Welcome to class!

AGENDA

- Feedback review
- Paired assignment retrospective and review
- Welcome to the HTTP Party
- Routing and Controllers

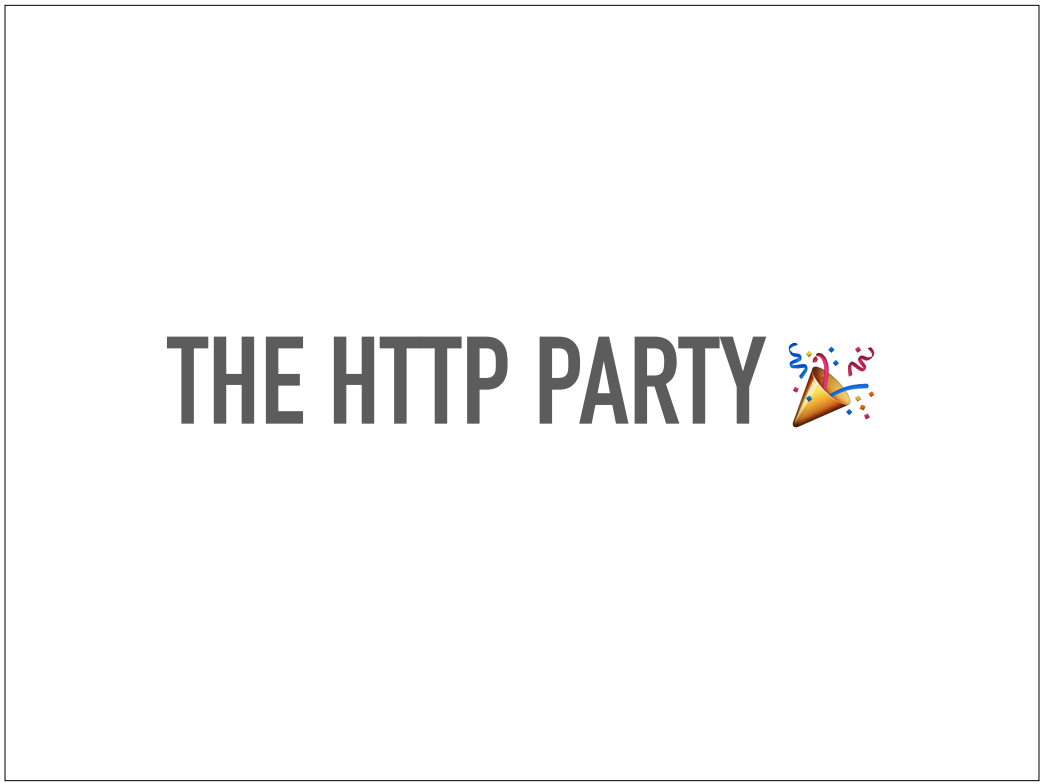
CLASS FEEDBACK REVIEW: MOST HELPFUL

- "The code review and pair programming are helpful and fun"
- "I think the stacked modules must be working because I feel more comfortable with concepts from last week or a couple days ago, even if I didn't understand them much initially."
- "Jonathan worked through some AJAX code"
- "Having to work with existing code was great."
- "The most hopeful thing we did all week was taking a day to not introduce something new on Thursday."
- "Thursday when we walked through how the example code was organized"
- "it was nice to have this past week be less work-stressful than the first week."

CLASS FEEDBACK REVIEW: NEEDS IMPROVEMENT

- "It'd be nice to have the TA's walk around a little more and see where students are at."
- "Demo code for the SQL day… wound up being more frustrating"
- "Monday…I felt like I wasn't properly equipped to handle the lab assignment."
- "This week the pair programming was really challenging for me."
- "User stories set unrealistic goals"
- "Maybe if there were a few mandatory (for grades) user stories, and the rest were for people who are doing great and want the extra challenge."
- "I hear everyone bashing on jQuery like it is the drunk inbred cousin of computer programming. He's invited to all the parties, but everyone makes fun of him! He passes out on the lawn, drooling, right next to PHP."

Notably, each topic (AJAX, SQL, FP) had a mix of people who enjoyed each, and who struggled with each.



You know that HTTP stands for “Head To This Place”, right? ;]

HTTP

- Hypertext Transfer Protocol
- “Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text.”
- HTTP is the foundation of data communication for the World Wide Web.
- HTTP is the abstraction layer between your browser and lower-level the Internet communication protocols (TCP/IP).

Thanks, Wikipedia: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

POWER TO THE BROWSER!

- Browser technology powers the World Wide Web
- Let's take a peek under the hood!
- What powers browser?

