**What is AngularJS**

Angular JS is an open source JavaScript framework that is used to build web applications. Angular Js is developed by Google.

**Advantage of AngularJS**

**Following are the advantages of AngularJS over other JavaScript frameworks:**

* **Dependency Injection:** Dependency Injection specifies a design pattern in which components are given their dependencies instead of hard coding them within the component.
* **Two way data binding:** AngularJS creates a two way data-binding between the select element and the orderProp model. orderProp is then used as the input for the orderBy filter.
* **Testing:** Angular JS is designed in a way that we can test right from the start. So, it is very easy to test any of its components through unit testing and end-to-end testing.
* **Model View Controller:** In Angular JS, it is very easy to develop application in a clean MVC way. You just have to split your application code into MVC components i.e. Model, View and the Controller.
* Directives, filters, modules, routes etc.

**AngularJS MVC Architecture**

MVC stands for Model View Controller. It is a software design pattern for developing web applications. It is very popular because it isolates the application logic from the user interface layer and supports separation of concerns.

The MVC pattern is made up of the following three parts:

1. **Model:** It is responsible for managing application data. It responds to the requests from view and to the instructions from controller to update itself.
2. **View:** It is responsible for displaying all data or only a portion of data to the users. It also specifies the data in a particular format triggered by the controller's decision to present the data. They are script-based template systems such as JSP, ASP, PHP and very easy to integrate with AJAX technology.
3. **Controller:** It is responsible to control the relation between models and views. It responds to user input and performs interactions on the data model objects. The controller receives input, validates it, and then performs business operations that modify the state of the data model.

**AngularJS Data Binding**

Data binding is a very useful and powerful feature used in software development technologies. It acts as a bridge between the view and business logic of the application.

AngularJS follows Two-Way data binding model.

**One-Way Data Binding**

The one-way data binding is an approach where a value is taken from the data model and inserted into an HTML element. There is no way to update model from view. It is used in classical template systems. These systems bind data in only one direction.

**Two-Way Data Binding**

Data-binding in Angular apps is the automatic synchronization of data between the model and view components.

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. If the model is changed, the view reflects the change and vice versa.

**AngularJS Expressions**

In AngularJS, expressions are used to bind application data to HTML. AngularJS resolves the expression, and return the result exactly where the expression is written.

**Difference between AngularJS Expressions and JavaScript expressions:**

AngularJS expressions can be written inside HTML, while JavaScript expressions cannot.

AngularJS expressions support filters, while JavaScript expressions do not.

AngularJS expressions do not support conditionals, loops, and exceptions, while JavaScript expressions do.

**Similarity between AngularJS Expressions and JavaScript expressions:**

AngularJS expressions and JavaScript expressions both can contain literals, operators and variables.

**AngularJS Directives**

AngularJS facilitates you to extend HTML with new attributes. These attributes are called directives.

There is a set of built-in directive in AngularJS which offers functionality to your applications. You can also define your own directives.

Directives are special attributes starting with ng- prefix. Following are the most common directives:

**ng-app:** This directive starts an AngularJS Application.

**ng-init:** This directive initializes application data.

**ng-model:** This directive defines the model that is variable to be used in AngularJS.

**ng-repeat:** This directive repeats html elements for each item in a collection.

**ng-app directive :**

ng-app directive defines the root element. It starts an AngularJS Application and automatically initializes or bootstraps the application when web page containing AngularJS Application is loaded. It is also used to load various AngularJS modules in AngularJS Application.

**ng-init directive :**

ng-init directive initializes an AngularJS Application data. It defines the initial values for an AngularJS application.

**ng-model directive :**

ng-model directive defines the model/variable to be used in AngularJS Application.

**ng-repeat directive :**

ng-repeat directive repeats html elements for each item in a collection. In following example, we've iterated over array of countries.

**AngularJS Controllers :**

AngularJS controllers are used to control the flow of data of AngularJS application. A controller is defined using ng-controller directive. A controller is a JavaScript object containing attributes/properties and functions. Each controller accepts $scope as a parameter which refers to the application/module that controller is to control.

**Note:**

Here, the AngularJS application runs inside the <div> is defined by ng-app="myApp".

The AngularJS directive is ng-controller="myCtrl" attribute.

The myCtrl function is a JavaScript function.

AngularJS will invoke the controller with a $scope object.

