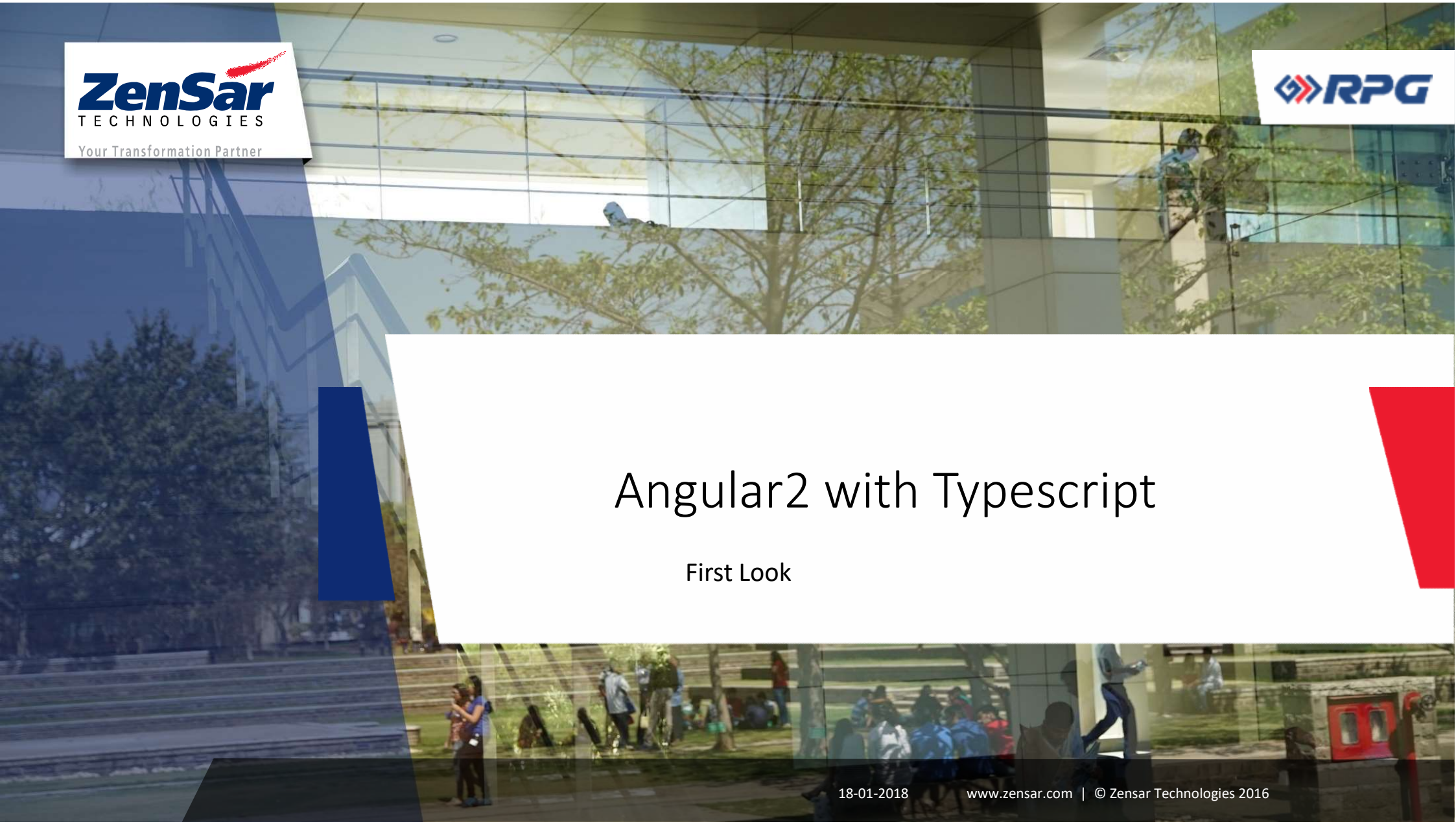


Angular2 with Typescript

First Look

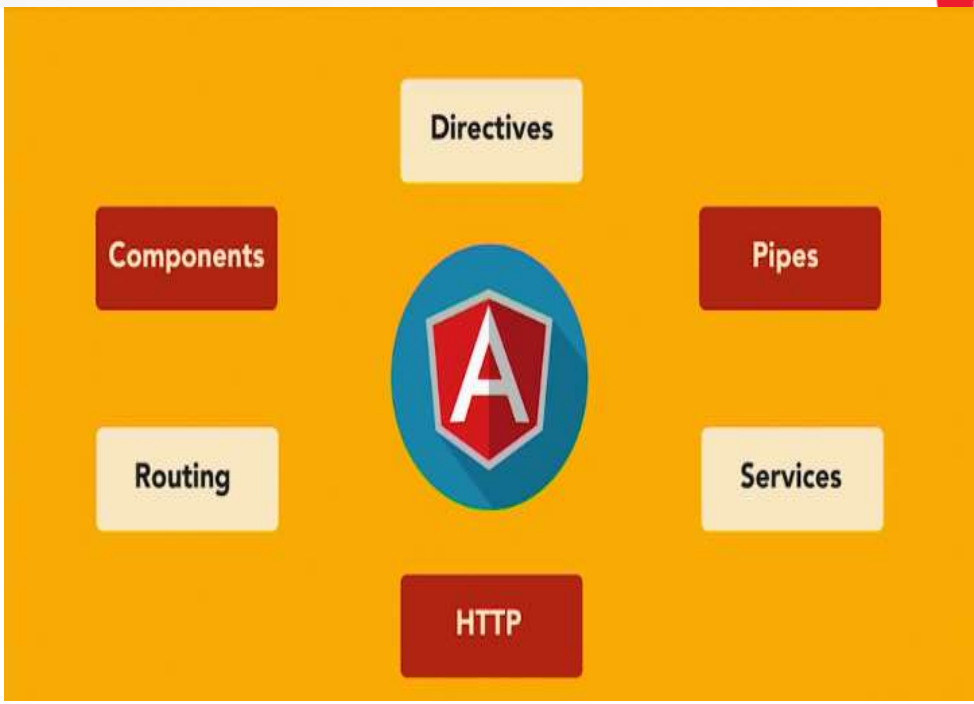
Introduction

- This course is a gentle introduction to the evolution of Angular 2.
- It provides an overarching view of Angular 2, and how the core concepts fit together to help us build applications.
- We'll learn how our Angular 1 skills translate and have prepared us to build Angular 2 applications.

Introduction

The major topics we'll cover are

- Components and their templates
- Template syntax
- Databinding
- Services using dependency injection
- Routing
- Data and HTTP.



By the end of this course, we'll have learned these concepts and more, and we'll be well on our way to building Angular 2 apps.

Prerequisites

- Basic knowledge of JavaScript
- HTML5
- Familiarity with Angular 1 and TypeScript will be helpful, they're certainly not required.

Angular

- As the web evolved, so did Angular, into Angular 2.
- It's a full and modern web framework that enables us to create powerful applications.

Why Angular 2 ?

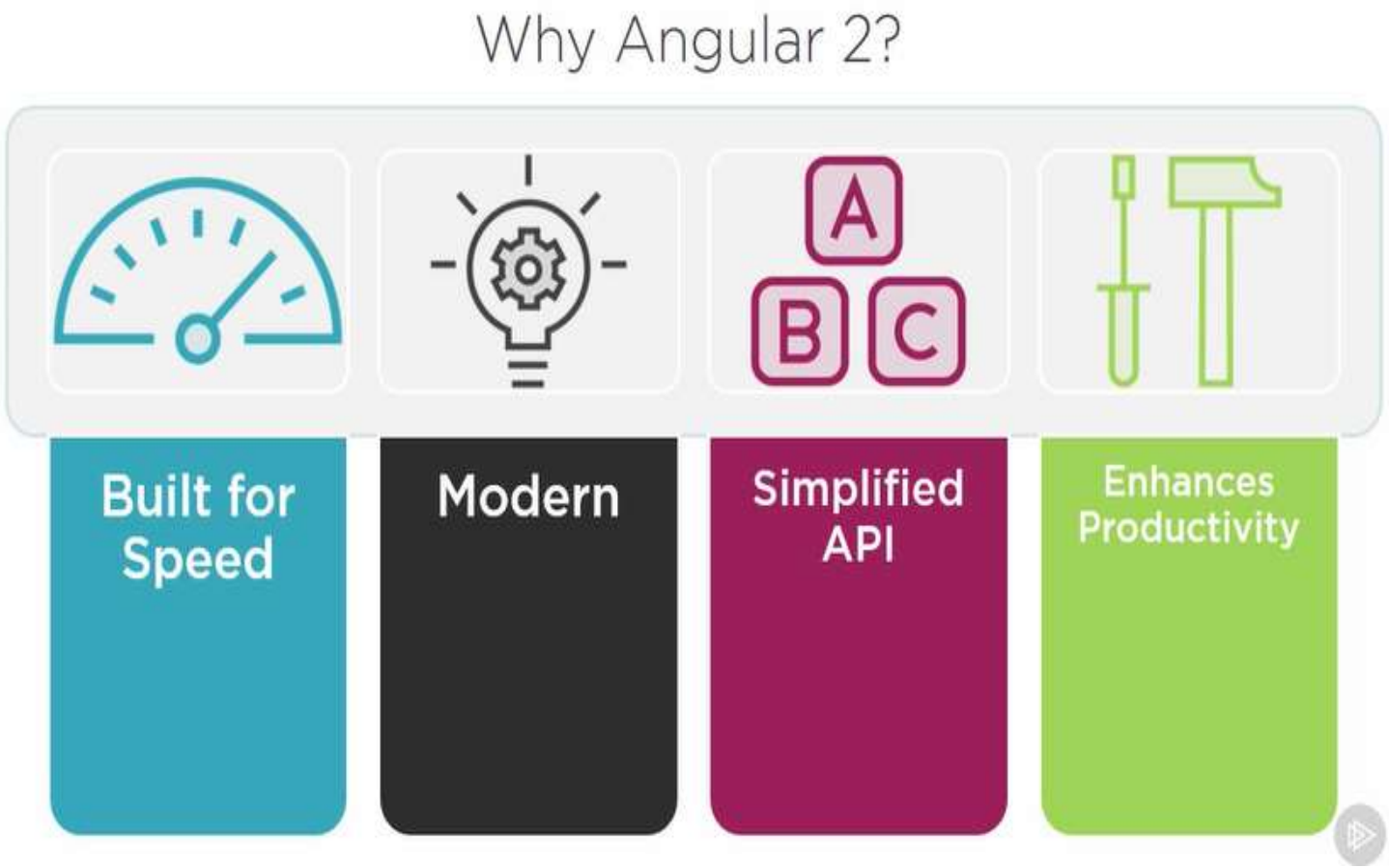
- The answer to that question lies in understanding Angular 2's value to companies and developers.
- Much faster than Angular 1, as it relies on modern web standards and practices.
- Angular 2 is a full framework that covers the common use cases we all encounter.
- There are less Angular constructs.
- For one example, Angular 2's templating syntax alone removes the need for dozens of features from Angular 1.

Why Angular2?

- Writing Angular 2 is easy.
- The concept count is less in Angular 2 than it was in Angular 1.
- The Angular 2 code is far more intuitive.

Fast Powerful Clean Easy

Why Angular2?



What is Angular2?

- It is a framework for developing client side applications
- There is dramatic upgrade from angular 1 to Angular 2.
- Angular 2 ,3 to 5 times faster than its previous version.

Angular2 Browser Support

iOS

Internet Explorer

9, 10, 11 and Edge

4.1+

Live Samples on the Web

Code along and run all samples live on the web.

No setup required.

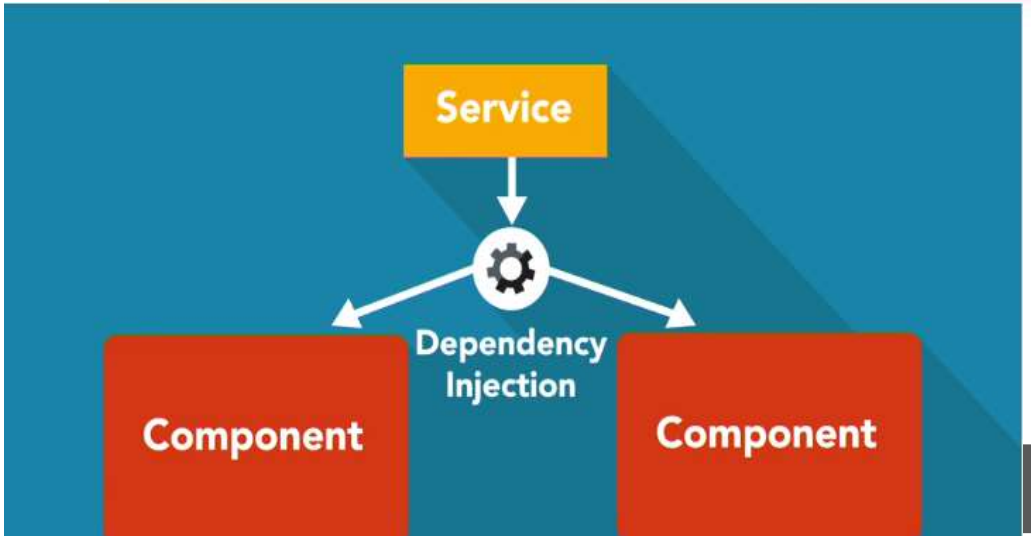
http://jpapa.me/a2firstlook

Beyond the First Look

Follow the QuickStart and tutorial on http://angular.io

Getting Started and Fundamentals courses on Pluralsight

Angular 2 Architecture, What's New and What's Different



In this module

- Language Choices
- Angular 1’s Impact
- Comparing concepts from Angular 1 to 2
- Resources

Angular 2 Architecture –Language Choices

There are four main choices.

a. ES5.

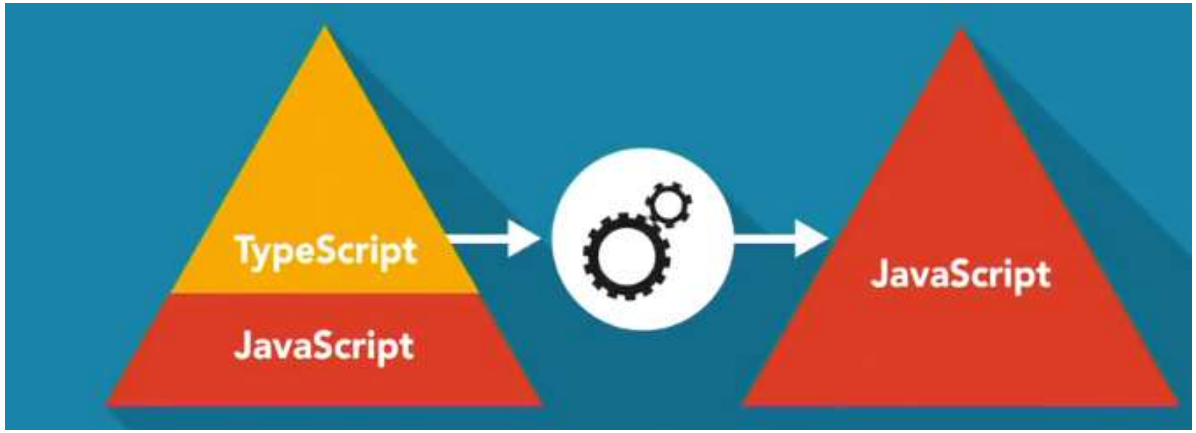
- It's been around for years, and all modern browsers support it out of the box.
- ES5 is just JavaScript.
- It doesn't require any compilation.
- ES5 is also the most common language that people use to write Angular 1.

b. ES6

- This another option we have.
- It offers modern language features such as classes, import and export of modules, ES6 destructuring.
- It has the simple but effective let statement.
- It's a much cleaner way to develop, but some help is required getting all browsers to process it.
- It is easy when we use a compiler like Babel.
- ES5 and ES6 both don't support types, but TypeScript does.

c. TypeScript

- It is an evolution of JavaScript, always adopting the standards and adding a few extra features.
- Angular team is actually writing Angular 2 in TypeScript
- Most notably, it adds interfaces and types, so TypeScript's big advantage is that it helps detect errors while we type.
- The modern editors use the TypeScript compiler to check the code and alert when a mistake is made.



c. TypeScript *contd*....

- Like Babel, with Typescript we also use a compiler to write ES5 so browsers can use it.
- ES6 is a superset of ES5, and TypeScript is a superset of ES6.
- TypeScript adopts emerging standards such as decorators.

d. Dart,

- It is another way we can go.
- Adoption for dart is the lowest among all the languages.

In this course we're going to be using TypeScript, but in everyday programming you can pick any one you want.

Angular 2 Architecture –Language Choices

Angular 2 Architecture –Typescript syntax overview

1. The import statement.
This is TypeScript syntax that will handle module loading.

2. The @Component block of code is a TypeScript decorator.

```
import {Component} from 'angular2/core';
import {FormBuilder} from 'angular2/common';

@Component({
    selector: 'media-tracker-app'
    templateUrl: 'app/app.component.html',
    styleUrls: ['app/app.component.css']
})
export class AppComponent {
    constructor(formBuilder: FormBuilder) {}
}
```

Within the class, there is a constructor function, which is an ES2015 syntax, but the parameter it takes in has a colon and then a type behind it.

3. The class syntax is actually ES2015, but the export keyword is TypeScript for turning the class into a module.

Angular 2 Architecture –Angular 1's Impact

- Let's take a look at the impact that Angular 1 has had on the modern development landscape.
- Angular 1's really widely used, and it's got a massive ecosystem.
- Ecosystem means, there are a lot of community efforts behind Angular.
- There are some great products like Firebase, and internationalization for Angular and Angular Material, and those are just to name a few.

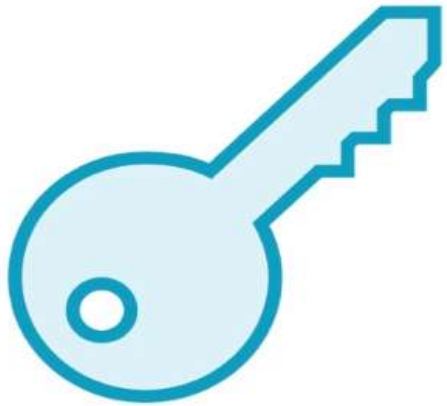
Why such a huge ecosystem?

- It's because there are 1.1 million developers have used Angular according to Brad Green, the director of the Angular team.

Angular 2 Architecture –Angular 1's Impact

- How we can lean on our Angular 1 skills to get us to Angular 2.
- Our Angular 1 skills matters and there's less to learn in Angular 2
- The **seven things** that we'll all be familiar with in our transition from Angular 1 to Angular 2 in the following slides

Angular 1 to Angular 2



Angular 2 Architecture –Controllers to Components

1

Controllers to Components

Angular 1

```
<body ng-controller="StoryController as vm">
  <h3>{{vm.story.name}}</h3>
  <h3 ng-bind="vm.story.name"></h3>
</body>

(function () {
 angular
   .module('app')
   .controller('StoryController', StoryController);

 function StoryController() {
   var vm = this;
   vm.story = { id: 100, name: 'The Force Awakens' };
 }
})();
```

Angular 2

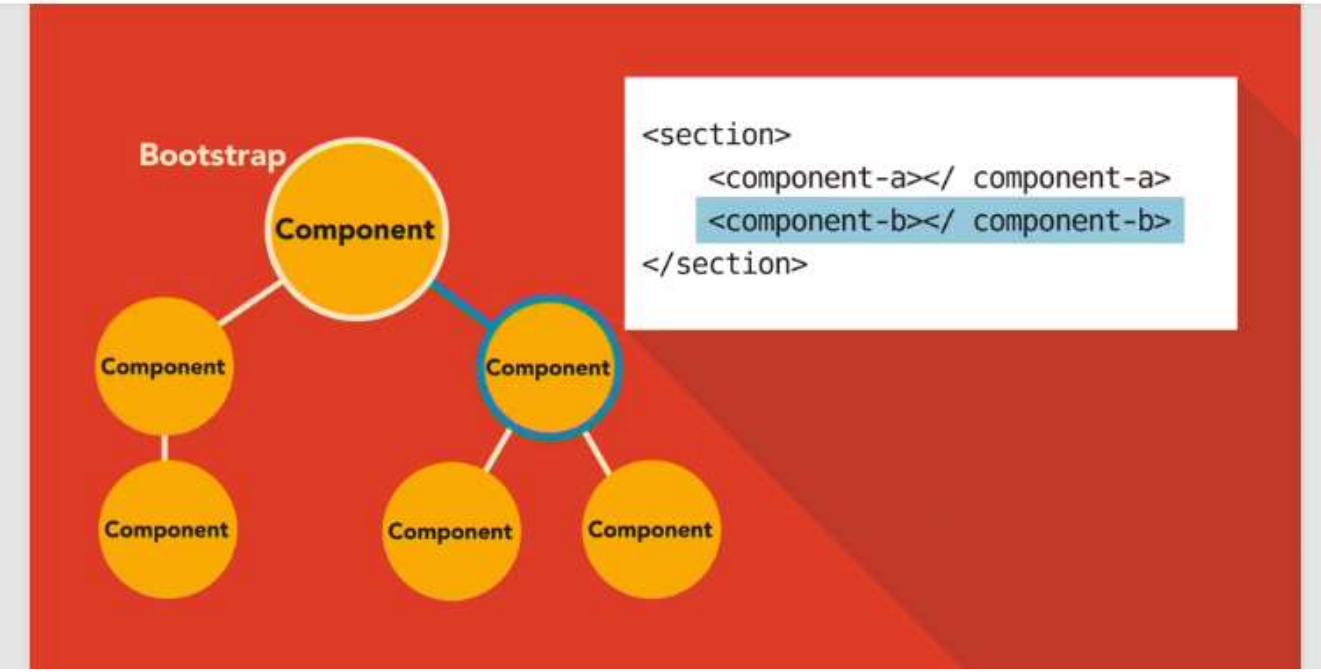
```
<my-story></my-story>


import { Component } from 'angular2/core';

@Component({
  selector: 'my-story',
  template: '<h3>{{story.name}}</h3>'
})
export class StoryComponent {
  story = { id: 100, name: 'The Force Awakens' };
}
```

3. If we want to use this component, we simply put **my-story** in our HTML page like this.

2.A **decorator**, @Component, describes the template that's showing where the HTML goes. And the name of our selector, which becomes an HTML element, is my-story.

1. Defining our **class** called StoryComponent, equivalent to the function StoryController on the left.

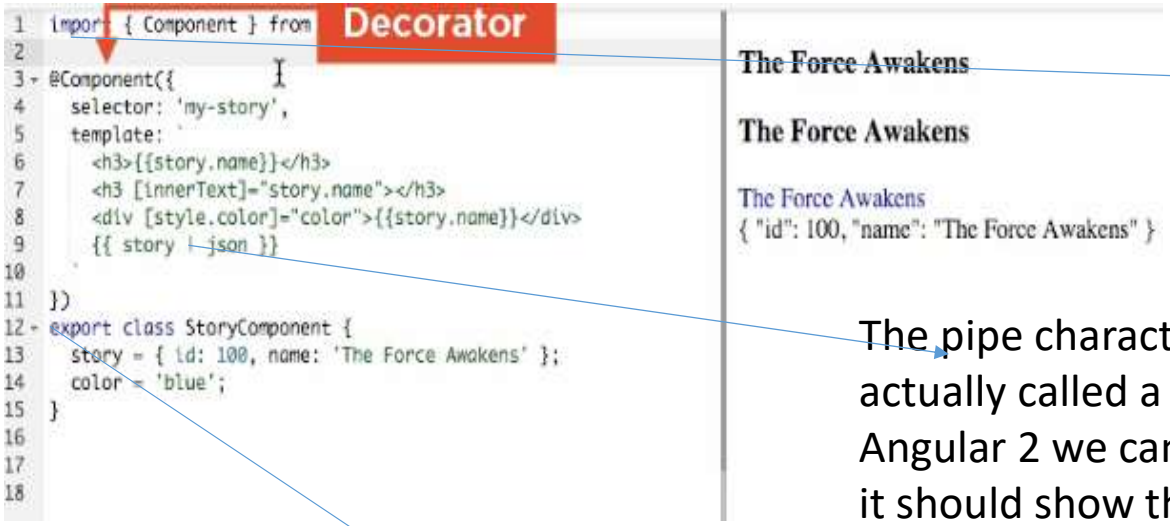
Angular 2 Architecture-First look at components…



A component in Angular is used to render a portion of html and provide functionality to that portion.
It does this through a component class in which you can define application logic for the component.

Angular is built upon components.
The starting point of an Angular app is the bootstrapping of the initial parent component like the html DOM tree that starts with an html element and then branches down from there, Angular runs on a component tree model.
After Angular loads the first component with the bootstrap call, it then looks within that component's html view and sees if it has any nested components.
If so, Angular finds matches and runs the appropriate component code on those.
This repeats for each component down the tree.

Angular 2Architecture- First look at components



import statement imports component from Angular's core and allows to use the component.

The pipe character, which is actually called a **pipe** now in Angular 2 we can put in json, and it should show the contents of that json.

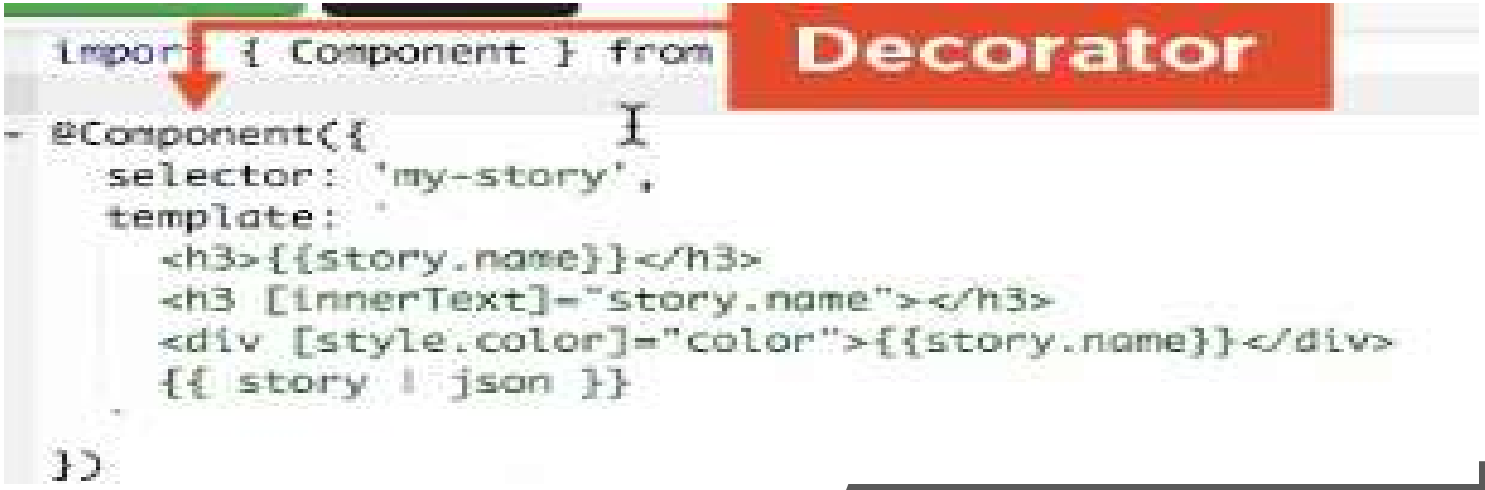
Defining a class called StoryComponent, and we're exporting it so somebody else can use it
So this export here exports the StoryComponent.
That means somebody else is probably importing it.

- StoryComponent just got a story with a name and an id property and the color blue.
- And up top we have an h3 in our template defining the name of the story, and that's why it's printing over here on the right in the h3.
- We've also data bound it to some other properties.

A component ENCAPSULATES the template, data and behavior of the view. Every angular 2 app has at least one component.

Angular 2 Architecture-First look at components…

- @Component on line three (prev slide) is actually a decorator.
- A decorator provides metadata for its class, in this case the StoryComponent.
- This decorator of @Component provides metadata, which is just data about the component saying the name of the selector is my-story, and the template, that's HTML, is right here.



Angular 2 Architecture-First look at components…

```
@Component({
  selector: 'my-story',
  templateUrl: 'app/somewhere.html'
})
```

- One can also specify a templateUrl, and could get rid of all that content.
- The templateUrl points to a file somewhere like app/somewhere.html.

Angular 2 Architecture-Bootstrapping angular

2

Bootstrapping Angular

Angular 1

```
<html ng-app="app">
```

Angular 2

```
import { bootstrap } from 'angular2/platform/browser';
import { AppComponent } from './app.component';

bootstrap(AppComponent);
```

In Angular 1, it was bootstrapped by adding ng-app directive on the html tag or the body tag or wherever you wanted to start up your app

In Angular 2, bootstrap is through code, *so goodbye to ng-app,* and say hello to bootstrap, which we pull out of this module, **angular2/platform/browser**, and then we call the bootstrap function, and we give it the starting component.
That component is basically the parent component for the entire application.

Angular 2 Architecture- Structural Directives

3

Structural Built-In Directives

In Angular 1 we use the keyword **in**

In Angular 2 we use the keyword **of, so we say #vehicle of vehicles.**

Angular 1

```
<ul>
  <li ng-repeat="vehicle in vm.vehicles">
    {{vehicle.name}}
  </li>
</ul>
<div ng-if="vm.vehicles.length">
  <h3>You have {{vm.vehicles.length}} vehicles</h3>
</div>
```

Angular 2

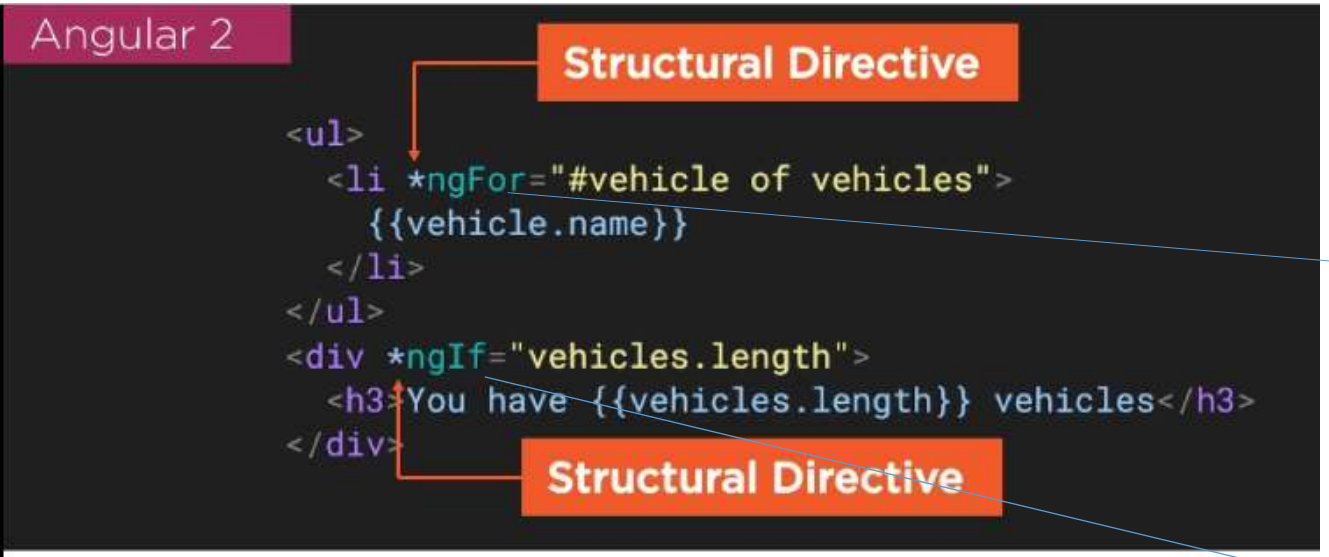
```
<ul>
  <li *ngFor="#vehicle of vehicles">
    {{vehicle.name}}
  </li>
</ul>
<div *ngIf="vehicles.length">
  <h3>You have {{vehicles.length}} vehicles</h3>
</div>
```

- In Angular 2 we have **ngFo**r and **ngIf.**
- These are structural built-in directives.
- Slightly different syntax here.
- See the usage of **camelCase** in Angular2 and the **little star right before them.**
- **The star represents a structural directive.**

- For our third key concept from Angular 1 to Angular 2, take a look at structural built-in directives.
- Angular 1 came with a whole bunch of directives like ng-app. The other two of the most common ones, ng-repeat and ng-if. We use these in Angular 1 to loop through a list of something. In the above case vehicles to display them, and then to display something conditionally.

A local variable up in the **ngFor, the #vehicle**, that allows us to refer to vehicle inside the li tag.
And also take and then in Angular 2 we use the keyword of, so we say #vehicle of vehicles.

Angular 2 Architecture- Structural Directives



This directive changes something in the view.

The ngFor changes the structure because it's adding more elements the li's.

