Day 13

Angular

Agenda

- Introduction to Angular
- Angular Versions
- Angular Features
- Development Setup
- Structure of Project
- Angular Architecture
  - Modules
  - Components
  - Templates
  - Metadata
- Data Binding
  - Interpolation
  - Property Binding
  - Interpolation Vs Property Binding
  - Event Binding
  - Two- way Data Binding
- Directives
  - Structural Directives
  - Attribute Directives
  - Custom Directives

Introduction

- Angular 2 is a JavaScript based open-source framework for building client-side web application.
- It is maintained by Google.
- It is a more streamlined framework that allows programmers to focus on simply building JavaScript classes.
- Angular 2 is a complete rewrite from the same team that built AngularJS but it is completely different from AngularJS
- Improved on that functionality and made it faster, more scalable and more modern.
- It empowers developers to build applications that live on the web, mobile, or the desktop
- Angular 2 and TypeScript are bringing true object oriented web development to the mainstream

Angular Versions

Angular Version	Release Date
AngularJS	October 20, 2010
Angular 2	September 14, 2016
Angular 4	March 23, 2017
Angular 5	November 1, 2017
Angular 6	May 4, 2019

Angular 2 is based on

- ES6/ Typescript
- DOM
- Web Components
- Zone.js
- RxJS
- Observables
- Module Loaders
- Angular 1.x

Angular 2 is based on Cont..

- Web Components
  - Web components are a set of standard APIs that make it possible to natively create custom HTML tags that have their own functionality and component lifecycle. The main goal of web components is to encapsulate the code for the components into a nice, reusable package for maximum interoperability.
- Zone.js
  - In Angular 1 you have to tell the framework that it needs to run this check by doing scope.$apply. You don’t need to worry about it in Angular 2. Angular 2 uses Zones to know when this check is required. This means that you do not need to call scope.$apply to integrate with third-party libraries.

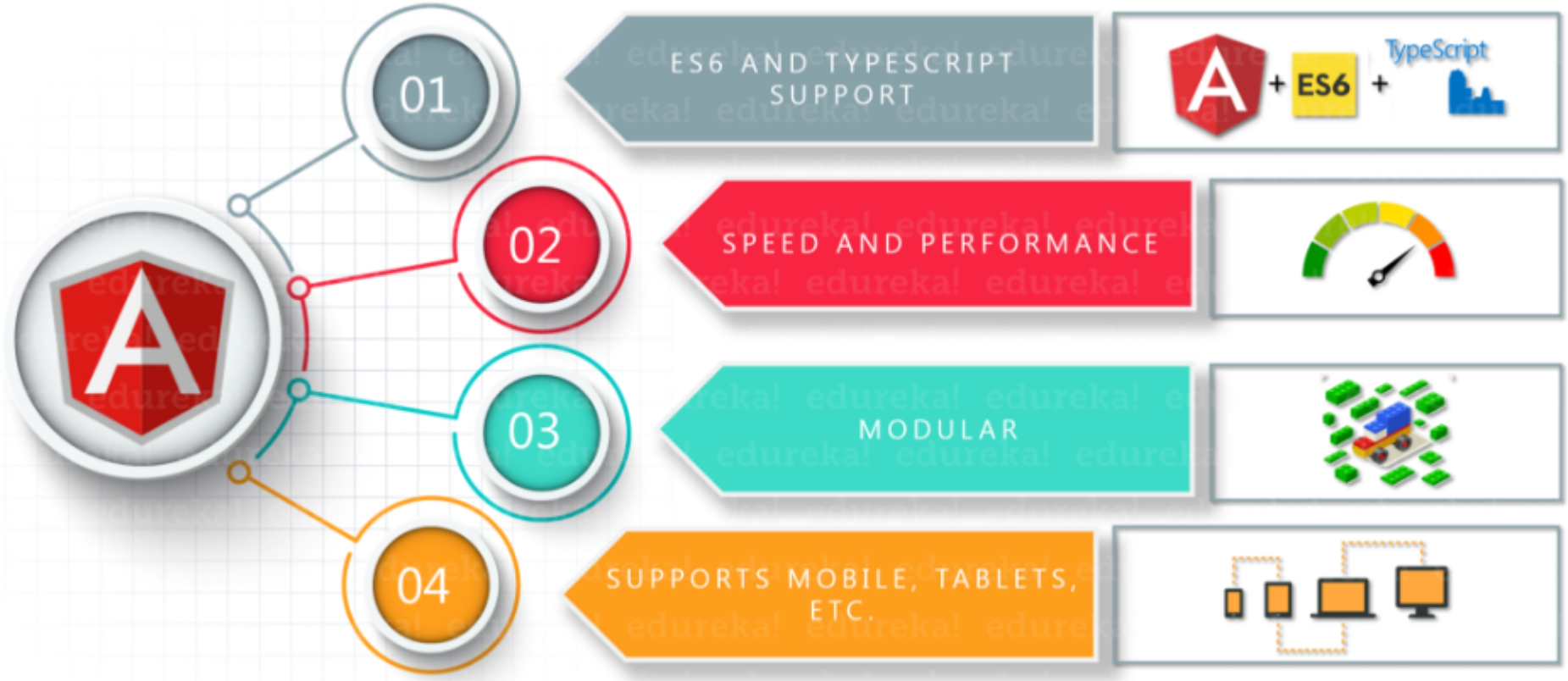
Angular 2 is based on Cont..

