What is Angular?

Angular is a platform that makes it easy to build applications with the web. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop.

Angular is an open source framework written and maintained by angular team at Google and the Father of Angular is Misko Hevery (Agile Coach at Google, Attended Santa Clara University and Lives in Saratoga, CA.)

Angular is written in TypeScript and so it comes with all the capabilities that typescript offers.

What's New In Angular 7? Let's Introduce Angular 7 New Features?

Angular 7 being smaller, faster and easier to use and it will making developers life easier. This is a major release and expanding to the entire platform including -

1. Core framework,
2. Angular Material,
3. CLI

For updating Angular 6 to Angular 7, you should use a command - `ng update @angular/cli @angular/core`. Many of new things are added in Angular 7 and also many of us will curious to know these new features and bug fixes.

Let's introduce added and modified new features of Angular 7 -

1. Added a new interface - UrlSegment[] to CanLoad interface, DoBootstrap interface
2. Angular 7 added a new compiler - Compatibility Compiler (ngcc)
3. Introduce a new Pipe called - KeyValuePipe
4. Angular 7 now supporting to TypeScript 2.9.
5. Added a new elements features - enable Shadow DOM v1 and slots
6. Added a new router features - warn if navigation triggered outside Angular zone
7. A new mappings for ngfactory and ngsummary files to their module names in AOT summary resolver.
8. Added a new "original" placeholder value on extracted XMB
9. Added a new ability to recover from malformed URLs
10. Added a new compiler support dot (.) in import statements and also avoid a crash in ngc-wrapped
11. Update compiler to flatten nested template fns

Bug Fixes - There are some bug fixes are available: -

1. Now using performance.mark() instead of console.time()
2. Upgrade to trigger $destroy event on upgraded component element

Bug fixes on the core -

1. Do not clear element content when using shadow DOM
2. Size regression with closure compiler
3. Add a new hostVars and support pure functions in host bindings

Bug fixes on elements -

1. Added a new compiler dependency
2. Added a new compiler to integration

Angular Compatibility Compiler (ngcc):-

- The ngcc Angular node_module compatibility compiler - The ngcc is a tool which "upgrades" node_module compiled with non-ivy ngc into ivy compliant format.
- This compiler will convert node_modules compiled with Angular Compatibility Compiler (ngcc), into node_modules which appear to have been compiled with TSC compiler transformer (ngtsc) and this compiler conversions will allow such "legacy" packages to be used by the Ivy rendering engine.

- TSC transformer which removes and converts @Pipe, @Component, @Directive and @NgModule to the corresponding definePipe, defineComponent, defineDirective and defineInjector.

Ivy rendering engine:-

The Ivy rendering engine is a new backwards-compatible Angular renderer main focused on -

1. Speed Improvements
2. Size Reduction
3. Increased Flexibility

The template functions for creating dynamically views are no longer nested functions inside each other.
Now we use for loops that are nested inside other loops.
The following Example -

```
function AppComponent(rf: RenderFlags, ctx: AppComponent) {
    function ulTemplateFun(rf1: RenderFlags, ctx0: any) {
        function liTemplateFun(rf1: RenderFlags, ctx1: any) {...}
    }
}
```

No longer create multiple functions instances for loops that are nested inside other loops.
The following Example -

```
<ul *ngFor="let student of students">
    <li *ngFor="let subject of student'> {{ subject }} </li>
</ul>
```

DoBootstrap -

Angular 7 added a new lifecycle hook that is called ngDoBootstrap and an interface that is called DoBootstrap.

The following Example -

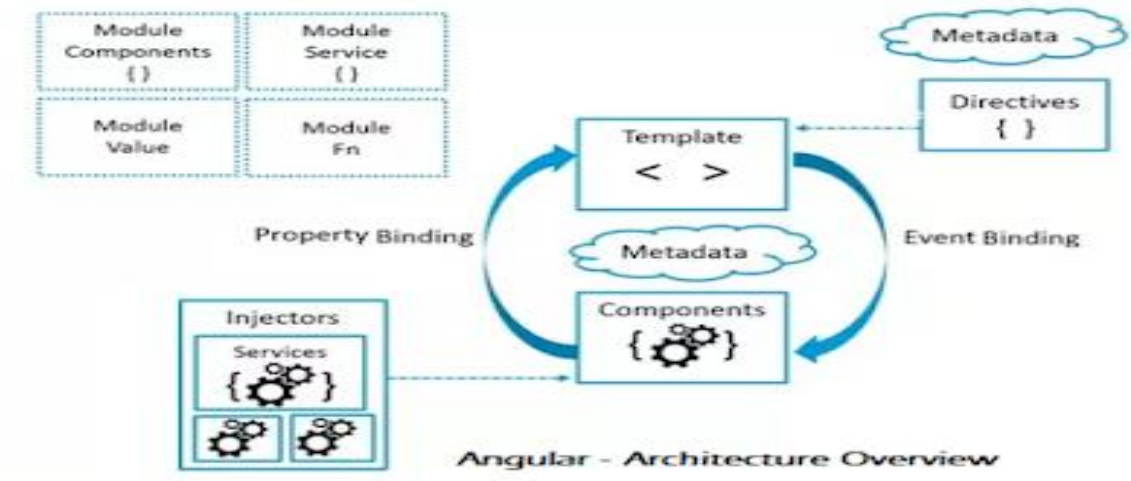
```
//ngDoBootstrap - Life-Cycle Hook Interface
class AppModule implements DoBootstrap {
    ngDoBootstrap(appRef: ApplicationRef) {
        appRef.bootstrap(AppComponent);
    }
}
```

Important Notes About Angular 7 New Features :-

Most of the sites sharing below points but these are fake points. Someone made up fake release notes as a joke but developers/peoples started treating it like it was true.

A. Splitting of @angular/core
B. A new ng-compiler
C. A new @angular/mine
D. A new @aiStore

What Is Architecture Overview of Angular?



With helps of above architecture overview, you can identify the seven main building blocks of an Angular Application.

1. Component;
2. Templates;
3. Metadata;
4. Data Binding;
5. Directives;
6. Services;
7. Dependency Injection

What Is Bootstrapping (bootstrap) in Angular?

The Bootstrap is the **root** *'AppComponent'* that Angular creates and inserts into the "**index.html**" host web page.

```
<app-root></app-root>
```

The bootstrapping process creates the components listed in the bootstrap array and inserts each one into the browser (**DOM**).

Each bootstrapped component is the base of its own tree of components. Inserting a bootstrapped component usually triggers a cascade of component creations that fill out that tree.

While you can put more than one component tree on a host web page, most applications have only one component tree and bootstrap a single root component.

This one root component is usually called *AppComponent* and is in the root module's bootstrap array.

By default Bootstrap file is created in the folder "**src/main.ts**" and "**main.ts**" file is very stable. Once you have set it up, you may never change it again and its looks like -

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
if (environment.production) {  enableProdMode(); }
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));
```

What Are Components in Angular?

Components are the most basic building block of a UI in Angular applications and it controls views (HTML/CSS). They also communicate with other components and services to bring functionality to your applications. Technically **components** are basically TypeScript classes that interact with the HTML files of the components, which get displayed on the browsers. The component is the core functionality of Angular applications but you need to know to pass the data into the components to configure them.

Angular applications must have a root component that contains all other components.

Components are created using **@Component** decorator that is part of **@angular/core module**.

The @Component decorator identifies the class immediately below it as a component class, and specifies its metadata. The metadata for a component tells Angular where to get the major building blocks that it needs to create and present the component and its view. In particular, it associates a *template* with the component, either directly with inline code, or by reference. Together, the component and its template describe a *view*.

In addition to containing or pointing to the template, the @Component metadata configures, for example, how the component can be referenced in HTML and what services it requires.

You can create your own project using Angular CLI, this command allows you to quickly create an Angular application like - generate components, services, pipes, directive, classes, and modules, and so on as per your requirements.

EX: -> `ng g c Component-Name`

The most useful **@Component** configuration options –

1. selector – It is a CSS selector that tells Angular to create an instance of this component wherever it finds the corresponding tag in template HTML. For example, it is - <app-login></app-login>
2. templateUrl – It is the module-relative address of this component's HTML template and you can also provide the inline HTML template.
3. styleUrls - It can be used for CSS rules and it will affect the style of the template elements and you can also provide the inline style CSS.

What is an entryComponent?

The entry component is used to define components and created **dynamically** using the ComponentFactoryResolver.

An entry component is any component that Angular loads imperatively, (which means you're not referencing it in the template), by type. You specify an entry component by bootstrapping it in an NgModule, or including it in a routing definition.

There are two main kinds of entry components:

- The bootstrapped root component.
- A component you specify in a route definition.

The bootstrapped entry component -

A bootstrapped component is an entry component that Angular loads into DOM at the application launch and the other root components loaded dynamically into entry components.

The angular loads a root dynamically because it is bootstrapped in the Angular Module. In the below example, AppComponent is a root component so that angular loads dynamically.

Example –

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
```

Why does Angular need entryComponents?

The entry components improve the performance, smallest, fastest and reusable code of your production apps. For example, if you want to load the smallest, fastest and reusable code in your production apps. These codes contain only the classes that you actually need and it should exclude the components that are never used, whether or not those components are declared in the apps.

As you know, many libraries declare and export components you will never use in your app. If you do not reference them, the tree shaker drops these libraries and components from the final code package.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]  // bootstrapped entry component
})
export class AppModule { }
```

If a component is not in an entry component, the compiler skips compiling for this component.

What Angular decorator?

Decorator that marks a class as an Angular component and provides configuration metadata that determines how the component should be processed, instantiated, and used at runtime. They give Angular the ability to store metadata for classes and streamline our workflow simultaneously.

There are four main types:

- **Class decorators**, e.g. @Component and @NgModule

- **Property decorators** for properties inside classes, e.g. @Input and @Output
- **Method decorators** for methods inside classes, e.g. @HostListener
- **Parameter decorators** for parameters inside class constructors, e.g. @Inject

Each decorator has a unique role

Class Decorators

Angular offers us a few class decorators. These are the top-level decorators that we use to express *intent* for classes. They allow us to tell Angular that a particular class is a component, or module, for example. And the decorator allows us to define this intent without having to actually put any code inside the class.

A @Component and @NgModule decorator example with classes:

```
import { NgModule, Component } from '@angular/core';

@Component({
  selector: 'example-component',
  template: '<div>Woo a component!</div>',
})
export class ExampleComponent {
  constructor() {
    console.log('Hey I am a component!');
  }
}
@NgModule({
  imports: [],
  declarations: [],
})
export class ExampleModule {
  constructor() {
    console.log('Hey I am a module!');
  }
}
```

Notice how both classes by themselves are effectively the same. No code is needed within the class to tell Angular that it is a component or a module. All we need to do is decorate it, and Angular will do the rest.

Property Decorators

These are probably the second most common decorators that you'll come across. They allow us to decorate specific properties within our classes - an extremely powerful mechanism.

Let's take a look at @Input(). Imagine that we have a property within our class that we want to be an input binding.

Without decorators, we'd have to define this property in our class anyway for TypeScript to know about it, and then somewhere else tell Angular that we've got a property that we want to be an input.With decorators, we can simply put the @Input() decorator above the property - which Angular's compiler will automatically create an input binding from the property name and link them.