The ngIf changes structure because that h3 is not going to be displayed if the vehicles has a length of 0

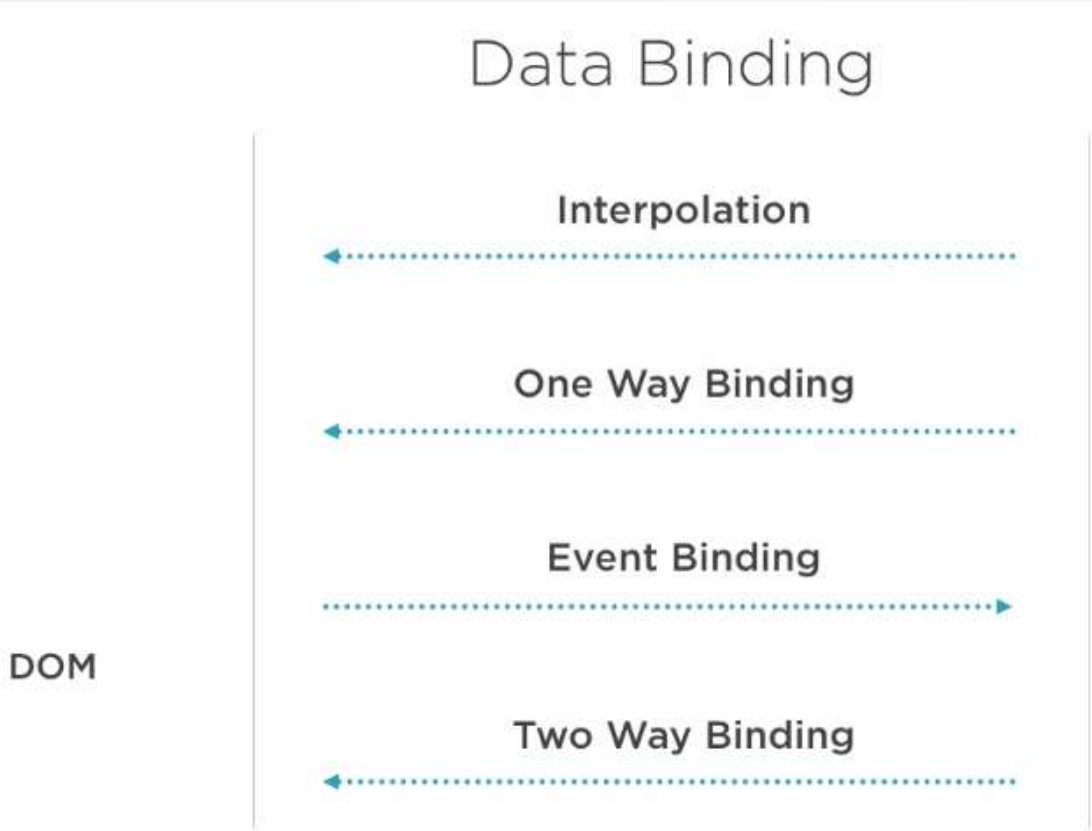
Structural Directives

Indicated by the * prefix

Changes the structure

Angular 2 Architecture- Data Binding

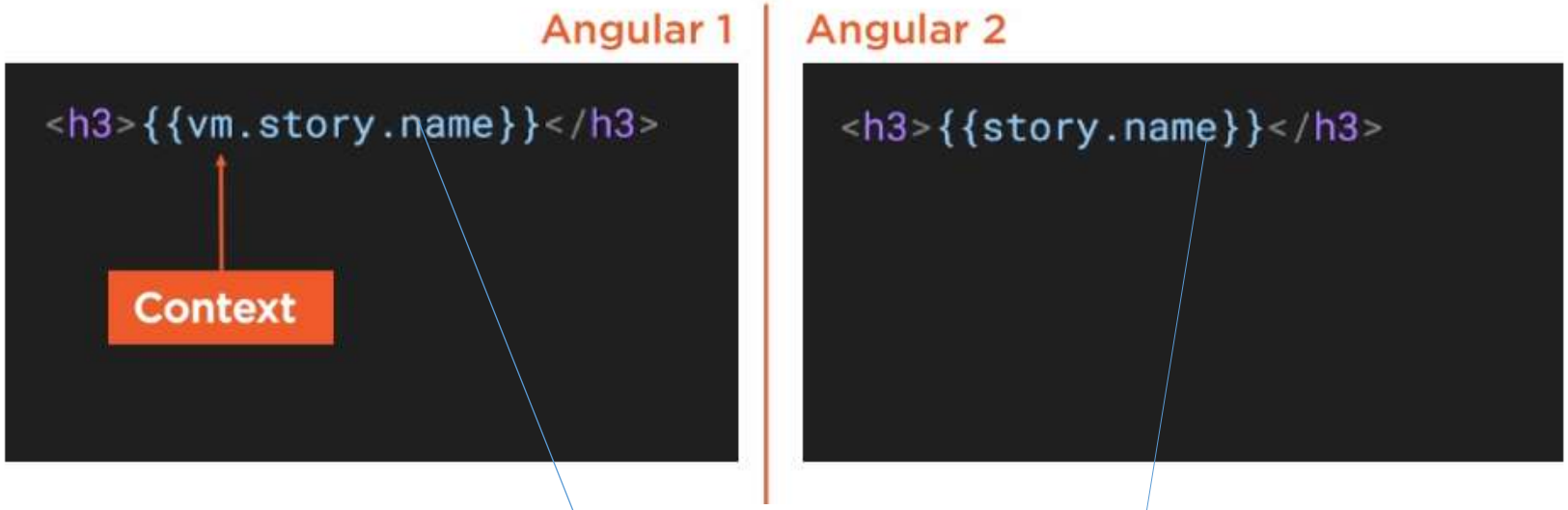
4



Data binding -one of the favorite features from Angular 1 made its way into Angular 2 in a sense.

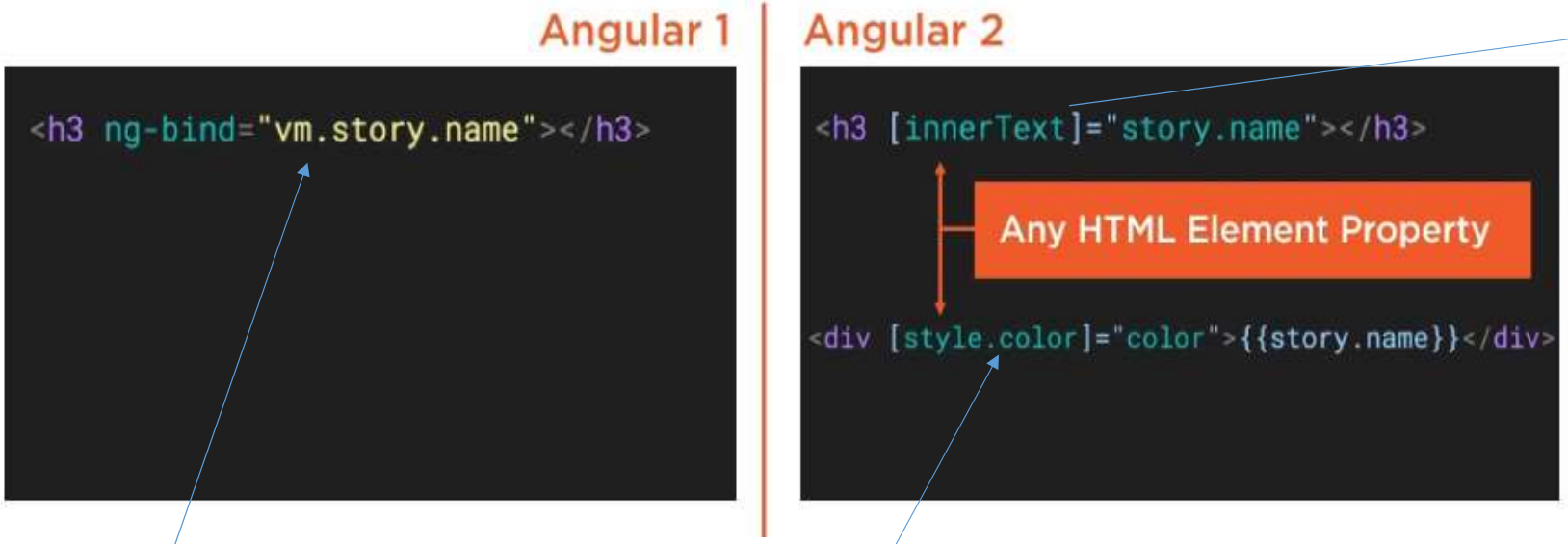
- It behaves a little bit differently in Angular2
- With data binding we achieve……
- We've got some data, in this case in a component in Angular 2, and then we want to get that data over to the DOM, which is our view
- In Angular 1 and Angular 2 we have the following options still.
- We have **interpolation, that's the curly braces**, that gets the data in read only form up to the DOM.
- Then we've got **one way binding**.
- We also have **event binding, things like clicks**.
- And then we have **two way binding for things like the input element.**

Angular 2 Architecture- Data Binding with *INTERPOLATION*



In an interpolation with Angular 1, this vm.story.name, and the value is to be shown here. The vm is the context from our controller.
In Angular 2 It's pretty much the same.
The curly braces are used, but instead of having to say vm, we've already got the context, so we just saved a little bit. Okay, that's interpolation.

Angular 2 Architecture- Data Binding with *1 Way Binding*



In Angular 1, if we wanted to, we could do ng-bind.
That's effectively what interpolation ended up being anyway.

So I might have a color model property, and I want to set that to be the style.color.

Then wrap style.color with the square braces and set it to my property, which is on my component.
And remember, this can be done with any HTML element property.

- In Angular 2, we could use the square brackets around a property and the HTML is valid.
- In Angular2 take any HTML element property, like
- innerText, wrap it with the square braces, and then we can bind that to a model, story.name.
- Although with the h3 here, using probably just do interpolation.

- For 1 way binding we can do instead is might have a div where I want to set the color equal to maybe red or blue or yellow.

Angular 2 Architecture- Data Binding with *Event Binding*

Angular 1

```
<button
  ng-click="vm.log('click')"
  ng-blur="vm.log('blur')">OK</button>
```

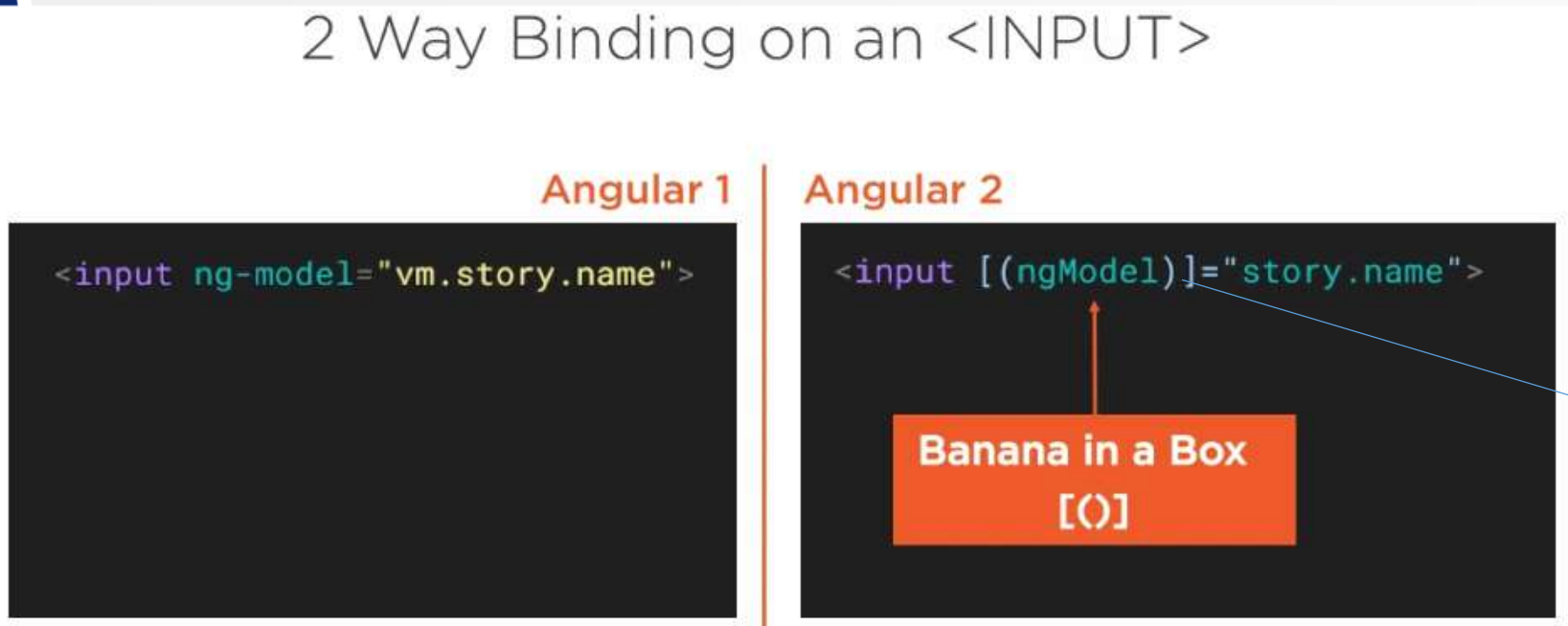
Angular 2

```
<button
  (click)="log('click')"
  (blur)="log('blur')">OK</button>
```

In Angular 1 we did ng-click or ng-blur

In Angular 2 the same property is taken that's on the HTML element.
In this case it's the event called click or the event called blur, and we wrap it with parentheses.
There are no click or blur directives in Angular 2.
These are in fact HTML element events

Angular 2 Architecture- Data Binding with *2 Way Binding*

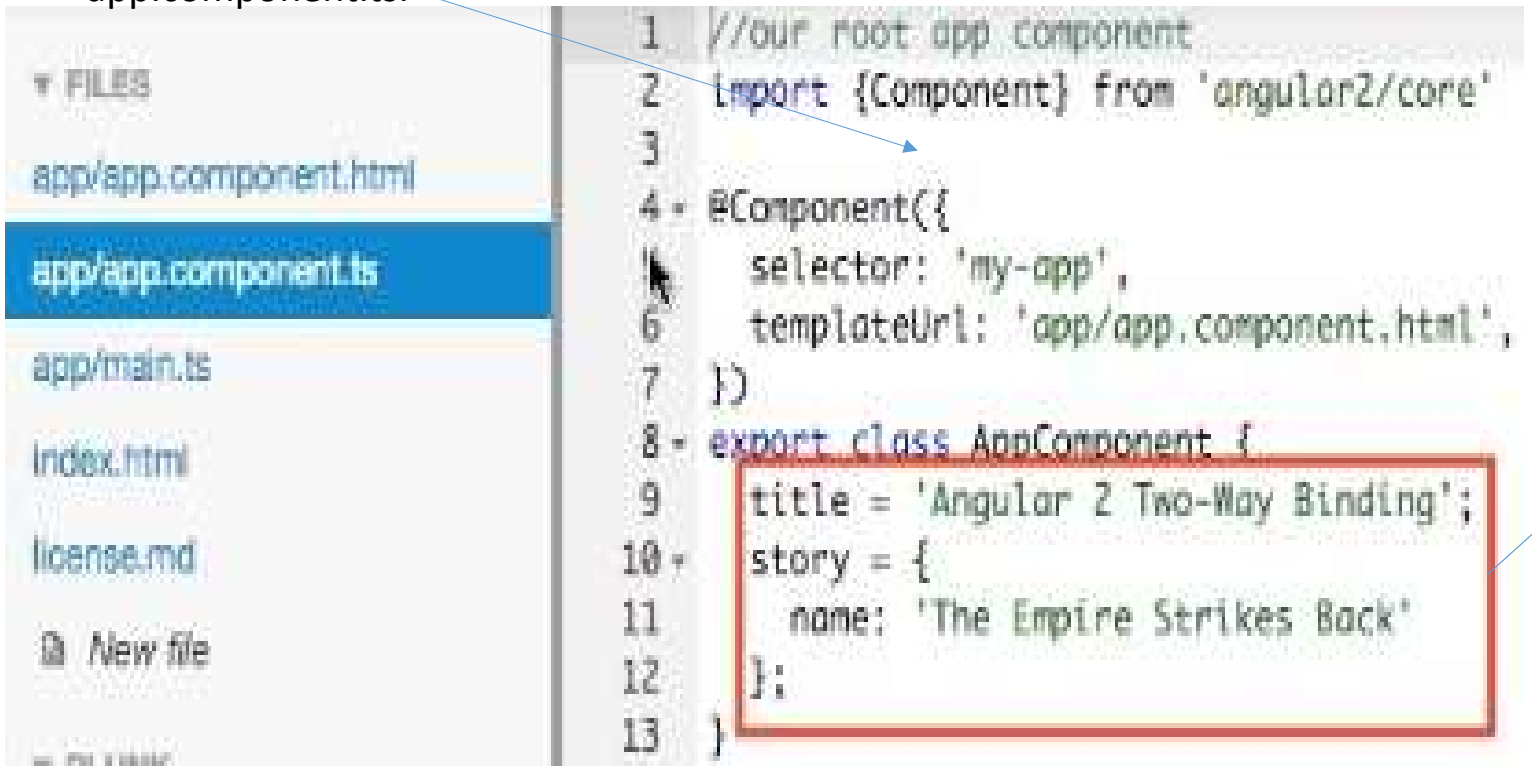


- In Angular 1 we say input **ng-model**, and set it to the model, and then get the two way binding.
- We can type a name on one side in the view, and then it goes to the controller and then back and forth.

- In Angular 2 we have a special directive called ngModel, which we bind to story.name.
- Notice the syntax, which is wrapping ngModel with the square brackets because it's property, but also with these parentheses.
- That's a special syntax that's known **as Banana in a Box.**

Angular 2 Architecture- Data Binding with *2 Way Binding First Look*

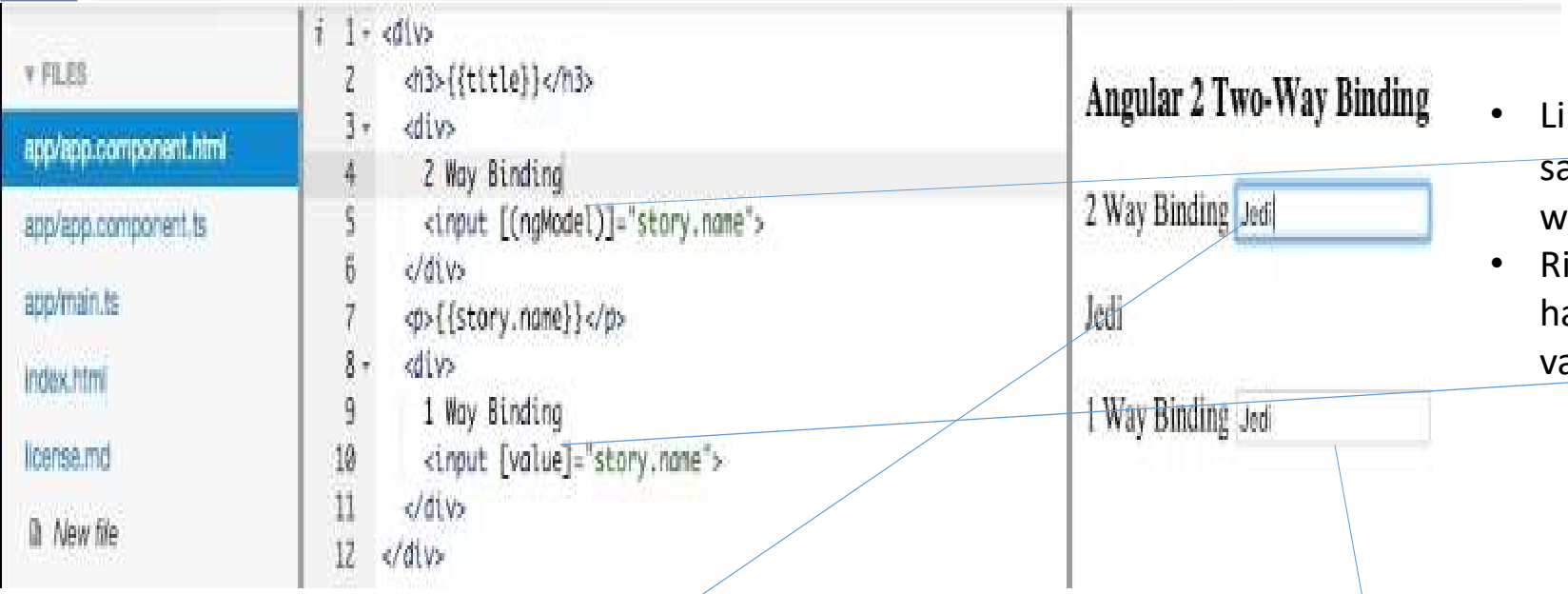
app.component.ts.



Here we've got a **title** and a **story**, and the **name of that is Empire Strikes Back.**

The html, for the above ts file is in the next slide.

Angular 2 Architecture- Data Binding with *2 Way Binding First Look*



- Line 5 we've got an input that says ngModel that's the two way binding for story.
- Right down here on line 10 we have an input, which is binding value to the story name.

- If the value in 1st input type is changed to Jedi, notice it's being reflected everywhere else.
- That's because of two way data binding.
- If the value in input type 2 is changed to something like Clone, notice nothing else is changing.
- That's because we're seeing the one way data binding in effect.
- One way, because the value property is only getting the values, the unidirectional dataflow.
- In this case, it's going in one direction from the component to our review or template.
- So this input effectively only gets values.

Angular 2 Architecture- *Fewer Built-In Directives*

5 Removes the Need for Many Directives

Angular 1

```
<div ng-style=
  "vm.story ?
    {visibility: 'visible'}
    : {visibility: 'hidden'}">

  <img ng-src="{{vm.imagePath}}">
  <br/>
  <a ng-href="{{vm.link}}">
    {{vm.story}}
  </a>

</div>
```

Angular 2

```
<div [style.visibility]=
  "story ? 'visible' : 'hidden'">

  <img [src]="imagePath">
  <br/>
  <a [href]="link">{{story}}</a>

</div>
```

- In Angular 2 we don't need any of those directives.
- We can simply just bind to style.visibility or to src in image or to the href inside of an anchor.

In fact, there are over 40 Angular 1 built-in directives that go away in Angular 2.

Angular 2 removes the need for many of the directives used in Angular 1.
we have an ng-style, ng-src, ng-href in Angular 1.

Angular 2 Architecture- *Fewer Built-In Directives*

No Longer Need These Directives Either

Angular 1	Angular 2
ng-click="saveVehicle(vehicle)"	(click)="saveVehicle(vehicle)"
ng-focus="log('focus')"	(focus)="log('focus')"
ng-blur="log('blur')"	(blur)="log('blur')"
ng-keyup="checkValue()"	(keyup)="checkValue()"

Angular 2 Architecture- *Fewer Built-In Directives*

Story input is using two way data binding for story

```
1  <div>
2    <h3>{{title}}</h3>
3    <div>
4      <p>When there is a story,
5        the image and link are shown</p>
6      <div>
7        Story:<input [(ngModel)]="story">
8      </div>
9      <div [style.visibility]="story ? 'visible' : 'hidden'">
10       <img [src]="imagePath">
11       <br/>
12       <a [href]="link">{{story}}</a>
13     </div>
14   </div>
15 </div>
16
```

We also notice the imagePath is being bound.
The link, if we hover over it, it's going to go to angular.io.

Angular 2 Architecture- *HTML Element Property and Event Binding*

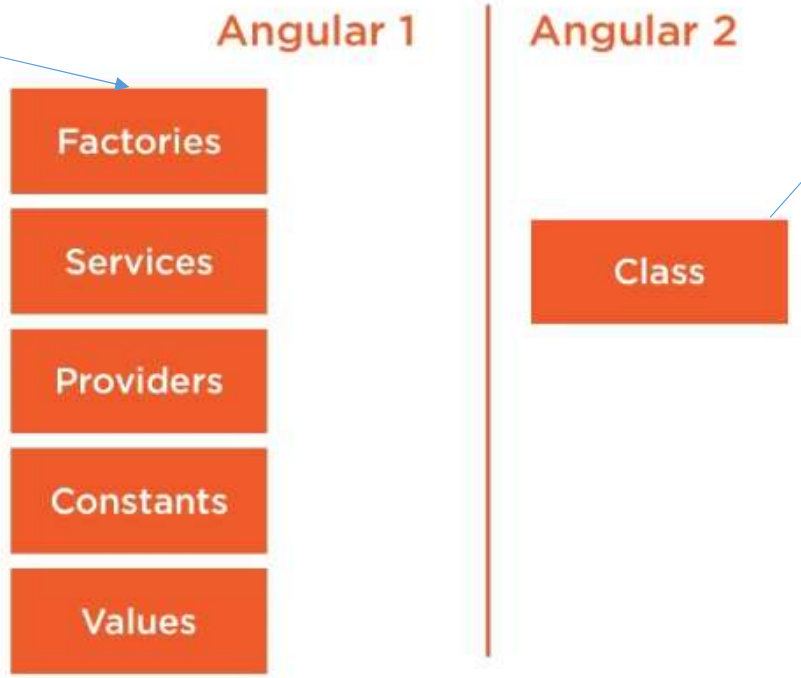
```
1  <div>
2    <h3>{{title}}</h3>
3    <img [src]="imagePath"
4      (mouseover)="log('mouseover')"
5      (mousedown)="log('mousedown')"
6      (mouseleave)="log('mouseleave')"
7      (mouseup)="log('mouseup')"
8      (click)="log('click')"
9    >
10   <div>
11     <input
12       (blur)="log('blur')"
13       (focus)="log('focus')"
14       (keydown)="log('keydown', $event)"
15       (keyup)="log('keyup', $event)"
16       (keypress)="log('keypress', $event)"
17       >
18   </div>
19   <ul class="messages">
20     <li *ngFor="#msg of messages">{{msg}}</li>
21   </ul>
22 </div>
23
```

Angular 2 Architecture- *Services*

6

Services

Angular 1, get some shared data out of somewhere, you create a service, or factory or service or a constant or a value, which are all just a provider.



In Angular 2 we still have the same need, but all of these end up consolidating inside just a **class**, and hence simplified.

Code on Services in angular 2 in the next slide.

Angular 2 Architecture- *Services*

Creating Services

Angular 1

```
angular
  .module('app')
  .service('VehicleService', VehicleService);

function VehicleService() {
  this.getVehicles = function () {
    return [
      { id: 1, name: 'X-Wing Fighter' },
      { id: 2, name: 'Tie Fighter' },
      { id: 3, name: 'Y-Wing Fighter' }
    ];
  }
}
```

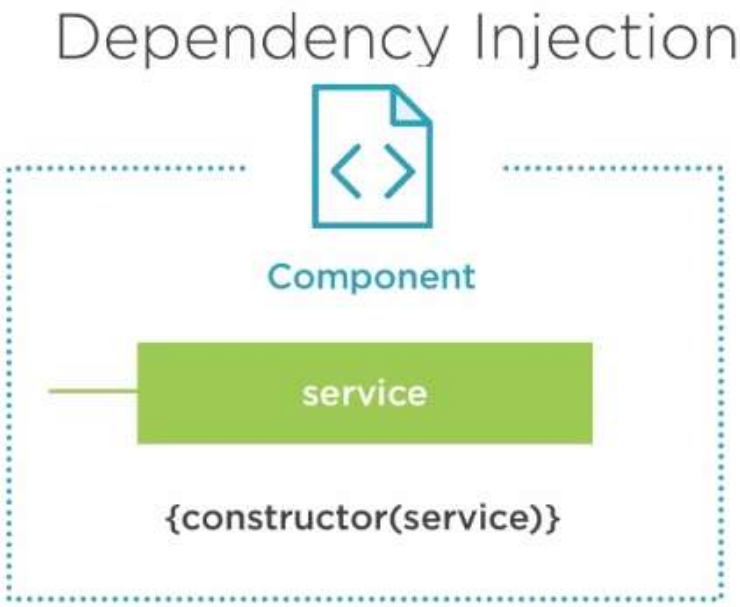
Angular 2

```
import {Injectable} from 'angular2/core';

@Injectable()
export class VehicleService {
  getVehicles = () => [
    { id: 1, name: 'X-Wing Fighter' },
    { id: 2, name: 'Tie Fighter' },
    { id: 3, name: 'Y-Wing Fighter' }
  ];
}
```

- In Angular 1 lets consider a service named maybe VehicleService.
- Notice that we register it as VehicleService, and then we define a method called getVehicles, which is just returning some hard-coaded data.

- In Angular 2 we define a class called VehicleService, and it has the same method, getVehicles.
- The syntax is very similar. The only real difference is we're looking at ES5 on the left and TypeScript on the right.

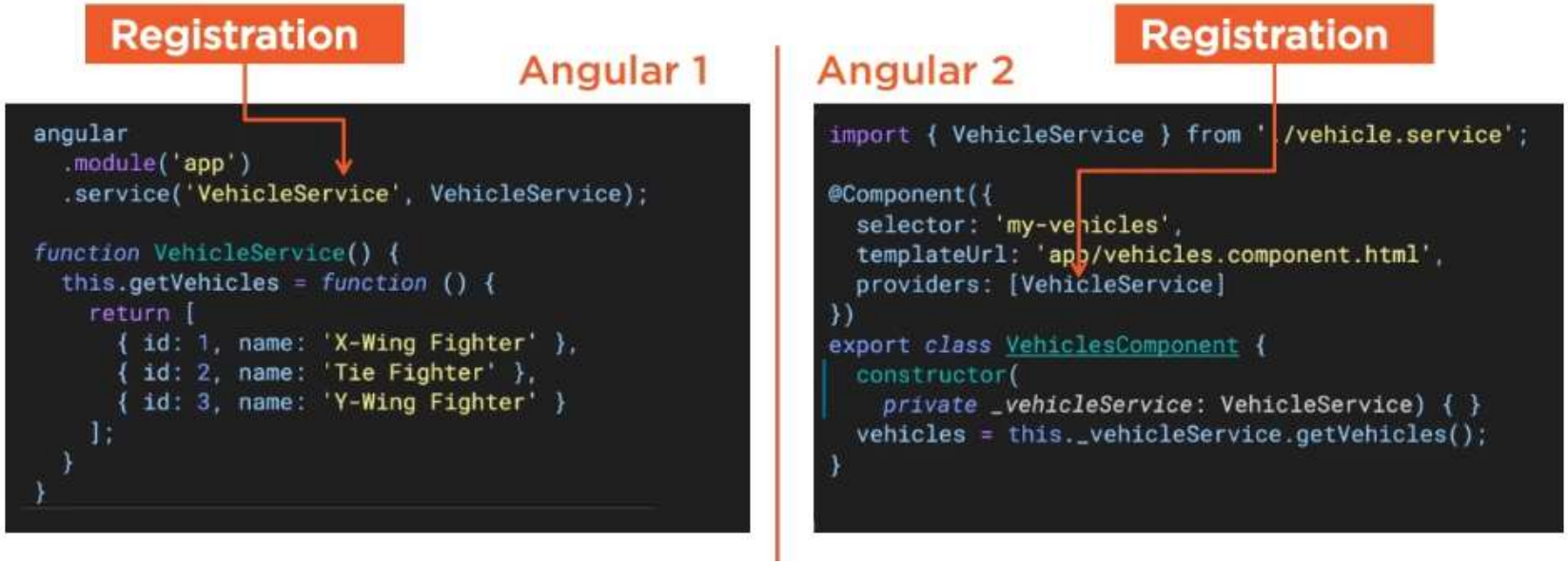
Angular 2 Architecture- *Dependency Injection*



- we may have a component, and that component needs a service.
- Maybe it's a vehicle component, and it needs to get that vehicle serviced to get its data.
- For this we require a constructor where we inject that service.
- Well, it's going to get that from the Angular 2 injector, and it's going to pull it in.
- For all that to work, we have to know how to register our services with the injector.

Angular 2 Architecture- *Dependency Injection*

Registering Services with the Injector



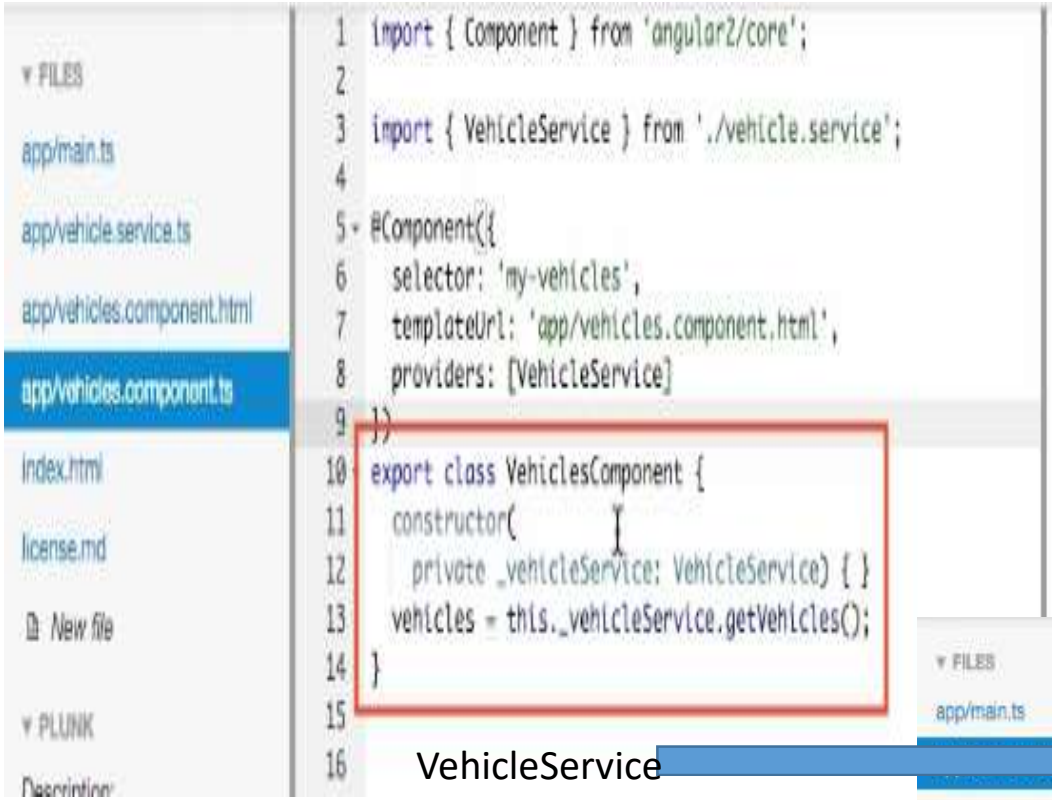
We have created a service in Angular 1, the VehicleService, and right here is where we're registering it with Angular 1.

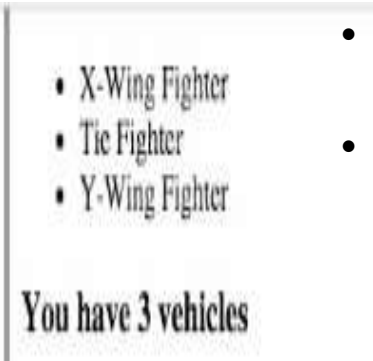
- In Angular 2 there's no magical string.
- A provider is created called
- VehicleService. (in our eg).
- Provide that to this component or any component who's a child of this component.
- So we just do this once per app for anything we want to share throughout the entire app.