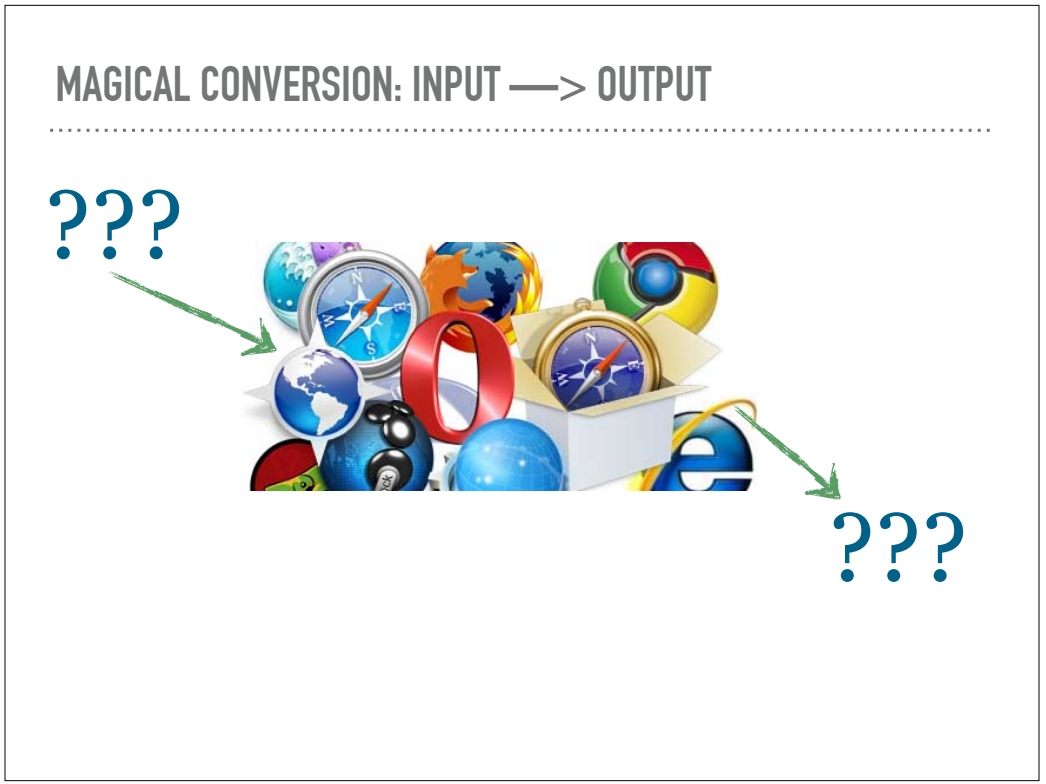
We are here to talk web applications. For the most part, the action happens in the browser.

This amazing program that runs on your computer can reach out to destinations all over the world, and in a matter of seconds, show you news, video, images, stories, cat pictures. Amazingly helpful tools to help you connect, boost a business, or caption cats, all run in this generic container we call a browser.

So let's take a look at how that happens, on a high level.

We'll dive right in to the acronym soup with URLs, DNS, HTTP, and how the concept of abstraction makes it easier than ever to make it all happen.

Image: http://napkindiagrams.files.wordpress.com/2010/08/browser_engine2.png?w=630



Whatever the browser, they are all designed to handle the same basic input. And, based on that input, we expect the same output.

What is the input? A single URL.
What is the output? A rendered web page.

The URL tells the browser a lot about how to make that conversion happen:
* How to ask
* Who to ask
* Where to ask
* What to ask for

Image: http://www.vanseodesign.com/blog/wp-content/uploads/2011/08/strucbrowser-logos.jpg

URLWTF

- Some URLs are nice:

http://twitter.com/brookr

https://www.codefellows.org/blog.html?page=2

- Some URLs are not:

https://plus.google.com/u/0/110552115013326646668/posts

http://www.amazon.com/dp/B00VKIY9RG/
ref=s9_acsd_bw_dcd_odsbncat_c6_pd?
pf_rd_m=ATVPDKIKX0DER&pf_rd_s=merchandised-search-2

You can make a reasonable guess at what some URLs will show you.

Others: not so much.
Can you guess what this last one is for? It's Amazon's flagship Kindle Fire HD10 tablet.

