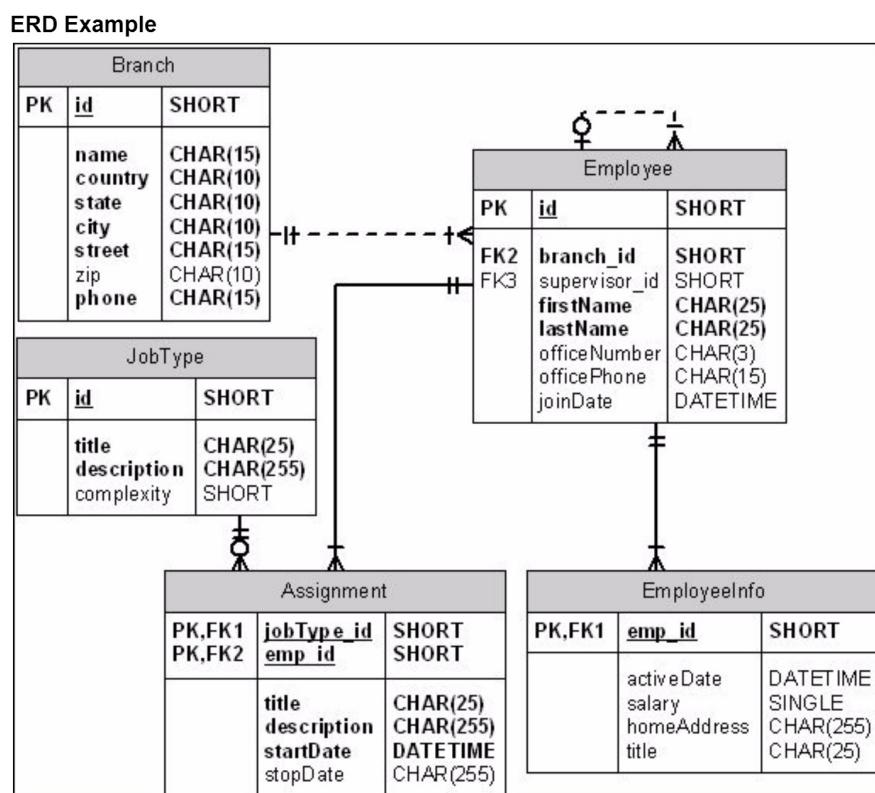
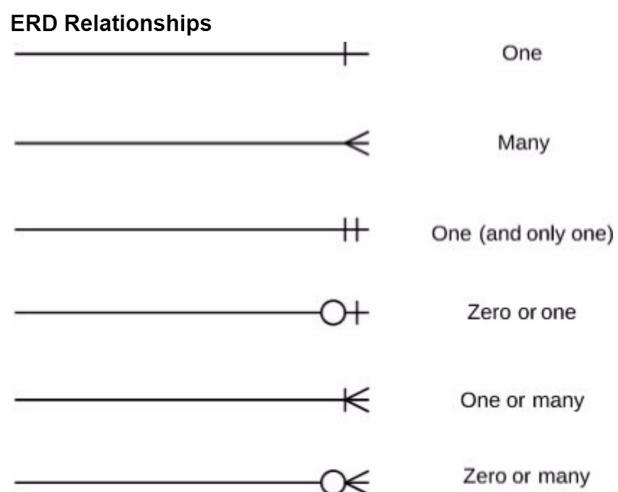
- So we have tools to organize our data.
- Let's think about how useful this can be.
- Can you already think of how certain apps store their data?
- Can you think about how you'll store your own data?

## ERD Lab

- Lab time! Let's break up into groups of four. We're going to discuss and build ERDs for popular website.
- Wait to be prompted to build your ERDs.
- Each group will be chosen to present **one** website on their board towards the end of this lab.

# 5.1 ERD Relationships Diagram

## ERD Example: Employee Database



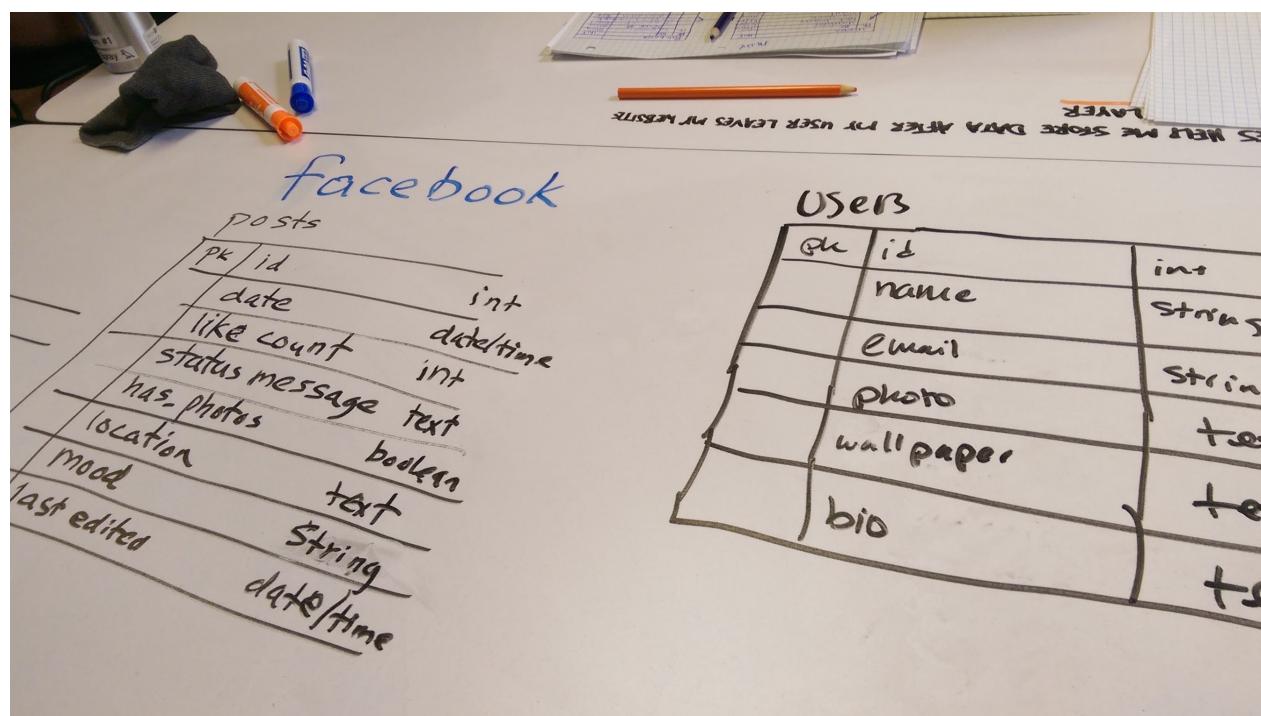
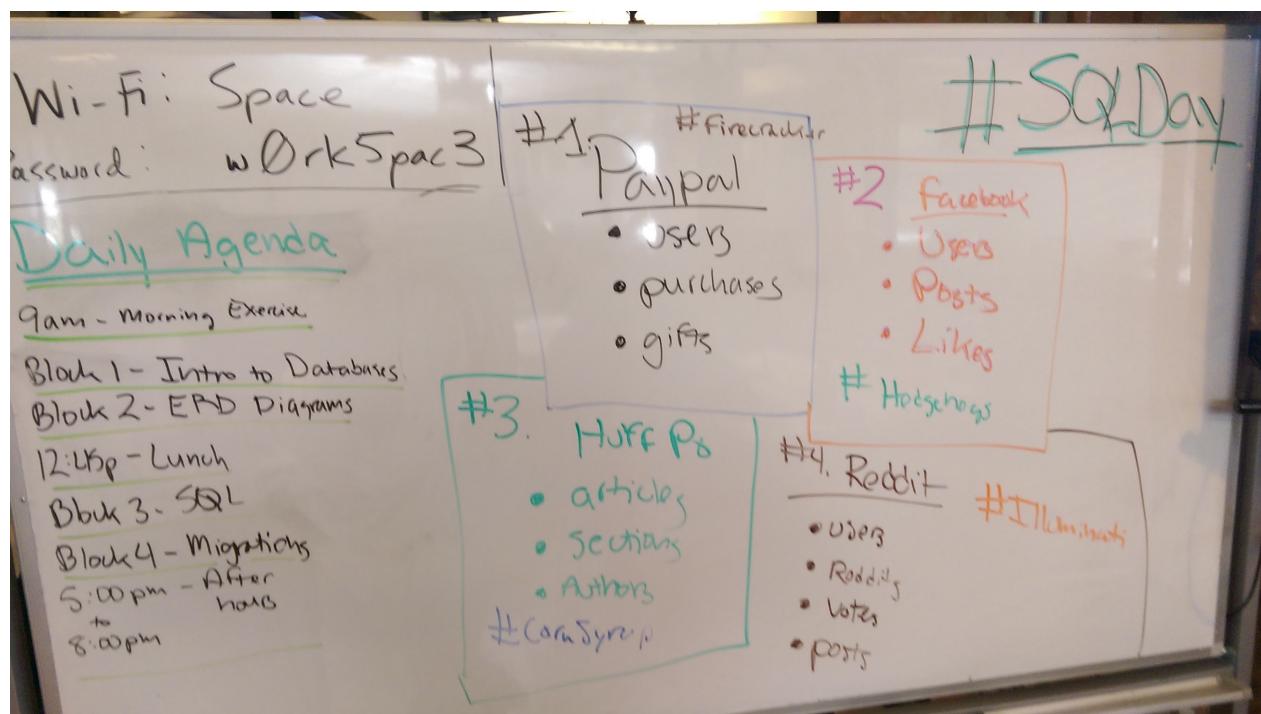


## 5.1 Postgres Data Types

Name	Aliases	Description
bigint	int8	signed eight-byte integer
bigserial	serial8	autoincrementing eight-byte integer
bit [ (n) ]		fixed-length bit string
bit varying [ (n) ]	varbit	variable-length bit string
boolean	bool	logical Boolean (true/false)
box		rectangular box on a plane
bytea		binary data ("byte array")
character [ (n) ]	char [ (n) ]	fixed-length character string
character varying [ (n) ]	varchar [ (n) ]	variable-length character string
cidr		IPv4 or IPv6 network address
circle		circle on a plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer
interval [ fields ] [ (p) ]		time span



## 5.1 ERD Lab



## Lab Slide Deck

- <https://presentations.generalassemb.ly/c1150bdb839d58cdd765#/>



# 5.1 Structured Query Language (SQL)

## Agenda

- Quiz
- Installing Postgres
- Using the Command line to access Postgres
- Creating a Database
- Connecting to a database
- Creating Tables inside of a Database
- Adding rows to a database
- Selecting rows from a database
- Deleting rows from a database
- Migrations

## QUIZ

1. ERDs allow you to visualize.... \_.
2. Databases store the attributes of \_ in structures called \_.
3. Tables can be linked together by natural \_.
4. What are some types of relationships between tables?

## Installing Postgres



1. Run the command `brew install postgres`
2. Let's install **Postgres.app**
3. Move the app to your `/Applications/` directory.
4. Now, double-click it to run it.
5. Select **Open Postgres** in the bottom-right corner.

## LIST ALL THE DATABASES!

- We've provided a cheatsheet of Postgres commands to help you out.
- First thing's first - let's see if we have any databases!
- We can list **all** databases using the following command:
- `\l`

## Creating a database

- This is pretty straightforward!
- Let's tell Postgres to create a database with a name!
- Syntax **matters**. Close each statement with a `;`
- `CREATE DATABASE vader;`
- Run the list command to see our new database!

## Connecting to your database

- Time to connect to our database.

- We can do that with the following syntax:
- `\c databasename`
- In our case, `\c vader`
- We can't do anything inside of our DB until we connect to it.

## Listing tables in a database

- To see a list of all tables in a database...
- Run the following command:
- `\dt`

## Creating tables in a Database

- We specify to `CREATE TABLE name_of_table();`
- We pass in attribute names inside of the parentheses.
- `CREATE TABLE students (id SERIAL PRIMARY KEY, name varchar(255), email String);`
- Let's break this down...
- `SERIAL PRIMARY KEY` sets our Primary Key.
- We can now define the rest of our values using `attribute_name` with an associated `value_type`
- We organize them using commas to split them up.

## Adding rows to a database

Check this syntax out:

```
INSERT INTO students (name, email)
VALUES
('James', 'jamest@ga.co');
```

- We don't need to include a Primary Key.
- Why? Remember auto-increment?
- When we add a new row we get an automatic ID!
- We just specify the attributes to add into.
- And then we specify the **Values!**

## Selecting rows from a database

- We can select all the rows!

- We use the **SELECT** statement!

```
SELECT * FROM students;
```

- We can look for specific rows!

```
SELECT FROM STUDENTS WHERE id = 1;
```

## Deleting rows from a database

- We use the **Delete** keyword!
- We can be specific like SELECT!

```
DELETE FROM students WHERE id = 1;
```

Nice, eh?

## Migrations

Let's think about this. Wouldn't it be nice if we just had a script that we could copy/paste all of this into? Especially when we want to let a new user try our app out or run it on a server somewhere - having a script that can create our databases and tables for us is a *huge* win!

- We need to connect to SQL
- Create our database
- Create our tables
- How would we write this?
- Each command would be line-by-line

## Migrations Lab

We're going to build do something fun!

## 5.1 Postgres SQL Cheat Sheet

### Getting Started

- Open Postgres in Terminal (bash): `psql`
- List all databases in `psql`: `\l`
- Connect to a database in `psql`: `\c database_name`
- List all tables in current database: `\dt`
- Quit `psql`: `\q` + `ENTER`

### SQL Commands

- Create a database: `CREATE DATABASE db_name;`
- Create a table: `CREATE TABLE table_name (id SERIAL PRIMARY KEY, name varchar(255));`
- Read all values: `SELECT * FROM table_name;`

## 5.1 SQL Examples

```
create a table
CREATE TABLE students(id SERIAL PRIMARY KEY, name varchar(255), email varchar(255));

insert new rows into a table
INSERT INTO students (name, email)
VALUES
('James', 'jamest@google.com');

select all rows from a table
SELECT * FROM students;

querying... or finding
SELECT * FROM students WHERE id = 1;
SELECT * FROM students WHERE name = 'James';
```

## 5.1 SQL: Twitter Migrations

Below is an example of a `migrations.sql`. It models out **Twitter**.

### Comments

- **SQL keywords:** UPPER CASE
- **names (identifiers):** lower\_case\_with\_underscore
- `#comments in sql`
- You create a table via stating `create table`!
- You then give it a name in `snake_case`
- You then have a set of `parameters` inside of `()`
- Inside you `declare_name SQLTYPE()` for `names` of things and the `SQL datatype`
- Example:
- `CREATE TABLE some_table_name_snake_case (id SERIAL PRIMARY KEY, some_value varchar(255));`

### Example

```
#1 create db
CREATE DATABASE twitter;

#2 connect to db
\c twitter

#3 create tables
CREATE TABLE users (id SERIAL PRIMARY KEY, name varchar(255), password varchar(255),);

CREATE TABLE tweets(id SERIAL PRIMARY KEY, user_id integer, tweet varchar(140));
```

### Adding A New Feature!

1. The fine folks at Twitter want *you* to add a new feature to their platform!
2. You are to add `targeted_ads` to their platform!
3. `targeted_ads` has a few properties...
  - A primary key
  - A string value of `ad_name`

- A string value of `ad_text`
  - A numerical value of `total_unique_views`
4. Add this table to your `migrations.sql` after testing to verify it works!

## Oh snap! It is a hit! Time to add another feature!

1. Twitter now wants to support emoji!
2. You are to add `emoji` to their platform!
3. `emoji` has a few properties...
  - A primary key
  - A string value of `emoji_name`
  - A string value of `emoji_value`
  - A numerical value of `total_uses_of_all_time`
  - A Boolean value of `is_enabled` ... for those seasonal emoji!
4. Add this table to your `migrations.sql` after testing to verify it works!

## 5.1 Homework

### 1. One more ERD...

- Create an ERD (on paper, paint, photoshop, whatever...) for Instagram
- Take a photo with your camera / save it to your `05.../your_name/` folder as `instagram_erd.png`
- You should have the following tables:
  - Users
  - Photos
  - Hashtags
- Unsure how to use Instagram? Ask a classmate or look it up!

### 2. Migrations

- Create a `migrations.sql` for your Instagram ERD!
- So create a database, connect to it, and add tables!

### 3. Migrations

- Create a `migrations.sql` for each of the four applications you had to model today!
- So create a database, connect to it, and add tables!
- Those apps are...
  - Paypal: users, purchases, gifts
  - Facebook: users, likes, status\_updates
  - Huffington Post: sections, articles, authors
  - Reddit: users, subreddits, votes

### 4. Adding and Querying Data

- In a file called `add_query.sql` ...
- Create a new script with these commands (after testing these commands in the database).
- In one of your database migrations created below, insert 10 rows of data into each table.
- Select the first item of each table.
- Select all of the items in a table.



## 5.2 Building 'Physical' Servers

### Objectives

Today, we are going to build and connect to remote servers using *Digital Ocean* as our platform. We will also build a baseline provisioning script to allow for expedited server creation in the future.

### Slide Deck

- <https://presentations.generalassemb.ly/abcf7108c93adfc1e3ba#/>

## 5.2 Building a Web Server

### Agenda

- Web Servers
- Message-Passing and Listening
- Cloud Computing
- Goals
  - Create a cloud-based web server
  - Connect to the cloud based server and configure it
  - Deploy your Sinatra application to the web server
  - Run migrations and update environment / API keys
  - Use Rack host your application for the world to see!
  - Use DNS to assign a domain to your website

### Web Servers

- We've been using web servers for a while ...
- Heroku is the primary example
- Your laptops are also web servers
- You've used Rack (Rackup) to serve applications

### Message-Passing and Listening

- The internet is a series of messages
- Messages that are passed back and forth
- Web servers **listen** for these messages
- And react appropriately
- For example, Rackup listens on port 9292

### Cloud Computing



## Cloud Computing

- Companies have been investing in infrastructure
- Building out platforms that allow us to spend pennies for processing power
- They build out giant warehouses of small computers
- And we run our web servers on them
- This is extremely cost effective for us!
- But we never own the hardware

## Today's Goal

Get your Sinatra project from last week online using Digital Ocean! You'll also learn how to build a **web server**!

## Creating a Web Server

- We need to register Digital Ocean accounts
- Sign up for a new account
- GA is going to give you \$50 in credits to use!
- Warning: make sure you setup a billing threshold
- Your credit card **could** be charged

## Learning Objectives

1. Register for Digital Ocean and get teh monies!@!@12
2. Identify what Droplets are and how to create one
3. Work with a remote server (droplet) via command line
4. Get a working Sinatra app on the web for all to see!

## 5.2 Deployment with Digital Ocean

### Introduction

*Digital Ocean* is a **Platform as a Service (PaaS)**. It is designed to do one thing and to do it well - create instances of web servers for you to use. Once you're out working on a job, you'll be exposed to a variety of PaaS providers - Amazon's EC2, Heroku, AppHarbor, and of course, Digital Ocean. Today you're going to sign up for Digital Ocean, create a droplet (their word for a server), and write an app and put it out for the world to see! Because as cool as it is to show your classmates what you've been working on, wouldn't it be cooler to show your friends and prospective employers? Plus, this will give you a leg up when looking for a job: being able to say **I can configure and setup a server** is kind of a big deal.

### 1. Registering for Digital Ocean

*Digital Ocean requires a credit card on file. General Assembly is providing free credits for our students so you should not need to worry about being billed; however, please be aware that if you run out of credits, your card will be billed.*

1. Browse to <https://www.digitalocean.com/>
2. Sign up for a new account on the main page by entering your email address and creating a password.
3. You'll be sent a confirmation email (can take up to 5 minutes).
4. Once you confirm your account, you'll need to enter in credit card information. Do so.
5. Now, before doing anything else, select the *profile* icon and select *billing*.
6. Enter in the unique code sent to you on Slack for your free credits!

### 2. Identify what Droplets are and how to create one

A **droplet** is a scalable server offered by Digital Ocean. Digital Ocean supports a variety of Linux platforms to develop on (amongst others). A droplet can serve a small website that you use for your own portfolio and it can scale up to host an enterprise application! One of the best things about a droplet is that it can scale - if your site blows up, you can expand the resources it has without needing to create a new server!

- Now, we need to locate something to make logging into our droplet easy and secure.

## WE DO - Locate an SSH Key

We need to create a secure way for you to log into any Droplet that you create. We're going to use an private key that you already are using on your computer. You should only share private keys with entities you trust! I only share mine with my computers and the servers I run. I even have a copy of mine in my will! They're private!

Because we want to make sure that you and only you - not some hacker in Russia, not some script kiddie in China - has access to your droplet, we'll use a private key that we're already comfortable with to connect to the server.

Open up terminal and enter in the following commands:

1. `ls -al ~/.ssh` - list all of the keys in the `./ssh` directory. You should see an `id_rsa.pub`. This is your public key.
2. `atom ~/.ssh/id_rsa.pub` - Open the key in atom so we can use it in just a moment.

Now that we have our SSH key, it is time to create a droplet!

## YOU DO - Create a Droplet

1. Log in to Digital Ocean if you have not already done so.
2. Select "Create Droplet" in the top-right corner.
3. Give your droplet a name. It can be `my-site` or `myawesomesite.com`. The name is just used for reference.
4. Select the \$5/month size for your Droplet.
5. Select a region.
6. Ignore the available settings.
7. Select the **Ubuntu 14.04 x64** operating system.
8. Select **Add SSH Key**. You will copy/paste the SSH key that we retrieved just moments ago into the text box.
9. Select **Create Droplet**.

10. Annnnnnd we wait!

### 3. Work with a remote server (droplet) via command line

#### Log in and setup a server in 10 steps!

1. Log into the remote server (Droplet)
2. `ssh root@0.0.0.0`
3. I'm magically logged in because it used my private key from earlier to authenticate who I am!
4. I need to update the system! `apt-get update`
5. I need to install Ruby! `apt-get install ruby 6.`
6. Now that I have Ruby installed, I can now install gems! `gem install bundler , gem install pry`
7. Awesome, I have everything I need to run an app! Time to make one!
8. `touch Gemfile config.ru app.rb`
9. `nano Gemfile` , ctrl-x to exit, S to save; repeat x3

#### Basic Provisioning Cheat Sheet

- `ssh root@[your.ip.address.here]`
- `apt-get update > y at prompt`
- `apt-get install git`
- `apt-get install ruby`
- `gem install bundler`
- `gem install pry`
- `bundle`
- `rackup config.ru --port 80 --host 0.0.0.0`

#### JSON Gem Warning!

An error occurred while installing json (1.8.2), and Bundler cannot continue. Make sure that `gem install json -v '1.8.2'` succeeds before bundling. `root@coffee-shop:~/coffee-shop.ninja# gem install json -v '1.8.2"`

`apt-get install build-essential -- plox`

```
apt-get install ruby-dev
```

## Want to run your `rack` application and KEEP it running...

```
nohup bundle exec rackup -p 80 --host 0.0.0.0
```

### What is nohup?

- `nohup` : Do not listen to the `hup` signal when terminal is closed
  - `bundle exec` : Use the gem versions in the `Gemfile.lock` to execute the command
  - `rackup -p 80` : Run the application on port 80
  - `&` : Run this command in the background
- 
- **Nano**
  - Exit - `ctrl-x`
  - Prompts you to save - select Y or N
  - Prompts you to confirm where saving. Either edit or press return/enter.

## 5.2 Building a Server in a Nutshell

```
apt-get update
updates the list of software our server knows about

apt-get install ruby-dev
installs the tools we need for a ruby environment

apt-get install build-essential
essential build tools such as GCC

apt-get install git
what's going on with git right now?

postgres database!
apt-get install postgresql
apt-get install postgresql-contrib

create a user
sudo -u postgres createuser -s $USER
createdb $USER
touch ~/.psql_history
apt-get install libpq-dev
gem install pg

gems!
gem install json -v 1.8.2
gem install bundler
```

## 5.2 HOW DO WE GET AN APP ON OUR SERVER?

1. We need a Git repository!
  - Use HTTPS
  - *Example:* <https://github.com/ga-chicago/wedding-rsvp.git>
2. We also need a working database!
  - Opening PSQL
  - Copy / run migrations + hit enter
  - Exit with \q
3. Change into the directory
4. `bundle`
5. `nohup bundle exec rackup -p 80 --host 0.0.0.0`

## 5.2 Homework

For the first portion of your extended weekend, we have a Javascript challenge for you! This will require a lot of research and you'll likely have to look some of these terms up on MDN (and that's ok). You're going to take a look at code in Javascript that allows you to:

1. Select an image using a `<input type=file>` element
2. Convert the image into usable text (using a **Base 64 Encoding**)
3. A live, working example is here: [http://ga-chicago.github.io/filereader\\_base64/](http://ga-chicago.github.io/filereader_base64/)

### Source Code

- Project URL: [https://github.com/ga-chicago/FileReader\\_Base64\\_Upload](https://github.com/ga-chicago/FileReader_Base64_Upload)
- Clone this down to your computer
- Open the `app/index.html` in Google Chrome
- Open DevTools to inspect the source code

### What's going on here?

- Using the Javascript FileReader API, we can upload images as Base64 strings.
- We log out the base64 string via the `event` in line:67 of `index.html`.
- All FileReader related elements and code are contained in the `index.html` file.
- This project contains the MUI CSS framework and corresponding libraries.

### Code Behind This

```
var reader = new FileReader();
reader.onload = function (event) {
 // try to read whatever file has been 'readAsDataURL'
 try {
 // event target result is our base64 encoded type
 // this is whatever file has been reader during 'readAsDataURL'
 console.log("File as base 64:");
 console.log(event.target.result);
 // catch an error if one occurs...
 } catch (ex) {
 // output a warning in the DevTools console
 throw new Error("Couldn't convert file: " + ex);
 }
}
// read the file argument
reader.readAsDataURL(binaryData);
```

## Homework Assignment

- Inspect how this source code works
- Create a brand new Sinatra app inside of `your_name` for this week.
- Render a `view` that allows users to convert an image into text
- Place that text version inside of a new `<input type='text'>` element as a value so users can copy/paste this.
- Create a pull request when it works.

## 5.2 Bonus Homework: Dragon Latin

We're going to play around with Ruby! Let's imagine that we want to build a method that would check paragraphs of texts for words. Let's define a method that accepts in some blob of text (as a `String`) called `dragon_latin`.

Translating English to Dragon language is pretty simple. A general rule of thumb for translating is below.

1. **For Each** word in a paragraph
2. **If** the words has more than 1 character
  - Step 1: The last letter of each word becomes the first character
  - Step 2: "ur" is appended to the end of the word
3. **Else**
  - The word remains unchanged

### Example:

- English: "I think we love the world"
- Dragon Latin: "I kthinur ewur elovur ethur dworlur"

### Starter Code

```
def dragon_latin(text)
 output = text.split(' ').map do |word|
 ## code to go here
 ## remember that strings are just arrays of characters
 ## walk through the steps for translating in the code here
 ## the output will be returned below
 end

 return output.join(' ')
end
```

## 5.3 Models & Active Record

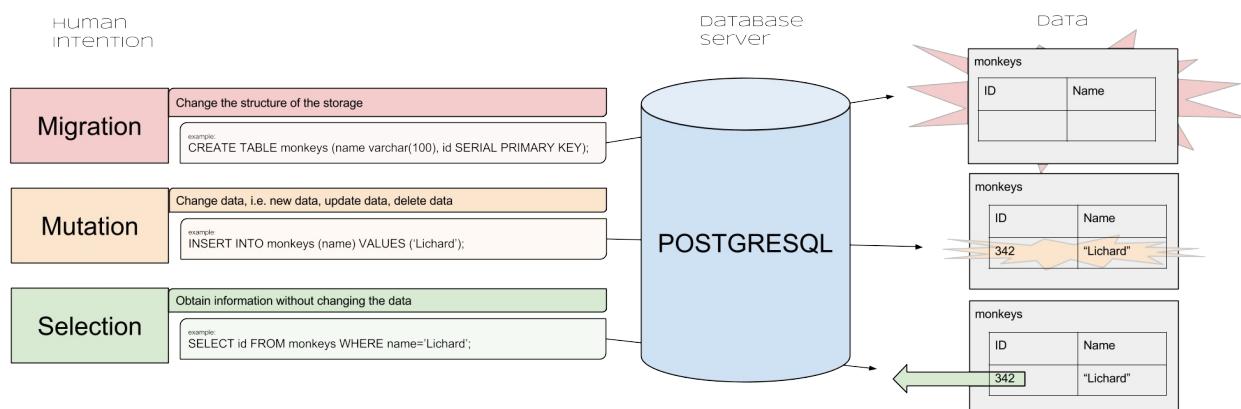
Welcome back! Hopefully you're stuffed full of delicious food from Thanksgiving! Leftover turkey sandwiches, anyone? Today we're going to connect SQL databases to Ruby. To do this, we'll create Models and utilize a tool called ActiveRecord.

### From Database to Model

Let's warm up by creating a database and then accessing it via Ruby!

#### Objectives:

- Understand how to create a Model
- Create a database for your Model
- Create a table for your Model
- Add some content to your table
- Understand the concept of an ORM
- Install ActiveRecord
- Connect to a database using ActiveRecord



### Create, Read, Update, & Destroy Data

Now that we can use ActiveRecord, it is time to manipulate some data...

#### Objectives:

- Instantiate and create a new instance of a Model using ActiveRecord
- Query a model to read it using ActiveRecord
- Edit a model's attributes Active Record

- Destroy a model using ActiveRecord

## 5.3 Wireframe Shopping List

This week we're going to build a shopping list application. This application will allow you to log in to a Shopping List website. From there, you will be able to view, add, update, and remove items that you need to purchase. The business logic is the same for all applications but feel free to get creative with the design and wording of the app.

