AngularJS, Part 1

Where were we?

A sea of Javascript toolkits

Introducing AngularJS

Disclaimers

- Remember, I said I don't much like Javascript
- Remember, I said I'm not an Angular expert
- Remember, I'm not recommending Angular over other frameworks as some end-all be-all way of writing client-side MVC
- Remember, I'm old and losing most of my marbles

I expect you are looking at the same resources and tutorials I am for AngularJS, so this deck will include very little in the way of tutorial-like process for you to learn the framework.

My motivation is to help you interpret what a popular client-side MVC framework looks like in the context of the Architecture concepts we have discussed in this class.

A sea of Javascript frameworks

Backbone, Knockout, Ember, Angular, ReactJS, RequireJS …

- *And I'm sure I've offended somebody for not including their favorite latest greatest shiny new toy.*

So let's stick to MV* frameworks:

1. Backbone - Pretty similar to what we will do in Angular in concept
2. Ember - MVC, but "views" require knowing Handlebars templates
   - I found some of the documentation language inconsistent
3. Knockout - Nice little library, but seems dedicated to data-binding only, not a full-fledged MVC (no explicit controller I could find…)
4. Angular – we'll go with this as it has gained the most traction in the past couple of years

These slides include very little in the way of a tutorial for you to learn Angular.

My motivation is to help you interpret what a popular client-side MVC framework looks like in the context of the architecture concepts we have discussed in this class.

AngularJS Overview

History

- Google supported (so it has to be good right?)
- From Wikipedia: "AngularJS was originally developed in 2009 by Miško Hevery and Adam Abrons as the software behind an online JSON storage service, … This venture was located at the web domain "GetAngular.com", … the two decided to abandon the business idea and release Angular as an open-source library."

Key concepts

- *Single-page app* – HTML page as container.
  - Server provides a template and data, then waits for AJAX calls
- *MVC*
  - The DOM is the *View*
  - Javascript class are the *Controllers*
  - Objects hold the *Model*
- *Data Binding* – automagically wiring changes in model state to the View so the View is updated continuously (no inner.HTML!)
- *2-way Binding* – automagically wiring the reverse way: changes to the View in the UI update the Model w/out writing an event listener.

Some Syntactic Sugar

You gotta look at some examples, but a summary:

1. Declare yourself to be an ng-app somewhere (html, body, div)
   - Use html or body a lot, div if you are worried about messing up existing stuff
   - Angular manipulates the DOM for you, and somebody else might do that too
2. Make sure you include angular from your server or Google's CDN:

   `https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js">`
3. Declare your Controller class (function)
   - Just like *ng-app*, add *ng-controller* to an element that you want to apply this Controller class too.
   - The write the Javascript for it
4. Use custom AngularJS HTML attribute extensions to establish 2-way data binding (i.e. wire the Model and the View)
   - Use attribute "*ng-model*" to establish a binding from View to Model
   - You can also just use plain-old Javascript variables
   - Use template syntax {{object.property}} to bind Model data into the View at a particular point. This data will automatically update when the Model changes.
   - The Controller wires all these together through a special "blackboard" variable called $*scope*

A Super-Simple Example

```
// From W3Schools http://www.w3schools.com/angular/angular_examples.asp
<!DOCTYPE html>
<html>
<script src= "http://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js"></script>
<body>
<p>Try to change the names.</p>
<div ng-app="myApp" ng-controller="myCtrl">

First Name: <input type="text" ng-model="firstName"><br>
Last Name: <input type="text" ng-model="lastName"><br>
<br>
Full Name: {{firstName + " " + lastName}}
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
    $scope.firstName= "John";
    $scope.lastName= "Doe";
});
</script>

</body>
</html>
```

We are an Angular app

Controller is MyCtrl below

Input widgets data tied to model data

Template binding View to Model changes

Controller sets initial state of Model object greeting

So what is the big whoop?

- We did not manipulate the DOM ourselves
  - We didn't read it for data
  - We didn't write data back into it somewhere (we may not find)
- AngularJS can play nice with other code
  - We use custom attributes on HTML
  - We can define what scope to apply ng-app and ng-controller to
- Less verbose syntax
- Cross-browser goodness
  - Ng-repeat and ng-click examples on previous slide
- 2-way data binding gives us nice interactive apps
  - This is a key feature
  - This is how MVVM is achieved

Another example – todo.html

```
<!doctype html>
<html ng-app="todoApp">
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js"></
    script>
    <script src="todo.js"></script>
    <link rel="stylesheet" href="todo.css">
  </head>
  <body>
    <h2>Todo</h2>
    <div ng-controller="TodoListController as todoList">
      <span>{{todoList.remaining()}} of {{todoList.todos.length}} remaining</span>
      [ <a href="" ng-click="todoList.archive()">archive</a> ]
      <ul class="unstyled">
        <li ng-repeat="todo in todoList.todos">
          <input type="checkbox" ng-model="todo.done">
          <span class="done-{{todo.done}}">{{todo.text}}</span>
        </li>
      </ul>
      <form ng-submit="todoList.addTodo()">
        <input type="text" ng-model="todoList.todoText"  size="30"
               placeholder="add new todo here">
        <input class="btn-primary" type="submit" value="add">
      </form>
    </div>
  </body>
</html>
```

A simple iterator over the todoList

Again, tie View elements to Model data

Simple event listener, invokes a Controller method

Another example – todo.js

```
angular.module('todoApp', [])
    .controller('TodoListController', function() {
        var todoList = this;
        todoList.todos = [
            {text:'learn angular', done:true},
            {text:'build an angular app', done:false}];
        todoList.addTodo = function() {
            todoList.todos.push({text:todoList.todoText, done:false});
            todoList.todoText = '';
        };
        todoList.remaining = function() {
            var count = 0;
            angular.forEach(todoList.todos, function(todo) {
          count += todo.done ? 0 : 1;
            });
            return count;
        };
        todoList.archive = function() {
            var oldTodos = todoList.todos;
            todoList.todos = [];
            angular.forEach(oldTodos, function(todo) {
          if (!todo.done) todoList.todos.push(todo);
            });
        };
    });
```

AngularJS (Phase 1) Concepts: Good and Bad

MVVM: Provides a nice way of doing it

- 2-way data binding basically provides it head-on
- Avoids round-trips to server to update view-model of widget
- Avoids complex, easy to mess up DOM manipulation code

Intrusiveness (this is a bad thing)

- You start adding ng-* attributes everywhere
- $scope is basically a global variable, anyone can $watch
  - This is reminiscent of a *blackboard architecture style* where multiple clients can all post/get data from a single blackboard
  - No information hiding; easy to clobber

There really isn't a "C" yet

- Well kinda…
- Controllers do wire "M" and "V" but on the client the Model logic we see so far is pretty rudimentary and scoped locally
- But wait until we start interacting with the server (stay tuned…)

AngularJS (Phase 1) Concepts (cont.)

Just to reiterate…

- Remember the key difference in MVVM and MVC is that the *ViewModel* provides a way to couple the UI data with the UI widget that reads and writes it
- Basic AngularJS does this very well (ok, pretty well)

What about the big "M" – the world Model

- There is still a bunch of data and logic on the server, why can't we just blow it away now?
  - The world Model is shared by many clients
  - You can't, and shouldn't put too much real Model data into the ViewModel for scaling and security reasons (healthcare.gov)
  - That logic on the server may be in legacy server-side components

So, what's next?

- Extending the Controller concept to route calls to the server, and to *select* a View (just like server-side MVC does)
- Manage code scale with modules