

# CS 290B

## Scalable Internet Services

Andrew Mutz

November 17, 2014



# Today's Agenda

- The Client-side Renaissance



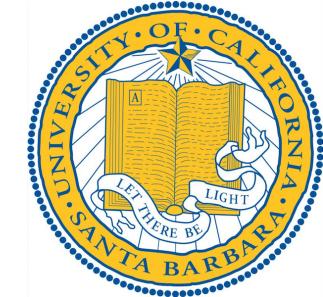
# Motivation

- <http://backbonejs.org/examples/todos/>



# The Browser Wars

- Mid-90s Netscape reigns supreme
- Microsoft releases initial version of Internet Explorer in 1995
- Competition between Netscape and Microsoft produces significant innovation in browsers
  - Javascript
  - Cookies
  - CSS



# The Browser Wars

Microsoft bundles Internet Explorer to Windows 98

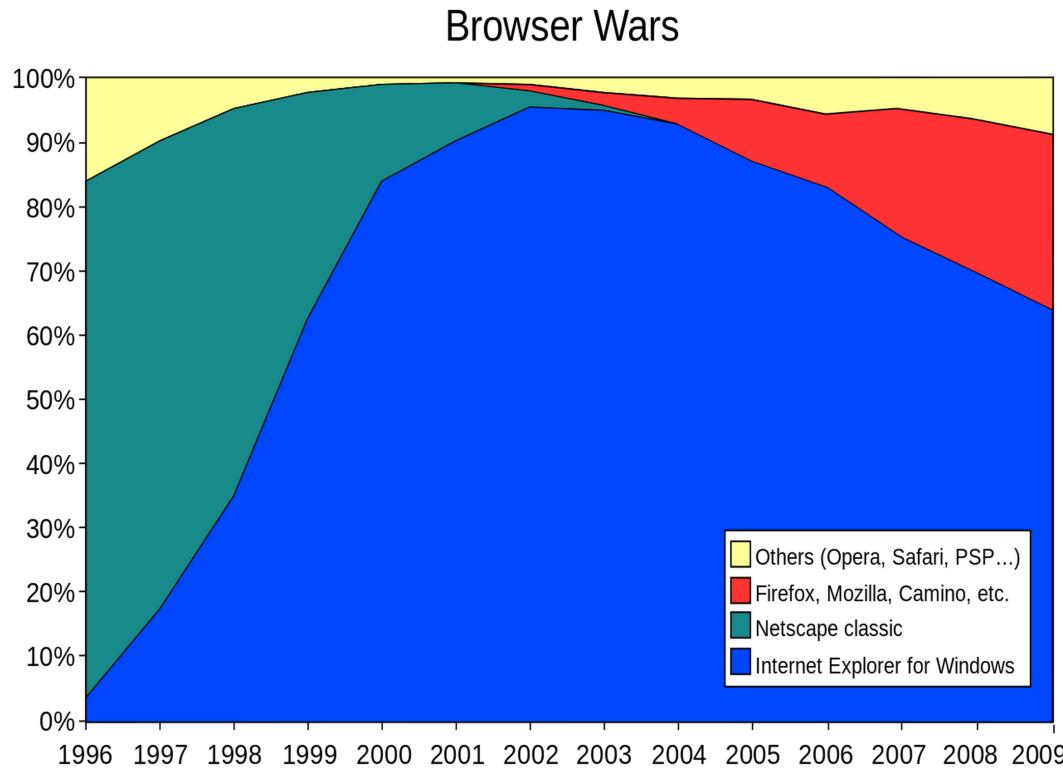
- Every file management window is a browser
- Eventually triggers an antitrust lawsuit against Microsoft

Meanwhile, Netscape focuses on open-sourcing it's browser

- Eventually creates the Mozilla foundation
- Acquisition by AOL



# The Browser Wars



Microsoft wins.  
Internet Explorer  
becomes the  
dominant browser  
for roughly a decade.