In AngularJS, $scope is the application object (the owner of application variables and functions).

The controller creates two properties (variables) in the scope (firstName and lastName).

The ng-model directives bind the input fields to the controller properties (firstName and lastName).

**AngularJS Module**

In AngularJS, a module defines an application. It is a container for the different parts of your application like controller, services, filters, directives etc.

A module is used as a Main() method. Controller always belongs to a module.

**How to create a module**

The angular object's module() method is used to create a module. It is also called AngularJS function angular.module

Now we can add controllers, directives, filters, and more, to AngularJS application.

**How to add controller to a module**

If you want to add a controller to your application refer to the controller with the ng-controller directive.

**AngularJS Scopes**

The Scope is an object that is specified as a binding part between the HTML (view) and the JavaScript (controller). It plays a role of joining controller with the views. It is available for both the view and the controller.

**How to use Scope :**

To make a controller in AngularJS, you have to pass the $scope object as an argument.

**AngularJS Dependency Injection**

AngularJS comes with a built-in dependency injection mechanism. It facilitates you to divide your application into multiple different types of components which can be injected into each other as dependencies.

Dependency Injection is a software design pattern that specifies how components get holds of their dependencies. In this pattern, components are given their dependencies instead of coding them within the component.

Modularizing your application makes it easier to reuse, configure and test the components in your application. Following are the core types of objects and components:

value

factory

service

provider

constant

These objects and components can be injected into each other using AngularJS Dependency Injection.

**Value**

In AngularJS, value is a simple object. It can be a number, string or JavaScript object. It is used to pass values in factories, services or controllers during run and config phase. //define a module

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

//create a value object and pass it a data.

myModule.value("numberValue", 100);

Here, values are defined using the value() function on the module. The first parameter specifies the name of the value, and the second parameter is the value itself. Factories, services and controllers can now reference these values by their name.

**Injecting a value**

To inject a value into AngularJS controller function, add a parameter with the same when the value is defined.

myModule.controller("MyController", function($scope, numberValue) {

console.log(numberValue);

});

**Factory :**

Factory is a function that is used to return value. When a service or controller needs a value injected from the factory, it creates the value on demand. It normally uses a factory function to calculate and return the value.

**Note:** It is not the factory function that is injected, but the value produced by the factory function.

**Service :**

In AngularJS, service is a JavaScript object which contains a set of functions to perform certain tasks. Services are created by using service() function on a module and then injected into controllers.

**Provider :**

In AngularJS, provider is used internally to create services, factory etc. during config phase (phase during which AngularJS bootstraps itself). It is the most flexible form of factory you can create. Provider is a special factory method with a get() function which is used to return the value/service/factory.

**Constants :**

You cannot inject values into the module.config() function. Instead constants are used to pass values at config phase.

mainApp.constant("configParam", "constant value");

**AngularJS Filters**

In AngularJS, filters are used to format data.

*How to add filters to expressions*

You can add filters to expressions by using the pipe character |, followed by a filter.

*How to add filters to directives*

Filters can be added to directives, like ng-repeat, by using the pipe character |, followed by a filter.

**Building Custom Directives**

An Angular directive comes in four flavors in terms of appearance.

A new HTML element **(<date-picker></date>**).

An attribute on an element **(<input type="text" date-picker/>).**

As a class **(<input type="text" class="date-picker"/>).**

As comment **(<!--directive:date-picker-->).**

**How a typical directive is written in Angular ?**

It is registered in the same way as a controller, but it returns a simple object (directive definition) that has several properties to configure the directive.

//Hellow World directive

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

app.directive('helloWorld', function() {

return {

restrict: 'AE',

replace: 'true',

template: '<h3>Hello World!!</h3>'

};

});

app.directive() function registers a new directive in our module.

1)The first argument to this function is the directive name.

2)The second argument is a function which returns a directive definition object

If your directive has dependencies on external objects/services such as $rootScope, $http, or $compile, they can be injected here.

The directive can be used in HTML

as an element - <hello-world/>

as an attribute - <div hello-world></div>

**restrict –** This provides a way to specify how a directive should be used in HTML

**template/templateUrl** – This specifies the HTML markup that will be produced when the directive is compiled and linked by Angular.

*The template can be complex, often involving other directives, expressions ({{ }}), etc.*

**The link Function and Scope**

By default a directive does not get a new child scope. Rather, it gets the parent’s scope. This means that if the directive is present inside a controller it will use that controller’s scope.