We're going to build out wireframes for this application. To get started, we need to think about the `views` that our project needs. By now, you should have created (but maybe not designed) the following views:

- Account Registration
- Account Login
- List of Shopping Items

**Whatever you are unable to finish this morning should be completed during workshop time this afternoon.**

### Wireframe #1: Account Registration

This wireframe must contain the following fields on a `registration` view:

#### Registration:

- User's email
- User's password
- User's password (confirmation)
- Submit Box

Upon submission, a user will either be auto-logged in + welcomed **or** warned that some sort of registration error occurred. We need two additional views for these:

#### Successfully Registered:

- Congrats! Welcome aboard message.
- A redirect link to their shopping list page.

#### Error when Registering

- Something went wrong!
- Try registering again!

- A link back to the registration page
- 

## Wireframe #2: Login

Pretty much the same requirements as the registration, minus the double-password confirmation.

### Login:

- User's email
- User's password
- Submit Box

### Successfully Logged In:

- Congrats! Welcome back message.
- A redirect link to their shopping list page.

### Error when Logging in

- Something went wrong!
  - Try logging in again!
  - A link back to the login page
- 

## Wireframe #3: Shopping List

We are going to need a few views for our shopping list. Those are:

### A List All View for Shopping Items:

- A table of all shopping items
- Each row has the name and quantity of each item
- Each row also has a button/icon to **update** or **delete** an item.
- There is also a **create** button on the page that allows users to add new shopping items.
- There should also be a location on the page that shows the user a message (such as 'item added successfully!' or 'item deleted successfully!').

### Create Shopping Item:

- A form to add a shopping item by name and quantity

- A submit button that adds the item

### **Confirm Deletion of Shopping Item:**

- A form to confirm deletion of an item with the name and quantity of it.
- A submit button that deletes the item

### **Update Shopping Item:**

- A form to update a shopping item by name and quantity
- A submit button that updates the item's information

## 5.3 Models with ActiveRecord

### Agenda

- Quiz
- Using Ruby to talk to databases
- Object-relational Mapping (ORM)
- Introduction Active Record
  - Required Gems
  - Connecting to the Database
  - Creating Models
- CRUD with Models
  - Create
  - Read
  - Update
  - Destroy
- Lab/Exercise

### QUIZ

1. What command in `psql` connects to a database?
2. What does SQL stand for? What is it meant to do?
3. What does `SERIAL PRIMARY KEY` do when assigned to a column?
4. If I `INSERT` into a Table, what am I doing?
5. If I `SELECT` in a Table, what am I doing?

### Using Ruby to talk to databases

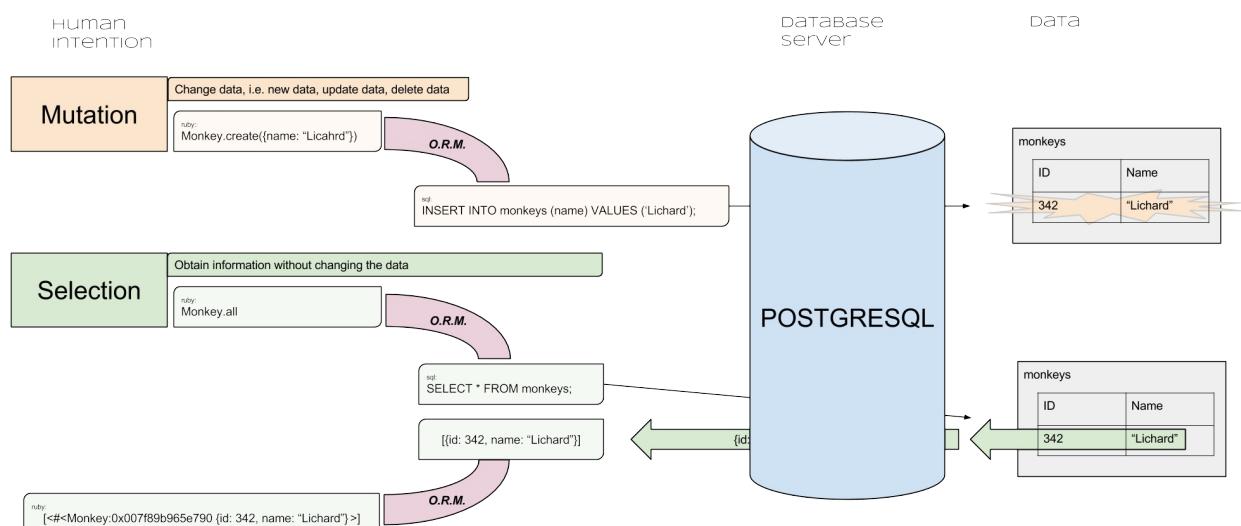
- Wouldn't it be nice if you could just *talk* to databases using Ruby?
- That'd make life a lot simpler, wouldn't it?
- Understanding SQL is important...
- But what if we could make our lives as developers just a little bit easier?

### Object-relational Mapping (ORM)

- ORMs are designed to **map** databases to Objects
- Imagine having an Object that acts as translator between you and SQL!

- It is the Babel Fish of SQL!
- They really simplify development...
- ...and make our lives easier

## Human Intention behind Active Record



## How ORMs Work

- **ORMs** are designed to perform **transactions**.
- When a program makes a request to the database, the transaction **begins**.
- The ORM then translates the request into SQL!
- It then attempts to run the SQL query that it created...!
- If changes are made to the database **and** the query was successful the changes are **committed**.
- If the query fails... no changes are made and the transaction is **rolled back**.
- If this occurs, an error is thrown.

## Introduction to ActiveRecord

- ActiveRecord is an ORM written in Ruby
- It was originally designed for Rails
- It has been ported to Sinatra and other languages
- It is reallllllllllyyyyyyyy awesome.

## Required Gems

You need to include the following libraries in your **Gemfile**:

```
gem 'pg' #postgresql
gem "sinatra-activerecord" #activerecord orm
```

## Connecting to the Database

In your application file, you need to establish a connection to your database using ActiveRecord. We do so by calling the `establish_connection` method that ActiveRecord provides. We'll pass in the following arguments:

- `:adapter => "postgresql"`
- `:database => "your_db_name_here"`

That's it! From here on out, you can query and modify the database with ease!

## Creating Models

- To create a model, we need to create a new file.
- The structure of the naming should be `TablenameModel.rb`
- We can then include this file using the `require('./TablenameModel')` command
- A model... is simple.
- Let's take a look at one!

---

## Sample Model

```
class Students < ActiveRecord::Base
end
```

Great, isn't it?

## CRUD

CRUD stands for 'create', 'read', 'update' and 'destroy'. You perform these actions on tables inside of your database. Let's imagine that you have the following SQL table named `Sms`:

id	message	sender
1	"Hi there!"	"Kathew"
2	"sup man."	"Lichard"
3	"Just learned Rails!"	"Kathew"
4	"hey you guys!"	"Cecelious"

## Create

```
sms = Sms.new # create a new
sms.message = "Hi there!" # add values
sms.sender = "Kathew"
sms.save # saves to the database
```

## Read

### Find an object in a list of objects

```
sms = Sms.find(2) # find by id
=> sms = { id: 2, message: "sup man.", sender: "Lichard" }
sms.message
=> "sup man."
sms[:message]
=> "sup man."
```

## Update

```
sms = Sms.find(4) # find by id
sms.sender = "sms ninja" # update a value
sms.save # save to database
```

## Destroy

```
sms = Sms.find(1) # find by id
sms.destroy # delete from database
```

## Conclusion

- Is ActiveRecord awesome?

- Yes?
- Ready to use it?

## Lab

### Migrations to Models

- We're going to create a students app based on our `dragons` database!
- We need to connect to our database!
- We need to create Models for our `students` table!
- We need to add items using ActiveRecord!
- We need to read them via JSON!

## 5.3 Model CRUD with Active Record

### Getting Started

You need to include the following libraries in your **Gemfile**:

```
gem 'pg' #postgresql
gem "sinatra-activerecord" #activerecord orm
```

### Establishing Connection

In your application file, you need to establish a connection to your database using ActiveRecord. We do so by calling the `establish_connection` method that ActiveRecord provides. We'll pass in the following arguments:

- `:adapter => "postgresql"`
- `:database => "your_db_name_here"`

That's it! From here on out, you can query and modify the database with ease!

### CRUD

CRUD stands for 'create', 'read', 'update' and 'destroy'. You perform these actions on tables inside of your database. Let's imagine that you have the following SQL table named *Sms*:

1	"Hi there!"		"Kathew"
2	"sup man."		"Lichard"
3	"Just learned Rails!"		"Kathew"
4	"hey you guys!"		"Cecelious"

### Create

```
sms = Sms.new # create a new
sms.message = "Hi there!" # add values
sms.sender = "Kathew"
sms.save # saves to the database
```

## Read

### Hash Recap

```
obj = { id: 3, message: 'Hi there!', sender: 'Kathew' }
```

### Accessing values via symbols Recap

```
obj[:message]
=> 'Hi there!'
obj[:message] + " (sent from " + obj[:sender] + ")"
=> "Hi there! (sent from Kathew)"
```

### Find an object in a list of objects

```
sms = Sms.find(2) # find by id
=> sms = { id: 2, message: "sup man.", sender: "Lichard" }
sms.message
=> "sup man."
sms[:message]
=> "sup man."
```

## Update

```
sms = Sms.find(4) # find by id
sms.sender = "sms ninja" # update a value
sms.save # save to database
```

## Delete

```
sms = Sms.find(1) # find by id
sms.destroy # delete from database
```

## 5.3 ERB (Embedded Ruby) In Depth

Imagine that we have a route that supplies us some information. In our route, we can declare a few variables for use in an **ERB** page. We then call the **ERB** method to load a **view**. For example:

```
get '/darth_vader' do

 # let's have some variables
 name = 'Darth Vader'
 force_power = 'Force Choke'
 lightsaber_colour = 'red'

 # load a view using ERB
 erb :darth_vader

end
```

When this route is called, `/views/darth_vader.erb` will be rendered. The ERB file looks like the following:

```
<h1><%= name %></h1>

<p><%= name %> uses his <%= force_power %> to get his job done. He rocks a <%= lightsab
```

This would render out to...

```
<h1>Darth Vader</h1>

<p>Darth Vader uses his force choke to get his job done. He rocks a red lightsaber.</p>
```

## Rendering Content vs Executing Ruby Code

To **Render** content, we use the `<%= %>` tag. Inside of it, we can place variable names that are declared in our route (or class).

To **Execute** ruby code, we use the `<% %>` tag. Notice the lack of the `=` sign.

## Loops in ERB

Now, imagine we have just fetched a list of `Subordinants` using ActiveRecord in a route. Each one has two attributes: `name` and `rank`.

```
get '/subordinates' do

 # load all the subordinates from the Subordinates table
 subordinates = Subordinates.all

 erb :subordinates

end
```

To render them, we'd want to conceptually just loop through them and put them somewhere. In command line ruby this would look like:

```
subordinates.each do |minion|
 puts minion.name + ' has a rank of ' + minion.rank
end
```

Our **ERB** view will look like that, but uses the *executing* ERB tag to run Ruby code. It also uses the *Render* tag as well... but only when we want to render content.

```
<h1>Subordinates</h1>

<% subordinates.each do |minion| %>

 <%= minion.name %> has a rank of <%= minion.rank %>

<% end %>
```

## 5.3 Why use rake?

- When you make migrations using rake, it makes timestamped migration file for you.
- When you use migrations, you can now have templates for your objects that will be made with postgresql, this lets other users use whatever database you set up instead of it living in your machine only

## Rake

- The rake gem needs a Rakefile to work in order for the gem to know where to put the db and migrate folder
- So in the project folder we need to add Rakefile.rb

```
require "sinatra/activerecord/rake"

namespace :db do
 task :load_config do
 require "./app"
 end
end
```

## Rake commands

- to see all rake commands:
  - `bundle exec rake -T`
- to create migration
  - `bundle exec rake db:create_migration NAME=create_users`
- to migrate database (actually creates tables in database)
  - `bundle exec rake db:migrate`

## 5.3 MOAR REPS with Sinatra's Songs

We need *moar reps*. Let's run through our process again before jumping into full CRUD actions tomorrow. Pick a random partner in the class (someone you haven't worked with before) and create a **new** Github repository. You're going to pair program to create an application called `sinatra_songs`.

1. We need a database to store songs!
2. Songs have `artist` and `title`. Perhaps even a `release_year`?
3. Let's create a `songs` table to store them.
4. Now that we know this works, create a migrations that store the connection/creation information regarding our database.
5. Once our database is all set, create an application in Sinatra!
6. This application should render all of the `songs` in our table on a page.
7. Utilize Models (and ActiveRecord) and Views (ERB) to render them out!

## 5.3 Homework

### Shopping List Wireframes

- Complete the Shopping List Wireframes morning exercise
- Prepare to use these during the week as we work on a Shopping List application!

### My First CRUD App

You're to create an application of your own! This can be anything you want! However, you must meet the following technical requirements:

- Migrations file should be created and tested. It should contain **three** tables.
- A simple Sinatra application that contains **three** routes.
- Each **route** must correspond to a specific model.
- Each **route** must render a View that lists **all** of the rows in a table converted to JSON.

Get creative! You'll create this application in your

`05_fullstack_sinatra/your_name/my_first_crud/` folder.

## 5.4 Controllers: The 'Glue' between the Model & View

### Models, Models, Models

*We're going to spend a good chunk of the day working on models as a second pass today.*

### Controllers and Routing

*We're going to discuss routing and controllers. By the end of this lesson, you will have learned to create dedicated controllers for each model.*

#### Objectives:

- Understand the purpose of a controller (to collect information from models and then render the view)
- Create controllers in Sinatra for various Models
- Render the data from a Controller into a View

## 5.4 Q&A - Models and SQL

*We're going to start by taking some questions on Models and SQL. We'll answer your questions and write them here along with the answers!*

### Table Names in SQL

- Should be plural! :)
- `yelp_reviews`

### What is a Migrations file?

- It is a general SQL file
- Contains how to CREATE database, connect to database, and CREATE tables.
- Meant for re-use in deploying and creating databases and tables on any computer
- Located in `db\migrations.sql`

### What is a seeding file?

- Seeding file (`db\seed.sql`) is designed to populate a database with dummy or test data
- In production environments, we use real data.
- LOTS of `INSERT INTO` statements.

### What is ActiveSupport (in relation to ActiveRecord)?

- It is a set of tools bundled into a library of classes.
- Written for Rails 3.
- ActiveSupport allows ActiveRecord to talk to Rake commands.

### What is the difference between ActiveRecord and Sinatra-ActiveRecord Gems?

- One gem is designed 100% with Sinatra in mind.
- One gem is for Rails only.
- When using Rake for Sinatra, we need ActiveRecord to communicate for lost functionality in Sinatra-ActiveRecord

## Where does my database's information physically live?

- `/usr/local/var/postgres`
- Do not edit this at all!

## Process for Setting up Databases on Servers

- Manually creating a database using a migration file.
- Populating that database with a seed file.
- OR use Rake, Gulp, or something similar.

## What are naming conventions?

- SQL names should be in `snake_case`
- Models name should be `UpperCamelCase`
  - This is because Ruby classes are capitalized
- Binding a table to a name!
- `table_name` correlates to `TableName` in Ruby
- If this does not work and your table cannot be located....

```
class TableName < ActiveRecord::Base
 #specify the actual table's name
 self.table_name = 'table_name'
 # snakesssss
 #self.table_name = 'snakeessss'
end
```

## What is a migrations.sql file?

```
CREATE DATABASE some_name;
\c some_name
CREATE TABLE yelp_reviews;
```

## ActiveRecord Class for this...

```
class YelpReview < ActiveRecord::Base
end
```

## Mapping a Table to ActiveRecord Naturally

- First, we create a table in some database.
- In our case, `yelp_reviews` is our table name.
- ActiveRecord Models should map up to the name of our table.
- For example...
- Our model will be called `YelpReview`
- If this does not work, use the `self.table_name` fix above.
- Models should not be plural
- ActiveRecord tries to map a singular model name to a plural (or singular) table name.

## Reserved names

- These protect software from users
- Especially so nothing can internally be overwritten
- Google-fu reserved words (ie: `user` )

## Naming Conventions

- Model `UserAccountsModel` (C#, Java, Django)
- Model `UserAccount` (Rails)

## Naming Files

- Models should be `name_of_model.rb`
- For our `YelpReview` model...
- `yelpreview.rb`

## ActiveRecord's SQL Counterparts

- [http://guides.rubyonrails.org/active\\_record\\_querying.html](http://guides.rubyonrails.org/active_record_querying.html)

## How to connect to a Database

*ActiveRecord*

```
ActiveRecord::Base.establish_connection(
 :database => 'our_db_name',
 :adapter => 'postgresql'
)
Res.all
```

*Without ActiveRecord*

```
require 'pg'
conn = PGconn.open(:dbname => 'test')
res = conn.exec('SELECT 1 AS a, 2 AS b, NULL AS c')
res.getvalue(0,0)
```

## Escaping Characters

To escape apostrophes within your string, just type another one before it. For example,

Doc B''s

## 5.4 Bundling Your Knowledge Together

We're going to put together a project based on what you have learned this morning. This will reinforce your understanding of how to build an application. Together, we're going to build a step-by-step guide on how to get a Sinatra application put together from start to finish.

### Subject Matter

Early on in the course we asked students to recommend places they would refer to newbies to the Chicago area. Some of you contributed to the repository here:

<https://github.com/ga-chicago/recommendations>

Today, we're going to take that data and create an application out of it. By the end of the morning, we're going to have a fully functional API and application that lists all of the locations!

## 5.4 Sinatra Walkthrough guide

### Database Design

- We're going to draw out ERD (Entity Reference Diagrams)
- Create a `db\migrations.sql` file.
  - `CREATE DATABASE`
  - `\c db_name`
  - `CREATE TABLE`
- Populate database with test OR production data. We do this with a `db\seed.sql` file.
  - Lots of `INSERT INTO` statements!

### Creating a Sinatra Application

- We need 3 files, right?
- `app.rb` , `config.ru` , `Gemfile`

### Adding Models

- Create a model!
- `models\name_of_model.rb`
- Now, require it in our `config.ru`

### Project Code

- <https://github.com/code-for-coffee/chicago-recommendations>

### Videos

- Part One: <https://youtu.be/gKg0IpFKtOg>
- Part Two: <https://youtu.be/dzxzemDg8GY>
- Part Three: <https://youtu.be/wkwHV6x-UIA>

## 5.4 CRUD Controllers

When building out an application, it sure is nice to have a starter template. Here is a CRUD controller template to get you started! It includes appropriate GET and POST methods for CREATE, READ, UPDATE, and DESTROY. The Index is designed to list all items.

```
class Controller < ApplicationController

 # list all
 get '/' do
 erb :index
 end

 # create
 get '/create' do
 erb :create
 end

 post '/create' do
 erb :create_success
 end

 # read
 get '/read' do
 erb :read
 end

 # update
 get '/update' do
 erb :update
 end

 post '/update' do
 erb :update_success
 end

 # destroy
 get '/destroy' do
 erb :destroy
 end

 post '/destroy' do
 erb :destroy_success
 end

end
```



## 5.5 Relationship Advice

*Today we're going to take a look into MVC in Depth as well as how to relate models in ActiveRecord.*

### MVC Recap

- Video for Lecture: <https://www.youtube.com/watch?v=wVm-KFexMal>

### Lab: Tskrrr!

You're going to be given a fully operational CRUD operation. Except we've removed code from your `migrations.sql` and `TaskController`! You need to use ActiveRecord and your Database wizard skills to get the app up and running.

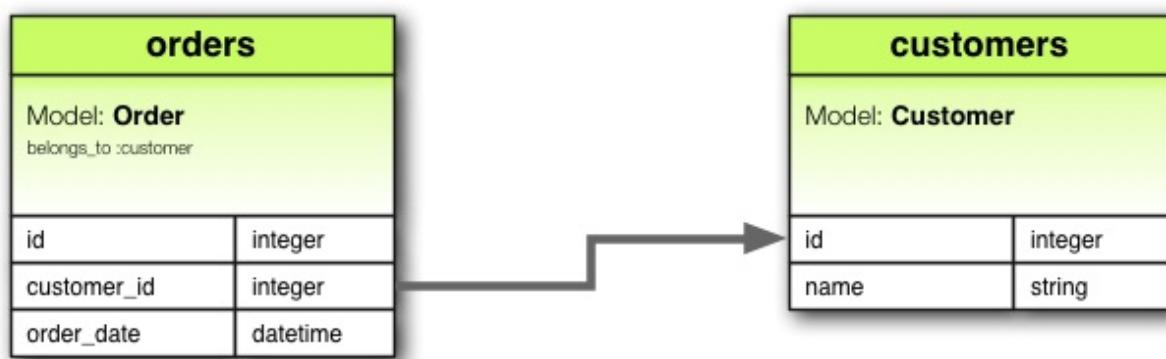
1. Fork this repository: <https://github.com/ga-chicago/Tskrrr>
2. Clone it to your WDI directory.
3. With a partner, inspect all of your code.
4. Solve the missing code with the notes inside of both files.

### ActiveRecord Relationships

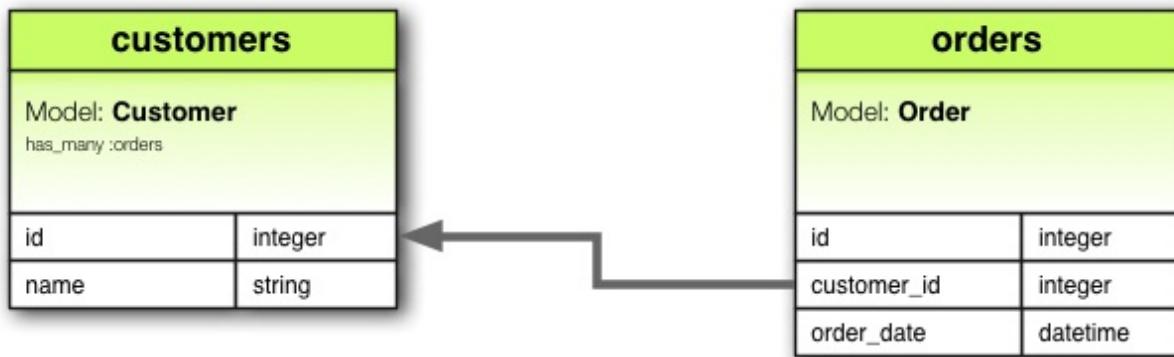
- Video for lecture: <https://www.youtube.com/watch?v=23vJacmW88k>
- Lecture code: [https://github.com/alcastaneda/active\\_record\\_lesson](https://github.com/alcastaneda/active_record_lesson)

## 5.5 ActiveRecord Relationships

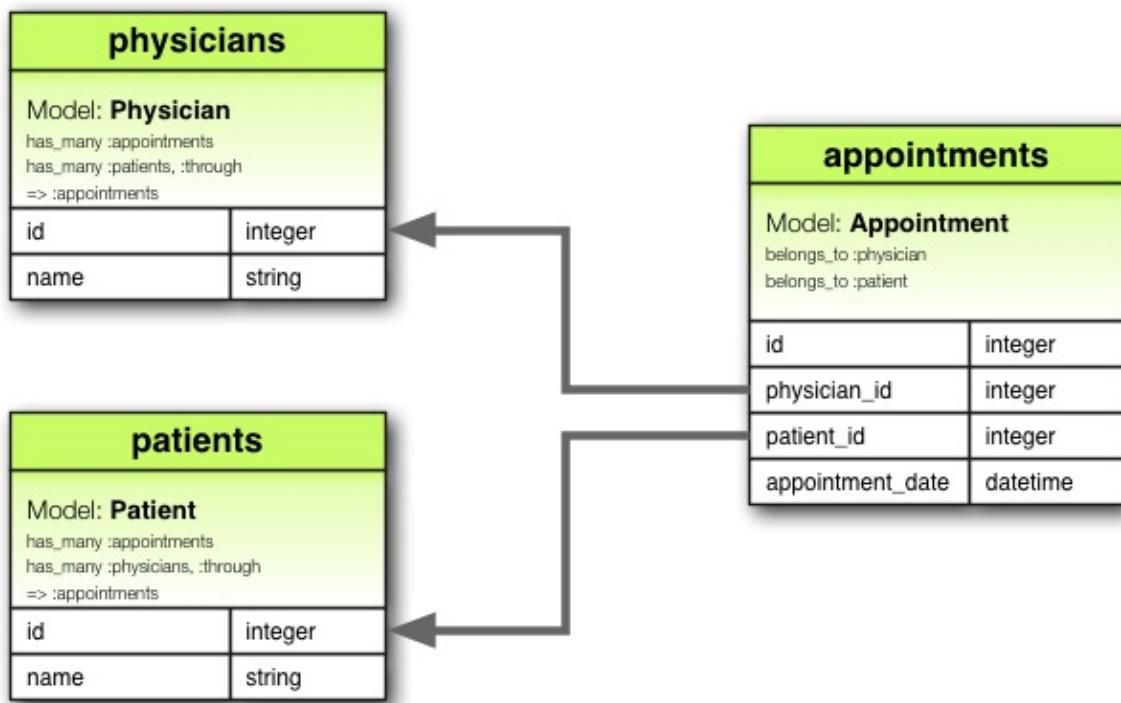
- ActiveRecord establishes relationships between tables in the model files
- Basic ActiveRecord relationships:
  1. belongs\_to - one to one relationship
  2. has\_many - one to many relationship
  3. has\_many through: - one to many relationship through a join table



```
class Order < ActiveRecord::Base
 belongs_to :customer
end
```



```
class Customer < ActiveRecord::Base
 has_many :orders
end
```



```

class Physician < ActiveRecord::Base
 has_many :appointments
 has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
 belongs_to :physician
 belongs_to :patient
end

class Patient < ActiveRecord::Base
 has_many :appointments
 has_many :physicians, :through => :appointments
end

```

## More complex relationships

When you want to be more explicit in naming properties you would have to use some of the following ActiveRecord declarations:

1. `class_name`: - when you use a different name for a model you need to tell ActiveRecord what model to use

2. foreign\_key: - when you use a different name for a model id in your table you need to tell ActiveRecord what value in your table to use instead
3. source: - when you go through a table and use a different name for a model id you need to tell ActiveRecord what value in the table to use instead

## Documentation

ActiveRecord Documentation is very easy to read and understand! If you have any questions about ActiveRecord look at the documentation. If you need any help let me know! ~Adriana

Documentation: [http://edgeguides.rubyonrails.org/association\\_basics.html](http://edgeguides.rubyonrails.org/association_basics.html)

## 5.5 Homework: Shopping List

This evening, you're going to build out your shopping list application. This is going to require you to build out a full-stack MVC application in Sinatra. You will need **models**, **controllers**, and **views**. You built out the wireframes for this application earlier in the week; now you're going to code it!

You will create a brand new Github repository (if you do not already have one) for your shopping list.

### Database Requirements

You must have a database that contains a single table: `items`. We will also add an `accounts` table **later** (do not add this one now). The `items` table should contain the following columns:

- `name` as a String
- `quantity` as an numeric type

### Technical Requirements

Your application must have the following parts to it:

- Your `migrations.sql` and `seed.sql` should be inside of a `db/` folder in your project.
- ApplicationController to get your application started and connected to your database.
- ItemsController to handle all of the routes to your root (`/`) route and any resources under it. The ItemsController should contain complete CRUD functionality. Users should be able to `create`, `read`, `update`, and `destroy` items on their shopping list.
- Corresponding views for each CRUD action.
- A basic `readme.md` in your repository explaining what the application is.

### Homework Submission

- Push all of your work online.
- Email your instructors when you have completed the repository. Our email

addresses are (for copy-paste): `jamest@google.com`, `james.haff@google.com`,  
`adriana.castaneda@google.com`

- Provide a link to your repository in the email.
- List any problems you encountered that were brand new to you in this email and how you overcame them.

## Important

We're going to use this project tomorrow to teach you how to add user authentication to your application. This way, a user must be logged in to see their shopping list. Only when they are logged in can they access it!

## 5.6 Securing Your Application

### Security 101

#### Objectives:

- Describe the types of authentication
  - Something you have
  - Something you are
  - Something you know
- Describe HTTP (port 80) vs HTTPS (port 443)
- Describe security system types
  - Open access
  - API Key
  - User & Role Based Authentication
- Understand how to secure a password
- Describe what a **session** is and how it can be used
- Use BCrypt to generate salt and hashes

### Basic User Authentication

#### Objectives

### Lab: Secure your Shopping List Application

You're going to work on your shopping list application to secure it this afternoon.

## 5.6 Server-side Authentication

### Agenda

- Quiz
- So we have these secure resources
- How can we lock them safely?
- Public vs API Key vs Users/Roles
- Today's Goals
  - Understand how to make logging in enjoyable
  - Implement UserModels and Account Registration
  - Gatekeeping Resources

### QUIZ

1. What is the primary difference between the HTTP Verbs `get` and `post` ?
2. What is a **Postback**?
3. What is the primary difference between an Ajax request and a Postback?
4. If MVC were smores, models would be the graham cracker and the views would be chocolate. What is the controller?
5. HTML Input Elements have a `name` attribute that binds to what hash in Ruby when passed to the server as request?

### So we have these secure resources

Imagine that you work for a bank. You can't let customer's balance information run wild, could you? No.

- We should lock them up!
- Keep them under wraps.
- We don't want to get sued, do we?

### How can we lock them safely?

There are many ways to lock resources away from the general public. A few of these patterns are:

- Public APIs - hey, just let the data be free *man*\*!

- Requiring API keys - maybe we want people to have access to resources but want to throttle their usage.
- Users accounts and various roles - great for complex applications that have different levels of use.