- RxJS
  - Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code
- Observable
  - Observables open up a continuous channel of communication in which multiple values of data can be emitted over time. From this we get a pattern of dealing with data by using array-like operations to parse, modify and maintain data. Angular uses observables extensively - you'll see them in the HTTP service and the event system.

Angular 2 is based on Cont..

- Module Loaders
  - Modules allow code to be compartmentalized to provide logical separation for the developers. We need a way to have many Angular files in dev, but they need to be loaded into the browser in bulk (not a script tag for each one).
  - This is why people look to module loaders. Typescript uses the ES6 import and export syntax to load modules. Angular uses following Module loaders
    - SystemJS
    - Webpack
    - RequireJS

Angular Features



Cross Platform

- Angular use modern web platform capabilities to deliver app-like experiences.
- High performance and zero-step installation.
- Angular 2 was designed for mobile from the ground up. Aside from limited processing power, mobile devices have other features and limitations that separate them from traditional computers.
- Create desktop - installed apps across Mac, Windows, and Linux.

Speed and Performance

- Performance in Angular 2 is much improved — thanks to the fast change detection.
- The support for dynamic loading and asynchronous templating are features that help in improving the page load and response times considerably.
- Serve the first view of your application on node.js, .NET, PHP, and other servers for rendering in just HTML and CSS.
- Angular turns our templates into code that's highly optimized for today's JavaScript machines.
- Angular apps load quickly with the new Component Router.

Productivity

- Quickly create UI views with simple and powerful template syntax.
- Command line tools:
  - Start building fast
  - Add components and tests
  - Then instantly deploy.

- Get intelligent code completion, instant errors in popular editors and IDEs.

Development

- Build features quickly with simple, declarative templates. Extend the template language with your own components.
- Angular was written from the ground up to be testable using Karma.
- Protractor makes our scenario tests run faster and in a stable manner.
- Create high-performance, complex choreographies and animation timelines with very little code through Angular's intuitive API.