```
import { Component, Input } from '@angular/core';

@Component({
          selector: 'example-component',
          template: '<div>Woo a component!</div>'
})
export class ExampleComponent {
@Input()
          exampleProperty: string;
}
```

We'd then pass the input binding via a component property binding:

```
<example-component  [exampleProperty] ="exampleData">
</example-component>
```

The property decorator and "magic" happens *within* the ExampleComponent definition.

In Angular there is a single property exampleProperty which is decorated, which is easier to change, maintain and track as our codebase grows.

Method Decorators

Method decorators are very similar to property decorators but are used for methods instead. This let's us decorate specific methods within our class with functionality. A good example of this is @HostListener. This allows us to tell Angular that when an event on our host happens, we want the decorated method to be called with the event.

```
import { Component, HostListener } from '@angular/core';

@Component({
selector: 'example-component',
template: '<div>Woo a component!</div>'
})
export class ExampleComponent {
@HostListener('click', ['$event'])
onHostClick(event: Event) {
// clicked, `event` available
}
}
```

Parameter Decorators

Parameter decorators are quite interesting. You may have come across these when injecting primitives into a constructor, where you need to manually tell Angular to inject a particular provider.

Parameter decorators allow us to decorate parameters in our class constructors. An example of this is @Inject that lets us tell Angular what we want that parameter to be initiated with:

```
import { Component, Inject } from '@angular/core';
import { MyService } from './my-service';

@Component({
selector: 'example-component',
template: '<div>Woo a component!</div>'
})
export class ExampleComponent {
constructor(@Inject(MyService) myService) {
console.log(myService); // MyService
}
}
```

Due to the metadata that TypeScript exposes for us we don't actually have to do this for our providers. We can just allow TypeScript and Angular to do the hard work for us by specifying the provider to be injected as the parameter *type*:

```
import { Component } from '@angular/core';
import { MyService } from './my-service';

@Component({
selector: 'example-component',
template: '<div>Woo a component!</div>'
})
export class ExampleComponent {
constructor(myService: MyService) {
console.log(myService); // MyService
}
}
```

Decorator functions

Let's quickly make a decorator that we can use on a class to demonstrate this a little further. This decorator is just going to simply log the class to the console:

```
function Console(target) {
  console.log('Our decorated class', target);}
```

Here, we have created Console (using the uppercase naming convention Angular uses) and are specifying a single argument called target. The target will in fact be the class that we decorate, which means we can now decorate any class with our decorator and see it outputted in the console:

```
@Console
```

```
class ExampleClass {
  constructor() {
    console.log('Yo!');
  }
}
```

When we use the decorators in Angular we pass in some form of configuration, specific to the decorator. For example, when we use @Component we pass through an object, and with @HostListener we pass through a string as the first argument (the event name, such as 'click') and optionally an array of strings for further variables (such as $event) to be passed through to the decorated method.

What Angular decorators actually do?

Every type of decorator shares the same core functionality. From a purely decorative point of view, @Component and @Directive both work in the same way, as do @Input and @Output. Angular does this by using a factory for each type of decorator.

Storing metadata

The whole point of a decorator is to store metadata about a class, method or property. When you configure a component for example, you're providing metadata for that class that tells Angular that we have a component, and that component has a specific configuration. Each decorator has a base configuration that you can provide for it, with some defaults applied for you. When the decorator is created using the relevant factory, the default configuration is passed through. For instance, let's take a look at the possible configuration that you can use when creating a component:

```
{
  selector: undefined,
  inputs: undefined,
  outputs: undefined,
  host: undefined,
  exportAs: undefined,
  moduleId: undefined,
  providers: undefined,
  viewProviders: undefined,
  changeDetection: ChangeDetectionStrategy.Default,
  queries: undefined,
  templateUrl: undefined,
  template: undefined,
  styleUrls: undefined,
  styles: undefined,
  animations: undefined,
  encapsulation: undefined,
  interpolation: undefined,
  entryComponents: undefined
}
```

There are a lot of different options here, and you'll notice that only one has a default value - changeDetection. This is specified when the decorator is created so we don't need to add it whenever we create a component. You may have applied this line of code to modify the change strategy:

changeDetection: ChangeDetectionStrategy.OnPush;

An annotation instance is created when you use a decorator. This merges the default configuration for that decorator (for instance the object you see above) with the configuration that you have specified, for example:

```
import { NgModule, Component } from '@angular/core';

@Component({
  selector: 'example-component',
  styleUrls: ['example.component.scss'],
  template: '<div>Woo a component!</div>',
})
export class ExampleComponent {
  constructor() {
    console.log('Hey I am a component!');
```

```
  }
}
```

Once this annotation instance has been created it is then stored so Angular can access it.

Chaining decorators

If a decorator is used on a class for the first time, it creates a new array and pushes the annotation instance into it. If this isn't the first decorator that has been used on the class, it pushes it into the existing annotation array. This allows decorators to be chained together and all stored in one place.

For example, in Angular you could do this for a property inside a class:

```
export class TestComponent {
  @Input()
  @HostListener('click', ['$event'])
  onClick: Function;
}
```

At the same time, Angular also uses the reflect API (commonly polyfilled using reflect-metadata) to store these annotations, using the class as an array. This means that it can then later on fetch all of the annotations for a specific class just by being pointed to the class.

How decorators are applied to a class?

Decorators aren't native to JavaScript just yet - TypeScript currently provides the functionality for us. This means that we can check the compiled code to see what actually happens when we use a decorator.

Take a standard, ES6 class -

```
class ExampleClass {
  constructor() {
    console.log('Yo!');
  }
}
```

TypeScript will then convert this over to a function for us:

```
var ExampleClass = (function() {
  function ExampleClass() {
    console.log('Yo!');
  }
  return ExampleClass;
})();
```

Now, if we decorate our class, we can see where the decorators are then actually applied.

```
@ConsoleGroup('ExampleClass')
class ExampleClass {
  constructor() {
    console.log('Yo!');
  }
}
```

TypeScript then outputs:

```
var ExampleClass = (function() {
  function ExampleClass() {
    console.log('Yo!');
  }
  return ExampleClass;
})();
ExampleClass = __decorate([ConsoleGroup('ExampleClass')], ExampleClass);
```

This gives us some actual context as to how our decorators are applied.

What Is Modules (@NgModule decorator)?

The NgModule is a TypeScript class and work with the @NgModule decorator function and also takes a metadata object that tells Angular how to compile and run module code. The **Angular** module helps you to organize an application into associative blocks of functionality and plays a fundamental role in structuring Angular applications.

The NgModule is used to simplify the ways you define and manage the dependencies in your applications and also you can consolidate different components and services into associative blocks of functionality.

Every Angular application should have at least one module and it contains the components, service providers, pipes and other code files whose scope is defined by the containing NgModule.

The purpose of the module is to declare everything you create in Angular and group them together. Every application has at least one Angular module, the root module that you bootstrap to launch the application. The Angular root module is called AppModule.

The module is a way to organize your dependencies for

1. Compiler
2. Dependency Injection

A module –

- Can import other modules and can expose its functionality to other modules.
- Can be loaded eagerly when the application starts or lazy loaded asynchronously by the router.
- The angular loads a root dynamically because it is bootstrapped in the Angular Module.
- An Angular app needs at least one module that serves as the root module.

You can use CLI commands to generate an app, the default AppModule is as follows –

```
ng new yourApp
```

The above CLI command is used to create a new Angular project and this CLI command automatically creates several folders and files which are necessary for project development, testing, and configuration and so on.

Angular CLI (Command Line Interface)

It is a tool to initialize, develop, scaffold and maintain Angular applications. To use this we need to install it first and it should be installed globally on your machine.

```
npm install -g @angular/cli
```

What are the @NgModule Metadata Properties? What is the difference between declarations, providers, and import in NgModule?

The @NgModule decorator identifies AppModule as a NgModule class.

The @NgModule takes a metadata object that tells Angular how to compile and launch the application.

The NgModule's important metadata properties are as follows –

```
@NgModule({
  providers?: Provider[]
  declarations?: Array<Type<any> | any[]>
  imports?: Array<Type<any> | ModuleWithProviders | any[]>
  exports?: Array<Type<any> | any[]>
  entryComponents?: Array<Type<any> | any[]>
  bootstrap?: Array<Type<any> | any[]>
  schemas?: Array<SchemaMetadata | any[]>
  id?: string
})
```

Providers – A list of dependency injection (DI) providers and it defines the set of injectable objects that are available in the injector of this module.

Declarations - A list of declarable classes, components, directives, and pipes that belong to this module. The compiler throws an error if you try to declare the same class in multiple modules.

Imports - A list of modules and it used to import the supporting modules like FormsModule, RouterModule, CommonModule, or any other custom made feature module.

Exports - A list of declarable components, directives, pipes, and modules that an importing module can be used within a template of any component.

EntryComponents - A list of components that should be compiled when this module is defined. By default, an Angular app always has at least one entry component, the root component, AppComponent.

A bootstrapped component is an entry component that Angular loads into DOM during the application launch and other root components loaded dynamically into entry components.

Bootstrap – A list of components that are automatically bootstrapped and the listed components will be added automatically to entryComponents.
Schemas - Defines a schema that will allow any non-Angular elements and properties.
Id – The Id used to identify the modules in getModuleFactory. If left undefined, the NgModule will not be registered with getModuleFactory.

Why use multiple NgModules?

Multiple NgModules provides some potential benefits. Actually, the modules help you to organize an application into associative blocks of functionality. First one is organizing an application code. If you are putting around 99 resource files in the default app module and see the happing. And the second one is - It opens the possibility of lazy loading via the router.

What Are the Purpose of @NgModule?

The NgModule is used to simplify the ways you define and manage the dependencies in your applications and also you can consolidate different components and services into cohesive blocks of functionality.
The @NgModule metadata divided into three categories as follows.

1. Static: - It is compiler configuration and configured via the declarations array.
   ```
   declarations: [], //declarations is used for configure the selectors.
   entryComponents: [], //entryComponents is used to generate the host factory.
   ```
2. Runtime: - It is injector configuration and configured via the provider's array.
   ```
   providers: [], // providers is used for runtime injector configuration.
   ```
3. Composability/Grouping: - Introducing NgModules together and configured via the imports and exports arrays.
   ```
   imports: [], // imports used for composing NgModules together.
   exports: [] //A list of declarations components, directives, and pipes classes that an importing module can use.
   ```

What Types of NgModules?

There are five types of NgModules –

1. Features Module: - The feature modules are NgModules for the purpose of organizing an application code.
2. Routing Module: - The Routing is used to manage routes and also enables navigation from one view to another view as users perform application tasks.
3. Service Module: - The modules that only contain services and providers. It provides utility services such as data access and messaging. The root AppModule is the only module that should import service modules. The HttpClientModule is a good example of a service.
4. Widget Module: - The third party UI component libraries are widget modules
5. Shared Module: - The shared module allows you to organize your application code. You can put your commonly used components, directives, and pipes into the one module and use whenever required to this module.

What Are the Types of Feature Modules?

The feature modules are modules that goal of organizing an application code. It also helps you partition the app into focused areas when you can do everything within the root module.
There are five types of feature modules which are the following-

1. Domain Feature Modules
2. Routed Feature Modules
3. Routing Modules
4. Service Feature Modules
5. Widget Feature Modules

Domain Feature Module - Domain feature modules deliver a user experience dedicated to a particular application domain like editing a customer or placing an order.

Routed Feature Module - Routed feature modules are domain feature modules that components targets of router navigation routes. A lazy-loaded routed feature module should not be imported by any module. Routed feature modules do not export anything because their components never appear in the template of an external component.

Routing Module - A routing module provides routing configuration for another module and the routing module focus on the following. A routing module should only be imported by its companion module.

1. Defines Routes
2. Adds Router Configuration to the module's imports
3. Adds service providers to the module's providers
4. A routing module doesn't have its own declarations. The components, directives, and pipes are the responsibility of the feature module and not the routing module.

Service Feature Module - Service modules provide utility services and used to communicate with the server. The HttpClientModule is a great example of a service module. The root AppModule is the single module that should import service modules.

Widget Feature Module - A widget module makes components, directives, and pipes available to external modules. The third party UI components and libraries are widget modules. Import widget modules in any module whose component templates need the widgets.

Why you use BrowserModule, CommonModule, FormsModule, RouterModule, and HttpClientModule?

BrowserModule – The browser module is imported *from @angular/platform-browser* and it is used when you want to run your application in a browser.

CommonModule – The common module is imported from *@angular/common* and it is used when you want to use directives - NgIf, NgFor and so on.

FormsModule – The forms module is imported from *@angular/forms* and it is used when you build template driven forms.

RouterModule – The router module is imported from *@angular/router* and is used for routing RouterLink, forRoot, and forChild.

HttpClientModule –The HttpClientModule is imported from *@angular/common/http* and it used to initiate HTTP request and responses in angular apps. The HttpClient is more modern and easy to use the alternative of HTTP.

What are the differences in NgModules and JavaScript Modules?

JavaScript and Angular use modules to organize code, and though they organize it differently, Angular apps rely on both.

The Angular module classes differ from JavaScript module class in three key respects:

1. An Angular module bounds declarable classes only. Declarables are the only classes that matter to the Angular.
2. Instead of defining all member classes in one giant file (as in a JavaScript module), we list the module's classes in the @NgModule.declarations list.
3. An Angular module can only export the declarable classes it owns or imports from other modules. It doesn't declare or export any other kind of class.

NgModules vs. JavaScript Modules –

- The NgModule is a TypeScript class decorated with @NgModule Decorator - is a fundamental feature of Angular. ----- JavaScript also has its own module system for managing collections of JavaScript objects. It is completely different from the NgModule system.
- In JavaScript, each file is a module and all objects defined in the file belong to that module. The module declares some objects to be public by marking them with the export keyword.
- Other JavaScript modules use import statements to access public objects from other modules.

The following is an example of specifying an export and import statements -

