Introduction

What is Angular 2?

Angular 2 is the latest version of Google's massively popular AngularJS framework for building complex applications in the browser (and beyond). Unlike its predecessor, Angular 2 is built upon a component model and is packed with some amazing features like dependency injection, routing mechanism and ultra fast change detection.
Angular 2 is cross platform, meaning, with Angular 2, you can not only create web apps, but also mobile and desktop apps. This is achieved with the help of strategies from Ionic Framework, NativeScript, and the ability to access native OS APIs.

Angular turns your templates into code that's highly optimized for today's JavaScript virtual machines, giving you all the benefits of hand-written code with the productivity of a framework.

With Angular, we can quickly create UI views with its simple and powerful template syntax. And with its command line tools like Angular CLI, we can build our applications even faster and deployment is a breeze.

Architecture of an Angular 2 Application

Angular 2 has a whole new model of writing apps and understanding it's building blocks will help us get up to speed faster. In this post, let's take a look at some of the important parts of Angular 2 architecture.

Writing an application in Angular 2 typically involves using the following in one way or the other:

1. Modules
2. Components
3. Templates
4. Metadata
5. Data binding
6. Directives
7. Services
8. Dependency injection

Let's try to understand each one of these building blocks in detail.

Modules

If you plan to write Angular 2 app in plain old JavaScript, you wouldn't need the module system. But, if you plan to develop apps using TypeScript, modular design is highly recommended.

A Module in Angular 2 is a block of code dedicated to a single purpose. A module typically exports a component class. We will see more about components in the next section but for now it is enough to know that a component class is what we would export from a module.

Consider the following line of code

```
export class AppComponent { }
```

In the above code, *export* statement tells TypeScript that this is a module whose *AppComponent* class is public and accessible to other modules of the application.

Now, to refer to this component in another part of the application we would use *import.* You would find below code in ***app/main.ts***

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

The *import* statement tells the system that it can get *AppComponent* from a module named *app.component.*

`The module name or id is often the same as the filename without its extension.`

In short, an Angular 2 module is a cohesive block of code dedicated to a single purpose. Modules export things — classes, function, values — that other modules import. It is recommended to write our application as a collection of modules, each module exporting one thing

Some modules are a *collection* of other modules called libraries. Angular itself ships as a collection of library modules within several npm packages. Their names begin with the @angular prefix. Through out the application source code you would find that we have imported a number of Angular libraries. The syntax for importing Angular libraries is as follows:

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

Components

Consider a view that can be divided into a number of parts with each part having some specific functionality. We have seen a number of such scenarios for example, an e-commerce website. Top bar, bottom bar, cart display, product carousel are all different parts of the same view with some specific functionality. In Angular 2 each of those parts can be built into *components*.

A component typically consists of meta data information (templateUrl, styleUrls, selector, etc.) and a class with application logic. The class interacts with the view through an API of properties and methods.

Following is a simple Component that renders **GeekHours**, and a button that triggers a method to print the tagline on to the console

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


@Component({
  selector: 'geek-hours-component',
  template: '<div>{{websitename}}. <button (click)="printTagLine()">
Print Tag Line</button></div>'
})
export class MyComponent {
  constructor() {
    this.websitename = 'GeekHours'
  }
  printTagLine() {
    console.log('Better be a geek than an idiot!')
  }
}
```

When we use the `<geek-hours-component></geek-hours-component>` tag in our HTML, this component will be created, our constructor called, and rendered.

Templates

Templates in Angular 2 is very similar to the ones in Angular 1.x but there are some syntactical changes that make it more clear of what exactly is happening. Let's try to understand them with an example:

```
<h2>Movies</h2>
<p><i>Select a movie from the list</i></p>
<ul>
  <li *ngFor="let movie of moviess" (click)="selectMovie(movie)">
    {{movie.name}}
  </li>
</ul>
<input [(ngModel)]="selectedMovie">
<movie-detail *ngIf="selectedMovie" [movie]="selectedMovie"></movie
-detail>
```

The above code is a template and we see some new syntaxes:

Rendering

To render a value, we can use the double-curly syntax

```
{{movie.name}}
```

Binding Properties

To resolve and bind a variable to a component, we use the [] syntax.