Angular 2 Architecture- *Dependency Injection*

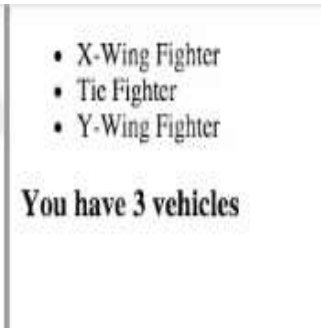
- In Angular 2 is a very similar process except we're not using strings.
- Here we say go get this thing called VehicleService, that's the right- hand arrow, that's the type, and then put in this variable in the constructor.

Angular 2 Architecture- *Dependency Injection*





VehicleService

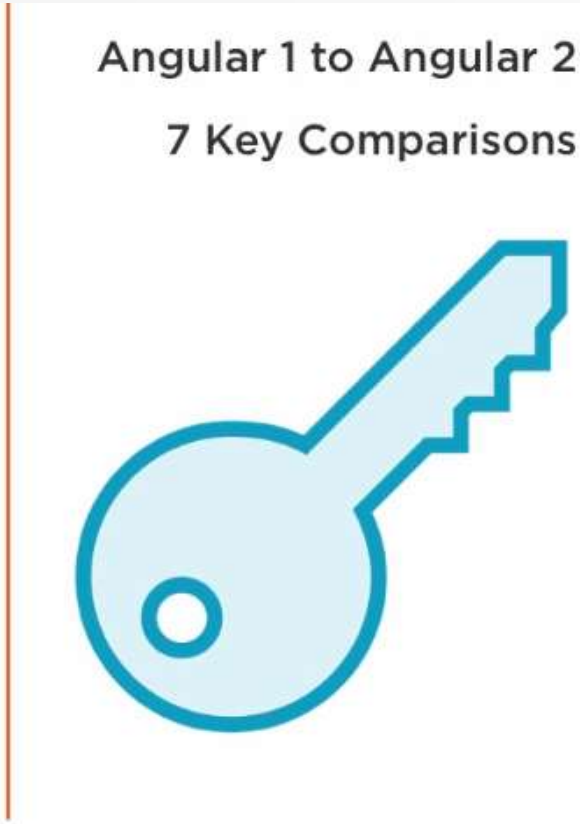


- In the code we have both component & service.
- The VehiclesComponent, is saying in the constructor to accept in this VehicleService. That's the type of it, and it's going to
- put it in this variable _vehicleService.
- The service has to be registered, and the providers statement.

Angular 2 Architecture- *Dependency Injection*

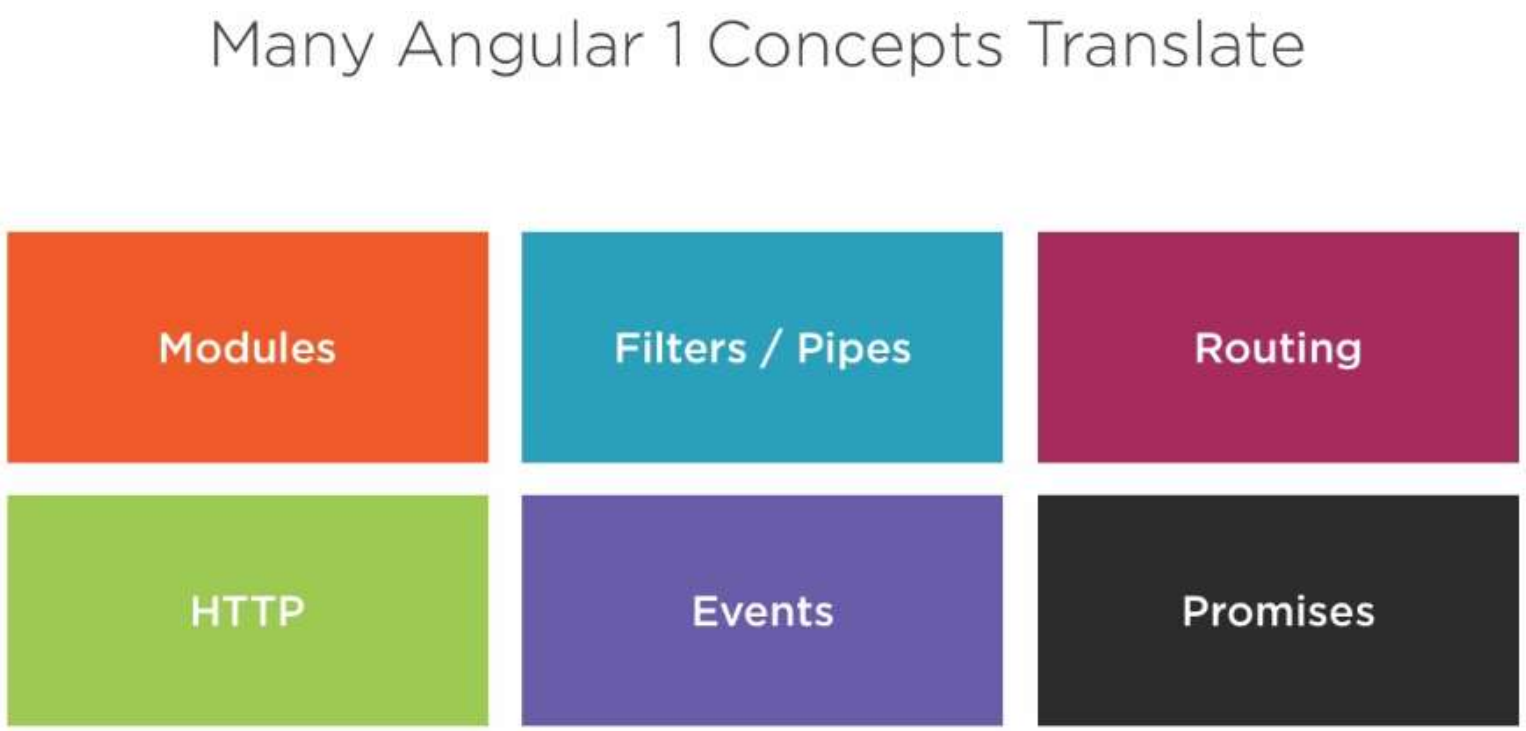
- If there are multiple components and they all need the VehicleService, it has to be **registered just once at the highest level component, the parent component.**
- This is because then, anybody who's a child of that component, let's say vehicles is the parent, anybody who's a child of that component would also have access to this VehicleService.
- If registerd in every component, it would actually get a different instance to the service everywhere, and that's probably not the requirement in case states are managed.

Just explored seven key comparisons from Angular 1 to Angular 2



1. Components are a key aspect because that's what became of controllers, so our components now contain our logic and a reference to our HTML template.
2. Learned how to bootstrap our apps in Angular 2.
3. Data binding, one way, two way, interpolation as well.
4. There are a lot of different directives in Angular 1, ngdashes, that have been removed for Angular 2.
5. But still have some structural directives, like ngFor and ngIf for putting things in the DOM.
6. Services still exist, they're just simpler.
7. How to dependency injection into the components.

Angular 2 Architecture- *Anuglar 1 Concepts That will be used in Angular 2*



- These are many other Angular 1 concepts that translate over to Angular 2 as well.
- We'll be using **ES6 modules**, and **filters became pipes**.
- We still have **routing** and **HTTP**.
- There's a new way to do eventing, using something called the EventEmitter.
- We still have **promises**, although we have another option called RxJS, which uses observables.

Angular 2 Resources

Angular Documentation

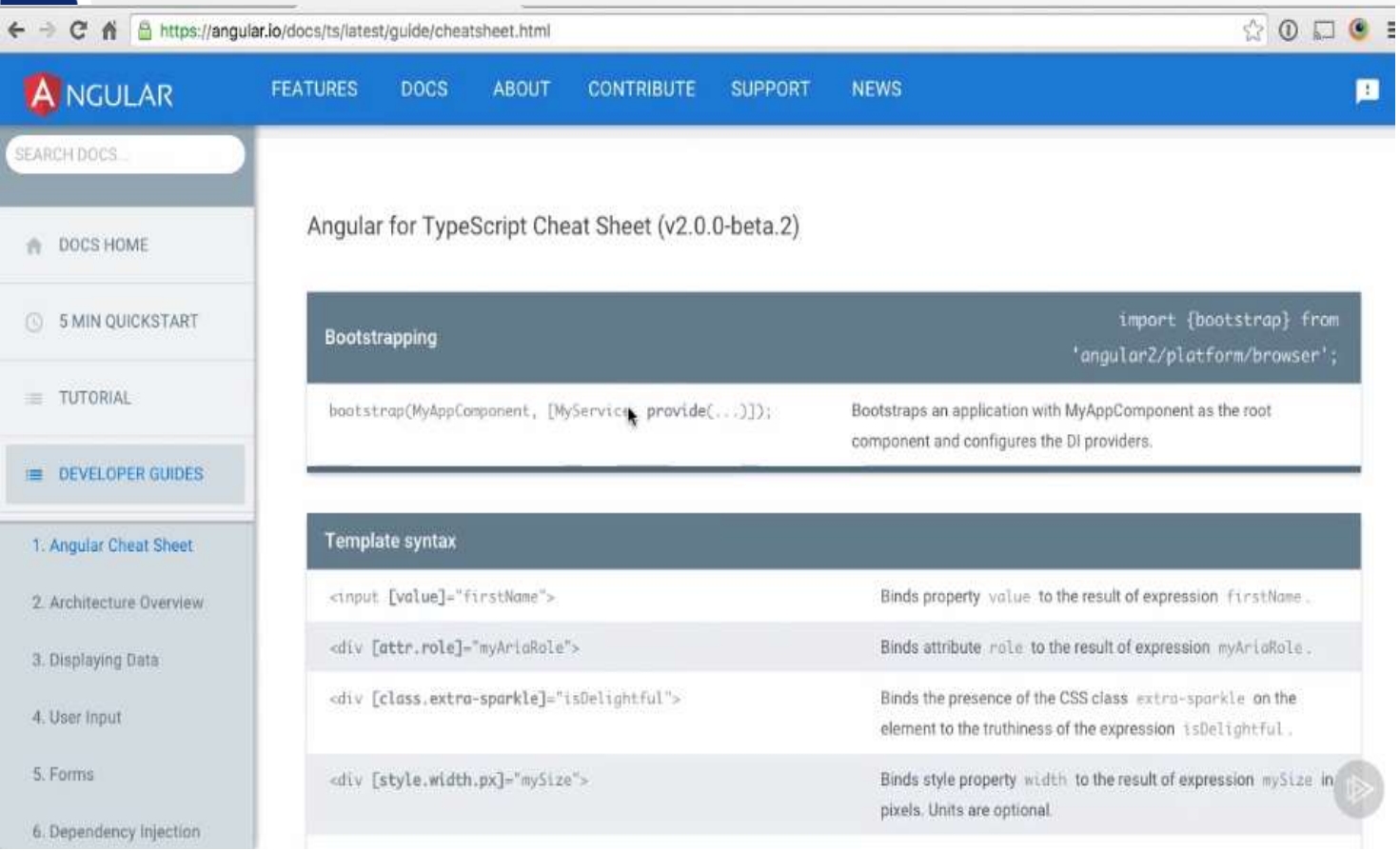
http://angular.io/docs

- First, they've got an Angular cheat sheet, which they have one for each
- language type.
- This one here is a snippet of the one for TypeScript. So if you're looking for a little bit of syntax here or there, this is a great one-stop shop.

Angular 2 Resources

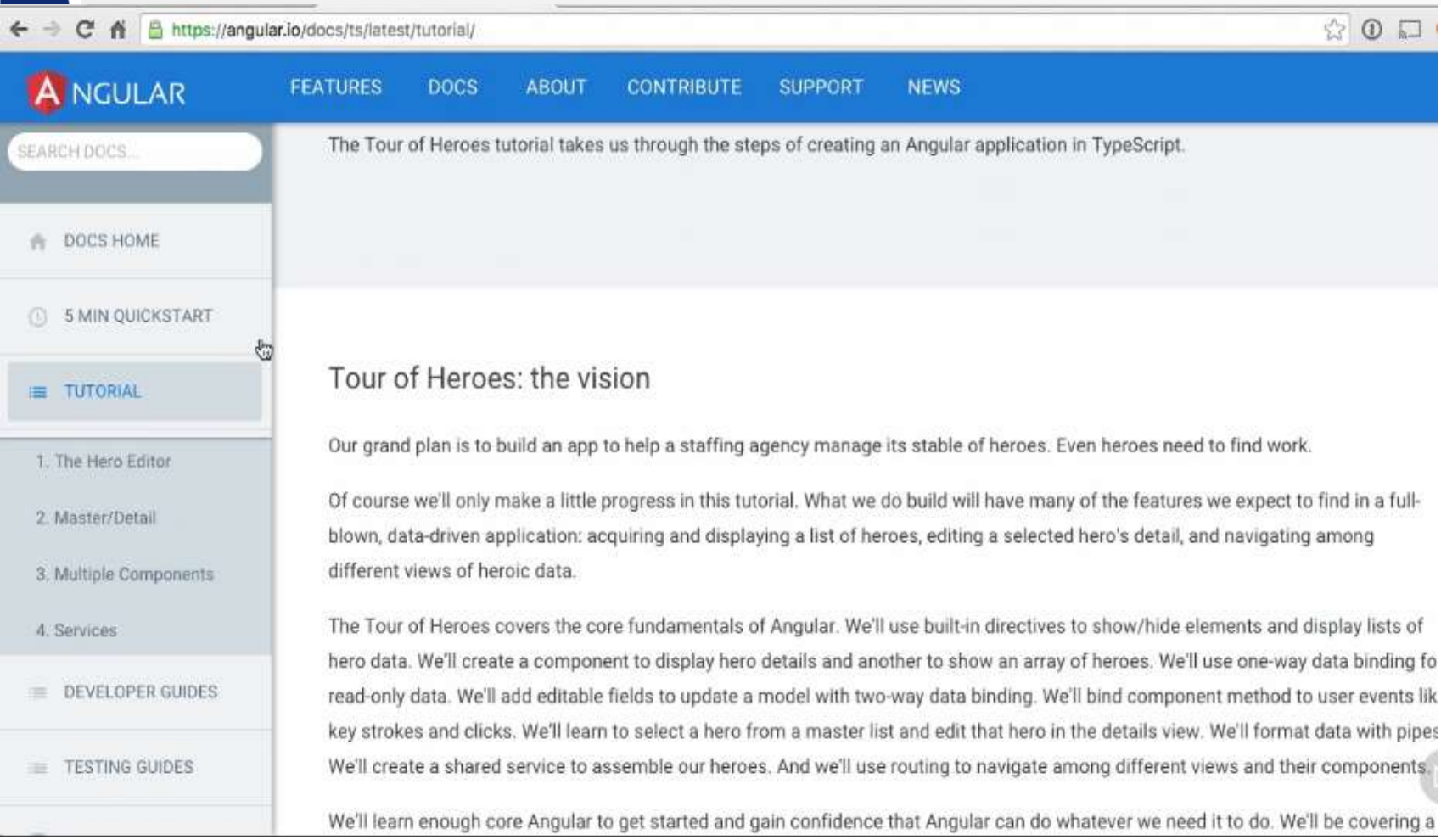
- They also have a tutorial called the **Tour of Heroes**, which will walk you step by step through building an application where the application's a list of heroes, and we get to walk through creating components and services and go through all the basics of Angular.

Angular 2 Resources



Here's the angular.io website.
And if we click up on DOCS, we'll see over here on the left there's a DEVELOPER GUIDE, and there's a tutorial.
Let's click on DEVELOPER GUIDES, and the first thing under there is the Cheat Sheet.
We click on that, we can see down here it shows us how we can bootstrap, our templating syntax.
It also shows us all the built-in directives that we have.
Notice how short that list is.

Angular 2 Resources



- And if we go over to the TUTORIAL, you can see the Tour of Heroes, which walks you through the different steps that are there, and they're adding steps all the time.
- Of course there's also a 5 MIN QUICKSTART and some TESTING GUIDES and API docs.

Steps for Up Angular2 Application

npm

Node Package Manager

Command line utility

Installs libraries, packages, and applications

https://www.npmjs.com/

Steps for Up Angular2 Application

Setting up an Angular 2 Application

1. Create an application folder
2. Add package definition and configuration files
3. Install the packages
4. Create the app's Angular Module
5. Create the main.ts file
6. Create the host Web page (index.html)

Ways to Setup an Angular2 Application…..

Setting up an Angular 2 Application

Manually perform each step

www.angular.io Quick Start

Download the results of these steps

https://github.com/angular/quickstart

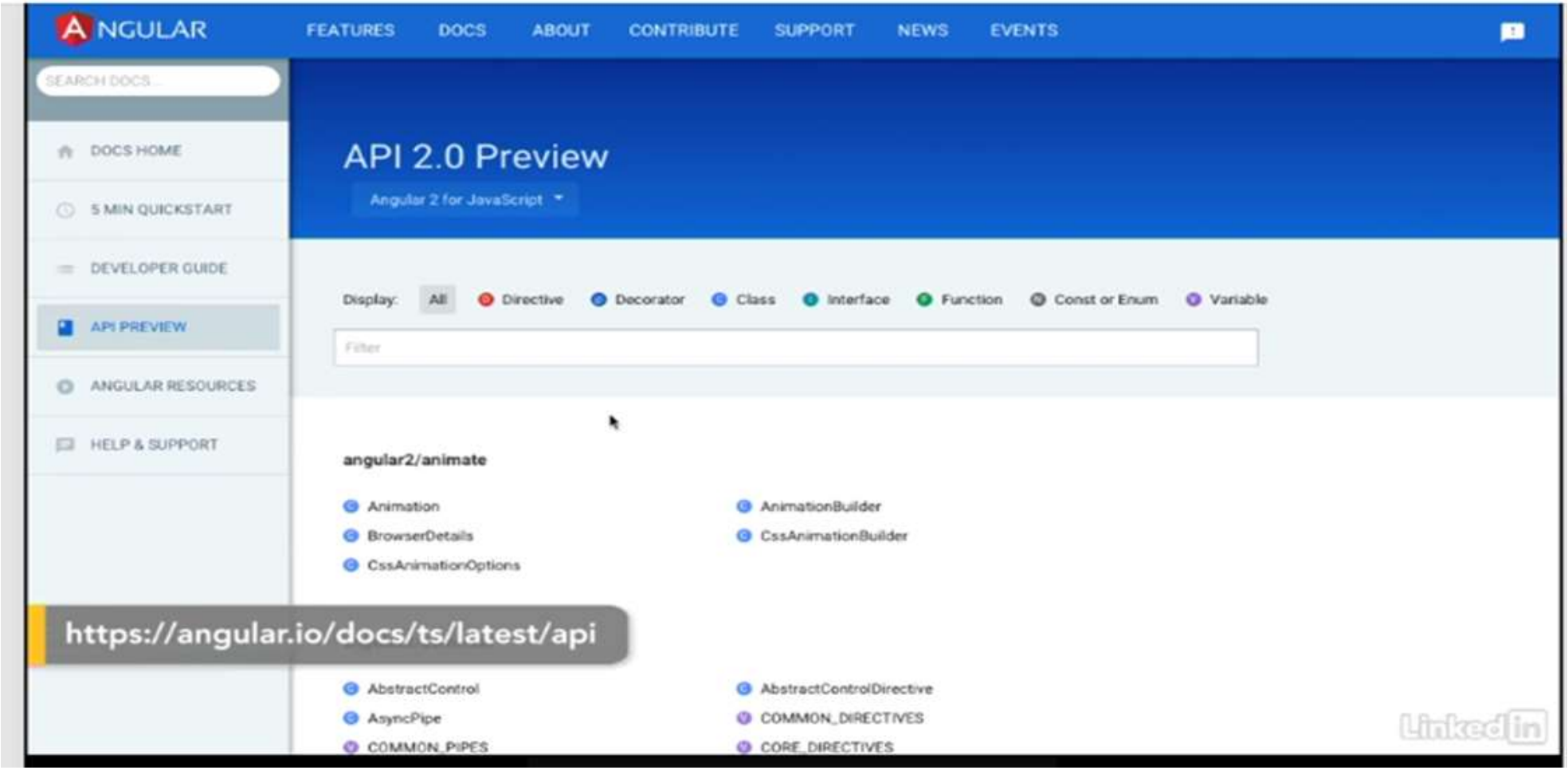
AngularCli

https://github.com/angular/angular-cli

Starter files

https://github.com/DeborahK/Angular2-GettingStarted

Angular 2 Essentials: Downloading Angular2 Script file



Angular 2 Essentials:The Angular2 project files

1. Karma.conf.js: It's the karma configuration file at the root of the project
module.exports=require('./config/karma.conf.js').
Karma is the testing framework used for testing

2. Next we have the package.json file. The start script will launch the webserver to allow us to preview the project.
test will be used to test the project. Build will be used to compile the project using webpack

package.json

```
{
  "name": "angular2-webpack-starter",
  "version": "1.0.0",
  "description": "A webpack starter for Angular",
  "scripts": {
    "start": "webpack-dev-server --inline -progress
      --port 8080",
    "test": "karma start",
    "build": "rimraf dist && webpack --config
      config/webpack.prod.js --progress --profile --bail",
```

Angular 2 Essentials:The Angular2 project files

Project Files

package.json(continued)

Versions of these packages may be different in the exercise files.

```
"dependencies": {
    "@angular/common": "~2.1.0",
    "@angular/compiler": "~2.1.0",
    "@angular/core": "~2.1.0",
    "@angular/forms": "~2.1.0",
    "@angular/http": "~2.1.0",
    "@angular/platform-browser": "~2.1.0",
    "@angular/platform-browser-dynamic": "~2.1.0",
    "@angular/router": "~3.1.0",
    "core-js": "^2.4.1",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "^0.6.25",
    "foundation-sites": "^6.2.4",
    "jquery": "^3.1.1"},
```

Angular 2 Essentials:The Angular2 project files

package.json(continued)

```
    "@types/core-js": "^0.9.34",
    "@types/jasmine": "^2.5.38",
    "@types/node": "^6.0.51"
    }
}
```

In angular 2 projects, done previously, we had to imort types into the project manually.
Types are definition files that allow typescript to understand the syntax of 3rd party library.
With the release of typescript version 2, these types are now installed through npm which eases the process

Angular 2 Essentials:The Angular2 project file

3. The tsconfig.json file handles the configuration of the typescript compiler. Angular 2 uses the decorators which is mentioned here

tsconfig.json

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "moduleResolution": "node",
        "sourceMap": true,
        "emitDecoratorMetadata": true,
        "experimentalDecorators": true,
        "removeComments": false,
        "noImplicitAny": true,
        "suppressImplicitAnyIndexErrors": true
    }
}
```

4. webpack.confg.js file
5. Inside the configuration folder we have helpers.js. This file stores the correct path of webpack

Project Files

helpers.js

```
var path = require('path');

var _root = path.resolve(__dirname, '..');

function root(args) {
  args = Array.prototype.slice.call(arguments, 0);
  return path.join.apply(path, [_root].concat(args));
}

exports.root = root
```

Angular 2 Essentials: The Angular2 project files

Angular 2 Essentials:The Angular2 project files

6. The source folder contains the index.html file. This is the start up of the application.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <base href="/">
    <title>Angular With Webpack</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1">
  </head>
  <body>
    <my-app>Loading...</my-app>
  </body>
</html>
```

Angular 2 Essentials: Webpack

Webpack is a module bundler.
It is an alternative to system.js file used in angular 2
System.js works fine but webpack has a few features
1. Webpack allows scripts to add dynamically in our project. In index .html file we have not added references of .js and .css files. This is because webpack will add those references
2. Webpack can handle various file types (.js, .html,.css,.png) to its various loaders. For example generally to compile a sass file a separate process needs to be set up but with webpack sass loader it will compiled on the fly.
3. The most appealing part of webpack is its module bundling. This will take all the files and compile into one big file which is advantageous when deployment.
4. Webpack imports vendor libraries such as Angular 2, jQuery etc

Webpack Configuration

webpack.common.js

```
var webpack = require('webpack');
var HtmlWebpackPlugin = require('html-webpack-plugin');
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var helpers = require('./helpers');
```

Webpack.common.js file contains common webpack setting that will be used across the application

Angular 2 Essentials: Webpack

webpack.common.js (continued)

```
module.exports = {
  entry: {
    'polyfills': './src/polyfills.ts',
    'vendor': './src/vendor.ts',
    'app': './src/main.ts'
  },
```

Everything in webpack.js is wrapped inside export statement.
This will allow to export files into other configuration files.
The first thing we have is the entry point, which represents the bundle files generated when the project is built

Angular 2 Essentials: Webpack

polyfills.ts

```
import 'core-js/es6';
import 'core-js/es7/reflect';
require('zone.js/dist/zone');

if (process.env.ENV === 'production') {
  // Production
} else {
  // Development
  Error['stackTraceLimit'] = Infinity;
  require('zone.js/dist/long-stack-trace-zone');
}
```

This file contains the compatibility libraries that are needed for angular 2 to work

The vendor.ts file consist the imported libraries that make our app work.

The resolve section of webpack.js has a module directory that points at the node modules folder from where webpack knows to find the libraries it needs

The loaders section of webpack.js loads the various file types of our app

Angular 2 Essentials: The Root Module

An angular 2 application can have multiple modules but it should have at least one root module (app.module.ts)

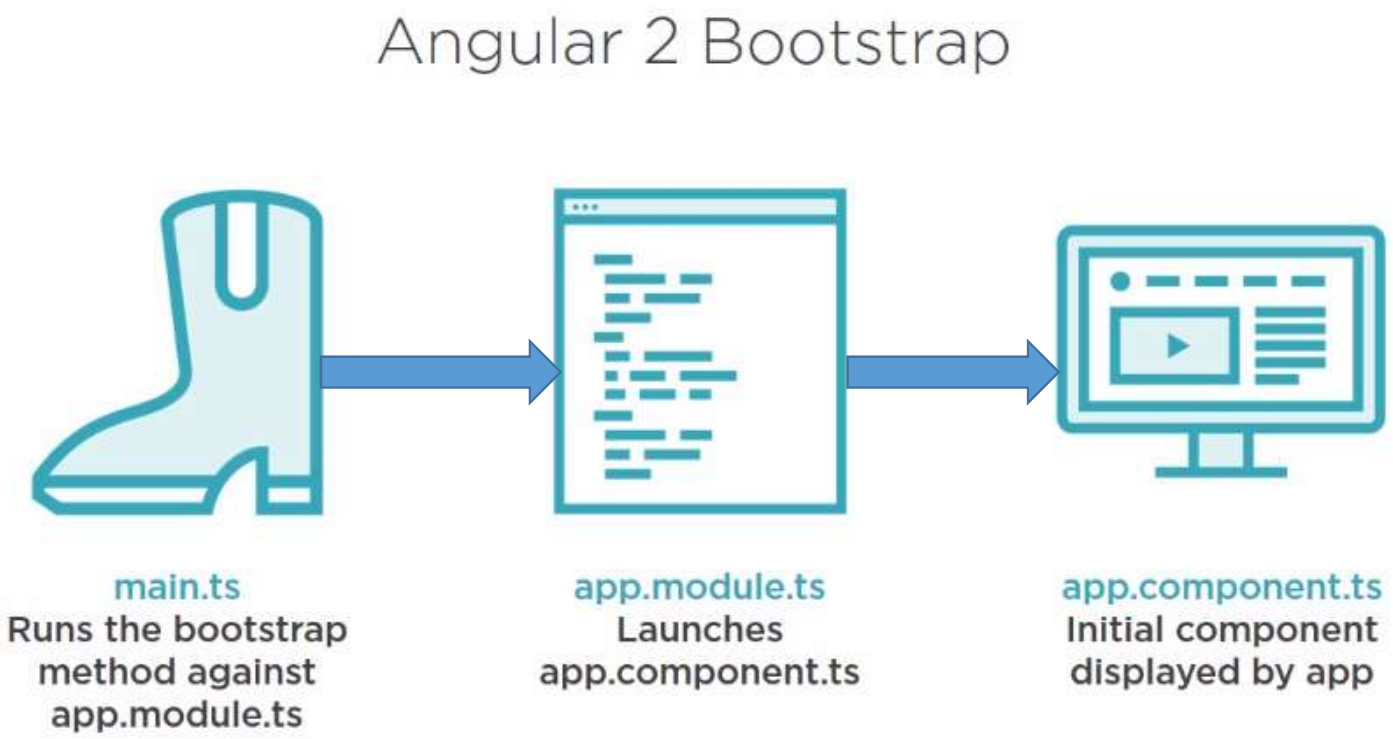
app.module.ts

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

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

```
export class AppModule { }
```

Always Needed for Browser Apps!

Angular 2 Essentials: Bootstrapping ng 2 application



Angular 2 Essentials: main.ts to bootstrap the app

main.ts

Entry point to app bundle

Bootstrap

Angular 2 Essentials: Bootstrap the app

main.ts

```
import { platformBrowserDynamic } from '@angular/platform-
browser-dynamic';

import { enableProdMode } from '@angular/core';

import { AppModule } from './app/app.module';

if (process.env.ENV === 'production') {

  enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Angular 2 Form Technologies

Template-driven Forms	Reactive Forms
Use a Component's Template	Use a Component's Template
Unit Test Against DOM	Create a Form Model in TypeScript (this must be in sync with the template)
	Unit Test Against Form Model
	Validation in Form Model

Data Binding and ngModel

```
<!-- no binding -->
<input name="firstname" ngModel>

<!-- one way binding -->
<input name="firstname" [ngModel]="firstName">

<!-- two way binding -->
<input name="firstname" [ngModel]="firstName"
                        (ngModelChange)="firstName=$event">

<!-- two way binding -->
<input name="firstname" [(ngModel)]='firstName'>
```

Angular 2 Essentials: Form Validation

CSS Classes for Validation

ng-untouched ng-touched	ng-pristine ng-dirty	ng-valid ng-invalid

CSS Classes for Validation

ngModel Properties for Validation

untouched touched	pristine dirty	valid invalid

Reactive Extensions

A library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators.

Its is a comprehensive library for async programming and has nothing to do with angular.
Angular 2 has used some part of rxjs , the reference of which is found in index.html and is also listed in package.json dependencies

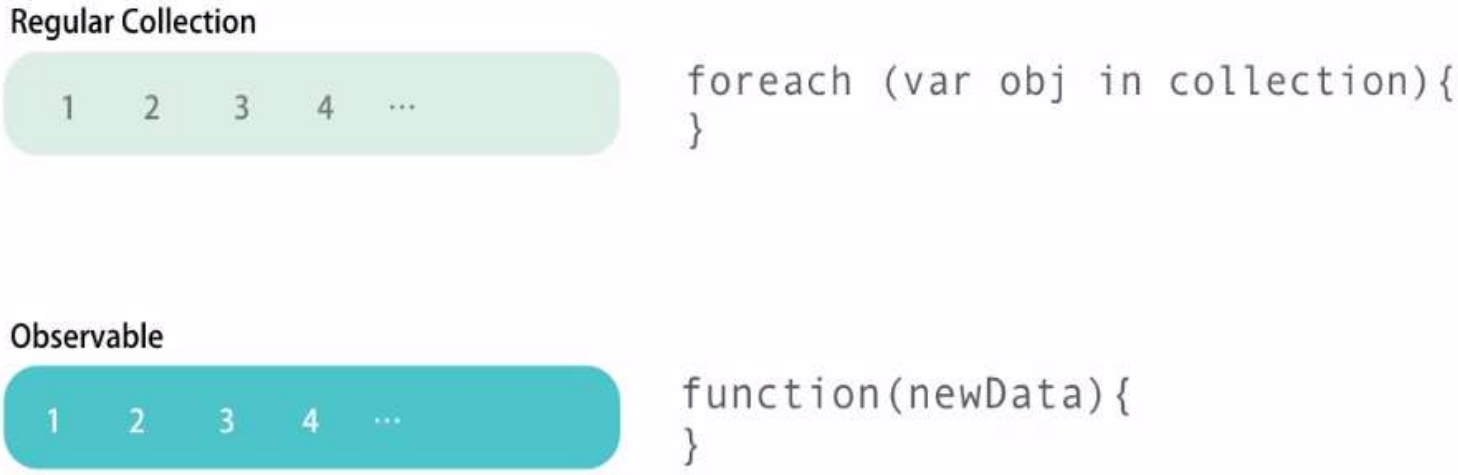
Angular 2 Essentials: Reactive Extensions

```
/// <reference path="../typings/tsd.d.ts" />

import {Component} from 'angular2/core';

@Component({
    selector: 'my-app',
    template: `
        <input id="search" type="text" class="form-control" placeholder="Sea
    `
})
export class AppComponent {
    constructor(){
        $("#search").keyup(function(e){
            var text = e.target.value;

            var url = "https://api.spotify.com/v1/search?type=artist&q=" + te
            $.getJSON(url, function(artists){
                console.log(artists);
            });