# The Browser Wars

Version	Release Year
IE 1	1995
IE 2	1995
IE 3	1996
IE 4	1997
IE 5	1999
IE 6	2001
IE 7	2006
IE 8	2009

Lull in browser innovation commences

- Lack of competition means no reason to innovate
- Time between releases increases significantly

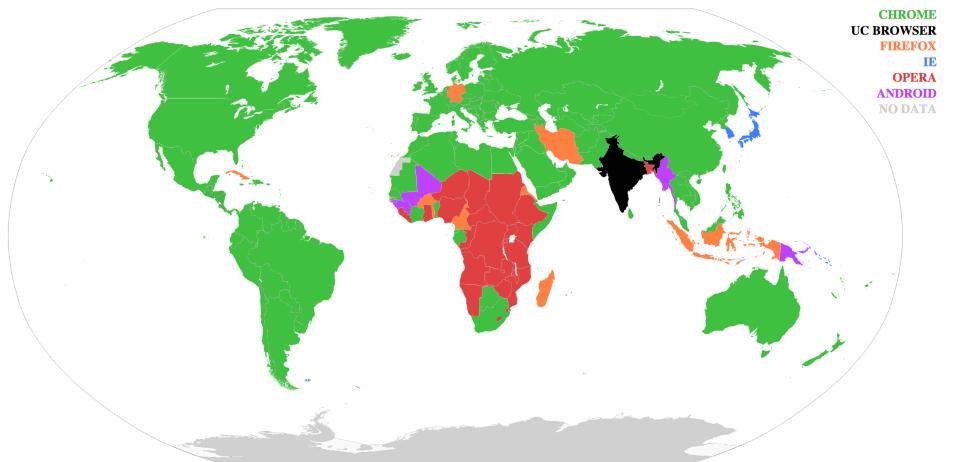
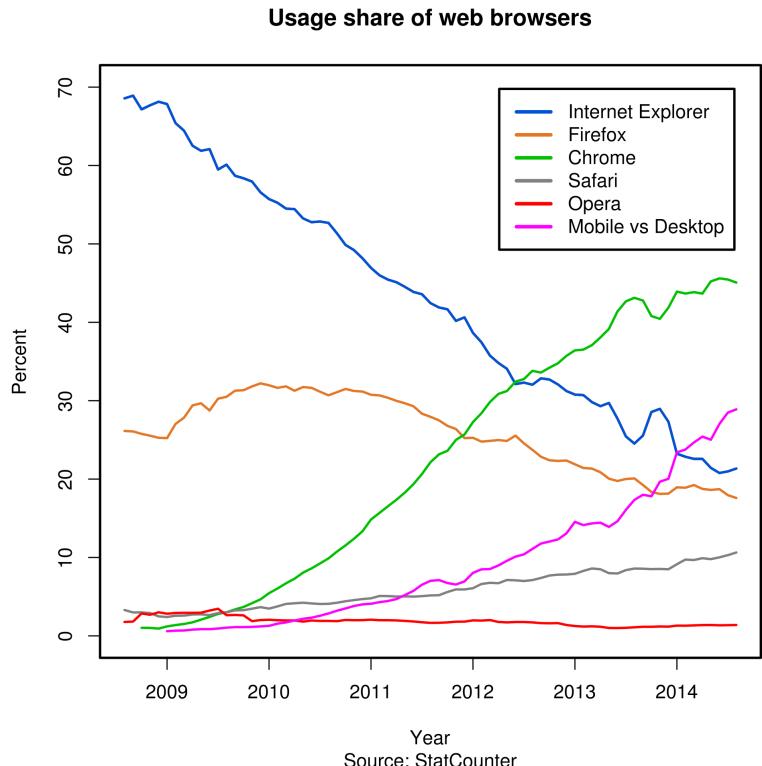


# The Browser Wars

- Due to a variety of factors, Microsoft slowly loses market share to Firefox (Mozilla)
  - Security
  - Performance
- As Microsoft is slowly bleeding market share, Google announces Chrome
- Browser innovation reignites, today there are at least 4 viable browsers.



