AngularJS for Absolute Beginners : Medialoot

Let's face it, writing web applications is hard. Battling to make a functional front-end is one of the biggest pain points. AngularJS eases this pain. The learning curve may be steep, but we'll break down the complexities in plain English to get you up and running in no time. By the end of this tutorial you'll be armed with all of reasons to use Angular and have built an Angular app in around 20 lines of JavaScript code.

Prerequisites

The only requirement for this tutorial is a basic understanding of front-end languages such as HTML, CSS, and JavaScript. If you've worked with jQuery before then you should be able to following along with ease.

It's all about structure

Before we get into the nitty-gritty, let's take a look at why we would want to use Angular and some of the problems it solves.

If you've ever worked on the front-end of a web app before, you've probably written code that looks like this:

```
$('#btAdd').on('click', function() {

  var $txtNewTodo = $('#txtNewTodo'),
      newTodo = $txtNewTodo.val(),
      $todoList = $('#todoList');

  $todoList.append('<li>' + newTodo + '</li>');

});
```

In the snippet above we are just simply adding a click handler to a button that just handles adding a new todo to a todo list. Even though this is just a simple example it's soaked with bad practices.

Imperative

The good news is that our JavaScript (using jQuery in this case) is unobtrusive. This means that it's separated from our HTML. The bad part is, without looking at the JavaScript code we have no idea that the button has a click handler attached to it.

This is referred to as **imperative programming**. The statement (the click handler) changes the state web application by creating a flow that appends new elements to a list.

In web development this can be seen as a negative because it can hide important functionality.

Direct DOM manipulation

Every time you directly insert a HTML element into a page, an angel loses its wings. The DOM is a fickle creature. Manually inserting elements like the list item, can easily be broken. If we just change one element's ID, we are completely hosed.

Global Scope

The click handler is declared on global scope. This means whenever it is referenced in any page it will try to add this event to an element with the id of "#btAdd". As the app grows this can be difficult to maintain and cause a lot of confusion.

Lack of organization

Most importantly of all, this code snippet has no meaningful structure. As we add more functionality to the app the file will just become more cluttered and confusing.

So what can we do? We need to create some meaningful and reusable structure for application. Since we are sane people, we won't implement our own framework, we'll use

AngularJS.

AngularJS to the rescue

AngularJS is a front-end JavaScript framework. It's nothing at all like using a library like jQuery. Comparing the two is like comparing a hammer to a toolkit.

Angular provides us with an opinionated way to build a powerful front-end for our application. The key word is opinionated. Angular forces you to follow a specific pattern. This can been seen a good or bad thing. In this case, we'll decide that we like it for now.

The Structure

Angular is built to isolate specific responsibilities of an application from each other. We should do things like manipulate the DOM, get data from services, and create structured templates all in different places.

These different places are referred to as **components**. Each component has a different name. In this tutorial we will focus on **controllers** and **views**.