To utilize the scope, we can make use of a function called link.

<body ng-controller="MainCtrl">

<input type="text" ng-model="color" placeholder="Enter a color" />

<hello-world/>

</body>

var app = angular.module("myApp", []);

app.controller("MainCtrl", function($scope) {

});

app.directive('helloWorld', function() {

return {

restrict: 'AE',

replace: true,

template: '<p style="background-color:{{color}}">Hello World</p>',

link: function(scope, elem, attrs) {

elem.bind('click', function() {

elem.css('background-color', 'white');

//scope.color = "white";

scope.$apply(function() {

scope.color = "white";

});

});

elem.bind('mouseover', function() {

elem.css('cursor', 'pointer');

});

}

};

});

**scope** – The scope passed to the directive. In this case it’s the same as the parent controller scope.

**elem** – The jQLite (a subset of jQuery) wrapped element on which the directive is applied.

**attrs** – An object representing normalized attributes attached to the element on which the directive is applied.

The link function is mainly used for attaching event listeners to DOM elements, watching model properties for changes, and updating the DOM.

**The compile Function**

The compile function is used to perform any DOM transformation before the link function runs. It accepts the following arguments.

**tElement** – The element on which the directive is applied.

**attrs** – The normalized list of attributes declared on the element.

Just note that the compile function does not have access to the scope, and must return a link function. But, if there is no compile function you can configure the link function as usual. The compile function can be written as:

app.directive('test', function() {

return {

compile: function(tElem,attrs) {

//do optional DOM transformation here

return function(scope,elem,attrs) {

//linking function here

};

}

};

});

Once all the directives have been identified, Angular executes their compile functions. As previously mentioned, the compile function returns a link function which is added to the list of link functions to be executed later. This is called the compile phase. If a directive needs to be cloned multiple times (e.g. ng-repeat), we get a performance benefit as the compile function runs once for the cloned template, but the link function runs for each cloned instance. That’s why the compile function does not receive a scope.

**Changing a Directive’s Scope**

By default a directive gets the parent’s scope. But we don’t want that in all cases.So, we have two other options:

A **child scope** – This scope prototypically inherits the parent’s scope.

An **isolated scope** – A new scope that does not inherit from the parent and exists on its own.

The scope can be configured with the scope property of the directive definition object.

app.directive('helloWorld', function() {

return {

scope: true, // use a child scope that inherits from parent

restrict: 'AE',

replace: 'true',

template: '<h3>Hello World!!</h3>'

};

});

The above code asks Angular to give the directive a new child scope that prototypically inherits from parent scope

**The Other Option is Isolated Scope** :

This directive uses a new isolated scope that does not inherit from the parent. Isolated scopes are good when we want to create reusable components. By isolating the scope we guarantee that the directive is self contained and can be easily plugged into an HTML app.

app.directive('helloWorld', function() {

return {

scope: {}, // use a new isolated scope

restrict: 'AE',

replace: 'true',

template: '<h3>Hello World!!</h3>'

};

});

Isolating the scope does not mean that you have no access to the parent scope’s properties. There are techniques that allow you to access the parent scope’s properties and also watch for changes on them.

Option 1: Use @ for One Way Text Binding

Option 2: Use = for Two Way Binding

Option 3: Use & to Execute Functions in the Parent Scope

**Parent Scope vs. Child Scope vs. Isolated Scope**

By default a directive does not create a new scope and uses the parent’s scope.

Letting all the directives use the same parent scope is not a good idea because anybody can modify our scope properties.

**Parent Scope (scope: false)** – This is the default case. If your directive does not manipulate the parent scope properties you might not need a new scope. In this case, using the parent scope is okay.

**Child Scope (scope:true)** – This creates a new child scope for a directive which prototypically inherits from the parent scope. If the properties and functions you set on the scope are not relevant to other directives and the parent, you should probably create a new child scope. With this you also have all the scope properties and functions defined by the parent.

**Isolated Scope (scope:{})** – This is like a sandbox! You need this if the directive you are going to build is self contained and reusable. Your directive might be creating many scope properties and functions which are meant for internal use, and should never be seen by the outside world. If this is the case, it’s better to have an isolated scope. The isolated scope, as expected, does not inherit the parent scope.