```
export class AppComponent {
   //...
}
```

After export your class, you can import that file code in another file.

```
import { AppComponent } from './app.component';
```

Both the JavaScript and Angular use modules to organize applications code.

What's Angular Elements? - Angular 6 Most Appreciated Features!

Angular Elements resolve the problems of code reuse across multiple frameworks and provides a great way to use Angular components in non-Angular environments.

Where would we use Angular Elements?

Rob Wormald covered three main areas with examples in his ng-conf 2018 talkon Angular Elements.

The list of browsers support for custom Angular elements -

Chrome, Opera, Safari, Firefox, Edge

What's Angular Elements?

Angular Element is a package which is part of the Angular framework- @angular/elements.

Angular elements will give you the ability to use your Angular components in other environments like a jQuery app or Vue.js app or anything else. It is very useful, especially when you working with dynamically loaded HTML code. It also offers functionality that allows you to convert a normal Angular component to a native web component.

Angular Elements give you an easy way to implement a web standard. There is no smoke and mirrors.

Custom Elements, let you create custom tags in a framework-agnostic way. They let you reuse your Angular components in any webpage. Yes, you can embed Angular Elements inside a React.js or Vue.js page without any knowledge of Angular.

It was introduced in Angular 6 and hopefully, this will improve with Angular 7 or higher versions. Angular Elements was the brainchild of Angular's and it one of the most anticipated features of Angular 6 release.

How to Create a Custom Angular Elements?

Custom Elements remove the need to rewrite a widget every-time a new framework pops up.

Simply you can create a normal Angular component with inputs & outputs and import this component inside your angular module with helps of @angular/elements. Use the Angular CLI to automatically set up your project with the correct polyfill:

```
ng add @angular/elements --name=*your_project_name*.
```

The "ng add" command is built on schematics. When you run ng add @angular/elements the CLI scans your project and updates your code to support Angular Elements. It's automatic. There's no manual configuration required. After successfully added in your project two dependencies are added to the package.json which is the following-

1. @angular/elements: "^6.0.1
2. document-register-element: ^1.7.2

Now, your elements are ready to use inside a simple HTML page-

```
<my-custom-element message="This is my custom element "></my-custom-element>
```

How do Angular Elements work?

The features of Angular Elements -

1. They are self-Bootstrapping.
2. They actually host the Angular Component inside a Custom Element.
3. They're a bridge between the DOM APIs and Angular Components.
4. Anyone can use your component without having to know Angular.

What are *Advantages of Angular Elements?*

Advantages of Angular Elements -

1. Reusability - Reuse components in across your apps
2. Widgets – You can use Angular components in other environments.
3. CMS Pages
4. And many more.

What Are Angular Directives?

Angular Directive is a TypeScript class which is declared as a @directive decorator.

The directives allow you to attach behaviour to DOM elements and the **@directive** decorator provide you an additional metadata that determines how directives should be processed, instantiated, and used at run-time.

Directive decorator and metadata Properties -

```
@Directive({
    selector?: string
    inputs?: string[]
    outputs?: string[]
    host?: {...}
    providers?: Provider[]//list of providers available to this component and its children
    exportAs?: string
    queries?: {...}// To configure queries that can be injected into the component
})
```

Inputs– The list of class property names to data-bind as component inputs

Outputs - The list of class property names that expose output events that others can subscribe too

Host– These properties use to map the class property to host element bindings for properties, events, actions, and attributes. The host looks like this -

```
@Directive({
  selector: 'button',
  host: {'(click)': 'onClick($event.target)'}
})
```

We have 3 types of Directives in Angular -

1. Component
2. Attribute Directives
3. Structural Directives

Components - The component is a directive with their own templates and it is responsible for how a component should be processed, instantiated and used at run-time.

Attribute Directives - The Attribute directive is a directive and it is responsible for change the behaviour of a specified element or component.

Structural Directives - The structural directive is a directive and it is responsible for change the DOM layout by adding, removing, and manipulating elements.

The most of the common built-in structural directives are NgIf, NgFor, and NgSwitch.

What are the differences between @Component and @Directive?

- The components are used, when you want to **create new elements in the DOM** with their own HTML template.
- The attribute directives are used, when you want to change or update the **existing elements** in the DOM.

How to Create Custom Directives?

The CLI command for creating your custom directive –

```
ng g directive myCustomDirective
```

After execute the above CLI command, created two files in the project - src/app folder

1. src/app/my-custom.directive.spec.ts
2. src/app/my-custom.directive.ts

And update files reference automatically in your project module – "src/app/app.module.ts"
Lest see in the code-sample, how it look like-

my-custom.directive.ts –

```
import { Directive } from '@angular/core';

@Directive({
  selector: '[appMyCustom]'
})
export class MyCustomDirective {
  constructor() { }
}
```

And app.module.ts –

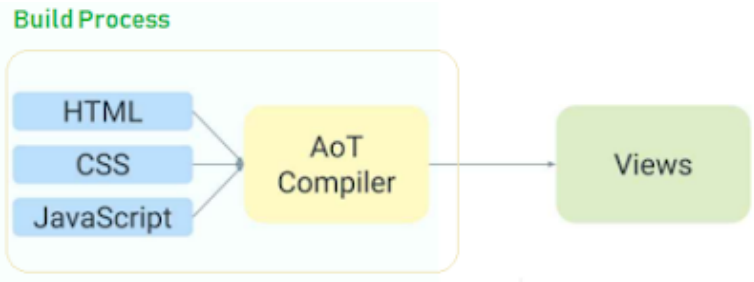
```
import { MyCustomDirective } from './my-custom.directive'

//AppModule class with @NgModule decorator
@NgModule({
  //Static, this is the compiler configuration
  //declarations is used for configure the selectors.
  declarations: [
    AppComponent,
    MyCustomDirective,
  ],
  //Composability and Grouping
  //imports used for composing NgModules together.
  imports: [
    BrowserModule
  ],
  //Runtime or injector configuration
  //providers is used for runtime injector configuration.
  providers: [],
  //bootstrapped entry component
  bootstrap: [AppComponent]
})
export class AppModule { }
```

What Is the Angular Compiler?

The **Angular** compiler converts our applications code (HTML and TypeScript) into **JavaScript** code before browser downloads and runs that code.

The @NgModule metadata plays an important role in guiding the compilation process and also tells the compiler what components to compile for this module and how to link this module with other modules.



The Angular offers two ways to compile our application code-

1. **Just-in-Time** (**JIT**) - JIT compiles our app in the browser at runtime (compiles before running).
2. **Ahead-of-Time** (**AOT**) - AOT compiles our app at build-time (compiles while running).

The **JIT** compilation is the default when we run the **build** or **serve** CLI commands -

- ng build
- ng serve

The **AOT** compilation, we append the --**aot** flags to **build** or **serve** CLI commands -

- ng build --aot
- ng serve --aot

Why we need Compilation in Angular?

We need compilation for achieving a higher level of efficiency, performance improvements, faster rendering and also sometimes detect template errors earlier in our Angular applications.

Why Compile with AOT?

1. ***Faster rendering****:* - With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.
2. ***Fewer asynchronous requests: -*** The compiler *inlines* external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.
3. ***Smaller Angular framework download size: -*** There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.
4. ***Detect template errors earlier: -*** The AOT compiler detects and reports template binding errors during the build step before users can see them.
5. ***Better security: -*** AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

What Is the difference between JIT compiler and AOT compiler?

JIT (Just-in-Time) -

1. JIT compiles our app in the browser at runtime.
2. Compiles before running
3. Each file compiled separately
4. No need to build after changing our app code and it automatically reflects the changes in your browser page
5. Highly secure
6. Very suitable for local development

AOT (Ahead-of-Time) -

1. AOT compiles our app code at build time.
2. Compiles while running
3. Compiled by the machine itself, via the command line (Faster)
4. All code compiled together, inline-ing HTML/CSS in the scripts
5. Highly secure
6. Very suitable for production builds

Angular Compiler Class –

```
class Compiler {
  // Compiles the given NgModule and all of its components
  compileModuleSync<T>(moduleType: Type<T>): NgModuleFactory<T>
    //Compiles the given NgModule and all of its components
  compileModuleAsync<T>(moduleType: Type<T>): Promise<NgModuleFactory<T>
  //creates ComponentFactories for all components
  compileModuleAndAllComponentsSync<T>(moduleType:Type<T>):
ModuleWithComponentFactories<T>

  ////creates ComponentFactories for all components
  compileModuleAndAllComponentsAsync<T>(moduleType: Type<T>):
Promise<ModuleWithComponentFactories<T>>