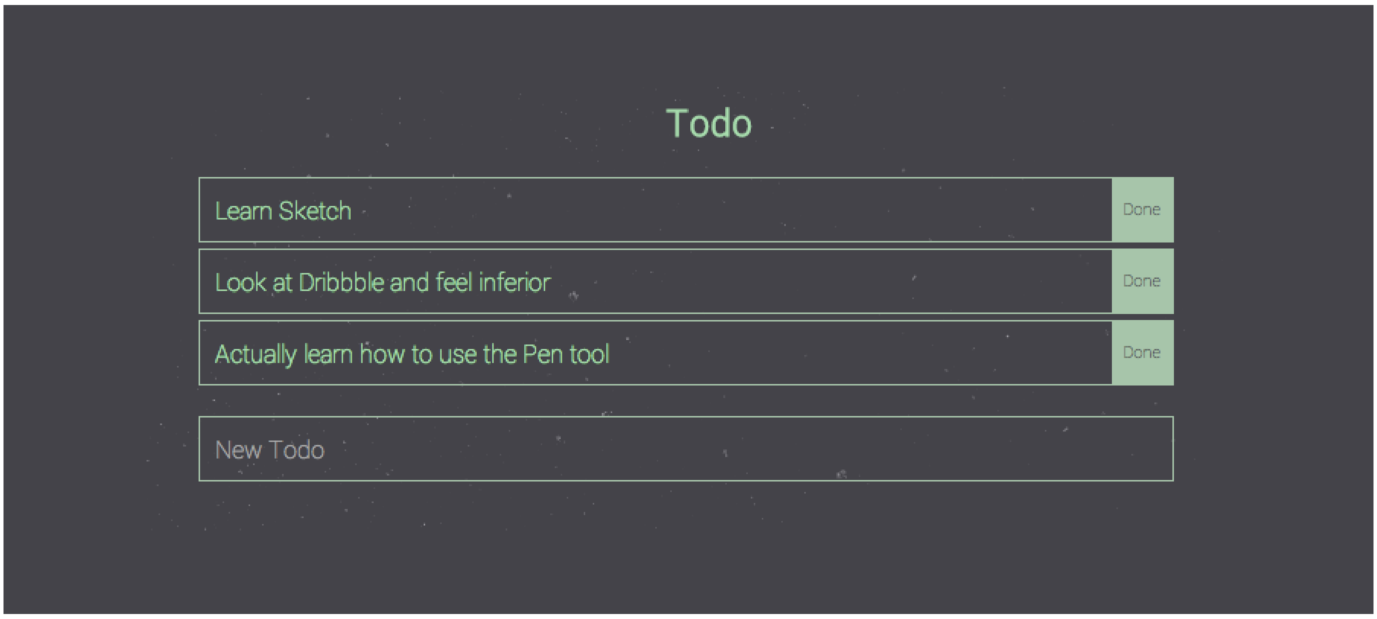
A controller is used to set state and add behavior to a page. This means if you're making a todo list, you would create an array of todos in a controller that would be appended to a page. The controller itself does not append the todos, it just simply houses them for another component to use.

The page these todos will be appended to is called a view. A view is just a HTML file. However, in Angular we can augment the functionality of HTML by using **directives**. Just think of directives as custom HTML that we can use to manipulate the DOM. You want to attach a click handler? There's a directive for that. You want to append multiple items to a list? There's a directive for that too.

This is in contrast to jQuery. With jQuery we would create a click handler in a script file somewhere. With Angular, we declare that there's a click event on the element, but write the logic elsewhere (in the controller). This is referred to as **declarative programming**. We express that our element does something without defining the functionality within it.

Essentially, this means we can store data in a controller. The controller can pass that data to the view, and the view can use directives to manipulate the DOM on the page. A view has a controller, and a controller has that same view. A place for everything and everything in its place. If you understand that, then half of the battle is already over.

The Demo


Todo
Learn Sketch
Done
Look at Dribbble and feel inferior
Done
Actually learn how to use the Pen tool
Done
New Todo

Now that you've battled through the explanation of Angular, we can finally write some code.

It's absolutely obligatory nowadays to do a Todo application when first learning a new framework. This demo won't break from that tradition. Below is the HTML and CSS for our application. Currently, there is no JavaScript. Although, we have included the JavaScript files for Angular and our app.

1. Create the module

The first thing we need to do is initialize our app by creating a module. Don't worry, it's just one line of code.

```
var app = angular.module('Todo', []);
```

At the highest level, everything in Angular is inside of a **module**. Every module has a name that has to be unique within the application of use. Modules can be used within other modules for added functionality. The empty array is used for declaring what other modules the app will be using. You will commonly hear these referred to as dependencies, because your app depends on them to run.

In this demo we'll keep it simple by creating one module that represents our application. This will leave our dependency array blank.

Right now, if you are only going to understand one thing about modules, know that they are used for hold onto our controllers and other relevant code about our application. This also helps keep the code we write from being declared at a global scope since it is inside of the module.

2. Initialize the View

Now that we have a our module declared, we can make our view aware of our Angular app.

```
<body ng-app="Todo">
```

Above, we are adding an attribute to our body tag called "ng-app" and setting its value to the name of our module.

If you're wondering, is that "ng-app" thing a directive? Feel happy because you're absolutely right. If not, don't worry it looks very strange at first.

Our view is now ready for Angular excellence. However, we need a controller to set the state and behavior of our view.

3. Create the controller

Using the app variable we've stored our module in, we can create a controller.

```
var app = angular.module('Todo', []);
app.controller('TodoCtrl', function() {

});
```

We can add a controller into our module by calling the controller function on the module. This function takes in two parameters. First, the name of our controller. We're using a common convention of ending our controller with "Ctrl". Second, we're providing a function that will house our controller's logic.