        });
    }
```

Angular 2 Essentials: Observables

Angular 2 Essentials: Observables

Observable as a Collection

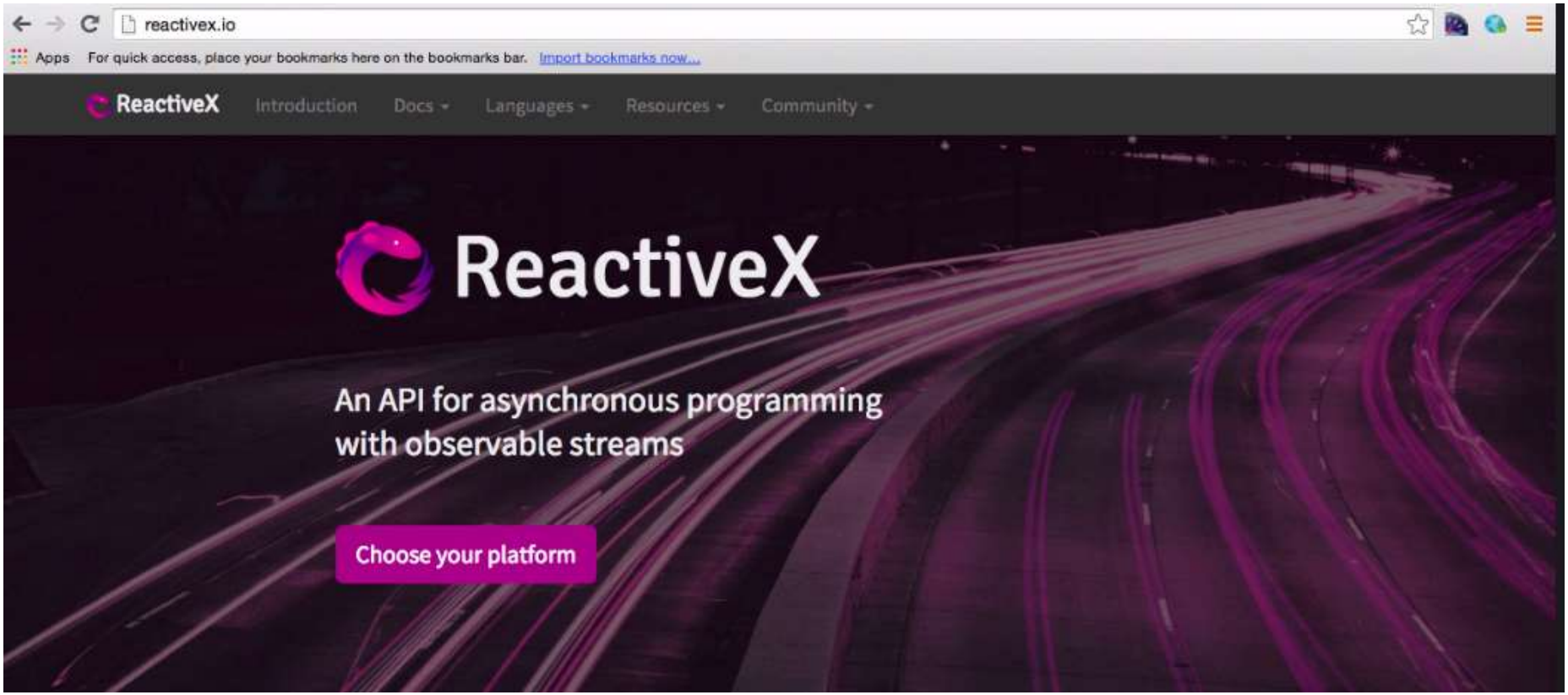


You can think of observables as a collection.
In regular collection we pull one object at a time.
In observable streams we subscribe to the collection and give it a callback function..
When new data is arrived the data is pushed by notifying.
Hence this is called observable as it notifies when data arrives asynchronously

Angular 2 Essentials: Reactive Extensions

Observables provide a set of standard and unified operators that we can use to transform,filter, aggregate, map combine etc.

Angular 2 Essentials: ReactivExtensions



Angular 2 Essentials: Modules, Components, Templates & Metadata

Overview

Modules

Components

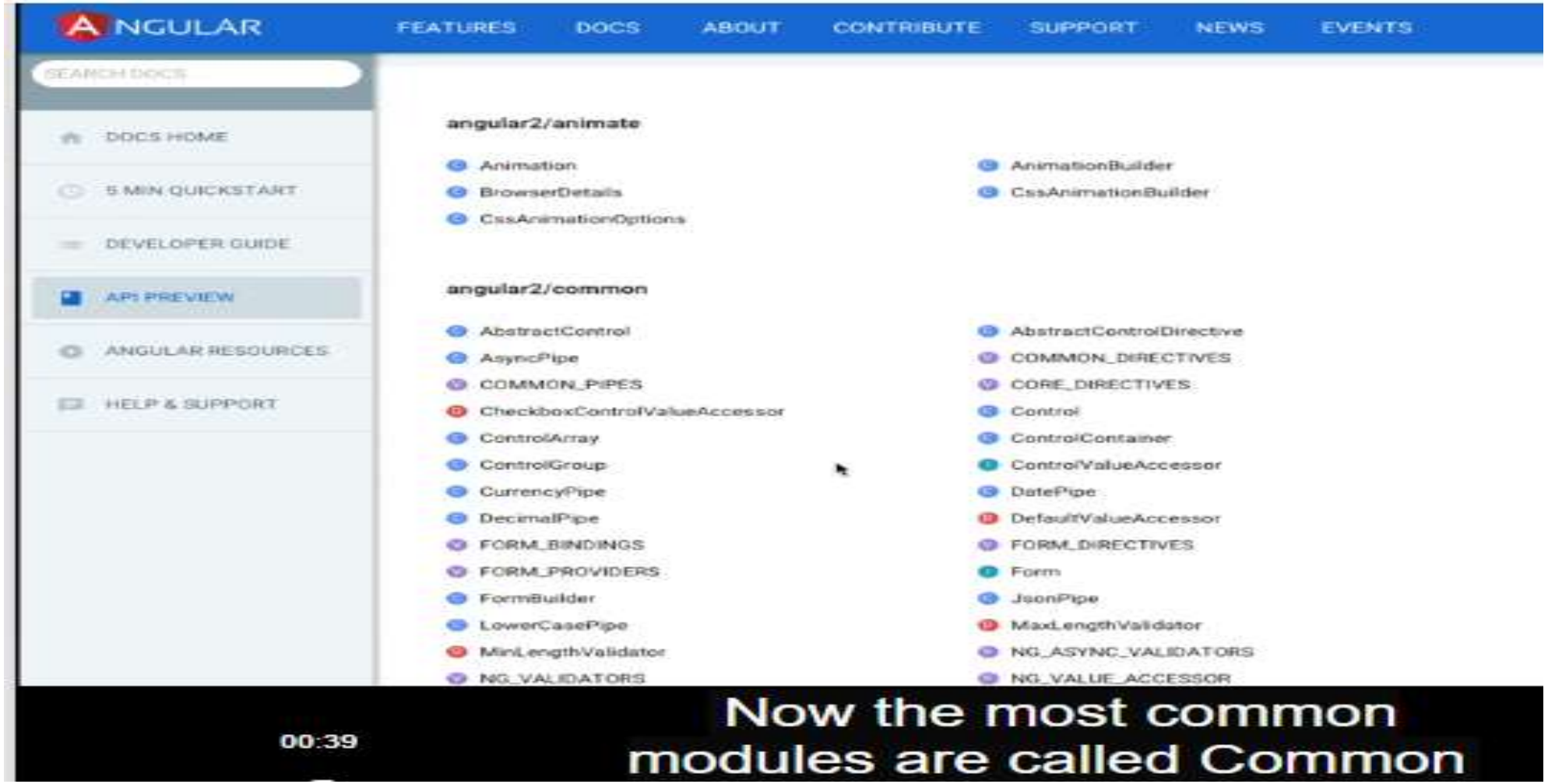
Templates

Metadata

- Every framework has a short list of critical players, players that the framework can't exist without.
- In Angular, those are components and templates.
- In this module, we'll introduce the essential aspects of an Angular 2 application.

- **Components** contain the application logic.
- The **template** represents the interaction with the user.
- A series of component and template combinations are organized into modules.
- We assemble our application for **modules**, which export something of value, such as a component.
- Component controls a region of the user interface through a template, which tells Angular how to render the component. We use **metadata** to tell Angular 2 about the objects we build, such as where to find a component's template, what directives a component will use, and which services and inputs are required by the component.

The common modules are found as below, in the same link in prev. slide

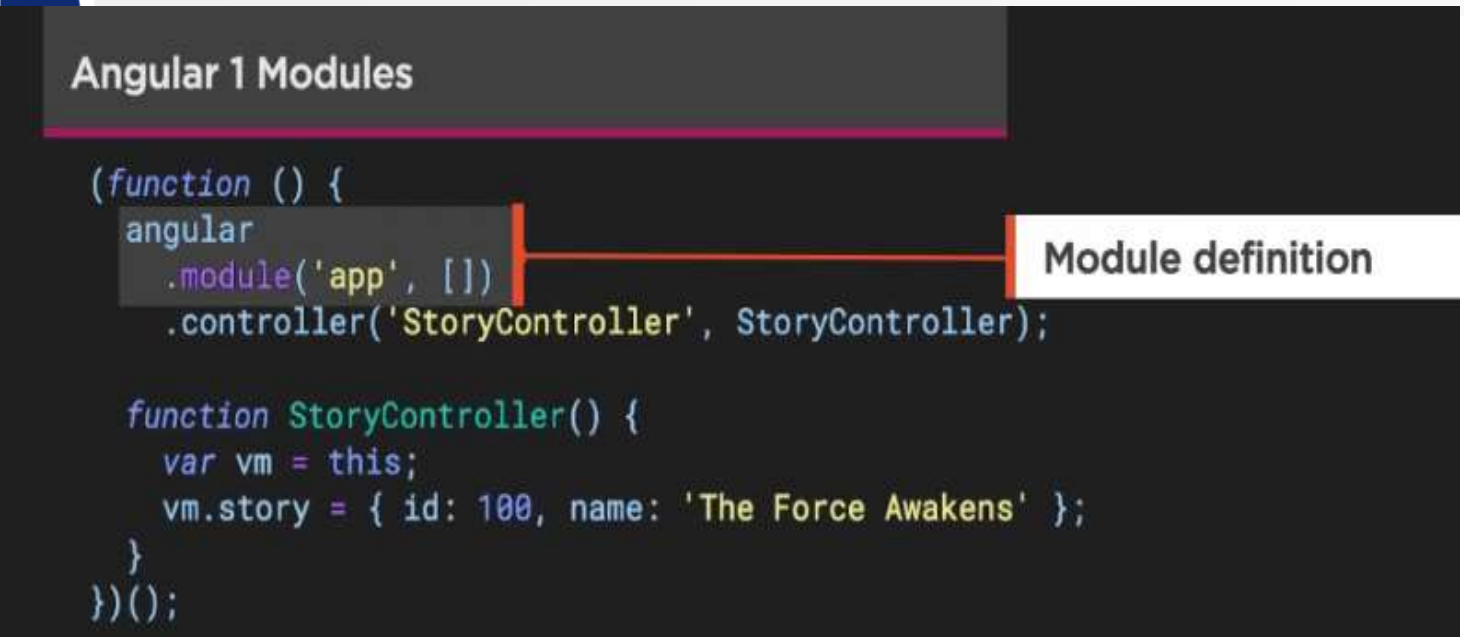


Modules

We assemble our application from modules.

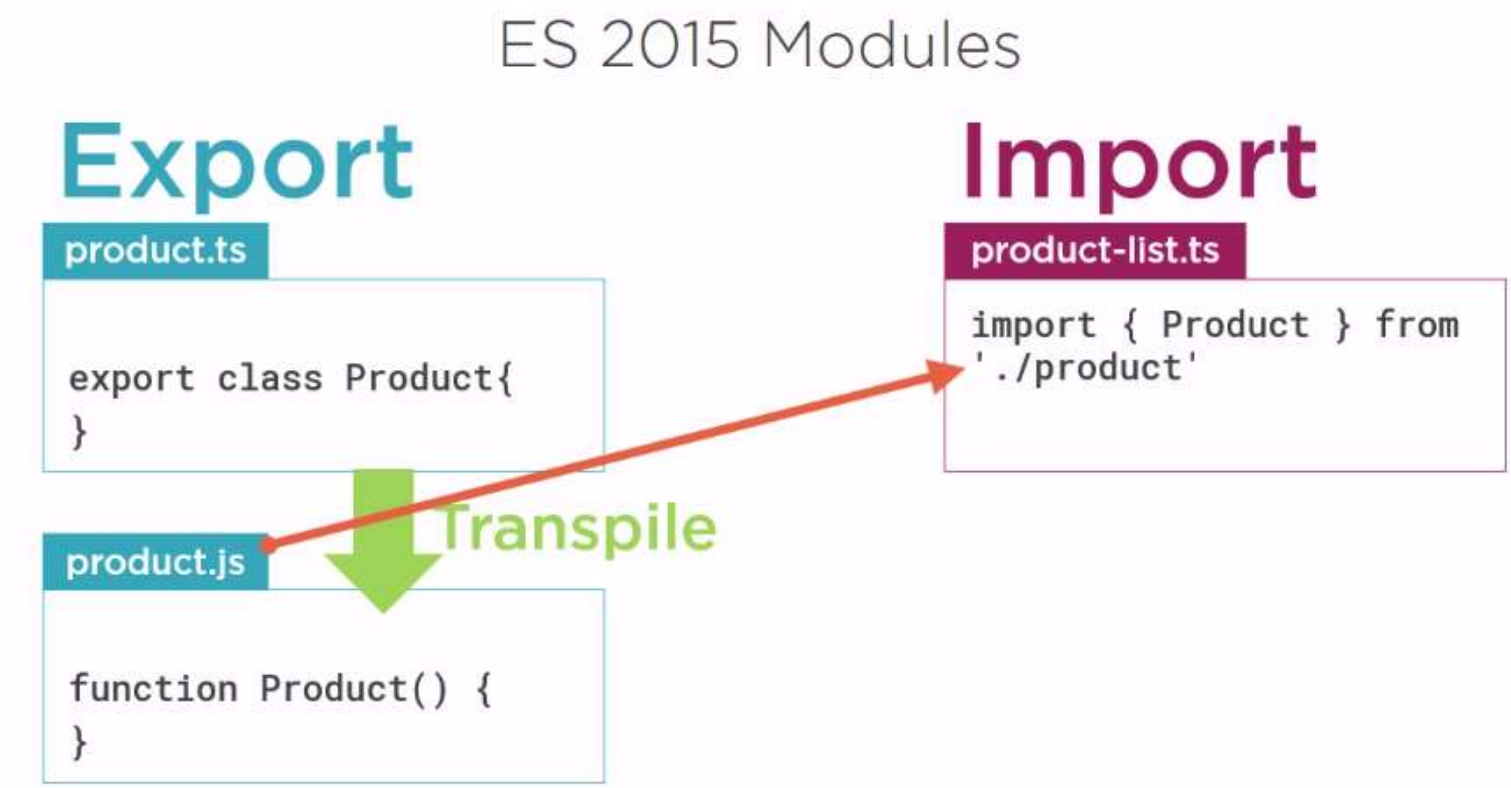
A module exports an asset such as a Service, Component, or a shared value

Modules

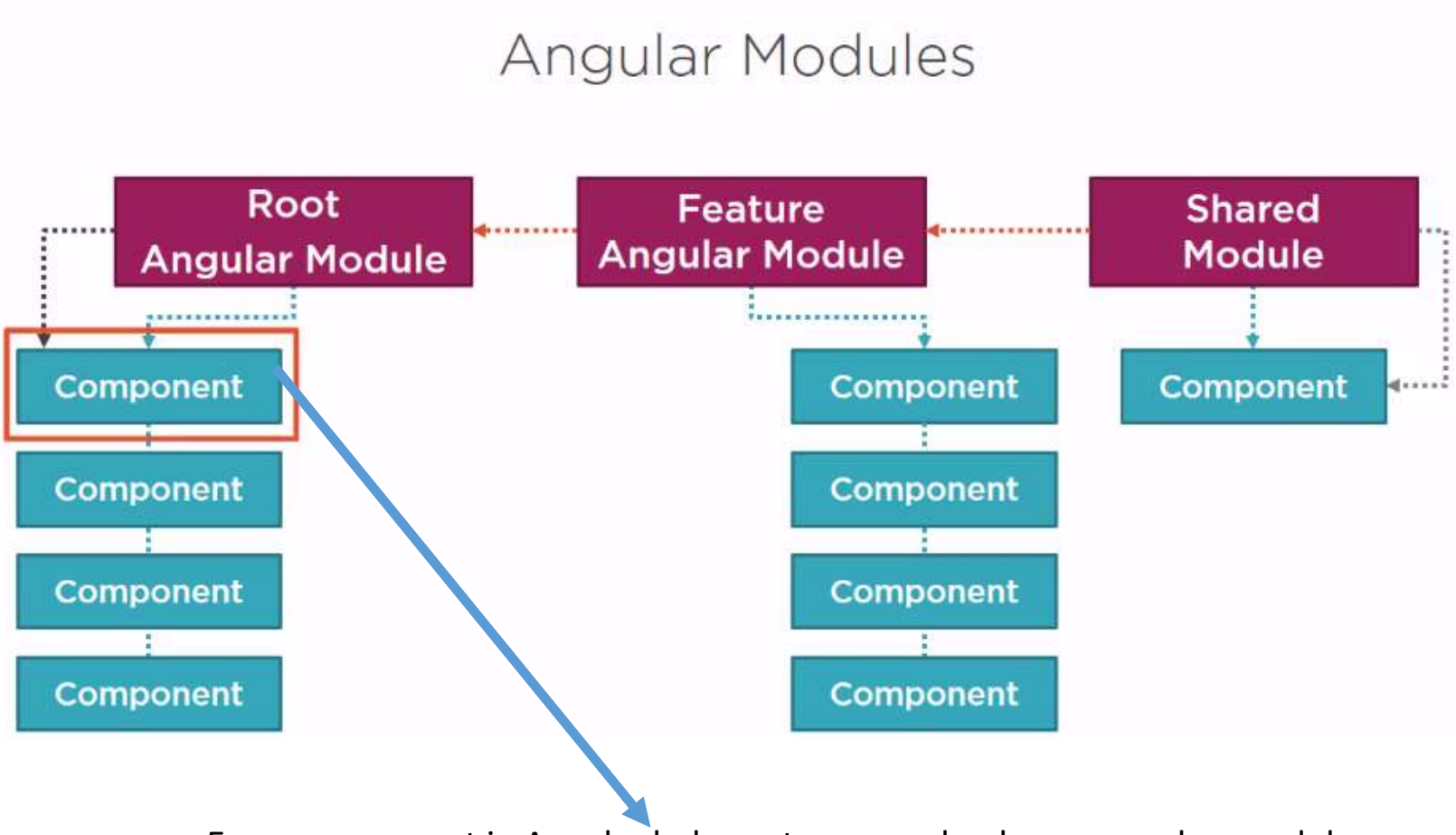


We use ES6 style modules with Angular 2

Modules


ES 2015 Modules
Export
product.ts
export class Product{
}
Transpile
product.js
function Product() {
}
Import
product-list.ts
import { Product } from
'./product'

Modules



Every component in Angular belongs to one and only one angular module

Modules

ES Modules	Angular Modules
Code files that import or export something	Code files that organize the application into cohesive blocks of functionality
Organize our code files	Organize our application
Modularize our code	Modularize our application
Promote code reuse	Promote application boundaries

Components –Steps to create a component

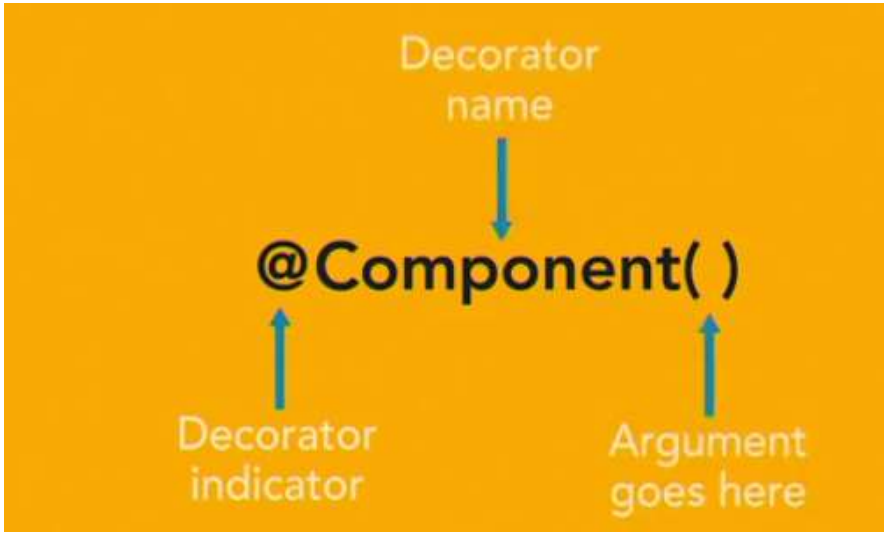
Angular leverages decorators to help configure ES2015 classes.
A decorator is an expression that evaluates to a function that makes it possible to annotate and modify classes at design time.
Angular has a bunch of decorators that it provides in the framework.
TypeScript provides support for decorators through its transpiler.
The component decorator provided by the Angular framework to tell Angular that you intend a class to be treated as a component.
The syntax for using a decorator is the @ symbol followed by the decorator name and then a pair of parentheses.
To build an Angular component, you need to use the component decorator on a class.

Components –Steps to create a component

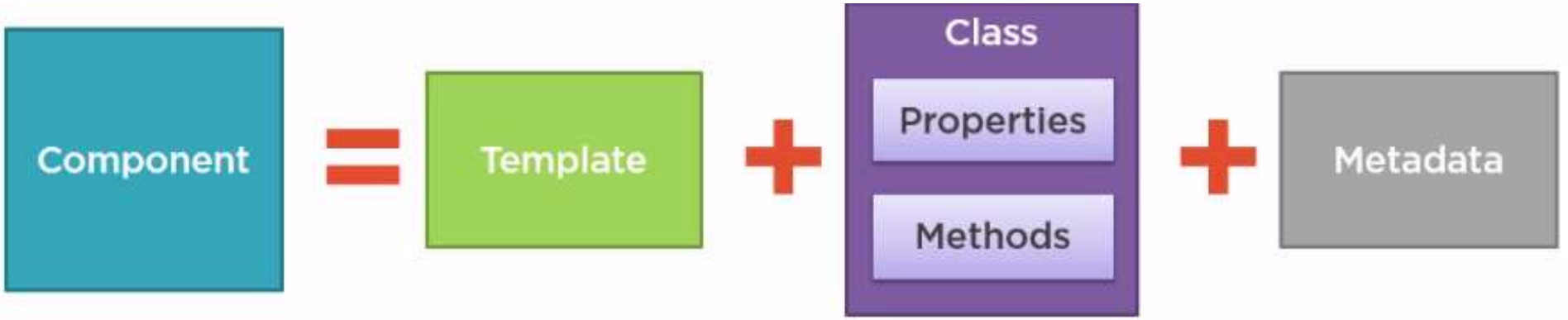
But to use the decorator, you need to import it using the ES2015 module loading syntax that TypeScript supports.
The first bit of code we need to write is an import statement.
In our case, we want to import the Component decorator.

Decorator

Expression that evaluates to a function allowing annotation of classes at design time

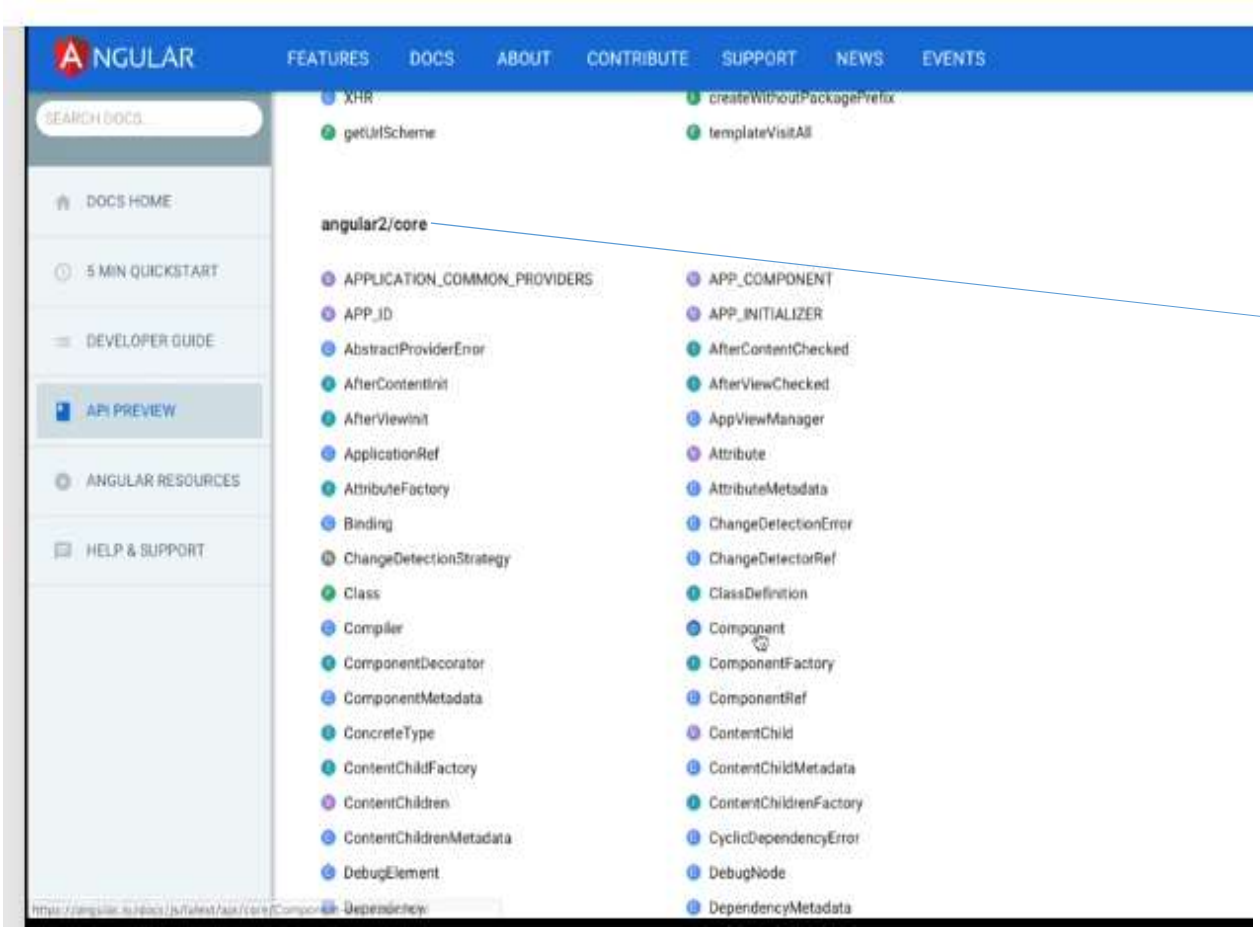


Angular 2 -Anatomy



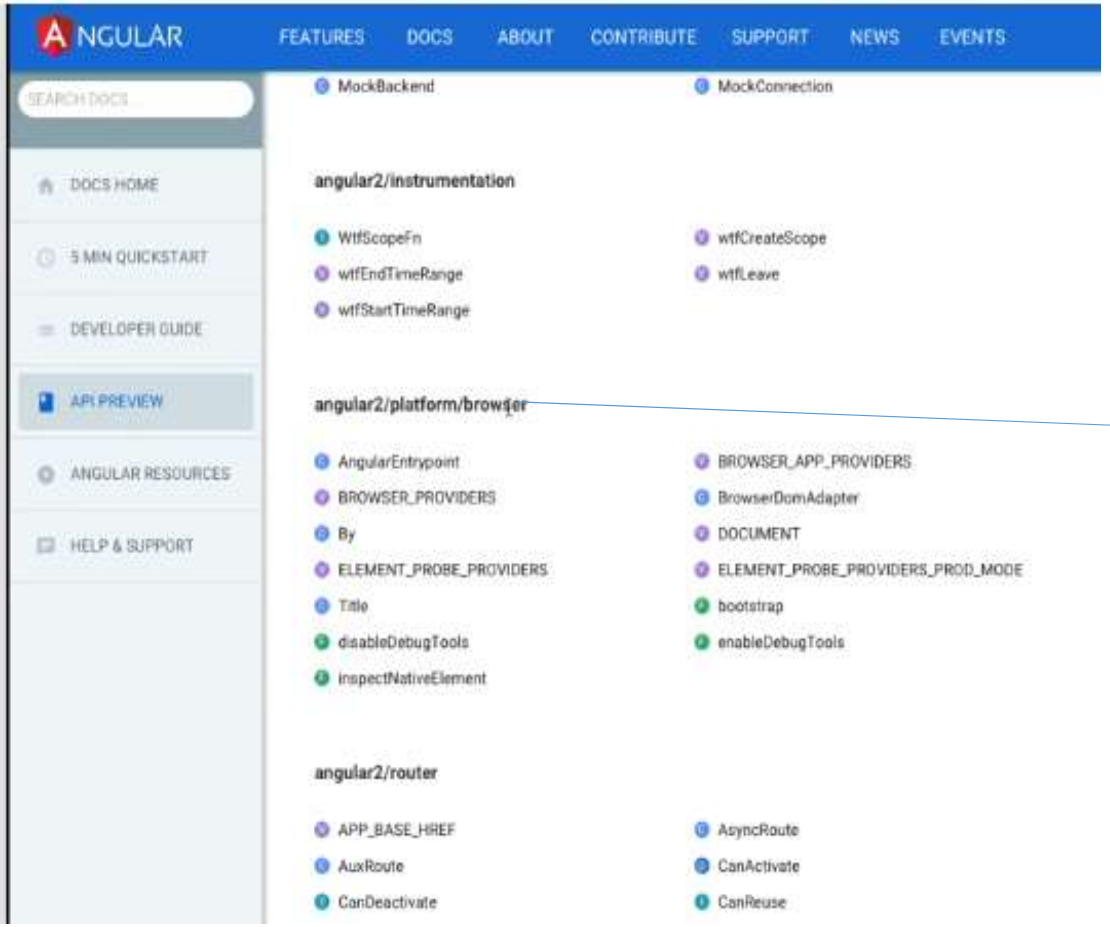
Each **component** in angular comprises of a **template**. The template is the HTML, the UI fragment defining the view for the application.
The component contains **a class** that contains the code associated for that view. This class contains the *properties* and *methods* for the data elements and actions for the view respectively.
A component also has a **metadata** that provides additional information about the component.

Components



- The other lib called Core, is in https://angular.io/docs/ts/latest/api
- In order to create a Component, we need to import the Component definition from Angular to Core.
- You can see it right here.

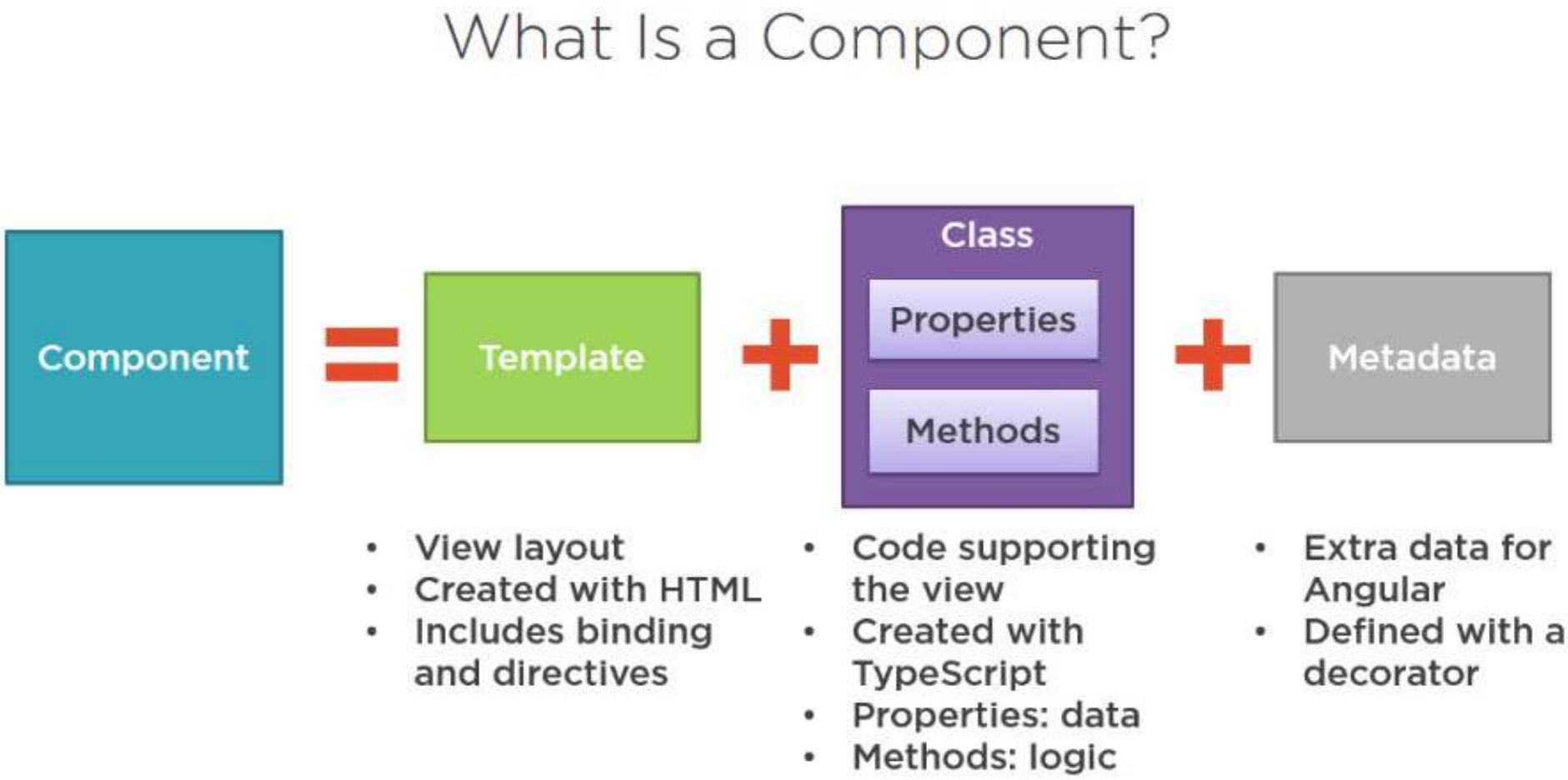
Click to edit Master title style



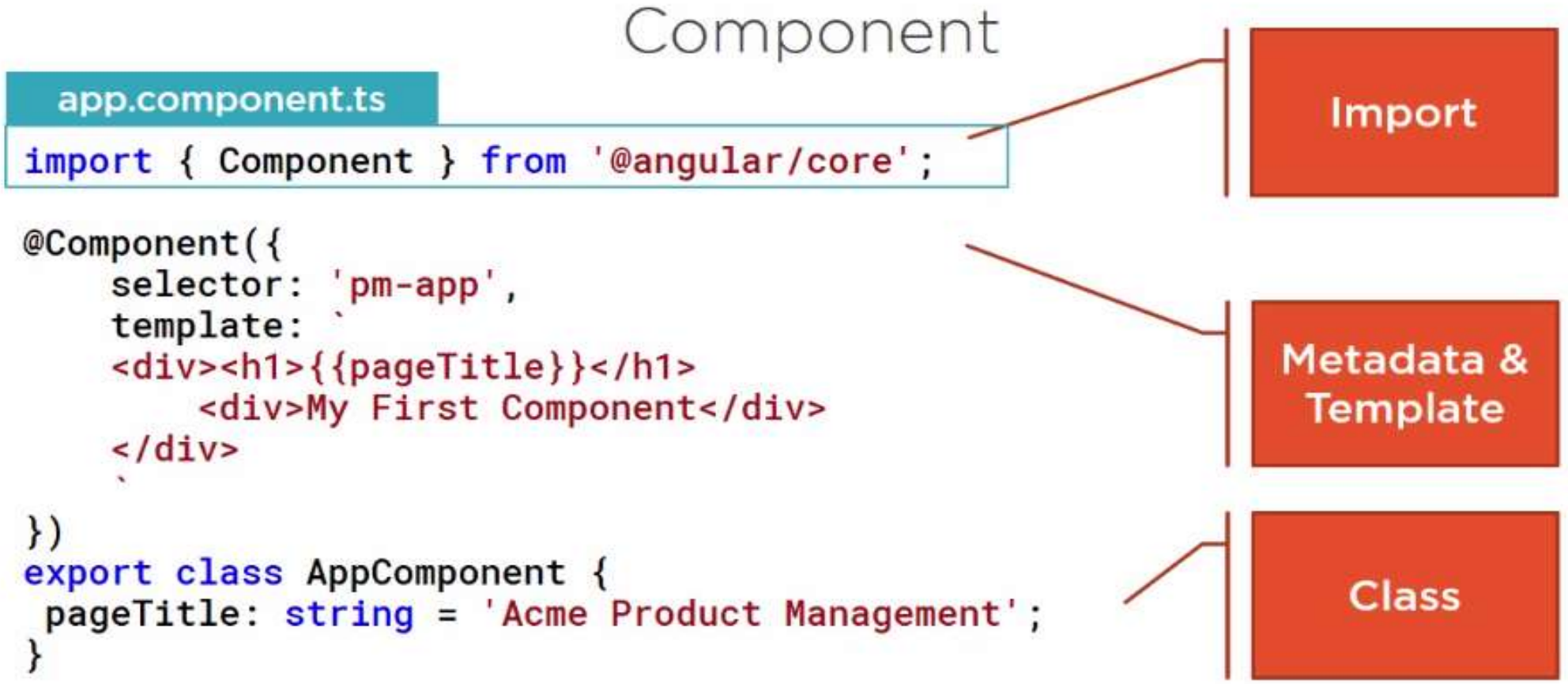
- Futher it is required to Bootstrap module and that one is further down in angular2/platform/browser.

  https://angular.io/docs/ts/latest/api

- The Boostrap module is right here.
- It's really a function that starts up the application.


What Is a Component?
Component
=
Template
+
Class
Properties
Methods
+
Metadata
• View layout
• Created with HTML
• Includes binding and directives
• Code supporting the view
• Created with TypeScript
• Properties: data
• Methods: logic
• Extra data for Angular
• Defined with a decorator

Components


Component
app.component.ts
import { Component } from '@angular/core';
@Component({
selector: 'pm-app',
template: `
<h1>{{pageTitle}}</h1>
My First Component

`
})
export class AppComponent {
pageTitle: string = 'Acme Product Management';
}
Import
Metadata & Template
Class

Creating the Component Class

app.component.ts