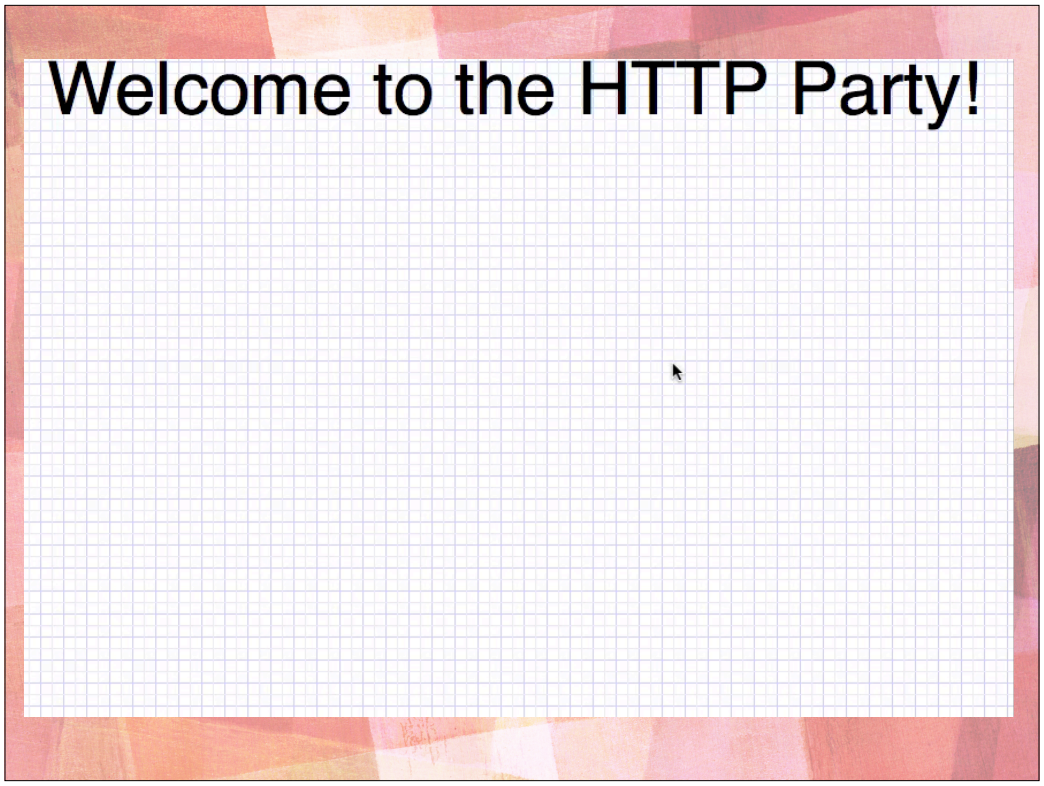
URLWTF: BREAKDOWN

- Ugly or nice, they can all have the same parts:
  - protocol: https://
  - [sub]domain: www.codefellows.org
  - path: /blog
  - format: .html
  - named anchor: #today
  - parameters: ?page=2&admin=true

https://www.codefellows.org/blog.html#today?page=2&admin=true

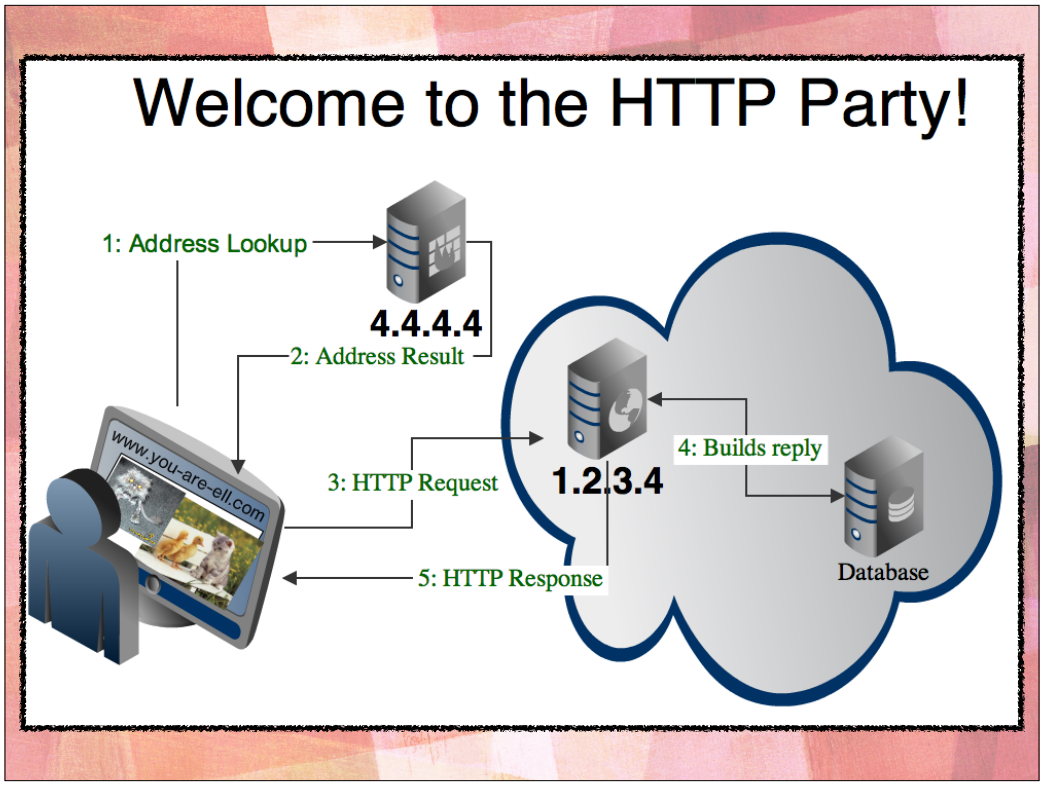
These pieces give the browser all the info needed to get that web page.