```
<movie-detail *ngIf="selectedMovie" [movie]="selectedMovie">
</movie-detail>
```

Handling Events

To listen for an event on a component, we use the () syntax

```
<li *ngFor="let movie of moviess" (click)="selectMovie(movie)">
```

Two-Way data binding

To keep a binding up to date given user inputs and other events, we use the [()] syntax.

```
<input [(ngModel)]="selectedMovie">
```

The Asterisk

* indicates that this directive treats this component as a template and will not draw it as-is. For example, ngFor takes our <movie-detail> and stamps it out for each item in items, but it never renders our initial <movie-detail> since it's a template.

```
<movie-detail *ngIf="selectedMovie" [movie]="selectedMovie">
</movie-detail>
```

Metadata

We tell Angular that a class is a component by attaching metadata to it. In TypeScript, we attach metadata by using a decorator. Here's an example:

```
@Component({
  selector:    'movie-list',
  templateUrl: 'app/movie-list.component.html',
  providers:   [ MovieService ]
})
export class MovieListComponent implements OnInit {
/* . . . */
}
```

Here are a few of the frequently used @Component configuration options:

1. **`selector`**: CSS selector that tells Angular to create and insert an instance of the component.
2. **`templateUrl`**: address of this component's template.
3. **`directives`**: array of the components or directives that this template requires.
4. **`providers`**: array of dependency injection providers for services that the component requires.
5.

Data Binding

Angular supports data binding, a mechanism for coordinating parts of a template with parts of a component. We add binding markup to the template HTML to tell Angular how to connect both sides. There are four forms of data binding as explained in the "Templates" section.

1. Interpolation **{{}}**
2. Property Binding **[]**
3. Event binding **()**
4. Two-way data binding **[()]**

Directives

When Angular renders templates, it transforms the DOM according to the instructions given by directives.

A directive is a class with directive metadata. In TypeScript we apply the `@Directive` decorator to attach metadata to the class.
There are three forms of directives:

Component :
A component is a *directive-with-a-template*; a @Component decorator is actually a @Directive decorator extended with template-oriented features.

Structural :
Structural directives alter layout by adding, removing, and replacing elements in DOM. **ngFor* and **ngIf* are examples of built-in structural directives.

Attribute :
Attribute directives alter the appearance or behavior of an existing elements. In templates they look like regular HTML attributes, hence the name. The `ngModel` directive, which implemtnts two-way data binding, is an example of an attribute directive.

Services

A service is typically a class with a narrow, well-defined purpose. A service can be a value, function, or feature that our application needs and majorly consumed by our components. When designing our components we have to make sure that they do not fetch data from the server or validate user input. Such tasks are delegated to services.

Angular makes it easy to factor our application logic into services and make those services available to components through *dependency injection*.
Some examples of services include:

- logging service
- data service
- message bus
- tax calculator
- application configuration

Below is an example of how a logger service might look like:

```
export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}
```

Dependency Injection

Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

A provider in Angular can create or return a service, typically the service class itself. We can register providers in modules or in components.

We often add providers to the root module so that the same instance of a service is available everywhere.

```
providers: [
  BackendService,
  MovieService,
  Logger
],
```

We can also register services at a component level in the providers property of the `@Component` metadata:

```
@Component({
  selector:    'movie-list',
  templateUrl: 'app/movie-list.component.html',
  providers:   [ MovieService ]
})
```

Is it Angular 2 or AngularJS 2?

I have noticed that a number of people are still confused on the name of the framework. Some call it AngularJS 2 while others call it Angular 2. So, who is right? In this post, let's clear the clouds around this.

In 2014, when Google announced that it has started working on Angular 2, it re-branded and re-purposed the framework. The 'JS' letters which stood for JavaScript were dropped from its brand, as TypeScript became its preferred dialect.

And since Angular 2 applications can be built using TypeScript, Dart, or JavaScript, this super heroic framework is now just called Angular or Angular 2, and definitely not AngularJS 2.

Bibliography

- What is Angular 2? http://www.geekhours.com/2016/11/03/what-is-angular-2/
- Architecture of an Angular 2 Application : http://www.geekhours.com/2016/08/24/building-blocks-angular-2-application
- Is it Angular 2 or Angular JS? http://www.geekhours.com/2017/02/15/angular-2-angularjs-2/