```
export class AppComponent {
    pageTitle: string = 'Acme Product Management';
}
```

class keyword

Class Name

export keyword

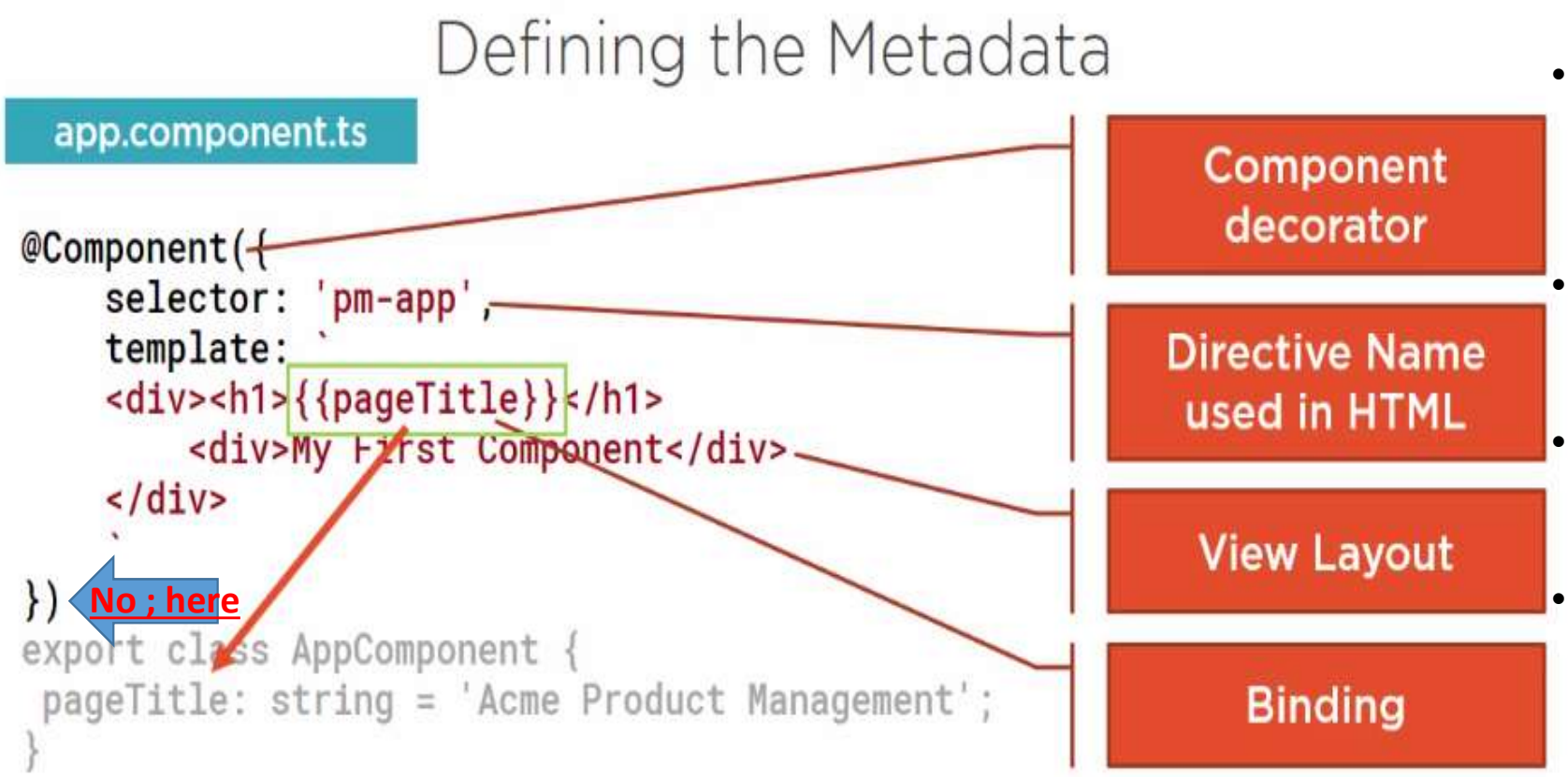
Component Name when used in code

Creating the Component Class

app.component.ts

```
export class AppComponent {
    pageTitle: string = 'Acme Product Management';
}
```

Components



- In the example here we are decorating a class.
- The decorator is defined immediately above the class signature.
- Decorator is a function hence round brackets after decorator name
- Then use the JS object syntax to pass various properties to decorator.
- The mandatory property inside a decorator is **"template"** or "templateUrl"

Decorator

A function that adds metadata to a class, its members, or its method arguments.

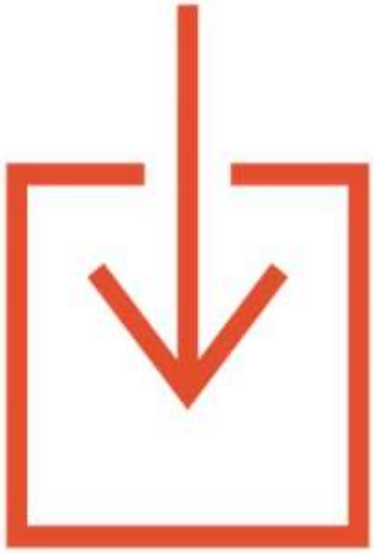
Prefixed with an @.

Angular provides built-in decorators.

```
@Component()
```

- The scope of the decorator is limited to the feature it decorates
- A decorator is a JS language feature that is implemented in Typescript and proposed for ES2016

Importing What We Need



Before we use an external function or class, we define where to find it

`import` statement

`import` allows us to use exported members from external ES modules

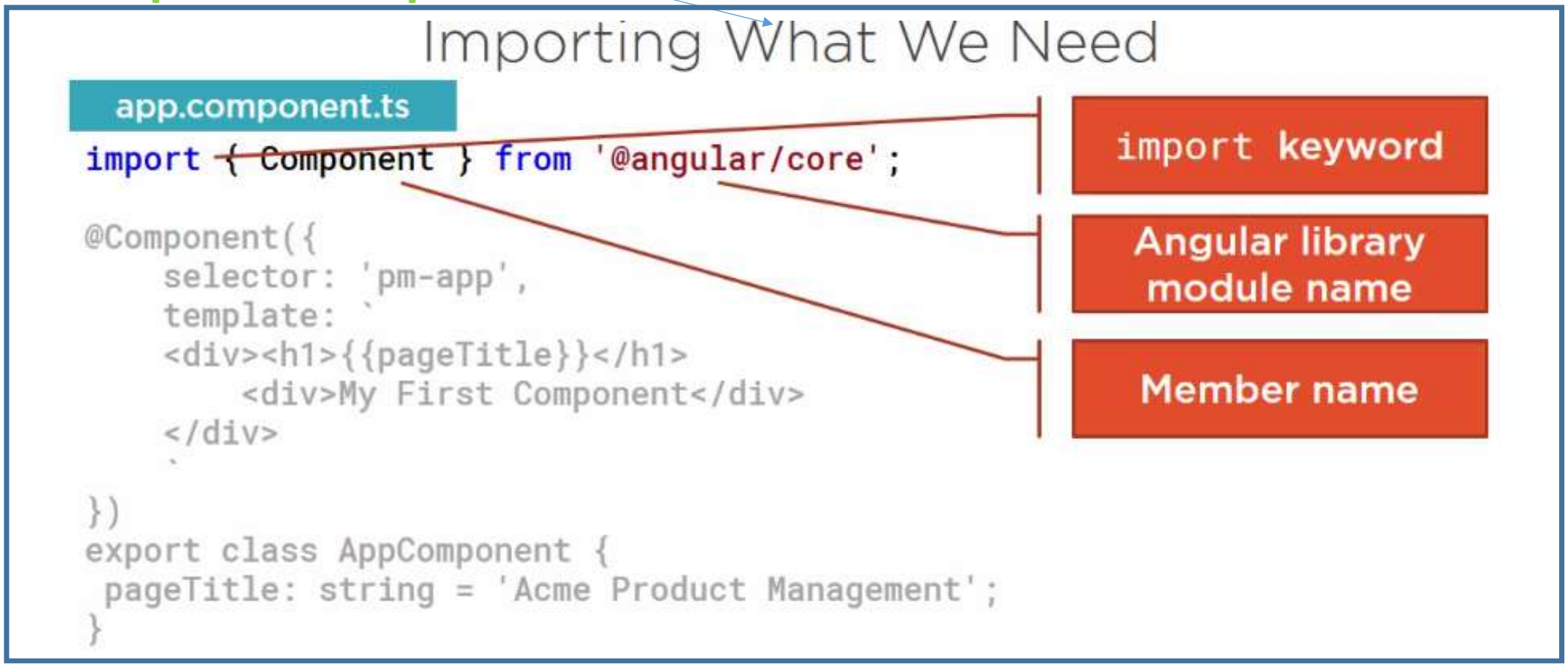
Import from a third-party library, our own ES modules, or from Angular

Components

- We can import from angular because angular is modular
- The following are some commonly used modules from angular 2 that can be imported into our angular application
- The complete list of angular 2 modules can be found in the link below

https://www.npmjs.com/~angular

A completed component



Bootstrapping the AppComponent

- **Bootstrapping** is the process of telling angular to load the root component through certain process
- Set up the index.html file to host our application
- **index.html** is the main page of the angular application
- Most of the time index.html is the only web page of the application and hence angular applications are often called as a Single Page Application (SPA)

Bootstrapping the AppComponent

Hosting the Application

index.html

```
<body>
  <pm-app>Loading App ...</pm-app>
</body>
```

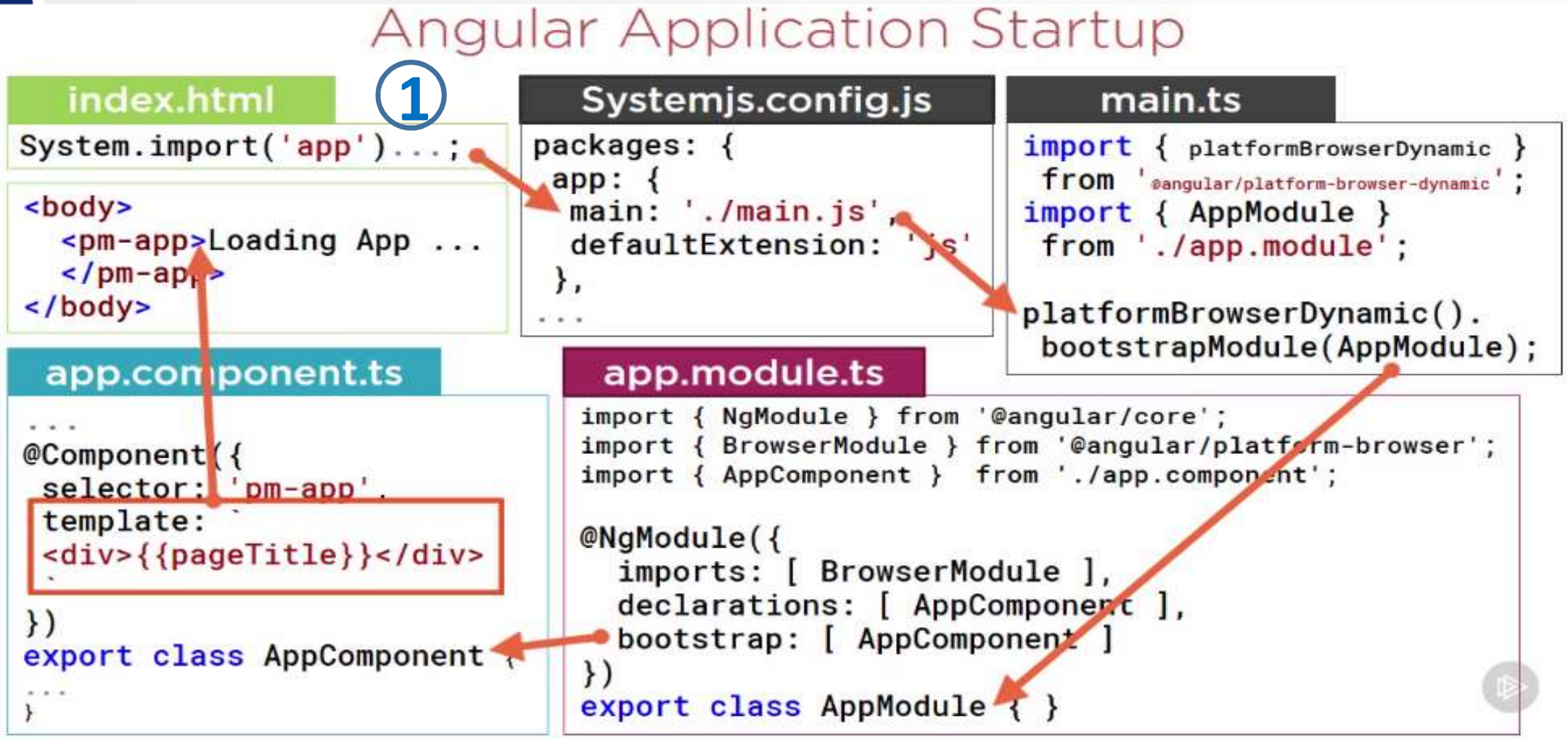
app.component.ts

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

@Component({
    selector: 'pm-app',
    template:
    <div><h1>{{pageTitle}}</h1>
        <div>My First Component</div>
    </div>

})
export class AppComponent {
 pageTitle: string = 'Acme Product Management';
}
```

- Selector is the name of the component that is used as directive in index.html
- Template defines the html that we want to display

- So in index.html file we simply add the selector where we want our template displayed. In the template we call it as directive.
- A directive is actually a custom HTML element

Bootstrapping the component



Component Checklist: Class

Clear name

- Use PascalCasing
- Append "Component" to the name

`export` **keyword**

Data in properties

- Appropriate data type
- Appropriate default value
- camelCase with first letter lowercase

Logic in methods

- camelCase with first letter lowercase

Component Checklist: Metadata

Component decorator

- Prefix with @; Suffix with ()

selector: Component name in HTML

- Prefix for clarity

template: View's HTML

- Correct HTML syntax

Component Checklist: Import

Defines where to find the members that this component needs

`import` **keyword**

Member name

- Correct spelling/casing

Module path

- Enclose in quotes
- Correct spelling/casing

Something's Wrong! Checklist

F12 is your friend

Recheck your code

- HTML
  - Close tags
  - Angular directives are case sensitive
- TypeScript
  - Close braces
  - TypeScript is case sensitive

Components --What we learnt so far

What is a Component?

Creating the Component Class

Defining the Metadata with a Decorator

Importing What We Need

Bootstrapping Our App Component

Defining a Template in a Component

Inline Template

```
template:
"<h1>{{pageTitle}}</h1>"
```

Inline Template

```
template: `
<div>
 <h1>{{pageTitle}}</h1>
 <div>
     My First Component
 </div>
</div>
`
```

ES 2015 Back Ticks

Linked Template

```
templateUrl:
'product-list.component.html'
```

Binding

Coordinates communication between the component's class and its template and often involves passing data.

Values from the class are provided to the template for display

The template in turn raises events to pass user actions

The binding syntax is always defined in the template

There are different ways to bind template to component class.

Here we are doing binding with **INTERPOLATION**

Interpolation

Template

```
<h1>{{pageTitle}}</h1>

{{'Title: ' + pageTitle}}

{{2*20+1}}

{{'Title: ' + getTitle()}}

<h1 innerText={{pageTitle}}></h1>
```

Class

```
export class AppComponent {
 pageTitle: string =
      'Acme Product Management';
 getTitle(): string {...};
}
```

Templates, Interpolation and Directives

- Interpolation uses the {{ }} syntax
- <h1>{{pageTitle}}</h1>, here pageTitle is the property of a component class
- Interpolation is one way binding from class to template
- With interpolation we can perform concatenation {{‘Title:’ +pageTitle}}
- Simple calculations {{2*20+1}}
- Can call class methods {{‘Title :’ + getTitle()}}
- We can use interpolation with element property assignments <h1 innerText={{pageTitle}}></h1>
- The syntax between {{ }} is called a template expression.
- Angular evaluates this expression using component as the context

Directive

Custom HTML element or attribute used to power up and extend our HTML.

- Custom
- Built-In

Angular Built-in Directives

A Structural Directive modifies the structure or layout of the view by adding, removing or manipulating elements
All structural directives are preceded by the *

Structural Directives

- *ngIf: If logic
- *ngFor: For loops

*ngIf Built-In Directive

```
<div class='table-responsive'>
 <table class='table' *ngIf='products && products.length'>
  <thead> ...
  </thead>
  <tbody> ...
  </tbody>
 </table>
</div>
```

*ngFor Built-In Directive

```
<tr *ngFor='let product of products'>
    <td></td>
    <td>{{ product.productName }}</td>
    <td>{{ product.productCode }}</td>
    <td>{{ product.releaseDate }}</td>
    <td>{{ product.price }}</td>
    <td>{{ product.starRating }}</td>
</tr>
```

Templates, Interpolation and Directives

*ngFor="let product of products" why 'of' and not 'in'
The reason is to do with ES2015 For loops.
ES2015 has both for of loop and for in loop
The for..of loop is similar to a for each loop style

for...of vs for...in

for...of

- Iterates over iterable objects, such as an array.
- Result: di, boo, punkeye

```
let nicknames= ['di', 'boo', 'punkeye'];

for (let nickname of nicknames) {
  console.log(nickname);
}
```

for...in

- Iterates over the properties of an object.
- Result: 0, 1, 2

```
let nicknames= ['di', 'boo', 'punkeye'];

for (let nickname in nicknames) {
  console.log(nickname);
}
```

Angular Template Syntax-Interpolation

- Interpolation is a way to get data displayed in the view.
- Interpolation is done by using a pair of matching curly braces in the markup. **{{ }}**
- The contents of the double curly braces is a JS expression that Angular will evaluate and then convert it to a string.

Angular Template Syntax-Interpolation

The most common use of **interpolation**, is to display some data from a property we have set on the component class.

Angular Template Syntax-Interpolation

media-item.component.ts

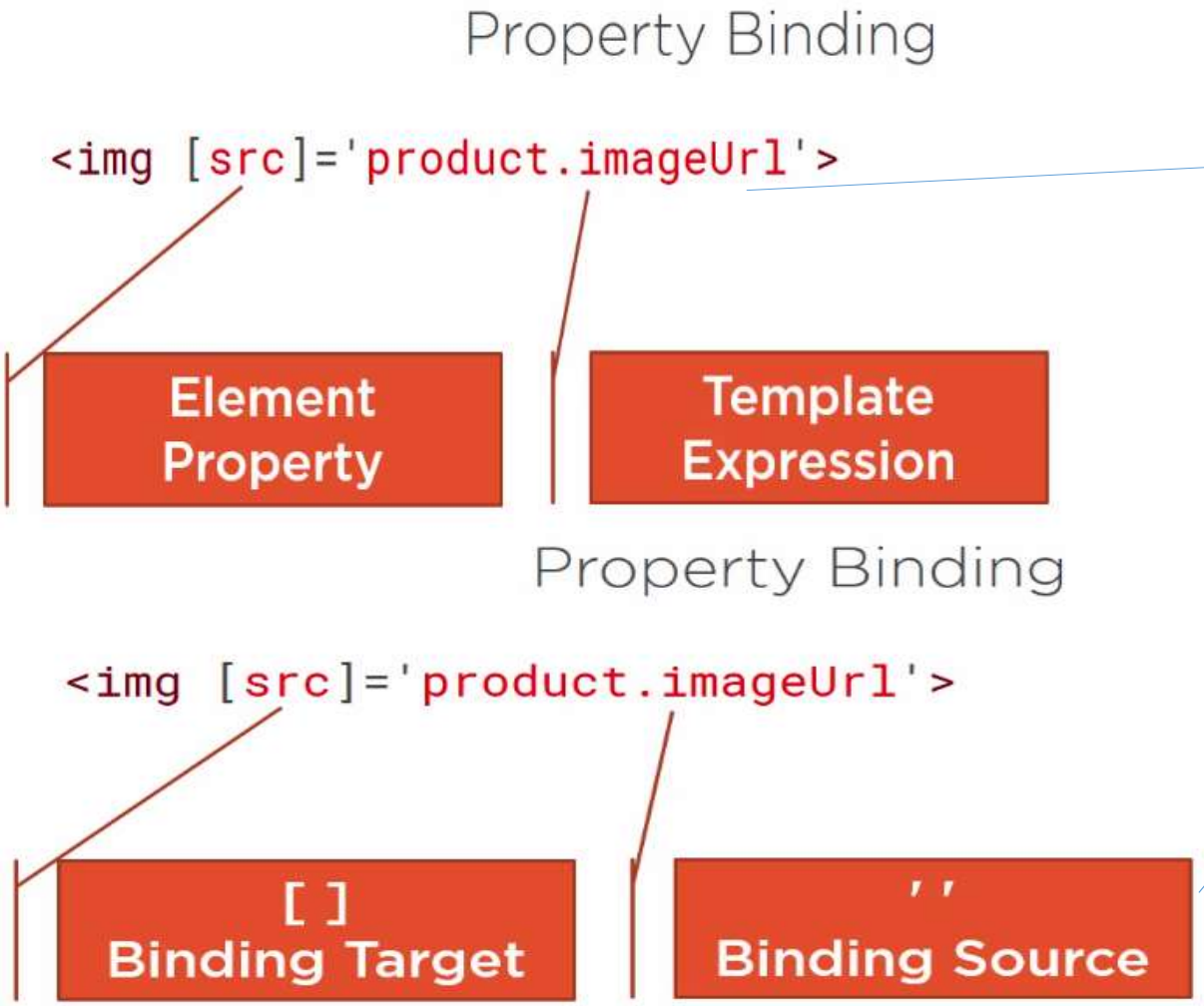
```
import {Component} from 'angular2/core';

@Component({

    selector:'media-item',
    templateUrl:'app/media-item.component.html',
    styleUrls:['app/media-item.component.css']
    })
  export class MediaItemComponent{
    name='The Redemption';
}
```

media-item.component.html

```
<h2>{{name}}</h2>
<div>Watched on 1/13/2016</div>
<div>Action</div>
<div>2016</div>
<div class="tools">
    <a class="delete">
        remove
    </a>
    <a class="details">
        Watched
    </a>
```

Angular makes all properties on the component class available to the template when it processes it

Data Bindings & Pipes



- Property Binding allows to set a property of an element to the value of the template expression
- Here we bind the source property of the image element to the imageUrl property of the product

- The binding target is always enclosed in the square brackets to the left of the =

The Binding Source is included in to the right of the = and specifies the template expression

Property Binding

```
<img [src]='product.imageUrl'>
```
→ Property Binding Syntax

```
<img src={{product.imageUrl}}>
```
→ Interpolation Binding Syntax

```
<img src='http://openclipart.org/{{product.imageUrl}}'>
```

- Property binding is preferred to interpolation binding
- However if you want to include template expression as part of a larger expression (3rd line above) then interpolation is preferred.

Event Binding

Template

```
<h1>{{pageTitle}}</h1>
<img [src]='product.imageUrl'>
```

Class

```
export class ListComponent {
 pageTitle: string = 'Product List';
 products: any[] = […];

}
```

- So far we have seen one way binding, from component class property to the element property
- But when user interacts with the UI (btn click), the component listens to the user action using Event Binding

Event Binding

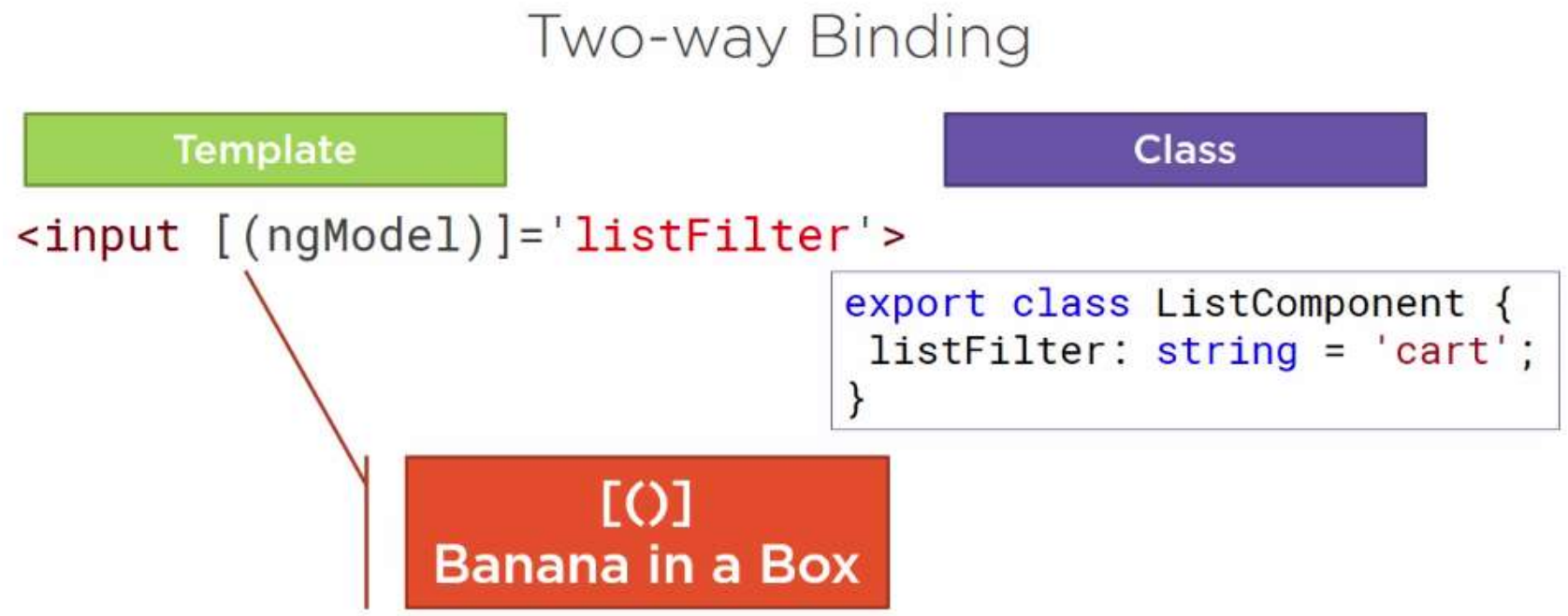
Template

```
<h1>{{pageTitle}}</h1>
<img [src]='product.imageUrl'>
<button (click)='toggleImage()'>
```

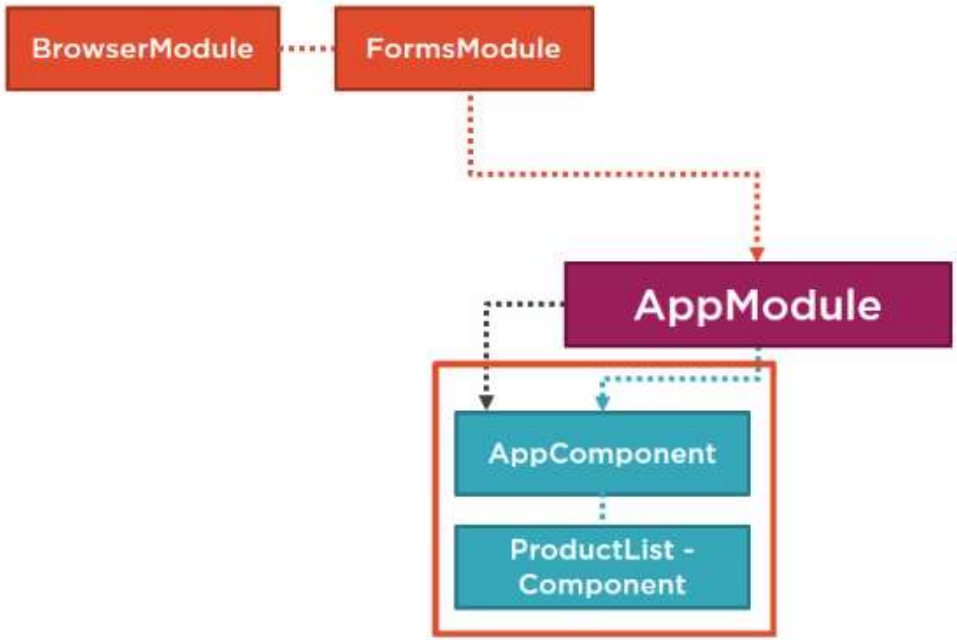
Class

```
export class ListComponent {
 pageTitle: string = 'Product List';
 products: any[] = […];
 toggleImage(): void {…}
}
```

https://developer.mozilla.org/en-US/docs/Web/Events → Link for valid DOM Events



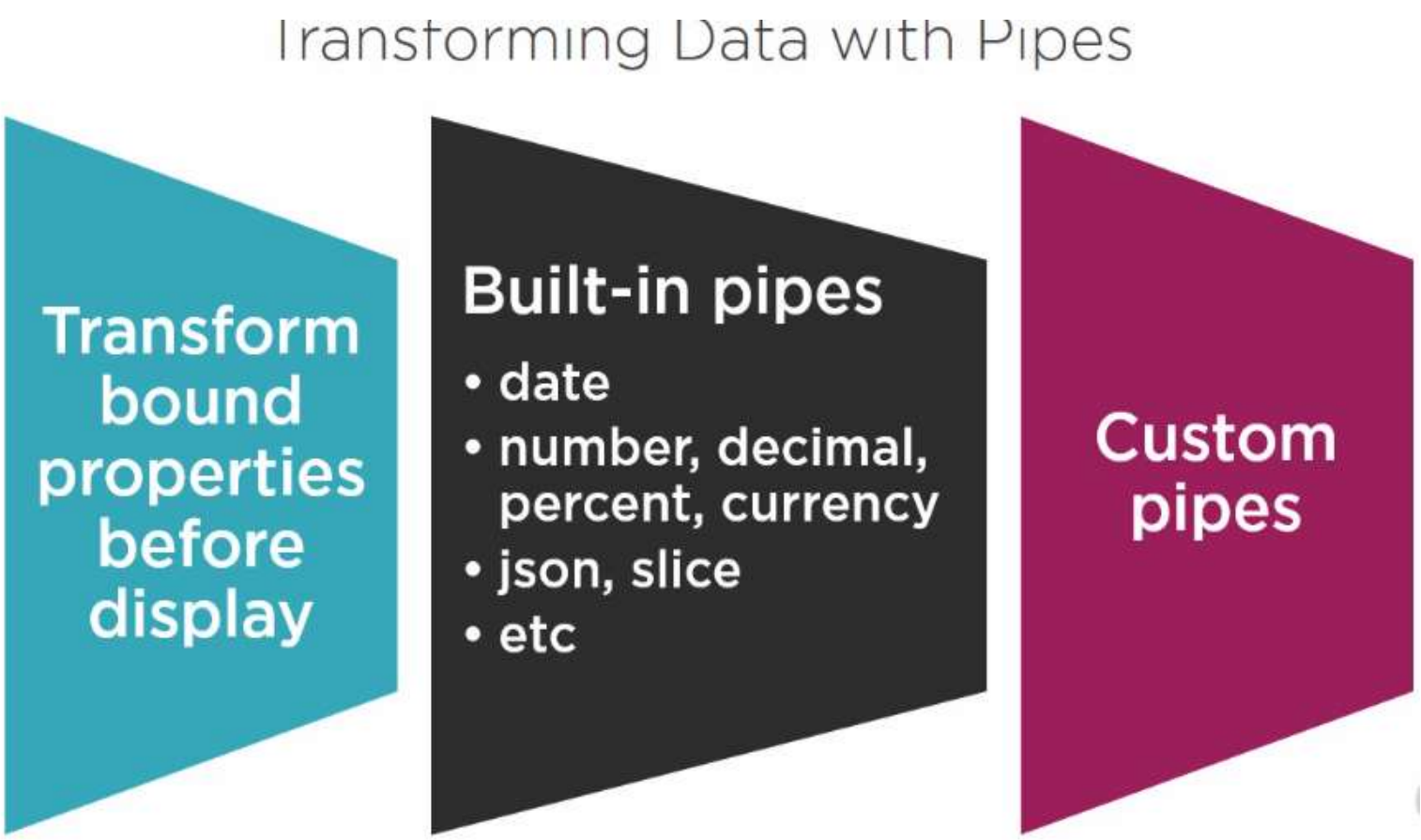
ngModel is part of FormsModule so we import that here.