# Browser Market Share Today



Source: [http://en.wikipedia.org/wiki/Usage\\_share\\_of\\_web\\_browsers](http://en.wikipedia.org/wiki/Usage_share_of_web_browsers)



# Client-side renaissance

During the browser dark ages, three things eventually spur the client side renaissance

- XMLHttpRequest
- DOM Manipulation
- V8



# XMLHttpRequest (Ajax)

Allows JavaScript on the page to asynchronously request resources from the server.

```
var req = new XMLHttpRequest();
req.onload = function() {
    console.log("I'm a callback!");
};
req.open("get", "/comments", true);
req.send();
```

Originally added to IE to enable the Outlook Web Access team to asynchronously communicate with the server (year ~2000).

- Other browsers implement it, becomes de facto standard by ~2004.



# XMLHttpRequest (Ajax)

Prior to this, two-way communication with the server meant a full page refresh

- Either a clicked link or a submitted form.

This method allows javascript in your browser to send requests and receive responses completely programmatically.

- Example: As you type, google.com will show you intermediate results.

Most people don't use this method directly. jQuery is commonly used as an abstraction layer:

```
$.get('/comments', function(data) { alert("I'm a callback!");});
```



# XMLHttpRequest (Ajax)

## Security limitation:

- These requests can only go to the originating domain that the Javascript was served from.
- This prevents randomsite.com from (say) accessing your gmail inbox
  - All requests use existing cookies, of which your session is one.
- This works pretty well for things like Google Analytics, they can phone home to Google, but can't access the server-side resources of the website they are on.

This technique is now referred to as Ajax

- Asynchronous JavaScript and XML
- XML was originally envisioned as the transport format, but there is nothing XML-specific about it.
- Today, XML is not as regularly used as JSON for transport.



# DOM Manipulation

- Document Object Model
  - A standardized way of representing the structure of a web page as a tree of in-memory objects
  - These objects are accessed via Javascript and can be queried and manipulated
- Example:

```
var newDiv = document.createElement("div");
var newContent = document.createTextNode("Hello World!");
newDiv.appendChild(newContent);
document.body.appendChild(newDiv);
```



# DOM Manipulation

## Progression of DOM Manipulation:

- DOM Level 0: Navigator 2, IE 3 (~1995).
  - Allowed reading the values of forms and links.
  - Not standardized (called legacy DOM)
- DOM Level 1: Navigator 4, IE 5 (~1998)
  - Allowed access and modification of anything by index
    - `document.forms[1].elements[2]`
- DOM Level 2: ~ 2000
  - added `getElementById()`, DOM event model
- DOM Level 3: ~2004, current
  - XPath support, keyboard event handling



# V8: Javascript Gets Fast

September 2008, Google releases Chrome

- In addition to other novel features, it includes the V8 Javascript engine
- V8 applies modern, state of the art VM techniques to Javascript
  - Not interpreted: dynamically compiled to machine code
    - Re-compiled and re-optimized at runtime
  - Garbage collector is fast
    - Generational: separates allocated memory into young and old groups, treats them differently
    - Incremental: doesn't need to perform all GC at once
  - Applies other modern VM optimizations:
    - inlining, code elision, inline caching



# V8: Javascript Gets Fast

V8 treats Javascript performance seriously, and triggers other browsers to do the same.