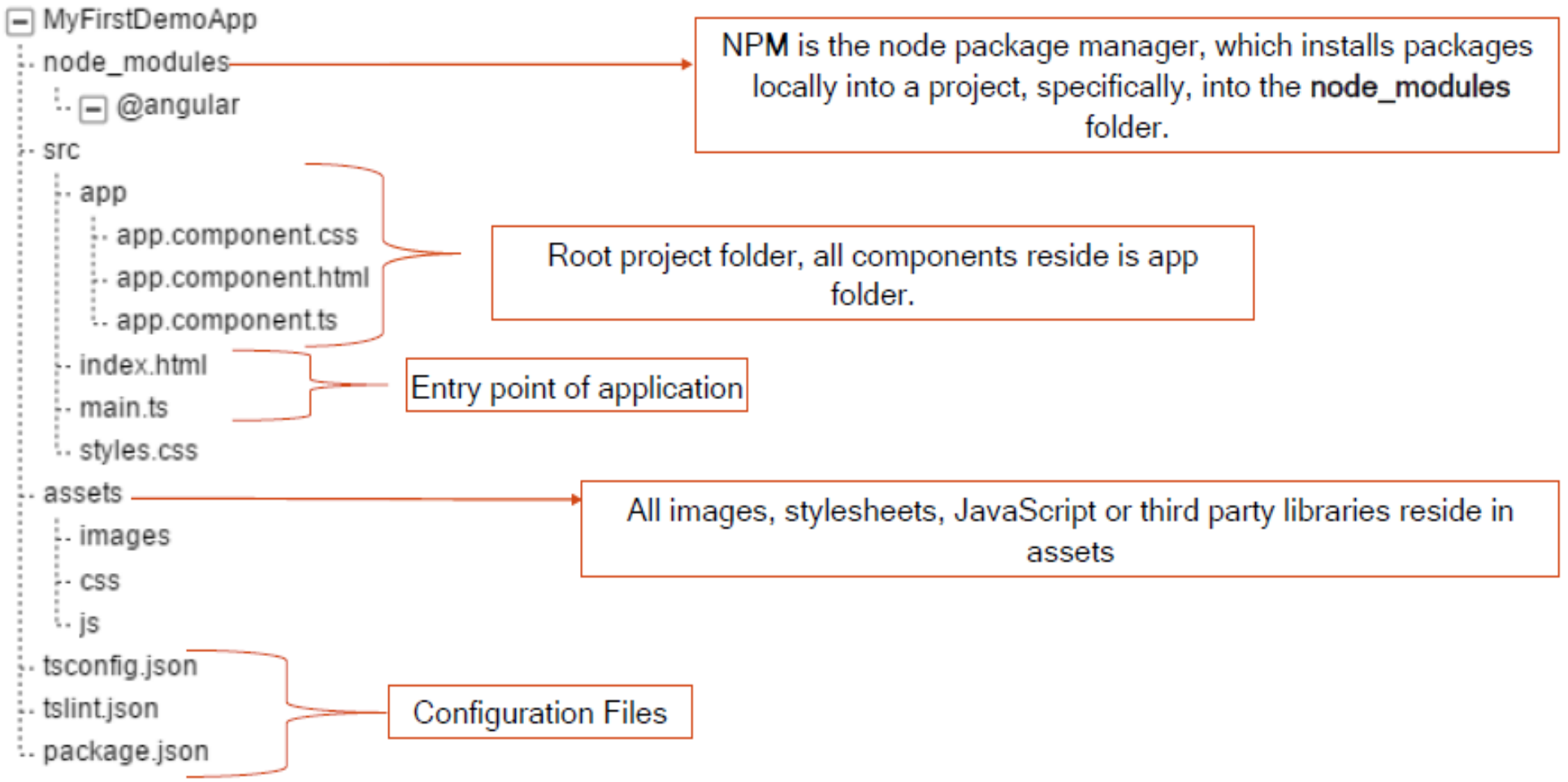
Development Environment Setup

- **The first step is to install Node.js and npm**
  - https://nodejs.org/en/download
    - Node.js is an open-source, cross-platform JavaScript run-time environment for executing JavaScript code server-side.
    - Download latest version of Node.js
- **Why it is Required**
  - We are going to use NPM to install Angular, TypeScript, SystemJS and any other packages/modules required by our application. NPM is can be used to upgrade these packages as and when necessary. Without NPM, we have to download and install all these packages manually.

Development Environment Setup

- Check version of Node.js and npm.
  - **Syntax : node –v**
  - **Syntax : - npm –v**
- Download the "Quick Start Files" from the below link
  - https://github.com/angular/quickstart
- Install Packages
  - **Syntax : npm install**
    - The NPM requires the Package.json file, which should contain the list of modules/packages used in your Application. We need to add all the list of dependencies required by our application the configuration file.

Structure of project



Structure of project Cont..

- **e2e**
  - This folder contains the files required for end to end test by protractor. Protractor allows us to test our application against real browser. You can learn more about protractor from this **link**
- **node_modules**
  - All our external dependencies are downloaded and copied here by NPM Package Manager.
- **src**
  - This where our application lives.

tsconfig.json

- If you're working on a large project with many .ts files, it may be helpful to create a tsconfig.json file.
- The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.
- A project is compiled in one of the following ways:
  - By invoking tsc with no input files, in which case the compiler searches for the tsconfig.json file starting in the current directory and continuing up the parent directory chain.
  - By invoking tsc with no input files and a --project (or just -p) command line option that specifies the path of a directory containing a tsconfig.json file, or a path to a valid .json file containing the configurations.