Two Way Binding

AppModule should import FormsModule when using ngModel directive

Data Bindings & Pipes



Pipe Examples

```
{{ product.productCode | lowercase }}

<img [src]='product.imageUrl'
     [title]='product.productName | uppercase'>

{{ product.price | currency | lowercase }}

{{ product.price | currency:'USD':true:'1.2-2' }}
```

Pipe

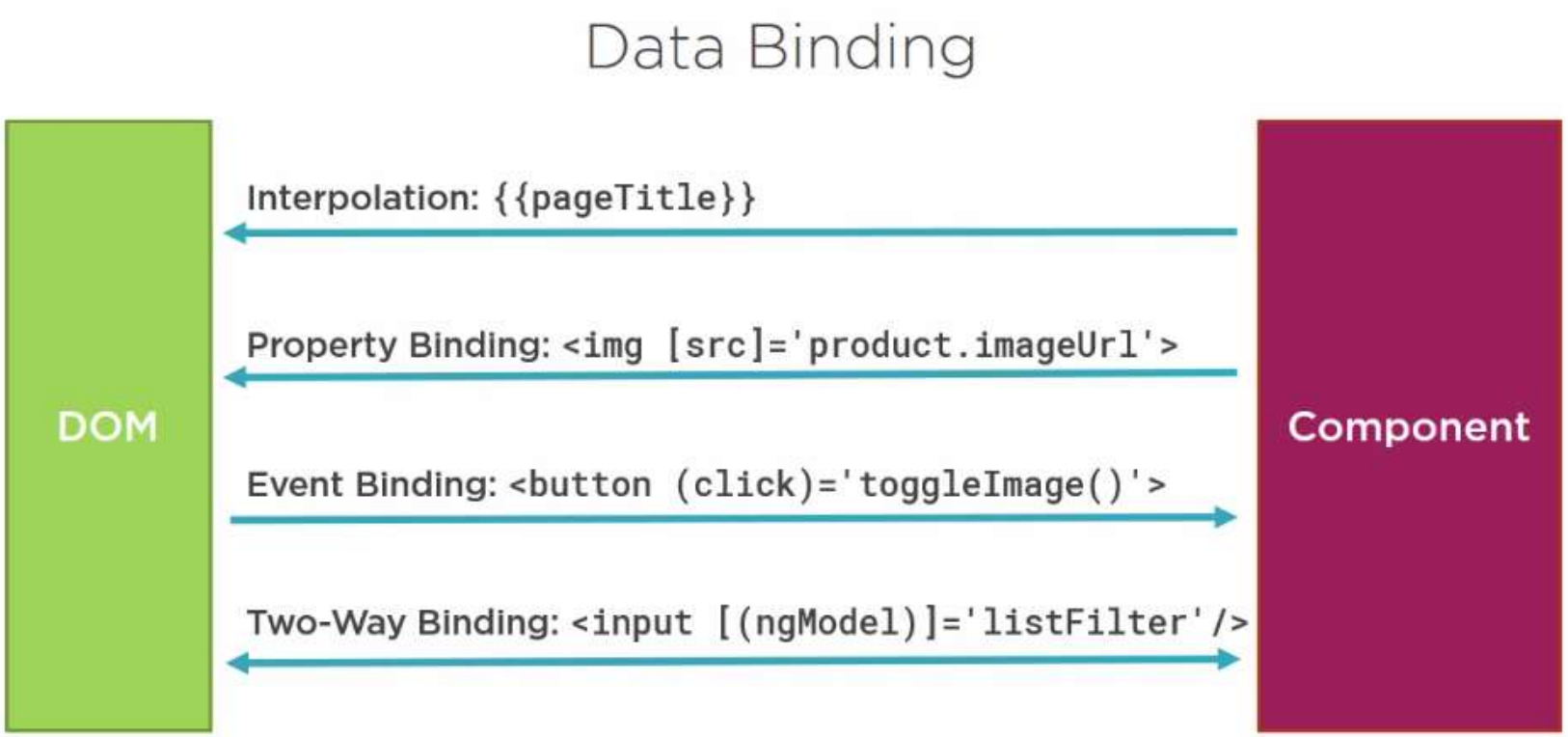
Default currency pipe is 3 letter currency code

A currency pipe has three parameters

1. The desired currency code
2. A Boolean value indicating whether to display the currency symbol
3. Digit info
    1. First digit : Minimum number of integer digit
    2. Second digit: Minimum number of fractional digit
    3. Third Digit : Maximum number of fractional digit

The currency pipe has been changed in Angular v5.
The symbolDisplay option (third parameter) is now a string instead of a boolean.
The accepted values are "code", "symbol" or "symbol-narrow"

Data Bindings & Pipes Summary



Data Bindings & Pipes Summary

Checklist: Pipes

Pipe character |

Pipe name

Pipe parameters

- Separated with colons

Example

```
- {{ product.price |
     currency:'USD':true:'1.2-2' }}
```

More on Components

Improving Our Components

- Strong typing & interfaces
- Encapsulating styles
- Lifecycle hooks
- Custom pipes
- Relative Paths with Module Id

More on Components

- One of the advantage of typescript is its strong typing
- Every property has a type
- Every method has a return type
- Every method parameter has a type
- Strong typing of Typescript helps in better syntax checking and tooling
- In some cases , however we do not have a predefined type **products any[]=[…]**
- The any[] negates the benefits of strong typing
- To specify custom types we can define a **Interface**
-

Interface

A specification identifying a related set of properties and methods.

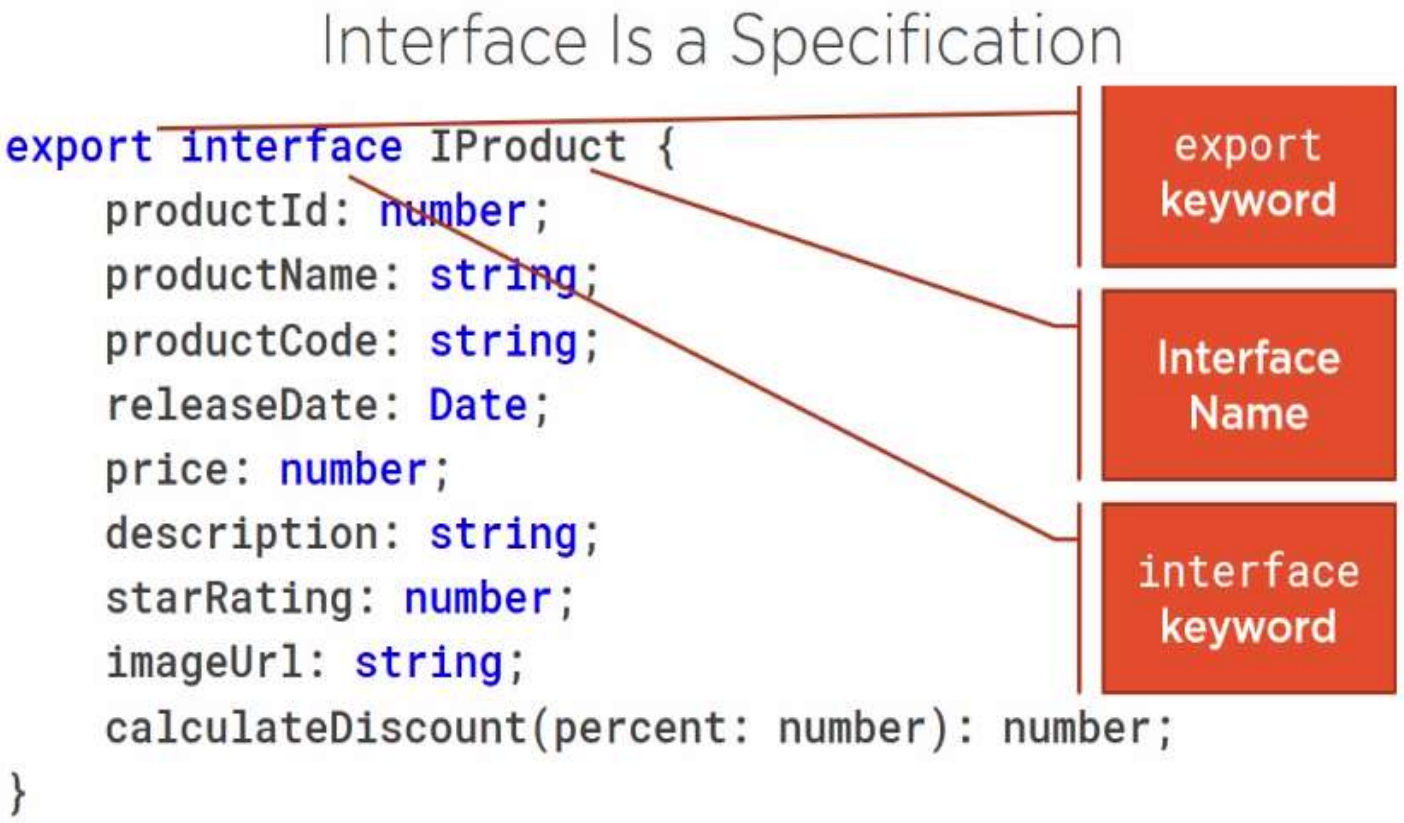
A class commits to supporting the specification by implementing the interface.

Use the interface as a data type.

Development time only!

- ES5 and ES2015 do not have support for interfaces but typescript does
- Hence after trans-pile the interfaces are not found in the resulting JS file
- Interfaces are development time only
- The purpose of interfaces is to provide strong typing and better tooling support as we build, debug and maintain the code

More on Components



- The body of the interface defines the set of properties and methods of the current business object
- For each property the interface includes the property name, a colon, and the property data type
- For each method the interface includes the method name, the list of parameters of the methods with their data types, a colon and the method return type

Using an Interface as a Data Type

```
import { IProduct } from './product';

export class ProductListComponent {
  pageTitle: string = 'Product List';
  showImage: boolean = false;
  listFilter: string = 'cart';

  products: IProduct[] = […];

  toggleImage(): void {
      this.showImage = !this.showImage;
   }
}
```

- To use the interface has the datatype we first import the interface
- After importing, use the interface name as the data type

Handling Unique Component Styles

Templates sometimes require unique styles

We can inline the styles directly into the HTML

This will make our html non readable and no code reusability

We can build an external stylesheet and link it in index.html

This will add load on index.htm because we need to add reference of each page style here.

There is a better way!

More on Components

Encapsulating Component Styles

styles

```
@Component({
    selector: 'pm-products',
    templateUrl: 'app/products/product-list.component.html',
    styles: ['thead {color: #337AB7;}']})
```

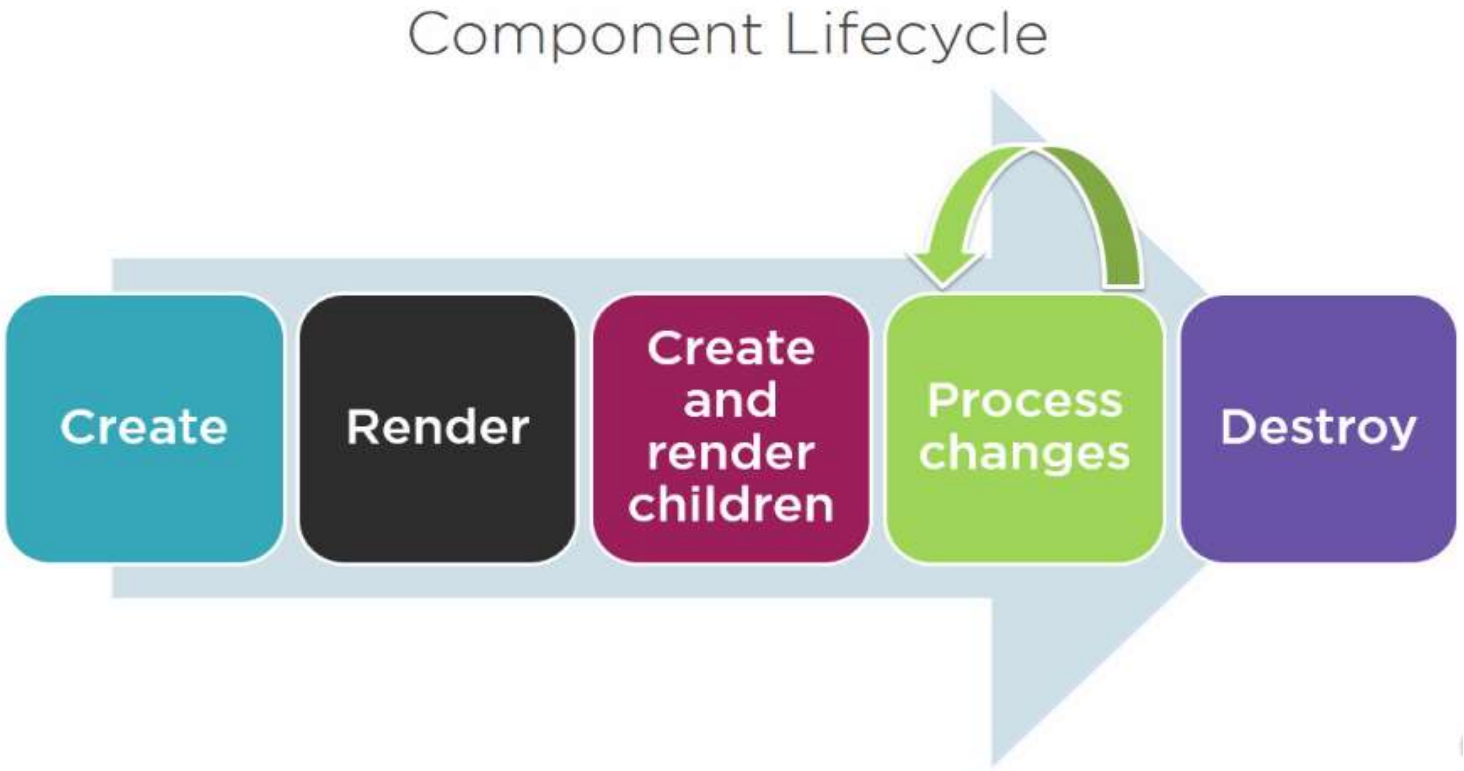
styleUrls

```
@Component({
    selector: 'pm-products',
    templateUrl: 'app/products/product-list.component.html',
    styleUrls: ['app/products/product-list.component.css']})
```

1st way: we can add styles directly to the component using the styles property.
Styles property is an array so we can add multiple styles separated by **“,”**

2nd Way:
Create an external stylesheet and identify them with the styleUrls property.
styleUrls property is an array so multiple style urls can be added seprated **by “,”.**
Encapsulating styles this way, the styles will be available only for that components template

More on Components



A component has a life cycle managed by angular
Angular creates the component, renders it.
Creates and renders its children
Processes changes when its data bound properties change
Lastly it destroys it before removing its template from the DOM
Angular provides a set of life cycle hooks that can be used to perform operations as needed

More on Components

Component Lifecycle Hooks

OnInit: Perform component initialization, retrieve data

OnChanges: Perform action after change to input properties

OnDestroy: Perform cleanup

This being a basic course we will see only three life cycle hooks

Using a Lifecycle Hook

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

  export class ProductListComponent
1                  implements OnInit {
    pageTitle: string = 'Product List';
    showImage: boolean = false;
    listFilter: string = 'cart';
    products: IProduct[] = […];

3   ngOnInit(): void {
          console.log('In OnInit');
    }
  }
```

Step 1 is optional because ES5 and ES2015 do not support interface, but writing the 1st step is good practice

Steps for life cycle hooks

1. Implement the OnInit interface. This is built in from angular/core.

Builtin interfaces do not have the letter I

2. Import the interface from the library angular/core
3. The method of the OnInit interface is prefixed with ‘ng’, so the method name is ‘ngOnInit’

Building a Custom Pipe

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

@Pipe({
    name:'productFilter'
})
export class ProductFilterPipe
                    implements PipeTransform {

  transform(value: IProduct[],
            filterBy: string): IProduct[]{
  }
}
```

Building a Custom Pipe

```
export class ProductFilterPipe
                    implements PipeTransform {

  transform(value: IProduct[],
            filterBy: string): IProduct[]{
  }
}
```

Step 1:

Implement the PipeTransform interface which has one method transform.

Code will be written in the transform method to transform the value and return it.

1st parameter of the transform method is the value we are transforming.

In our example we are transforming an array of products into a filtered array of products

Notice the usage of Iproduct interface for strong typing

The 2nd parameter of the transform method is the argument needed to perform the transformation

In our case we want to pass the filter criteria entered by the user.

The return type is also the array of products

```
@Pipe({
    name:'productFilter'
})
```

Step 2:

Next we add a pipe decorator to the class to define it as a pipe
The pipe decorator has an object property name, which is the name of the pipe

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

Step 3:

Then we have the import statement to import whatever needed.

Note: Angular provides a very consistent coding experience. See the similarity of pipes with components

Using a Custom Pipe

Template

```
<tr *ngFor ='let product of products | productFilter: listFilter'>
```

Module

```
@NgModule({
  imports: [
      BrowserModule,
      FormsModule ],
  declarations: [
      AppComponent,
      ProductListComponent,
      ProductFilterPipe ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

We define the pipe in the declarations array of the AppModule

More on Components

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

The above is the mdn link for more information on array filter function

More on Components

Relative Paths and Module Id

product-list.component.ts

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

@Component({
    selector: 'pm-products',
    templateUrl: 'app/products/product-list.component.html',
    styleUrls: ['app/products/product-list.component.css']
})
export class ProductListComponent {
 pageTitle: string = 'Product List';
 ...
}
```

Absolute path of .html and .css.
This style is error prone
This way is harder to to reuse in other components
Tools like ahead of time compiler requires relative path and not an absolute path.

Relative Paths and Module Id

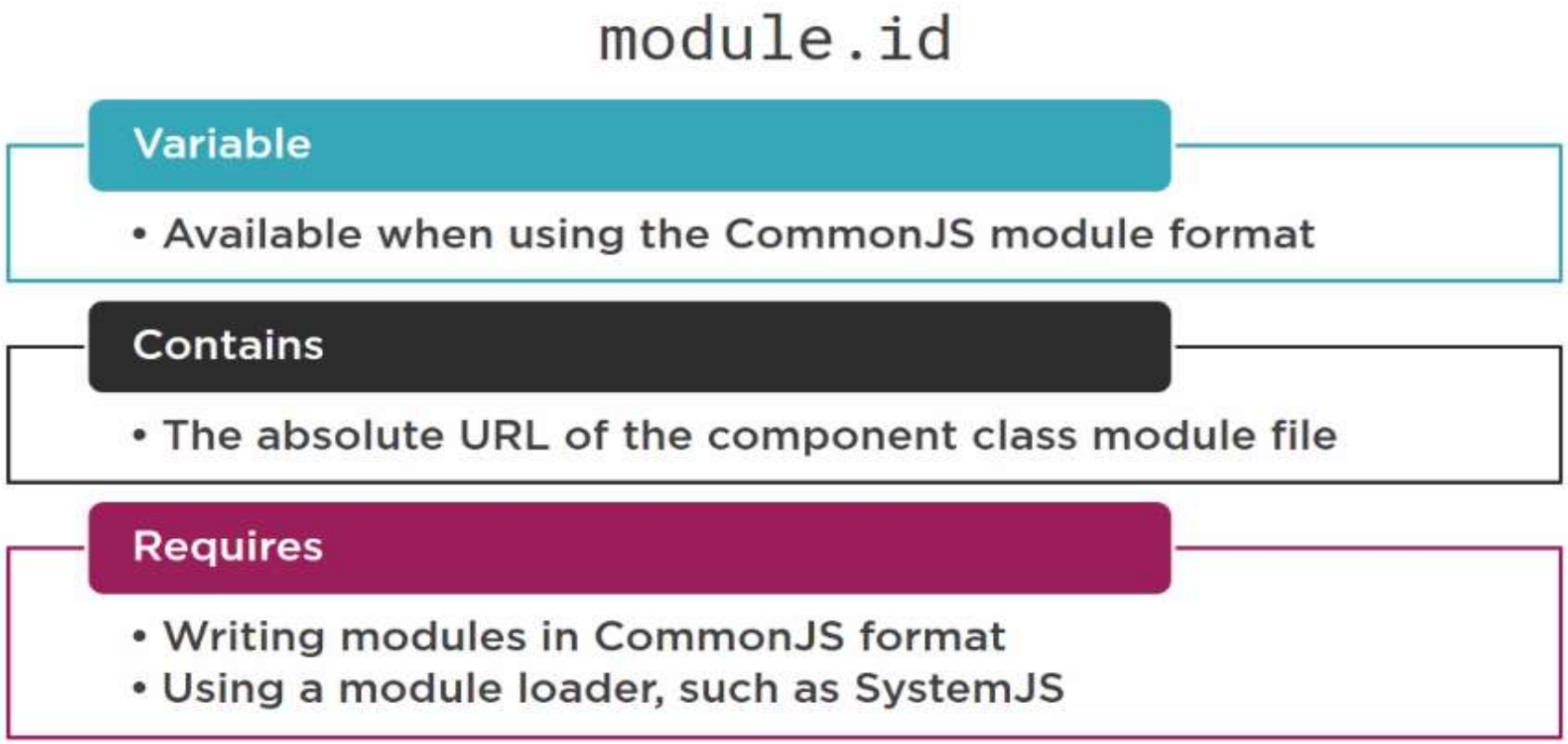
product-list.component.ts

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

@Component({
    selector: 'pm-products',
    moduleId: module.id,
    templateUrl: 'product-list.component.html',
    styleUrls: ['product-list.component.css']
})
export class ProductListComponent {
 pageTitle: string = 'Product List';
 ...
}
```

To do away with the absolute path approach, we can use the relative path approach.

This is done by setting the moduleId property in the Component

The ModuleId is a semi-global variable of commonJS format



More on Components Summary

Checklist: Interfaces

Defines custom types

Creating interfaces:

- **`interface`** keyword
- export it

Implementing interfaces:

- **`implements`** keyword & interface name
- Write code for each property & method

Checklist: Encapsulating Styles

`styles` property

- Specify an array of style strings

`styleUrls` property

- Specify an array of stylesheet paths

Checklist: Using Lifecycle Hooks

Import the lifecycle hook interface

Implement the lifecycle hook interface

Write code for the hook method

More on Components Summary

Checklist: Building a Custom Pipe

Import Pipe and PipeTransform

Create a class that implements PipeTransform

- **export** the class

Write code for the Transform method

Decorate the class with the Pipe decorator

Checklist: Relative Paths with Module Id

Set the moduleId property of the component decorator to module.id

Change the Url to a component-relative path:

- templateUrl
- styleUrls

Checklist: Using a Custom Pipe

Import the custom pipe

Add the pipe to the declarations array of an Angular module

Any template associated with a component that is also declared in that Angular module can use that pipe

Use the Pipe in the template

- Pipe character
- Pipe name
- Pipe arguments (separated with colons)

Nesting Components

Building a Nested Component

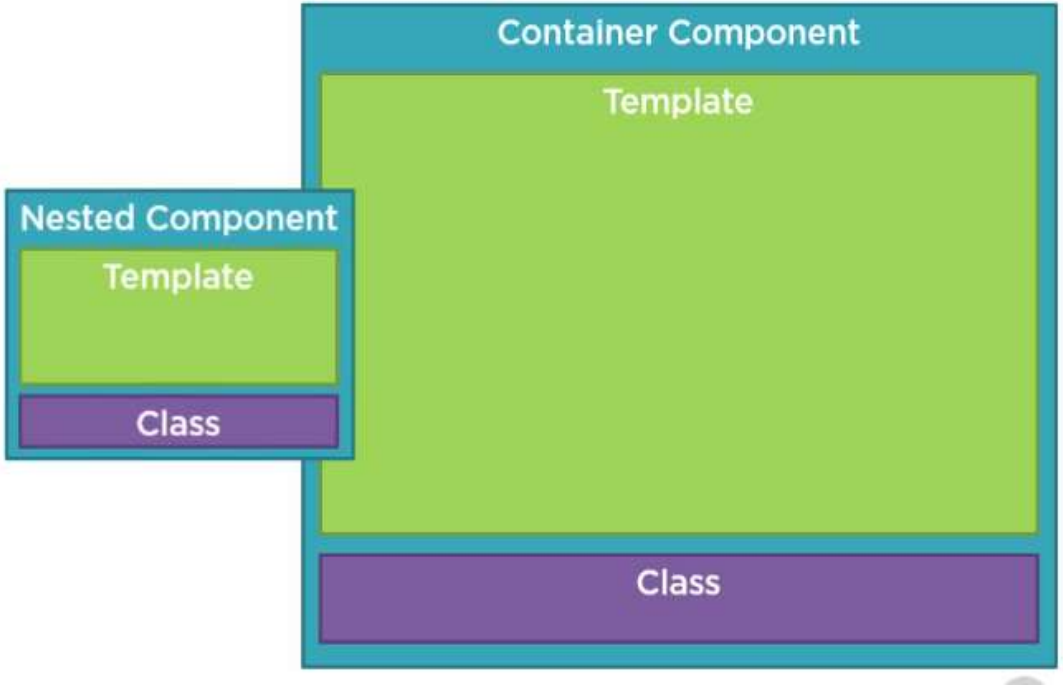
Using a Nested Component

Passing Data to a Nested Component Using @Input

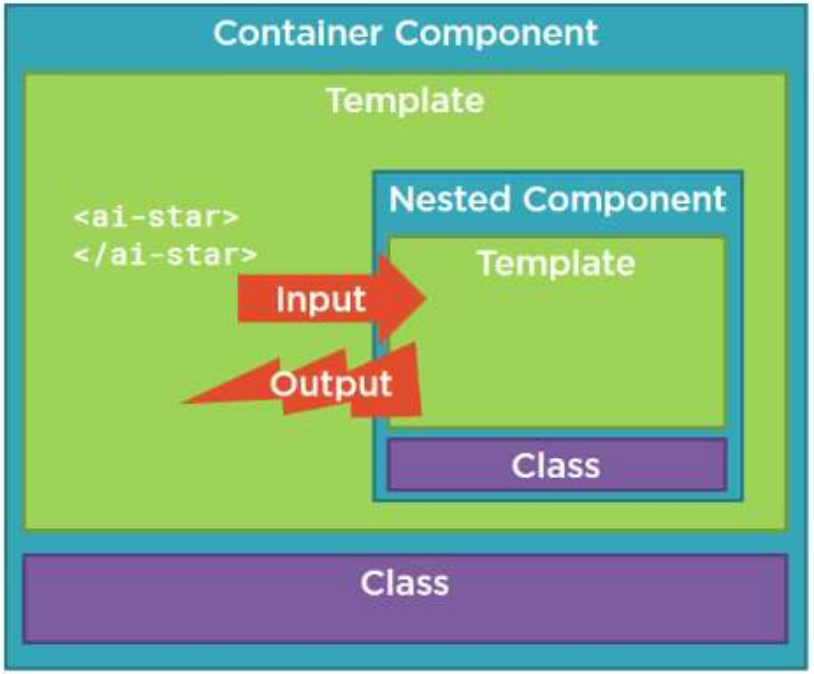
Raising an Event from a Nested Component Using @Output

Building Nested Components

Building a Nested Component



Building a Nested Component



Building Nested Components

- It is component within a Component
- The outer Component is called as the Container or Parent
- The inner Component is referred to as Child or Nested Component
- The nested or child component needs to communicate with the Parent Component
- The nested/Child component receives information from its Container/Parent using input properties
- The nested/Child often outputs information back to its container by raising events

- In the PM application We will make the 5 star rating column as nested Component
- For the star component to display the correct number of stars the Container/Parent must provide the rating number to our nested component as input
-

Nested Components

Using a Nested Component as a Directive

BEFORE

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL: 'product-list.component.html'
})
export class ProductListComponent { }
```

product-list.component.html

```
<td>
   {{ product.starRating | number }}
</td>
```

star.component.ts

```
@Component({
  selector: 'ai-star',
  templateURL: 'star.component.html'
})
export class StarComponent {
  rating: number;
  starWidth: number;
}
```

Nested Components

Using a Nested Component as a Directive

AFTER

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL: 'product-list.component.html'
})
export class ProductListComponent { }
```

product-list.component.html

```
<td>
   <ai-star></ai-star>
</td>
```

star.component.ts

```
@Component({
  selector: 'ai-star',
  templateURL: 'star.component.html'
})
export class StarComponent {
  rating: number;
  starWidth: number;
}
```

Nested Components

Telling Angular About Our Component

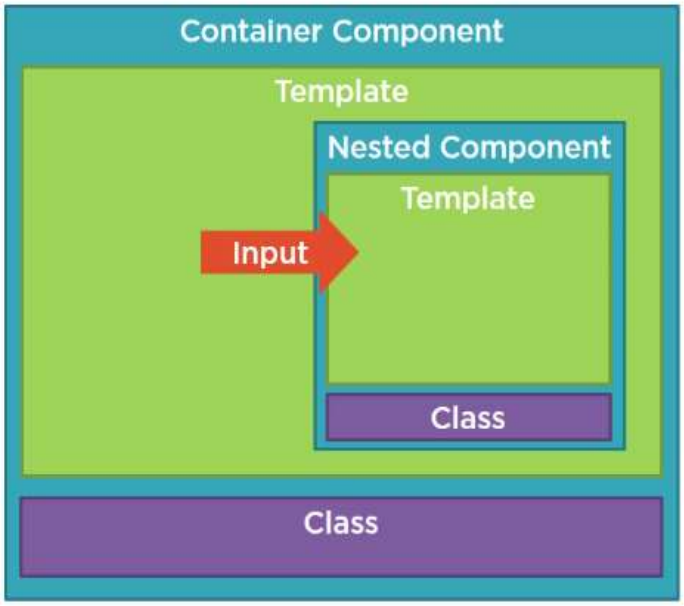
app.module.ts

```
...
import { StarComponent } from './shared/star.component';

@NgModule({
  imports: [
      BrowserModule,
      FormsModule ],
  declarations: [
      AppComponent,
      ProductListComponent,
      ProductFilterPipe,
      StarComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Importing StarComponent

Nested Components

Passing Data to a Nested Component (@Input)



- If nested component wants to receive input from the container it should use the property called @Input decorator
- The @Input decorator is used to decorate any property in the nested components class.
- In our example we want the rating number passed into the nested component.
- Hence that property is marked with an @Input decorator
- The Container/parent component then passes data to the nested component by setting this property with property binding

Passing Data to a Nested Component (@Input)

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL: 'product-list.component.html'
})
export class ProductListComponent { }
```

star.component.ts

```
@Component({
  selector: 'ai-star',
  templateURL: 'star.component.html'
})
export class StarComponent {
  @Input() rating: number;
  starWidth: number;
}
```

product-list.component.html

```
<td>
  <ai-star [rating]='product.starRating'>
  </ai-star>
</td>
```

rating can be used as a property binding syntax because of the @Input decorator

Nested Components

Raising an Event (@Output)

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL: 'product-list.component.html'
})
export class ProductListComponent { }
```

star.component.ts

```
@Component({
  selector: 'ai-star',
  templateURL: 'star.component.html'
})
export class StarComponent {
 @Input() rating: number;
 starWidth: number;
 @Output() notify: EventEmitter<string> =
              new EventEmitter<string>();
}
```

product-list.component.html

```
<td>
  <ai-star [rating]='product.starRating'>
  </ai-star>
</td>
```

TypeScript supports generics.
EventEmitter<string> = new EventEmitter()

- If the nested component wants to send information back to its container, it can raise an event.
- The nested component exposes an event that it can use to pass output to its container using the **@Output decorator**
- We can use the @Output decorator to decorate any property of the nested component class
- The condition is the property type must be an event.
- The only way a nested component can pass data to its component is through an event
- In angular an event is defined by EventEmitter Object.
- Create new instance of the EventEmitter

Nested Components

- When creating the EventEmitter the generic argument can be any valid datatype.
- In our application it is “string”
- The generic object decides the type of payload
- In our application we define a notify event with a string payload.
- When the user clicks on the stars , the star component receives that event
- We use event binding to call the Star Components onClick()
- To call the star components on the click method in the onclick method we use the notify event property and call its emit method to raise the notify event.