  //Clears all caches.
  clearCache(): void

  //Clears the cache for the given component/ngModule.
  clearCacheFor(type: Type<any>)
}
```

What Is Ivy Renderer?

The new Ivy renders and it's not stable for now and it's only in beta version. It will stable in future for production.

Ivy Renderer is new rendering engine which is designed to be backward compatible with existing render and focused to improve the speed of rendering and it optimizes the size of the final package.

The main goal of Ivy render is to speed up its loading time and reduce the bundle size of your applications. Also for uses a different approach for rendering Angular components.

For Angular, this will not be default renderer, but you can manually enable it in compiler options.

New Ivy engine in Angular 6 -

1. Smaller builds
2. Faster rebuild times
3. Faster development
4. A simpler, more hack-able pipeline
5. Human readable code

Angular 6 Bazel Compiler - What Is Bazel Compiler?

The Bazel Complier is a build system used for nearly all software built at Google.

From Angular 6 release, will start having the Bazel compiler support and when you compile the code with Bazel Compiler, you will recompile entire code base, but it compiles only with necessary code.

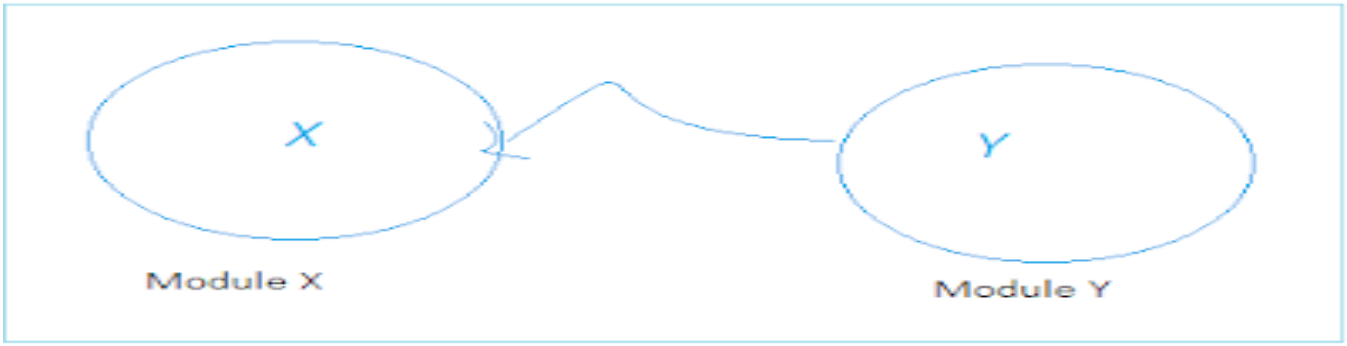
The Bazel Complier uses advanced local and distributed caching, optimized dependency analysis and parallel execution.

Bazel allows us to break an application into distinct build units. In Angular, build units are defined at the NgModule level. This means the scope of a build can be as granular as a single NgModule. If a change is internal to an NgModule, only that module needs to be rebuilt.

Angular, Angular Universal, NgRx, and Tsickle all switched to Bazel as the build tool, and ship Bazel-built artefacts to npm.

What Is a Dependency?

When module X in an application needs module Y to run, then module Y is a dependency of module X.



What Is a dependency?

When module X in an application needs module Y to run, then module Y is a dependency of module X.

What Is Dependency Injection (DI)?

Dependency Injection is a powerful pattern for managing code dependencies. DI is a way to create objects that depend upon other objects. Angular has its own DI framework pattern, and you really can't build an Angular application without Dependency injection (DI).

A DI system supplies the dependent objects when it creates an instance of an object.

Example to build a fully functional CAR.

1. Car class 2. Wheel class, 3. Headlight class 4. Outer door class 5. Inner door class 6. Glass class 7. Window class 8. Fuel level sensor class

Let's see what happen, without Dependency Injection (DI) –

To complete the CAR class, we need to import all eight classes here and make one fully functional CAR.

Now, here we have created eight classes instance in the constructor of CAR class.

Note that, the CAR class is totally dependent on these eight classes. Otherwise, it will not complete the CAR.

We are creating the instances in the CAR constructor. So Wheel, Headlight, Outer door, Inner door, Glass, Window, and Fuel are not decoupled from the CAR class.

Let's see what happen, with Dependency Injection (DI) –

If we are using Dependency Injection then, we do not need to create the instances in the constructor.

First, we need to provide all the dependencies to the "*app.module.ts*" class -

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
//Import App Component
import { AppComponent } from './app.component';
//Import CAR classes.
import {Wheel} from './car/wheel';
import {Headlight} from './car/headlight';
import {Glass} from './car/glass';
import {InnerDoor} from './car/inner-door';
import {OuterDoor} from './car/outer-door';
import {Window} from './car/window';
import {FuelLevel} from './car/fuel-level';
//AppModule class with @NgModule decorator.
@NgModule({
  //Static, This is the compiler configuration
  //declarations is used for configure the selectors.
  declarations: [
    AppComponent,
  ],
  //Composability and Grouping
  // imports used for composing NgModules together.
  imports: [
    BrowserModule
  ],
  //Runtime or injector configuration
  //providers is used for runtime injector configuration.
  providers: [Wheel, Headlight, Glass, InnerDoor, OuterDoor, Window, FuelLevel],
  //bootstrapped entry component
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Then, In the CAR class, inject those dependencies into CAR constructor –

```
import {Wheel} from './car/wheel';
import {Headlight} from './car/headlight';
import {Glass} from './car/glass';
import {InnerDoor} from './car/inner-door';
import {OuterDoor} from './car/outer-door';
import {Window} from './car/window';
import {FuelLevel} from './car/fuel-level';
export class Car {
    constructor(public wheel: Wheel,
                public headlight: Headlight,
                public glass: Glass,
                public innerdoor: InnerDoor,
                public outerdoor: OuterDoor,
                public window: Window,
                public fuellevel: FuelLevel) {}

}
```

When CAR instance is created at that time, also all the other instances of other classes are also created.

What Is Dependency Injection pattern?

DI is an application design pattern and you really cannot build an Angular application without dependency injection (DI).

What Is Injectors?

A service is just a class in Angular until you register with an Angular dependency injector. The injector is responsible for creating angular service instances and injecting them into classes. You rarely create an injector yourself and Angular creates automatically during the bootstrap process. Angular doesn't know automatically how you want to create instances of your services or injector. You must configure it by specifying providers for every service. Actually, providers tell the injector how to create the service and without a provider not able to create the service. Bootstrap defines the components that should be bootstrapped when this module is bootstrapped. The components listed here will automatically be added to entryComponents.

What Are @Injectable providers?

The @Injectable decorator identifies services and other classes that are intended to be injected. It can also be used to configure a provider for those services. To inject the service into a component, Angular provides an Injector decorator: @Injectable(). A provider defines the set of injectable objects that are available in the injector of this module. The @Injectable decorator marks a class as available to an injector for instantiation. An injector reports an error when trying to instantiate a class that is not marked as @Injectable.

Injectors are also responsible for instantiating components. At the run-time the injectors can read class metadata in the JavaScript code and use the constructor parameter type information to determine what things to inject.

Injectable decorator and metadata -

```
@Injectable({
  providedIn?: Type<any> | 'root' | null
  factory: () => any
})
```

To inject the service into a component, Angular provides an Injector decorator: @Injectable().

Here we configure a provider for CustomerService using the @Injectable decorator on the class.

We have the following steps to create a Service-

1. Create the service class
2. Define the metadata with a decorator
3. Import what we need.

In the above example, providedIn tells Angular that the root injector is responsible for creating an instance of the CustomerService. The Angular CLI sets up provider automatically when you generating a new service.

Why @Inject()?

The @Inject is a special technique for letting Angular knows that a parameter must be injected.

Inject decorator and metadata-

```
@Inject({
  token: any
})
```

When @Inject () is not present, Injector will use the type annotation of the parameter.

```
import { Component, OnInit, Inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
@Component({
  selector: 'app-customer',
  templateUrl: './customer.component.html',
  styleUrls: ['./customer.component.css']
})
export class CustomerComponent implements OnInit {
```

```
    constructor(@Inject(HttpClient) private http) {
      // use this.http which is the Http provider.
    }
    ngOnInit(){ }
}
```

At this point, @Inject is a manual way of specifying this lookup token, followed by the lowercase http argument to tell Angular what to assign it against.

What Is Hierarchical Dependency Injectors?

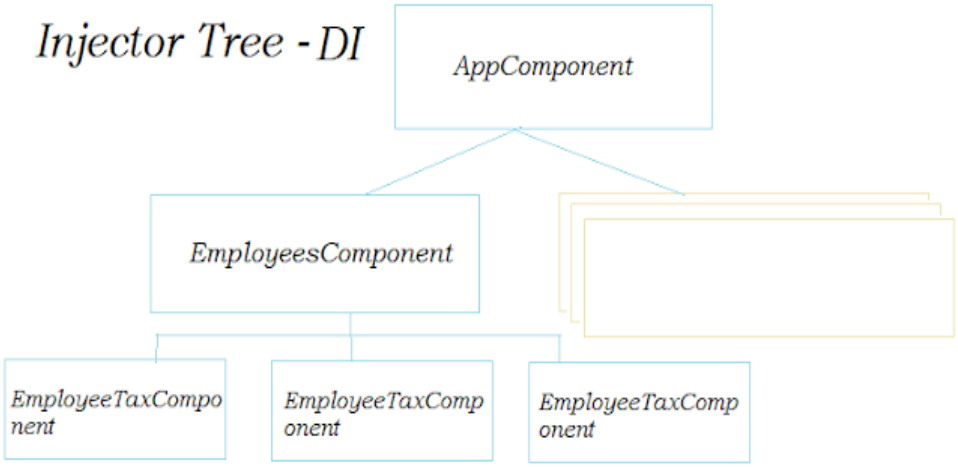
Angular has a Hierarchical Dependency Injection system. There is actually a tree of injectors that parallel an application's component tree. You can reconfigure the injectors at any level of that component tree.

What Is Injector tree?

In the Dependency Injection guide, you learned how to configure a dependency injector and how to retrieve dependencies where you need them.

An application may have multiple injectors. An Angular application is a tree of components. Each component instance has its own injector. The tree of components parallels the tree of injectors.

Three level component tree –



What Is Injector bubbling?

When a component requests a dependency, Angular tries to satisfy that dependency with a provider registered in that component's own injector. If the component's injector lacks the provider, it passes the request up to its parent component's injector. If that injector can't satisfy the request, it passes the request along to the next parent injector up the tree. The requests keep bubbling up until Angular finds an injector that can handle the request or runs out of ancestor injectors. If it runs out of ancestors, Angular throws an error.

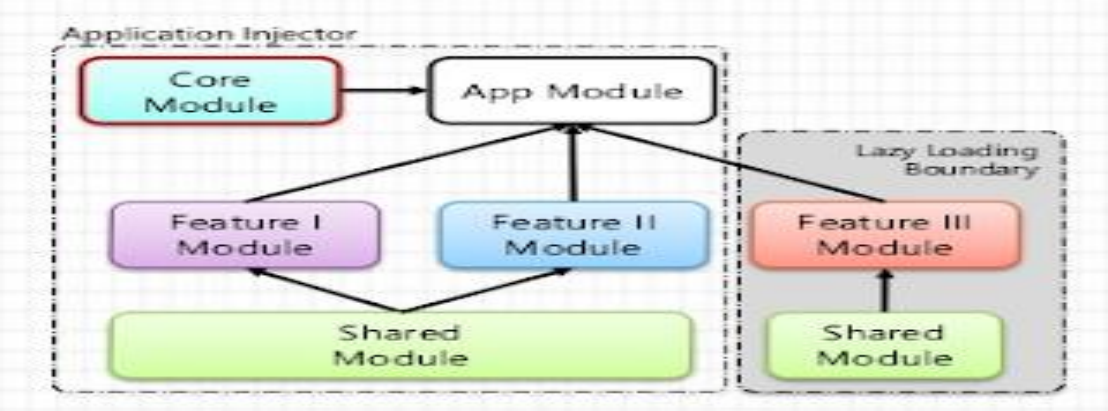
If you have registered a provider for the same DI token at different levels, the first one Angular encounters is the one it uses to provide the dependency. If, for example, a provider is registered locally in the component that needs a service, Angular doesn't look for another provider of the same service.

You can cap the bubbling by adding the @Host() parameter decorator on the dependant-service parameter in a component's constructor. The hunt for providers stops at the injector for the host element of the component.

If you only register providers with the root injector at the top level (typically the root AppModule), the tree of injectors appears to be flat. All requests bubble up to the root injector, whether you configured it with the bootstrapModule method, or registered all providers with root in their own services.

Why is it bad if a shared module provides a service to a lazy-loaded module?

The lazy loaded scenario causes your app to create a new instance every time, instead of using the singleton.



Lazy loading is the best practice of loading expensive resources on-demand. This can greatly reduce the initial startup time for single page web applications (SPA). Instead of downloading all the application code and resources before the app starts, they are fetched just-in-time (JIT), as needed.

The eagerly loaded scenario your app to create a singleton, instead of creates new instance every time.

What happen if you import the same module twice?

No problem! You can import the same module twice but Angular does not like modules with circular references and raise the circular dependency warnings on builds. Actually, the module helps you to organize an application into associative blocks of functionality. For example – Class-A and Class-B can absolutely be in the same file if needed. This warning is not an opinion on bad behavior.

What Is Angular Service?

Services are commonly used for storing data and making HTTP calls.

Angular services are singleton objects which get instantiated only once during the lifetime of an application. They contain methods that maintain data throughout the life of an application, i.e. data does not get refreshed and is available all the time. The main objective of a service is to organize and share business logic, models, or data and functions with different components of an Angular application.

An example of when to use services would be to transfer data from one controller to another custom service. The main idea behind a service is to provide an easy way to share the data between the components and with the help of dependency injection (DI) you can control how the service instances are shared.

Services use to fetch the data from the RESTful API.

Why Should We Use Services in Angular?

The separation of concerns is the main reason why Angular services came into existence. An Angular service is a stateless object and provides some very useful functions. These functions can be invoked from any component of Angular, like Controllers, Directives, etc. This helps in dividing the web application into small, different logical units which can be reused.

For example, your controller is responsible for the flow of data and binding the model to view. An Angular application can have multiple controllers, to fetch data which is required by the entire application. Making an AJAX call to the server from the controller is redundant, as each controller will use similar code to make a call for the same data. In such cases, it's extremely useful to use a service, as we can write a service which contains the code to fetch data from the server and inject the service into the controller. Services will have functions to make a call. We can use these functions of services in the controller and make calls to the server, that way we need not write the same code again and it can be used in components other than controllers as well. Also, controllers no longer have to perform the task of fetching the data, as services take care of this, thus achieving the objective of Separation of Concerns.

Deploying code to production can be filled with uncertainty. Reduce the risks, and deploy earlier and more

How to setup and create services?

Now, create a new project using the CLI command

The command for generating service class-`ng g service MyService`

It will create the two files in the folder - src/app/

1. my-service.service.spec.ts
2. my-service.service.ts

Now, import my service file into the angular module - app.module.ts file. It looks like this.