Example tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [ "es2015", "dom" ],
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true
  }
}
```

Example tsconfig.json

- **CompilerOptions:**
  - You can customize the compiler options using **compilerOptions**
- **Target:**
  - Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2019', 'ES2016', 'ES2017', or 'ESNEXT'.
- **moduleResolution:**
  - Specify module resolution strategy: 'node' (Node.js) or 'classic' (TypeScript pre-1.6).
- **sourceMap:**
  - Generates corresponding '.map' file.

Example tsconfig.json

- **emitDecoratorMetadata:**
  - Enables experimental support for emitting type metadata for decorators.
- **experimentalDecorators:**
  - Enables experimental support for ES7 decorators
- **lib:**
  - Specify library files to be LLCluded in the compilation
- **noImplicitAny:**
  - Raise error on expressions and declarations with an implied 'any' type.
- **suppressImplicitAnyIndexErrors**
  - Suppress --noImplicitAny errors for indexing objects lacking index signatures.

TSLint

- Linting is among the most common and helpful types of static analysis for JavaScript. It can help catch bugs, enforce uniform code style, and prevent overly complex code.
- TSLint is a tool you can use to ensure that your programs are written according to specified rules and coding styles.
- You can configure TSLint to check that the TypeScript code in your project is properly aligned and indented, that the names of all interfaces start with a capital *I*, that class names use CamelCase notation, and so on.

Configuration

- **Npm install tslint** command creates basic configuration file **tslint.json** file that looks like this:

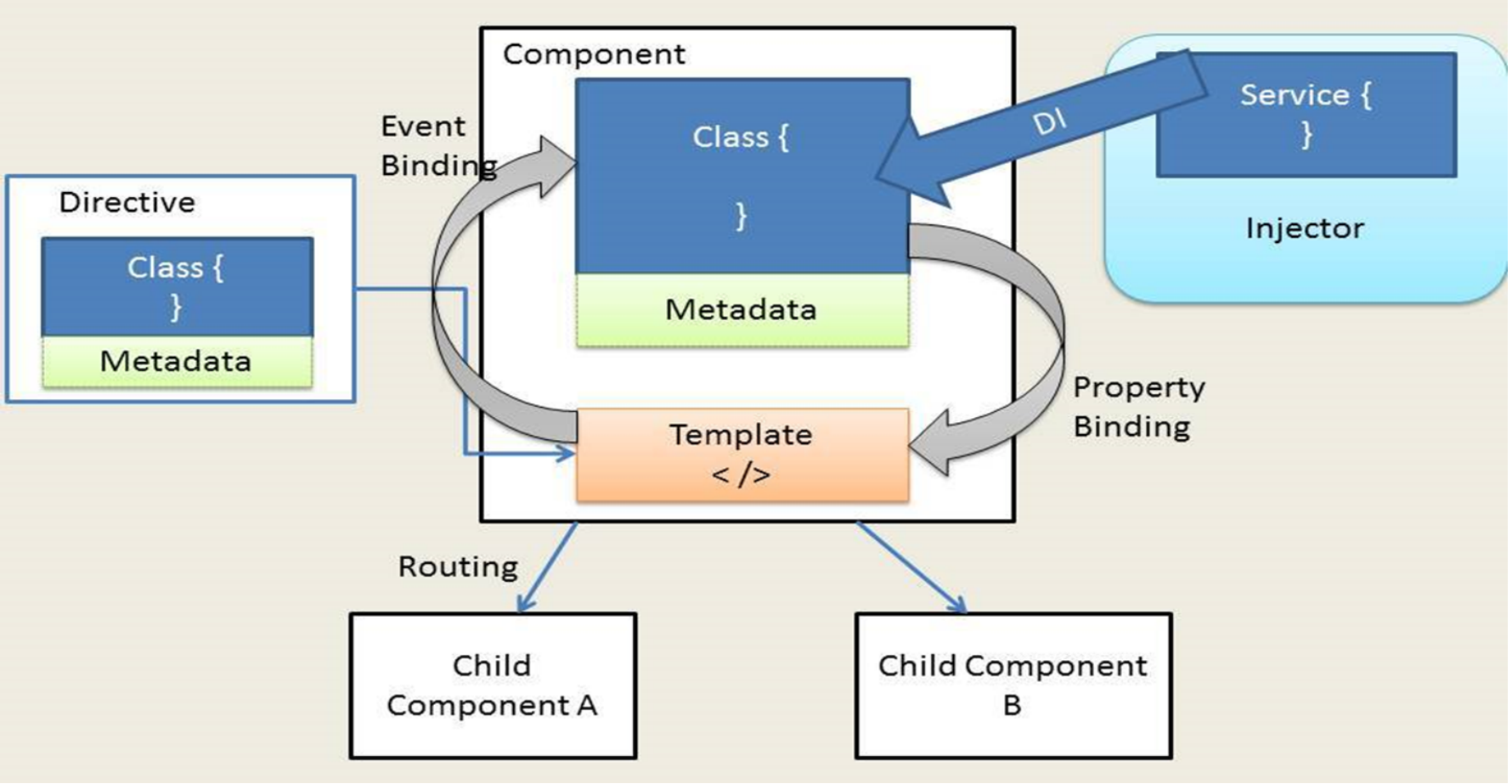
```
{
  "extends": [
    "tslint:recommended"
  ],
  "rulesDirectory": [],
   "rules": {
  "quotemark": [true, "single"]
   },
}
```

Configuration

- **Extends?:** string | string[]: The name of a built-in configuration preset or a path or array of paths to other configuration files which are extended by this configuration.