Nested Components

Checklist: Nested Component

Input decorator

- Attached to a property of any type
- Prefix with @; Suffix with ()

Output decorator

- Attached to a property declared as an EventEmitter
- Use the generic argument to define the event payload type
- Use the new keyword to create an instance of the EventEmitter
- Prefix with @; Suffix with ()

Checklist: Container Component

Use the directive

- Directive name -> nested component's selector

Use property binding to pass data to the nested component

Use event binding to respond to events from the nested component

- Use $event to access the event payload passed from the nested component

Service

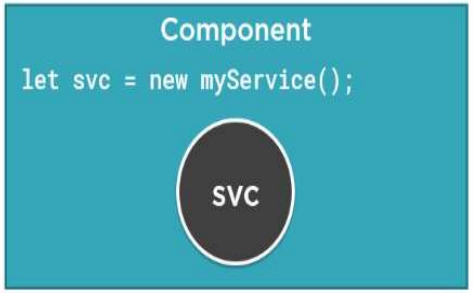
A class with a focused purpose.

Used for features that:

- Are independent from any particular component
- Provide shared data or logic across components
- Encapsulate external interactions

- By shifting the responsibility from component to service , the code is easier to test, debug and reuse.

Services and dependency injection

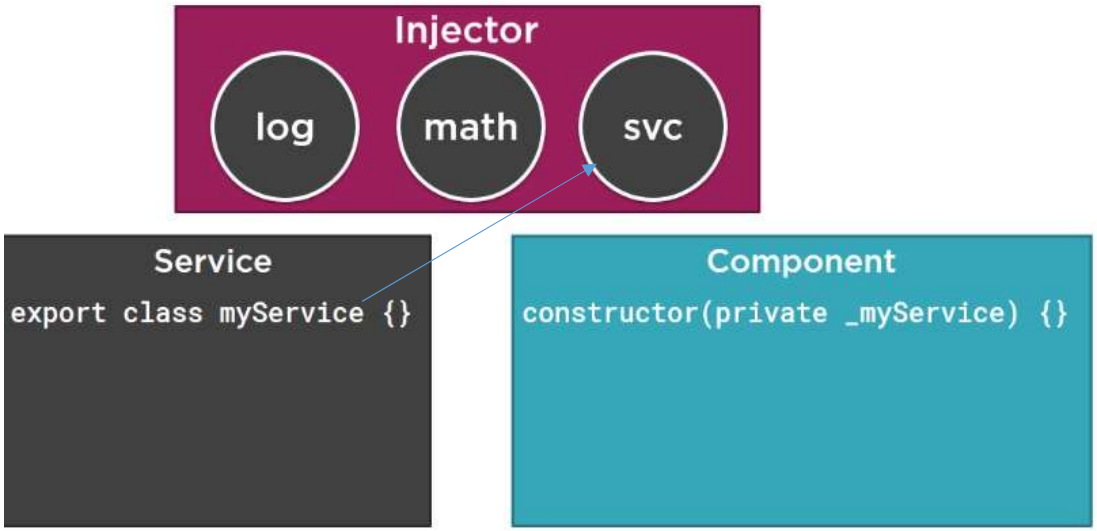


There are 2 ways a component can work with the service

1. By creating instance of the service class and use it

   The instance is local to the component, so data cannot be shared , testing would be difficult,

Services and dependency injection



2nd Way to use the services
Register the service with angular
Angular creates a single instance of the service class
Angular provides a built in injector.
The service is registered with the angular injector.
Injector is a container to created services
If the component class needs the service, it defines the service as dependency
The angular injector then injects the service class instance when the component class is instantiated
This process is called dependency injection

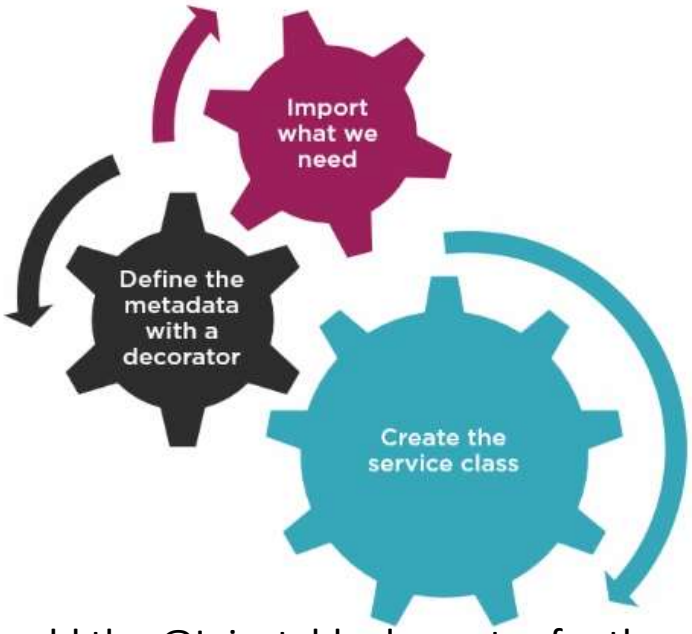
Dependency Injection

A coding pattern in which a class receives the instances of objects it needs (called dependencies) from an external source rather than creating them itself.

In Angular this external source is an injector

Services and dependency injection

Building a Service



We add the @Injectable decorator for the service metadata
This decorator is required only if the service itself has injected dependency.
It is good practice to use the @Injectable decorator even if the service has no dependency

Building Service

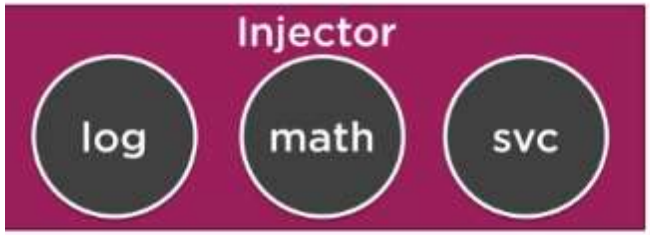
2

product.service.ts

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

@Injectable()
export class ProductService {

  getProducts(): IProduct[] {
  }

}
```

1

Registering a Service



Register a provider

- Code that can create or return a service
- Typically the service class itself

Define in component OR Angular module metadata

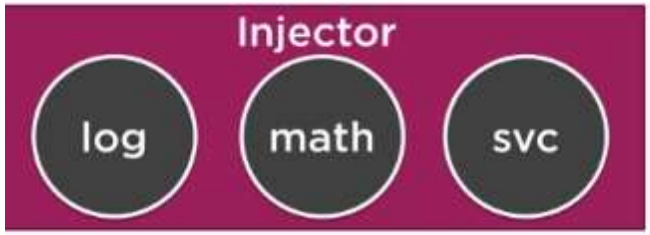
Registered in component:

- Injectable to component AND its children

Registered in Angular module:

- Injectable everywhere in the application

Registering a Service



Register a provider

- Code that can create or return a service
- Typically the service class itself

Define in component OR Angular module metadata

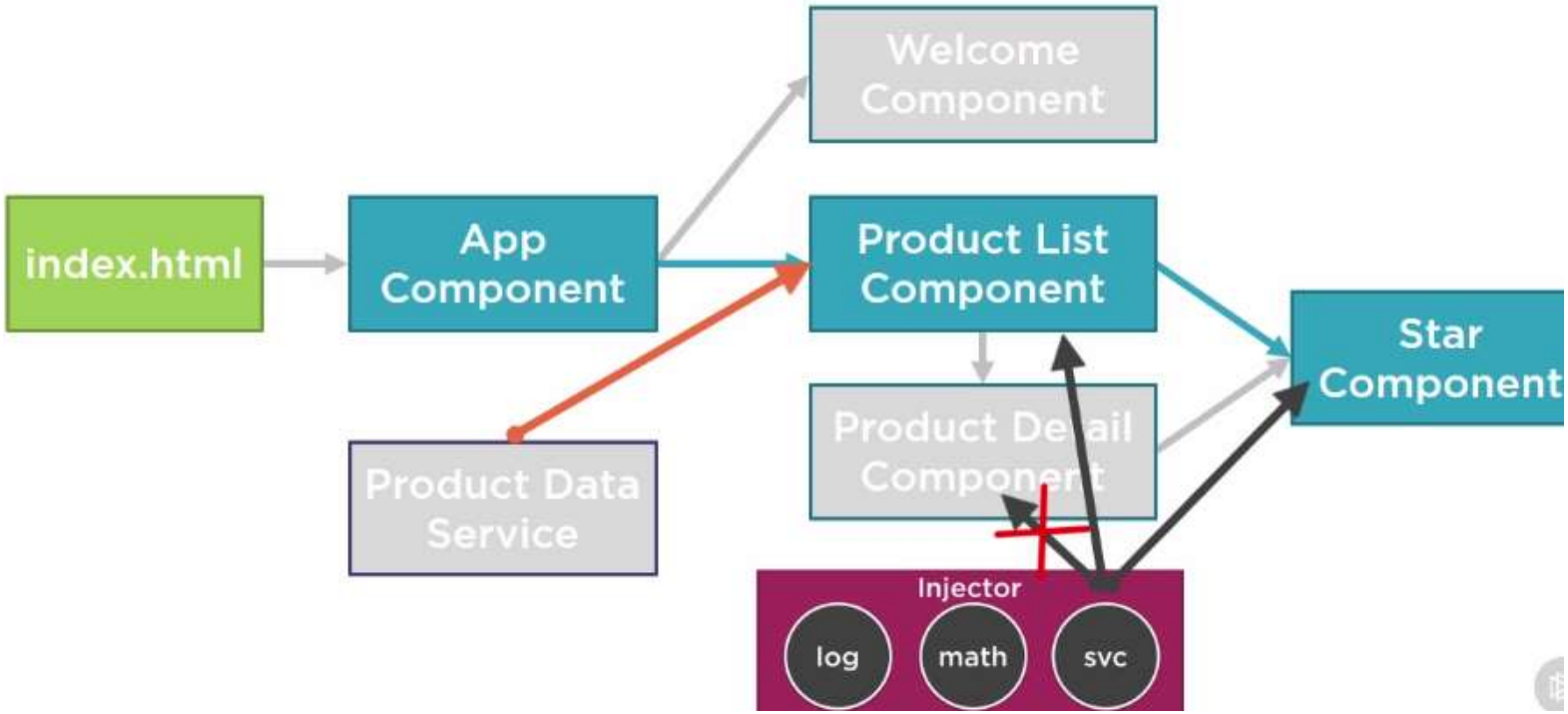
Registered in component:

- Injectable to component AND its children

Registered in Angular module:

- Injectable everywhere in the application

Services And Dependency Injection



Registering a Provider

app.component.ts

```
...
import { ProductService } from './products/product.service';

@Component({
  selector: 'pm-app',
  template: `
    <div><h1>{{pageTitle}}</h1>
      <pm-products></pm-products>
    </div>
    `,
  providers: [ProductService]
})
export class AppComponent { }
```

Injecting the Service

product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateUrl: 'product-list.component.html'
})
export class ProductListComponent {

 constructor() {
 }

}
```

1

product-list.component.ts

```
import { ProductService } from './products/product.service';

@Component({
  selector: 'pm-products',
  templateUrl: 'product-list.component.html'
})
export class ProductListComponent {

 constructor(private _productService: ProductService) {
 }

}
```

3

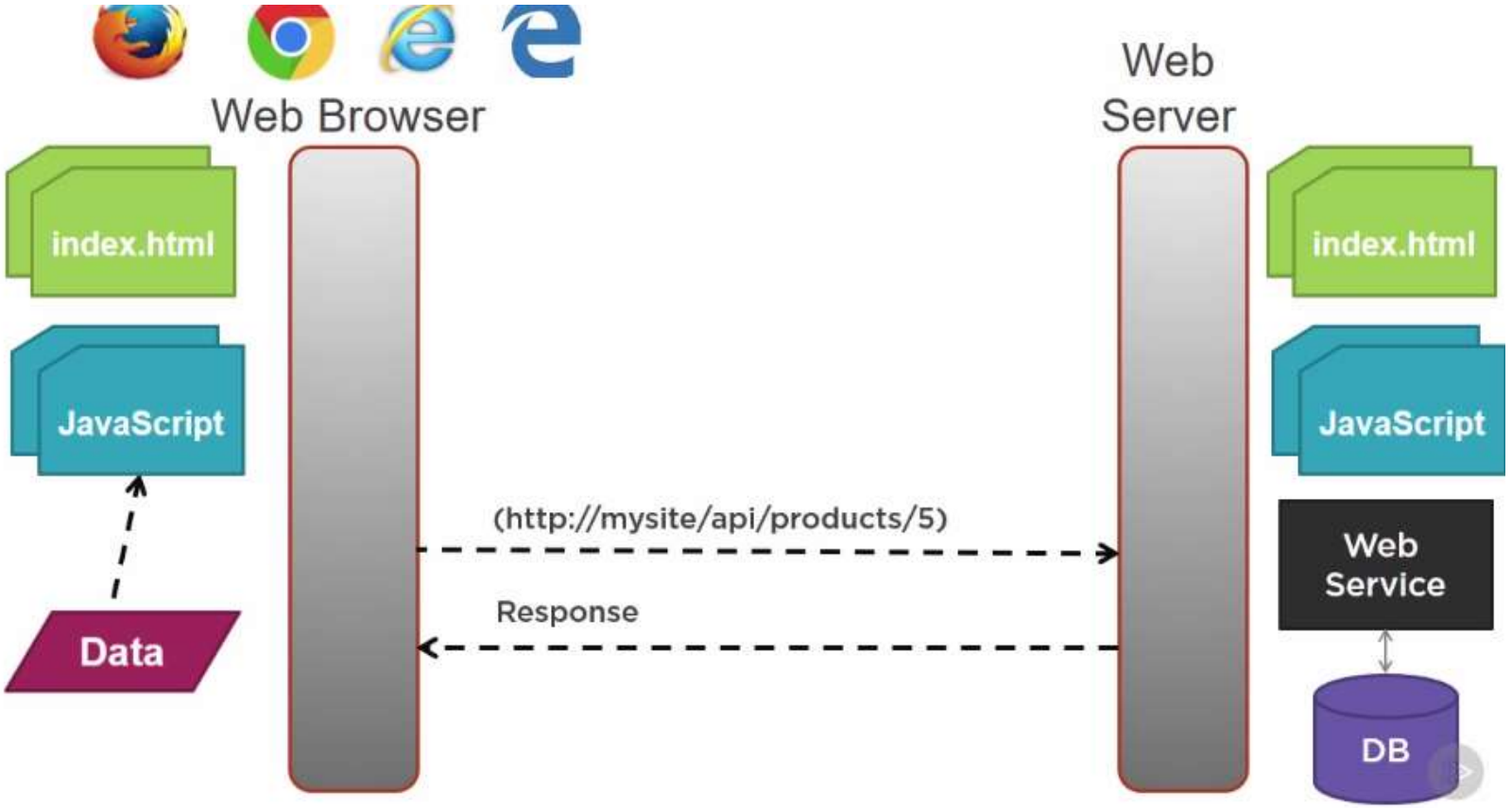
Short hand method

product-list.component.ts

```
import { ProductService } from './products/product.service';

@Component({
  selector: 'pm-products',
  templateUrl: 'product-list.component.html'
})
export class ProductListComponent {
 private _productService;
 constructor(productService: ProductService) {
   _productService = productService;
 }

}
```

2

Retrieving Data Using Http



Observables and Reactive Extensions

Sending an Http Request

Exception Handling

Subscribing to an Observable

Observables and Reactive Extensions

Help manage asynchronous data

Treat events as a collection

- An array whose items arrive asynchronously over time

Are a proposed feature for ES 2016

Use Reactive Extensions (RxJS)

Are used within Angular

Observable Operators

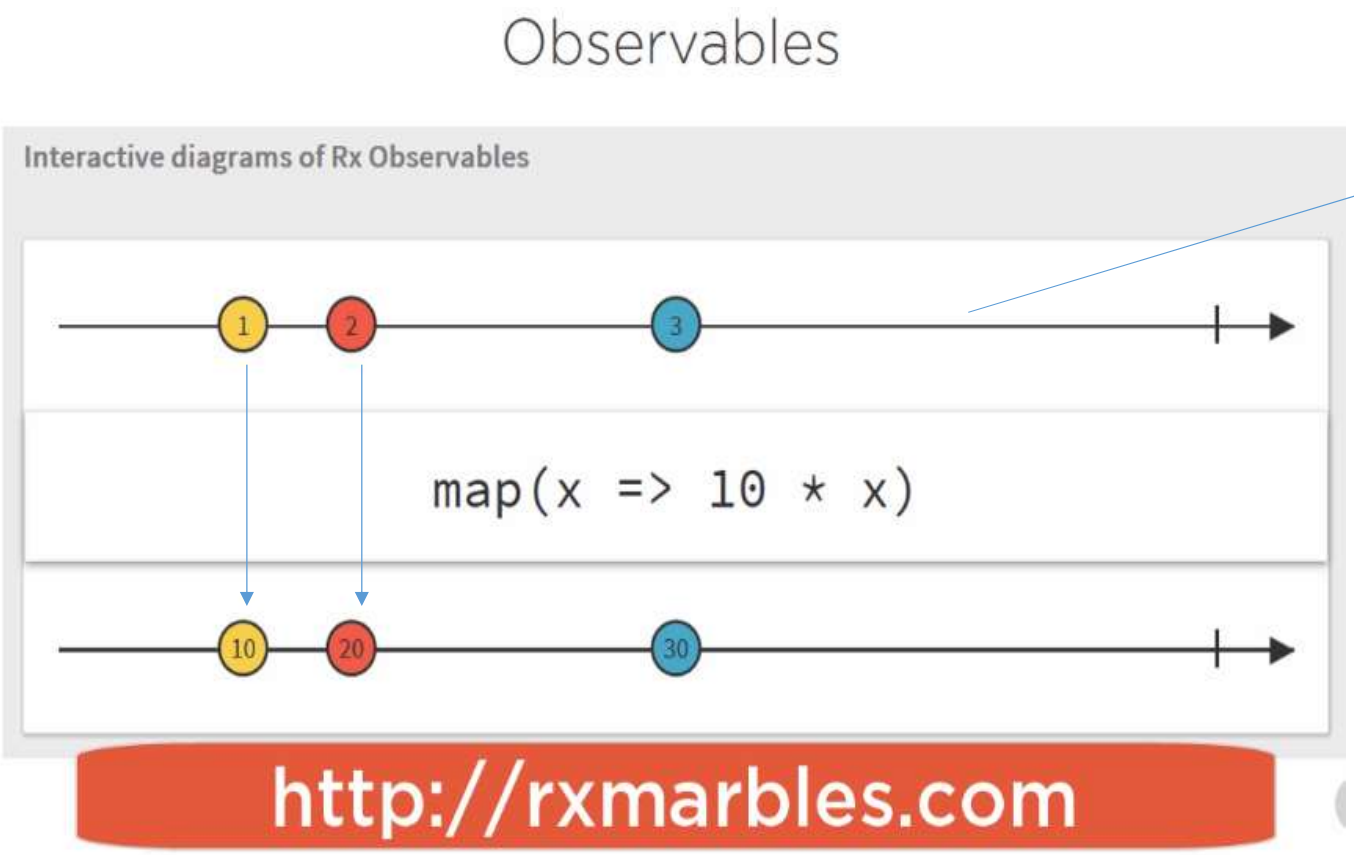
Methods on observables that compose new observables

Transform the source observable in some way

Process each value as it is emitted

Examples: map, filter, take, merge, ...

Retrieving data using HTTP



Data sequences are a stream of events like a response from backend service, system notifications or user input etc.
Here there are there are 3 elements

Reactive extensions represent a data sequence as an observable sequence usually called as OBSERVABLE
A method in the code can subscribe to an observable to receive async notifications as new data arrives
The method can then react as data is pushed to it.
The method is notified when there is no more data or when there is error

Observables support operators such as the map operator in previous slide
The map operator will transform the incoming data
The argument to the map operator is an arrow function
This arrow function takes each data and transforms it to 10 times its value (for e.g.)
The depiction in the previous slide is called marble diagram

Retrieving data using HTTP

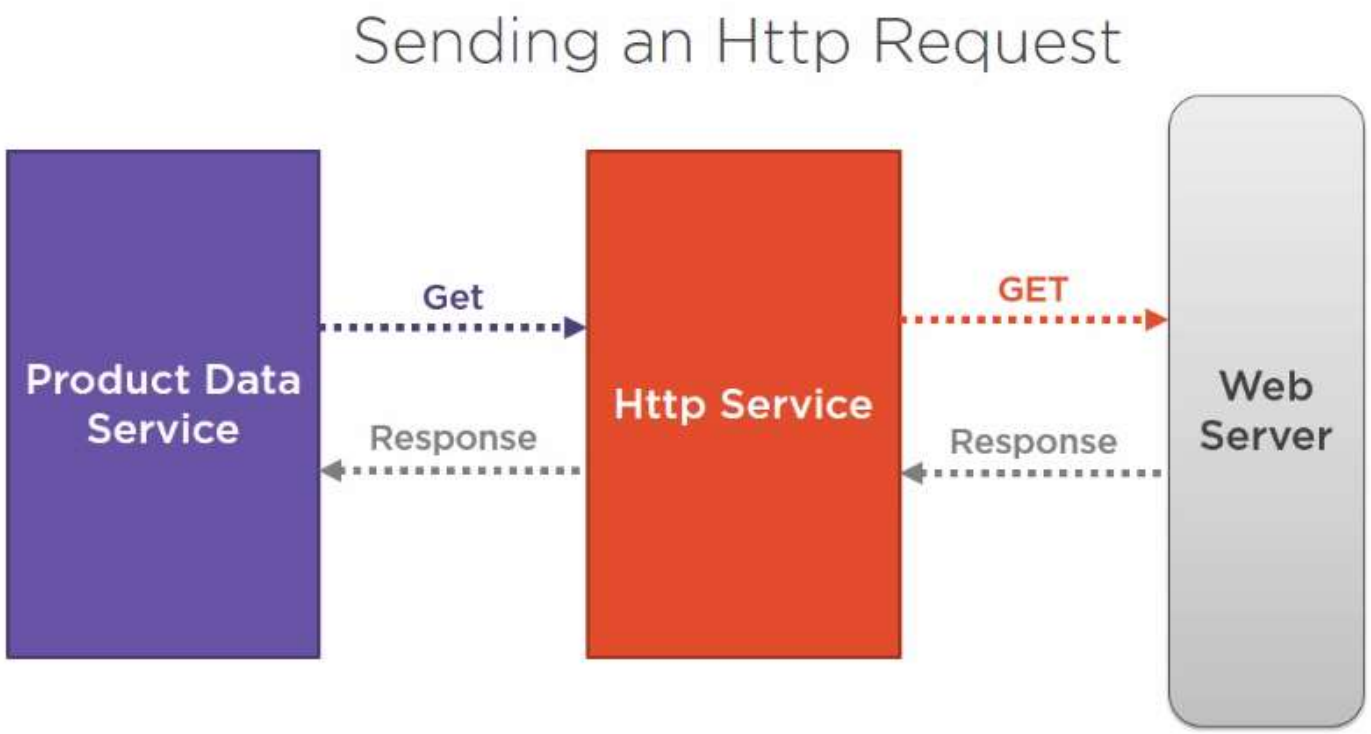
Promise vs Observable

Promise	Observable
Provides a single future value	Emits multiple values over time
Not lazy	Lazy
Not cancellable	Cancellable
	Supports map, filter, reduce and similar operators

Will not emit values until they are subscribed to

Can be cancelled by unsubscribing

Retrieving data using HTTP



Retrieving Data Using HTTP

Sending an Http Request

product.service.ts

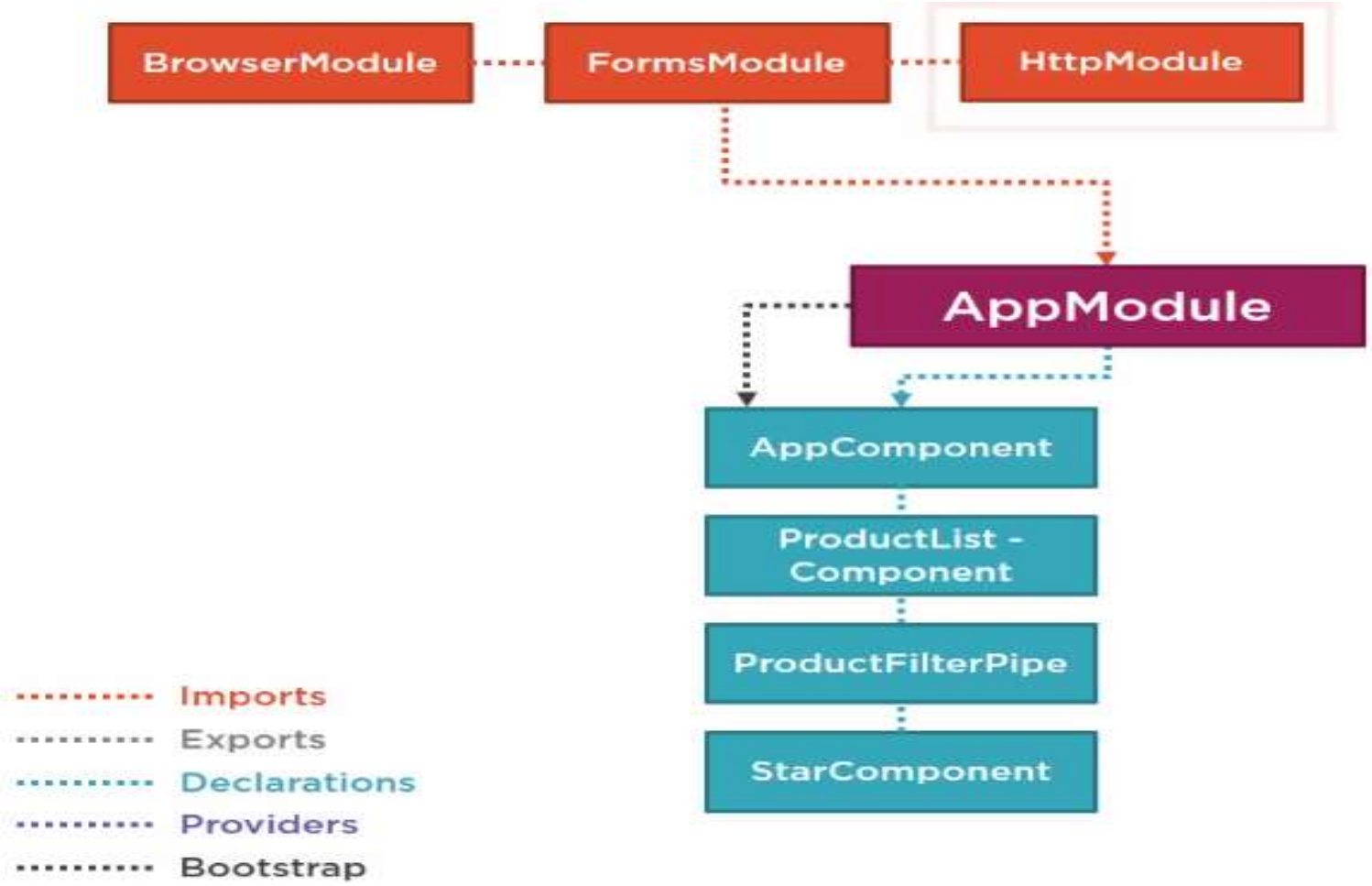
```
...
import { Http } from '@angular/http';



@Injectable()
export class ProductService {
  private _productUrl = 'www.myWebService.com/api/produc

  constructor(private _http: Http) { }

  getProducts() {
   return this._http.get(this._productUrl);

  }
}
```

Registering the Http Service Provider

app.module.ts

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

@NgModule({
  imports: [
      BrowserModule,
      FormsModule,
      HttpModule ],
  declarations: [
      AppComponent,
      ProductListComponent,
      ProductFilterPipe,
      StarComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Retrieving Data Using HTTP



Exception Handling

product.service.ts

```
...
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/catch';
...

  getProducts(): Observable<IProduct[]> {
   return this._http.get(this._productUrl)
                .map((response: Response) => <IProduct[]>response.json())
                .do(data => console.log('All: ' + JSON.stringify(data)))
                .catch(this.handleError);
  }

  private handleError(error: Response) {
  }
```

Subscribing to an Observable

```
x.then(valueFn, errorFn)                    //Promise
x.subscribe(valueFn, errorFn)               //Observable
x.subscribe(valueFn, errorFn, completeFn)   //Observable
let sub = x.subscribe(valueFn, errorFn, completeFn)
```

product-list.component.ts

```
ngOnInit(): void {
    this._productService.getProducts()
            .subscribe(
              products => this.products = products,
              error => this.errorMessage = <any>error);
}
```

Retrieving Data Using HTTP

Http Checklist: Service

Import what we need

Define a dependency for the http client service

- Use a constructor parameter

Create a method for each http request

Call the desired http method, such as get

- Pass in the Url

Map the Http response to a JSON object

Add error handling

Http Checklist: Subscribing

Call the subscribe method of the returned observable

Provide a function to handle an emitted item

- Normally assigns a property to the returned JSON object

Provide an error function to handle any returned errors

How Routing Works

```
▼<pm-app>
  ▼<div>
    ▶<nav class="navbar navbar-default">…</nav>
    ▼<div class="container">
      ::before
      <router-outlet></router-outlet>
      ▼<ng-component _nghost-jfk-3>
        ▼<div _ngcontent-jfk-3 class="panel panel-primary">
          <div _ngcontent-jfk-3 class="panel-heading">
              Product List
            </div>
          ▼<div _ngcontent-jfk-3 class="panel-body">
            ::before
            ▶<div _ngcontent-jfk-3 class="row">…</div>
            ▼<div _ngcontent-jfk-3 class="table-responsive">
              ▼<table _ngcontent-jfk-3 class="table">
                ▶<thead _ngcontent-jfk-3>…</thead>
                ▶<tbody _ngcontent-jfk-3>…</tbody>
                </table>
              </div>
              ::after
            </div>
          </div>
        </ng-component>
        ::after
      </div>
    </div>
  </pm-app>
```

Configure a route for each component

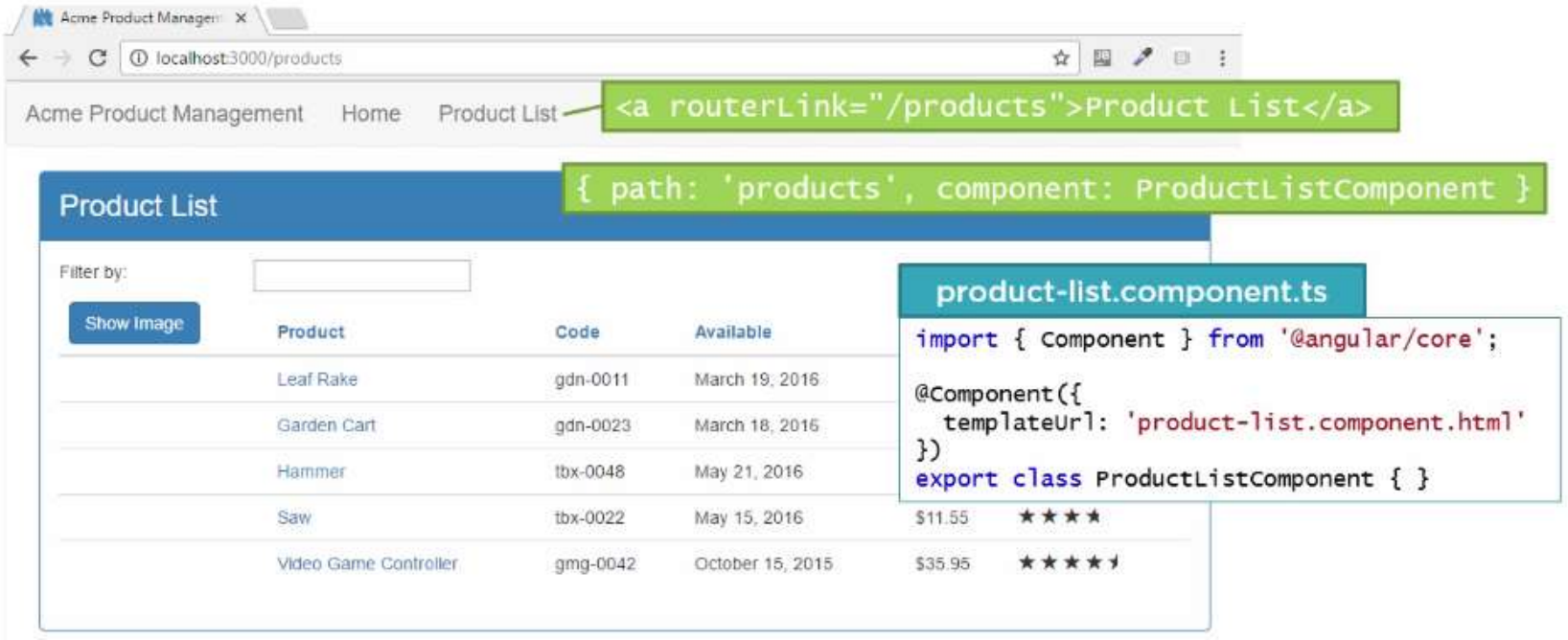
Define options/actions

Tie a route to each option/action

Activate the route based on user action

Activating a route displays the component's view

How Routing Works



Configuring Routes

www.myWebService.com/api/products

```
[
 { path: 'products', component: ProductListComponent },
 { path: 'product/:id', component: ProductDetailComponent },
 { path: 'welcome', component: WelcomeComponent },
 { path: '', redirectTo: 'welcome', pathMatch: 'full' },
 { path: '**', component: PageNotFoundComponent }
]
```

Configuring Routes

app.module.ts

```
...
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule
  ],
  declarations: [
    ...
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Registers the router service
Declares the router directives
Exposes configured routes

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([], { useHash: true })
  ],
```

To get the # in the url.