```
import { AppComponent } from './app.component';
import {MyService}  from './my-service.service';//Import Service
//AppModule class with @NgModule decorator
@NgModule({
  //Static, This is the compiler configuration
  //declarations is used for configure the selectors.
  declarations: [
    AppComponent
  ],
  //Composability and Grouping
  //imports used for composing NgModules together.
  imports: [
    BrowserModule
  ],
  //Runtime or injector configuration
  //providers is used for runtime injector configuration.
  providers: [MyService],
  //bootstrapped entry component
  bootstrap: [AppComponent]
})
export class AppModule { }
```

And let's see the created service class -my-service.service.ts

```
import { Injectable } from '@angular/core';

@Injectable()
export class MyService {

  constructor() { }
}
```

Now you need to create service methods to get, post, put, and delete the users.

```
import { Injectable } from '@angular/core';

@Injectable()
export class MyService {
  constructor() { }

  //I'm using the static data
  //You can also fetch the data using HttpClient service for backend APIs.
  users = [
     { "id": 1, "name": "Anil Singh", 'age' :32 },
     { "id": 2, "name": "Aradhya" , 'age' :32},
     { "id": 3, "name": "Reena Singh" , 'age' :32}
  ]

  /* (method) MyService.getUsers(): {
    "id": number;
    "name": string;
    'age': number;
    }[] */
  getUsers(){
    return this.users;
  }
}
```

Now, use this service in the user component for display on UI - my-user.component.ts

```
import { Component, OnInit } from '@angular/core';
import {MyService} from '../my-service.service';
@Component({
  selector: 'app-my-user',
  templateUrl: './my-user.component.html',
```

```
  styleUrls: ['./my-user.component.css']
})
export class MyUserComponent implements OnInit {
  userList =[];

  constructor(private service :MyService) {
    //Get the user list by using the my service.
    this.userList = service.getUsers();
  }

  ngOnInit() { }
}
```

And my-user.component.html// Finally, you got the user list on your components HTML.

```
<h4>User List -</h4>
<table border="1">
  <tr>
      <th>ID</th>
<th>Name</th>
<th>Age</th>
  </tr>
  <tr *ngFor="let user of userList">
    <td>{{user.id}}</td>
<td>{{user.name}}</td>
<td>{{user.Age}}</td>
  </tr>
</table>
```

What Is Singleton Service?

In Angular, two ways to make a singleton service -

1. Include the service in the AppModule
2. Declare that the service should be provided in the application root.

The preferred way to create a singleton service - Form beginning to Angular 6 is –

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class CustomerService {
}
```

Another way to create a singleton service - Include service in the AppModule

customer.service.ts –

```
import { Injectable } from '@angular/core';

@Injectable()
export class CustomersService {

  constructor() { }
}
```

And app.module.ts -

```
import {CustomerService} from './customers.service';

//AppModule class with @NgModule decorator
@NgModule({
  //Static, this is the compiler configuration
  //declarations is used for configure the selectors.
  declarations: [
    AppComponent
  ],
  //Composability and Grouping
  //imports used for composing NgModules together.
  imports: [
    BrowserModule
  ],
```

```
  //Runtime or injector configuration
  //providers is used for runtime injector configuration.
  providers: [CustomerService],
  //bootstrapped entry component
  bootstrap: [AppComponent]
})
export class AppModule { }
```

What Is Pipe?

Pipes transform displayed values within a template. A pipe class must also implements **PipeTransform** interface transform method that accepts an input value and returns the transformed result.

Use the @Pipe annotation to declare that a given class is a pipe. It allows you to define the pipe name that is globally available for use in any template in the across Angular apps.

There will be one additional argument to the transform method for each parameter passed to the pipe.

```
ng generate pipe PipeName
```

Pipe decorator and metadata –

```
@Pipe({
  name: string
  pure?: boolean
})
```

The pipe name is used for template bindings.

To use the pipe you must set a reference to this pipe class in the module.

Why use Pipes?

- Sometimes, the data is not displayed in the well format on the HTML templates that times were using pipes.
- You also can execute a function in the HTML template to get its returned value.

For example - If you want to display a credit card number on your web apps - you can't display the whole number on your web app - you should write a custom logic to display card number as like ****-****-2485 using your custom pipe.

What Is PipeTransform interface?

The Pipe class implements the PipeTransform interface that accepts input value (It is optional parameters) and returns the transformed value. The transform method is an important method to a pipe.

To create a Pipe, you must implement this interface. Angular invokes the transform method with the value of a binding as the first, and second argument in list form.

The PipeTransform interface looks like -

```
export interface PipeTransform {
    transform(value: any, ...args: any[]): any;
}
```

And it imported from Angular core -

```
import {Pipe, PipeTransform} from '@angular/core';
```

Two Categories of Pipes in Angular –

1) pure
2) impure

Every pipe has been pure by default. If you want to make a pipe impure that time you will allow the setting pure flag to false.

What Is Impure Pipe?

Angular executes an impure pipe during every component change detection cycle. An impure pipe is called often, as often as every keystroke or mouse-move.

If you want to make a pipe impure that time you will allow the setting pure flag to false.

```
@Pipe({
  name: 'currency',
  pure:false
})
```

The example for impure pipe –

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'currency',
  pure:false
})
export class CurrencyPipe implements PipeTransform {

  transform(value: any, args?: any): any {
    if (!value) {
      return '1.00';
    }

    return value;
  }
```

What Is Pure Pipe?

Angular executes a pure pipe only when it detects a pure change to the input value. A pure change can be primitive or non-primitive. Primitive data are only single values, they have not special capabilities and the non-primitive data types are used to store the group of values.

```
@Pipe({
  name: 'currency'
})
Or--
@Pipe({
  name: 'currency',
  pure: true
}
```

And another example for a pure pipe –

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'currency'
})
export class CurrencyPipe implements PipeTransform {

  transform(value: any, args?: any): any {
    if (!value) {
      return '1.00';
    }

    return value;
  }
}
```

The Pipe operator (|) –

The pipe operator is used to specify a value transformation in an HTML template or view.

What Is Parameterizing Pipe?

A pipe can accept any number of optional parameters to achieve output. The parameter value can be any valid template expressions. To add optional parameters follow the pipe name with a colon (:). Its looks like- *currency: 'INR'*

In the following example –

```
<h2>The birthday is - {{ birthday | date:"MM/dd/yy" }} </h2>
<!-- Output - The birthday is - 10/03/1984 -->
```

What Is Chaining Pipe?

The chaining Pipe is used to perform the multiple operations within the single expression. This chaining operation will be chained using the pipe (I).

In the following example, to display the birthday in the upper case- will need to use the inbuilt date-pipe and upper-case-pipe.

In the following example –

```
{{ birthday | date | uppercase}}
<!-- The output is - MONDAY, MARCH 10, 1984 -->
```

What Are Inbuilt Pipes in Angular?

Angular defines various Pipes API lists – That is called Inbuilt Pipes.

Similarly, you can also create a custom pipe (as per your needs) and configure in a module that is globally available in across angular apps.

Angular DatePipe -

The DatePipe is used to format a date with the help of locale rules.

```
{{ value_expression | date [ : format [ : timezone [ : locale ] ] ] }}
```

The Example for date pipe –

The full date provides you full date for the date. The short date converts the date to a short date and the long date provides you long date for the date.

```
<h3>{{TodayDate}}</h3>
<h3>{{TodayDate | date:'shortDate'}}</h3>
<h3>{{TodayDate | date:'longDate'}}</h3>
<h3>{{TodayDate | date:'fullDate'}}</h3>
```

Angular CurrencyPipe –

The CurrencyPipe is used to format a currency with help of locale rules.

```
{{ value_expression | currency [ : currencyCode [ : display [ : digitsInfo [ : locale ] ] ] ] }}
```

The CurrencyPipe formats a number as a currency of a specific country. It takes country currency type as a parameter.The example for the currency pipe –

```
<tr>
  <td>{{employee.salary | currency}}</td>
  <td>{{employee.salary | currency : 'INR'}}</td>
  <td>{{employee.salary | currency : 'INR' : true : '6.2'}}</td>
</tr>
```

Angular AsyncPipe –

Angular provide a special kind of pipe that are called AsyncPipe and the AsyncPipe subscribes to an observable or promise and returns the latest value it has emitted. The AsyncPipe allows you to bind your HTML templates directly to values that arrive asynchronously manner that is a great ability for the promises and observables.

The expression with Async pipe-

```
{{ obj_expression | async }}
```

OR

```
<ul><li *ngFor="let account of accounts | async">{{account.ACNo }}</li></ul>
```

The object expression can be observable, promise, null, or undefined.

The example for AsyncPipe -

```
@Component({
  selector: 'app-async-pipe',
  template:`<ul><li  *ngFor="let  account  of  accounts  |  async">    A/C  No-
{{account.ACNo }} </li></ul>`,
  styleUrls: ['./async-pipe.component.css']
})
export class AsyncPipeComponent implements OnInit {
accounts = [];//accounts declarations
apiURL:  string  =  'https://api.github.com/anilsingh/accounts/';  //fetching  json
data from Rest API

//AsyncPipe Component constructor
constructor(private accountService: AccountService) { }

//Load the account list
ngOnInit() {
    this.accountService.getAccount(this.apiURL)
                        .subscribe(data => this.accounts = data);
}
}
```

Angular PercentPipe -

Angular provides a PercentPipe and it is used to format a number as a percentage according to below rules.