## Public Resources

- Open source
- Share as much as you want
- Usually run on donations or sponsorship
- Usually projects of love and happiness
- Small, dedicated teams

## API Keys

- Web servers can be expensive
- We strive to share our data
- Maybe for free? OMDB, OpenWeatherMap, City of Chicago
- Maybe profit? Companies want to charge for their proprietary data
- Either way, API keys are great for requiring some level of authorization

## Users and Role Infrastructure

- We have a lot of roles
- Maybe some roles do different jobs
- Our computers do this, right?
- Maybe we want to let admins curate content
- But allows registered users to post content
- And to hide content until someone signs up?
- User accounts and roles allow for us to handle these situations

## Today's Goals

- Understand how to make logging in enjoyable
- Implement UserModels and Account Registration
- Gatekeeping Resources
  - Require Registration to access resources
  - Legal reasons
  - Competitive Reasons

- HIPAA and other data retention laws
- HTTPS warning!
- Implementing `is_authorized` as a **filter**

## Understand how to make logging in enjoyable

- Or at least, not frustration/annoying
- User Story: Registration and Logging In
- Mock out our Views
- Create our Views in ERB

## Implement UserModels and Account Registration

- We'll need a new table in our database
- And a new Model!
- And learn how to securely work with passwords
- HTTPS warning!
- Build controllers to bind the model to the view

## Gatekeeping Resources

- Require Registration to access resources
- Legal reasons
- Competitive Reasons
- HIPAA and other data retention laws
- HTTPS warning!
- Implementing `is_authorized` as a **filter**

## 5.6 Building an Account Model

### Video (47 minutes long!)

- <https://youtu.be/1G0YalgHK84>

### Database Migration

```
CREATE DATABASE bee_crypt_bzz;
\c bee_crypt_bzz
CREATE TABLE accounts (id SERIAL PRIMARY KEY, user_name VARCHAR(255), user_email VARCHAR(255));
\dt
```

### Account Model

```
class Account < ActiveRecord::Base

 include BCrypt #bzzzzzzz

 # setter for password
 # define password = pwd (arg)
 # Account.password = 'meowington42hooloovooBlue@u'
 def password=(pwd)
 # set the password_digest column
 # to BCrypt's Password.create method
 # using the user's input of `pwd`
 self.password_digest = BCrypt::Password.create(pwd)
 end

 # getter for password
 # define method to return password
 # Account.password
 def password
 BCrypt::Password.new(self.password_digest)
 end

 # create a method to test if we are allowed authorization
 # so we need to authenticate
 # we log in with a user_name & password..
 # this method handles all that on the backend!
 # awww yissss
 # Usage: Account.authenticate('james', '42hooloovoo4U')
 # Usage: Account.authenticate(params[:username], params[:password])
 def self.authenticate(user_name, password)
 # search for user
 # Account model.find_by column name with value to search
 current_user = Account.find_by(user_name: user_name)
 # return our current user IF passwords match
 if (current_user.password == password)
 return current_user
 else
 return nil
 end
 end

end
```

## 5.6 Tux!

Tux is a console environment for Sinatra. You can install it using:

- `gem install tux`

You also should include `pry` in your Gemfile! Then, navigate to the directory of your application (where your `config.ru` lives). Run:

- `tux`

You will now be in a terminal environment of your server. This is similar to using `pry` but you get direct console for your application.

To exit, just enter `exit`.

## 5.6 Account Controller

### Video (57 minutes long!)

- <https://www.youtube.com/watch?v=G1PvSl2CoL4>

### Source Code

- In Class: [https://github.com/ga-chicago/secure\\_my\\_splat](https://github.com/ga-chicago/secure_my_splat)
- Reference with Controllers: [https://github.com/code-for-coffee/not\\_broken\\_bcrypt](https://github.com/code-for-coffee/not_broken_bcrypt)

### Controller Code

```
require 'bundler' # requiring the bundler
Bundler.require # bundle our !$#
establishing connection to postgresql db
ActiveRecord::Base.establish_connection(
 :database => 'bee_crypt_bzz',
 :adapter => 'postgresql'
)

this will go in ApplicationController
enable :sessions # so easy a 5year maybe could do it

helper method to see if username exists!
does_user_exist(params[:user_name])
def does_user_exist(username)
 user = Account.find_by(:user_name => username)
 if user
 return true
 else
 return false
 end
end

does our user have access to something?
def authorization_check
 if session[:current_user] == nil
 redirect '/not_authorized'
 else
 return true
 end
end
```

```

basic template routes
get '/' do
 # for any resource i want to protect...
 # i perform an authorization_check
 authorization_check
 @user_name = session[:current_user].user_name
 # return some resource
 #return { :hello => 'world' }.to_json
 erb :index
end

get '/not_authorized' do
 erb :not_authorized
end

registration for user
get '/register' do
 erb :register
end
post '/register' do
 # check if the user someone is trying to register exists or NOT
 if does_user_exist(params[:user_name]) == true
 return { :message => 'womp, womp... user exists' }.to_json
 # return erb :view_name
 end

 # if we make this far the user does not exist
 # let's make it!
 user = Account.create(user_email: params[:user_email], user_name: params[:user_name]

 p user

 #session is a hash!
 session[:current_user] = user
 # session[:sam_is_cool] = 'fuck yeah he is'
 # session[:roger_says_hi] = 'HI ROGER'

 redirect '/' # instead of calling a view to render...
 # i want to redirect to a route
end

login
get '/login' do
 erb :login
end
post '/login' do
 user = Account.authenticate(params[:user_name], params[:password])
 if user
 session[:current_user] = user
 redirect '/'
 else
 @message = 'Your password or account is incorrect'
 end
end

```

```
erb :login
end

logout
get '/logout' do
 authorization_check
 session[:current_user] = nil
 redirect '/'
end
```

## 5.7 Sinatra's Second Pass

*We spent a lot of time covering some of the finer details of Sinatra. Here are the notes from today's lectures.*

### Source Code

- [https://github.com/ga-chicago/parmesan\\_friday](https://github.com/ga-chicago/parmesan_friday)

### Videos

- Sinatra Routing & Mapping Controllers
- Ruby Classes, Revisited
- Active Record Relationships - Source code:  
[https://github.com/alcastaneda/active\\_record\\_lesson](https://github.com/alcastaneda/active_record_lesson)

## 5.7 Morning Exercise with Divvy

### Source

```
// Make a Constructor of Pets With three properties and two methods

function Pets(name, sound, age){
 this.name = name;
 this.sound = sound;
 this.age = age;
 this.talk = function(){
 console.log("This animal goes " + this.sound);
 };
 this.sing = function(){
 console.log("I was running down the road one day, and someone hit a POSSUM")
 }
}

var franklin = new Pets('Franklin', "Woof Woof", 2);
Franklin.name;
Franklin.talk();
Franklin.sing();

var haileyAnn = new Pets('Baby girl', "ahhh Woooooooo", 14);
haileyAnn.name;
haileyAnn.talk();
haileyAnn.sing();

// Instantiate two instances of Pets and access the properties and methods in dev tool

Loop over an array of five colors and print out each item to the console along with a
var colors = ["blue", "purple", "orange", "red", "yellow"];

for (var i=0; i < colors.length; i++){
 console.log("The color is " +colors[i]);
}

// Maybe write a function now that takes in any array and prints out a message for each

Create a function that returns a random integer between one and twenty.
function randomNum(){
 return Math.floor(Math.random() *20 +1);
}

// maybe create a function that takes in two arguments and returns a value between them

// Create a funtion that returns a random item from an array
```

```
var seinfeld = ["george", "jerry", "Cosmo", "Elaine"];
function item (arr){
 return seinfeld[Math.floor(Math.random()* arr.length)];
}

// Api Call

$(document).ready(function(){
 $.ajax(getDivy);

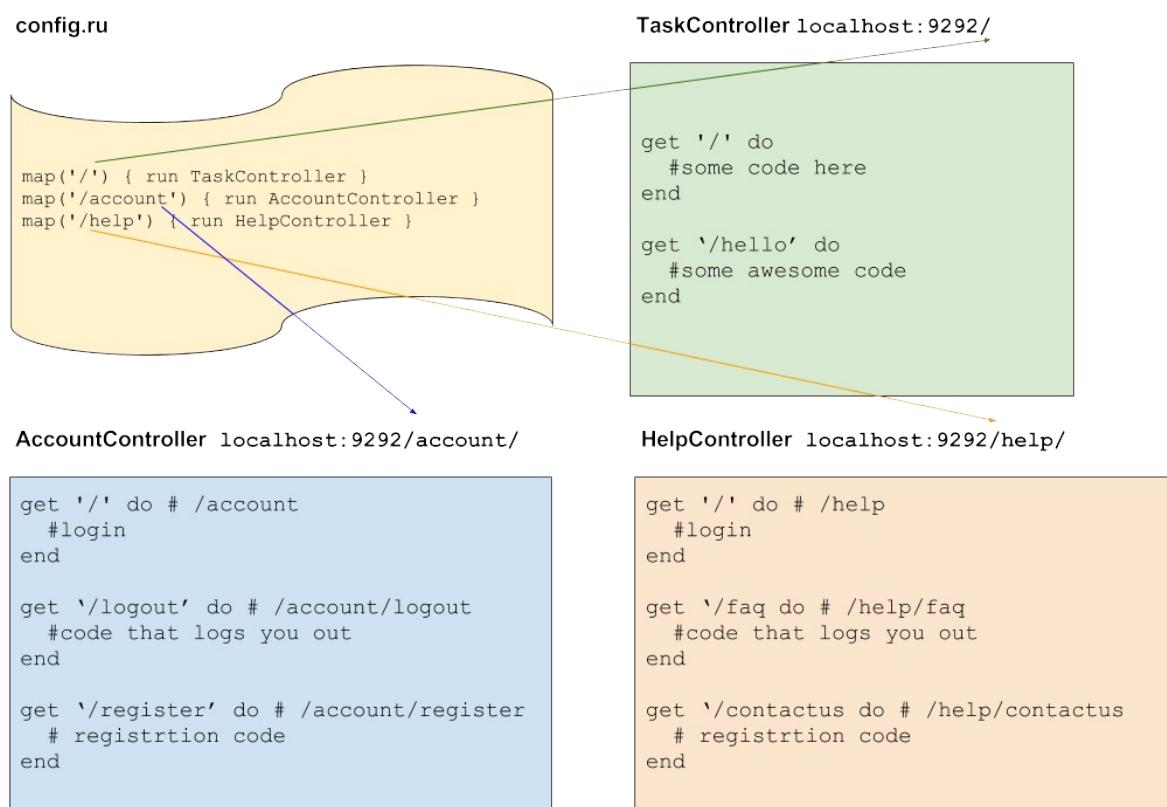
});

var getDivy = {
 type: 'get',
 url: 'https://data.cityofchicago.org/resource/bk89-9dk7.json',
 dataType: 'json',
 success: function(data){
 console.log(data);
 console.log(data[0].station_name)
 for (var i =0; i < 30; i++){
 $('body').append("The divy bikes at the location of "+ data[i].station_name + "");
 }
 }, error: function() {
 console.log("It didn't work big dumb idiot")
 }
};

});
```

## 5.7 Mapping Routes to Controllers

### Diagram



### Example Config

`config.ru`

```
require 'sinatra/base'

require('./controllers/application') # /controllers/application.rb
require('./controllers/home') # /controllers/home.rb
require('./controllers/task') # /controllers/task.rb
require('./controllers/account') # /controllers/account.rb
require('./controllers/api') # /controllers/api.rb

map resource to a controller
http://localhost:9292/ <-- ROOT RESOURCE
http://localhost:9292/task <-- some other resource
http://localhost:9292/account <-- another resource

map('/') { run HomeController } # browse to localhost/
get '/' do ... end -> localhost/
get '/meow' do ... end -> localhost/meow

map('/task') { run TaskController } # browse to localhost/task
get '/' do ... end -> localhost/task
get '/meow' do ... end -> localhost/task/meow
get '/create' do ... end -> localhost/task/create

map('/account') { run AccountController } # browse to localhost/account
get '/' do ... end -> localhost/account
get '/meow' do ... end -> localhost/account/meow
get '/create' do ... end -> localhost/account/create
get '/register' do ... end -> localhost/account/register
get '/task/account' do ... end ->

map('/api') { run ApiController } # browse to localhost/api
get '/' do ... end -> localhost/api
get '/create' do ... end -> localhost/api/create
get '/update' do ... end -> localhost/api/update
get '/destroy' do ... end -> localhost/api/destroy
```

# 5.7 Classes, Revisited

## Getting and Setting

```
Classes - getting and setting
class User
 attr_accessor :name, :email, :password
initialize optional
 def initialize(name, email, password, array)
 @name = name
 @email = email
 @password = password
 @array = array
 end
WAT
 def change_array
 @a = @array
 @b = @array
 p @a.each{|word| p word.upcase!}
 @a += ["anything"]
 p @b
 p @a
 p @array
 end
end
adriana= User.new("adriana", "adri@gmail.com", "password123", ["hello"])
p adriana.name
p adriana.name= "adrian"
adriana.change_array
```

## Exercise

1. Create a class for pea plants
2. You have 2 instance variables:
  - o number\_of\_leaves
  - o flower\_colour
3. Create three different generations using classes
4. The first plant should have 4 leaves and white flowers
5. The second plant should have 5 leaves and pink flowers
6. The third plant should have 2 leaves and orange-mutated flowers
7. Instantiate each and call their `to_s` method.
8. Use `attr` helpers to change the `Ancestor`'s `@eye_gene`.

9. On each of your instantiated classes, call the `to_s` method again.
10. Describe the effect you see here.

## Example

```
class Ancestor

 def initialize
 @eye_gene = 'blue eyes'
 @hair_gene = 'black hair'
 @name = 'Eldin Webber'
 end

 def to_s
 return 'My name is ' + @name + ' and I have ' + @eye_gene + ' and ' + @hair_gene
 end

 def viking_slayer
 return 'I AM CONQUEROR!!! ROARRRR!!!!21212'
 end

end

class You < Ancestor
 def draw_things
 return 'drawing the coolest shit'
 end
 # def initialize
 # end
 def birth
 @name = 'Marty'
 end
end

class Jamesette < You

 def birth
 @name = 'Jamesette'
 @hair_gene = 'red hair'
 end

end

eldin = Ancestor.new
p eldin.to_s

marty = You.new
marty.birth
```

```
p marty.to_s

daughter = Jamesette.new
daughter.birth
p daughter.to_s
p daughter.viking_slayer
```

## 5.7 Class in Java

*Just for fun, here is what a class looks like in another programming language: Java.*

```
public class Person {
 Integer age = 0;
 String ToString() {
 String ret = "I am " + this.age + " years old";
 return ret;
 }
}

Person you = new Person();
you.ToString();
// what happens?
```

## 5.7 JSON Reader

As requested, here is our trusty **JSONReader!**

```
class JSONReader

 def initialize(filename)
 @json = String.new
 File.foreach(filename) do |line|
 @json = @json + line
 end
 #binding.pry
 end

 def to_hash
 return JSON.parse(@json)
 end

end
```

## Usage

```
reader = JSONReader.new('data.json') # data.json in same dir as this File
model = reader.to_hash
```

## 5.7 BCrypt Example

- The following source code may be found at [https://github.com/ga-chicago/sinatra\\_cru](https://github.com/ga-chicago/sinatra_cru) on the **ga-chicago** organization. This demonstrates actual use in a Sinatra application.

```
how does bcrypt work under the surface?

require 'bcrypt'
our password is turkey dinner
it needs some salt to give flavour
salt = BCrypt::Engine.generate_salt
p salt

pwd = 'hooloovoo42'
bcrypt combines salt + password
combined_password = pwd + salt
p combined_password

we encrypt our password as a hash
hash_secret does the above!
hash_secret makes our combined password
and generates a hash for us
hash = BCrypt::Engine.hash_secret(pwd, salt)
p hash

#login
first off, you need the user's salt
save it to db, cookie, etc
may need to have a way to reset
usr passes in their password

users_entry = params[:password]
compare hash in db
compare results
look in postm /login: https://github.com/ga-chicago/sinatra_crud/blob/master/contro.
```

## 5.7 Application Controller Architecture

```
class ApplicationController < ActiveRecord::Base

ApplicationController is a configuration controller
not_found - 404 so all child controllers have it
sessions! all controllers need access to sessions
public! Setting your public folder so all controllers have access
views! direct our views somewhere
auth checks! all the controllers should have access!
database connections!
cookies!
anything else all controllers should have access to
goes in ApplicationController

do not override def initialize in any controller

end
```

## Basic Application Controller & Inheritance Example

```
class ApplicationController < Sinatra::Base

 enable :sessions
 ActiveRecord::Base.establish_connection(
 :database => 'martys_life',
 :adapter => 'mongodb'
)
 def self_check
 return 'you better check yoself before you wreck yoself'
 end
 not_found do

 end
 get '/' do
 return { :message => 'marty party' }.to_json
 end

 end

 class AccountController < ApplicationController

 # this.self_check()
 self.self_check

 get '/' do

 end
 end

 class FlightController < ApplicationController

 end
```

## 5.7 Videos: Production Servers on Digital Ocean with Unicorn & Nginx

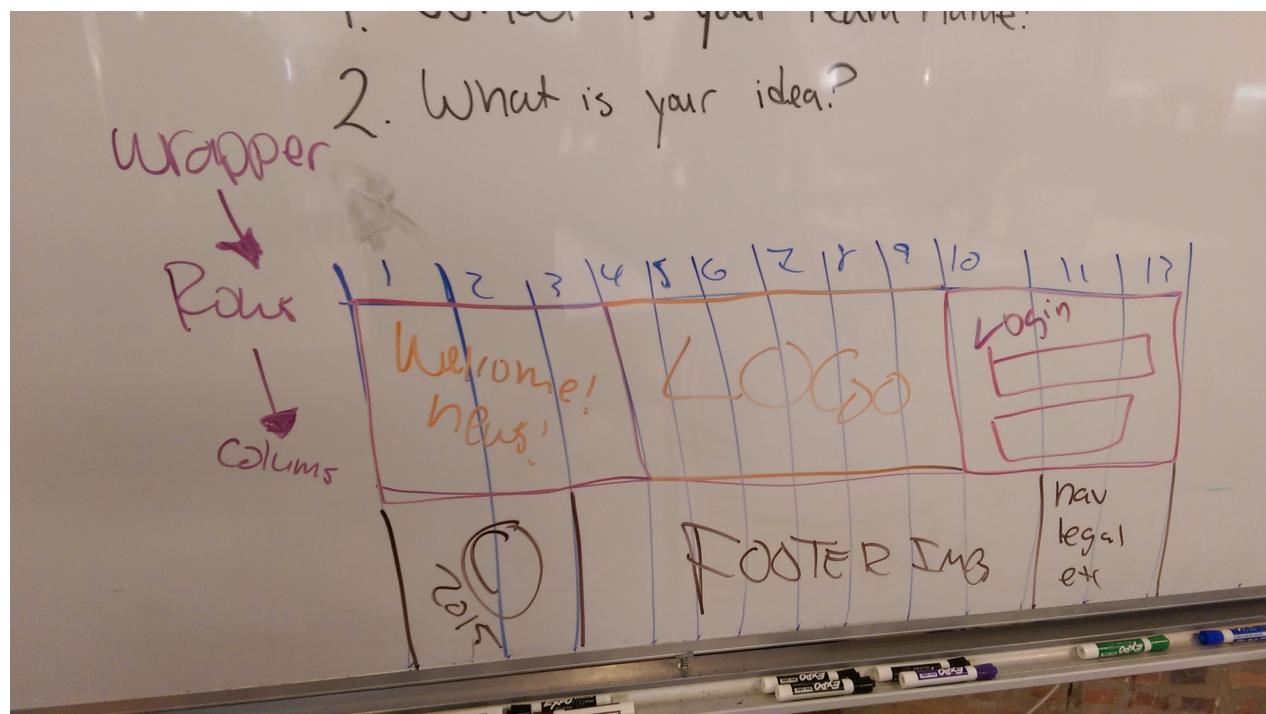
### Digital Ocean with Unicorn & Nginx

- *Part One - Provisioning Droplet:* [https://www.youtube.com/watch?v=M6U8\\_VN8eFE](https://www.youtube.com/watch?v=M6U8_VN8eFE)
- *Part Two - postgresql:* <https://www.youtube.com/watch?v=YpLdDvL0dWw>
- *Part Three - Connecting to Github:* <https://www.youtube.com/watch?v=Ft3iLLX-3nc>
- *Part Four - Nginx & Unicorn:* [https://www.youtube.com/watch?v=YHqO\\_8Mps-c](https://www.youtube.com/watch?v=YHqO_8Mps-c)

# Bootstrap 101 Self-Practice

- Get Bootstrap
- Grid Example Template

## Grid/Columns



## Individual Practice: Diagram

If you would like to visualize how Bootstrap works, you may do so.

- On Graph paper, create a `12 x 6` grid with a landscape orientation
- Place `header`, `nav`, `article`, `aside`, `footer` tags and more
- Understand the class naming system of Bootstrap for columns
- `col-md-x` where `x` is a value of `1` through `12` for the grid.
- 12 grids are placed inside of a row that live inside of `container`s or `wrapper`s.

## Boilerplate

```
<section class='section'>
 <article class='row'>
 <div class='col-md-3'>3</div>
 <div class='col-md-6'>6</div>
 <div class='col-md-3'>3</div>
 </article>
</section>
```

**Columns should add up to 12!**

# Relationships using ActiveRecord

Let's have some colourful fun!

- `gem install rainbow`
- `require 'rainbow'`

## Building the SQL

### 1:Many

```
CREATE TABLE cows (id SERIAL PRIMARY KEY, farmer_id INTEGER, name VARCHAR(255));
CREATE TABLE farmers (id SERIAL PRIMARY KEY, name VARCHAR(255));
```

### Many:Many

```
CREATE TABLE travelers (id SERIAL PRIMARY KEY, name VARCHAR(255));
CREATE TABLE vacation_spots (id SERIAL PRIMARY KEY, city VARCHAR(255));
CREATE TABLE visits (id SERIAL PRIMARY KEY, vacation_spot_id INTEGER, traveler_id INTEGER);
```

## Many to Many Models

We're going to create two models. They will have a many to many relationship.

*Models*

```

MANY
class VacationSpot < ActiveRecord::Base
 has_many :visits
 has_many :travelers, :through => :visits
 def to_s
 self.city + '... The greatest City in the World!'
 end
end

MANY
class Traveler < ActiveRecord::Base
 has_many :visits
 has_many :vacation_spots, :through => :visits
 def to_s
 self.name + '... seeing the world!'
 end
end

JOIN
class Visit < ActiveRecord::Base
 belongs_to :traveler
 belongs_to :vacation_spot
 def to_s
 self.traveler.to_s + " - visits - " + self.vacation_spot.to_s
 end
end

```

## Let's See Our Output

```

---- Examples ----
puts Rainbow("\n***** What happens in the city stays in the city... cause we have
puts Rainbow("*****")
puts Rainbow("*****")
puts Rainbow("*****")

puts Rainbow("\nCreate A Travler:").green
puts Rainbow("lichard = Traveler.create({name: 'Lichard DeGray'})").magenta
lichard = Traveler.create({name: 'Lichard DeGray'})

puts Rainbow("\nCreate A Vacation Spot:").green
puts Rainbow("alp_lake = VacationSpot.create({city: 'Aplville'})").magenta
alp_lake = VacationSpot.create({city: 'Aplville'})

puts Rainbow("\nCreate a visit joining a travler and a vacation spot").green
puts Rainbow("weekend_get_away = Visit.create({traveler: lichard, vacation_spot: alp_
weekend_get_away = Visit.create({traveler: lichard, vacation_spot: alp_lake})")

puts Rainbow("\nTravler:").green
puts Rainbow("puts lichard").magenta

```

```
puts lichard

puts Rainbow("\nVaction Spot:").green
puts Rainbow("puts alp_lake").magenta
puts alp_lake

puts Rainbow("\nVisit:").green
puts Rainbow("puts weekend_get_away").magenta
puts weekend_get_away

puts Rainbow("\nTravler's Visits:").green
puts Rainbow("puts lichard.visits").magenta
puts lichard.visits

puts Rainbow("\nTravler's Vacation Spots:").green
puts Rainbow("puts lichard.vacation_spots").magenta
puts lichard.vacation_spots

puts Rainbow("\nVacation Spot's Vists:").green
puts Rainbow("puts alp_lake.visits").magenta
puts alp_lake.visits

puts Rainbow("\nVacation Spot's Travelers:").green
puts Rainbow("puts alp_lake.travelers").magenta
puts alp_lake.travelers

puts Rainbow("\nVisit's Traveler:").green
puts Rainbow("puts weekend_get_away.traveler").magenta
puts weekend_get_away.traveler

puts Rainbow("\nVisit's Vacation Spot:").green
puts Rainbow("puts weekend_get_away.vacation_spot").magenta
puts weekend_get_away.vacation_spot

puts ''
puts Rainbow("***** Be sure to visit us again soon *****").blue
puts Rainbow("*****").red
puts Rainbow("*****").blue
puts ''
```

## 1:Many Models

```
** ONE-to-MANY **

ONE
class Farmer < ActiveRecord::Base
 has_many :cows
 def to_s
 "...and ' + self.name + ' was his/her name... oh"
 end
end

MANY
class Cow < ActiveRecord::Base
 belongs_to :farmer
 def to_s
 self.name + '.... moooooo'
 end
end
```

## Let's View Our Code, Again

```
puts Rainbow("\nCreate A Cow:").green
puts Rainbow("mooooonalisa = Cow.create({name: 'lisa'})").magenta
mooooonalisa = Cow.create({name: 'lisa'})

puts Rainbow("\nCreate A Farmer:").green
puts Rainbow("lichard = Farmer.create({name: 'Lichard DeGray'})").magenta
lichard = Farmer.create({name: 'Lichard DeGray'})

puts Rainbow("\nAssociate the farmer and the cow:").green
puts Rainbow("lichard.cows << mooooonalisa").magenta
lichard.cows << mooooonalisa

puts Rainbow("\nFarmer").green
puts Rainbow("puts lichard").magenta
puts lichard

puts Rainbow("\nFarmer's cows").green
puts Rainbow("puts lichard.cows").magenta
puts lichard.cows
```

## Further Reading

- [ActiveRecord Association Basics](#)



## Sending Emails with Mandrill

Mandrill is an API that allows us to send emails to anyone for free. They are sent out in bulk transactions. This can result in a 10 minute delay for emails. However, it is **free**.

Example source code is available here: [https://github.com/ga-chicago/ruby\\_email\\_server](https://github.com/ga-chicago/ruby_email_server)



# Project #2: Building Your First Full-stack Application

## Overview

This second project is your first foray into **building a full-stack application**. You'll be **building a Sinatra app**, which means you'll learn about what it takes to build a functional application from the ground up yourself.

**This is exciting!** It's a lot, but we'll give you the tools over the next few weeks to be able build what you need, and you get to decide what you do with it. And you get to be creative in choosing what sort of application you want to build!

**You will be working in pairs for this project**, and you'll be designing the app together. We hope you'll exercise creativity on this project, sketch some wireframes before you start, and write user stories to define what your users will want to do with the app. Make sure you have time to run these ideas by your instructors to get their feedback before you dive too deep into code! Remember to keep things small and focus on mastering the fundamentals – scope creep/feature creep is the biggest pitfall for any project!

---

## Technical Requirements

Your app must:

- **Have at least 2 models** (more if they make sense!) – one representing someone using your application, and one that represents the main functional idea for your app
  - **Include sign up/log in functionality**, with encrypted passwords & an authorization flow
  - **Utilize an ORM to create a database table structure** and interact with your relationally-stored data
  - **Include wireframes** that you designed during the planning process
  - **Have semantically clean HTML and CSS**
  - **Be deployed online** and accessible to the public
-

## Necessary Deliverables

- A **working full-stack application, built by you**, hosted somewhere on the internet
  - A **link to your hosted working app** in the URL section of your Github repo
  - A **git repository hosted on Github**, with a link to your hosted project, and frequent commits dating back to the **very beginning** of the project. Commit early, commit often.
  - A **readme.md file** with explanations of the technologies used, the approach taken, installation instructions, unsolved problems, etc.
  - **Wireframes of your app**, hosted somewhere & linked in your readme
  - A link in your **readme.md** to the publically-accessible **user stories you created**
- 

## Suggested Ways to Get Started

- **Begin with the end in mind.** Know where you want to go by planning with wireframes & user stories, so you don't waste time building things you don't need. Keep it lean and keep it elegant.
  - **Don't hesitate to write throwaway code to solve short term problems**
  - **Read the docs for whatever technologies you use.** Most of the time, there is a tutorial that you can follow, but not always, and learning to read documentation is crucial to your success as a developer
  - **Commit early, commit often.** Don't be afraid to break something because you can always go back in time to a previous version.
  - **User stories define what a specific type of user wants to accomplish with your application.** It's tempting to just make them *todo lists* for what needs to get done, but if you keep them small & focused on what a user cares about from their perspective, it'll help you know what to build
  - **Write pseudocode before you write actual code.** Thinking through the logic of something helps.
- 

## Potential Project Ideas

### Cheerups

The world is a depressing place.

Your task is to create an app that will allow people to create and share "cheerups" - happy little quips to brighten other peoples' days. Cheerups will be small - limited to 139 characters. Members will be able to promote Cheerups that they like and maybe even boost the reputation of the Cheerupper.

### **Bookmarket**

You will create an application where users can bookmark links they want to keep.

But what if users could trade bookmarks for other bookmarks? Or sell bookmarks for points? Or send bookmarks to your friends. Or something even crazier.

### **Photo sharing app**

Users will be able to register and create albums and photos. Albums and photos will need to be named and described by their owners. Users will be able to view other users' albums. Maybe users can comment on photos, or either up/down vote them.

---

## **Useful Resources**

- [Heroku \(for hosting your back-end\)](#)
  - [Using a SQL Database with Heroku for database support](#)
  - [Writing Good User Stories \(for a few user story tips\)](#)
  - [Presenting Information Architecture \(for more insight into wireframing\)](#)
- 

## **Project Feedback + Evaluation**

- **Project Workflow:** Did you complete the user stories, wireframes, task tracking, and/or ERDs, as specified above? Did you use source control as expected for the phase of the program you're in (detailed above)?
- **Technical Requirements:** Did you deliver a project that met all the technical requirements? Given what the class has covered so far, did you build something that was reasonably complex?
- **Creativity:** Did you add a personal spin or creative element into your project submission? Did you deliver something of value to the end user (not just a login button and an index page)?

