AngularJS Part 1

Welcome to AngularJS Part 1. During this exercise, you will be learning what AngularJS is and the basics of building single-page applications with it.

Getting Started

Open the console and navigate to the "angularjs-part-01/" directory

run `python -m SimpleHTTPServer 9000`

Open your browser to http://localhost:9000

Start editing files!

This material was originally delivered as a lecture with accompanying Powerpoint slides, followed by class exercises. Please read through the lecture summary notes below. You can also view the slides as PDF's by clicking the link "Angular Lecture Slides" under "Additional Reading Material" on the home page. All referenced material can be found in the "angularjs-part-01/" directory, and don't forget to reference your Angular documentation in Zeal.

Lecture Summary

What is AngularJS?

- AngularJS is a "Javascript MVW" by Google.
- How many are familiar with MVC?
- Who can tell me what an MVC is?
- Why is that important?
- There is a problem though, it is conceptually very fluid.
- If you think it might be tricky to wrap your mind around, it still does not get much easier.
- Google seemed to agree on this and decided to replace the *Controller* in MVC with *Whatever works for you* in concession to the fact that the model is not perfect, and should be adapted to best suit your needs.

Checkpoint - So what is it good for?

- Look at conventional web apps.
- Up until the last several years, the most common way to interact with a website was through downloading entire HTML documents each time you needed something new or to communicate something to a server.
- This process can be optimized, but it is still slow.
- It also doesn't feel good. The entirely context of the page is lost and reset.
- Consider how mobile apps work - data is incorporated into the app and the context is adjusted.
- Unfortunately, HTML on it's own isn't sufficient to replicate this user experience.
- AngularJS helps to fill this gap and improve the user experience by providing a foundation for Single Page Applications.

Checkpoint - What is a Single Page Application? (SPA)

- A single page application doesn't need to load entirely new HTML documents to update the page.
- The entire web app exists within the context of one static HTML page, but renders additional pages dynamically.
- Protip: This is accomplished by URL fragment identifier, or through HTML5 URLs.
  - Fragment identifier: /MySPA/#/NewPage
    - AngularJS listens for and responds to the HashChange event.
  - HTML5 URLs: /MySPA/NewPage
    - This method relies on interacting with the browser's history API and requires some configuration on the server side to work correctly.
- AngularJS loads template "partials" from the server as they are needed, and allow the web app to query the server for data used inside of those templates.
- Most commonly, this is done through a Restful interface that returns JSON data.
- Restful interfaces are standardized APIs that can be implemented by any number of server backends - Node.js, PHP, Django, Rails, etc.
- JSON is a standard data format that stands for JavaScript Object Notation.
- The end result is that you have a better user experience (UX) that allows you to provide more rich interactions.

Checkpoint - What are we going to try to accomplish?

- You're going to be building a Single Page App.
- We'll be covering many of the core concepts that AngularJS introduces to help you build great SPAs.
- Let's start with data binding.
  - `<p>Hello {{name}}</p>`
  - Here you see a small HTML fragment that binds the variable "name" to the view.
  - AngularJS processes the HTML as part of its "Digest Cycle" and refers to the data model called name to replace the placeholder with a real value.
  - Protip: The braces are visible until Angular loads. Use ngCloak to hide content until AngularJS loads, or use ngBind to add the variable without using curly braces.
  - Pro tip: How does AngularJS "process" HTML?
    - Compiles expression to a function: {{ name }} becomes function () { return $scope.name; }
    - Adds watcher to scope. Watcher stores the watched function and the previous return value of that function.
  - Pro tip: What is the Digest Cycle?
    - Every watcher's function is invoked and the return value is stored
    - The function is invoked again an return value is compared with the stored one. (For that reason every watcher is invoked at least twice in a digest cycle).
    - The operation is repeated until all watcher calls return the exact same value in two consecutive iterations. Then the model is considered stable and the Digest Cycle finishes. Otherwise (if the number of iterations reaches 10) angular throws error.
- Now let's discuss two-way data binding.

  - AngularJS really shines with this feature.
  - Two-way data binding allows your web app and the user to simultaneously update data models.
    - `<p>Name: <input type="text" ng-model="name"></p>`
    - `<p>Hello {{name}}</p>`
  - The example above demonstrates how we can use a form field to to the "name" model.
  - With this in place, any value you enter in the text field will be the value of the model.
    - Protip: Form field input can be processed by AngularJS "parsers." This will translate input into a data-model friendly format, but will continue to appear in a user-friendly format.
  - Also, if anything else in the app changes the model, the value will be reflected back in the text field without any further developer intervention.
    - Protip: AngularJS will process the value of the data model in the "formatters" pipeline before displaying it on the page. This also provides you with a chance to make it look more user-friendly if the data model is difficult to decipher.
  - If at any point, the name variable changes, AngularJS updates page immediately.
    - Protip: There is a caveat to this statement. AngularJS is not omniscient. If a variable is updated in a way that AngularJS isn't aware of, then the page will not be updated. One way this occurs is if you use setTimeout - Angular doesn't understand that the action was deferred in this context.
  - This does not require a page reload as it would in a conventional web app.