- **rulesDirectory?:** string | string[]: A path to a directory or an array of paths to directories of custom rules.
- **rules?:** { [name: string]: RuleSetting }: A map of rule names to their configuration settings.

Angular Architecture

- The Architecture of an Angular Application is based on the idea of Components.



Module

- An Angular module is a container for a group of related components, services, directives, and so on.
- All elements of a small application can be located in one module (the root module), whereas larger apps may have more than one module.
- All apps must have at least a root module that is bootstrapped during the app launch.

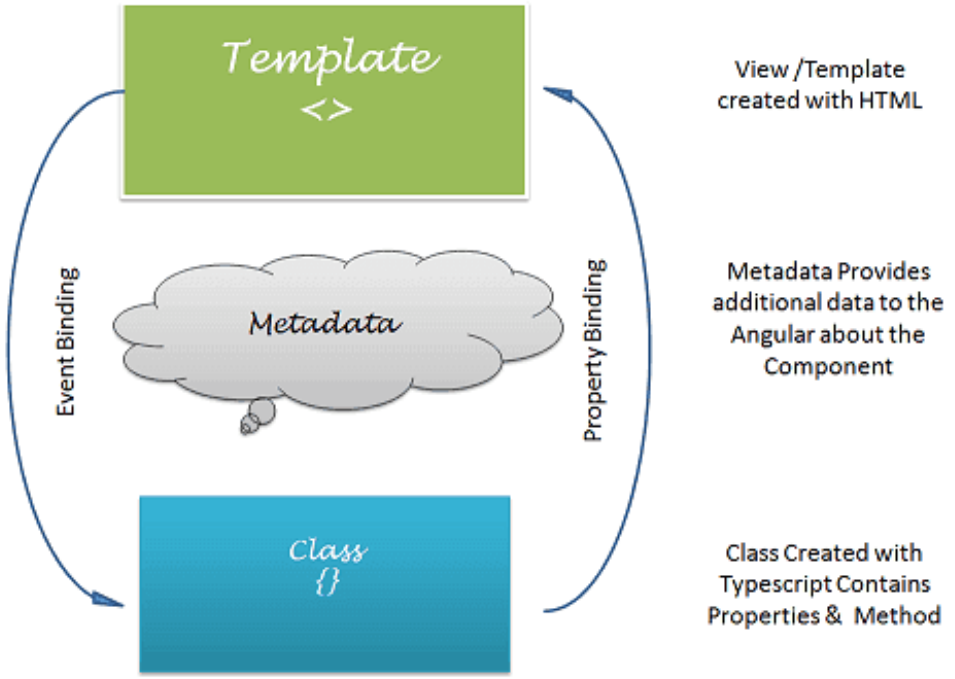
```
@NgModule({
    declarations: [
        AppComponent
    ],
    imports: [
        BrowserModule
    ],
    providers: [],
    bootstrap: [AppComponent]
})
```

Module Cont..

- The ©ngModule Metadata above has four fields.
- The **declarations**, **imports**, **providers**, & **bootstrap**
  - **Imports** Metadata tells the angular list of other modules used by this module.
  - **Declaration** Metadata lists the components, directives & pipes that are part of this module.
  - **Providers** are the services that the other components can use.
  - **Bootstrap** Metadata identifies the root component of the module. When Angular loads the appModule it looks for bootstrap Metadata and loads all the components listed here. We want our module to load AppComponent , hence we have listed it here.

Components

- The Component is the main building block of an Angular Application.
- The Components consists of three main building blocks **Template, Class and Metadata**



Components Cont..

- The Angular Components are plain JavaScript classes and defined using **©component Decorator**. This Decorator provides the component with the View to display & Metadata about the class
- An app must have at least one module and one component, which is called the **root component**.

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

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    title = 'app';
}
```