Default is false

Array of routes passed here

Configuring Routes

```
[
 { path: 'products', component: ProductListComponent },
 { path: 'product/:id', component: ProductDetailComponent },
 { path: 'welcome', component: WelcomeComponent },
 { path: '', redirectTo: 'welcome', pathMatch: 'full' },
 { path: '**', component: PageNotFoundComponent }
]
```

Usually used to show errors like page not found

Most specific routes should be placed before the least specific.
The order Matters

Defines a default route.
The redirect here translates the empty route to the desired default path segment
The Redirect route requires a path match property to tell the router how to match the url path segment to the path of the route
We want the default route when the client side portion of the route is empty

Navigating the Application Routes

- Menu option, link, image or button that activates a route
- ✓ Typing the Url in the address bar / bookmark
- ✓ The browser's forward or back buttons

Tying Routes to Actions

app.component.ts

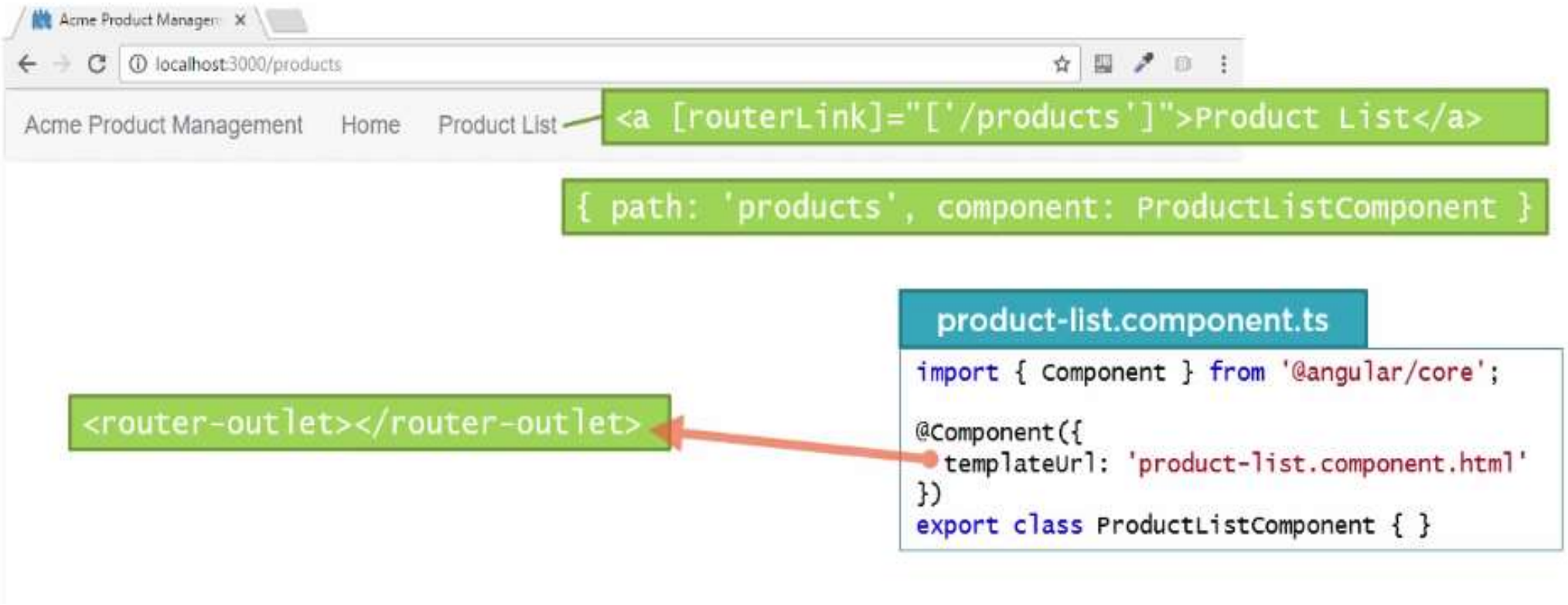
```
...

@Component({
    selector: 'pm-app',
    template: `
     <ul class='nav navbar-nav'>
       <li><a [routerLink]="['/welcome']">Home</a></li>
       <li><a [routerLink]="['/products']">Product List</a></li>
     </ul>
    `
})
```

Placing the Views

app.component.ts

```
...

@Component({
    selector: 'pm-app',
    template: `
     <ul class='nav navbar-nav'>
       <li><a [routerLink]="['/welcome']">Home</a></li>
       <li><a [routerLink]="['/products']">Product List</a></li>
     </ul>
     <router-outlet></router-outlet>
    `
})
```

How Routing Works



Passing Parameters to a Route

product-list.component.html

```
<td>
  <a [routerLink]="['/product', product.productId]">
     {{product.productName}}
  </a>
</td>
```

app.module.ts

```
{ path: 'product/:id', component: ProductDetailComponent }
```

Activated Route Service

Reading Parameters from a Route

product-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';

    constructor(private _route: ActivatedRoute) {
        console.log(this._route.snapshot.params['id']);
    }
```

app.module.ts

```
{ path: 'product/:id', component: ProductDetailComponent }
```

Placing the Views

app.component.ts

```
...

@Component({
    selector: 'pm-app',
    template: `
     <ul class='nav navbar-nav'>
       <li><a [routerLink]="['/welcome']">Home</a></li>
       <li><a [routerLink]="['/products']">Product List</a></li>
     </ul>
     <router-outlet></router-outlet>
    `
})
```

Placing the Views

app.component.ts

```
...

@Component({
    selector: 'pm-app',
    template: `
     <ul class='nav navbar-nav'>
       <li><a [routerLink]="['/welcome']">Home</a></li>
       <li><a [routerLink]="['/products']">Product List</a></li>
     </ul>
     <router-outlet></router-outlet>
    `
})
```

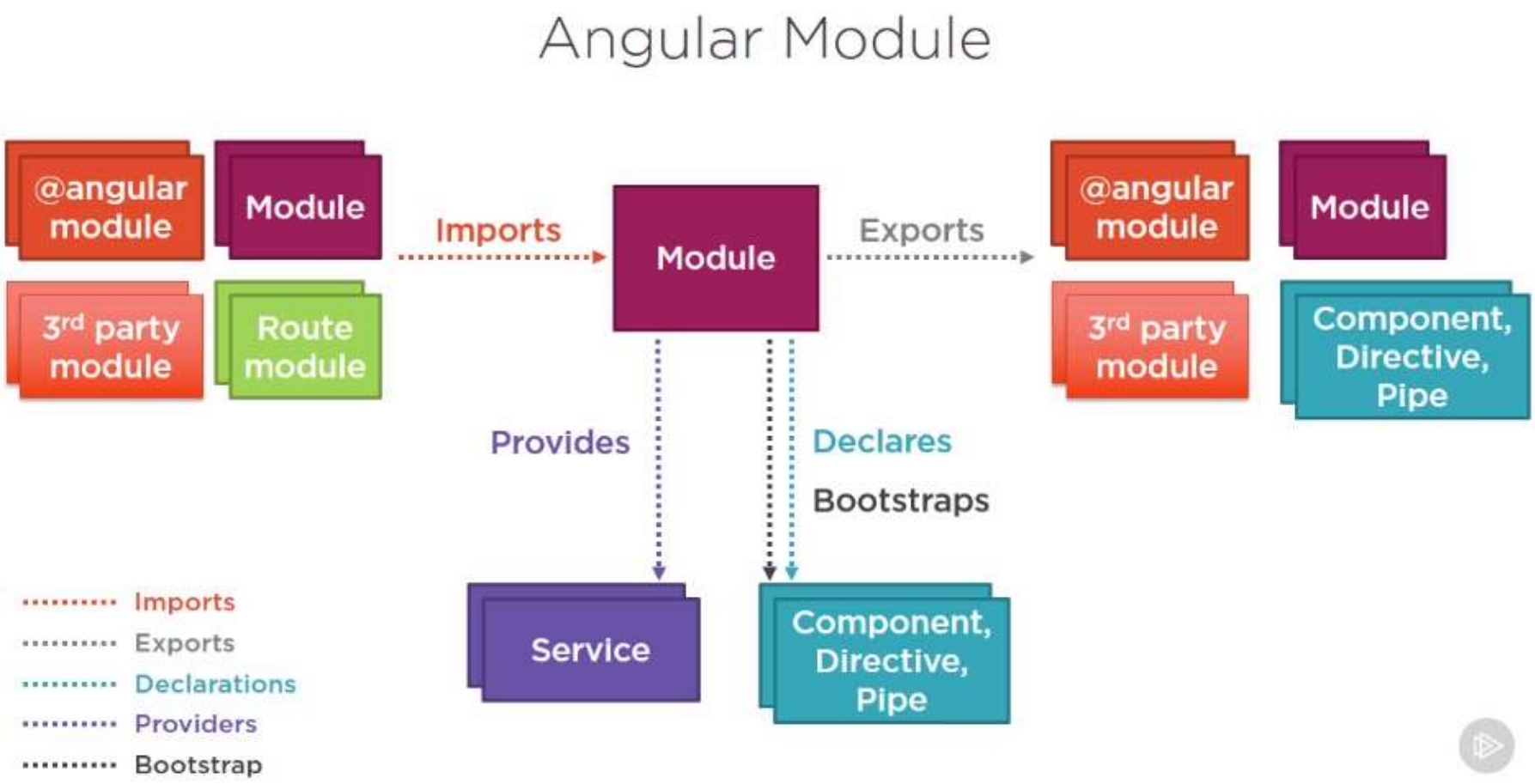
What Is an Angular Module?

A class with an NgModule decorator

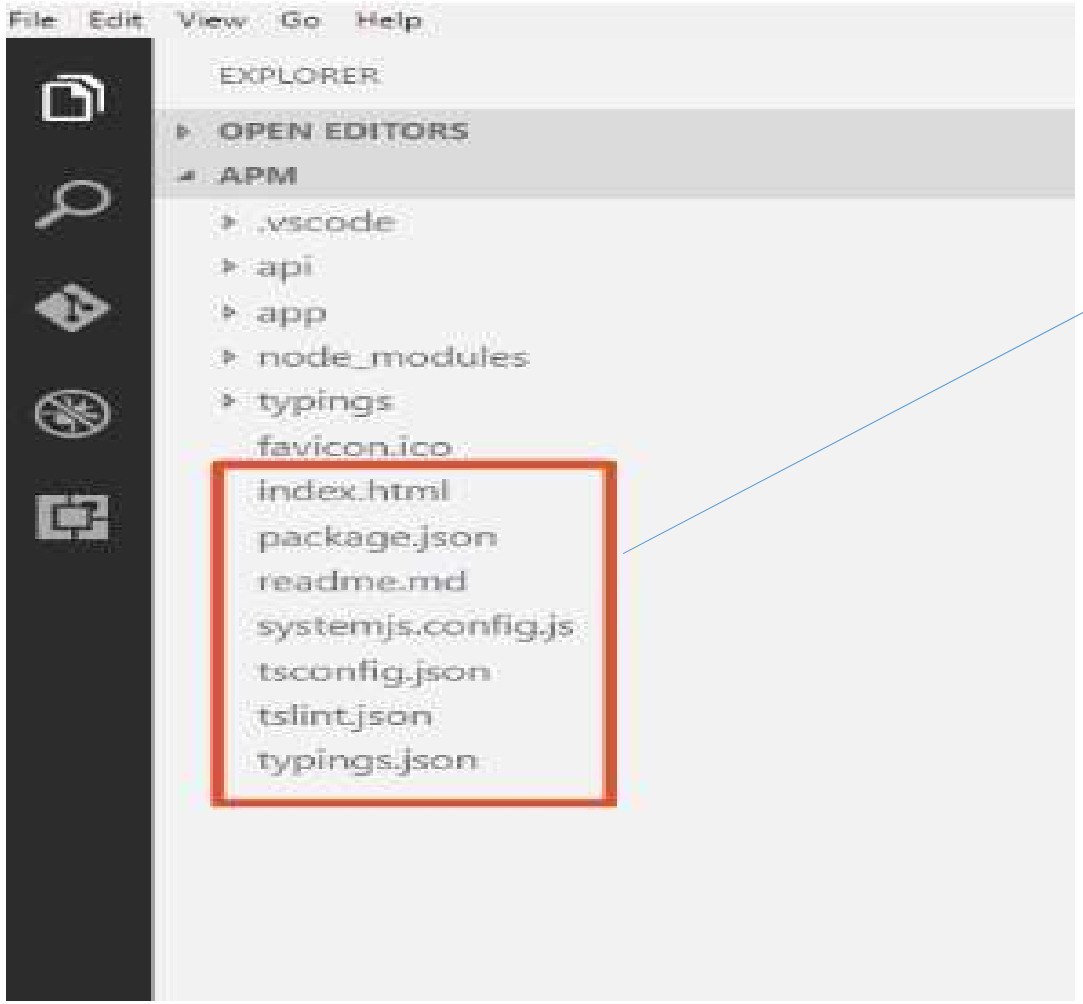
Its purpose:

- Organize the pieces of our application
- Arrange them into blocks
- Extend our application with capabilities from external libraries
- Provide a template resolution environment
- Aggregate and re-export

Module In Angular



Angular 2 Setup Revisited



Start Up files

Angular 2 Setup Revisited

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true
  }
}
```

- This is a Typescript configuration file.
- It specifies Typescript compiler options and other settings
- The typescript compiler tsc reads this file and compiles the code

Target ES% because browsers understand ES5

Means whether typescript should generate map files

Provides support for decorators. Must be set to true else application will not compile

For strongly type

Angular 2 Setup Revisited

typings.json

```
{
  "globalDependencies": {
    "core-js": "registry:dt/core-js#0.0.0+20160725163759",
    "node": "registry:dt/node#6.0.0+20160831021119"
  }
}
```

Brings ES2015 capabilities to ES5 browser

Node to develop and build application.
Also for lite server if node is not used

When installation is done using npm, the typescript definition files for each module is installed. In case it does not get installed the typings .json file can be used to find and retrieve these files
This file is processed automatically

Angular 2 Setup Revisited

package.json

```
{
    "name": "product-management",
    "version": "1.0.0",
    "author": "Deborah Kurata",
    "description": "Package for the Acme Product Management sample application",
    "scripts": {
        "start": "tsc && concurrently \"tsc -w\" \"lite-server\" ",
        "tsc": "tsc",
        "tsc:w": "tsc -w",
        "lint": "tslint ./app/**/*.ts -t verbose",
        "lite": "lite-server",
        "typings": "typings",
        "postinstall": "typings install"
    },
    "license": "ISC",
    "dependencies": {
        "@angular/common": "2.0.0",
        "@angular/compiler": "2.0.0",
        "@angular/core": "2.0.0",
        "@angular/forms": "2.0.0",
        "@angular/http": "2.0.0",
        "@angular/platform-browser": "2.0.0",
        "@angular/platform-browser-dynamic": "2.0.0",
```

This one of the most important file for successful setup and installation of angular 2 application.
Scripts, Dependencies and devDependencies are used when installing and running an Angular Application.
When we write npm install, npm reads all the dependencies locates each of these libraries and installs with the specified version.
Npm start runs both the typescript compiler and the lite server at the same time.
Postinstall is called automatically when npm successfully installs all the libraries of package .json

Basic info about the application

This keeps the compiler running, awaits changes for any ts file and recompiling as needed.

Lite server is a lightweight static file server that is used to run the application without any backend server.
Lite server is specifically written for angular 2

Angular 2 Setup Revisited

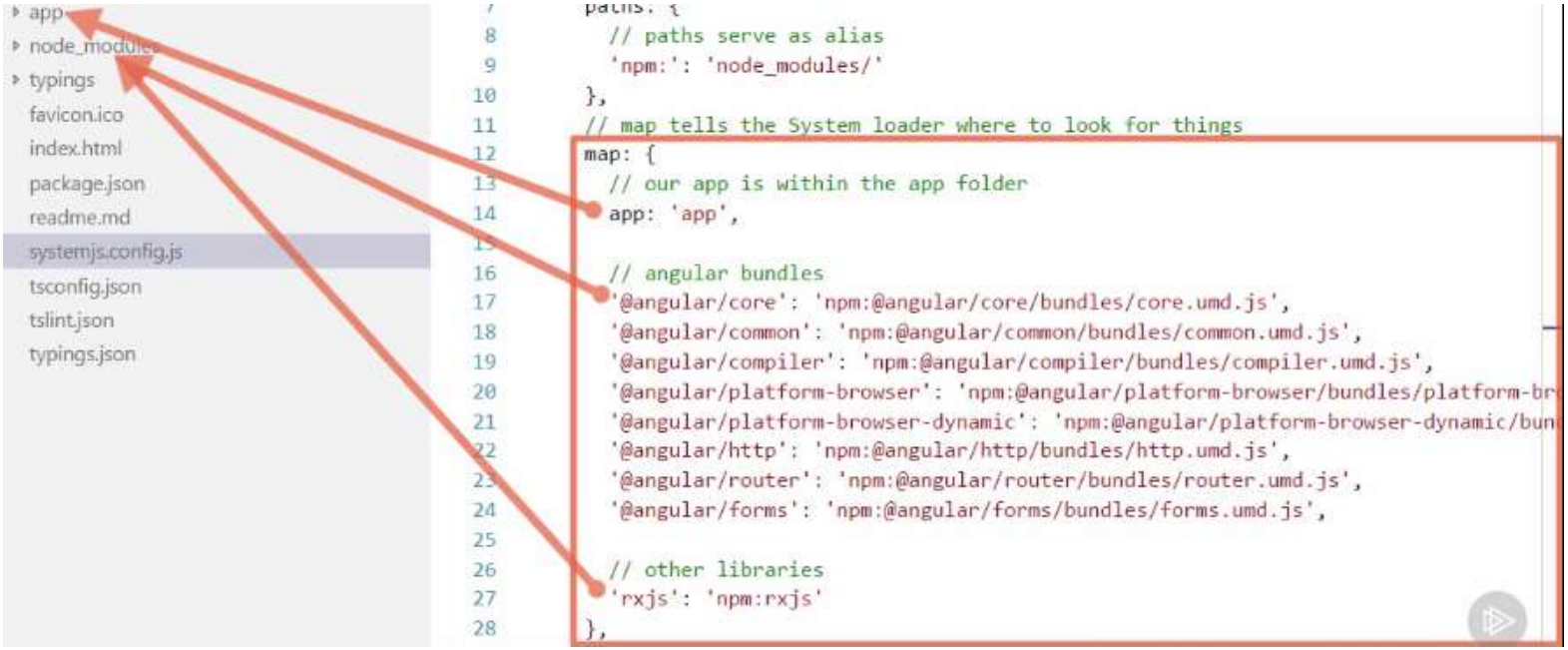
systemjs.config.js

```
3   * Adjust as necessary for your application needs.
4   */
5   (function (global) {
6     System.config({
7       paths: {
8         // paths serve as alias
9         'npm:': 'node_modules/'
10      },
11      // map tells the System loader where to look for things
12      map: {
13        // our app is within the app folder
14        app: 'app',
15
16        // angular bundles
17        '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
18        '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
19        '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
20        '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-br
21        '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-dynamic/bun
22        '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
23        '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
24        '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
```

This file configuration file configures SystemJS
SystemJS is an ES module loader that automatically loads each of the files for our application.
Hence we do not add script tags for every file used in the application.

The paths property defines an alias where the system files are located

Angular 2 Setup Revisited



SystemJS.config.js

The map property defines to the module loader where all the angular pieces are located

Angular 2 Setup Revisited

SystemJS.config.js

```
29      // packages tells the System loader
30      packages: {
31        app: {
32          main: './main.js',
33          defaultExtension: 'js'
34        },
35        rxjs: {
36          defaultExtension: 'js'
37        }
38      }
39    });
40  })(this);
```

The app property defines which file in the app folder should be loaded first.
It is main.js file.
defaultExtension tells that .js is the default extension

Extra Slides

Angular 2 Components- Property Binding

```
1  <h2 [textContent]="name"></h2>
2  <div>Watched on 1/13/2016</div>
```

- This is the property binding syntax
- Another way to bind data in templates with angular is through **property binding**.
- HTML elements have backing DOM properties that track stayed on elements and Angular 2 property binding syntax to wire into those properties.
- This is done with a specific syntax **a pair of square brackets around a property name** on an element.
- And you set these equal to a template expression following the same rules as you do for interpolation.

Angular 2 Components- Event Binding

```
<a (click)="onDelete()" class="delete">
    remove
</a>
```

media-item.component.ts app

```
1   import {Component} from 'angular2/core';
2
3   @Component({
4       selector: 'media-item',
5       templateUrl: 'app/media-item.component.html',
6       styleUrls: ['app/media-item.component.css']
7   })
8   export class MediaItemComponent {
9       onDelete() {
10          console.log('deleted');
11      }
12  }
```

- **Syntax for Event Binding**
- The event binding template syntax in angular allows to easily wire up event handlers from within the component templates.
- Events can be wired up native DOM element events as well as custom events created for components to emit.

Directives and Pipes "ng-If"

- Angular has two types of directives, **structural** and **attribute**.
- A structural directive is designed to alter the DOM layout by adding or removing DOM elements.
- Angular directive, **ng-if.**
- ng-if will conditionally render the DOM element that the directive is on.
- Structural directives are applied to normal DOM elements using an asterisk template syntax.
- So on the element, we need to put *ng-if and set that equal to a statement that will evaluate the true or false.
- Only for custom directives, you do need add them in the component metadata directives property.

Directives and Pipes "ng-If"

- The **ng-if** structural directive changes the DOM structure.
- The asterisk is what we refer to as syntactic sugar.
- Syntactic sugar is a short-hand pattern for writing something that the framework will interpret and convert to the actual syntax.
- Structural directives work with template elements to modify the DOM.
- A structural directive placed on an element named template, will handle either rendering or not rendering the inner-children of that template element in place of the template element, itself.

```
<h2>{{mediaItem.name}}</h2>
<div *ngIf="mediaItem.watchedOn">{{mediaItem.watchedOn}}</div>
```

Directives and Pipes "ng-If"

The asterisk syntax allows you to skip having to write the template tag, and simply put the structural directive directly on the element that is the contents of the template.
Note that both ways work the same.

ng-If with template syntax

```
<h2>{{ mediaItem.name }}</h2>
<template [ngIf]="mediaItem.watchedOn">
    <div>Watched on {{ mediaItem.watchedOn }}</div>
</template>
```

The Angular team has released a command line interface that will stream line the process of setting up an angular 2 project

An Angular application starts with an Angular module.
Angular modules help to keep application code organized by blocks of functionality and features.
A root module acts as a starting point module for an Angular application.
We will begin by creating the root module class in a file named app.module.ts, and that is in the app folder for the project.
We need to use a decorator to annotate that class so Angular will know it's an Angular module.
To inform Angular that the class code here is intended to be an Angular module, it needs to be decorated with the NgModule decorator.

Angular exports the NgModule decorator from its core scoped package.

Angular provides a handful of modules in its platform that it exposes via scoped packages.

The NgModule decorator comes from the core scoped package in Angular.

Since this is going to be our app root module, it will be named as AppModule.

The NgModule decorator takes in an object with some known properties to configure the class decorated as an Angular module.

These properties are known as metadata.

For the Angular root module, the imports, declarations and Bootstrap metadata properties are necessary properties.

All of these can be set up as an array.

The ngModule metadata properties

The **import property** is used to bring in other Angular modules that the module will need.

The **declarations property** is used to make components, directives and pipes available to your module that don't come from another module or created by the developer for the application.

The **Bootstrap property** is used for a root module and will let Angular know which component or components will be the starting point for the Bootstrap process, basically the entry point for application code.

The Browser Module

Since we are building a browser based app, we make use of the browser module that the Angular platform has available.
The browser module contains core directives, pipes and more for working with the DOM and can be found in the platform browser scoped package

Import {BrowserModule} from @angular/platform-browser;

The Starting component of the Application

The starting component is going to be named AppComponent and it is going to come from a file named app.coponent.ts.
This file will be located right next to the app.module.ts file.
Import statement for that AppComponent, and instead of loading from a bare module name, we can use a string that represents the path to the file relative to this file.
The extension is not needed due to the way the project is configured for system js module loading.
Add the AppComponent to the declarations property, as it's a component we want made available to this Angular module.
Finally, we need to add the AppComponent to the Bootstrap property for Bootstrapping the app.

The Starting component of the Application

Since this app module is being used as the root module,Angular will use the AppComponent as a target.

To build an Angular component, you need to use the component decorator on a class.

The component decorator comes from the core scope package in Angular.

To decorate a component, you need to provide two metadata properties at a minimum, **Selector,** and **template**, or **templateUrl**.

The selector property is what Angular will use to locate a custom HTML element, and apply the component too.

Import {AppComponent} from ‘./app.component’;

The index.html

The index.html file in the project has a custom HTML element named app in it, so this selector will target that.
Angular will use the template property content to fill the inner HTML of the targeted custom element when it's processed.

app.component.ts index.html

```
9     <script src="node_modules/reflect-metadata/Reflect.js"></script>
10    <script src="node_modules/systemjs/dist/system.src.js"></script>
11    <script src="systemjs.config.js"></script>
12    <link href="resets.css" rel="stylesheet">
13    <style>
14      body {
15        margin: 0px;
16        padding: 0px;
17        background-color: #32435b;
18      }
19    </style>
20    <script>
21          System.import('app').catch(function(err){ console.error(err);
              });
22      </script>
23  </head>
24
25  <body>
26    <app>Loading...</app>
27  </body>
28
29  </html>
```

Bootstrapping the module for the browser

The bootstrapping logic is written in main.ts file, which is inside the app folder.
Angular actually has support for running on multiple platforms.
The browser is considered a platform, the server and web worker are examples of other platforms.
Other third-party bootstraps could also be used to provide support for other platforms.
For this app, we are targeting the browser, so we need to bootstrap from that platform.
Angular exports a platformBrowserDynamic function for targeting the browser from the platformBrowserDynamic-scoped package, so we can import platformBrowserDynamic from there.
This function returns a platform object that has a bootstrap module function on it and this function is used to bootstrap your root module on the platform.

Bootstrapping the module for the browser

Note that earlier we were importing class types, and here we are importing a function.

The module-loading syntax supports importing all kinds of exported things, from class types and functions to constants, variables, and even JSON file data.

we can make the call to the platformBrowserDynamic function which will return an instance of a platform object.

That platform object has a method named bootstrapModule.

With platformBrowserDynamic call, we can call bootstrapModule.

This function is expecting a root module which is already created from AppModule.

We write an import statement to load this file, and then pass the AppModule type into the bootstrapModule function call.

Bootstrapping the module for the browser

At this stage all the initial starting bits written to get this Angular app up and running in the browser.

Let's open up a command line or terminal, run the **npm start** command to kick off the TypeScript build and web server and we'll watch for changes.

In the browser, we can see the content from the app component template is displayed.

main.ts

```
1  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
2  import { AppModule } from './app.module';
3
4  platformBrowserDynamic().bootstrapModule(AppModule);
```

EXTRA SLIDES

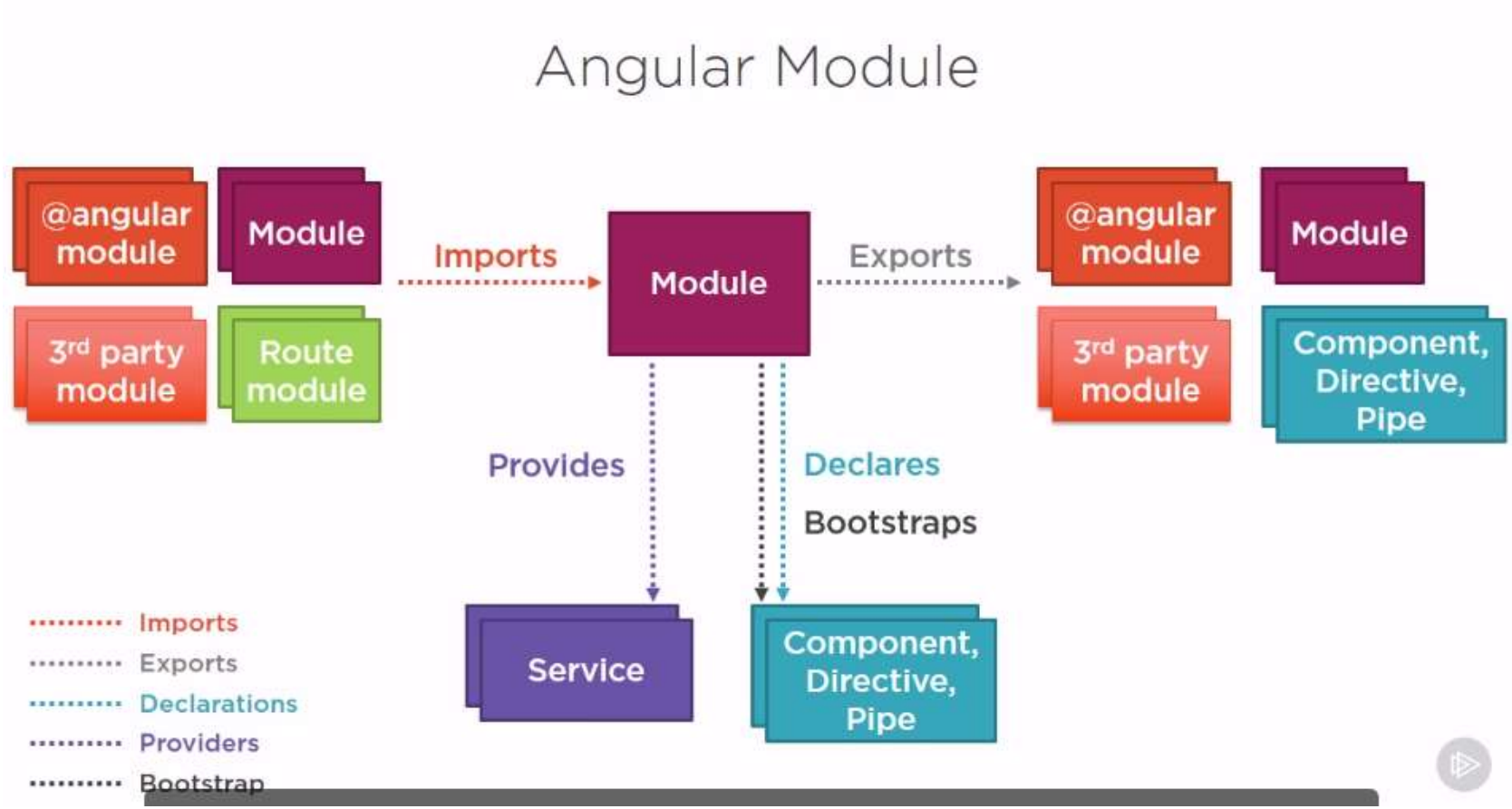
What Is an Angular Module?

A class with an NgModule decorator

Its purpose:

- Organize the pieces of our application
- Arrange them into blocks
- Extend our application with capabilities from external libraries
- Provide a template resolution environment
- Aggregate and re-export

The following can be done by ang angular module



HTTP Communication

- Communicating with the Server Using HTTP, Observables, and Rx
- It's important to understand the basics of HTTP communication so that we can see where RxJS fits into the mix. Here is a simple diagram of HTTP.
- The client sends a request to the server, and the server responds.
- This is an asynchronous communication pattern.
- The time between the request and the response could be several hundred milliseconds or more, so our client continues to process while this is happening.

HTTP Communication

- First : Using Callbacks
- In the old days, we handled this with a simple callback shown here. We made the request, and some kind of callback function handled the response whenever it returned.
- Second : Using Promises
- Then along came promises.
- Promises were nice because we could arrange our handling of async comprehensions a bit better.
- Multiple pieces of code could listen to the result of a single call, and we could avoid nested callback hell.
- Third:RxJS
- And now we have RxJS and their main feature, observables.

HTTP Communication

Callbacks

```
Server.request(requestData, function(responseData) {
    // asynchronously handle the data.
});

// this will execute before the callback
doMoreThings()
```

Promises

```
var promise = http.get(url, data);
promse.then(function(responseData) {
    // handle response
})

// this will execute before the then function
doMoreThings()
return promise;
```

HTTP Communication

Observables

```
var obs = http.get(url, data);
// manipulate the observable if desired
obs.subscribe(function(responseData) {
    // handle response
});
doMoreThings()
return obs;
```

Differences between promises and observables

Promises	Observables
Promises represent a single value that comes back some time in the future.	Observables, on the other hand, represent 0 or more values that come back either immediately or in the future.
Promises are limited to a single value	observables are not,
promises are asynchronous	observables are either synchronous or asynchronous.
promises do not.	There're plenty of other features that observables support
	Observables are often referred to as a stream of observable data or, in essence, any value that changes over time.

Promises are limited to single value where as observables are not
Promises are async but observables can be both sync and async

Observables

- Observables are often referred to a stream of observable data or any value that changes over time
- Consider for eg mouse clicks each of the mouse clicks have a specific coordinate and each click happens at a specific time.
- Observables are great to handle data like this.
- Let's say, for example, that we wanted to just react to each X position, not caring about the Y position.
- We can manipulate the stream so that we are only dealing with the X position like so.
- This is generally done through a map operation where you map an incoming value to a new value.
- With observables, this is a core activity and very easy.
- This comes in very handy when dealing with HTTP responses since the data that comes back is often way more than we want.
- There'll be status information, headers, etc.
- But really all we want is a return data.
- So mapping response to just a return data is very natural with observables.

- Observables have a lot of other features that are very valuable as well.
- They can be synchronous or asynchronous.
- They have improved error handling over how promises work.
- They can be closed independently of returning a value.
- They can also deal with time in a way that promised cannot.
- There're a bunch of advanced operations that they have as well, such as mathematical aggregation, buffering so that we get several results together in batches, or debouncing so that we don't get too many results if they happen too quickly.
- We can only deal with distinct values or we can filter the values down to just the ones we want.

- We can also combine multiple observables into a single observable.
- They have a built-in retry mechanism so that if the first try fails, they can retry again.
- When dealing with RxJS, it's not packaged like most libraries are.
- Most libraries have a single file that contains everything in the library.

Other Observable Features

Can Be Synchronous

Improved Error Handling

Can Be Closed Independently of Returning a Value

Can Deal with Time

Advanced Operations

- Mathmetical Aggregation
- Buffering
- Debounce
- Distinct
- Filtering
- Combining Observables
- Retry

RxJS Library Requests

- RxJS is different because it has so many operators, most of which won't be used on a typical project.
- Because of that, the library creators decided to pack each piece of RxJS separately.
- That means that every piece you need is in a separate file and, therefore, is a separate HTTP request.

RxJS Library Requests