How: Using the HTTPS protocol.
From whom: This specific sub-domain of this domain.
From where at that domain: At this path.
What are we looking for: This format, modified by these paramaters.



This is how your URL gets converted into web page contents.
You sit down at your desk, fire up a web browser, and type in a URL.
Your browser needs to communicate with your OS to talk to a DNS server.

Then it can send a REQUEST to the server.
The server may want to talk to other computers, to find the info you asked for.
The server then sends back a RESPONSE.



This is how your URL gets converted into web page contents.

THE DNS DANCE

- Talk to a known DNS Server
  - Global index of domain names
- Give a name
- Get an IP address
- Now ready to ask for that web page!

To make that happen, the browser needs the domain name converted into a numeric address.

EXAMPLE: Like looking up someone's phone number in the phone book. You know where to look, you know their name. The book tells you the number to reach them directly.

If you've ever registered your own domain name before, setting up DNS is part of the process. This is why: people need to be able to find it.

THE HTTP PARTY

- The browser creates a HTTP Request Object
- HTTP Request has 3 parts:
  - URL (twitter.com)
  - Method (GET)
  - Headers (sender info: user agent, cookies, etc)

Remember these 3 parts! You'll need them later.

Image: http://rdr.zazzle.com/img/imt-prd/pd-172254667462737476/isz-m/tl-R.S.V.P+Wedding+Postage+Stamp.jpg

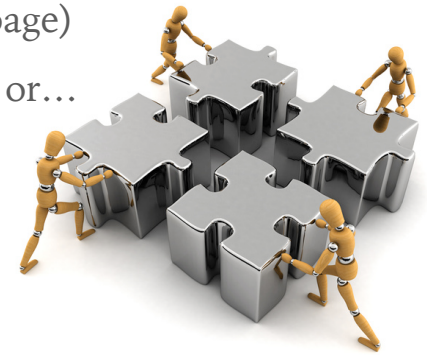
THE HTTP PARTY

- The server:
  - receives the request
  - builds a response
  - sends it back to the clie

The server's response could vary greatly, from a simple error message, to a plain static file, to a dynamically generated page that matches the unique user's unique and wonderful request.

THE HTTP PARTY: STATUS CODES

- An HTTP Response also has 3 parts:
  - Status code (200, 301, 404, 500, etc)
  - Headers (info about the server & file sent)
  - Body (the content of the page)
    - HTML, CSS, JavaScript, or…



The response is stateless: It's simply a reply to the request. It doesn't consider the requestor's history, or who they are, or where they come from. Some of that info can be used to build the response, but only if it was included in the request.

Anyone know what any of these codes mean?

THE HTTP PARTY: STATUS CODES

- See: http://httpstatusdogs.com/
- See also: https://http.cat/

Take a look at some status codes…

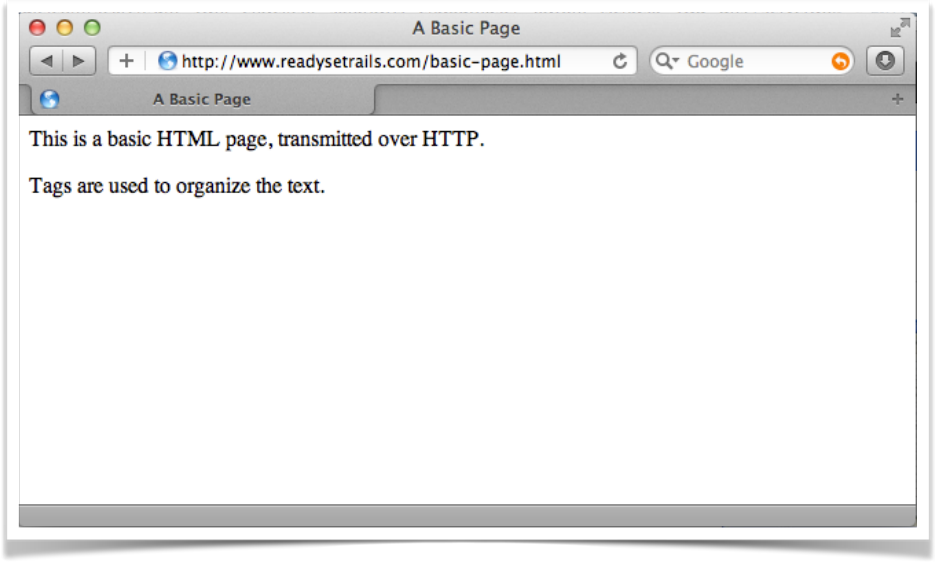
The server's response to an initial request is often an HTML file. It's a Markup language!