Checkpoint - This is great, but where is the javascript?

- Let's take a look at the repo. All of these items can be found in the same folder as this README file.
- index.html - will serve as the foundation for our SPA. It will reference all further scripts we create.
- The img folder - will store static images we choose to reference.
- The style folder - will store static css files we choose to reference.
- The templates folder - will include HTML partials that will be used to render individual pages.
- The scripts folder - is where you'll be spending much of your time. So let's discuss what you'll be loading up in there.

Checkpoint - Where's the javascript? continued.

- AngularJS - /scripts/angularjs - is a folder with static assets to run the framework, don't change them.
- app.js - /scripts/app.js - is used to declare our application to AngularJS. You won't need to spend a lot of time here, but just make a note that the app's definition and configuration can be adjusted from here.
  - Protip: Anything outside the element for which the ng-app attribute was defined will stay out of the scope and angular won't process it. For this reason it's good to define it on the body element (even on the html element if we plan to interact with stylesheets / meta tags.)
- routes.js - /scripts/routes.js - is our app's "directory." You'll be spending time here organizing your views and which controllers they should be wired into. This is our app's router.
- In AngularJS, apps and features are expressed as modules. Modules can be bootstrapped to the index.html file through use of the ng-app attribute. This declares where AngularJS is responsible for the content of the page.
- Modules can also be "injected" into other modules. This is a more advanced concept.

Checkpoint - Where's the javascript? continued.

- Included in the scripts folder is a controllers folder. What are controllers?
- Controllers provide instructions to AngularJS on how to interact with users.
- Controllers have access to the Scope, which is used to track data models and update variables and pages accordingly. We could say it glues the model to the view. So if something is attached to the scope, then it can be used on the page.
    - Protip: Additional controller information can be accessed by the page, but this is a more advanced concept.
- You should declare your controllers as files here, which are part of the myApp module. There are examples in the repo.
- A controller can store and retrieve data that should be displayed on the page, and capture or handle events in the view.
- For instance, if you wanted to handle a click event on a button, you'd use the attribute ng-click="expression" in the template.

Checkpoint - A note on services.

- The services folder - /scripts/services - will store any services you've declared for the myApp module.
- Services are different from Controllers, which are attached to page views.
- A service does not have direct access to the scope, and thus cannot directly interact with users.
- However, it persists for the lifecycle of the single page app. Controllers lose their state as soon as they are no longer attached to the view.
- Think of services as like an API for your app. You can define functions that are available across the entire app.
    - Protip: Naming of services occurs at a "global" level. To avoid naming collisions, use a namespacing strategy.
- Services are best suited at storing common data and functionality. Business logic should live here.
- If you put business logic in controllers, then you're going to end up repeating yourself.

Checkpoint - Repository content walkthrough.

- Let's take a few minutes to look at the sample content in the repo.
- I've included basic demonstrations of core functionality in AngularJS and examples of how everything is "wired" together to ensure you're not getting hung up on the details of actually loading your files into the page.
- It is possible to write all of your code in a single file, but this can become difficult to troubleshoot and organize as your files grow. It is best to plan this type of file organization up front and stick to it.
    - Protip: There are lots of ways to organize your code. The example repo that's been provided is just one way.

Review Demo Content in Repo

- Let's look at index.html first.
- A number of important things are going on here. It does look like a regular HTML page though.
- Firstly, we want to look at the body tag.

- Here we see an attribute which isn't part of the standard HTML spec: ng-app
- Every element of the page that falls under an element with ng-app will be governed by AngularJS and the module named in the attribute.
- In our case, that will be myApp. At this point you can begin using some of the simple binding examples provided earlier.

Checkpoint - Views and Controllers

- There is a DIV tag that follows which uses the ngInclude directive. This will add an arbitrary HTML file to the view.
- The next block of HTML we'll see starts with a div tag that uses the ng-controller attribute.
- This further breaks down what code takes responsibility for the page. The value of this attribute is the name of the controller you've created.
- Please note! Controllers have a file name and a controller name. AngularJS isn't concerned with the file name, it just must be referenced correctly so that it is loaded into the browser. The name of the controller is what Angular is most concerned with.
- By connecting the DemoController to a block of HTML, we will be able to bind data that it defines there. This includes...
    - A list named "items".
    - A function named "sayHello" that takes one argument.
    - And a function named "sayHelloFromService" which provides an example of how to use functions that are defined in services.
- This block of HTML interacts with all of these in the scope.

Checkpoint - ngView and Routing

- Following the previous block of HTML is an example of how you might use the ng-view directive.
    - Side note: Directives are just the name of extensions to HTML that AngularJS provides. You can define your own, but this is an advanced concept. For now, we'll just use the one's Angular defines for us.
- Even though there is no value for the attribute, AngularJS uses it as a marker for where to place route related views. You can try this out for yourself by visiting #/demo or clicking the href.
- I've defined these controllers and templates as:
    - /scripts/controllers/demoRoute.js
    - /scripts/controllers/demoRoute2.js
    - /scripts/templates/demoRoute.html
    - /scripts/templates/demoRoute2.html
- The route using demoRoute2 files will provide an example of how to define parameters as part of the URL.
- You'll find comments outlining the basic concepts in each file.

