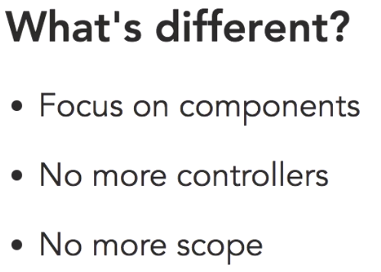


Angular JS is a modern framework for app development. That means that it's a structure that helps you create applications faster by providing several services and objects. Now those objects are going to make things easier for app developers. This is a dramatic upgrade to Angular One. It's really a rethinking of how applications should be put together.

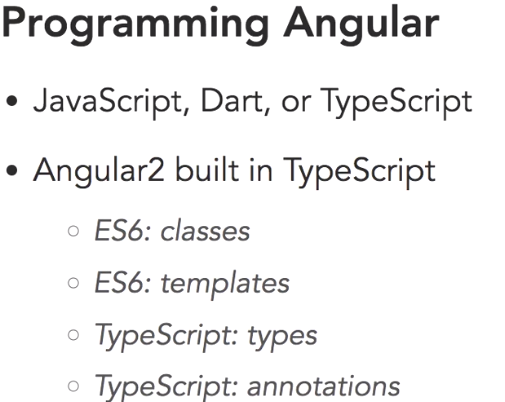
Angular JS is a modern framework for app development. That means that it's a structure that helps you create applications faster by providing several services and objects. Now those objects are going to make things easier for app developers. This is a dramatic upgrade to Angular One. It's really a rethinking of how applications should be put together.



First, in the new version of Angular, the structure revolves around components. Components are like Angular One's custom directives.

They'll let you create functionality around custom tags. So instead of adding an ng-app directive to a section of code, you simply invent your own tags and then program the functionality for these. It's a lot like using web components and it's one of the best features of the framework. Components are much simpler than creating custom directives. Since most of the work is handled by components, there's really no need for controllers in this new version of Angular

Another thing to get adjusted to is the fact that there's no more scope. Now this is a pretty big adjustment for somebody coming from Angular One. Scope is how we handled communication between our templates and our controllers.



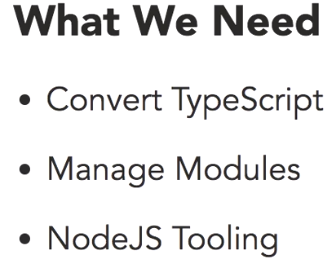
TypeScript is a language from Microsoft that is a super-set of JavaScript which means that it lets you use any of ES6's features and adds a couple of important features of its own. From version two on, Angular was built with TypeScript. So, there is a significant advantage to learning how to work with it. Mainly most of the documentation you'll find on the web and on the Angular site is specifically for TypeScript.

So, with TypeScript you get some of ES6's features and one of them is classes. ES6, which is also known as ECMAScript 2015, adds a few interesting features to this new version of Angular. And the first is classes.

ES6 also has a rich templating system that is extensively used in this new version of Angular, and it makes creating templates easier and much more powerful.

Now because TypeScript and ES6 are not supported in current browsers, you'll need to do something to convert this code to regular JavaScript

That means that you'll need to use a Build tool like Webpack or Gulp to handle the conversion. In this course I'll be using NPM Scripts.



**SystemJS**

Now because we're using the TypeScript version of Angular, we're going to need to have a process in place to convert the files to JavaScript so that our browser can understand it. In addition to that, because this version of Angular is modular in nature, and JavaScript doesn't understand natively how to manage modules, we need to have something that can do that for us. A popular library to take care of this is called SystemJS. It is used to process files.

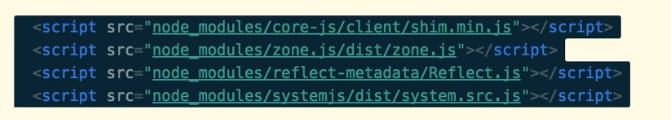
**Packages.json:**

Project has a lot of dependencies, mainly all the AngularJS dependencies.