- **Code Quality:** Did you follow code style guidance and best practices covered in class, such as spacing, modularity, and semantic naming? Did you comment your code as your instructors as we have in class?
- **Deployment and Functionality:** Is your application deployed and functional at a public URL? Is your application free of errors and incomplete functionality?
- **Total:** Your instructors will give you a total score on your project between:

Score | Expectations ----- | ----- **0** | *Incomplete. 1 | Does not meet expectations.*

**2** | *Meets expectations, good job!* **3** | *Exceeds expectations, you wonderful creature, you!*

This will serve as a helpful overall gauge of whether you met the project goals, but **the more important scores are the individual ones** above, which can help you identify where to focus your efforts for the next project!

# Project Week Schedule

	Monday - 12/7	Tuesday - 12/8	Wednesday - 12/9	Thursday - 12/10	Friday - 12/11
9am	9 - 9:40am: Project Kickoff	9am - Standup	9am - Standup	9am - Standup	9am - Standup
10am	9:40am - 9:50: Standup				
11am	9:50am -10: Break				
12pm	10am - 11:15am Project Pitches & Approval	10am - 10:45am Lab: Hiding Keys w/DOTENV	10am - 11:15am Lab: Security Revisited		
1pm	11:15am - 11:30: Break				11:30am - 12:30pm Lunch
2pm	12:45pm - 2pm Lunch	12:45pm - 2pm Lunch	12:45pm - 2pm Lunch	12:45pm - 2pm Lunch	12:30pm - 2:45pm Project Presentations
3pm	2pm - 3:15pm Lab: Rake Tasks				
4pm	3:30pm - 4:45pm Lab: Models & Relationships		3:30pm - 4:30pm Outcomes w/Amy	3pm to 5pm Hang out with Razorfish	Weekend Starts!
5pm					
6pm	5-6pm Support: Jim	5-6pm Support: Adriana	5-6pm Support: James	5-6pm Support: Adriana	Completed Application
	Wireframes/User Stories	ERDs/Database/Models	MVC Application	Polished Front End	

# Project Teams

- **Eventski:** Sam Vredenburgh & Kevin Deutscher - *Ski Resort + Concerts with Songkick API* - <https://github.com/Eventski>
- **Silent Spies:** David Beeler & Jan Christian Bernabe - *Thoughtful with Image API* - <https://github.com/SilentSpies>
- **LiLa:** Lidia Santos & Lamthong Keophalychanh - *Social Accountability Bucket* - <https://github.com/LiLaChicago>
- **Team Roger Panella:** Sam Groesch & Anna Sherman - *Travel Map App* - <https://github.com/rogerpanella>
- **Senate Orphan:** Stephen Delaney & Aaron Krueger - *RunningM8* - <https://github.com/senateOrphans/runningM8>
- **TBD:** Simran Khosla & Daniel Tabion - *Dude Where's My Car?* - <https://github.com/GA-TBD>
- **Me Grognak!:** Jeff Steed & Jason Bratt - *Split Screen* - <https://github.com/Me-Grognak>
- **Nucleus:** Paul Boyle & Nick Espinosa - *Retail Inventory* - <https://github.com/Nuc1eus>
- **Developer Tools:** Martin Ryan & Jason Tham - *Historical Fantasy Baseball* - ???
- **Syntax Samurais:** Andrew Dushane & Jen Kahn - *Simply News* - <https://github.com/TeamSyntaxSamurais/simply-news>
- **Programming Philosophers:** Ezra Chang & Adam Wilson - *Album Memories* - <https://github.com/programmingphilosophers>
- **You got served:** Roger Panella & Katie Ude - *Travel Diary* - <https://github.com/YouGotServedChicago>
- **MiniPig:** Ruth Thatcher & Paul Vasich - *Group Decisions* - ???



## Project Scope

*When building a project scope, you need to ask yourself...*

1. What type of website will I be building? (Is this going to be a landing page? A blog? An e-commerce site?)
2. When do I need this website completed?
3. What is my budget for this project?
4. What is the title of my project?
5. Who is the intended audience of my project?
6. Who will I deliver the final project to?



## Initiation Stakeholder Survey

*Below are sample questions that you should ask your focus group/stakeholders...*

1. What do you expect out of the final result of our project?
2. What type of devices will you be using to access the final project?
3. What operating system(s) will you be using to access the final project?
4. What is your preferred web browser?
5. What is your secondary web browser?
6. What version of your current operating system?
7. Will the project need to be **ADA Compliant** for those with disabilities?  
(<http://www.ada.gov/pcatoolkit/chap5toolkit.htm>)
8. Will the project need to be **Cookie Law** compliant for citizens in the UK/EU?  
(<http://www.cookie-law.org/faq/>)
9. Is any content on the website sensitive (such as personal identifiable information or payment information)?
10. What features would you like to see implemented on the website?
11. If there is a current website, what feature(s) do you think works well?
12. If there is a current website, what feature(s) do you think do not convey their intended use or perform them well?



## Project Scope Survey

1. Which questions should this survey answer?
2. How will you organize yourselves within your team for the project to succeed?
3. Who is to be worked with and when? How can and will we communicate during the project? Will we use email? Slack? Something else?
4. What restrictions are imposed due to the budget of the project?
5. Who are your focus groups (potential *users* and *stakeholders*)?
6. What is the *minimal viable product* due date?
7. How much time do we have for the project until completion? What is the hard deadline?
8. What practical challenges are we already aware of (e.g. language, availability, hosting, design resources)?
9. Which ownership challenges do we have to resolve (e.g. ownership of the result of the project)?

## The MEAN Stack

Week 10

~~javascript as a Server~~



## What is the MEAN Stack?

The *MEAN* stack is a full stack for development similar to Ruby on Rails or Sinatra (+ whatever database you use). It is built upon cutting-edge, wild-wild west crafted software:

- Angular.js (<https://angularjs.org/>)
- Express.js (<http://expressjs.com/>)
- Node.js (<https://nodejs.org/>)
- MongoDB (<https://mongodb.org>)

## Let's compare Ruby on Rails with MEAN

- In *Ruby on Rails* we have been using **Postgres** as our database. We write most of our code in **Ruby**. We rely on **Rails** as our framework to host our application and serve APIs/Views. We use **ERB** and various **Javascript frameworks** like **Backbone** to display our views.
- In the *MEAN* stack, we will be using **MongoDB** as our database. We write most of our code in **Javascript**. We rely on **Node** and **Express.js** to host our application and serve APIs/Views. We use **AngularJS** to render our views that are served by **Express**.
- **Fun Fact:** *Express.js* looks and feels like *Sinatra*. If you liked Sinatra, you'll probably love Express.js because the syntax is very similar.

## How the stack works together



## 7.0 From Ruby to Javascript

Today we're going to transition from Ruby back into Javascript. By the end of the day, you should be ready to tackle full stack Node.js.

### Recap: Ruby 101, Classes, & Methods

- We will review variables, classes, and methods in class then you will go through a rep exercise in class

Reps:

### Recap: Sinatra

- We are going to make a very simple dummy sinatra app with no backend!

### Recap: Javascript 101

- soon

### Javascript Lab

You will perform the **Javascript Recap, Reps, and Resilience** exercise.

### Homework

For tonight's homework, you should read [Javascript for Cats](#). It is a fun, friendly read. As you read through it, take notes of concepts that are still hazy for you and bring them up tomorrow.

You should also read [this chapter on HTTP](#) in Eloquent Javascript.

# Introduction to Test Driven Development

## Learning Objectives

- Understand why we should test our code with specs
- Describe what assertions and expectations are
- Manually write a test for a DOM element

## We ASSERT something that should match an EXPECTED value

- Asserts are what our actual value we get.
- They are what fails a test.
- We *EXPECT* that an *ASSERT* matches our *EXPECTED* value
- Let's visualize this... together.

## Testing Code

```
/*
 * Verifies that an image has a src attribute.
 * @test
 */
function doesImgHaveSrcTest(imgObj) {
 var assert = imgObj;
 var expected = true;
 if (assert.hasOwnProperty('src') == expected) {
 return true;
 }
 return false;
}
```

## Refactoring Tests: Before

*We want to break our tests down to where we compare our assert and our expectation.*  
Here are a few tests.

## Testing Pi

```
/**
 * Verifies that pi = '3.14'
 * @test
 */
function verifyPiTest(pi) {
 var assert = pi;
 var expected = '3.14';
 if (assert.toString().substring(0,4) == expected) {
 return true;
 }
 return false;
}
```

## Refactoring Tests: After

*You can see our tests and how they now just compare our assert and our expectation.*

```
/**
 * defines a test
 * @test
 */
function doesBodyExistTest(dom) {
 var assert = dom.getElementsByTagName('body');
 var expect = 1;

 if (assert.length == expect) {
 return true;
 }
 return false;
}

/**
 * Verifies that an image has a src attribute.
 * @test
 */
function doesImgHaveSrcTest(imgObj) {
 var assert = imgObj.hasOwnProperty('src');
 var expected = true;
 if (assert == expected) {
 return true;
 }
 return false;
}

/**
 * Verifies that pi = '3.14'
 * @test
 */
function verifyPiTest(pi) {
 var assert = pi.toString().substring(0,4);
 var expected = '3.14';
 if (assert == expected) {
 return true;
 }
 return false;
}
```

## Your Turn

Write a test to verify that a `<div>` has content (such as ... `innerHTML`...).

```
/**
 * Verify that a div has content
 * @test
 */
function doesDivHaveContentTest(divObj) {

 var assert;
 var expect;

 if (assert == expect) {
 return true;
 }
 return false;
}
```

## Make an animal class

1. The class should be instantiated with a name, cell type (eukaryotic), and 2 other properties an animal has
2. The class should have 2 methods of your chose that pertains to the class

## Make a cat class

1. the cat class should inherit from the animal class
2. the cat class should have 2 methods specific to a cat

## Make a whole bunch o catssss

1. make an array of cat names called cat\_names
2. go through the array and make a new cat with each name

## Make a simple sinatra template

Gemfile

```
source 'https://rubygems.org'

gem 'sinatra'
gem 'json'
```

Config.ru

```
require './app'
run Sinatra::Application
```

App.rb

```
require 'bundler'
Bundler.require()

get '/' do
 "hello world"
end
```

Run the app with bundle exec rackup!

## Make an about me and favorite things

1. make a route called aboutme and one called favorites
2. in your app.rb make an array of hashes that hold things about your favorite things
3. use html and erb to list your favorite things and properties
4. use html and css to make an about me page and style your app
5. include links to your new pages in your root route '/'

## Pair "White boarding"

1. make a function that adds 2 numbers
2. make a blank array and add to that array using a javascript method
3. make a for loop that goes through an array of 2 numbers and logs the number
4. write an if statement
5. write a while loop
6. make a javascript object with 2 properties

# 7.1 Javascript Recap, Reps, and Resilience

## Instructions

You will create an individual file for each of these tasks. Each task will be named as `one.js` , `two.js` , etc. based on the number that corresponds to the task. Please save these in `07_fullstack_node/your_name` and create a pull request when completed. **NOTE** that not **all** of these must be completed. You know what skills you should focus on. We leave hints to help you; you must do your own research.

You may also need to use HTML and/or CSS to complete this task. We just want the Javascript code. Feel free to use tools like **Codepen** and **JSFiddle** and link to them if you *do* use them in a comment.

## 1. Data Types

Create the following variables and in a comment after the variable, state the type that it is. If you are unsure of the type, use the function `typeof` to find it out. Note that some types may not be what you expect...

- The number that represents *pi*.
- Your full name.
- A list of prime numbers.
- A `hash`-like structure to represent the attributes of yourself.
- If you are taking WDI or not.

## 2. Array Manipulation

- Create an array.
- Add 3 places you have visited in your life.
- Remove two of them from the front of the array.
- Add 3 places that you would like to travel to.
- Remove one of them from the end of the array.

## 3. Objects; Objects Everywhere

- Create an Object that represents a laptop.
- Give it 3-5 attributes.

- Give it one ability that tells the world about itself (similar to a `to_s` in Ruby)

## 4. Objects... again

- Using the Object you created for #3....
- Research the Object method that lists all of the `keys` in an Object. Use it to create an array.
- Loop through this array using a `for` (not a `for-in`) loop.
- `console.log` the keys.

## 5. Conditional Trials

- Create a logical block that detects if a user is able to purchase alcohol.
- If so, `console.log` out that they may have a drink.
- Else, warn them via `console.log` to not break the law.

## 6. Loops, Loops, Merry-go-round

- Create an array of your friends.
- Now, create a loop that will `pop` a friend out every time the loop runs.
- Tell a funny story about how they leave each time. `console.log` is your friend here.

## 7. Vanilla Selectors

- Create a *vanilla* Javascript **selector** to select the following:
- `<section id='vanilla'>I really like artificial vanilla flavour</section>`
- Console.log the interior content of this tag.

## 8. jQuery Selectors

- Create a *jQuery* Javascript **selector** to select the following:
- `<article id='strawberry'>I really like strawberry milkshakes from McDonalds!`  
`</article>`
- Console.log the interior content of this tag.

## 9. Functions!

- Create a re-usable function that performs a square root operation on a number.
- You will need at least one argument for your function.

- You will need the `Math` object here as well.

## 10. Story Time!

- Create a function that tells a story.
- It should feel like mad-libs.
- It should accept at least 5 arguments.
- The return of this function should be a story.
- You should use only `string` types.

## 11. Constructors

- Constructors are essentially classes/blueprints for Javascript.
- Create a re-usable constructor for `bands`
- `bands` have members, names, date formed, and albums.
- They also have an `onTour` attribute as well.
- Use this constructor to create three of your favourite bands!

## 12. The Functions Strike Back!

- It is a dark time for the Ruby Rebellion.
- Javascript is on the rise.
- We use Backbone to create models...
- Backbone models can `get()` and `set()` properties (similar to Ruby Classes).
- Create two functions that mimics `model.get` and `model.set` for an object.
- These functions will be `abilities` on an object called `model`.

## 13. AJAXXXXXXXXXXXXXXX

- Use jQuery to create an `$.ajax` request that gets JSON.
- You will get **JSON** from Chicago public data.
- `Console.log` the output.
- Create a `success` and `error` function that log out information.
- You do not need to get very detailed; this is a basic function that just fetches and logs out **all the data**.

## 14. MOAR AJAXXXXXXXXXXXXX

- Create an `$.ajax` request that fetches your favourite movie from OMDB.

- <http://www.omdbapi.com/>
- In the `success` function, you need to `console.log` five attributes about the movie.
- In the `error` function, you need to `console.log` that *'an error has occurred. sorry!'*

## 15. Return of the Functions

- Create a function that draws a rectangle on a canvas element.
- This function should accept arguments for `height`, `width`, and `colour`.
- Return a successfully draw rectangle using Canvas.

## 7.1 Setup your Node Environment

### OS X

```
brew update
brew install node
brew install npm
brew install mongodb
```

### Linux

```
apt-get update -y
apt-get install node -y
apt-get install npm -y
apt-get install mongodb -y
apt-get install nginx -y
apt-get install git -y
```

## 7.2 Intro to the Future (Node.js)

### Introduction to Node

#### Objectives:

- Explain what Node.js is & why it exists
- Compare and contrast Node/Express vs. Ruby/Sinatra/Rails
- Use module.exports and require to organize code
- Understand the asynchronous programming model & the Node event loop

### Creating Servers in Node

#### Objectives:

- Understand how to create servers in node
- Understand how to serve files in node

### Lab: Request/Response Forecast.io

- Fetch data from an API using Node.js
- Use Modules to contain critical logic
- Use module exports to access logic in a module

### Field Trip: Razorfish

We'll be visiting Razorfish this afternoon and are schedule to arrive at **4pm**. We need to leave SPACE no later than 3:30.

Razorfish, 222 W Merchandise Mart Plaza #1200, Chicago, IL 60654, United States

## 7.2 Introduction to Jasmine

- [Official Jasmine.js Documentation](#)
- [Jasmine Boilerplate from Class](#)

### Goals

- Understand why we should test our code with specs
- Describe what assertions and expectations are
- Implement Jasmine with a Node application
- Create a suite of tests and individual tests inside of a suite

### Getting Started with Jasmine

1. Create a new project folder.
2. Change directory to that folder.
3. `npm init` a new application. Enter, enter, enter...
4. Let's take a look at our folder structure: `tree`. Now, add an `app` and a `spec` folder. The `app` folder will contain your application and your `spec` folder will contain your tests.
5. We need to install Jasmine. To do so, `npm install jasmine-node --save`
6. Now, we need to find where our executable for tests lives...  
`node_modules/.bin/jasmine-node spec`
7. Let's run that! `node_modules/.bin/jasmine-node spec`
8. In your `package.json`, under the `"scripts"` for `test`, replace the value with  
`node_modules/.bin/jasmine-node spec`
9. Run `npm test`
10. Verify your tests run.
11. Wait. what tests?! Let's add some! We'll add them to the `spec` folder.
12. Inside of the `spec` folder, create a new Javascript file. Such as `testSpec.js` or `somethingSpec.js`. For tests to successfully be ran, you must end the file with `Spec`.
13. Now, run `npm test` !

### Describe Suites of Tests

- We need to *describe* a suite, or set, of tests
- A suite includes multiple things to test

- Suites can include other suites as children
- Inside of each suite of tests, we include individual tests

```
describe('a set of problems to solve', function() {
 // tests go here...
});
```

## Describe a Test

- Each test is *described*
- *it* has a variety of *expectations*

```
describe('a set of problems to solve', function() {

 describe('a failing test', function() {
 it('fails', function() {
 //expect(false).toBe(true);
 });
 });

 describe('an individual problem to solve', function() {

 it('a basic test', function() {
 var expected = 'solution';
 //var assert = 'womp womp';
 var assert = expected;

 expect(assert).toBe(expected);
 });

 });
});
```

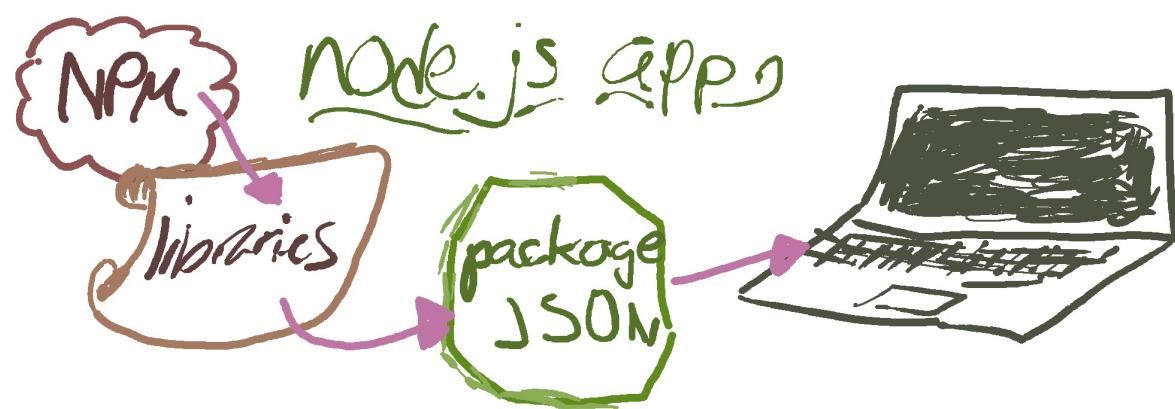
## Jasmine Matchers

```
expect(fn).toThrow(e);
expect(instance).toBe(instance);
expect(mixed).toBeDefined();
expect(mixed).toBeFalsy();
expect(number).toBeGreaterThan(number);
expect(number).toBeLessThan(number);
expect(mixed).toBeNull();
expect(mixed).toBeTruthy();
expect(mixed).toBeUndefined();
expect(array).toContain(member);
expect(string).toContain(substring);
expect(mixed).toEqual(mixed);
expect(mixed).toMatch(pattern);
```

## 7.2 Intro to Node

### Introduction to Node

#### Getting Started



#### 1. package.json

We're going to create a `package.json` file for our new application. This contains information about your app, such as:

- Application's name
- Author
- Git(hub) repository
- Current version
- License And in the future, it will also save any dependancies your application may have.

#### Let's Give it a Try

- To get started, create a new project folder. **Change directory** to that folder.
- Next, create an empty Javascript file to host our application: `touch app.js`
- Then, enter the following command:



- That's right: `npm init`

You'll be prompted through a variety of questions, such as:

```
name: an_api
version: (1.0.0) 0.1.0
description: My first express api
entry point: (index.js) app.js
test command:
git repository: https://github.com/code-for-coffee/express_api
keywords:
author: your-user-name
license: (ISC) MIT
```

This generates the following `package.json` file:

```
{
 "name": "Tasker",
 "version": "0.1.0",
 "description": "The MEAN way to track your Tasks!",
 "main": "app.js",
 "scripts": {
 "test": "echo \\\"Error: no test specified\\\" && exit 1"
 },
 "repository": {
 "type": "git",
 "url": "https://github.com/code-for-coffee/Tasker"
 },
 "author": "code-for-coffee",
 "license": "MIT",
 "bugs": {
 "url": "https://github.com/code-for-coffee/Tasker/issues"
 },
 "homepage": "https://github.com/code-for-coffee/Tasker"
}
```

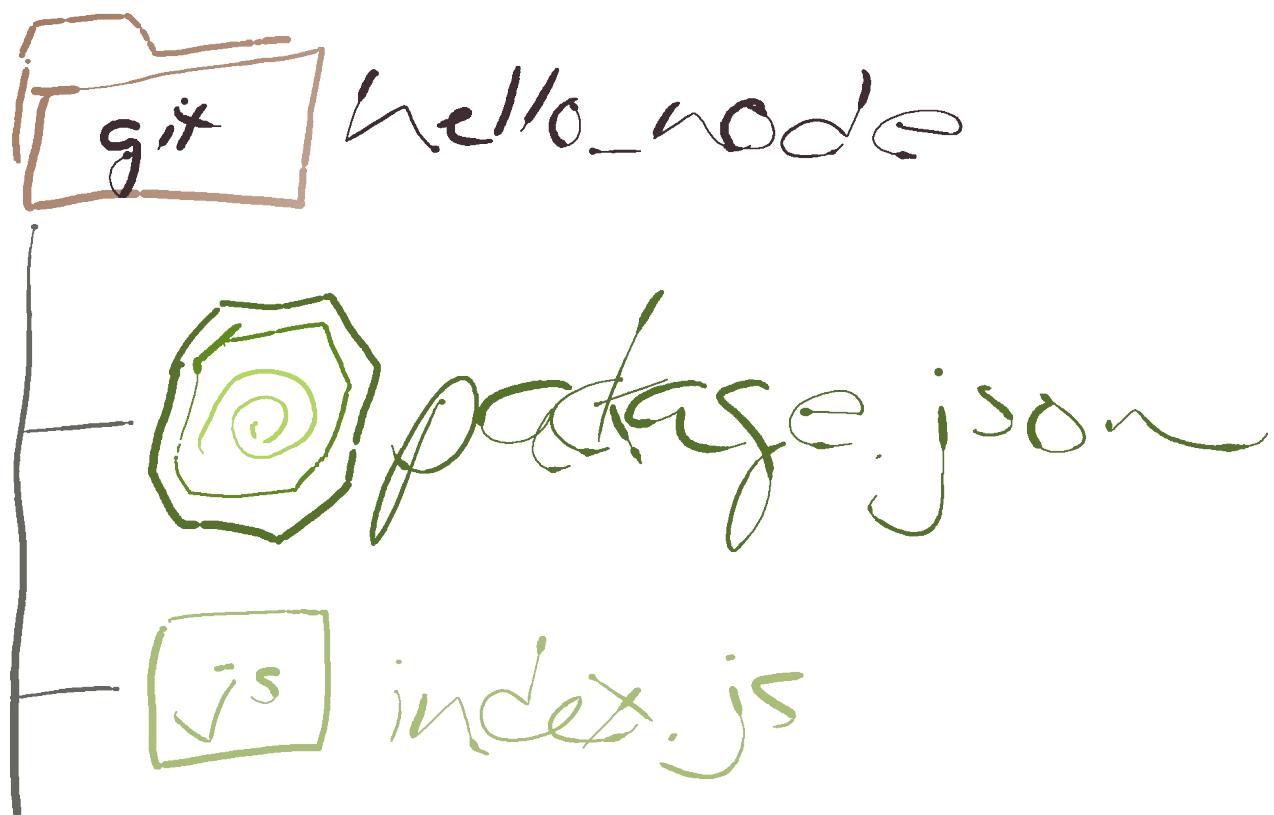
## 2. Setting Application dependancies

Use `npm install <pkg> --save` afterwards to install a package and save it as a dependency in the `package.json` file. Once you run this command, a dependency object will be created inside of your `package.json`. Go ahead - take a look! It looks like this, right?

```
"dependencies": {
 "express": "^4.12.4"
}
```

All node package files are saved to `node_modules` folder. If provided a `package.json` file from another project, you can run the `npm install` command to fetch/install the required packages for that app!

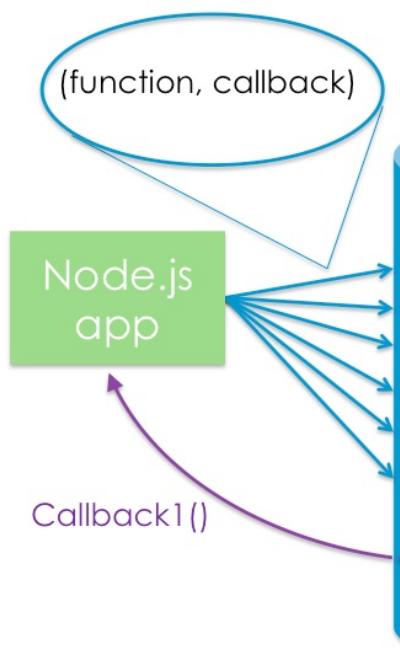
## Node File structure



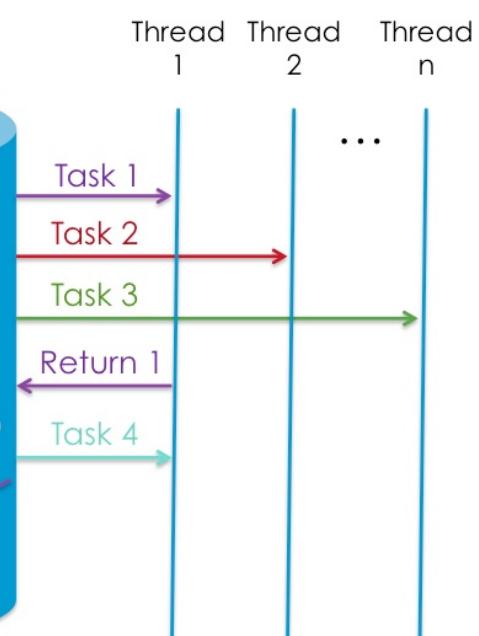
```
~\ProjectDirectory\
- package.json
- index.js
```

## 7.2 Node Event Loop

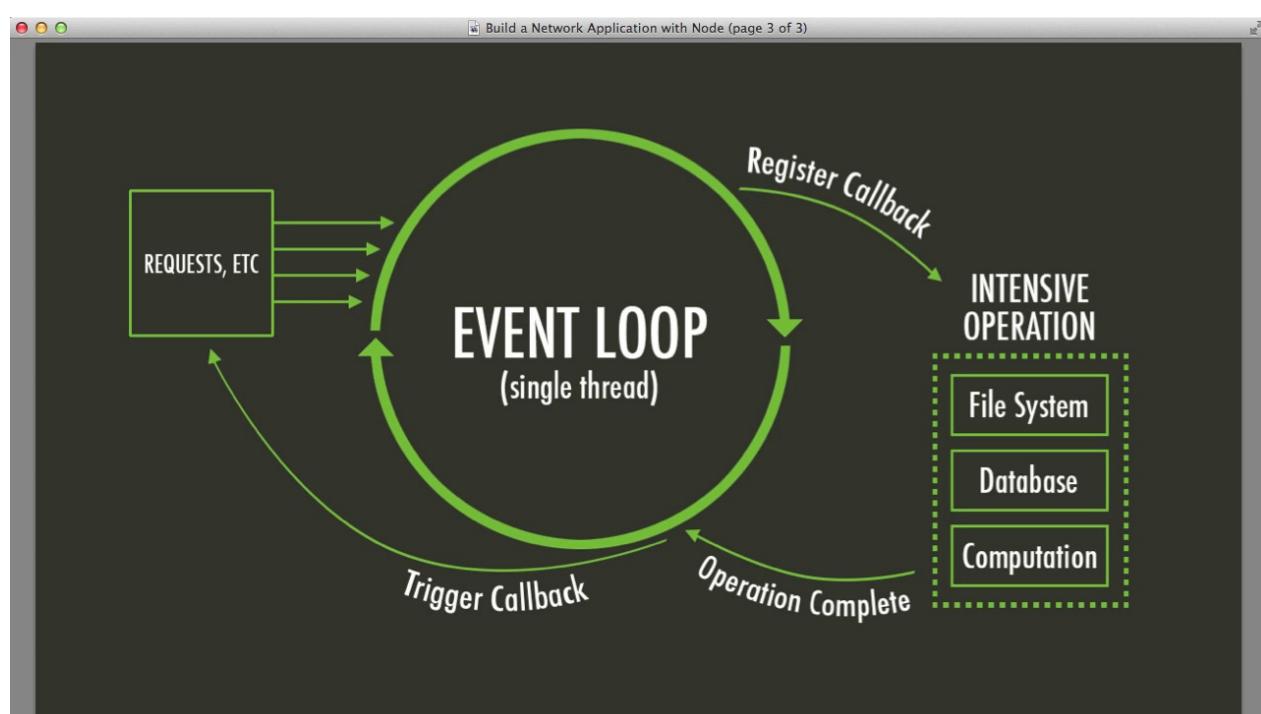
- 1** Node apps pass async tasks to the event loop, along with a callback



- 2** The event loop efficiently manages a thread pool and executes tasks efficiently...