The expression rule with percent -

```
{{ value_expression | percent [ : digitsInfo [ : locale ] ] }}
```

The input value to be formatted as a percentage and it can be any type. The digitsInfo is optional string parameters and by default is undefined. The locale is optional string parameters and by default is undefined.
The example as,

```
<h2>Result- {{marks | percent}}</h2>
<!-- output result is - '98%'-->
```

Angular LowerCasePipe -

Angular provides a LowerCasePipe and it is used to transforms given a text to lowercase.
The expression with lowercase -

```
{{ value_expression | lowercase }}
```

The example as,

```
import { Component } from '@angular/core';

@Component({
  selector: 'lowercase-pipe',
  template: `<div>
    <input type="text" #name (keyup)="changeLowerCase(name.value)">
    <p>LowerCase - <h2>'{{value | lowercase}}'</h2>
  </div>`
})
export class LowerCasePipeComponent {
  value: string;

  changeLowerCase(value: string) {
    this.value = value;
  }
}
```

Angular UpperCasePipe –

Angular provides an UpperCasePipe and it is used to transforms given a text to uppercase.
The expression with uppercase -

```
{{ value_expression | uppercase }}
```

The example as,

```
import { Component } from '@angular/core';

@Component({
  selector: 'uppercase-pipe',
  template: `<div>
    <input type="text" #name (keyup)="changeUpperCase(name.value)">
    <p>UpperCase - <h2>'{{value | uppercase}}'</h2>
  </div>`
})
export class UpperCasePipeComponent {
  value: string;

  changeUpperCase(value: string) {
    this.value = value;
  }}
```

Angular TitleCasePipe –

The TitleCasePipe is used to converts the text (string type data) in which the first alphabet of each word is made capital latter and the rest will be in the small case letter.
The expression with titlecase -

```
{{ value_expression | titlecase }}
```

The example as,

```
import { Component } from '@angular/core';

@Component({
  selector: 'titlecase-pipe',
  template: `<div>
    <input type="text" #name (keyup)="changetitlecase(name.value)">
    <p>titlecase - <h2>'{{value | titlecase}}'</h2>
  </div>`
```

```
})
export class titlecasePipeComponent {
  value: string;

  changetitlecase(value: string) {
    this.value = value;
  }
}
```

What Is HttpClient in Angular?

Most front-end applications communicate with backend services over the HTTP protocol. Modern browsers support two different APIs for making HTTP requests: the XMLHttpRequest interface and the fetch() API.

The HttpClient in *@angular/common/http* offers a simplified client HTTP API for Angular applications that rests on the XMLHttpRequest interface exposed by browsers. Additional benefits of HttpClient include testability features, typed request and response objects, request and response interception, Observable apis, and streamlined error handling.

What Is the role and responsibility of HttpClient?

HttpClient is performing HTTP requests and responses.

Most of all web applications communicate with backend services over the HTTP protocol and all modern browsers support two different APIs for making HTTP requests i.e.

1. XMLHttpRequest interface
2. fetch() APIs

The HttpClient is more modern and easy to use the alternative of HTTP.

HttpClient is an improved replacement for HTTP. They expect to deprecate http in Angular 5 and remove it in a later version. The new HttpClient service is included in the HTTP Client Module that used to initiate HTTP request and responses in angular apps. The HttpClientModule is a replacement of HttpModule.

HttpClient also gives us advanced functionality like the ability to listen for progress events and interceptors to modify requests or responses.

Before using the HttpClient, you must need to import the Angular HttpClientModule and the HttpClientModule is imported from *@angular/common/http*.

You must import HttpClientModule after BrowserModule in your angular apps.

First, you'll need to imported HttpClientModule from @angular/common/http in your app module and it must be import HttpClientModule after BrowserModule in your angular apps.

The following example as given below –

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';

//Import App Component
import { AppComponent } from './app.component';

//AppModule class with @NgModule decorator
@NgModule({
  imports: [
    BrowserModule,
    //import HttpClientModule after BrowserModule
    HttpClientModule,
  ],
  //Static, compiler configuration
  //declarations is used for configure the selectors
  declarations: [
    AppComponent,
  ],
  //Runtime or injector configuration
  providers: [],
  //bootstrapped entry component
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

After imported HttpClientModule into the AppModule, you can inject the HttpClient into your created service. The following example as give below–

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class CustomerService {

  //Inject HttpClient into your components or services
  constructor(private httpClient: HttpClient) { }

}
```

HttpClient supports mutating a request, which is, sending data to the server with HTTP methods such as PUT, POST, and DELETE. HttpClient is use the XMLHttpRequest browser API to execute HTTP request.
HttpClient Perform HTTP requests –

1. GET method – get()
2. POST method – post()
3. PUT method – put()
4. DELETE method – delete()
5. HEAD method – head()
6. JSONP method – jsonp()
7. OPTIONS method – options()
8. PATCH method – patch()

And the HttpClient class looks like –

```
class HttpClient {
  constructor(handler: HttpHandler)
  request(first: string | HttpRequest<any>, url?: string, options: {...}):
Observable<any>
  delete(url: string, options: {...}): Observable<any>
  get(url: string, options: {...}): Observable<any>
  head(url: string, options: {...}): Observable<any>
  jsonp<T>(url: string, callbackParam: string): Observable<T>
  options(url: string, options: {...}): Observable<any>
  patch(url: string, body: any | null, options: {...}): Observable<any>
post(url: string, body: any | null, options: {...}): Observable<any>

  put(url: string, body: any | null, options: {...}): Observable<any>
}
```

The options contain the list of parameters –

1. headers
2. observe,
3. params,
4. reportProgress,
5. responseType,
6. withCredentials

The following options parameters look like -

```
 options: {
    headers?: HttpHeaders | {
        [header: string]: string | string[];
    };
    observe?: 'body';
    params?: HttpParams | {
        [param: string]: string | string[];
    };
    reportProgress?: boolean;
    responseType: 'blob';
    withCredentials?: boolean;
  }
```

Benefits of HttpClient -

1. HttpClient include the testability features
2. HttpClient include typed request
3. HttpClient include response objects
4. HttpClient include request and response interception
5. HttpClient include Observable APIs
6. HttpClient include error handling

HttpInterceptor - HttpInterceptors is an interface which uses to implement the intercept method.
Intercepts HttpRequest and handles them.
Intercepts is an advanced feature that allows us to intercept each request/response and modify it before sending/receiving.
Interceptors capture every request and manipulate it before sending and also catch every response and process it before completing the call.

Firstly, we can implement own interceptor service and this service will "catch" each request and append an Authorization header.
You can see in the following example,

```
import {Injectable} from '@angular/core';
import {HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from
'@angular/common/http';

@Injectable()
export class MyInterceptor implements HttpInterceptor {
  //Intercepts HttpRequest and handles them.
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>
{
    const reqHeader = req.clone({headers: req.headers.set('Authorization',
'MyAuthToken')});

    return next.handle(reqHeader);
  }
}
```

After that you can configure your own interceptor service (MyInterceptor) and HTTP_INTERCEPTORS in the app Module.

```
import {NgModule} from '@angular/core';
import {HTTP_INTERCEPTORS} from '@angular/common/http';

@NgModule({
  providers: [{
    provide: HTTP_INTERCEPTORS,
    useClass: MyInterceptor,
    multi: true,
  }],
})
export class AppModule {}
```

Following this logic, Authorization token will be appended to each request. It's also possible to override request's headers by using set() method.

What's the difference between HTTP and HttpClient?

Angular HTTP vs. HttpClient –

- The HttpClient is used to perform HTTP requests and it imported form `@angular/common/http`.

The HttpClient is more modern and easy to use the alternative of HTTP.

- HttpClient is an improved replacement for Http. They expect to deprecate Http in Angular 5 and remove it in a later version.

It's an upgraded version of http from @angular/http module with the following improvements –

1. Immutable request and response objects
2. Interceptors allow middleware logic to be inserted into the pipeline

3. Progress events for both request and response
4. Typed
5. Event firing
6. Synchronous response body access
7. Support JSON body types and JSON by default and now, no need to be explicitly parsed
8. Automatic conversion from JSON to an object
9. Post request verification
10. A flush based testing framework
11. Simplified syntax for headers

The example with HttpClient -

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class CustomerService {
   //Inject HttpClient into your components or services
  constructor(private http: HttpClient) { }
}
```

And other example with Http –

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';

@Injectable()
export class CustomerService {

  //Inject HttpClient into your components or services
  constructor(private http: Http) { }
}
```

What's the difference between HttpModule and HttpClientModule?

HttpClientModule -

The HttpClientModule imported form -

```
import { HttpClientModule } from '@angular/common/http';
```

NgModule which provides the HttpClient and associated with components services and the interceptors can be added to the chain behind HttpClient by binding them to the multi-provider for HTTP_INTERCEPTORS.

HttpModule –

Http deprecate `@angular/http` in favour of `@angular/common/http`.

HttpModule imported from –

```
import { HttpModule } from '@angular/http';
```

They both support HTTP calls but HTTP is the older API and will eventually be deprecated.

The new HttpClient service is included in the HttpClientModule that used to initiate HTTP request and responses in angular apps. The HttpClientModule is a replacement of HttpModule.

What Are Angular HttpHeaders?

The Http Headers is immutable Map and each and every set() returns a new instance and applies the changes with lazy parsing. An immutable set of Http headers, with lazy parsing.

HttpHeaders Constructor -

```
constructor(headers?: string | { [name: string]: string | string[];});
```

Imports HttpHeaders from -

```
import {HttpHeaders } from '@angular/common/http';
```

HttpHeaders class contains the list of methods -

1. has() - Checks for existence of header by given name.
2. get() - Returns the first header that matches given name.
3. keys() - Returns the names of the headers
4. getAll() - Returns list of header values for a given name.
5. append() - Append headers by chaining.
6. set() - To set a custom header on the request for a given name
7. delete() - To delete the header on the request for a given name

How to set a custom header on the request?

To set a custom header on the request, firstly we need to instantiate HttpHeaders() object and pass ('header', 'value') into a function.

```
let headers = new HttpHeaders().set('Content-Type', 'text');
```

In the above example we set "Content-Type" header value to be "text" and the default header "Content-Type" is – "application/json"

It is of type immutable Map so if you assign a new value it will reinitialize the object.

```
let requestHeaders = new HttpHeaders().set('Content-Type', 'application/json');
requestHeaders = requestHeaders.set('authorization', 'Bearer ' + token);
```

We can also append headers by chaining HttpHeaders() constructor and will look like this-

```
let requestheaders = new HttpHeaders().set('Content-Type', 'application/json')
                    .set('authorization', 'Bearer ' + token);
```

And final request with custom headers will look like this –

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
@Injectable()
export class CustomerService {
    //Inject HttpClient into your components or services
  constructor(private http: HttpClient) { }
  //Set Headers
  requestHeaders = new HttpHeaders().set('Content-Type', 'text')
                .append('Authorization', 'CustomToke_AFA96A3429A9524');
    //Get Customer list
  getCustomers() {
    this.http.get('https://code-sample.com/customerjson', {
        headers: this.requestHeaders
    }).map((data:HttpEvent<object>) => { console.log(data) })
  }
}
```

How to catch and log specific Angular errors in your app?