Take note that we're not returning our controller into a variable. We are actually adding it to our module.

So how do we get data from the controller to the view? We need to use a component that links the controller to the view. This component is called **scope**.

4. Set up $scope

To get scope included in our controller we need to "inject" it. We just simply write `$scope` as a parameter of our controller's function. The dollar sign in front indicates that it's a core service of the Angular framework. We don't have to instantiate it inside of our controller since we are "injecting" it. Angular will know how to resolve this for us. This is known as **dependency injection**.

```
var app = angular.module('Todo', []);
app.controller('TodoCtrl', function($scope) {

});
```

Now that we have our `$scope` object we can attach data to it that we want on the view.

```
var app = angular.module('Todo', []);
app.controller('TodoCtrl', function($scope) {
  $scope.message = 'Angular is pretty cool.';
});
```

Remember that **$scope** is the link between our controller and our view. We'll add a message onto it and we'll get it added to the page. The process of adding it to the page is called **data binding**.

5. Tell the view about the controller

```
<body ng-app="Todo">
  <div class="page-container" ng-controller="TodoCtrl">

  </div>
</body>
```

To get our view to know about the controller we need to use the "ng-controller" directive on our page container. Any of the Angular related code inside of this `page-container` will come from the `$scope` of the `TodoCtrl`.

6. Bind data to the page

Before we go off adding todos we need to see how a simple data binding example works. In our HTML file we can bind data to the page by using a special syntax with double curly brace on each side.

```
<body ng-app="Todo">
  <div class="page-container" ng-controller="TodoCtrl">
    <span> {{ message }} </>
  </div>
</body>
```

When we run the page we should see the message on the page. This is pretty awesome but it's about to get even better.

```
<body ng-app="Todo">
  <div class="page-container" ng-controller="TodoCtrl">
    <input type="text" ng-model="message" />
    <span> {{ message }} </>
  </div>
</body>
```

If you type into the textbox you'll notice that the text in the span updates as well. This is known as **two way data binding**. As of the current stable version (1.2.17) all binding within Angular is two way.

We can use this feature to do some pretty amazing stuff. This is a big deal because it won't require us to keep notifying that controller that the data on the page has been updated, it's all handled for us.

We were able to set this up by declaring the "ng-model" directive on the textbox. This tells Angular to have the textbox update the value for the message string on our $scope. With the combination of directives and templates we easily render our data to the page.

7. Create our view layout

Check out the beginning Plunker to see what we're starting with. This is a static version of our view. We have a list and then a textbox to add new todos. When a user hits enter after typing their new todo we'll add it to the list.

8. Set up data on the controller

```
var app = angular.module('Todo', []);
app.controller('TodoCtrl', function($scope) {
  $scope.todos = [
    'Learn Sketch',
    'Look at Dribbble and feel inferior',
    'Actually learn how to use the Pen tool'
  ];
});
```

Our todo list is best represented by an array. We can easily add and remove todos from an array with some simple coding. Now we just need to bind this list from the controller to the view.

9. Bind the todos to the view

```
<body ng-app="Todo">
  <div class="page-container" ng-controller="TodoCtrl">
    <h2>Todo</h2>

      <ul class="todo-list" ng-repeat="todo in todos track by
$index">

      </ul>

  </div>
</body>
```

On the unordered list we add the "ng-repeat" directive. This allows us to bind multiple items that are organized in a list or an object to the view. We specify that we are binding the todos by using an expression.

An expression is a statement that tells the "ng-repeat" directive what the items we are looping through are. When we say todo in todos it's simply telling Angular that each item in the array is a todo. We also will write track by $index so there are no issues with duplicate entries. Don't worry too much about the details on that.

10. Set up the template

```
<body ng-app="Todo">
  <div class="page-container" ng-controller="TodoCtrl">
    <h2>Todo</h2>

      <ul class="todo-list" ng-repeat="todo in todos track by
$index">
        <li>
          <span>{{ todo }}</span>
          <button class="bt bt-achieve">Done</button>
        </li>
      </ul>

  </div>
</body>
```

Once we have our expression in place, we can set up our template. Inside of the todo list we aren't going to manually insert any elements. We are going to set up a template for how they will be structured when bound to the view.