The HTML that comes back from the server is a set of instructions that tells the browser how to display the content.

This file is filled with link tags and script tags that require the browser to make dozens more requests.

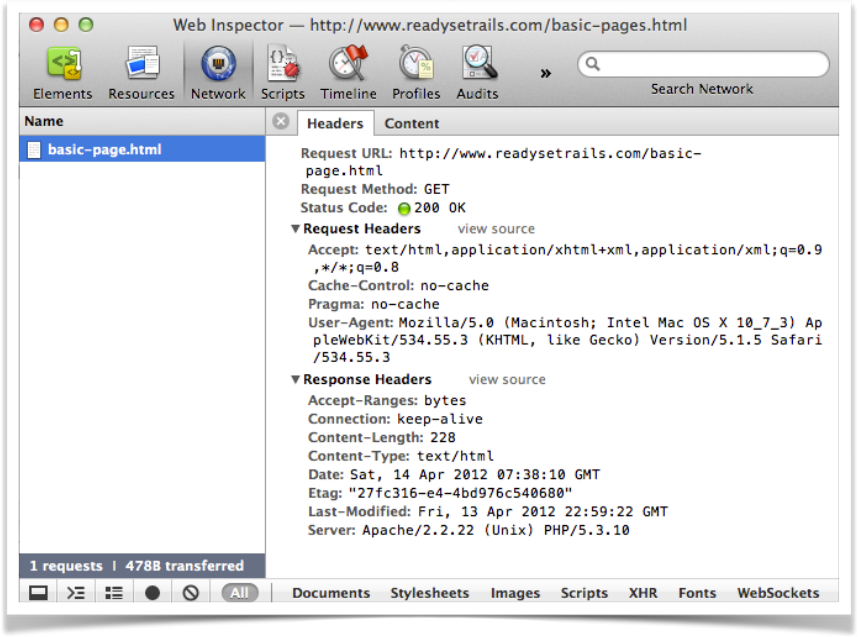
The browser fetches these individually, or in parallel… You may have seen: "Loading 34 of 44…" in the status bar.

THE HTTP PARTY: EXAMPLE



Let’s break it down in slo-mo. Here’s a browser loading a simple HTML page.

THE HTTP PARTY: EXAMPLE



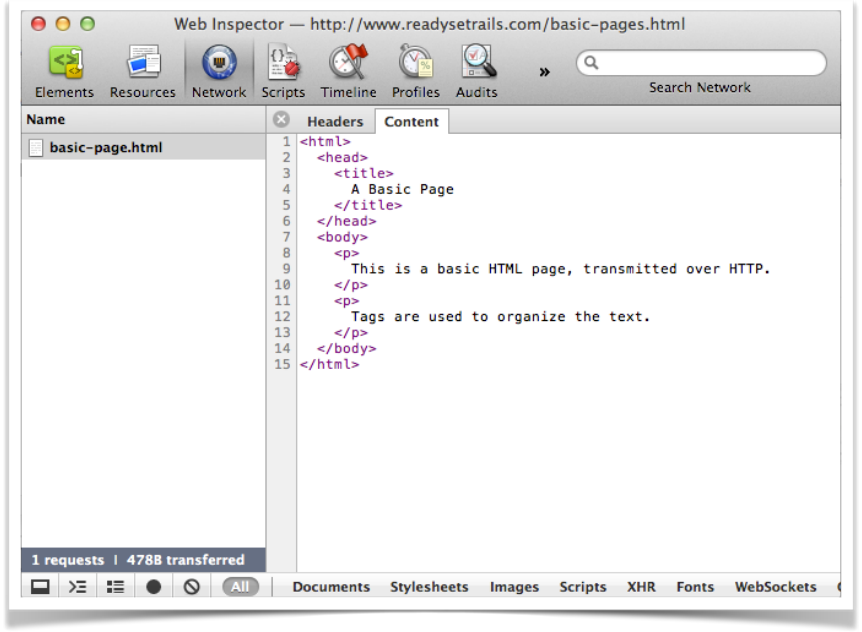
Let's take a look behind the scenes, and see what actually happened there.

What were the 3 parts of the request again? And the response?