- 3** ...and executes each callback as tasks complete





# Node Examples

## Simple Web Server

```
// assign a variable to an NPM library
var http = require('http');

// http library: create a server, and listen for requests
// and responses
http.createServer(function(request, response) {
 response.writeHead(200); // status message 200 ok
 response.write('holy f I just built a server'); // send data to server
 response.end(); // stop sending
}).listen(5000); // localhost:5000

console.log('My server is running :)'');
```

## Serving JSON as an API

*This example assumes that a data.json file is in the same directory as our application.*

```
var http = require('http'); // library for http
var fs = require('fs'); // library for filesystem

var json = fs.readFileSync('./data.json');

http.createServer(function(request, response){
 response.writeHead(200);
 response.write(json);
 response.end();
}).listen(5000);

// console message to let us know it is running
console.log('api server is ready');
```

## Serving HTML

*This example assumes that an index.html file is in the same directory as our application.*

```
var http = require('http');
var fs = require('fs'); // file system access!

var html = fs.readFileSync('./index.html');

http.createServer(function(request, response) {
 response.writeHead(200);
 response.write(html);
 response.end();
}).listen(5000);

console.log('HTML server on port 5000');
```

# Ruby/Rails/Sinatra vs. JS/Node/Express

While not strictly a competition (one of the skills you have to practice is knowing which frameworks you should use in which situations), let's compare some differences.

## Why Choose Sinatra/Rails?

- Quickest path to building app with full CRUD
- Better at working with complex data relationships - ActiveRecord rocks!
- When full page refreshes aren't an issue
- Synchronous programming is probably a little easier to grasp in building a straightforward program

## Why Choose Node/Express?

- JavaScript everywhere, one language to rule them all
- Asynchronous means generally faster performance
- Better *concurrency* – it can serve data to more users with fewer computer resources
- Designed to make realtime applications

# Node Modules

Like most other modern languages, Node is modular. In essence, if a file puts something inside of module.exports, it can be made available for use in any other file using

```
require()
```

For example, lets make two files: `touch my-module.js main.js`

```
// my-module.js
var number = 7
module.exports.name = "Kenaniah"
module.exports.arr = [1, 2, 3]
module.exports.getNumber = function(){
 console.log("Get number called. Returning: ", number)
 return number
}

console.log("End of my-module.js file")
```

```
// main.js

// here we're grabbing everything that's "exported" in our other file, and storing it
var my = require('./my-module')

// Variables and such that were not exported aren't in scope
console.log("number is " + typeof number) // undefined

// Anything exported can be accessed on the object
console.log("Name is: ", my.name)

// Closures are still closures
console.log("The number is: " + my.getNumber())

// JavaScript is still JavaScript
console.log("The array contains " + my.arr.length + " elements")

// Let's see the module we imported
console.log(my)
```

Then try running:

```
node my-module.js
node main.js
```

## Exercise: Modules

Partner up with your neighbor - your task is to make a module together (`car.js`) and that defines a car – with both properties and functions – and export it as a module to a `main.js` file.

In the `car.js` file:

Properties should include:

- color, convertible (boolean), speed (0, at first)

Functions specs:

- include accelerate and decelerate
  - these should take one argument, the speed, and add or subtract it from the current speed
  - return a string with the old speed and new speed
- call these functions at the bottom of the file

In the `main.js` file, be sure to require the module and console log a message about your car object, including the current speed of the car.

# Homework: Requesting with Node

Tonight you need to use Node.js to make a request to **OMDB**.

- `npm init` a new project.
- Define an `movie.js` and an `app.js`.

## movie.js

- The `movie` file will contain a **Module**.
- This module should *export* a function named `get(movieTitle)`.
- `get(movieTitle)` should make a *request* to OMDB with the `movieTitle` as an argument.
- You should `console.log` your output.
- You should test this to verify it works!

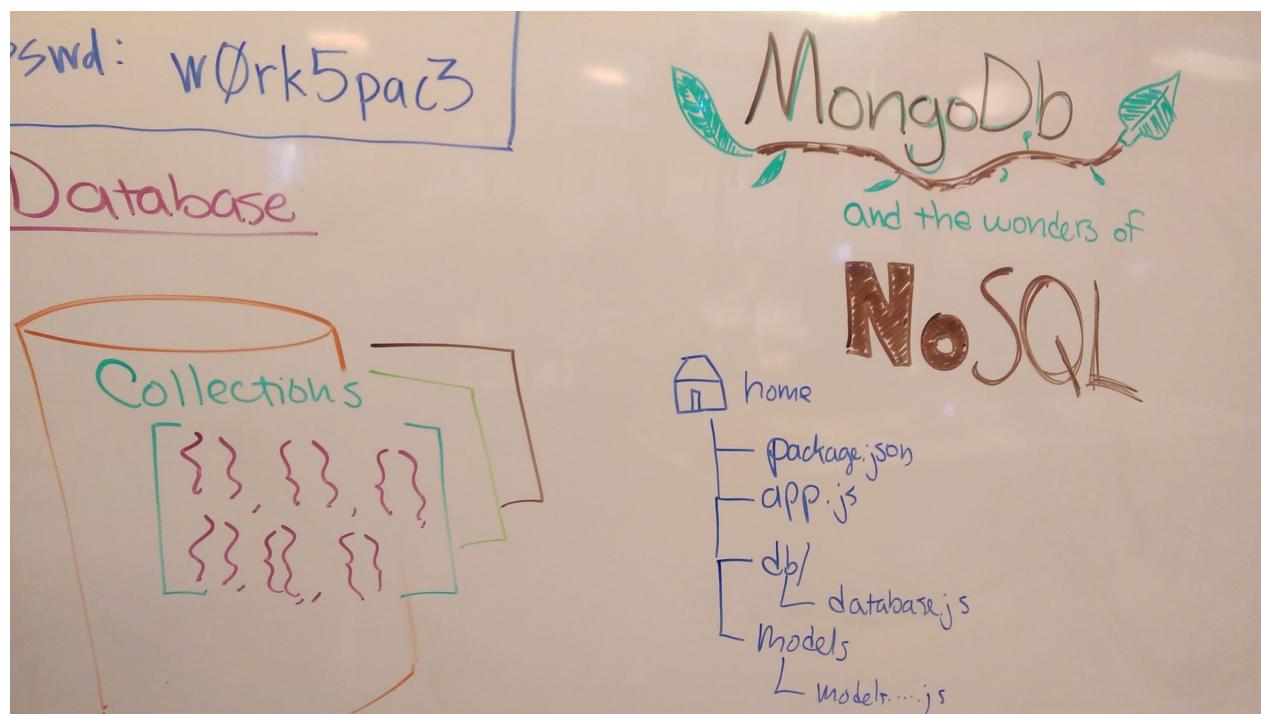
## app.js

- The `app` should *require* your movie module.
- You should define your `threeFavouriteMovies` as an array.
- You should then use the `Array.forEach()` function to loop through them.
- As you loop through them, you should call on your movie module's `get(movieTitle)`

## Starter Code: app.js

```
var movie = require("./movie");
var threeFavouriteMovies = [];
threeFavouriteMovies.forEach(function(film){
 // your code here
})
```

# MongoDB and the Wonders of NoSQL



## Introduction to MongoDB

### Objectives:

- Understand what NoSQL is and what it accomplishes
- Compare NoSQL to traditional SQL and understand each other's pros/cons
- Install MongoDB on your system
- Import JSON as a MongoDB document into a database as collection
- Use the Mongo console to interact with data

## Mongo + Node

### Objectives:

- Use Mongod to run a MongoDB server
- Use the Mongoose ORM to connect to MongoDB
- Define Models as Schemas in MongoDB
- Connect your Models to a Node application

## Mongo CRUD

### **Objectives:**

- Use a Model to create an entry in a collection
- Use a Model to query a collection
- Use a Model to update an entry in a collection
- Use a Model to destroy an entry in a collection

### **Code Examples**

- [https://github.com/ga-chicago/playing\\_with\\_mongo](https://github.com/ga-chicago/playing_with_mongo)
- [https://github.com/ga-chicago/my\\_first\\_node\\_api](https://github.com/ga-chicago/my_first_node_api)

## npm Installation Notes

Check and see if you are using npm 2.0~. You can do so by `npm -v`. If you are using 3.0 or newer, you should run the following commands to install **npm v2.~**.

```
brew uninstall npm brew install npm npm install -g npm@2.14.2 npm install
mongoose -g
```

# Installing MongoDB

**MongoDB:** <https://mongodb.org/>

To install MongoDB, we need to install it and create a folder for the database. We've provided an automated install script here: <https://github.com/ga-chicago/MongoDBInstallScript/>

*Annotated version of the script*

```
echo -e "**** Hi! This script will install MongoDB."
echo -e "You could be asked for your admin password up to four times."
update homebrew
brew update
use homebrew to install
echo -e "Installing MongoDB"
sudo brew install mongodb -Y
create the mongodb swap directory
echo -e "Creating the MongoDB directory..."
sudo mkdir -p /data/db
set read/write permissions for this directory to root
echo -e "Setting Permissions to the /data/* directory for MongoDB"
sudo chmod -777 /data/*
start mongodb!
echo -e "Starting Mongod..."
sudo mongod
```

If you run into any problems during installation, please refer to this document:  
<http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/>

## Starting MongoDB

To start mongodb, simply run the following command: `sudo mongod`. Use CTRL-C to stop the service.

# Mongo & Importing JSON to a Database

## Console w/MongoDB

- To allow connections to your MongoDB database (such as applications needing to connect or Mongoose...), run `sudo mongod`. Think of it as `mongod` oversees **MongoDB**.
- To connect to the MongoDB terminal (similar to `psql`), enter the `mongo` command.

## Console commands

- `help` : Show help.
- `db.help()` : Show help for database methods.
- `db.<collection>.help()` : Show help on collection methods. The can be the name of an existing collection or a non-existing collection.
- `show dbs` : Print a list of all databases on the server.
- `use <db>` : Switch current database to . The mongo shell variable db is set to the current database.
- `show collections` : Print a list of all collections for current database
- `show users` : Print a list of users for current database.
- `show roles` : Print a list of all roles, both user-defined and built-in, for the current database.
- `show profile` : Print the five most recent operations that took 1 millisecond or more.
- `show databases` : No description needed.

## Import JSON into a database

```
mongoimport --db <databaseName> --collection <collectionName> <jsonFile.json> --jsonArray
mongoimport --db britney --collection songs britney.json --jsonArray
```

Only use the `-jsonArray` flag if you are importing an array of objects! Otherwise, you don't need it!

# The Mongoose ORM

**Mongoose** connects our *application* to our MongoDB *database*. To add it as a dependency to our project, we just need to install it as follows:

```
npm install mongoose --save
```

Now, create a new folder in your project called `models`. Then, create a file called `db.js` -- this is where we're going to store our database specific files. *No actual models* will be in this file. Inside of our `db.js` file, we're going to require **Mongoose**:

```
var mongoose = require('mongoose');
```

## Connect to Mongo using Mongoose

```
// mongodb://username:password@localhost:27027/database-name
var connectionString = 'mongodb://localhost/test';

// connect to database
mongoose.connect(connectionString);
```

*Connect Error?* If you are getting a connection error, you may need to replace "localhost" with 127.0.0.1.

## Mongoose Events

These events tell us the status of our connection to MongoDB; this are *VITAL* to debugging.

- **connected**: connected to database
- **error**: an error has occurred with the database
- **disconnected**: the connection to the server has been terminated

*Example*

```

mongoose.connection.on('connected', function () {
 console.log('Mongoose connected to ' + connectionString);
});
mongoose.connection.on('error',function (error) {
 console.log('Mongoose connection error: ' + error);
});
mongoose.connection.on('disconnected', function () {
 console.log('Mongoose disconnected!');
});

```

## Including Your Database in your application

In your primary `app.js` file, we need to include our `/models/db.js` file. This will establish our connection to our database.

### `app.js`

```

// include mongodb via /models/db.js
require('./models/db');

```

## 4. "Documents", Schemas, and Models

MongoDB stores data in each database as **documents**. These documents are stored as **BSON**; this is just *binary* JSON data. The tool that we're using to communicate with Mongo - **mongoose** takes this BSON data and turns it into JSON for our ease of use. *One of the primary advantages of using MongoDB with Javascript is that everything is essentially a Javascript object!*

Each time you create a **document** in MongoDB, it is given an `_id`. MongoDB automatically creates this when each document is created and assigns it a unique ObjectId value. It can be considered the *primary key* of your Document.

## Schema Data Types

MongoDB allows the following *schema* types inside of a `document` :

- String
- Number
- Date
- Boolean

- Buffer (binary information such as images and video)
- Mixed (mixed data)
- Array (arrays of the same data type *or* an array of nested sub-documents)
- ObjectId (used with `_id`)

## Building a Model with a Schema

Let's create a new file: `/models/tasks.js`; the following Schema will be placed inside of there.

We now need to model some data. For example, say we have a list of tasks we'd like to complete. We're going to store our **tasks** as documents in MongoDB. A task as a Javascript object would look like:

```
var task = {
 name: "My task",
 description: "This is something I need to do",
 completed: false
};
```

We can use the `mongoose.Schema({})` method to create a schema.

```
// require mongoose
var mongoose = require('mongoose');

var TaskSchema = new mongoose.Schema({
 name: String,
 description: String,
 completed: Boolean
});
```

Notice how we're declaring the *type* of data stored in the object. Say we want to have a default value, though. When you create a task, it is not completed yet, no?

```
// require mongoose
var mongoose = require('mongoose');

var TaskSchema = new mongoose.Schema({
 name: String,
 description: String,
 completed: {type: Boolean, "default": false }
});

// export our Model for use in our node app// export!
module.exports = mongoose.model('Task', TaskSchema);
```

We put the word `default` in quotes because it is a reserved word in Javascript. All we've changed is adding a `default` value as well as specifying a `type` (based on what is allowed in the Schema).

Finally, to make our actual MVC-esque **model**, we need to tell Mongoose to create a model:

`var Task = mongoose.model('Task', TaskSchema);` Where we declare `Task` as the new model name and assign a schema to this model.

## 5. Mongoose Query Methods

- `find` : General search based on a supplied query object
- `findById` : Look for a specific ID
- `findOne` : Get the first document to match the supplied query
- `geoNear` : Find places geographically close to the provided latitude and longitude
- `geoSearch` : Add query functionality to a geoNear operation

## 6. MongoDB CRUD

```
var mongoose = require('mongoose');
var Task = require('../models/Tasks');

// get ALL items in an array
Task.find(function (err, tasks) {
 console.log(tasks)
});

// create an item
Task.create({ name: "task #1", description: "a simple task"}, function (err, task) {
 console.log(task);
});

// get a specific item by _id
Task.findById(id, function (err, task) {
 console.log(task);
});

// update a task by _id
Task.findByIdAndUpdate(id, { description: "updating this" }, function (err, task) {
 console.log(task);
});

// delete by _id
Task.findByIdAndRemove(id, { params: "object stuff" }, function (err, task) {
 console.log("Deleted:");
 console.log(task);
});
```

# Mongoose Model Example

```
var mongoose = require('mongoose');

// declare a schema (schema) for a model
// mongoose schema data types:

// String
// Number
// Date
// Boolean
// Buffer - mixed data types (images, video, etc)
// Mixed - mixed data types
// Array (arrays must contain data of the same type!)
// ObjectId (_id) - unique GUID or ID associated with your new model

var TaskSchema = new mongoose.Schema({
 name: String,
 completed: Boolean,
 description: String,
 updated_at: { type: Date, default: Date.now }
});

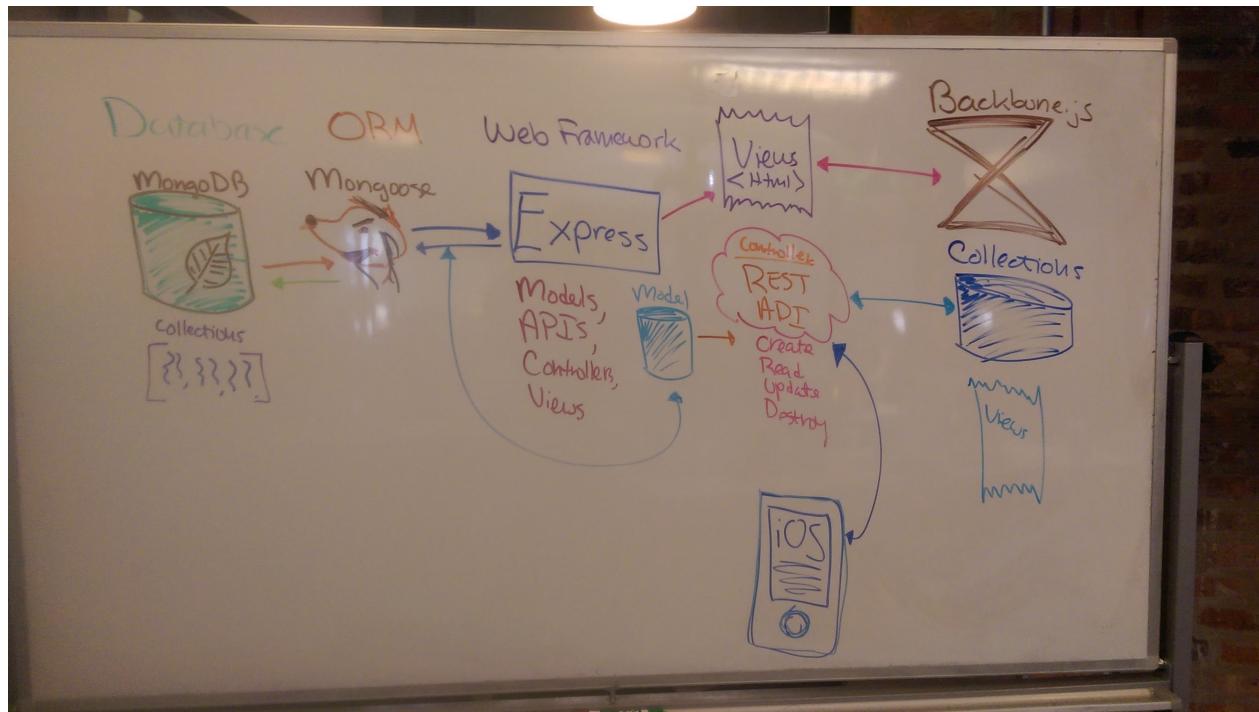
// hey node! I want to export the following module for re-use
// mongoose - create a model with a (name, schema)
module.exports = mongoose.model('Task', TaskSchema);
```

# Using Mongoose + Models with Node

```
// require out database
require('./db/database');
// require our model
var Item = require('./models/Item');

// query the db
Item.find(function(error, items) {
 if (error) console.log(error);
 console.log(items);
});
```

# Express.js (aka Sinatra for Node)



## Express 101

### Objectives:

- Learn how to include Express in a project
- Compare Sinatra to Express
- Create routes using Express
- Refactor a former project to use Express

## RESTful API with Express

### Objectives:

- Generate an Express.js application
- Include a MongoDB database and Mongoose models in a project
- Create an API that allows users to perform CRUD applications
- Test your RESTful API to verify all actions work

## Use Views and Templates

### Objectives:

- Use Express.js Views to add visual content to a server
- Use template frameworks to dynamically add content similar to ERB
- Implement views in your Express apps

# Q&A, Node Recap So Far

- How do we start a Node project from scratch? [Video](#)
- Relationships with MongoDB?
- Little nitty details about Mongoose
  - *self.table\_name* for Mongoose? [Thanks, Stack Overflow](#)
  - You can specify a specific table name as the optional third argument when using `mongoose.model : mongoose.model('UserInfo', UserInfo, 'NameOfMyCollectionBecauseIAmCool');`
- How to use just SQL in Node? [Sequelize](#)

## 7.4 Getting Started with Express

To get started, let's go ahead and add **Express.js** to our project by running the following command:

```
npm install express --save
```

### Installing the Express 4.0 Application Generator

We want to be able to generate applications in the same way that Ruby on Rails can.

We're going to run a command to install an Express.js application generator:

```
sudo npm install -g express-generator
```

We'll touch more on this later.

### Make sure we're all set

We're going to run a few commands to make sure we have the required dependencies for our application:

```
node --version #checks node version
npm --version #checks npm version
express --version #checks express version
```

## 7.4 Express.js (Sinatra for Node)

Express: <http://expressjs.com/>

### 1. Your First Express project

We're going to build an API in Express as well as render a View (similar to Ruby's MVC pattern).

If you haven't created an `app.js` file, go ahead and do so.

### Let's build our first API

Inside of your `app.js`, we're going to...

```
// require express and use it
var express = require('express');
var app = express();

// create some JSON to send to a user
var data = {
 "message": "It.. is... alive!",
 "description": "your very first express API call"
};

// setup a 'get' route... JS version of sinatra!
app.get('/', function(req, res){
 res.send(data);
});

// setup a 'post' route.. again, look familiar?
app.post('/', function(req, res){
 res.send({
 "message": "thanks.. you've sent me some useful data"
 });
 console.log(req);
});

app.listen(3000);
console.log("Your Express app is running...")
```

### 2. Request object (body, params, and query)

The **request** object (seen as `req`) contains `body`, `params`, and `query` objects.

Install body-parser: `npm install body-parser --save`

To parse the `body` (where all data is sent via `$.ajax` and POSTMAN), we need to include the following Node Modules:

```
// include the body-parser module so we can see inside of req.body
var bodyParser = require('body-parser');
app.use(bodyParser.json()); // support json encoded bodies
app.use(bodyParser.urlencoded({ extended: true })); // support encoded bodies
```

Now we can appropriately read data from query strings, the request body, and params.

Let's modify our above API route to read from the request body.

```
// create an API 'post' route.. let's look at the different sets of data we can get back
app.post('/api', function(req, res){
 res.send({
 "message": "thanks, " + req.body.name + "... you've sent me some useful data"
 });

 // we'll log out the params, body, and query objects here for you to see what is included
 console.log("req.params");
 console.log(req.params);
 console.log("req.body");
 console.log(req.body);
 console.log("req.query");
 console.log(req.query);

});
});
```

```
// create another 'post' route that accepts 'params'
// this would look like... http://localhost:3000/items/42
app.post('/api/items/:id', function(req, res) {
 console.log(req.params);
 res.send("Your item #" + req.params.id + " is not in stock :(... womp womp");
});
```

We can test out our `post` routes using an Ajax call:

### Sending over data to the `request.body`

```
$.post("/api", { name: "James"}, function(data) {
 console.log(data);
});
```

## Sending over data using `request.params`

```
$.post("/api/items/42", { item: "Hitchhiker's Guide to the Galaxy"}, function(data
 console.log(data);
});
```

## 3. Generating an Express project

We're going to use the *Express* Generator to create an application. While the Express framework is lean and lightweight compared to Ruby on Rails, there are a few application settings that you can use.

The Express Generator allows you to easily choose which CSS preprocessor you'd like to use as well as which template framework you'd like to use.

You use the Express generator in a new folder (we're going to use whichever folder you created in the first part of this walkthrough). The syntax is as follows:

```
express --css preprocessor_name --template
```

## Pre-processors

Express supports two out-of-the-box pre-processors:

- LESS (<http://lesscss.org/>) - Similar to Sass; one major difference is variable declaration (`$colour` vs `@colour`)
- Stylus (<https://learnboost.github.io/stylus/>): Robust, Ruby-like syntax

*Usage:*

```
express --css less
express --css stylus
```

## Template Frameworks

Express supports the following template frameworks out of the box:

- Jade (default; all projects use this unless specified otherwise)
  - <http://jade-lang.com/>
  - Ruby-like syntax; similar to Stylus
- EJS (embedded Javascript)
  - <http://www.embeddedjs.com/>
  - Very similar to ERB
  - Familiar to Underscore
  - **We're using this in class!**
- Hogan
  - <http://twitter.github.io/hogan.js/>
  - Similar to Handlebars, Mustache, Angular
  - Similar to Twig/Smarty from PHP
- JSHTML
  - <http://james.padolsey.com/cool-stuff/introducing-jshtml/>
  - Sort of spaghetti-code like
  - Uses comments to create UI elements

*There is a node\_module that supports other frameworks like Handlebars...:*

<https://www.npmjs.com/package/consolidate>

*Usage:*

```
express --css less # uses jade
express --css stylus --ejs # uses EJS
express --css stylus --hogan # uses EJS
express --css stylus --jshtml # uses EJS
```

Now, we need to organize our application to make sense. We're going to open the project and move some folders around as well as create some.

## 4. Rendering a View

We need to install and save the EJS view engine as a dependency of our project: `npm install ejs --save`

Next, we'll include the ability to read files via the path:

```
var path = require('path');
```

Now, we'll tell our application to use the EJS view engine and where to look for View files:

```
app.set('views', path.join(__dirname, 'views'));
// use the EJS view engine
// make sure you npm install ejs --save first!
app.set('view engine', 'ejs');
```

## Working Example

```
// include the path reader... we need this to read files
var path = require('path');

// view engine setup
// set to look for views inside of "/views"
app.set('views', path.join(__dirname, 'views'));
// use the EJS view engine
// make sure you npm install ejs --save first!
app.set('view engine', 'ejs');

// render an index file... (/views/index.ejs)
app.get('/', function(req, res, next) {
 res.render('index', { title: 'Stuff', message: "Hi friends. Let's send some data to
});
```

### A sample EJS View /views/index.ejs

```
<!DOCTYPE html>
<html>
 <head>
 <title><%= title %></title>
 </head>
 <body>
 <h1><%= title %></h1>
 <p><%= message %></p>
 </body>
</html>
```

Now, when you browse to `http://localhost:3000` you should see a welcome page!

## 5. EJS Overview

- <http://www.embeddedjs.com/>
- <https://www.npmjs.com/package/ejs>

- Supports many Ruby on Rails like helpers:  
<https://code.google.com/p/embeddedjavascript/wiki/ViewHelpers>
- Should look and feel just like ERB.

## Template Syntax

A conditional statement:

```
<% if (user) { %>
 <h2><%= user.name %></h2>
<% } %>
```

## Compiling Teplate/Using

```
// declare template (str: string template)
var template = ejs.compile(str, options);
template(data);
// => Rendered HTML string

ejs.render(str, data, options);
// => Rendered HTML string
```

## Images

```
<%= img_tag('maid.jpg') %>
```

## Loops

```
<h1><%= title %></h1>

 <% for(var i=0; i<supplies.length; i++) { %>

 <a href='supplies/<%= supplies[i] %>'>
 <%= supplies[i] %>

 <% } %>

```

## 6. Express.js Recap/Sinatra Comparison

## GET requests in Express

```
app.get('/api', function(request, response) {
 // stuff happens
});
```

## Sending Data

```
app.get('/api', function(request, response) {

 var someData = {
 description: "I am really tired",
 message: "zzzzzzzzzz"
 };

 response.send(someData);

});
```

```
get '/api' do
 someData = { description: "I am really tired", message: "ZzzZZzzz" }
end
```

## Rendering Views

```
app.get('/', function(request, response) {

 var someData = {
 description: "I am really tired",
 message: "zzzzzzzzzz"
 };

 // template name, data to use with template
 response.render('nameOfMyView', someData);

});
```

```
get '/api' do
 someData = { description: "I am really tired", message: "ZzzZZzzz" }
 erb :nameOfMyView
end
```

## POST requests in Express

```
app.post('/api/givemeallyourinfo', function(request, response) {

 // request.body, request.query, request.params
 response.send({
 "status": "200",
 "message": "Thanks.. we got all your info"
 });

});
```

```
post '/api/givemeallyourinfo' do
 {
 "status": "200",
 "message": "Thanks.. we got all your info"
 }
end
```

## POST with Params

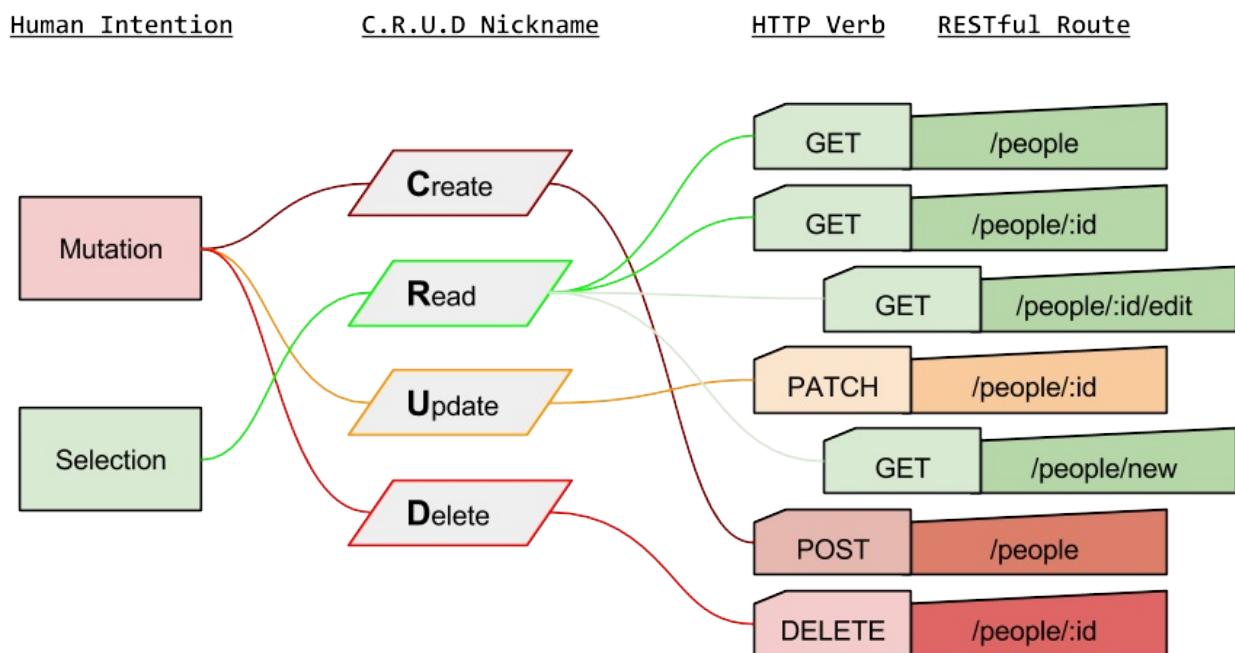
```
// localhost:5000/api/people/42/james
app.post('/api/people/:id/:name', function(request, response) {

 var id = request.params.id; // :id => 42
 var name = request.params.name; // :name => james
 response.send("You've requested #" + id + "!");

});
```

# RESTful API with MongoDB

In `app.js` we need to include additional files and dependencies. We also need to create a `routes/tasks.js` file to include. Go ahead and do that now.