Components Cont..

- Each ©Component annotation must define selector and template (or templateUrl) properties, which determine how the component should be discovered and rendered on the page.
- Properties and methods of the component class are available to the template
- The selector property is similar to a CSS selector. Each HTML element that matches the selector is rendered as an Angular component.
- The template property defines user interface(UI)

Templates

- A template is nothing but a form of HTML tags that tells Angular about how to render the component.
- A template looks like regular HTML, except for a few differences.
- Each component must define a view, which is specified in either a template or a templateUrl property of the ©Component decorator
- Templates LLClude data bindings as well as other components and directives
  - In-lined template in the component
    - **template=`<h1>This is Angular2 Demo</h1>`**
  - In an external HTML file
    - **templateUrl='template.html'**

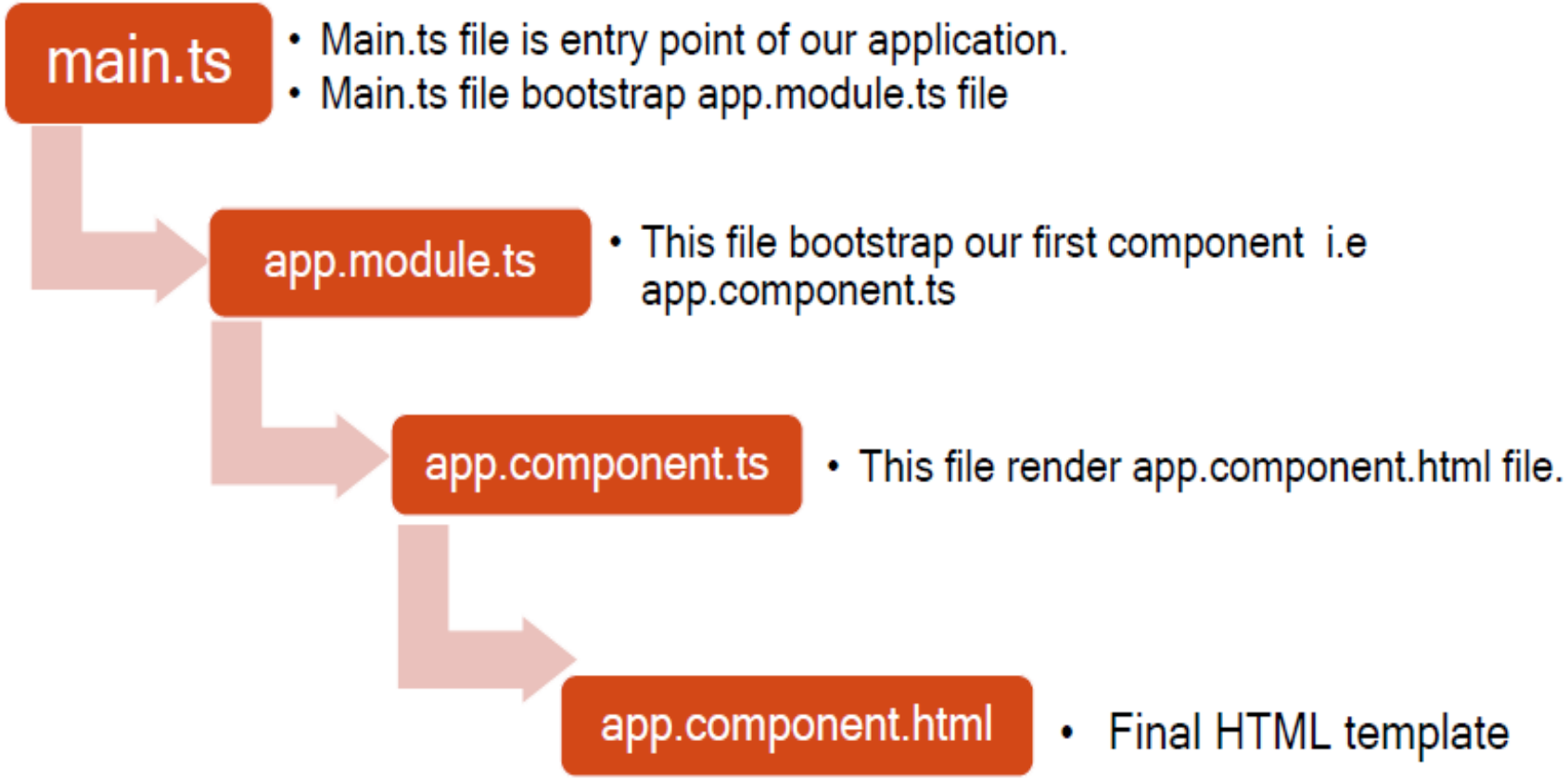
Metadata

- Metadata tells Angular how to process a class. To tell that AppComponent is a component, **metadata** is attached to the class.
- In TypeScript, you attach metadata by using a **decorator.**
- With metadata we describe parts of the Angular application and bind those parts together:
  - ©**NgModule** decorator
    - Configures module
  - ©**Component** decorator
    - Configures component
  - ©**Service** decorator
    - Configures service
  - ©**Pipe** decorator
    - Configures pipe
  - ©**Directive** decorator
    - Configures directive

Component metadata properties

- **Selector**
  - Selector specifies the simple CSS selector, where our view representing the component is placed by the Angular.
- **Providers**
  - The Providers are the services, that our component going to use. The Services provide service to the Components or to the other Services.
- **Directives**
  - The directives that this component going to use are listed here.
- **Styles/styleUrls**
  - The CSS Styles or style sheets, that this component needs. Here we can use either external stylesheet (using styleUrls) or inline styles (using Styles). The styles used here are specific to the component
- **template/templateUrl**
  - The HTML template that defines our View. It tells angular how to render the Component's view.