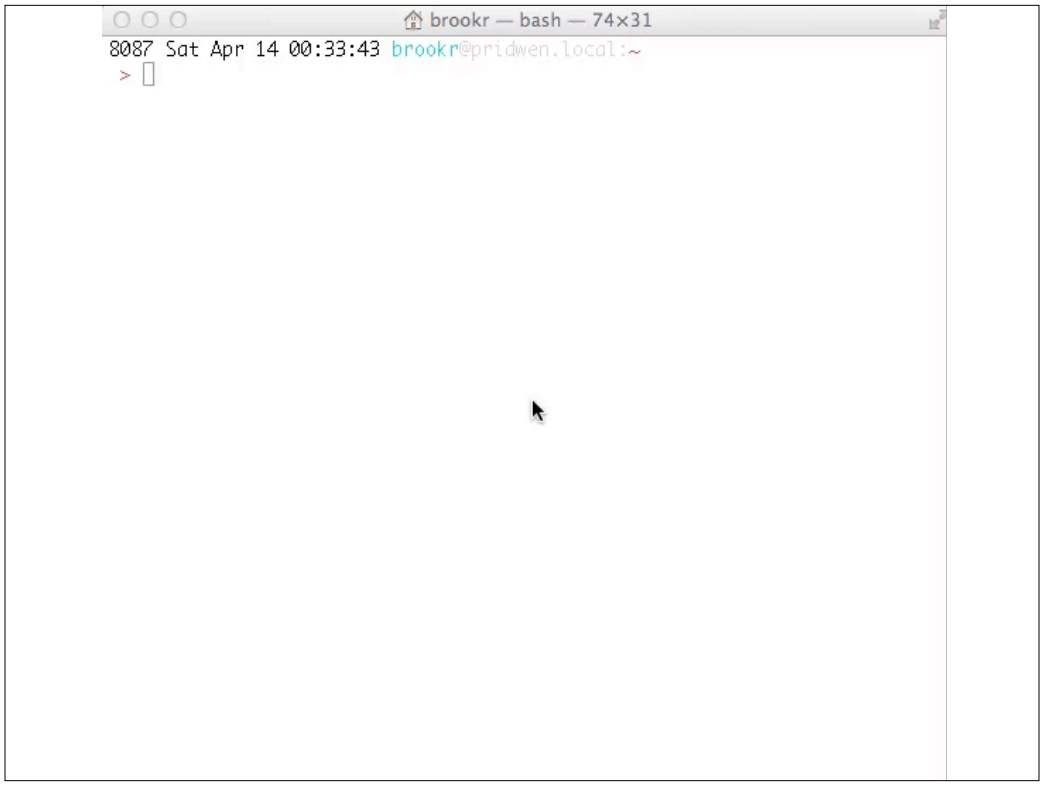
We see the request parts: URL, Method, Headers

We see the response parts: Status, Headers, Content

THE HTTP PARTY: EXAMPLE



Content in this example is HTML.



One more look: the bare bones of what's happening. Watch as we send a request and receive a response, all directly from the terminal. Your assignment today is… just kidding!

We don't need to do our web browsing this way, because when we use a graphical browser, these details are abstracted away.

It turns out this is a very important concept in programming.

See screencast at: http://dl.dropbox.com/u/635532/ReadySetRails/RSR-HTTP_Party-Manual_web_browsing.mov

DEMO: TRY IT OUT

- Go to a site you are signed in to
- Inspect it: Explore the Inspector tabs
- Identify what is going back and forth
- Figure out how to delete cookies. What happens?

We said that HTTP requests are stateless, but most web apps seem to have an awareness of who you are. How do they do that? With cookies of course!

Challenge: Find the cookies for a site you are currently signed in to. Delete them and see what happens.

Challenge: How many requests does your blog page make right now? How might you reduce the number of requests, in production?

HTTP: A LAYER OF ABSTRACTION

```
80007214 <dcache_init>:
:
800072ce:           2f 23 c2 00         232f00c2     lsr        r3,r3
800072d2:           40 22 42 00         22400042     add        r2,r2,1
800072d6:           4c 23 00 80         234c8000     cmp        r3,0
800072da:           f6 07 e2 ff         07f6ffe2     bnz.d      800072ce <dcache_init+0xba>
800072de:           0a 24 80 00         240a0080     mov        r4,r2
:
```

- Computers really only speak binary
- Thanks to the computer science concept of "Abstraction"…
- You don't have to. Yay!