In `app.js`, add the following:

```

// require additional modules
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

// require mongodb & model
require('./models/db'); // db
require('./models/Tasks'); // model

// include route files in /routes
var routes = require('./routes/index');
var tasks = require('./routes/tasks');

// start express app
var app = express();

// we need to parse req.body.. so we add parsers
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

// uncomment after placing your favicon in /public
//app.use(favicon(__dirname + '/public/favicon.ico'));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(require('less-middleware')(path.join(__dirname, 'public')));
app.use(express.static(path.join(__dirname, 'public')));

// define our http routes and bind those routes to routes in /routes
app.use('/', routes); // localhost:5000/
app.use('/api/tasks/', tasks); // localhost:500/api/tasks

```

Our **/routes/tasks.js** file will look just like this:

```

var express = require('express');
var router = express.Router();

var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var mongoose = require('mongoose');

```

```
var Task = require('../models/Tasks');

/* GET /api/tasks listing. */
router.get('/', function(req, res, next) {
 console.log(req.body);
 Task.find(function (err, tasks) {
 if (err) return next(err);
 res.json(tasks);
 });
});

/* POST /api/tasks */
router.post('/', function(req, res, next) {
 console.log(req.body);
 Task.create(req.body, function (err, task) {
 if (err) return next(err);
 res.json(task);
 });
});

/* GET /tasks/:id */
// http://localhost:5000/api/tasks/5566a21e1e3a211aa1c63495
router.get('/:id', function(req, res, next) {
 Task.findById(req.params.id, function (err, task) {
 if (err) return next(err);
 res.json(task);
 });
});

/* PUT /tasks/:id */
router.put('/:id', function(req, res, next) {
 console.log(req.body);
 Task.findByIdAndUpdate(req.params.id, req.body, function (err, task) {
 if (err) return next(err);
 res.json(task);
 });
});

/* PATCH /tasks/:id */
router.patch('/:id', function(req, res, next) {
 console.log(req.body);
 Task.findByIdAndUpdate(req.params.id, req.body, function (err, task) {
 if (err) return next(err);
 res.json(task);
 });
});

/* DELETE /tasks/:id */
router.delete('/:id', function(req, res, next) {
 console.log(req.body);
 Task.findByIdAndRemove(req.params.id, req.body, function (err, task) {
 if (err) return next(err);
 });
});
```

```
 res.json(task);
 });
});

module.exports = router;
```

# Lab: Express Generate

- We're going to use `express --css less --ejs` to build a website.
- This will auto-generate an Express application for us.
- When we use the generate tool, we can start the app using `npm start`

## Sinatra vs Express

- In `dragons`, create a new folder called `sinatra_vs_express`
- Change into `sinatra_vs_express`
- Run the following command: `express --css less --ejs`
- Open the folder in atom: `atom .`

## Comparing Differences

- With the person next to you...
- Take a look around for the next 20 minutes in your new app.
- Compare it directly to Sinatra.
- Create a list of 4 similarities between Sinatra/Express
- Be prepared to briefly share these with the class.

## Lab: Express REPS

1. Create an Express application that serves a static JSON file. We used the `fs` library to do this yesterday. Feel free to refer to your code from yesterday for assistance.
2. Create an Express RESTful API with the resource (URL) of `/api`.
3. Use the Express Generate command to create basic website; use EJS views to render a page (similar with ERB in Sinatra/Rails). You are to create a website that makes an `$.ajax/$.getJSON` request to an API on the server it is on (similar to Sinatra). This API should return JSON with the `message` of *it is a beautiful day* and a `note` of *Chicago summers rock*. It should then render this data somewhere on the page using jQuery.

## Second Pass Friday

Let's build a full Node app from scratch and record it, yeah?

# Prototypes

Prototypes are used in javascript for inheritance instead of classes. Below is some code written in ruby and javascript so you can compare how to do inheritance in each language.

```
class Animal
 attr_reader :kingdom, :cellular_structure
 attr_accessor :legs
 def initialize(legs)
 @kingdom = "Animalia"
 @cellular_structure = "Eukaryotic animal cells"
 @legs= legs
 end
end

class Dog<Animal
 attr_reader :sound

 def initialize(legs=4)
 super
 @sound = "bark"
 end

 def speak
 @sound
 end
end

fatso= Dog.new
p fatso.sound
p fatso.speak
p fatso.kingdom
```

```
function Animal(legs){
 this.kingdom= "Animalia"
 this.cellular_structure= "Eukaryotic animal cells"
 this.legs= legs
};

function Dog(sound){
 this.legs= 4
 this.sound= sound
 this.speak= function speak(){
 return sound
 }
};

Dog.prototype = new Animal(4);

var fatso = new Dog("bark");

console.log(fatso.kingdom)
console.log(fatso.speak())
console.log(fatso.sound)
console.log(fatso.legs)
```

# Full Stack from scratch

*Time for us to build a guide.... a comprehensive guide to building a full stack MEAN app from scratch.*

## Starting My App

- `npm init` to initialize my application!
- Generate an Express app using `express --css less --hbs` (using Handlebars)
- `npm install` to install Node module dependencies
- Verify app works! `npm start` OR `nodemon` to run the app
- Browse to `localhost:portnumber` to verify it runs

## Setting up a database

- If we want sample data (such as a seed file) we can import JSON directly into MongoDB using the `mongoimport` command.
- Start `mongod` as root (`sudo mongod`) otherwise we can't communicate back and forth to MongoDB with Mongoose. Do this after importing data.
- If I don't have any starting data, Mongo will automatically create a collection + database on connection and query.
- We need an ORM for MongoDB.... that is **Mongoose**
- `npm install mongoose --save` to install Mongoose in my project!
- We need a few folders to store important information about our database:
  - `db/`
  - `models/`
- The db folder contains the `database.js` connection file. It also contains JSON for seed data if it exists.
- Need a connection string + connect using Mongoose (`mongoose.connect()`)
- In our database.js, we need to listen for Mongo...
  - `connected`
  - `disconnected`
  - `error`
- The models folder contains a Schema (aka Model definition) for each collection.
- `require('./db/database')` in your `app.js` to connect to your db for the app!

## Serving Our Data!

- Routers and Controllers can be synonymous
- Our `routers` folder will contain individual routers.
- The default two: `index.js` and `users.js`
- I want to make a new router. I call it `routers/something.js`
- I need to `var nameOfRouter = require('./routers/something');`
- Ok, now it is time to map it to a resource, aka `localhost:3000/something/`
- `app.use('/resourceurl', nameOfRouterVariable)`
- That would map to `localhost:3000/resourceurl`
- Now, I need to add routes to my router... how do I do that?
- In my individual router file, I need to `router.httpVerb(...)`
  - `router.get`
  - `router.post`
  - `router.patch`
  - `router.put`
  - `router.delete`
- I can respond with 3 different things!
  - `res.send('a string of text')`
  - `res.json(someObjectToTurnToJson)`
  - `res.render('nameOfView', {dataObjectToUseOnPage})`

## Models

- required to be in that models folder.
- Each model represents a collection!
- Each model needs to `require('mongoose')` so it can talk to the ORM
- Each model also needs to define a `mongoose.Schema` ... this allows us to...
  - Blueprint data
  - Assign data types to keys
  - Minor data type validation
  - Even require certain keys have values
- Export `mongoose.Model('NameOfModel', schema, optionalCollectionNameLikeSelf.table_name)`

# ES6 Classes

For now, we can only use these on the server side with **Node** or using a transpiler such as **Babel** or **Traceur**. We can play with them live in JSFiddle, too. Example:

<https://jsfiddle.net/codeforcoffee/qmnky6j2/1/>

```
'use strict'

class Apprentice {
 constructor(properties,div){
 this.props = properties;
 this.el = document.getElementsByTagName(div)[0];
 }
 toObject(){
 return this.props;
 }
 render(){
 for(var prop in this.props){
 this.el.innerHTML = this.el.innerHTML + ' ' + this.props[prop];
 }
 }
}

var obiWan = new Apprentice({color: 'blue', master: 'Qui-Gon', hair: 'bad'},'div');
console.log(obiWan.toObject());
obiWan.render();

class Master extends Apprentice {
 toObject() {
 return 'I am not weak-minded. You will not know anything about me.';
 }
}

var quiGon = new Master({ color: 'green', hair: '...', beard: 'magnificent'}, 'div');
console.log(quiGon.toObject());
console.log(quiGon.props);
```

# Ajax on an Express View

```
<!DOCTYPE html>
<html>
 <head>
 <title></title>
 <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
 <link rel='stylesheet' href='/stylesheets/style.css' />
 </head>
 <script type="text/javascript">

 </script>
 <body>
 </body>
</html>
```

## This Week's Videos

1. Intro to Node - <https://www.youtube.com/watch?v=X1IfdK9Nqe0>
2. Mongoose/MongoDB Models - <https://www.youtube.com/watch?v=0o9pw9MJ8Ag>
3. Express.js - <https://www.youtube.com/watch?v=S38J0o2ZDLQ>
4. Second Pass: Building a Full Stack RESTful API - <https://www.youtube.com/watch?v=sEaGmnbrDjQ>

# Node Homework

This weekend your **optional** homework is to do the following:

1. Create a full stack RESTful API in Node (from scratch).
2. Use MongoDB to server a collection through an API route.
3. The API should be mapped to `localhost:3000/api` .
4. You should also render a view at `localhost:3000/` . This view should contain an **Ajax** Call to the API to return all of your items.
5. Use that Ajax call to render your API's data on your page.
6. An example of making an Ajax call is in this [Gitbook](#).

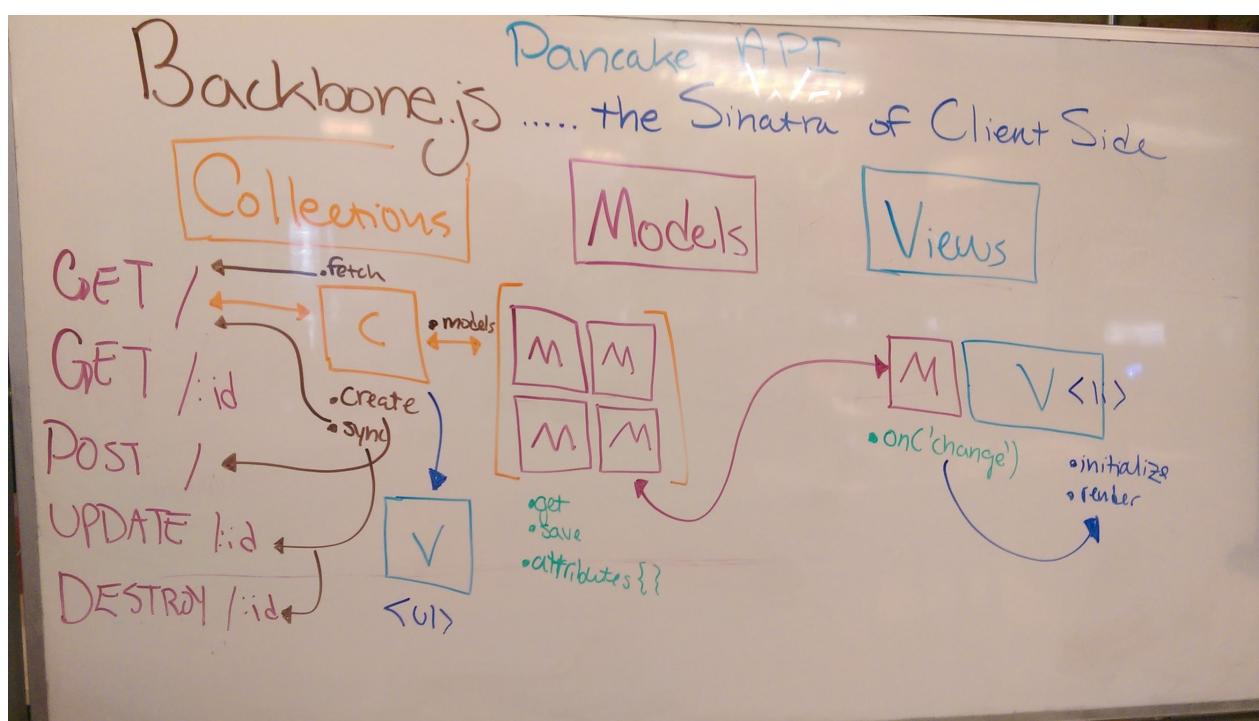
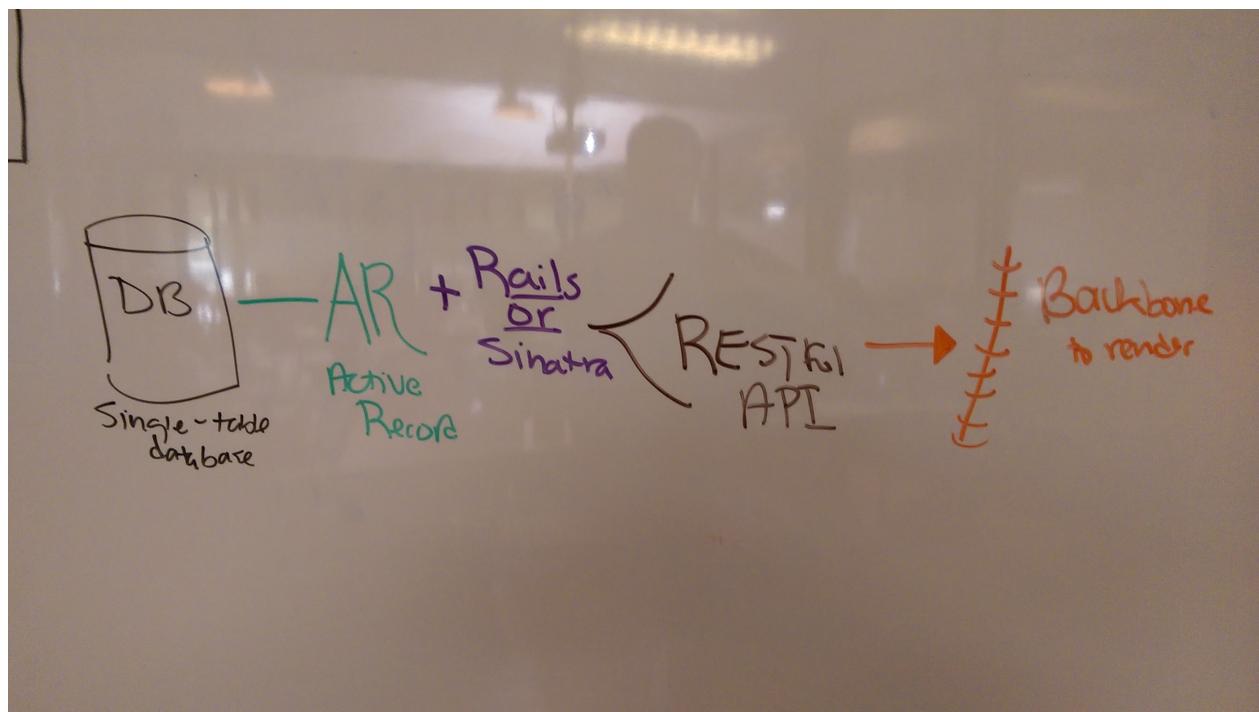
This should be submitted by pull request *if you do it*. This practice is going to lead into using Backbone.js next week! By knowing how we can make Ajax calls to our API on our server, we'll be able to do the same with Backbone.js (but easier).

# Backbone.js

## Getting Started:

- [Sinatra + Backbone.js Boilerplate](#) - just clone down, `bundle exec rackup`, browse to `localhost:9292` and play in the console!
- [Codepen with Backbone, jQuery, Underscore](#)

# Backbone.js



Today we're building a Pancake API!

- Source Code: [https://github.com/ga-chicago/mongo\\_pancake\\_rest\\_api](https://github.com/ga-chicago/mongo_pancake_rest_api)

## Awesome Sources

- Addy Osmani's [Developing Backbone.js](#) *Free book*
- Namespacing
- Backbone.js for Absolute Beginners
- Backbone Tutorials

## CDN Links

```
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.0
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/underscore
<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.3.3/backbone.js">
```

# What is JSON?

- Stands for *Javascript Object Notation*
- Similar in use to CSV/XML
- Incredibly Popular and gaining popularity
- Parsing JSON is fast thanks to tools like the V8 Engine

## The `JSON` Javascript Object

- Contains **two helpful methods**:
- `stringify(javascriptObject)` : converts a Javascript Object to a JSON string
- `parse(jsonString)` : converts a JSON string into a Javascript Object
- Fallback: If you're supporting older browsers, you'll need a polyfill such as `json2.js`

## JSON Examples

```
{
 "someKey": "someValue",
 "maybe": "sure?",
 "numbers?": 3.14,
 "booleanToo?": false,
 "someArray": [
 "one", "two", "three"
],
 "objected": {
 "myKey": "myValue"
 }
}
```

## JSON Warnings!

- no comments allowed
- last key/value pair must not have a ,
- keys must be in strings
- strings must utilize " (double quotes)

## More JSON Examples

```
{
 "name": "magda",
 "breed": "tabby",
 "age": 12,
 "colour": "grey and white",
 "has_microchip": true
}
```

```
{
 "favouriteNumber": 42,
 "firstPrime": 2,
 "arrayOfPrimes": [
 2,
 5,
 7,
 17,
 11,
 13
]
}
```

## XML Example

```
<tagname attribute='value'>
 <resource attribute='value' />
 <another_resource new_attribute='valuetwo' />
</tagname>
```

# Backbone.js Models

- Models are designed to interact with a data source.
- They are the *truth* in your application.
- Backbone's models are designed to connect to individual items in an API but can be used for singular objects.
- Open the console and create the following model line by line & describing:

```
// using http://openweathermap.org/weather-data#current
var Weather = Backbone.Model.extend();
// we're going to use open weather
// create a new instance of Weather
var forecast = new Weather();
// show empty attributes
console.log(forecast.attributes);
// set url to get data from
forecast.url = "http://api.openweathermap.org/data/2.5/find?q=Chicago&units=imperial"
// fetch data from API
forecast.fetch();
// console log out new attributes
console.log(forecast.attributes);
// we're going to use a model's getter to get a value from the attributes
var data = forecast.get("list");
// log out data
console.log(data[0].main);
// put some data together
var weatherData = {
 location: data[0].name,
 humidity: data[0].main.humidity,
 temp: data[0].main.temp
};
// cool, let's make a template
// ah, see where I'm going with this?
// an underscore template
var template = _.template("It is <%= temp %> degrees Farenheight in <%= location %> w.
// now render it all together
template(weatherData);
// now, let's add this to the DOM, shall we?
$('body').html(template(weatherData));
```

## Attributes and history inside of a model

- Models have the `attributes` object

- You **get** and **set** values; these are used to trigger events
- `attributes` is just where your data is temporarily stored!
- Here is a **Weather** model. It logs out the model's attributes before and after a fetch.

```
var Weather = Backbone.Model.extend({
 url: '',
 initialize: function() {
 console.log("- Weather View has been created. Here are its attributes -");
 console.log(this.attributes);
 this.fetch();
 console.log("- Weather View has called fetch()! It is super effective! here are now my attributes!");
 console.log(this.attributes);
 }
});
var weather = new Weather({url: "http://api.openweathermap.org/data/2.5/find?q=Chicago&units=imperial"});
```

- Models store information about themselves, too. There is also a `changed/history` object.
- Constructors automatically bind certain values in models like they do in views (remember `el`?)

## Questions to Consider

Q: How do you think that models work?

Q: What properties and abilities have you seen that models have?

Q: What properties and abilities do you think models have?

# Backbone.js Views

- A **view** is just a small piece of your website.
- They organize your website's interface into logical views.
- They are meant to be re-drawn without needing to load a webpage.
- They keep you from having to refresh your page!

## Attributes and Abilities

### Properties

Name	Description
<b>el</b>	Corresponding DOM element that represents this view. It can already exist or if not, it will be created for you.
<b>tagName</b>	Specific tag that will be created for this view's *el*
<b>className</b>	CSS class that will be applied to this view's *el*
<b>id</b>	CSS ID that will be applied to this view's *el*
<b>\$el</b>	A cached jQuery object for the view's element.
<b>attributes</b>	A hash of attributes that will be set as HTML DOM element attributes on the view's *el* (id, class, data-properties, etc.)
<b>template</b>	A template or reference to a template for use with the *render()* method

### Methods

Name	Description
<b>constructor</b>	Accepts several options that will be attached to the view automatically.
<b>initialize()</b>	An optional function that is called the moment a view is instantiated.
<b>render()</b>	A function to render your view as HTML into the DOM.
<b>remove()</b>	Removes this View from the DOM



# Underscore Templates

- Codepen with Backbone, jQuery, Underscore

## Ways to render strings with dynamic values

```

var name = 'vader',
 hobby = 'dueling',
 food = 'anything liquid',
 age = 49;

// string concatenation
var stringBuilder = name + ' enjoys ' + hobby + ' while eating ' + food + ' and is ' +
 age + ' years old';

// ruby
//var stringBuilder = '#{name} enjoys #{hobby} while eating #{food} and is #{age} years old';

// the underscore way
var story = "<%= name %> enjoys <%= hobby %> while eating <%= food %> and is <%= age %> years old";

```

## Building a Template with Underscore

- 1 - Define a Template

Take some re-usable HTML..

```

<h1>Welcome, Brad</h1>

<p>Today's outlook is positive, with a chance of sunshine. Have a great day!</p>

```

- 2 - And turn it into a template!

```

<h1>Welcome, <%= username %></h1>

<p>Today's outlook is <%= outlook %>, with a chance of <%= weather %>. Have a great day!

```

Now, place it in your HTML...

```
<div id='content'></div>

<script type='text/template' id='welcome'>
 <h1>Welcome, <%= username %></h1>
 <p>Today's outlook is <%= outlook %>, with a chance of <%= weather %>. Have a great
</script>
```

- 3 - Render Template

```
var newModel = {
 username: 'Grand Moff Tarkin',
 outlook: 'Grim',
 weather: 'the death star blowing up'
};
// create a re-usable template function
// based off of the template variable of newTemplate
var newCompiledTemplate = _.template(newTemplate);
// render that shit
$('#content').append(newCompiledTemplate(newModel));
```

**Result:** <http://codepen.io/code-for-coffee/pen/YXMqbv>

# Binding Backbone to an API

## Getting Started

1. First, we need to include all of our resources - jquery, underscore, backbone.js.
2. We also need a RESTful API. This can be from any framework.
3. We'll need to specify 4 different things:
  - o *Collection*: a special array of Models; each model corresponds to a *document* or *table row* in a database. The collection corresponds to a Collection or Table in a database. These will be rendered as parent DOM elements (ol, ul, section).
  - o *Model*: Individual models that represent the documents or rows of a table. These will be rendered as children DOM elements (li, article).
  - o *CollectionView*: This view is what will render our Collection. <ul> , <section>
  - o *ModelView*: This view is what will render our Model. <li> , <article>
4. We then need to define individual templates for both the ModelView and CollectionView. When putting these together, ask yourself: 'What should this look like in HTML?'
5. We can then define a basic Backbone application.
6. Note! If you need to support MongoDB in your application, use

```
Backbone.Model.idAttribute = "_id";
```
7. Add a \$(document).ready event.

## Code So Far

```
// namespace
var app = app || {};
var active = active || {};

// define my 4 important parts!
app.Collection = Backbone.Collection.extend({});

app.Model = Backbone.Model.extend({});

app.CollectionView = Backbone.View.extend({});

app.ModelView = Backbone.View.extend({});

// mongoDB support!
Backbone.Model.idAttribute = "_id";

// the document is ready
$(document).ready(function(){
});
```

## Our View

```

<section id='pancake-listing'>
 <h2>Pancakes Listing</h2>
</section>

<!-- defining an underscore.js template -->
<script type='text/template' id='recipe-template'>
<article>
 Ingredients: <%= ingredients %>

 Type: <%= type %>

 Topping: <%= topping %>

 Syrup: <%= syrup %>

 Instructions: <%= instructions %>

 Time: <%= time %>

</article>
</script>

<!-- loading custom app -->
<script src='/javascripts/app.js'></script>
</pre>

```

## Skeleton of How Backbone Renders

```

// namespace
var app = app || {};
var active = active || {};

app.Model = Backbone.Model.extend({
});

// define my 4 important parts!
app.Collection = Backbone.Collection.extend({
 model: app.Model, // what type of models will this collection hold?
 url: '/api',
 initialize: function() {
 var self = this;
 this.on('change', function() {
 console.log('Our Collection changed.');
 var view = new app.CollectionView({
 collection: self
 });
 });
 this.on('sync', function() {
 console.log('Our Collection synced with the API.');
 var view = new app.CollectionView({
 collection: self
 });
 });
 }
});
// get data from the API
this.fetch();

```

```

 }
});

Backbone.Model.idAttribute = "_id";

app.CollectionView = Backbone.View.extend({
 el: $('#pancake-listing'),
 initialize: function() {
 console.log('CollectionView is a go.');
 // when loaded, let us render immediately
 this.render();
 },
 render: function() {
 console.log('CollectionView is rendering.');
 // we expect our CollectionView to be bound to a Collection
 var models = this.collection.models;
 for (var m in models) {
 new app.ModelView({
 model: m
 });
 }
 }
});

app.ModelView = Backbone.View.extend({
 initialize: function() {
 console.log('ModelView instantiated and awaiting orders, sir');
 this.render();
 },
 render: function() {
 console.log('ModelView rendering.');
 }
});

// mongoDB support!
Backbone.Model.idAttribute = "_id";

// the document is ready
$(document).ready(function(){
 active.collection = new app.Collection();
});

```

## Final Product

```

// namespace
var app = app || {};
var active = active || {};

app.Model = Backbone.Model.extend({

```

```
});

// define my 4 important parts!
app.Collection = Backbone.Collection.extend({
 model: app.Model, // what type of models will this collection hold?
 url: '/api',
 initialize: function() {
 var self = this;
 this.on('change', function() {
 console.log('Our Collection changed.');
 var view = new app.CollectionView({
 collection: self
 });
 });
 this.on('sync', function() {
 console.log('Our Collection synced with the API.');
 var view = new app.CollectionView({
 collection: self
 });
 });
 // get data from the API
 this.fetch();
 }
});

Backbone.Model.idAttribute = "_id";

app.CollectionView = Backbone.View.extend({
 el: $('#pancake-listing'),
 initialize: function() {
 console.log('CollectionView is a go.');
 // when loaded, let us render immediately
 this.render();
 },
 render: function() {
 console.log('CollectionView is rendering.');
 // we expect our CollectionView to be bound to a Collection
 var models = this.collection.models;
 for (var m in models) {
 new app.ModelView({
 model: models[m],
 el: this.el
 });
 }
 }
});

app.ModelView = Backbone.View.extend({
 initialize: function() {
 console.log('ModelView instantiated and awaiting orders, sir');
 this.render();
 },

```

```
render: function() {
 console.log('ModelView rendering.');
 var data = this.model.attributes;
 console.log('Grabbing template...');
 var template = $('#recipe-template').html();
 console.log('Transforming template...');
 var compileTpl = _.template(template);
 console.log('Creating HTML from template and model data...');
 var html = compileTpl(data);
 console.log('Rendering to page...');
 this.$el.append(html);
 // vanilla - this.el.innerHTML = this.el.innerHTML + html;
}
});

// mongoDB support!
Backbone.Model.idAttribute = "_id";

// the document is ready
$(document).ready(function(){
 active.collection = new app.Collection();
});
```

## Videos

- Backbone Models - <https://youtu.be/FZDd0UTSye8>
- Backbone 101 - <https://youtu.be/gbzaYL5bATg>

# Homework

## Research Backbone.js

- Backbone.js by Thomas Davies [Video](#)
- Developing Backbone.js [Free book](#)
- Backbone.js for Absolute Beginners
- Backbone Tutorials

## Research JSON Resume

- Visit <http://jsonresume.org>
- Review the [Schema](#)
- Play around with the [GUI Editor](#) - you can **download** your JSON!
- [Get Started](#) with JSON Resume and turn your resume into a PDF! You do not need to publish it publically if you do not wish.

## Underscore Practice

- Create HTML elements for the following re-usable items.
- Turn them into `text/template` tags.
- Render content with them!
- This will require building out a template, render function, and compile!
- **Blog Post**
  - Title (posted by Author)
  - Blog post description

## ◦ of Comments

- **Event**
  - Event Name
  - Event Date
  - location
  - Price
  - Details



# Backbone Second Pass

## Namespacing Video

- <https://youtu.be/Dp1U-K9SAzQ>

# Backbone.js Models and Collections

## LO's

- Recap Underscore Templates and Views
- Populate models models with data from APIs
- Bind a RESTful API in real time to a Collection

## Intro & Connections

- We're going to manipulate our API with ease using Backbone.js.
- Companies such as Groupon, CenturyLink, Pivotal, NY Times, and more use this design style
- You'll be able to build flexible websites using Sinatra and Backbone.js

## Populate models and collections with data from APIs

## Demonstrate Models

- I will be performing my work inside of Chrome console (and expand it)
- Models are designed to interact with a data source.
- They are the *truth* in your application.
- Backbone's models are designed to connect to APIs.
- Open the console and create the following model line by line & describing:

```

// using http://openweathermap.org/weather-data#current
var Weather = Backbone.Model.extend();
// we're going to use open weather
// create a new instance of Weather
var forecast = new Weather();
// show empty attributes
console.log(forecast.attributes);
// set url to get data from
forecast.url = "http://api.openweathermap.org/data/2.5/find?q=Chicago&units=imperial"
// fetch data from API
forecast.fetch();
// console log out new attributes
console.log(forecast.attributes);
// we're going to use a model's getter to get a value from the attributes
var data = forecast.get("list");
// log out data
console.log(data[0].main);
// put some data together
var weatherData = {
 location: data[0].name,
 humidity: data[0].main.humidity,
 temp: data[0].main.temp
};
// cool, let's make a template
// ah, see where I'm going with this?
// an underscore template
var template = _.template("It is <%= temp %> degrees Farenheight in <%= location %>");
// now render it all together
template(weatherData);
// now, let's add this to the DOM, shall we?
$('body').html(template(weatherData));

```

## Discuss Properties of a model

Q: How do you think that models work?