- Safari's JavaScriptCore
- IE's Chakra
- Firefox's SpiderMonkey

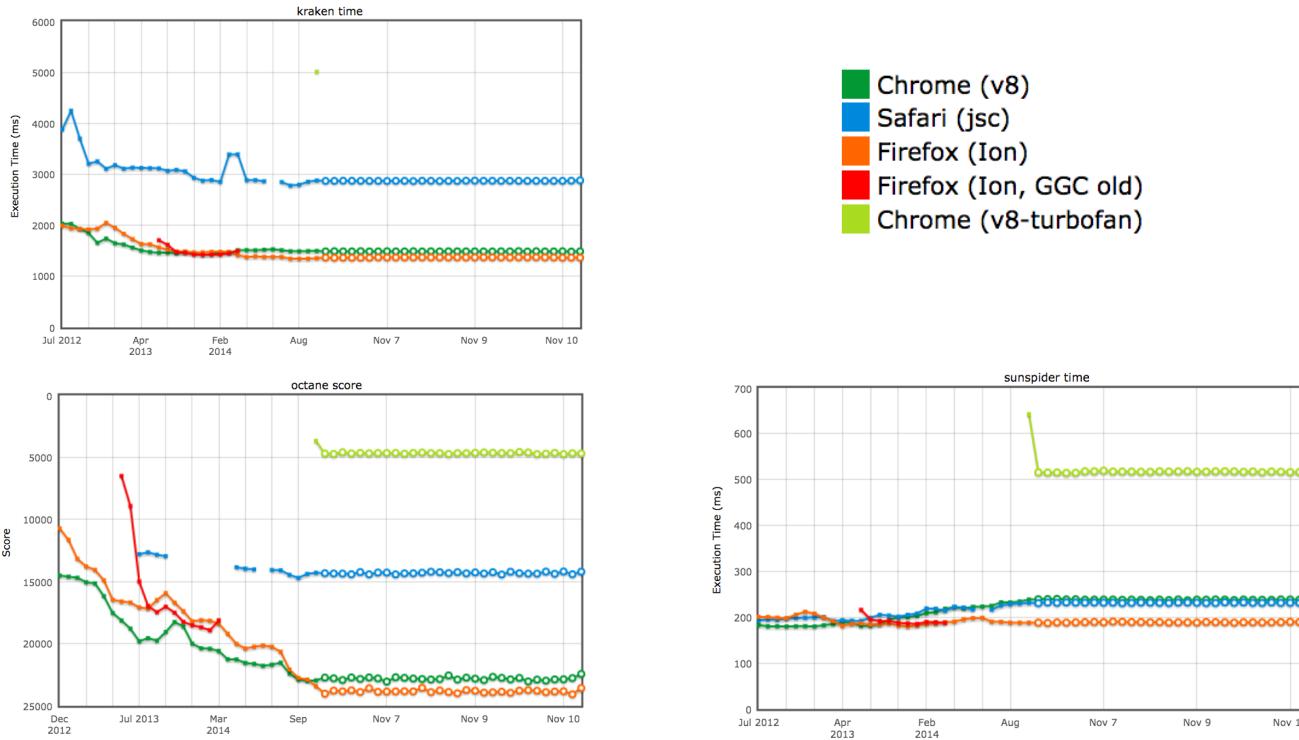
Today these VMs are all roughly evenly matched.

- Performance leader goes back and forth, but in the same ballpark

Sidenote: V8 was designed to also work well outside of the browser. It is the execution engine that Node.JS is built on.



# V8: Javascript Gets Fast



Source: <http://arewefastyet.com>



# Client-side Renaissance

So, by 2008, we have all the ingredients ready for a client-side renaissance:

- Globally installed virtual machines
- ...that can present content that can communicate via Ajax to the internet service they originated
- ...that have full programmatic control of the user interface (DOM).
- ...that use modern, high performance VM techniques
- ... that exist in a competitive marketplace
  - Four viable browsers, available on multiple OSes
  - Competing to stay ahead of the pack
  - Standards-compliance is a competitive advantage

These are things we enjoy today that we did not always have.



# Client-side Renaissance

What do these modern client-side applications look like?

- Instead of being a series of pages requested from a web server, we can serve up a running javascript application
  - This application is regularly sending user input back to the server
  - This application is regularly receiving structured data instead of rendered markup.
- These applications generally persist through user interactions
  - Clicks don't necessarily mean full-page refreshes
- Communication with the server is decoupled from user interaction
  - While the browser sits open, a javascript timer can go and check for new data and update the page as needed



# Client-side Renaissance

## Consequences of this shift

- Client side logic is much more complex and full page refreshes are more rare
- It's possible to build applications that work "offline"
- It's possible to build effective "push" mechanisms
- The "running application" is much more static and cacheable
- The apis you build to serve up structured data can be used by mobile applications and other internet services
- "Real" Javascript VMS enable very ambitious use of CPU resources