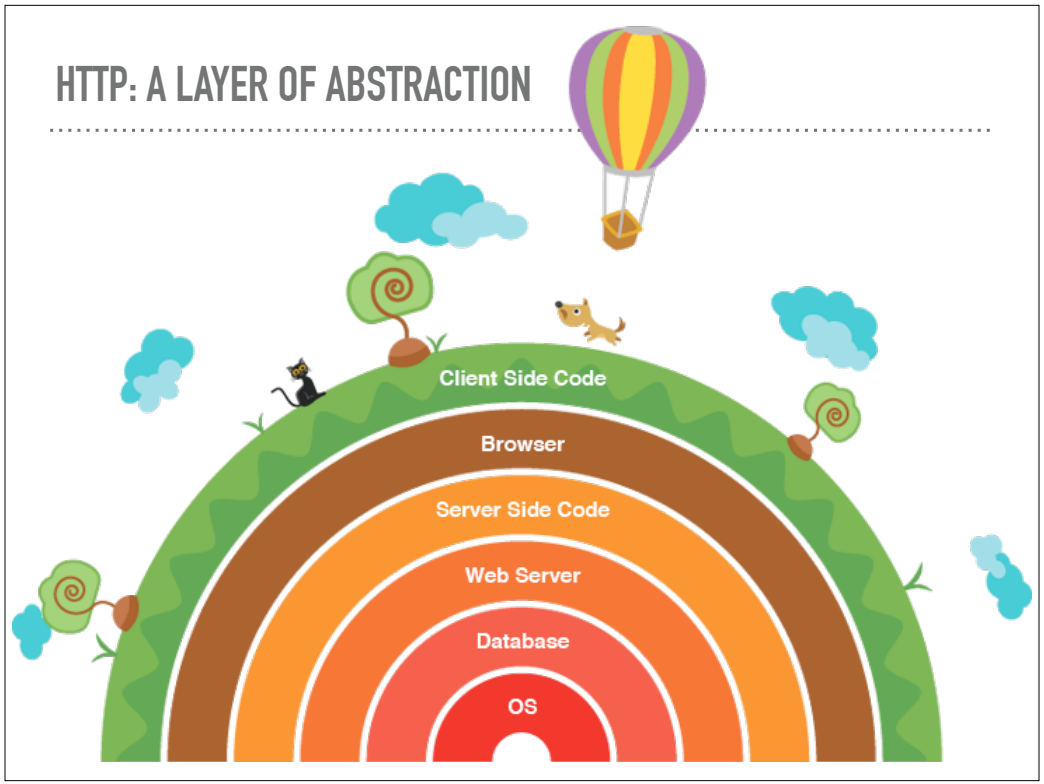
Early on, computer scientists decided they didn't want to use punch cards forever.

That's the old way people told a computer to do something, from the automated loom days. A big box of index cards, carefully hole-punched just so, and carefully sorted in just the right order.

Machine Language: specific binary commands to directly control the CPU.
Assembly provided a shorthand for Machine Language. "More readable", they say.
The C language set the standard for how to make a usable language. It is "compiled" down to assembly code for any give CPU instruction set.
Using C, many other languages have be written that are even easier to read for humans, and still executable by computers after being parsed, interpreted, compiled, assembled and finally converted back to binary.

So: See what's happening? At any given level, you don't have to know or care about what's happening further down the stack.

Image: http://groups.inf.ed.ac.uk/pasta/images/asm_code.png



The same is true not just for programming languages, but also for web technologies (and everything else around us).

A consulting firm, Schecter & Co created this image. Let's look at it form the perspective of abstraction.

OS abstracts away all the details of how a computer runs: how it saves and retrieves files, executes programs, moves the mouse pointer, connects to the network, etc etc etc. We get to not care! Yay!

We are operating in the green and brown band. We don't care about anything below.

The back-end of a web app could run on many web servers, with many different database options (or even multiple dbs within the same app), and on any major OS.

The front-end cares about the browser (is it mobile? is it IE?), and even backend apps need to generate the client side code (HTML, CSS, JS) customized for every request.

As far as the user is concerned, the browser abstracts away the details about what we are doing. And we need to do our work to ensure that they don't have to care.

ROUTING & CONTROLLERS

> The controller in a web app is a bit more complicated, **because it has two parts**. The first part is the web server (such as a servlet container) that **maps incoming HTTP URL requests to a particular handler** for that request. The second part is those handlers themselves, which are in fact often called “controllers" [or actions].
>
> So the C in a web app MVC includes both the web server "overlord" that routes requests to handlers and the logic of those handlers themselves, which **pull the data from the database and push it into the template**.
>
> *-Terence Parr*

From an interview: http://www.artima.com/lejava/articles/stringtemplate.html

ROUTES: YOUR PUBLIC API

- What resources does your app offer?
- Abstract away the details of html files
  - https://www.codefellows.org/blog.html
  - https://www.codefellows.org/blog/
  - Same page!

With a Router, we don't need to have a literal file for every single URL that our app can respond to.