TypeScript, as well as running a lite-server at the same time, and concurrently just makes that cross-platform.

Npm install command will download all the dependencies specified in the packages.json file.

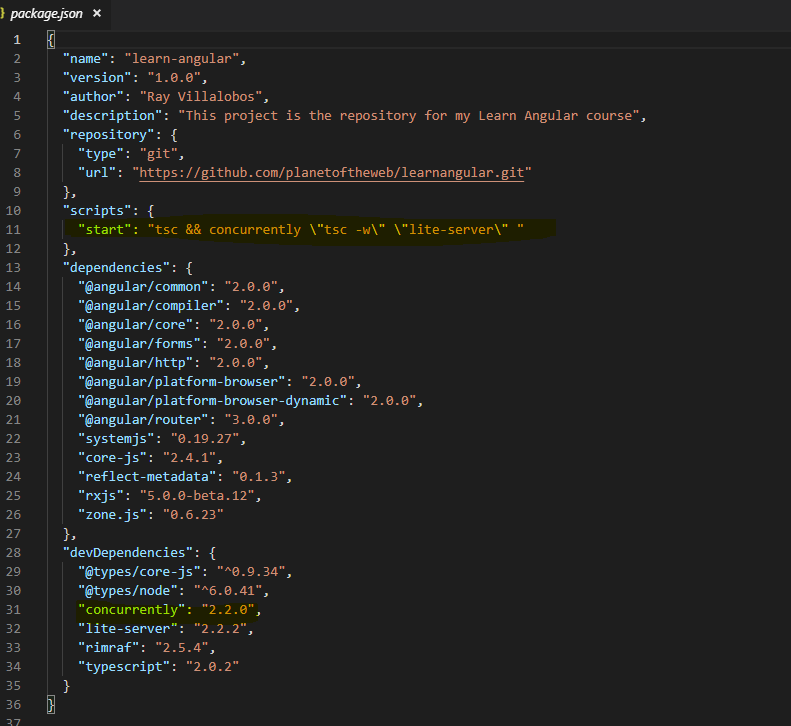
**Index.html:**



There are some scripts that Angular needs to be as backwards compatible as possible, and these are all them right here. Notice that this is using also the node\_modules folder. So these are all installations that are in the node\_modules folder.



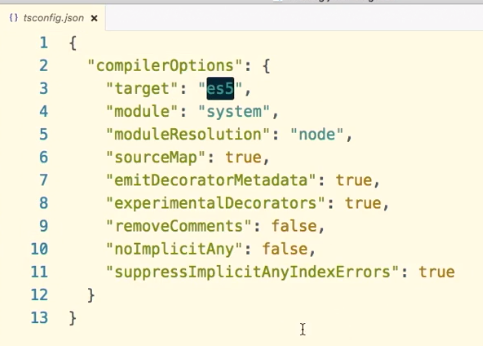
This file configures our SystemJS module loader.





**Systemjs.config.js** file sets up how the modules are going to be processed. So, we can see that the important parts of this is this **distribution** folder, which we've set to **app**, because our folder where we're going to place all our TypeScript is called app.