Checkpoint - Loading scripts into the page.

- First things first - we want to include Angular.js right off the bat. This happens on line 36 of the index.html file. Nothing else can happen unless it is loaded first.
- We're also loading the routing module on line 37 to allow you to take advantage of adding more views to your

app.

- app.js on line 39 declares that the module exists and registers it with AngularJS. This is important, because without this, the ng-app HTML we setup earlier will cause AngularJS to error out.
- The scripts following app.js on the page are the actual controller and service files. They've been grouped into two sets. The first are what are immediately relevant on index.html. The second set are tied to the router for the code, which will be available to you when you feel more comfortable tackling routing.

Your Mission

Exercise Set A - Binding Data to the View

1. Bind data to text on screen. Use the starter code example (Hello {{ name }}) to emulate adding data to the scope and present it on the page.
2. Bind data to a link tag. Demonstrate how you can dynamically update links with data driven by AngularJS. The link can be to a local file, or to one of the routes in the starter app. Just be sure to use a variable in the href attribute of the link.
3. Bind width value to image. Define a variable in the scope that represents a number. Bind that number to an image's width on the page. You can initialize the value using the ngInit directive. Then set the width attribute of the image in the same way you display a variable on a page using {{ variable }}.
4. Bind data to text field. Create your own example of a text field that binds itself to a model in the scope to demonstrate two-way data binding.
5. Bind data to text area field. Repeat the previous exercise, but using a textarea field instead of a text field.
6. Bind data to radio buttons field. Demonstrate how radio buttons can be used to set data in the scope to a predefined set of options.
7. Bind data to a checkbox field. Demonstrate how checkboxes can be used to setup flags in the scope.
8. Bind data to select field. Demonstrate how to present users with a streamlined list of options to select from to manipulate data in the scope.
9. Bind data to the password field.

Exercise Set B - Simple Events

1. Use ng-click to change the data in a model. Begin with the value being set to hello, then change it to world. For additional reference on AngularJS expressions, please see the "Angular Expressions Reference" documentation linked under "Additional Reading Material" on the home page..
2. Use ng-focus on a text field to add an item to a list that records page activity. This will be a list that records different events that occur while interacting with the page. Add entries for the focus event with the label "focus" and the time when the event occurred. For reference, the focus event occurs when the cursor is active in a field.
3. Use ng-blur to add more items to the page activity list. These should be labeled as "blur" and also contain the time when they occurred. For reference, the blur event occurs when a field loses focus - for instance, when the user clicks out of a field or tabs away.
4. Use ng-change in a select field to show an alert with the message "value has changed."
5. Use ng-mouseover to add events to the page activity list. These items should be labeled "mouseover" and include the time at which they occurred.

Exercise Set C - Manipulate the View

1. Use the ng-show directive to show and hide an image based the data value of a checkbox field.
2. Use ng-switch to display an image of an animal from select field. Images are provided in the /img/ directory.
3. Disable a form field after clicking a button. The form field should use the ng-disabled directive.
4. Use ng-include to add content from the remote partial HTML file named templates/animal.html.
5. Use data bound to radio buttons to determine which animal image to display by using the ng-if directive.
6. Display a list of data: ["peanuts", "cashews", "walnuts", "almonds", "pistachios", "coconuts", "acorns"] using the ng-repeat directive.

Exercise Set D - Filters

1. Sort the list of nuts by name using the orderBy filter.
2. Filter the list for items that start with A.
3. Filter the list based on the data bound in a text field. Show only the items that begin with the value in the text field.
4. Limit the list to the first 3 items.
5. Bonus: Sort the list of nuts and limit the list to the first three.
6. Filter numbers to be shown as currency.
7. Demonstrate use of the uppercase and lowercase filters.

Part 1 Lab - Todo List

Your assignment is to build a single-page app that tracks items in a todo list.

1. Present a list of items to do.
2. Add a field to define new items in the list. Use ng-click with a button to add the value to the list and clear the field afterwards.
3. Display buttons next to each of the items that will remove the item from the list. NOTE: You will want to use the removeItem function included with the tools service. You'll also want to use the $index variable provided when using ng-repeat.
4. Create a dedicated route to display the todo list.
5. Bonus: Store todo items in a service.
6. Bonus: Move your functions for managing the todo list from the controller to a new service, leaving only the code to load the list into the scope, and the ng-click handlers defined in the controller to add and remove items.
7. Create a dedicated route that allows you to view details about a todo item. For the purposes of this assignment, the detail view only displays the item, and a button to remove it which returns you to the list view. The new route should use the array index to determine which one to display details for. Use that index to retrieve the specified todo item from the todo list service, if you've created one. NOTE: You will need to use the $location service to direct users back to the list view.
8. BONUS: Load todo list from a JSON file.
9. BONUS: Save and load data from LocalStorage