ROUTES: YOUR PUBLIC API

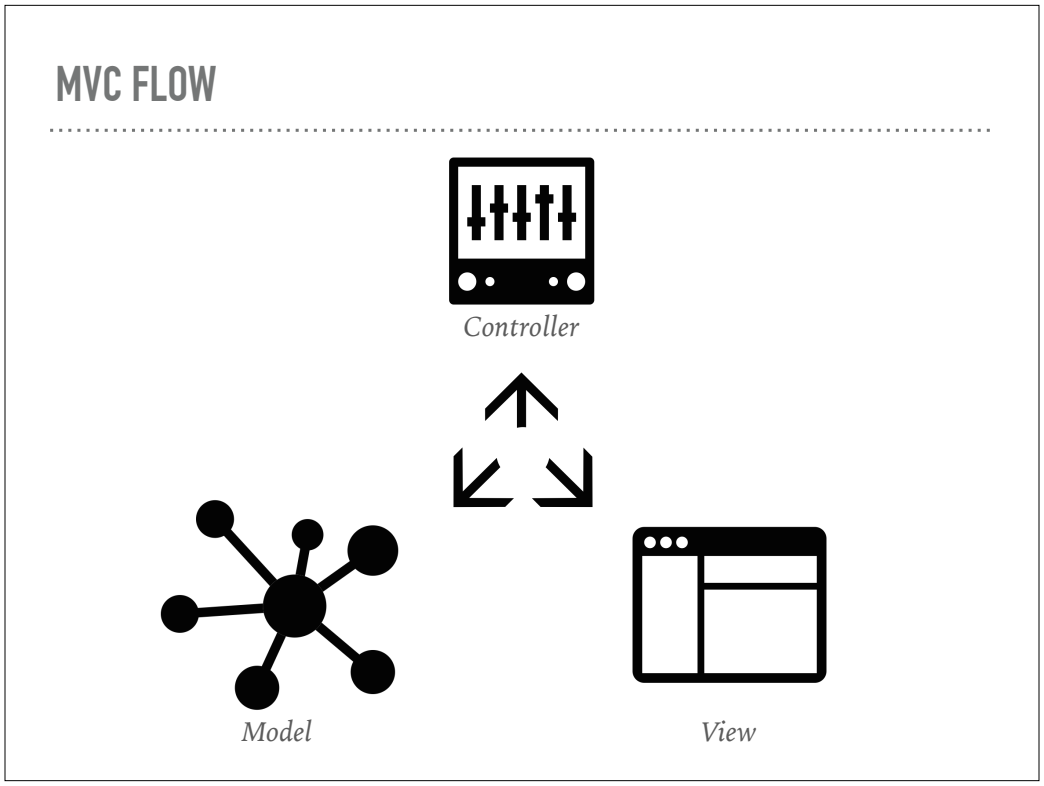
- Making files is slow
- We want programmatic control of what our app can do
- GET: /
- GET: /about
- GET: /articles/42
- GET: /articles/42/edit
- PUT: /articles/42

Our app's routes are the only way the world can interact with the resources we have.

CLIENT-SIDE ROUTING

- Client-side, the route can tell us a few things:
  - What resource the user wants:
    - hipmunk.com/flights
  - Additional info:
    - hipmunk.com/flights#f=SEA;t=HNL;d=2015-12-22
  - JavaScript can interpret this route, and…
    - take apart the pieces
    - call the proper function to handle it all
    - …all from a single index file: NO RELOAD!

Not reloading the page when clicking a link means that our app is FAST. No need to make a request/response cycle, as all the logic to handle the new page is



Remember this?

CLIENT-SIDE CONTROLLER

- Our controller will handle the user request
- The controller converts a route into displayed content…
- with the proper data load.
- Simply a list of functions (aka: “actions”), waiting to be called
- One controller per resource:
  - ArticlesController
  - FlightsController
  - UsersController

CLIENT-SIDE ROUTING: HELPFUL LIBRARIES

- page.js
  - Connect routes with handling function:
    - `page('/', user.list)`
    - `page('/', index);`
    - `page('/about', about);`
    - `page('/contact', contact);`
  - Many many more examples:
    - https://github.com/visionmedia/page.js
    - Install: copy/paste `page.js` file into your project

page.js will hook into the History API to capture clicks, call the right handler function, and push state to the address bar.

CLIENT-SIDE ROUTING: HELPFUL LIBRARIES

- `pushstate-server`
  - Sends all requests to: `index.html`
  - Passes through static files:
    - `/scripts/blogArticles.json`
    - `/templates/article.html`
  - Requires full-path URLS, with starting slash
  - More details: https://github.com/scottcorgan/pushstate-server
  - Install: `npm install -g pushstate-server`

ROUTING DEMO

Using `page()` library
- Create a router.js file
- Add: page('/', articlesController.index);
- Create a articlesController.js file
- Make an index action that will load the data, call the view.

RECAP

RECAP

- URLs power the browser
- JavaScript can leverage the URL to control the app, without going back and forth with a server
- Assignment:
  - Work from assignment directory code
  - Add a router
  - Add a controller for articles