The default implementation of ErrorHandler log error messages, status, and stack to the console. To intercept error handling, we write a custom exception handler that replaces this default as appropriate for your app.

```
import { Injectable, ErrorHandler } from '@angular/core';
import {ErrorLoggService} from './error-logg.service';

// Global error handler for logging errors
@Injectable()
export class GlobalErrorHandler extends ErrorHandler {
    constructor(private errorLogService: ErrorLoggService) {
       //Angular provides a hook for centralized exception handling.
       //constructor ErrorHandler(): ErrorHandler
        super();
    }

    handleError(error) : void {
        this.errorLogService.logError(error);
    }
}
```

How to Create a Custom ErrorHandler?

The best way to log exceptions is to provide a specific log message for each possible exception. Always ensure that sufficient information is being logged and that nothing important is being excluded.

The multiple steps involved in creating custom error logging in Angular -

1. Create a constant class for global error messages.
2. Create an error log service – ErrorLoggService.
3. Create a global error handler for using the error log service for logging errors.

Steps 1 – In the first step, we will create a constant class for logging global error messages and its look like this.

```
export class AppConstants {
  public static get baseURL(): string { return 'http://localhost:4200/api'; }
  public static get httpError(): string { return 'There was an HTTP error.'; }
  public static get typeError(): string { return 'There was a Type error.'; }
```

```
  public static get generalError(): string { return 'There was a general error.';
}
  public static get somethingHappened(): string { return 'Nobody threw an Error
but something happened!'; }
}
```

Steps 2 – In the second steps, we will create an error log service (ErrorLoggService) for error logging and it look like this.

```
import { Injectable} from '@angular/core';
import { HttpErrorResponse } from '@angular/common/http';
import{ AppConstants} from '../app/constants'

//#region Handle Errors Service
@Injectable()
export class ErrorLoggService {

  constructor() {  }

  //Log error method
  logError(error: any) {
    //Returns a date converted to a string using Universal Coordinated Time (UTC).
    const date = new Date().toUTCString();

    if (error instanceof HttpErrorResponse) {
      //The response body may contain clues as to what went wrong
      console.error(date, AppConstants.httpError, error.message, 'Status code:',
                (<HttpErrorResponse>error).status);
    }
    else if (error instanceof TypeError) {
      console.error(date, AppConstants.typeError, error.message, error.stack);
    }
    else if (error instanceof Error) {
      console.error(date, AppConstants.generalError, error.message, error.stack);
    }
    else if(error instanceof ErrorEvent){
      //A client-side or network error occurred. Handle it accordingly
      console.error(date, AppConstants.generalError, error.message);
    }
    else {
      console.error(date,      AppConstants.somethingHappened,      error.message,
error.stack);
    }
  }
}
//#endregion
```

Steps 3 – In the 3rd steps, we will create a global error handler for using the error log service for logging errors and its look like this.

```
import { Injectable, ErrorHandler } from '@angular/core';
import {ErrorLoggService} from './error-logg.service';
// Global error handler for logging errors
@Injectable()
export class GlobalErrorHandler extends ErrorHandler {
    constructor(private errorLogService: ErrorLoggService) {
       //Angular provides a hook for centralized exception handling.
       //constructor ErrorHandler(): ErrorHandler
        super();
    }
    handleError(error) : void {
        this.errorLogService.logError(error);
    }
}
```

Steps 4 – In the 4th step, we will Import global handler and error handler services in the NgModule and its look like this.

```
import {ErrorLoggService} from './error-logg.service';
```

```
import {GlobalErrorHandler} from './global-error.service';
//AppModule class with @NgModule decorator
@NgModule({
  //declarations is used for configure the selectors
  declarations: [
    AppComponent,
  ],
  //Composability and Grouping
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  //Runtime or injector configuration
  //Register global error log service and error handler
  providers: [ErrorLoggService, GlobalErrorHandler],
  //bootstrapped entry component
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Steps 5 – Finally, we got a custom error handler for log error in your application errors.

What Happens If the Request fails on the Server Due to Poor Network Connection?

HttpClient will return an error instead of a successful response.

What are the Validator functions?

There are two types of validator functions which are the following -

1. Async validators
2. Sync validators

Async validator functions that take a control instance and return an **observable** that later emits a set of validation errors or null.

Sync validator functions that take a control instance and return a set of validation errors or null.

Angular runs only Async validators due to some performance issues.

What Is a Template Reference variable?

A template reference variable is a way of capturing a reference to a specific element, component, directive, and pipe so that it can be used someplace in the same **template HTML**.

You should declare a reference variable using the **hash symbol (#)**.

The Angular components and directives only match selectors for classes that are declared in the Angular module.

Template Reference Variable Syntax –

You can use a template reference variable by two ways.

1. Using hash symbol (#)
2. Using reference symbol (ref-)

The following examples of specifying a template reference variable using Input Text Box –

I have declared a reference variable "cellnumber" using the **hash symbol (#)** and **reference symbol (ref-)**.

`<input type="text" ref-cellnumber>` //cellnumber will be a template reference variable.

And

`<input #cellnumber placeholder="Cell number">` //cellnumber will be a template reference variable.

I have created a reference to the input element that can be used later on in my template and the scope for "**cellnumber**" variable is the entire HTML template in which the reference is defined.

Here is how I could use that reference to get the value of the input for instance –

```
//cellnumber refers to the input element
<button (click)="show(cellnumber)">click to see</button>
```

In the below line of code, the variable "cellnumber" refer to the HTMLElement object instance for the input -

```
show(cellnumber: HTMLInputElement){
  console.log(cellnumber.value);
}
```

You can use the ViewChild decorator to reference it inside your component.

```
import {ViewChild, ElementRef} from '@angular/core';
```

```
// Reference cellnumber variable inside Component
@ViewChild('cellnumber') cellInputRef: ElementRef;
```

And finally, you can use this.nameInputRef anywhere inside your component class.

```
show(){
  this.contactNumber = this.cellInputRef.nativeElement.value
}
```

Template Reference Variable with NgForm –

Here we will discuss about how to access NgForm directive using template reference variable.

```
<form (ngSubmit)="onSubmitEmployee(empForm)" #empForm="ngForm">
  <label>F-Name </label><input name="f-
name" required [(ngModel)]="employee.fname">
  <label>L-Name </label><input name="l-
name" required [(ngModel)]="employee.lname">
  <label>Age    </label><input name="age" required [(ngModel)]="employee.age">
  <button type="submit" [disabled]="!empForm.form.valid">Submit</button>
</form>
```

In the above NgForm example contains a ngSubmit event and form directive.

The ngSubmit – The ngSubmit directive specifies a function to run when the form is submitted. Here on form submit onSubmitEmployee component method will be called.

The NgForm - It is a nestable alias of form directive. The main purpose of ngForm is to group the controls, but not a replacement of <form> tag.

As you know, the HTML does not allow nesting of form elements. It is very useful to nest forms.

How to bind to user input events to component event handlers?

Most of the DOM events are triggered by user input and bind to these events provides a way to get inputs from a user. The following example shows a click event binding – [on-click.component.ts]

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-on-click',
  templateUrl: './on-click.component.html',
  styleUrls: ['./on-click.component.css']
})
export class OnClickComponent implements OnInit {
  welcomeMsg = '';
  constructor() { }
  ngOnInit() { }
  onClick() {
    this.welcomeMsg = 'Welcome you, Anil!';
  }
}
```

And on-click.component.html -

```
<div class="msg">
  <button (click)="onClick()">Click Me!</button>
  <p>
    {{welcomeMsg}}
  </p>
</div>
OR
<!-- Canonical form, the (on-) prefix alternative -->
<div class="msg">
  <button on-click="onClick($event)">Click Me!</button>
  <p>
    {{welcomeMsg}}
  </p>
</div>
```

When the user clicks the button, Angular calls the onClick method from OnClickComponent.

How to get user input from the $event object?

The DOM events carry all information that is useful to the component.

The following example shows to get user input from the $event – key-up.component.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-key-up',
  templateUrl: './key-up.component.html',
```

```
  styleUrls: ['./key-up.component.css']
})
export class KeyUpComponent implements OnInit {
  values = '';
  constructor() { }

  ngOnInit() { }

  //KeyUp events.
  onKeyUp(event: any) {
    this.values += event.target.value + ' : ';
  }
}
And key-up.component.html –
<div class="event">
  <button (click)="onKeyUp($event)">KeyUp Event!</button>
  <p>
    {{values}}
  </p></div>
```

How to get user input from a template reference variable?

This is the other way to get the user data. It is also called #var. "A template reference variable is mostly a reference to a DOM element within a template. It can also be a reference to Angular components or directives and others."

It looks like this.

```
<input #name placeholder="Enter Name">
```

The following example shows to get user input from a template reference variable - template-reference.component.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-template-reference',
  templateUrl: './template-reference.component.html',
  styleUrls: ['./template-reference.component.css']
})
export class TemplateReferenceComponent implements OnInit {
  constructor() { }
  ngOnInit() {
  }
}
```

And template-reference.component.html –

```
<div class="event">
  <button #keydownVal (keydown)="0"></button>
  <p>
    {{keydownVal.value}}
  </p>
</div>
```

Angular Custom Form Validations –Email, Credit Card Number Validators

In this article, I am sharing about, how to create a custom validator for both model driven and template driven forms in Angular 5.

Two Types of Validators –

1) Built-in Validators
2) Custom Model Form Validators
   a) Email Validator
   b) Password Validator
   c) Secure Site Validator
   d) Credit card validator

Built-in Validators -

1. Validators .required - Requires a form control to have a non-empty value
2. Validators .minlength - Requires a form control to have a value of a min length
3. Validators .maxlength - Requires a form control to have a value of a max length
4. Validators .pattern - Requires a form control's value to match a given regex.
5. And so on

Built-in validator looks like –

```
this.empForm = new FormGroup({
  'email': new FormControl(this.employee.email,[Validators.required, ValidationService.emailValidator]),
  'name': new FormControl(this.employee.name, [Validators.required,Validators.minLength(4)]),
  'Dep': new FormControl(this.employee.Dep, [Validators.required, Validators.minLength(10)]),
  'Desc': new FormControl(this.employee.Desc, [Validators.required, Validators.minLength(100), Validators.minLength(500)]),
});
```

Custom Model Form Validators – Validators are core functions, they take as input a FormControl instance and returns either null if it's valid or flag for errors.