Q: What properties and abilities have you seen that models have?

Q: What properties and abilities do you think models have?

## Build a Model

- Choose a partner that is **not** next to you!
- In `w06/d01/you_name`, create a new file called `fun_with_models.js`
- Open `fun_with_models.js` in *atom*.
- Search the internet for an API that you can retrieve information from.

- Create a Backbone.js model inside of your file.
- Instantiate a *new* model.
- Give it a **url**.
- **fetch()** the data.
- Console log out your model's **attributes**
- Discuss with your partner the data that has been returned.

## Attributes and history inside of a model

- Discuss the attributes object
- Discuss **get** and **set** and how they trigger events
- 'attributes' is just where your data is temporarily stored!
- Here is some script in slack. It logs out the model's attributes before and after a fetch.

```
var Weather = Backbone.Model.extend({
 url: '',
 initialize: function() {
 console.log("- Weather View has been created. Here are its attributes -");
 console.log(this.attributes);
 this.fetch();
 console.log("- Weather View has called fetch()! It is super effective! here are");
 console.log(this.attributes);
 }
});
var weather = new Weather({url: "http://api.openweathermap.org/data/2.5/find?q=Ch"});
```

- Models store information about themselves, too. There is also a changed/history object.
- Constructors automatically bind certain values in models like they do in views (remember el?)

# Bind a RESTful API in real time to a Collection

## Discuss binding to RESTful APIs

Q: Wouldn't it be nice if we could automatically bind data to the APIs we've been building?

Q: If you could write a really, really simple way to work with an entire API, how would you?

## Describe Collections (5min)

- Imagine being able to communicate directly to an API
- A list of models for each entry in an API
- A... list.... a... COLLECTION
- A collection is the actual Backbone of Backbone.js
- A collection of models; each model is like a vertabre
- I'm going to show you what a collection can do.

## I DO - Walk through collections (and record video) (15min)

```
// create a model for each individual card
var card = Backbone.Model.extend({
 initialize: function(){
 console.log("Hey, a Card Model (no." + this.get("id") + ") was just created!")
 // fetch from the server
 this.fetch();
 }
});

// now, create a list.. a collection of them!
var cardList = Backbone.Collection.extend({
 model: card,
 url: "/api/cards",
 initialize: function() {
 console.log("Collection is a go!");
 }
});

// create a new instance of the list
```

```
var list = new cardList();

// get data from our API
list.fetch();

// create will create a new item on the API
list.create({
 title: "Hello", message: "from backbone.js!"
});

// refresh postman
list.create({
 title: "Good morning!", message: "Your cat says meow."
});

// refresh postman
list.create({
 title: "Weather", message: "It is currently 32C outside."
});

// find models that have a title of where
list.where({title: "Hello"});

// or by id
list.at(1);

list.pop({title: "pop pop", message: "community "});

list.push({title: "falcon punch", message: "woot"});

// if I don't like a model, I can destroy it and delete it
var model = list.at(0);
model.destroy();

// or if I just want to update a model, I can do that too
var anotherModel = list.at(1);

// I'm going to update the title
anotherModel.set("title", "I'm updated! awesome");

// and now sync my changes back to the api
anotherModel.sync();
```

# Backbone.js Events

## Or, how I saved myself from re-attaching events to buttons over and over

### Learning Objectives

By the end of this lesson, you will...

- Understand how to map events to functions
  - Map clicks on DOM elements to functions
  - How to keep those events bound after re-rendering
- 

### Introduction

- jQuery events are long and convoluted to write sometimes
- Backbone gives us a really easy way to map jQuery and Backbone.js events to functions
- It also will automatically re-bind events every time you re-render
- This is awesome, right?

### Real-world application

- Today we're going to demonstrate clicking how to create a login form using a Backbone View and Model.
- Sure, you can do this in jQuery. It'd look something like:

```
// clicking on a button inside of our $el
render: function() {
 // find a child DOM element (button), then bind a click handler to it
 $(this.$el).children('button').on('click', function(event){
 // log the event out
 console.log(event);
 // let us know we've detected a button click!
 alert('the button has been clicked!');
 });
}
```

- Let's do this in Backbone instead

```
events: {
 'button click': function() {
 console.log(event);
 // let us know we've detected a button click!
 alert('the button has been clicked!');
 }
}
```

- This can also be mapped to a method on the View!

```
events: {
 'button click': 'clickMe'
},
clickMe: function() {
 console.log(event);
 // let us know we've detected a button click!
 alert('the button has been clicked!');
}
```

# Backbone.js Views, Again

## 1. What is a view and what problem is it trying to solve? (10min)

---

Q: What is a view?

Q: What problem do you think it is trying to solve?

- A **view** is just a small piece of your website.
  - They organize your website's interface into logical views.
  - They are meant to be re-drawn without needing to load a webpage.
  - They keep you from having to refresh your page!
  - Small individual pieces can be updated on their own terms.
- 

## 2. What is Backbone.js and how do I use it?

---

- It is a library created by Jeremy Ashkenash
- JS Framework designed to work with Ruby easily
- Lightweight; it provides **tools**.
- No design patterns or usage is enforced.
- It contains **models**, **views**, **collections**, and **routers**.
- It is a mv\*/m-v-whatever/mvc like framework.
- Underscore.js is required to use it (also written by Ashkenash)

## Get Underscore/Backbone

- We are going to create a BackboneJS boilerplate page for you to use!
- We like boilerplates, right?
- Browse to github and select **create new repository** named *BackboneBoilerplate*
- `git clone` repository to `/dev/wdi`
- `touch index.html`
- `atom index.html`

- Create `<head>` with `<script>` tags for:
    - google: cdn jquery
    - google: cdn underscore.js
    - google: cdn backbone.js
  - Save index.html
  - Add, commit, and push to git! ... `git add .`, `git commit -m ""`, `git push origin masdter`
  - Every time you want to do create a page, application, or just play around you can re-use this boilerplate!
- 

### 3. Backbone.js Views

---

#### Backbone.js's parts

#### Properties

*Name*	*Description*
<code>**el**</code>	Corresponding DOM element that represents this view. It can already exist or if not, it will be created for you.
<code>**tagName**</code>	Specific tag that will be created for this view's <code>*el*</code>
<code>**className**</code>	CSS class that will be applied to this view's <code>*el*</code>
<code>**id**</code>	CSS ID that will be applied to this view's <code>*el*</code>
<code>**\$el**</code>	A cached jQuery object for the view's element.
<code>**attributes**</code>	A hash of attributes that will be set as HTML DOM element attributes on the view's <code>*el*</code> (id, class, data-properties, etc.)
<code>**template**</code>	A template or reference to a template for use with the <code>*render()*</code> method

#### Methods

*Name*	*Description*
**constructor**	Accepts several options that will be attached to the view automatically.
**initialize()**	An optional function that is called the moment a view is instantiated.
**render()**	A function to render your view as HTML into the DOM.
**remove()**	Removes this View from the DOM

## Reviewing a View

- Now, we're going to take a look at what a view in Backbone.js looks like.
- Then you'll find out what it can do!
- What parts do you see?

*file:*

```
var LogoView = Backbone.View.extend({
 tagName: "div",
 className: "column",
 template: _.template("<h1><%= brand %></h1>"),
 initialize: function() {
 this.render();
 },
 render: function() {
 var data = {brand:"Logo Goes Here"};
 var renderedTemplate = this.template(data);
 this.$el.html(renderedTemplate);
 }
});

var logo = new LogoView({
 el: $('body')
});
```

# Identify and use Underscore.js Templates

## Identify Underscore.js template

- It allows us to write parts of HTML
- And then insert various values into it dynamically as an object/hash/key value pair/dictionary
- Point it out in sample\_view.js
- Notice how I send in a data hash and it gets rendered
- Have students open underscore\_template.js/show on board
- walk through it step by step explaining what is going on

```
// create a template
var tpl = _.template("<h1>Hello, I am <%= name %>! I enjoy <%= hobby %> and <%= food %>.");
// some data object
var data = {
 name: "James",
 hobby: "writing software",
 food: "sunflower seeds"
};
// replace $('body')'s html with it
console.log("first pass");
var rendered_template = tpl(data);
$('body').html(rendered_template);

// can also be written as this.. and changed the data
console.log("second pass");
data.food = "Sunflower butter & lignon berry sandwiches";
$('body').html(tpl(data));
```

## Practice: Render content on your boilerplate

- Open a project with Backbone.js loaded.
- Create a script called `my_template.js` for future reference.
- In it, you'll do the following:
- Create an template to display "your name", "your favourite movie", "your favourite city", and "why I love WDI".
- Create a data object withn those values.
- Render it to `$(body).html()`
- Now, create two more templates and `$(document).append()` them once they are

rendered!

- Finally, copy/paste the contents of this script and run it inside of your Backbone Boilerplate

# Ruby (Sinatra) and Backbone.js

## Sinatra and POST Requests

- Discuss how we are reading raw data from the request.body
- request.body with backbone; params otherwise

```
create
post '/api/cards' do
 request_body = JSON.parse(request.body.read.to_s)
 # must read the request_body keys as strings; symbols return null
 card_args = { title: request_body["title"], message: request_body["message"] }
 card = Card.create(card_args)
 card.to_json
end
```

# Node Authentication w/Passport.js

## Dependencies

```
npm install the following with --save :
```

- passport
- passport-local
- passport-local-mongoose
- express-session

## Setup Passport in app.js

```
var mongoose = require('mongoose');
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;
```

## Sessions

Now, let's configure sessions and Passport:

```
app.use(require('express-session')({
 secret: 'roger panella unicorn man',
 resave: false,
 saveUninitialized: false
}));
app.use(passport.initialize());
app.use(passport.session());
```

## Model

Next, let's build our model:

```
// account model
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var passportLocalMongoose = require('passport-local-mongoose');

var Account = new Schema({
 username: String,
 password: String
});

Account.plugin(passportLocalMongoose);

module.exports = mongoose.model('Account', Account);
```

## Controller

And finally, our controller:

```
//account controller
var express = require('express');
var passport = require('passport');
var Account = require('../models/Account');
var router = express.Router();

router.get('/', function (req, res) {
 res.render('index', { user : req.user });
});

router.get('/register', function(req, res) {
 res.render('register', { });
});

router.post('/register', function(req, res) {
 Account.register(new Account({ username : req.body.username }), req.body.password
 if (err) {
 return res.render('register', { account : account });
 }

 passport.authenticate('local')(req, res, function () {
 res.redirect('/');
 });
 });
});

router.get('/login', function(req, res) {
 res.render('login', { user : req.user });
});

router.post('/login', passport.authenticate('local'), function(req, res) {
 res.redirect('/');
});

router.get('/logout', function(req, res) {
 req.logout();
 res.redirect('/');
});

router.get('/ping', function(req, res){
 res.status(200).send("pong!");
});

module.exports = router;
```

# Swimclan Guide to Backbone.js

```
// instafake.js
// the hip way to be....
// namespaces for our application
/*=====
 N A M E S P A C E S
=====*/
var instafake = instafake || {};
instafake.blueprints = instafake.blueprints || {} // classes&constructors
instafake.active = instafake.active || {} // instantied objects

// blueprints for models & collections
/*=====
 S I N G L E M O D E L C O N S T R U C T O R
=====*/
instafake.blueprints.model = Backbone.Model.extend({ // A simple class constructor for
 initialize: function() { //when an object of this class is instantiated by the collection
 console.log('a model is ready'); // simple console log statement tells us that the model is ready
 }
});

/*=====
 C O L L E C T I O N C O N S T R U C T O R
=====*/
instafake.blueprints.collection = Backbone.Collection.extend({ // define a collection
 url: '/api/instafake', // define the API endpoint that exposes the entire collection
 model: instafake.blueprints.model, // pointer to the single model class that is used
 initialize: function() { // run this function when a new collection of this class is created
 console.log('a collection is ready'); // console log that the collection was instantiated
 this.fetch(); // fetch the first entire collection of models when the collection is created
 this.on('change', function() { // set an event listener for collections of this class
 this.fetch(); //re-fetch whenever there is a change to the collection (new data)
 });
 }
});

// CREATE (CRUD)
/*=====
 C R E A T E F U N C T I O N
=====*/
instafake.create = function(username, post, description, hashtags) { // need a helper function
 if (!username || !post || !description || !hashtags) { // check to make sure that all parameters are present
 console.log('you are missing a parameter! oopsie'); // error message for the parameter
 return false; // bail on the function and do not create anything in the DB if incomplete
 }
 instafake.active.photosCollection.create({ // all is well, call the .create() method
 ...
 });
}
```

```

 username: username, //username field for the new row
 post: post, // post field for the new row
 description: description, // description field for the new row
 hashtags: hashtags // hahtag field for the new row
 });
 return true; // nice thing to do, return true back to the caller so they know that
};

// blueprints for views
// our collection view is sort of like a controller.. it is the * in MV*
/*=====
 C O L L E C T I O N V I E W C O N S T R U C T O R
=====*/
instafake.blueprints.collectionView = Backbone.View.extend({ // A Collection View class
 initialize: function() { // initialize function that runs when this collection view
 this.$el = $('#instafakes'); // every view has a target DOM element (el). Set it
 this.render(); // immediately call the render function of this view instance (see
 var that = this; // save the context of this collection view so that it can be re
 this.collection.on('sync', function() { // whenever there is a synchronize event
 that.render(); // render this collection view to keep it up to date with changi
 }); // keep in mind that this view's collection is not set here. It is passed in
 },
 render: function() { // this view's render ability that really just kicks modelView
 this.$el.html(''); // whenever the collection view is rendered it clears the table
 var models = this.collection.models; // get all the data rows (models) out of the
 for (var m in models) { // iterate over all of the data rows (models)
 var data = models[m]; // just to keep it clean, temporarily store this individual
 new instafake.blueprints.modelView({ // create an 'on the fly' model view instance
 model: data // pass the individual data row (model) to the model view instance
 });
 }
 }
});

/*=====
 S I N G L E M O D E L V I E W C O N S T R U C T O R
=====*/
instafake.blueprints.modelView = Backbone.View.extend({ // The single model View class
 initialize: function() { // when instantiated from the collectionView above (the kind
 this.$el = $('#instafakes'); // this views 'el' is the <tbody> that will show the
 this.template = _.template($('#table-row-template').html()); // pull in the under
 this.render(); // run the 'render' ability just below whenever a view of this clas
 },
 render: function() { // this is the actual writing to the DOM! When this ability is
 var data = this.model.attributes; // set a temporary data variable which will con
 this.$el.append(this.template(data)); // write to the DOM baby!!! The good stuff!
 }
});

```

```
// events/triggers/allthatjazz
/*=====
 E V E N T S A N D T R I G G E R S
=====

$(document).ready(function() {
 instafake.active.photosCollection = new instafake.blueprints.collection(); // insta
 instafake.active.photosCollectionView = new instafake.blueprints.collectionView({ /
 collection: instafake.active.photosCollection // pass the collection attribute to
 });
 $('#add-instafake').on('click', function(event) { // when the user clicks the add
 event.preventDefault(); // make sure that 'submit' buttons dont refresh the screen
 var username = $('#username').val(); // grab the username from the username form
 var post = $('#post').val(); // grab the post from the post form field
 var description = $('#description').val(); // grab the description from the descri
 var hashtags = $('#hashtags').val(); // grab the hashtags from the hashtags form
 instafake.create(username, post, description, hashtags); // call the create helper
 });
 $('#refresh-instafake').on('click', function() { // on click of the refresh button
 instafake.active.photosCollection.fetch(); // this fetch() call will trigger a 's
 });
});
```

# Homework

## 1. Bootflix

Complete the [Bootflix](#) project. You should **fork this repository** and work in it. To submit, create a Pull Request in `WDI_Chi_Dragons` under the Backbone folder with a link to your repository. This will expose you to working with existing assets and code.

## 2. Backbone Studying with the Animals API

- *Take a look at the `Backbone.js` code...*
- What is going on here?
- What happens when our page loads?
- Maybe annotate your source code?
- Here is the full API if you want to play around: [https://github.com/ga-chicago/animals\\_api](https://github.com/ga-chicago/animals_api)

```
var app = {};
app.AnimalModel = Backbone.Model.extend({});
app.AnimalCollection = Backbone.Collection.extend({
 model: app.AnimalModel,
 url: '/api/animals'
});
app.AnimalView = Backbone.View.extend({
 initialize: function(){
 this.listenTo(this.model, 'change', this.render);
 },
 tagName: 'div',
 className: 'animal-home',
 template: _.template('<h2><%= name %></h2><p><%= description %></p><button>Feed</button>'),
 events: {
 'click button': 'feedAnimal'
 },
 feedAnimal:function(){
 var name = this.model.get('name');
 alert(name + ' says "Yum Yum Yum!"');
 },
 render: function(){
 var data = this.model.attributes;
 this.$el.html(this.template(data));
 }
});
app.AnimalListView = Backbone.View.extend({
 initialize: function(){
```

```

 this.listenTo(this.collection, 'sync', this.render);
 },
 render: function(){
 this.$el.empty();
 var models = this.collection.models;
 for(var i=0; i < models.length; i++){
 var singleAnimal = models[i];
 var singleView = new app.AnimalView({model: singleAnimal});
 singleView.render();
 singleView.delegateEvents();
 this.$el.append(singleView.$el);
 }
 }
});
var animals = new app.AnimalCollection();
var animalPainter;
$(document).ready(function() {
 animalPainter = new app.AnimalListView({
 collection: animals,
 el: $('#animals-container')
 });
 animals.fetch();
});

```

### 3. Express.js Reps

- Create a new folder called `ExpressAPI`
- Create an API in Express that serves `get` requests.
- Render three routes using the **GET** request [ `'/api/books'` , `'/api/foods'` , `'/api/music'` ]
- Create three Javascript Objects to send back
- Serve these Models in your API
- Test this API using Postman. Use the Mac Grab app to provide screenshots.
- Post a screenshot in our Slack channel.

### 4. Research Tools and Frameworks

During your downtime, let's research a few things. Here is a list to get you started.

#### HTML & CSS Frameworks

- [Bootstrap](#)
- [Foundation](#)
- [Materialize](#)

- [Skeleton](#)

## **Web App (MV\*)**

- [AngularJS](#)
- [BackboneJS](#)
- [ReactJS](#)
- [EmberJS](#)
- [MeteorJS](#)

## **Visualization & Animations**

- [D3JS](#)
- [ThreeJS](#)
- [FamousJS](#)
- [VelocityJS](#)

## **Testing**

- [JasmineJS](#)
- [MochaJS](#)
- [RSpec](#)

## **Web App to Native App**

- [PhoneGap](#)
- [RubyMotion](#)

## **Microsoft Universal Apps**

## **Task Handlers**

- [Grunt](#)
- [Gulp](#)
- [Task-Manager](#)

## **Web Components**

- [Polymer](#)
- [X-Tags](#)

## **Other Languages**

- Java
- Asp.NET
- C#
- PHP
- Like... hundreds more



# Project #3: Building Your Own API

## Overview

You've already worked in small groups to accomplish various labs and exercises, but this time **we're going to challenge you to work on a whole project with a small team.**

Not only will you be asked to **exercise additional creativity** in designing your own project, you will partner you with other classmates to architect, design, and collaboratively build an API of your own design.

While your last project taught you how to get started with Ruby, SQL, & Sinatra, this project you'll be building something exciting with **Express & Mongo**.

**This is meant to push you both technically and collaboratively.** It's a lot harder to work in a team than to work by yourself, but that's most likely you're going to find yourself doing in your first development job after WDI, and **it's important to learn how to work together.**

Make it work, and make it awesome.

---

## Technical Requirements

Your app must:

- **Use Mongo & Express** to build an API and a front-end that consumes it
- **Create an API using at least 2 related models**, one of which should be a user
- Include **all major CRUD functions** in a **RESTful API** for at least one of those models
- **Consume your own API** by making your front-end with HTML, Javascript, & jQuery (Backbone is a huge plus!)
- **Add authentication to your API** to restrict access to appropriate users
- **Craft thoughtful user stories together**, as a team
- **Manage team contributions and collaboration** using a standard Git flow on Github

- Layout and style your front-end with **clean & well-formatted CSS (LESS is a huge plus)**
  - **Deploy your application online** so it's publically accessible
- 

## Necessary Deliverables

- A **working API, built by the whole team**, hosted somewhere on the internet
- A handmade front-end that **consumes your own API**, hosted somewhere on the internet
- A **link to your hosted working app** in the URL section of your Github repo
- A **team git repository hosted on Github**, with a link to your hosted project, and frequent commits from every team member dating back to the *very beginning* of the project
- A `readme.md` file with:
  - Explanations of the **technologies** used
  - A couple paragraphs about the **general approach you took**
  - **Installation instructions** for any dependencies
  - Link to your **user stories** – who are your users, what do they want, and why?
  - Link to your **wireframes** – sketches of major views / interfaces in your application
  - Descriptions of any **unsolved problems** or **major hurdles** your team had to overcome

---

## Suggested Ways to Get Started

- **Don't hesitate to write throwaway code** to solve short term problems.
- **Read the docs for whatever technologies / frameworks / API's you use.**
- **Write your code DRY** and **build your APIs RESTful**.
- **Be consistent with your code style.** You're working in teams, but you're only making one app per team. Make sure it looks like a unified effort.
- **Commit early, commit often.** Don't be afraid to break something because you can always go back in time to a previous version.
- **Keep user stories small and well-defined**, and remember – user stories focus on what a user needs, not what development tasks need accomplishing.
- **Write code another developer wouldn't have to ask you about.** Do your naming conventions make sense? Would another developer be able to look at your app and

understand what everything is?

- **Make it all well-formatted.** Are you indenting consistently? Can we find the start and end of every div, curly brace, etc?
  - **Comment your code.** Will someone understand what is going on in each block or function? Even if it's obvious, explaining the what & why means someone else can pick it up and get it.
  - **Write pseudocode before you write actual code.** Thinking through the logic of something helps.
- 

## Potential Project Ideas

For this project, we want you to work with your team to build a creative product that you actually think someone will want to use. We won't have time to do tons of customer research, but take some time to brainstorm. If you're struggling for ideas, the ones below could help get you started.

### **Bucketli.st**

Besides finishing WDI, you surely have one or two things you'd love to do with your life. Let's get 'em on paper! You could integrate with a third-party location-based API to allow users to search for a location or venue to add to their bucket list items.

### **Survey App**

Imagine sending out a survey to everyone in the class – what should we eat for lunch today? Or 1-5, how well did you understand what we just learned? It would be even more awesome if it were realtime, so you could see answers pouring in as they're submitted.

### **Hello, Comments**

Imagine the benefits of having an API where you could embed comments into any website you want. They could even update in realtime if you wanted, so that you'd never have to refresh the page. CMS providers across the world could quit writing code from scratch and just embed your widget instead.

---

## Useful Resources

- [MongoDB](#)
  - [Express JS](#)
  - [Getting Starts with Node.js on Heroku](#)
- 

## Project Feedback + Evaluation

- *Project Workflow:* Did you complete the user stories, wireframes, task tracking, and/or ERDs, as specified above? Did you use source control as expected for the phase of the program you're in (detailed above)?
- *Technical Requirements:* Did you deliver a project that met all the technical requirements? Given what the class has covered so far, did you build something that was reasonably complex?
- *Creativity:* Did you added a personal spin or creative element into your project submission? Did you deliver something of value to the end user (not just a login button and an index page)?
- *Code Quality:* Did you follow code style guidance and best practices covered in class, such as spacing, modularity, and semantic naming? Did you comment your code as your instructors as we have in class?
- *Problem Solving:* Are you able to defend why you implemented your solution in a certain way? Can you demonstrated that you thought through alternative implementations? (*Note that this part of your feedback evaluation will take place during your one-on-one code review with your instructors, after you've completed the project.*)
- **Total:** Your instructors will give you a total score on your project between:

Score | Expectations ----- | ----- **0** | *Incomplete. 1 | Does not meet expectations.*  
**2** | *Meets expectactions, good job!* **3** | *Exceeds expectations, you wonderful creature, you!*

This will serve as a helpful overall gauge of whether you met the project goals, but **the more important scores are the individual ones** above, which can help you identify where to focus your efforts for the next project!



## Project Scope

*When building a project scope, you need to ask yourself...*

1. What type of website will I be building? (Is this going to be a landing page? A blog? An e-commerce site?)
2. When do I need this website completed?
3. What is my budget for this project?
4. What is the title of my project?
5. Who is the intended audience of my project?
6. Who will I deliver the final project to?



## Project Scope Survey

1. Which questions should this survey answer?
2. How will you organize yourselves within your team for the project to succeed?
3. Who is to be worked with and when? How can and will we communicate during the project? Will we use email? Slack? Something else?
4. What restrictions are imposed due to the budget of the project?
5. Who are your focus groups (potential *users* and *stakeholders*)?
6. What is the *minimal viable product* due date?
7. How much time do we have for the project until completion? What is the hard deadline?
8. What practical challenges are we already aware of (e.g. language, availability, hosting, design resources)?
9. Which ownership challenges do we have to resolve (e.g. ownership of the result of the project)?



## Initiation Survey

*Below are sample questions that you should ask your focus group/stakeholders...*

1. What do you expect out of the final result of our project?
2. What type of devices will you be using to access the final project?
3. What operating system(s) will you be using to access the final project?
4. What is your preferred web browser?
5. What is your secondary web browser?
6. What version of your current operating system?
7. Will the project need to be **ADA Compliant** for those with disabilities?  
(<http://www.ada.gov/pcatoolkit/chap5toolkit.htm>)
8. Will the project need to be **Cookie Law** compliant for citizens in the UK/EU?  
(<http://www.cookie-law.org/faq/>)
9. Is any content on the website sensitive (such as personal identifiable information or payment information)?
10. What features would you like to see implemented on the website?
11. If there is a current website, what feature(s) do you think works well?
12. If there is a current website, what feature(s) do you think do not convey their intended use or perform them well?

# Project 3 Teams

You must:

- Form a team of 2-4 people
- Select one SCRUM leader
- Select one Git maintainer

You should share contact information with your team. We recommend sharing the two primary methods of contact that you use (for example, James uses Hangouts or email primarily but Jim uses iMessage and SMS).

## SCRUM Leader

This member is in charge of keeping the team organized. You are responsible for:

- Lead a daily SCRUM / standup with your group and the instructors (15 minutes daily).
- Assist your team in project planning and help lock down realistic deliverables.
- Daily recap with instructors regarding project status (15 minutes daily).

## Git Maintainer

This member is in charge of your project's Git organization.

- Create, invite, and administrate members to your project's Git organization and repositories.
- Work with teammates to resolve merge conflicts as they arise.

## Things to Consider

- Who wants to work on the front end Javascript?
- Who wants to work on the front end styling?
- Who wants to work on the servers and deployment?
- Who is in charge of acquiring API keys?
- Who wants to handle the database maintenance?
- Who is sourcing data for your API?
- Who wants to work on the server side development?



# Project Deliverables

Deliverables must be emailed by your team's SCRUM Leader to the instructional staff daily. All three emails are: [jamest@google.com](mailto:jamest@google.com), [adriana.castaneda@google.com](mailto:adriana.castaneda@google.com), [james.haff@google.com](mailto:james.haff@google.com)

- **Monday, 10pm CST:** Project Initiation Survey, ERD Diagrams, Project Scope
- **Tuesday, 10pm CST:** UI Wireframes, User Stories, Database Setup, Sample Data
- **Wednesday, 10pm CST:** Project status update. By now you should be working on the full site.
- **Thursday, 10pm CST:** MVP (minimal viable product) of application - both server and client side.
- **Friday, 11am CST:** Project Presentations! Each team gets 10 minutes (5 minutes presentation & 5 minute Q&A)

# Optional Lectures (will be recorded)

These optional lectures will be recorded and source code will be provided. They will be held in classroom #1.

- Sessions and User Authentication with Passport.js (Tuesday at 11am)
- Relating Mongoose Models (Wednesday at 11am)
- Node on Heroku with MongoLab (free tier) for hosting (Thursday at 11am)

# Conference Rooms

The following conference rooms have been booked out for our class's use this week. Anyone may use them. Please be respectful of other SPACE tenants and be ready to abandon these rooms after the scheduled times if need be.

- Monday: East & West Conference Rooms - 10am until 1pm
- Tuesday: East Conference Room - 10am until 1pm
- Wednesday: East & West Conference Rooms - 10am until 3pm
- Thursday: East & West Conference Rooms - 10am until 3pm

# 1:1s

- Angie will have 1:1s with each of you during the week (Tuesday through Thursday)
- You may schedule outcomes 1:1s with Amy at any point from Monday through Thursday
- Instructional staff will be holding 1:1s with each student for a quick 10 minute chat before Friday

# Week 10: Ruby on Rails

## Bookmarks!

- **Official Ruby on Rails website:** <http://rubyonrails.org/>
- **Official Ruby on Rails Github repository** (you should star this!):  
<https://github.com/rails/rails>
- **Official Rails Twitter:** <https://twitter.com/rails>
- **Getting Started with Rails:** [http://guides.rubyonrails.org/getting\\_started.html](http://guides.rubyonrails.org/getting_started.html)
- **Official Command Line Documentation:**  
[http://guides.rubyonrails.org/command\\_line.html](http://guides.rubyonrails.org/command_line.html)
- **Rails for Zombies:** <http://railsforzombies.org/>
- **Michael Hartl's Rails Tutorial:** <https://www.railstutorial.org/book/>
- **Learn Ruby and Rails:** <http://www.learnrubyandrails.com/>

# 10.0 Getting Started with Rails

## 1. Create & open a new Rails project

**Note:** If you do not have Rails installed, you must install the gem: `gem install rails !`

This is a step-by-step guide to creating a new Rails application. Let's get started! To create a new rails app, you use the following syntax:

- `rails new app_name -d postgresql -T`

Where `app_name` is the name of your application. For example, to create an app called `Coffee_Shop`, I would enter the following into my terminal: `rails new Coffee_Shop`

The `-d postgresql` tells rails to configure the application to use postgres

The `-T` tells rails to omit including the default rails testing framework

This will create a new Ruby on Rails application in a new desdecent folder named after my app's name in the `rails new` command. Enter `ls` to see the folder and then `cd your_app_name` to move into your new app. To open your new Rails project, in **atom**, enter the following command: `atom .`

## 2. Updating your Gemfile

Locate your application's `Gemfile`. Inside of it, you will comment out the following gem(s):

- `gem 'coffee-rails', '~> 4.1.0'`

You will also add the following gem(s):

- `gem 'rspec-rails'`

Once you've edited your `Gemfile`, it is time to feel the ground shake and `bundle !`

## 3. Working with your project locally

- To view your project in the browser you will need to run a server (similar to `rack` ).
- In your terminal, enter `rails server`
- Unlike `rack`, your project will be served at: `http://localhost:3000/`
- Note the difference in ports: `9292` vs `3000`

- To view your project in the console you will run `rails console`
- From this console, you can test out your models, for example.