How Application Starts



Data Binding

- Data binding is communication between business logic (Component) and views(Template).
- It enables data to flow from the component to template and vice-versa.
- Binding can be used display component class property values to the user, change element styles, respond to a user event etc.
- The data-binding syntax lowers the amount of manual coding.
- The Data binding is not new to Angular. It existed in AngularJS., but the syntax has changed.
- Data binding LLCludes **interpolation, property binding, event binding,** and **two-way binding.**

Interpolation(Template Expression)

- Interpolation markup is used to provide data-binding to text and attribute values.
- The content inside the double braces is called **Template Expression** in Angular.
- Using {{ }} we display values from component into template.

app.component.ts

```
1
2 export class AppComponent
3 {
4     title : string ="app"
5 }
6
```

app.component.html

```
1
2 <h1>
3     Welcome to {{title}}!
4 </h1>
5
```

Interpolation cont..

- **Perform some mathematical operations**

```
<p>{{100*80}}</p>
```

- **Concatenate two string**

```
<p>{{ 'Hello & Welcome to '+ ' Angular Data binding '}}</p>
```

- **Invoke a method in the component**
  - We can invoke the components methods using interpolation.

```
getTitle(): string {
    return this.title;
}
```

```
<p>{{getTitle()}}</p>
```

Property Binding

- Property binding allow us to bind values to properties of an element to modify their behavior or appearance.
- This can LLClude properties such as class, disabled, href or textContent.
- Using [ ] we bind valued from component to element's property.

```
<p [innerText]="title"></p>
<p [innerText]="getTitle()"></p>
<p [innerText]="'Hello & Welcome to '+ ' Angular Data binding '"></p>
<p [innerText]="100*80"></p>
<p [style.color]="color">This is red</p>
```

Property Binding Vs Interpolation

- Everything that can be done from Property binding can be done using the interpolation.
- But the only difference is that Interpolation requires expression to return a string. If you want to set an element property to a non-string data value, you must use property binding.
- interpolation, the button will always be disabled irrespective of isDisabled class property value is true of false.

```
<button [disabled]='isDisabled'>Try Me</button>
```

```
<button disabled='{{isDisabled}}'>Try Me</button>
```

Event Binding

- When a user interacts with your app, it's sometimes necessary to know when this happens.
- A click, hover, or a keyboard action are all events that you can use to call component logic within Angular.
- The Name of the event is enclosed in ().
- Event Binding is one way from View to Component

```
<button (click)='buttonClicked()'>Click Me</button>
<p> Button Clicked {{count}} Times </p>
```

```
count: number=0;

buttonClicked() : void {
    this.count=this.count+1;
    console.log("Button Clicked")
}
```

Two Way Binding

- Two way binding means send values from component to template, and returns changed values from template to component
- Using [()] we can achieve two way data binding
- Two-way data binding combines the property and event binding into a single notation using the ngModel directive.
- The ngModel directive is not part of the Angular Core library. It is part of the FormsModule library. You need to import the FormsModule package into your Angular module

```
name: string='';
```

```
Enter your Name : <input type='text' [(ngModel)]='name' />
You have entered {{name}}
```

Directives

- The Angular directive helps us to manipulate the DOM. You can change the appearance, behavior or a layout of a DOM element using the Directives. They help you to extend HTML.
- Directives are the most fundamental unit of Angular applications.

- There are three kinds of directives in Angular:
  - **Structural Directives**
  - **Attribute Directives**
  - **Custom Directive**

Structural Directives

- Structural directives can change the DOM layout by adding and removing DOM elements. All structural Directives are preceded by Asterix symbol
- ***ngFor**
  - The ngFor is an Angular 2 structural directive, which repeats a portion of HTML template once per each item from an iterable list (Collection). The ngFor is similar to ngRepeat in AngularJS

```
<tr *ngFor="let customer of customers;">
    <td>{{customer.customerNo}}</td>
    <td>{{customer.name}}</td>
    <td>{{customer.address}}</td>
    <td>{{customer.city}}</td>
    <td>{{customer.state}}</td>
</tr>
```

Local Variables in ngFor

- ngFor also provides several values to help us manipulate the collection. We can assign the values of these exported values to the local variable and use it in our template.
  - **index**
    - This is a zero-based index and set to the current loop iteration for each template context.
  - **first**
    - This is a boolean value, set to true if the item is the first item in the iteration.
  - **last**
    - This is a boolean value, set to true if the item is the last item in the iteration.
  - **even**
    - This is a boolean value, set to true if the item is the even-numbered item in the iteration.
  - **odd**
    - This is a boolean value, set to true if the item is the odd-numbered item in the iteration.

