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 -

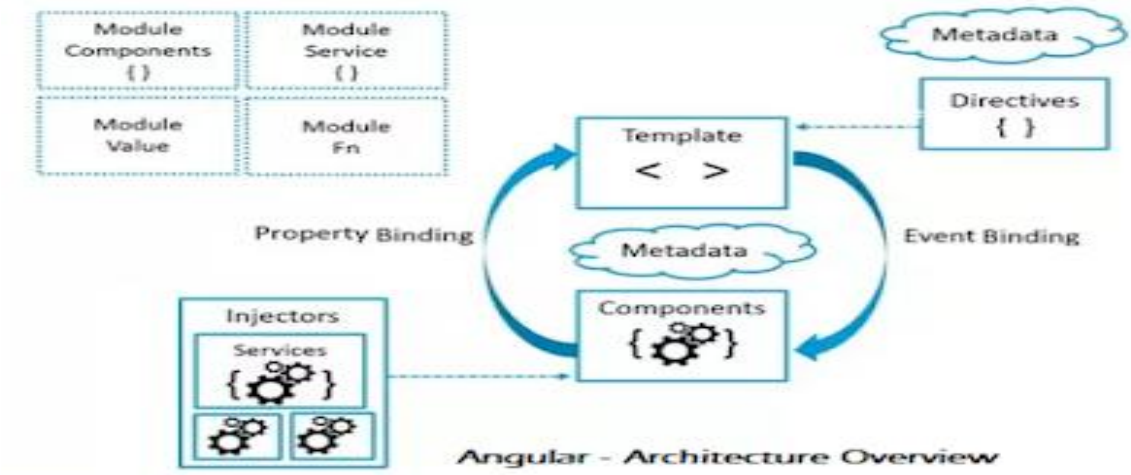
```
//ngDoBootstrap - Life-Cycle Hook Interface
class AppModule implements DoBootstrap {
    ngDoBootstrap(appRef: ApplicationRef) {
        appRef.bootstrap(AppComponent);
    }
}
```

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 –

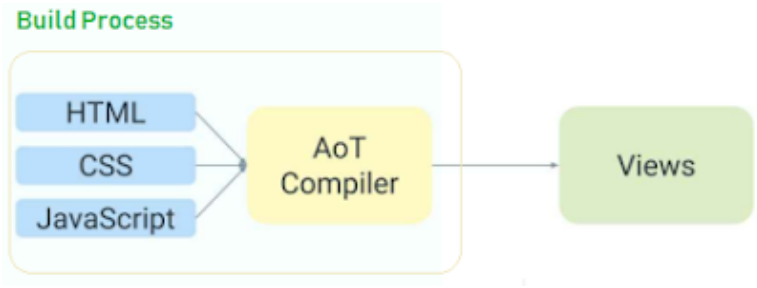
```
import { MyCustomDirective } from './my-custom.directive'

//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.

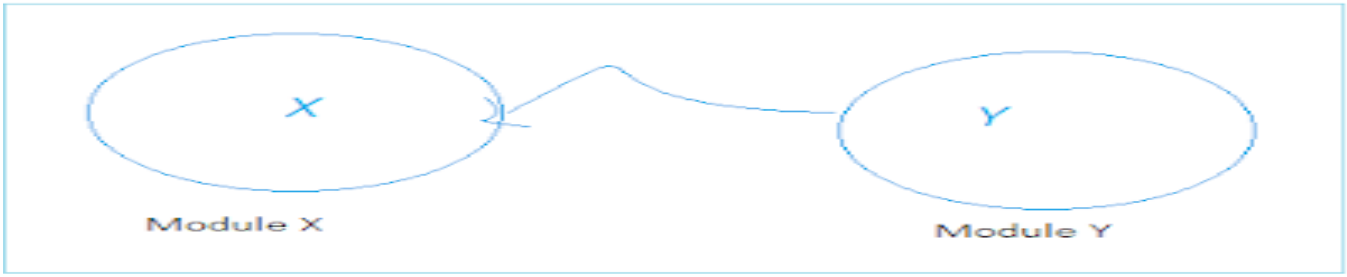
The Bazel Complier uses advanced local and distributed caching, optimized dependency analysis and parallel execution.

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?

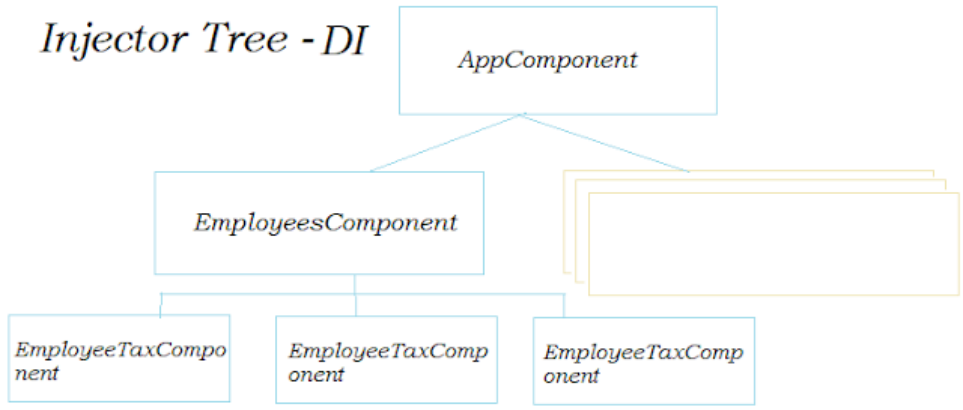
Angular has a Hierarchical Dependency Injection system. There is actually a tree of injectors that parallel an application's component tree. You can reconfigure the injectors at any level of that component tree.

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.

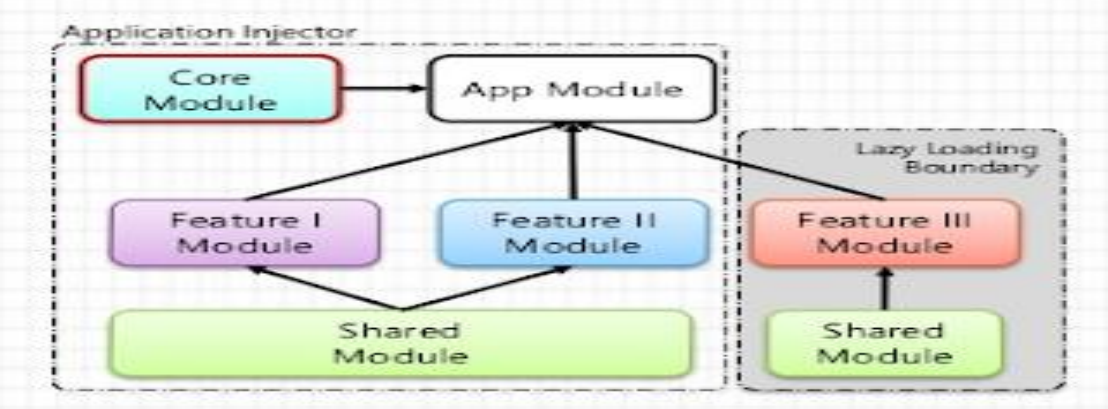
If you have registered a provider for the same DI token at different levels, the first one Angular encounters is the one it uses to provide the dependency. If, for example, a provider is registered locally in the component that needs a service, Angular doesn't look for another provider of the same service.

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.