You can use the custom validator to validate a specific requirement like -

1. Email Validator
2. Password Validator
3. Secure Site Validator
4. Credit card validator
5. And may more

The Following Steps involve CREATING custom validators -

Steps 1- Create validation service using the CLI command.

```
ng g service validation
```

Steps 2 - import validation service in your app NgModule –

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule, FormGroup} from '@angular/forms';
import {RouterModule} from '@angular/router';
import {HttpClientModule} from "@angular/common/http";
//MY COMPONENTS
import { AppComponent } from './app.component';
import { LoginComponent } from './login/login.component';
import { RegisterComponent } from './register/register.component';
import { EmployeeComponent } from './employee/employee.component';

//My Services
import { AuthServiceService } from './auth-service.service';
import { AuthGuard } from './auth.guard';
import { EmployeeService} from './employee.service';
import { ValidationService } from './validation.service';

@NgModule({
  declarations: [
    AppComponent,
    LoginComponent,
    RegisterComponent,
    EmployeeComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot([
      { path: '', component: AppComponent, pathMatch: 'full' },
      { path: 'register', component: RegisterComponent },
      { path: 'employee', component: EmployeeComponent},
      { path: 'login', component: LoginComponent}])
  ],
  providers: [EmployeeService, ValidationService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Steps 3 - Write the customer validation method in your validation.service.ts -

```
import { Injectable } from '@angular/core';
@Injectable()
export class ValidationService {
  constructor() { }
```

```
  //Check Site contains SSL Security protocol  or Not.
  static secureSiteValidator(control){
    if (!control.value.startsWith('https') || !control.value.includes('.in')) {
      return { IsSecureSite: true };
    }
    return null;
  }
  //Email Validator
  static emailValidator(control) {
      if            (control.value.match(/[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-
9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-
9-]*[a-z0-9])?/)) {
          return null;
      }
      else {
          return { 'InvalidEmail': true };
      }
  }
  //Password Validator
  static passwordValidator(control) {
      if (control.value.match(/^(?=.*[0-9])[a-zA-Z0-9!@#$%^&*]{6,100}$/)) {
          return null;
      }
      else {
          return { 'InvalidPassword': true };
      }
  }
}
```

Steps 4 - Use of validation service in your components and its looks like –

```
import { Component, OnInit } from '@angular/core';
import {Employee } from '../employee'
import { Validators, FormGroup, FormControl } from '@angular/forms';
import {EmployeeService} from '../employee.service'
import { ValidationService } from '../validation.service';

@Component({
  selector: 'app-employee',
  templateUrl: './employee.component.html',
  styleUrls: ['./employee.component.css']
})
export class EmployeeComponent implements OnInit {

  constructor( public _empService: EmployeeService) { }
  empForm:any;

  ngOnInit() {
    this.empForm = new FormGroup({
      'email':      new      FormControl(this.employee.email,[Validators.required,
ValidationService.emailValidator]),
      'name':                  new                  FormControl(this.employee.name,
[Validators.required,Validators.minLength(4)]),
      'Dep':     new     FormControl(this.employee.Dep,     [Validators.required,
Validators.minLength(10)]),
      'Desc':    new    FormControl(this.employee.Desc,    [Validators.required,
Validators.minLength(100), Validators.minLength(500)]),
    });
  }

  employee = new Employee(0,'','','','','');
  submitted = false;

  //Add new Employee
  onSubmit() {
    this.submitted = true;
    let isSuccess = this._empService.addEmployee(this.employee);
    if(isSuccess){
```

```
            //handle success
            console.log(isSuccess);
           }else{
             //handle errors
           }
         }
       }
```

And

```
<div class="container">
  <h1>Employee Form</h1>
  <form #empForm="ngForm" (ngSubmit)="onSubmit()">
      <div class="form-group">
          <label for="name">Email</label>
          <input      type="text"      class="form-control"      id="email”      required
[(ngModel)]="employee.email" name="email">
        </div>
    <div class="form-group">
      <label for="name">Name</label>
      <input       type="text"       class="form-control"       id="name”       required
[(ngModel)]="employee.name" name="name">
    </div>

    <div class="form-group">
      <label for="Dep">Department</label>
      <input       type="text"       class="form-control"       id="Dep”       required
[(ngModel)]="employee.Dep" name="Dep">
    </div>

    <div class="form-group">
      <label for="Desc">Desc</label>
      <input       type="text"       class="form-control"       id="Desc”       required
[(ngModel)]="employee.Desc" name="Desc">
    </div>

    <button              type="submit"              class="btn              btn-success"
[disabled]="!empForm.form.valid">Submit</button>
  </form>

  <div [hidden]="!submitted">
    <h4 style="color:green;">Record Added Successfully!</h4>
  </div>
</div>
```

What is a Cookie?

A cookie is a small piece of data sent from a website and stored on the user's machine by the user's web browsers while the user is browsing.

OR

Cookies are small packages of information that are typically stored in your browsers and websites tend to use cookies for multiple things.

Cookies persist across multiple requests and browser sessions should you set them to and they can be a great method for authentication in some web apps.

How to install a cookie in Angular?

Install cookie -

```
npm install ngx-cookie-service –save
```

If you do not want to install this via NPM, you can run npm run compile and use the *.d.ts and *.js files in the dist-lib folder

After installed successfully, add the cookie service in the Angular module - app.module.ts

```
import {CookieService} from 'ngx-cookie-service'

//AppModule class with @NgModule decorator
@NgModule({
  //Static, this is the compiler configuration
  //declarations is used for configure the selectors.
```

```
  declarations: [
    AppComponent
],
  //Composability and Grouping
  //imports used for composing NgModules together.
  imports: [
    BrowserModule
  ],
  //Runtime or injector configuration
  //providers is used for runtime injector configuration.
  providers: [CookieService],
  //bootstrapped entry component
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Then, import and inject it into a component -

```
import { Component, OnInit } from '@angular/core';
import {CookieService} from 'ngx-cookie-service'

@Component({
  selector: 'app-on-click',
  templateUrl: './on-click.component.html',
  styleUrls: ['./on-click.component.css']
})
export class OnClickComponent implements OnInit {

  cookieValue ="";
  constructor(private cookie:CookieService) {  }

  ngOnInit() {
    this.cookie.set('cookie', 'demoApp' );
    this.cookieValue = this.cookie.get('cookie');
  }
}
```

What are the cookies methods?

Angular cookies concept is very similar to the Angular 1.x but Angular added only one extra method to delete all cookies.

The All cookie methods are

1. Check – This method is used to check the cookie existing or not.
2. Get - This method returns the value of given cookie name.
3. GetAll - This method returns a value object with all the cookies
4. Set – This method is used to set the cookies with a name.
5. Delete – This method used to delete the cookie with the given name
6. deleteAll – This method is used to delete all the cookies

and so on

Cookie Methods –

The Angular cookies service contains the following methods.

```
export declare class CookieService {
    private document;
    private documentIsAccessible;
    constructor(document: any);
    /**
     * @param name Cookie name
     * @returns {boolean}
     */
    check(name: string): boolean;
    /**
     * @param name Cookie name
     * @returns {any}
     */
```

```
    get(name: string): string;
    /**
     * @returns {}
     */
    getAll(): {};
    /**
     * @param name    Cookie name
     * @param value   Cookie value
     * @param expires Number of days until the cookies expires or an actual `Date`
     * @param path    Cookie path
     * @param domain  Cookie domain
     * @param secure  Secure flag
     */
    set(name: string, value: string, expires?: number | Date, path?: string,
domain?: string, secure?: boolean): void;
    /**
     * @param name   Cookie name
     * @param path   Cookie path
     * @param domain Cookie domain
     */
    delete(name: string, path?: string, domain?: string): void;
    /**
     * @param path   Cookie path
     * @param domain Cookie domain
     */
    deleteAll(path?: string, domain?: string): void;
    /**
     * @param name Cookie name
     * @returns {RegExp}
     */
    private getCookieRegExp(name);
}
```

How to set in Angular cookies, type number values?

In typescript you can use the 'any' type to store values to get around being specific with strong types.

```
var myInteger: any;
var myString: any;

myInteger = 1 //valid
myString = "helloWorld" //valid
```

As ngx-cookie put method takes a string you would need to pass a string. I would call JSON.stringify on your object and pass that to the put request.

```
saveSearchCritereaCookie = new myCookieObject();
var searchCookie: String = JSON.stringify(saveSearchCritereaCookie);

* @param {string} key Id for the `value`.
 * @param {string} value Raw value to be stored.
 * @param {CookieOptions} options (Optional) Options object.
 */

put(key, searchCookie, options?): void;
```

Why is Token Based Authentication more preferable Then Cookie based?

The cookie-based authentication has been the default and the cookie-based authentication is stateful.

What is Stateful?

Keep and track the previously stored information which is used for a current transaction. A stateful service based on HTTP cookies uses the HTTP transport protocol and its ability to convey cookies, used as session context.

What are the Cookies Limitations?

We can only store around 20 cookies per web server and not more than 4KB of information in each cookie and they can last indefinitely should you choose to specify the max-age attribute.

Token Based Authentication -

The Token-based authentication has received expansion over last few years due to RESTful Web APIs, SPA and so on. The Token based authentication is stateless.

What is Stateless?

Every transaction is performed as if it was being done for the very first time and there is no previously stored information used for the current transaction.

Token Based Authentication steps -

A user enters their login credentials and the server verifies the entered credentials. Validating to the entered credentials, It's correct or not. If the credentials are correct, returns a signed token. This token is stored in local storage on the client side. We can also store in session storage or cookie.

Advantages of Token-Based Authentication -

1. Stateless,
2. Scalable
3. Decoupled
4. JWT is placed in the browsers local storage
5. Protect Cross Domain and CORS
6. Store Data in the JWT
7. Protect XSS and XSRF Protection

Where to Store Tokens?

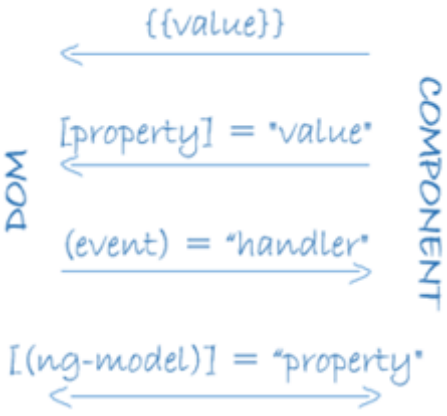
It does depend on you, where you want to store the JWT. The JWT is placed in your browsers local storage.

Data binding

Without a framework, you would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates. Writing such push and pull logic by hand is tedious, error-prone, and a nightmare to read, as any experienced jQuery programmer can attest.

Angular supports *two-way data binding*, a mechanism for coordinating the parts of a template with the parts of a component. Add binding markup to the template HTML to tell Angular how to connect both sides.

The following diagram shows the four forms of data binding markup. Each form has a direction: to the DOM, from the DOM, or both.



This example from the HeroListComponent template uses three of these forms.

src/app/hero-list.component.html (binding)

```
<li>{{hero.name}}</li>
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
<li (click)="selectHero(hero)"></li>
```

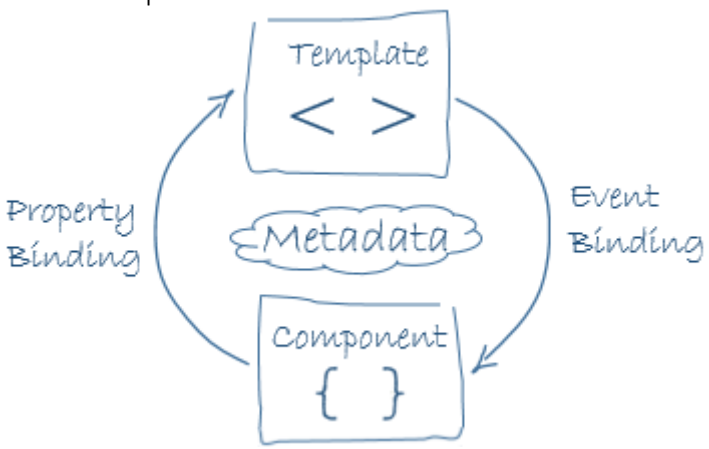
- The `{{hero.name}}` *interpolation* displays the component's `hero.name` property value within the `<li>` element.
- The `[hero]` *property binding* passes the value of `selectedHero` from the parent `HeroListComponent` to the `hero` property of the child `HeroDetailComponent`.
- The `(click)` *event binding* calls the component's `selectHero` method when the user clicks a hero's name.

Two-way data binding (used mainly in template-driven forms) combines property and event binding in a single notation. Here's an example from the HeroDetailComponent template that uses two-way data binding with the ngModel directive.

src/app/hero-detail.component.html (ngModel)

```
<input [(ngModel)]="hero.name">
```

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.
Angular processes *all* data bindings once for each JavaScript event cycle, from the root of the application component tree through all child components.



Data binding plays an important role in communication between a template and its component, and is also important for communication between parent and child components.