Angular JS - some basics

Friday, 5 October 2018 3:21 PM

Angular-Js is JS framework that helps build web apps.

Benefits of Angular JS above other frameworks

- Dependency Injection
- Two way data binding
- Easy Testing (both unit and e2e)
- Very easy to build using MVC way using angular.
- Controlling behaviour of DOM elements using directives, filters, etc.

Module

- A module is a container for different parts of your application i.e controllers, services, directives, filters, etc
- You can think of a module as a Main() method in other types of applications.
- Modules declaratively specify how the angular application should be bootstrapped.

The following example, creates a module.

```
var myApp = angular.module("myModule", [])
```

The **first parameter** specifies the **name of the module**.
The **second parameter** specifies the **dependencies** for this module

Controller

- In angular a controller is a JavaScript function. The job of the controller is to build a model for the view to display.
- The model is the data. In a real world application, the controller may call into a service to retrieve data from the database.

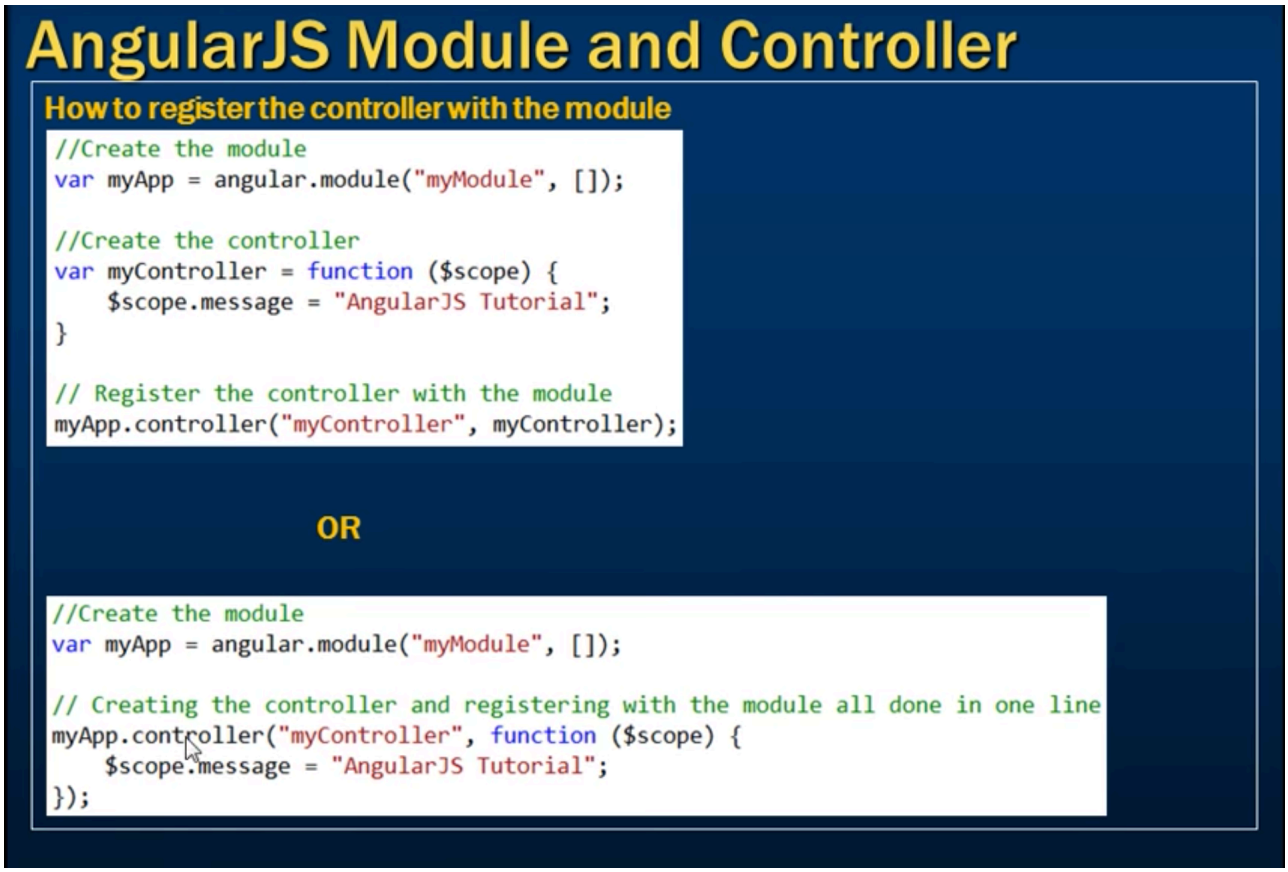
How to create a controller in angular Simple, create a JavaScript constructor function