# Client-side Renaissance

Client side logic is much more complex and full page refreshes are more rare	Client-side MVC, Today
It's possible to build applications that work "offline"	HTML5 local storage APIs, Today
It's possible to build effective "push" mechanisms	HTML5 WebSockets, future lecture
The "running application" is much more static and cacheable	"How to build a Content Distribution Network," next Tuesday, Nov 25
"Real" Javascript VMS enable very ambitious use of CPU resources	"ASM.js and Emscripten", Dec 2
The apis you build to serve up structured data can be used by mobile applications and other internet services	"REST, JSON, and Optimizing for Mobile", Dec 4

# Client-side Renaissance

So you're interested in building a web application that moves much of its logic and rendering to the client...

Let's look at an example of this transition, in our LabApp.



# Client-side Renaissance

## In a traditional web application:

- When you click the “New Submission” button, the browser makes a new HTTP request and loads the response
- The response is an entire web page, and with it are numerous assets.
- The page returned has form elements
- When you fill out the form and submit it, the server may find it invalid and send back another form.
- List of submissions only changes when you refresh the page.

Bboetest    Communities    New Community

### All submissions

Title	Url	Community
Use of GOTO considered harmful	<a href="http://www.google.com">http://www.google.com</a>	Programming>Show>EditDestroy
A Mathematical Theory of Communication	<a href="http://bell-labs.com">http://bell-labs.com</a>	Programming>Show>EditDestroy
Gardening with saltwater: Why isn't it working?	<a href="http://www.wikipedia.com">http://www.wikipedia.com</a>	Gardening    Show>EditDestroy

New Submission    Communities  
[Log In](#) | [Sign Up](#)

Bboetest    Communities    New Community

### New submission

Title

Url

Community

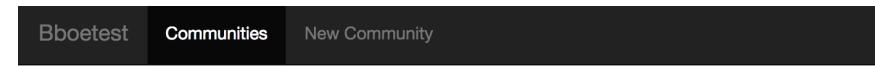
[Log In](#) | [Sign Up](#)



# Client-side Renaissance

In a client-side web application:

- When you click the “New Submission” button, javascript executes and redraws the page to show a form. No HTTP requests occur.
- When you fill out the form and submit it, the input is validated in the browser using javascript.
- If valid, an Ajax request is sent to the server
- List of submissions only changes live.



## All submissions

Title	Url	Community
Use of GOTO considered harmful	<a href="http://www.google.com">http://www.google.com</a>	Programming>Show>Edit>Destroy
A Mathematical Theory of Communication	<a href="http://bell-labs.com">http://bell-labs.com</a>	Programming>Show>Edit>Destroy
Gardening with saltwater: Why isn't it working?	<a href="http://www.wikipedia.com">http://www.wikipedia.com</a>	Gardening>Show>Edit>Destroy



## New submission

Title

Url

Community

[Create Submission](#)

[Back](#)  
[Log In | Sign Up](#)



# Client-side Renaissance

## Benefits:

- UI is extremely responsive
- Less network traffic
- Live updates!

Bboetest   [Communities](#)   [New Community](#)

## All submissions

Title	Url	Community
Use of GOTO considered harmful	<a href="http://www.google.com">http://www.google.com</a>	<a href="#">Programming</a> <a href="#">Show</a> <a href="#">Edit</a> <a href="#">Destroy</a>
A Mathematical Theory of Communication	<a href="http://bell-labs.com">http://bell-labs.com</a>	<a href="#">Programming</a> <a href="#">Show</a> <a href="#">Edit</a> <a href="#">Destroy</a>
Gardening with saltwater: Why isn't it working?	<a href="http://www.wikipedia.com">http://www.wikipedia.com</a>	<a href="#">Gardening</a> <a href="#">Show</a> <a href="#">Edit</a> <a href="#">Destroy</a>

[New Submission](#)   [Communities](#)  
[Log In](#) | [Sign Up](#)

Bboetest   [Communities](#)   [New Community](#)