How to Use Local Variables

- Using additional let operator, we can create an additional local variable and assign the value of the index to it. The syntax is shown below.

```
<li *ngFor="let item of items; let i=index;"> …. </li>
```

- The following code shows the list of movies along with the index.

```
<tr *ngFor="let movie of movies; let i=index;">
    <td> {{i}} </td>
    <td>{{movie.title}}</td>
    <td>{{movie.director}}</td>
    <td>{{movie.cast}}</td>
    <td>{{movie.releaseDate}}</td>
</tr>
```

Formatting odd & even rows

- We can use the odd & even values to format the odd & even rows alternatively. To do that create two local variable o & e. Assign the values of odd & even values to these variables using the let statement. Then use this to change the class name to either odd or even. The example code is shown below.

```
<tr *ngFor="let movie of movies; let i=index; let o= odd; let e=even;"
[ngClass]="{ odd: o, even: e }">
    <td> {{i}} </td>
    <td>{{movie.title}}</td>
    <td>{{movie.director}}</td>
    <td>{{movie.cast}}</td>
    <td>{{movie.releaseDate}}</td>
</tr>
```

- In Component's css file write following code.

```
.even { background-color: red; }
.odd { background-color: green; }
```

Structural Directives

- **ngSwitch**
  - The ngSwitch directive lets you add/remove HTML elements depending on a match expression. ngSwitch directive used along with NgSwitchCase and NgSwitchDefault

```
<div [ngSwitch]="Switch_Expression">
    <div *ngSwitchCase="MatchExpression1"> First Template</div>
    <div *ngSwitchCase="MatchExpression2">Second template</div>
    <div *ngSwitchCase="MatchExpression3">Third Template</div>
    <div *ngSwitchCase="MatchExpression4">Third Template</div>
    <div *ngSwitchDefault?>Default Template</div>
</div>
```

Structural Directives

- ***ngIf**
  - The ngIf Directives is used to add or remove HTML Elements based on an expression. The expression must return a boolean value. If the expression is false then the element is removed, else the element is inserted.

```
<div *ngIf="condition">
    This is shown if condition is true
</div>
```

Attribute Directives

- An Attribute or style directive can change the appearance or behaviour of an element.
- **ngModel**
  - The ngModel directive is used the achieve the **two-way data binding**.

```
name: string='';
```

```
Enter your Name : <input type='text' [(ngModel)]='name' />
You have entered {{name}}
```

Attribute Directives Cont..

- **ngClass**
  - The ngClass is used to add or remove the CSS classes from an HTML element. Using the ngClass one can create dynamic styles in HTML pages

```
<div [ngClass]="'red size20'">
    Red Text with Size 20px
</div>
```

```
.red {
    color: red;
}

.size20 {
    font-size: 20px;
}
```

Attribute Directives Cont..

- **ngStyle**
  - ngStyle is used to change the multiple style properties of our HTML elements. We can also bind these properties to values that can be updated by the user or our components.

```
<div [ngStyle]="{'color': 'blue', 'font-size': '24px', 'font-weight': 'bold'}">
    some text
</div>
```

Custom Directives

- The ©Directive decorator allows you to attach custom behavior to an HTML element (for example, you can add an autocomplete feature to an <input> element). Each component is basically a directive with an associated view, but unlike a component, a directive doesn't have its own view.
- The following example shows a directive that can be attached to an input element in order to log the input's value to the browser's console as soon as the value is changed.
- To bind events to event handlers, enclose the event name in parentheses. When the input event occurs on the host element, the onInput() event handler is invoked and the event object is passed to this method as an argument.

Custom Directives Example

- Here's an example of how you can create and attach the directive to an HTML element:

```
@Directive({
  selector: 'input[log-directive]',
  host: {
    '(input)': 'onInput($event)'
  }
})
class LogDirective {
  onInput(event) {
    console.log(event.target.value);
  }
}
```

- Here's an example of how you can attach the directive to an HTML element:

```
<input type="text" log-directive/>
```

Custom Directives Example

- This example shows a directive that changes the background of the attached element to blue:

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

@Directive({ selector: '[highlight]' })

export class HighlightDirective {
  constructor(renderer: Renderer, el: ElementRef) {
    renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'blue');
  }
}
```

- Here's how this directive can be attached to the <h1> HTML element:

```
<h1 highlight>Hello World</h1>
```