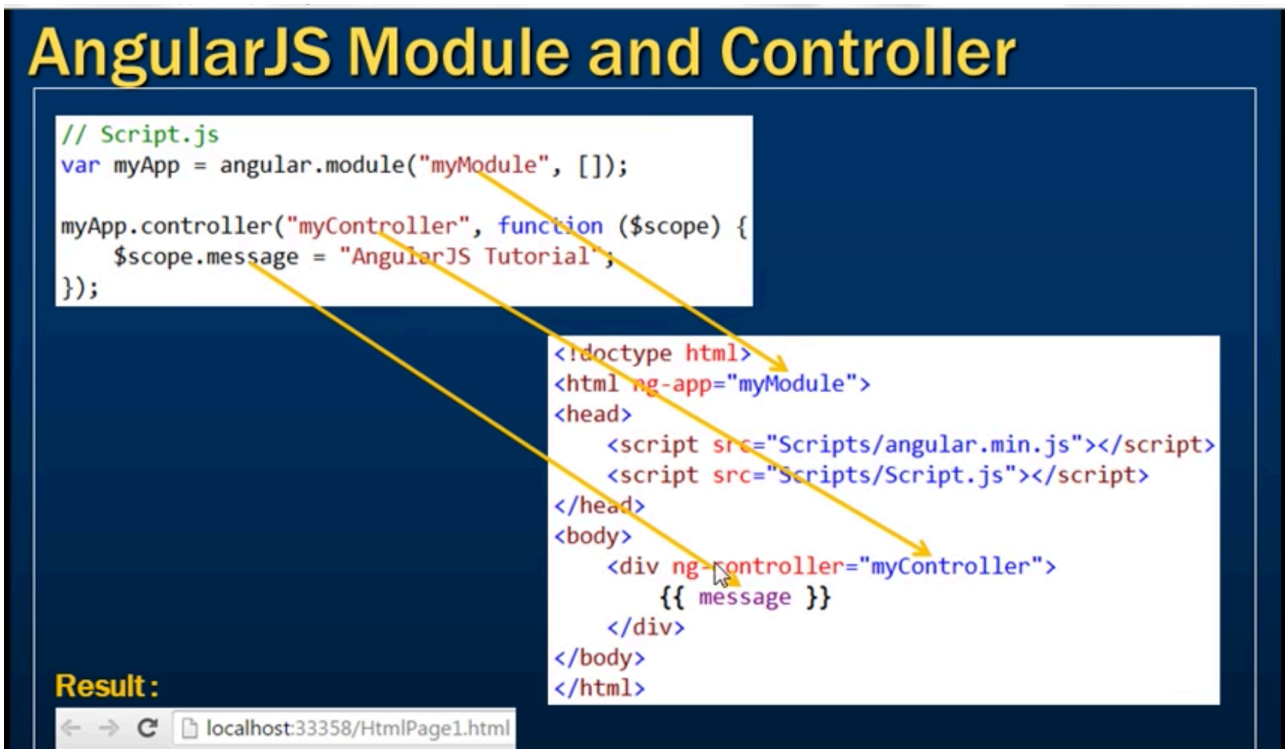
```
var myController = function ($scope) {
        $scope.message = "AngularJS Tutorial";
`
```

}

Use module object's controller function to register the controller with the module

myApp.controller("myController", myController);





$scope

is an angular object that is passed to the controller function by the angular framework. We attach the model to the $scope object, which will then be available in the view. With in the view, we use the databinding expression to retrieve the data from the scope object and display it.

Data attached to scope give **state** and **functions** attached to scope provides **behavior**.

Controller in Depth

The job of the controller is to build a model for the view. The controller does this by attaching the model to the **scope**.
The scope is not the model, it's the data that you attach to the scope is the model. We can attach objects, lists, complex objects etc. to scope object.
The view will then use the data-binding expression to retrieve the model from the scope. This means the controller is not manipulating the DOM directly, thus keeping that clean separation between the model, view and the controller

TIP : If controller name is misspelled -> data binding expressions come as they are typed. And error is raised(seen in developer tools).
TIP2: If property name in binding expression is misspelled. -> It will not give any error. Simply return null or undefined. i.e. on view we get nothing on expression space.