## New submission

Title

Url

Community

[Create Submission](#)

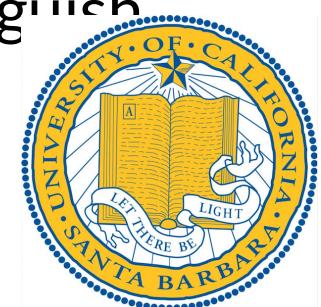
[Back](#)  
[Log In](#) | [Sign Up](#)



# Client-side Renaissance

Client side code is much more complex

- Before, the client was mostly displaying a static page.
- Now, the client must:
  - Understand the relationship between input events and corresponding DOM updates
  - Understand enough application logic to distinguish valid input from invalid input
  - Keep a persistent connection to the server and display updates as they come in.



# Client-side Renaissance

How should our application design adjust to this drastic increase in complexity?

- One design approach: “A tangled pile of jQuery selectors and callbacks, all trying frantically to keep data in sync between the HTML UI, your JavaScript logic, and the database on your server.”



# Client-side MVC

How should our application design adjust to this drastic increase in complexity?

- ~~One design approach: “A tangled pile of jQuery selectors and callbacks, all trying frantically to keep data in sync between the HTML UI, your JavaScript logic, and the database on your server.”~~
- One popular approach to managing this complexity is the use of MVC frameworks on the client.



# Client-side MVC

## MVC: Model-View-Controller

- You all know it from Rails
- Separates the presentation (View) of your data from the data itself (Model).
- Controller exists to accept and coordinate updates to the Models.
- Models encapsulate business logic and state.

There are many client side frameworks that implement variations of these. Today we will talk about three:

- Backbone
- Angular
- Ember



# Client-side MVC

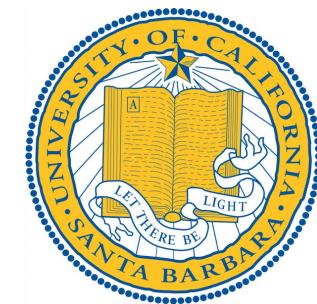
## Backbone.js

- Developed by Jeremy Ashkenas (creator of CoffeeScript and Underscore.js)
- Most lightweight of the libraries we will be discussing today
- Really Model-View-Router
  - Router maps URL fragments to functions



## Popular websites using Backbone:

- Airbnb, Hulu, Groupon, Pinterest, LinkedIn



# Client-side MVC

## Backbone Model Sample

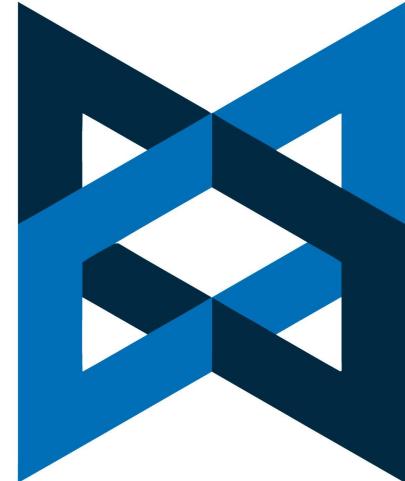
```
var Sidebar = Backbone.Model.extend({
  promptColor: function() {
    var cssColor = prompt("Please enter a CSS color:");
    this.set({color: cssColor});
  }
});
window.sidebar = new Sidebar;
sidebar.on('change:color', function(model, color) {
  $('#sidebar').css({background: color});
});
sidebar.set({color: 'white'});
sidebar.promptColor();
```