## 4. Generating Models

To generate a model for your project, you can run the following command:

- `rails generate model Name product_name:string description:text`

**Note: because we strive for efficiency, we're going to start calling our rake commands in the `bin/` directory inside of our project!!!**

## 5. Database Setup/Creation

To automatically generate SQL database tables based on your models, you can run the following command:

- `bin/rake db:create`

To run any changes in your migration scripts, you must run:

- `bin/rake db:migrate`

Finally, to create test entries for RSpec tests, you must run the following command:

- `bin/rake db:test:prepare`

## 6. Creating Routes/Controllers

Create a controller to go along with our model.

- `bin/rails generate controller Roasts index show edit new`

Where `rails generate controller` specifies that we're creating a new controller. `Roasts` is the name. `index show edit new` are the actions to be generated in our controller.

We can also delete actions:

- `bin/rails d controller Roasts index show edit new`
- `bin/rake routes` returns a list of routes in your Rails project.

## 7. Install RSpec in your Project

Once you have included `rspec` in your project, you may generate unit test specs using the following command(s) in your terminal:

```
rails generate rspec:install
```

To run your tests in the command line:

- `bin/rake spec`

## 8. Example RSpec Unit Test

```
require 'rails_helper'

RSpec.describe Object, type: :model do

 describe 'given an object' do
 before do
 @object = Object.new()
 end

 describe '#to_s' do
 it 'returns an object is an "Object"' do
 expectation = @object.to_s # should return 'Object'
 actual = 'Object'
 expect(expectation).to eq(actual)
 end
 end
 end
end
```

## Misc

- In RSpec, `.` is a class method and `#` is an instance method

## Further Reading

- **Official Command Line Documentation:**

[http://guides.rubyonrails.org/command\\_line.html](http://guides.rubyonrails.org/command_line.html)

# 10.0 Rails Cheatsheet

- comment out coffeescript
- add `rspec-rails`
- `bundle`
- `rails generate rspec:install`
- `rails g rspec:install`
- `bin/rails g model Movie title:string genre:string director:string release_year:integer plot:text`
- in RSpec, `.` is a class method and `#` is an instance method
- run tests: `bin/rake spec`

## Database Setup/Creation

- `bin/rake db:create`
- `bin/rake db:migrate`
- `bin/rake db:test:prepare`

## RSpec Unit Test

### Model

```
class Movie < ActiveRecord::Base

 def to_s
 self.title
 end

end
```

### Test

```

require 'rails_helper'

RSpec.describe Movie, type: :model do
 #pending "add some examples to (or delete) #{__FILE__}"

 describe 'given a movie' do

 before do
 @movie = Movie.new(title: "Empire Strikes Back")
 end

 describe '#to_s' do
 it 'displays the title of the movie' do
 expectation = 'Empire Strikes Back'
 actual = @movie.to_s
 expect(expectation).to eq(actual)
 end
 end
 end

end

```

## Setup Routes/Controllers

- bin/rails g controller movies index show edit new
- bin/rails g controller welcome index
- bin/rails d controller movies index show edit new
- bin/rails g controller movies index
- bin/rake routes

```

bin/rake routes
Prefix Verb URI Pattern Controller#Action
movies_index GET /movies/index(.:format) movies#index
 movies GET /movies(.:format) movies#index
 movie GET /movies/:id(.:format) movies#show
 root GET / welcome#index

```

- rails g model User username:string password\_hash:string email:string  
img\_url:string
- g controller users new edit show bin/rake routes

```
bin/rake routes
Prefix Verb URI Pattern Controller#Action
 users GET /users(.:format) users#index {:expect=>[:index]}
 POST /users(.:format) users#create {:expect=>[:index]}
 new_user GET /users/new(.:format) users#new {:expect=>[:index]}
edit_user GET /users/:id/edit(.:format) users#edit {:expect=>[:index]}
 user GET /users/:id(.:format) users#show {:expect=>[:index]}
 PATCH /users/:id(.:format) users#update {:expect=>[:index]}
 PUT /users/:id(.:format) users#update {:expect=>[:index]}
 DELETE /users/:id(.:format) users#destroy {:expect=>[:index]}
movies GET /movies(.:format) movies#index
 movie GET /movies/:id(.:format) movies#show
 root GET / welcome#index
```

# Homework

Make a dark\_mart app and add these 3 routes manually:

- home\_page
- about\_us
- team

These three routes will be accessed via a GET request. Make views for each route and make at least 1 other custom layout.

Then, create a "products" controller using scaffold, remove the line added in routes.rb by the generator (`resources :products`) and re-create the seven restful routes matching the 7 methods in the products controller manually.

# Introduction to Rails Video

- <https://youtu.be/iHwS6iODnaY>

# Controllers and Routes

## Rails VS Sinatra

When a user makes a request to the browser, the web-application needs to know what content to show them. Let's compare with code that we have previously written. In Sinatra we defined the routes within our controller action:

```
get '/posts' do
 @posts = Post.all
 erb : 'posts/index'
end

get '/posts/new' do
 @post = Post.new
 erb : 'posts/new'
end
```

A "route" is a combination of the path that was requested and the HTTP verb that was used to request that path.

Rails has a "routing engine" that separates this from the controller. The configuration for the routing engine is held in the file `config/routes.rb`.

```
Rails.application.routes.draw do
 #code goes here
end
```

All the ruby code inside the block above will be related to defining routes.

## REST

REST Route type	HTTP Verb	URI	ruby method name
index	GET	/posts(:format)	posts#index
create	POST	/posts(:format)	posts#create
new	GET	/posts/new(:format)	posts#new
edit	GET	/posts/:id/edit(:format)	posts#edit
show	GET	/posts/:id(:format)	posts#show
update	PUT	/posts/:id(:format)	posts#update
destroy	DELETE	/posts/:id(:format)	posts#destroy

## Ruby Routes

The following are all the restful routes done manually:

```
get "/posts" => "posts#index"
post "/posts" => "posts#create"
get "/posts/new" => "posts#new"
get "/posts/:id/edit" => "posts#edit"
get "/posts/:id"=> "posts#show"
put "/posts/:id"=> "posts#update"
delete "/posts/:id"=> "posts#destroy"
```

These routes can also be written with `resources :posts`

## Independent Practice

Re-use the application "blog" we've just created and add three routes to it manually:

- faqs
- terms\_and\_conditions
- team

These three routes will be accessed via a GET request. When the user is accessing a route, the browser should display the name of the route, therefore, you'll need to create the related method in a controller that renders text.

Then, create a "categories" controller using scaffold, remove the line added in routes.rb by the generator (`resources :categories`) and re-create the seven restful routes matching the 7 methods in the categories controller manually.

# Rails Views and Layouts

## Sinatra Vs Rails

With Sinatra, we've seen that if there is a file called `layout.erb`, this file will be used as the app layout by default.

If this file `layout.erb` contains a yield statement `<%= yield %>`, then the template rendered for the current route will be "injected" where the yield statement is in the layout. Combined, the templates and layouts will be sent back to the client.

This logic is pretty much the same with Ruby on Rails. When the app is created, Rails will automatically add a layout `application.html.erb` in

`app/views/layouts/application.html.erb`. This layout already contain a yield statement and all the links to css and js files in the head part of the html document.

## Rails Views

In Rails, the logic for the rendering a view is quite straightforward. Given that every route in Rails will execute a method inside a controller, when the method is executed, Rails will look for:

1. A folder inside view corresponding to the controller's name (folder `post` for `PostController` ).
2. A file inside that folder corresponding to an action in that controller (file `index.html.erb` ).

For example , if we call `http://localhost:3000/posts` , Rails will execute the method `index` in the controller `posts` and then, look for a view located in `app/views/posts/index.html.erb` This works when the method always renders the same view.

In some cases though, you may want to render a view with a different name than the current method. For example: if a post was saved successfully go to the index page, otherwise go back to the form. To show a different view than the default view, you would render a different view in a controller's action.

There are different syntaxes for render, and they all do the same thing.

```
render :switch
render action: :switch
render "switch"
render "switch.html.erb"
render action: "switch"
render action: "switch.html.erb"
render "home/switch"
render "home/switch.html.erb"
render template: "home/switch"
render template: "home/switch.html.erb"
render "/path/to/rails/app/views/home/switch"
render "/path/to/rails/app/views/home/switch.html.erb"
render file: "/path/to/rails/app/views/home/switch"
render file: "/path/to/rails/app/views/home/switch.html.erb"
```

## Rails Layouts

By default, Rails will render the layout `application.html.erb` - just like `layout.html.erb` in Sinatra - but sometimes, you want to render a template in a different layout.

You can render a differnt layout by doing the following:

```
render layout: "name_of_layout"
```

This line will just tell Rails to use the same logic of template rendering, but instead of using the default `application.html.erb`, it will render the template inside `name_of_layout.html.erb`.

You can also do `render layout: false` if you wanted to use ajax to pull in a view.

# Homework

Make a dark\_mart api where a user can interact with dark mart products.

1. Your api will only have a product model.
2. A product should have
  - name
  - price
  - img\_url
3. A user should be able to
  - get all products
  - get a specific product
  - add a product
  - edit a product
  - destroy a product
4. Use at least one private method
5. Use at least one validation

# Validations

Validations go in your model files.

example:

```
class User < ActiveRecord::Base
 has_many :songs

 validates :name, presence: true, length: { in: 2..30 }
 validates :email, uniqueness: true, confirmation: true
 validates :email_confirmation, presence: true
end
```

## Independent Practice

- Make your own app with at least one join table.
- Use at least 3 different validations.
- Seed your database using the seed file and `rake db:seed` command.
- Test everything in rails console.

# API with Rails

You guys have learned how to make api's in different frameworks. This is a guide to making one in rails. The rails api gem is a common gem to make a rails api. Many companies use rails for their backend and use a frontend framework for their views. For example, springleaf financial uses rails api with ember. You can use rails api with react or angular if you wanted to.

```
gem install rails-api
rails-api new app_name -d postgresql
```

```
def index
 @song = Song.all
 render json: @song, status: :ok
end

def show
 render json: @song
end

def create
 @song = Song.new

 if @song.save
 render json: @song, status: :created
 else
 render json: @song.errors, status: :unprocessable_entity
 end
end

def update
 if @song.update(song_params)
 head :no_content
 else
 render json: @song.errors, status: :unprocessable_entity
 end
end

def destroy
 @song.destroy

 head :no_content
end
```

## Common Status Codes

### 2XX Success Codes

Status Code	Status	Ruby Symbol
200	OK	:ok
201	Created	:created
204	No content	:no_content

### 4xx Client Error

Status Code	Status	Ruby Symbol
422	Unprocessable Entity	:unprocessable_entity

## Independent Practice

- Add show, edit, and delete functionality in api
- Add at least one private method
- Add at least one validation

# Homework

Make a CRUD application where a user can save their favorite restaurants. A restaurant must have:

- Name
- Description
- Menu URL

Your CRUD app must have authentication and sessions and a user needs to log in to see their restaurants.

# Forms in Rails

This form will work for both new and edit `_form.html.erb`

```
<%= form_for(@book) do |f| %>
 <%= f.label :title %>
 <%= f.text_field :title %>

 <%= f.hidden_field :checked_out, value: @checked_out %>

 <%= f.submit %>
<% end %>
```

## Form Helpers

- `text_field`
- `password_field`
- `text_area`
- `hidden_field`
- `radio_button`
- `check_box`
- `file_field`
- `label`

# Authentication

When we talk about passwords, the commonly used word is "encryption", although the way passwords are used, most of the time, is a technique called "hashing". Hashing and Encryption are pretty similar in terms of the processes executed, but the main difference is that hashing is a one-way encryption, meaning that it's very difficult for someone with access to the raw data to reverse it.

	Hashing	Symmetric Encryption -
	One-way function	Reversible Operation
Invertible Operation?	No, For modern hashing algorithms it is not easy to reverse the hash value to obtain the original input value	Yes, Symmetric encryption is designed to allow anyone with access to the encryption key to decrypt and obtain the original input value

## Rails Authentication

```
gem 'bcrypt', '~> 3.1.2'
bundle
rails g model User email password_digest
rake db:migrate
```

The field `password_digest` will be used to store the "hashed password", we will see what it looks like in a few seconds but know that the original password will never be stored. The logic for hashing a password the right way would be quite long to implement manually, so instead, we will just add a method provided by `bcrypt-ruby` to enable all the hashing/storing the hash logic, and we will add a validation for the email:

In app/models/user.rb :

```
class User < ActiveRecord::Base
 has_secure_password
 validates :email, presence: true, uniqueness: true
end
```

Now that we added this method `has_secure_password` to the user model, we can use two "virtual" attributes on the model, `password` and `password_confirmation`.

```
rails g controller users index new create
```

In "controllers/users\_controller.rb":

```
class UsersController < ApplicationController
 def index
 @users = User.all
 end

 def new
 @user = User.new
 end

 def create
 @user = User.new user_params
 if @user.save
 redirect_to users_path
 else
 render 'new'
 end
 end

 private
 def user_params
 params.require(:user).permit(:email, :password, :password_confirmation)
 end
end
```

In "config/routes.rb":

```
resources :users, only: [:new, :index, :create]
```

In "views/users/index.html.erb":

```
<h1> Users index </h1>
<% @users.each do |user|%>
 <%= user.email %>
 <%= user.password_digest %>
<% end %>
```

In "views/users/new.html.erb":

```

<h1>Sign Up</h1>
<%= form_for @user do |f| %>
 <% if @user.errors.any? %>

 <h2>Form is invalid</h2>

 <% for message in @user.errors.full_messages %>
 <%= message %>
 <% end %>

 <% end %>

 <%= f.label :email %>
 <%= f.text_field :email %>

 <%= f.label :password %>
 <%= f.password_field :password %>

 <%= f.label :password_confirmation %>
 <%= f.password_field :password_confirmation %>
 <%= f.submit %>
<% end %>

```

## Sessions Controller

Now to allow the user to login and out, we will need to create a sessions controller:

```
rails g controller sessions new create destroy
```

Now we can create routes for this controller. In "routes.rb" you should now have:

```

root "users#index"
resources :users, only: [:new, :index, :create]

get 'login', to: 'sessions#new'
resources :sessions, only: [:new, :create, :destroy]

```

In "sessions\_controller.rb" we'll need to add some logic to handle the user's input for email and password:

```

class SessionsController < ApplicationController
 def new
 end

 def create
 user = User.find_by_email(params[:email])
 if user && user.authenticate(params[:password])
 redirect_to root_path
 else
 render "new"
 end
 end

 def destroy
 redirect_to root_url, notice
 end

end

```

Now we need to add a log out form:

In "views/sessions/new.html.erb":

```

<h1>Login</h1>
<%= form_tag sessions_path do %>
 <%= label_tag :email %>
 <%= text_field_tag :email %>
 <%= label_tag :password %>
 <%= password_field_tag :password %>
 <%= submit_tag "Log in" %>
<% end %>

```

In Application controller write:

```

def current_user
 @current_user ||= User.find(session[:user_id]) if session[:user_id]
end

def authorize
 unless current_user
 redirect_to login
 end
end

```

And you're done!



# Client Side Frameworks

## Angular.js



This week we will be connecting our RESTful APIs to Angular. We will be taking the projects that were written using MongoDB, Node, and Express.js. However, you can attach Angular to any project that you wish to use!

### Real-World Examples

- **Tasker**
- The *MEAN* way to stay on task!
- <https://github.com/code-for-coffee/tasker>

# An Introduction to AngularJS

AngularJS: <https://angularjs.org/>

## 1. Setting up your application

We need to include Angular.js in our project. We can install it using npm!

```
npm install angular --save
```

This will download all of the files associated with Angular.js and save it to our `package.json` file. Angular is installed to `node_modules/`

It will include:

```
node_modules/angular on master[!]
$ tree
.
├── README.md
├── angular-csp.css
├── angular.js
├── angular.min.js # <-- use for production!
├── angular.min.js.gzip
├── angular.min.js.map # <-- include in same directory as angular.min.js
├── bower.json
└── index.js
└── package.json
```

From the root of our project's repository, we can copy the required files over using the `cp` command. `cp -r node_modules/angular/angular.min.js public/javascripts` `cp -r node_modules/angular/angular.min.map public/javascripts`

- Now, we need to open whichever view we want to include Angular in. For our case, we're going to create a new `route` in our `routes/index.js` file:

```
router.get('/angular', function(req, res, next) {
 // response: render the '/views/angular.ejs' template and
 // give it the data of { title: 'Tasker', library: "Angular"}
 res.render('angular', { title: 'Tasker', library: 'Angular' });
});
```

- Create a new EJS view: `/views/angular.ejs`
- Copy the contents of `/views/index.ejs` into `/views/angular.ejs`

- Include Angular in your `<head> : <script type="text/javascript" src="/javascripts/angular.min.js"></script>`
- Make sure you include this **after** jQuery!
- Specify where to add your `ng-app`
- `<div ng-app="myApp">...</div>`
- This allows us to *claim* all code inside of this element will be **owned** by Angular.

## Testing.. Testing.. 1..2..3..

Inside of your `/views/angular.ejs` view, let's add some content.

```
<section class="container">
 <article class="row">
 <div class="col-md-12">
 <h1><%= title %></h1>
 <h4>Using <%= library %></h4>
 <p>Not yet implemented!</p>
 </div>
 </article>
</section>
```

Let's let **Angular** do some work on the front end. This will be what is considered the Angular **view** (in their MVC structure).

```
<section class="container">
 <article class="row">
 <div class="col-md-12">
 <h1><%= title %></h1>
 <h4>Using <%= library %></h4>
 <h3>{{'AngularJS lets us ' + 'DYNAMICALLY ' + 'render data on a page!'}}</h3>
 </div>
 </article>
</section>
```

## 2. Dat Angular Template Syntax

Let's take a look at Angular's template syntax. If you're familiar with Handlebars, Hogan, or Mustache this will look semi-familiar:

```
// {{ ALL OF MY CONTENT GOES IN BRACES! HUGGED BY BRACES! }}
{{'AngularJS is up in here'}}
// Whoa - I can evaluate expressions.
{{ 6 * 7 }}
// I'm able to concatenate strings inside of a template, too!
{{'AngularJS lets us ' + 'DYNAMICALLY ' + 'render data on a page!'}}
```

### 3. Angular's MVC

- Angular follows an MVC pattern. It has distinct *views*, *models*, and *controllers*.
- They're supported by:
  - Dependency injection
  - Directives
  - Resources
  - Factories
  - Other modules!
- Angular's magic is in these tools.
- Otherwise we'd just be working with Backbone or plain EJS.

# Angular Controllers

## 1. Views

### Angular owns the DOM

- In our `/views/angular.ejs` file, we're going to specify which part of the DOM that Angular **owns**.
- We're going to give it a **name** `<html ng-app="taskerApp">`
- Angular **owns** this HTML element. That means that it can have full control to modify any DOM element inside of here. This is *extremely* powerful.

### Attaching an Angular controller

Angular allows us to bind a DOM element as a View directly to a controller. We're going to modify our container:

```
<article ng-controller="TaskListCtrl">
 <div class="col-md-12">
 </div>
</article>
```

We're going to tell our controller to render a list of our tasks:

```
<article ng-controller="TaskListCtrl">
 <div class="col-md-12">

 <li ng-repeat="task in tasks">
 {{task.name}}
 <p>{{task.description}}</p>
 <p>{{'Completed: ' + task.completed}}</p>

 </div>
</article>
```

Take a look at that view syntax. **What do you think is going to happen when Angular processes this Controller?**

Let's find out!

## 2. app.js

- We need to create an `app.js` file
- Let's do it here: `/public/javascripts/angular-app.js`
- In your `angular.ejs` view, add a link to it:
- `<script type='text/javascript' src="/javascripts/angular-app.js"></script>`
- This should come after `angular.min.js` !
- Inside of `angular-app.js`, we're going to create our application:

```
// namespace: taskerApp
var taskerApp = angular.module('taskerApp', []);
```

This **binds** our app to where we declared it in `<html ng-app="taskerApp">`.

Cool - now we need to bind that View we made to a controller.

## 3. Controllers

Inside `angular-app.js`...

- We're going to create a controller. To demonstrate Angular's dynamic rendering, we're going to create a fake model to represent what we should see from our API...

```
[{'name': 'Take out of the trash',
'description': 'Don\'t forget the recyclables!',
'completed': true },
{'name': 'Reload ventra card',
'description': 'Balance is low',
'completed': false },
{'name': 'Pick up surprise from UPS...',
'description': 'its a secret to everyone',
'completed': false }]
```

- Now, let's create a controller:

```
taskerApp.controller('TaskListCtrl', function ($scope) {
});
```

- Let's **scope** our sample data to the `<li ng-repeat="task in tasks">` in our view.

```
taskerApp.controller('TaskListCtrl', function ($scope) {
 $scope.tasks = [
 {'name': 'Take out of the trash',
 'description': 'Don\'t forget the recyclables!',
 'completed': true },
 {'name': 'Reload ventra card',
 'description': 'Balance is low',
 'completed': false },
 {'name': 'Pick up surprise from UPS...',
 'description': 'its a secret to everyone',
 'completed': false },
];
});
```

- We're just giving `tasks` some objects.
- This is also a for-in loop in our view.

## OPEN YOUR VIEW

- Seriously, go do it!
- Here's a link!
- <http://localhost:5000/angular>
- DID YOU SEE THAT? Those TASKS were rendered in a list on our page!
- This is MVC in action using Angular.
- **How does this feel?**

# Binding Angular to a RESTful API

## Binding our API to an Angular controller

- Brace yourselves ... awesomeness is coming.
- Time to update our TaskListCtrl controller.
- Use dependency injection to inject a scope
- We want our view to have access to items inside `$scope`
- In our view, `$scope.tasks` is used in `task in tasks`

```
taskerApp.controller('TaskListCtrl', function ($scope, $http) {
 $http.get('/api/tasks').success(function(data) {
 console.log('fetched data');
 console.log(data);
 $scope.tasks = data;
 });
});
```

- We're using the `$http` in Angular to fetch data.
- `$http` makes a *get* request to our specified API path.
- On Success, it logs out the data.
- It also binds the returned data from our API `tasks` to `$scope`
- Pretty cool, eh?

## Let's Clean Up Our View

Let's take care of some styling.

**angular.ejs**

```
<div class="navbar-header">
 <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1">
 Toggle navigation

 </button>

 <%= title %>
</div>

<article ng-controller="TaskListCtrl">
 <div class="col-md-12">

 <li ng-repeat="task in tasks">
 <h4> {{task.name}}</h4>
 <p>Description {{task.description}}</p>
 <p>Completed? {{task.completed}}</p>

 </div>
</article>
```

## style.css

```
ul {
 list-style-type: none;
 margin-left: -20px;
}
```

Now our app looks a little bit more.. professional!

## Creating/Deleting

### angular.ejs

```
<article ng-controller="TaskListCtrl">
 <div class="col-md-4">
 <h1>Create Task</h1>
 <form ng-submit="createTask(name, description)">
 <input class='form-control' type="text" ng-model="name" name="name" />
 <input class='form-control' type="text" ng-model="description" name="description" />
 <input class='btn btn-primary' type="submit" id="submit" value="Submit" />
 </form>
 </div>

 <div class="col-md-8">

 <li ng-repeat="task in tasks">
 <h4> {{task.name}}</h4>
 <p>Description {{task.description}}</p>
 <p>Done? </p>

 </div>
</article>
```

## angular-app.js

```
var taskerApp = angular.module('taskerApp', []);
taskerApp.controller('TaskListCtrl', function ($scope, $http) {

 $scope.fetch = function() {
 $http.get('/api/tasks').success(function(data) {
 console.log('fetched data')
 console.log(data);
 $scope.tasks = data;
 });
 };

 $scope.createTask = function(name, description) {
 $http.post("/api/tasks/", {name : name, description: description}).success(function() {
 $scope.fetch();
 });
 };

 $scope.completeTask = function(task) {
 console.log(task);
 var answer = confirm("Are you sure you want to complete this task?");
 if (answer == true) {
 console.log("Deleted Task with _id:" + task["_id"]);
 $http.delete('/api/tasks/' + task["_id"]);
 $scope.fetch();
 }
 };

 $scope.fetch();
});
```



# Unit #4 Project: Your Portfolio Project

## Overview

**You've come a long way, and it's time to show it.** This will be your most advanced project to date, and if you put creativity into it, it'll hopefully be the thing you want to show off most prominently in your portfolio.

**You get to call the shots and invent your own idea,** choosing a framework & tools that are appropriate for what you want to build. Pull from everything you've learned so far, and **tackle something that'll push you a little outside of your comfort zone.**

Make sure you review your project proposal with your instructor so you can make sure it's **something you can accomplish in the limited time we have**, and make sure it's **something that'll be impressive visually**. Sometimes people do judge a book by its cover – or an app by its design.

---

## Technical Requirements

Your app must:

- **Build a full-stack application** by making your own backend and your own front-end
  - **Have an API of your design**
  - **Have an interactive front-end**, preferably using a modern front-end framework
  - **Be a complete product**, which most likely means multiple relationships and CRUD functionality for at least a couple models
  - **Use a database**, whether that's one we've covered in class or one you want to learn
  - **Implement thoughtful user stories** that are significant enough to help you know which features to build and which to scrap
  - **Have a visually impressive design** to kick your portfolio up a notch and have something to wow future clients & employers
  - **Be deployed online** so it's publicly accessible
-

## Necessary Deliverables

- A **working API**, hosted somewhere on the internet
- A **working front-end**, hosted somewhere on the internet
- A **link to your hosted working app** in the URL section of your Github repo
- A **git repository hosted on Github**, with a link to your hosted project, and frequent commits dating back to the very beginning of the project
- A **readme.md file** with:
  - An embedded screenshot of the app
  - Explanations of the **technologies** used
  - A couple paragraphs about the **general approach you took**
  - **Installation instructions** for any dependencies
  - Link to your **user stories** – who are your users, what do they want, and why?
  - Link to your **wireframes** – sketches of major views / interfaces in your application
  - Link to your **pitch deck** – documentation of your wireframes, user stories, and proposed architecture
  - Descriptions of any **unsolved problems** or **major hurdles** you had to overcome

---

## Suggested Ways to Get Started

- **Don't get too caught up in too many awesome features** – simple is always better. Build something impressive that does one thing well.
- **Design first.** Planning with user stories & wireframes before writing code means you won't get distracted changing your mind – you'll know what to build, and you can spend your time wisely by just building it.
- **Don't hesitate to write throwaway code** to solve short term problems.
- **Read the docs for whatever technologies / frameworks / API's you use.**
- **Write your code DRY** and **build your APIs RESTful**.
- **Be consistent with your code style.** You're working in teams, but you're only making one app per team. Make sure it looks like a unified effort.
- **Commit early, commit often.** Don't be afraid to break something because you can always go back in time to a previous version.
- **Keep user stories small and well-defined**, and remember – user stories focus on what a user needs, not what development tasks need accomplishing.
- **Write code another developer wouldn't have to ask you about.** Do your naming

conventions make sense? Would another developer be able to look at your app and understand what everything is?

- **Make it all well-formatted.** Are you indenting, consistently? Can we find the start and end of every div, curly brace, etc?
  - **Comment your code.** Will someone understand what is going on in each block or function? Even if it's obvious, explaining the what & why means someone else can pick it up and get it.
  - **Write pseudocode before you write actual code.** Thinking through the logic of something helps.
- 

## Potential Project Ideas

### Q&A App

Think of how helpful sites like Quora & StackOverflow are. Maybe there's some other niche, or some surprising twist you can add to the question-and-answer game.

### Car Repair Log

Keep up with your vehicle as you commute every day to your final weeks of WDI save things like mileage, last inspect, oil changes, and maybe even receipts & important document you need to have backed up.

### Email-like Messages

Imagine a realtime messaging client, but with private messages, an inbox, unread messages, and who knows what else. This is your chance to re-invent email as we know it.

---

## Resources

- [HackDesign](#) (*beginner's reference for thinking like a designer*)
  - [Visual Design Hacking](#) (*a great tips-and-tricks focused video from Front Row*)
  - [Web Design For Non-designers](#) (*another great design-related course for all the nerds out there*)
-

## Project Feedback + Evaluation

- **Project Workflow:** Did you complete the user stories, wireframes, task tracking, and/or ERDs, as specified above? Did you use source control as expected for the phase of the program you're in (detailed above)?
- **Technical Requirements:** Did you deliver a project that met all the technical requirements? Given what the class has covered so far, did you build something that was reasonably complex?
- **Creativity:** Did you add a personal spin or creative element into your project submission? Did you deliver something of value to the end user (not just a login button and an index page)?
- **Code Quality:** Did you follow code style guidance and best practices covered in class, such as spacing, modularity, and semantic naming? Did you comment your code as your instructors do in class?
- **Problem Solving:** Are you able to defend why you implemented your solution in a certain way? Can you demonstrate that you thought through alternative implementations? (*Note that this part of your feedback evaluation will take place during your one-on-one code review with your instructors, after you've completed the project.*)
- **Total:** Your instructors will give you a total score on your project between:

Score | Expectations ----- | ----- **0** | *Incomplete. 1 | Does not meet expectations.*  
**2** | *Meets expectations, good job!* **3** | *Exceeds expectations, you wonderful creature, you!*

This will serve as a helpful overall gauge of whether you met the project goals, but **the more important scores are the individual ones** above, which can help you identify where to focus your efforts for the next project!