Use method chaining for better readable code.

TIPS:

- Binding expression with **image src** results in **404** error. We get the image but it is loaded in 2nd request after DOM is rendered, and after that when binding expression is evaluated then we get image. To resolve this use **ng-src** attribute. This makes sure request is made only after binding expression is evaluated.
- **Ng-repeat:** similar to for each loop.
- **Ng-include:** to embed a html page into another html page. Used when we

want to reuse a view.

Two way Data binding

Data-binding in AngularJS apps is the **automatic synchronization(at all times)** of data between the model and view components. The way that AngularJS implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa. You can think of the view as simply an instant projection of your model. **Ng-model** and **binding expression** used. The important thing in the example is that AngularJS provides ***live*** **bindings**

Handling events in Angular JS

See: http://csharp-video-tutorials.blogspot.com/2015/11/handling-events-in-angularjs.html
https://www.youtube.com/watch?v=8bf5aZSWp9A&index=7&list=PL6n9fhu94yhWKHkcL7RJmmXyxkuFB3KSl

Ng-click is used to call functions attached to **$scope** which gives behavior.

Angular JS Filters

- Used to **format**, **sort** and **filter** data.
- Can be used with a **binding expression** or with a **directive**.
- To apply filter use **|** character. Example:
    - {{ expression | filter:parameter }}

$http Service

- Used to make http request to remote server.
- It is a function that has a **single input** parameter i.e. **configuration** object.

****
***

What are Directives?

At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's **HTML compiler** (`$compile`) to attach a specified behavior to that DOM element (e.g. via event listeners), or even to transform the DOM element and its children.

AngularJS comes with a set of these directives built-in, like `ngBind`, `ngModel`, and `ngClass`

Some Imp Directives

- In the example above we use the `ng-app` attribute, which is linked to a directive that automatically initializes our application. Anything inside **ng-app** directive section is managed by angular.
- AngularJS also defines a directive for the `input` element that adds extra behavior to the element. The `ng-model` directive stores/updates the value of the input field into/from a variable.
- The second kind of new markup are the double curly braces `{{ expression | filter }}`. An expression in a template is a JavaScript-like code snippet that allows AngularJS to read and write variables. These are **Binding Expression**.

Dependency Injection

Everything within AngularJS (directives, filters, controllers, services, ...) is created and wired using dependency injection. Within AngularJS, the DI container is called the injector.

To use DI, there needs to be a place where all the things that should work together are registered. In AngularJS, this is the purpose of the modules. When AngularJS starts, it will use the configuration of the module with the name defined by the `ng-app` directive, including the configuration of all modules that this module depends on.

Controllers

Controller

Use controllers to:

- Set up the initial state of the `$scope` object.
- Add behavior to the `$scope` object.

Do not use controllers to:

- Manipulate DOM — Controllers should contain only business logic. Putting any presentation logic into Controllers significantly affects its testability. AngularJS has databinding for most cases and directives to encapsulate manual DOM manipulation.
- Format input — Use AngularJS form controls instead.
- Filter output — Use AngularJS filters instead.
- Share code or state across controllers — Use AngularJS services instead.
- Manage the life-cycle of other components (for example, to create service instances).

In general, a Controller shouldn't try to do too much. It should contain only the business logic needed for a single view. The most common way to keep Controllers slim is by encapsulating work that doesn't belong to controllers into services and then using these services in Controllers via dependency injection.

As an (optional) naming convention the name starts with capital letter and ends with "Controller".

See **Scope Inheritance Example** . It is important.

Services

AngularJS services are substitutable objects that are wired together using dependency injection (DI). You can use services to organize and share code across your app.

AngularJS services are:

- Lazily instantiated – AngularJS only instantiates a service when an application component depends on it.
- Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.

Application developers are free to define their own services by registering the service's name and **service factory function**, with an AngularJS module.

The **service factory function** generates the single object or function that represents the service to the rest of the application. The object or function returned by the service is injected into any component (controller, service, filter or directive) that specifies a dependency on the service.

```
var myModule = angular.module('myModule', []);
myModule.factory('serviceId', function() {
  var shinyNewServiceInstance;
  // factory function body that constructs
shinyNewServiceInstance
  return shinyNewServiceInstance;
});
```