# Client-side MVC

## Backbone View Sample

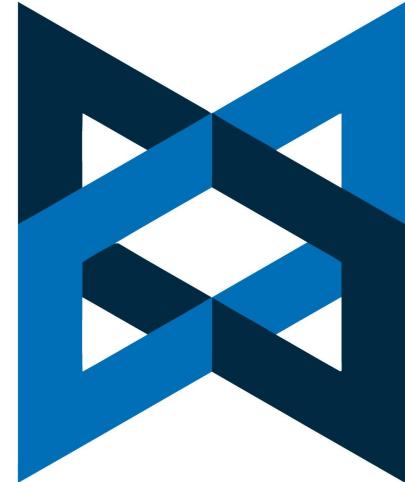
```
var DocumentRow = Backbone.View.extend({  
  tagName: "li",  
  className: "document-row",  
  events: {  
    "click .button.edit": "openEditDialog"  
  },  
  initialize: function() {  
    this.listenTo(this.model, "change", this.render);  
  },  
  render: function() {  
    this.$el.html(this.template(this.model.attributes));  
    return this;  
  }  
});
```



# Client-side MVC

## Backbone Router Sample

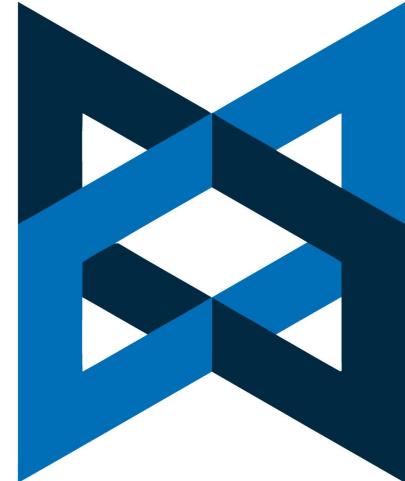
```
var Workspace = Backbone.Router.extend({  
  routes: {  
    "help": "help", // #help  
    "search/:query": "search", // #search/kiwis  
    "search/:query/p:page": "search" // #search/kiwis/p7  
  },  
  help: function() {  
    ...  
  },  
  search: function(query, page) {  
    ...  
  }  
});
```



# Client-side MVC

## Backbone Highlights:

- Lightweight (1700 lines)
- Templating agnostic
- Doesn't handle unbinding events
  - Can lead to memory leaks
- For full-blown Single Page Apps, you might be better off with one of the larger frameworks



# Client-side MVC

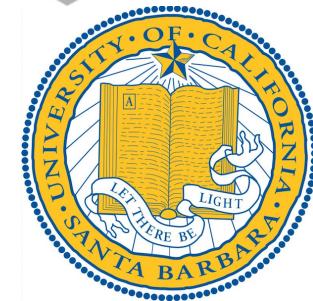
## Angular.js

- MVC framework supported and promoted by Google.
- Much larger and more complex than Backbone
  - Responsible for much more
- Suitable for Single Page Applications
- Emphasis on declarative style for building UI
- Uses two-way data binding



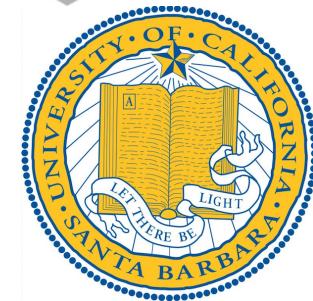
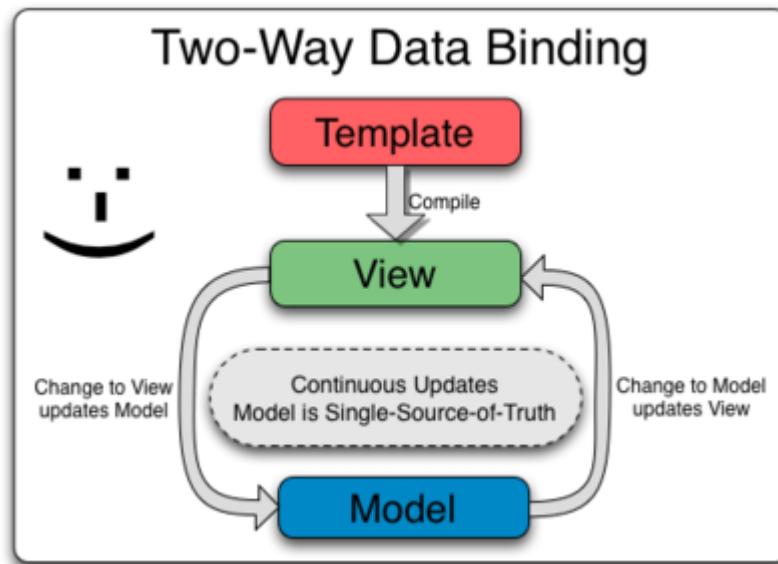
Notable websites using Angular:

- AWS console, HBO, VirginAmerica



# Client-side MVC

Two-way data binding:

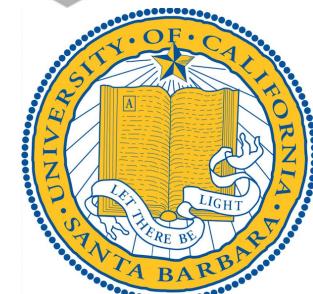


# Client-side MVC

## Data Binding Example:

```
<div ng-app ng-init="qty=1;cost=2">  
  <div>  
    Quantity: <input type="number" min="0" ng-model="qty">  
  </div>  
  <div>  
    Costs: <input type="number" min="0" ng-model="cost">  
  </div>  
  <div><b>Total :</b> {{qty * cost | currency}}</div>  
</div>
```

<http://plnkr.co/edit/EpVlAulGMdHymMakqGMx?p=preview>



# Client-side MVC

## Model Example:

```
<div ng-controller="Controller">  
  Hello <input ng-model='name'> <hr/>  
  <span ng-bind="name"></span> <br/>  
</div>
```

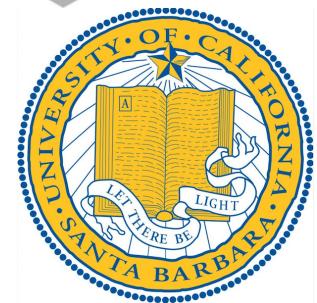
```
angular.module('docsBindExample', [])  
.controller('Controller', ['$scope', function($scope) {  
  $scope.name = 'Andrew';  
}]);
```



# Client-side MVC

## Angular Highlights:

- Ambitious and large framework that really turns the page into an application
- Data binding provides a lot of magic
- A framework you adopt wholesale
- Declarative style retains HTML traditional nature



# Client-side MVC

## Ember.js

- Created by prominent members of the Rails community
  - Yehuda Katz
- Has much in common with Angular
  - All-in framework
  - Two way binding
  - Templating has similar flavor



# Client-side MVC

## Ember Highlights

- Focus on convention over configuration
- Focus on standards compliance
- ES6 modules over custom modules
- ES6 polyfills in places



# HTML5 Localstorage

If we have a javascript application running in our browser, it might be nice for it to sometimes be less reliant on the network

- Data persistence is important, and the network isn't always up.
  - Mobile devices in particular
- We may want to intentionally enter a period of no network and still use the application
  - Airplane flight

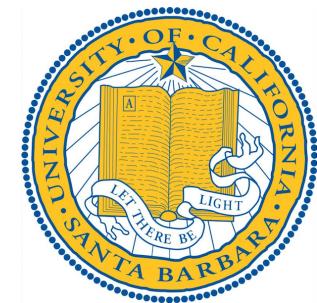


# HTML5 Localstorage

Modern browsers have “LocalStorage” APIs.

SessionStorage: data is persisted as long as the browser remains open.

LocalStorage: data persists between browser executions.



# HTML5 Localstorage

## LocalStorage

```
localStorage.setItem("key", "value");  
localStorage.getItem("key");
```

Most powerfully used when the value is a large, structured JSON document.

Application developer must manage synchronizing localstorage data with updates on server



# HTML5 Localstorage

Demo!

- <http://people.w3.org/mike/localstorage.html>



# For Next Time

Sample Funkload scripts are up ([https://github.com/scalableinternetservices/labapp/tree/master/load\\_testing](https://github.com/scalableinternetservices/labapp/tree/master/load_testing))

- Start working on writing Load Testing scripts. We will be providing EC2 instances configured to simplify load testing.
- Prep for tomorrow's demo (on EC2)

Thursday: Colin Kelley, CTO & Founder, Invoca