Inside we create a span to hold our text and then add a button that will remove the todo once it is done. Every todo in our list will get added to the todo list with the structure. Now we just need to set up the events for finishing a todo.

11. Finishing a todo

```
<body ng-app="Todo">
  <div class="page-container" ng-controller="TodoCtrl">
    <h2>Todo</h2>

      <ul class="todo-list" ng-repeat="todo in todos track by
```

```
$index">
        <li>
          <span>{{ todo }}</span>
          <button class="bt bt-achieve" ng-
click="done(todo)">Done</button>
        </li>
      </ul>

  </div>
</body>
```

Ideally, we would click the done button to clear the todo. To add a click handler in Angular we use the "ng-click" directive. For the value of the directive we supply a function that we call from the controller. In this case we are looping through todos so we also can pass the current todo as a parameter as well. This way we will have access to the todo that needs to be cleared.

This is completely different from the way we did it with jQuery. Rather than imperatively writing the click handler and hiding it away somewhere. We can declaratively state that the button has an action when clicked. This is still unobtrusive because the code that handles the click is removed from our HTML.

Right now this button won't do anything because the function done doesn't exist, so let's go write that in the controller.

12. Removing the todo in the controller

```
var app = angular.module('Todo', []);
app.controller('TodoCtrl', function($scope) {
  $scope.todos = [
    'Learn Sketch',
    'Look at Dribbble and feel inferior',
    'Actually learn how to use the Pen tool'
  ];

  $scope.done = function(todo) {
    var indexOf = $scope.todos.indexOf(todo);
    if (indexOf !== -1) {
      $scope.todos.splice(indexOf, 1);
    }
  };
});
```

The above snippet adds a function named done to the $scope object. This way we have access to it in our view. That function takes in the todo as a parameter. We then get the index of the todo inside of our array. If it's not -1 then we know it's inside of the array. Once we have an existing index we can splice it off of the array by passing through where we want to begin splicing and just removing one item.

Now when we click done, that todo will be removed from the array. We aren't talking to the DOM directly. We're simply just removing an item from an array. Since we have two way data binding this will automatically update the view.

13. Add a new todo

```
<body ng-app="Todo">
  <div class="page-container" ng-controller="TodoCtrl">
    <h2>Todo</h2>

      <ul class="todo-list" ng-repeat="todo in todos track by
$index">
        <li>
          <span>{{ todo }}</span>
          <button class="bt bt-achieve" ng-
click="done(todo)">Done</button>
        </li>
      </ul>

  </div>

  <ul class="add-todo">
    <li>
      <input type="text" class="txt" placeholder="New Todo"
             ng-model="newTodo" ng-keyup="add($event)" />
    </li>
  </ul>

</body>
```

Below our todo list we've added a a textbox with a "ng-model" directive set to a `newTodo` string that we will place on our $scope object. We also set an "ng-keyup" directive that is wired up to an add function. There is an `$event` object being passed back to the function. This will help us know which key was pressed because we are only interested in the enter button.

14. Adding a todo to the controller

```
var app = angular.module('Todo', []);
app.controller('TodoCtrl', function($scope) {
  $scope.newTodo = '';

  $scope.todos = [
    'Learn Sketch',
    'Look at Dribbble and feel inferior',
    'Actually learn how to use the Pen tool'
  ];

  $scope.done = function(todo) {
    var indexOf = $scope.todos.indexOf(todo);
    if (indexOf !== -1) {
      $scope.todos.splice(indexOf, 1);
    }
  };

   $scope.add = function(e) {
    if (e.which && e.which === 13) {
      $scope.todos.push($scope.newTodo);
      $scope.newTodo = '';
    }
  };
});
```

We've just added two simple things to our controller. There is now a `newTodo` object on the $scope and a function for adding new todos. This add function just checks to see which key was pressed and we know that by the code that was returned in the which property on the event. If it is 13 then it was the enter button. The todo gets pushed onto the array and then we clear out the text for the new todo.

Summary

It may have seen like an arduous journey, but this is a key piece to understanding Angular and writing better applications with better structure.

With Angular we declaratively define our events on our HTML elements, but wire up the code in the controller. This separation makes our application much easier to maintain.

Now the challenge is up to you to go out there an experiment. Show some off the cool stuff you came up with in the comments!