**Tsconfig.json:**

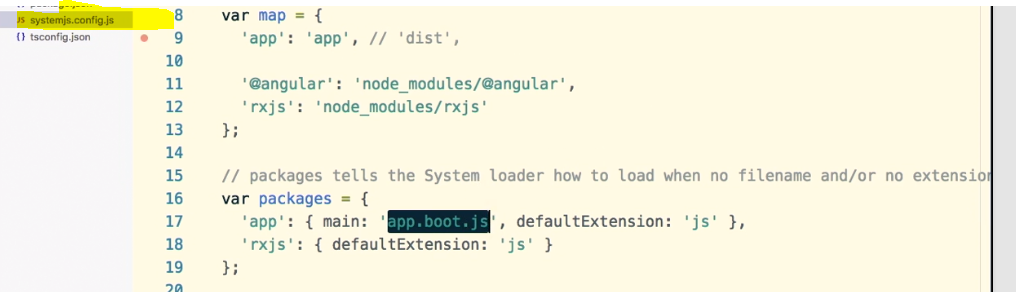


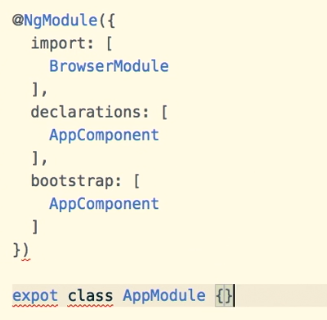
This determines how our TypeScript is going to be converted into regular JavaScript for us. So, we can see that we are targeting **es5**, which is the most common version that most browsers support.

We're using system as our module handler, as well as some of these other options.

To set up a typical application structure we're going to need **three** different files.

We need to use a file **to bootstrap our application (systemjs.config.js)**, another one **to describe all our modules(packages.json),** and then, finally, a module to start up what we want to do in the application**(app.boot.js)**. So first, if you notice, in our system configuration.js file, first, in the system.jsconfig file, we are calling this app.boot.js file, so this actually what our system loader is going to use to manage our modules.





NgModule is what we called a decorator, decorators are functions that modify JavaScript classes, essentially, it's just a configuration object, so think of it like a settings document for your specific component.

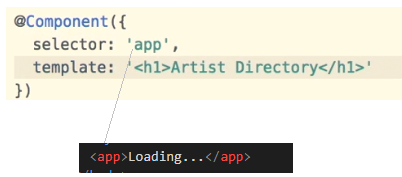
Inside NgModule we have 3 components:

1. **Import**: this is where we add modules that we need for our application to run
2. **Declarations**: these are the view classes that belong to this module.

Difference between the import command and the declarations is that the import imports modules that we're going to need, think of them as features for your application, while declarations will allow us to import all the modules that we have within our application

1. **Bootstrap**: Finally, we need to decide the bootstrap component for the application. This is going to be the component that has the other components, sometimes called the root component.

Now, other components are going to need this file to understand the relationships between the different files and components, so we need to go ahead and export this as an AppModule.



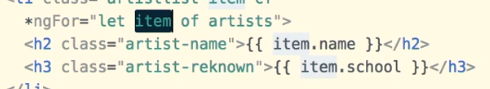
**Interpolation:**

Templates become more powerful when they can do things that are connected to our component. Now the simplest way to tie a template to a component is through interpolation. You can use double curly braces to bind the property name from your component to your template.



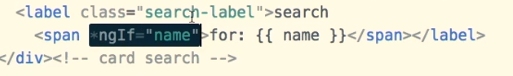


**To iterate through the list of arrays:**





**ngIf:**

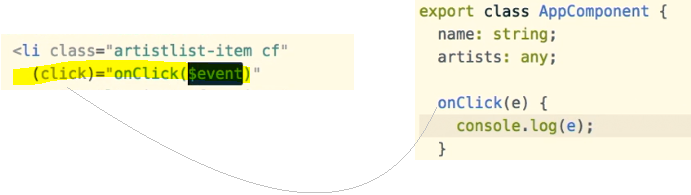


**If the name exists, then only print “for name”**

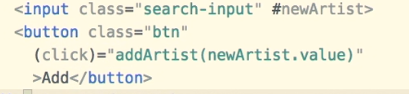
**Events:**

Parenthesis is used for events.

**Ex: Onclick event**

****

Template Variable:



**newArtist** is called as the template variable. In this case it is the input text variable

**Binding:**

Angular provides a way to bind properties to your template using **square brackets** So binding is just another way of connecting things with it you can control say a dom property through a variable or a method in your component

Bind-innerHTML and [innerHTML] will work similarly.

The square brackets notation gives us an easier way to access and modify properties that are in a template using something within the component.

**Two-Way Data Binding:**

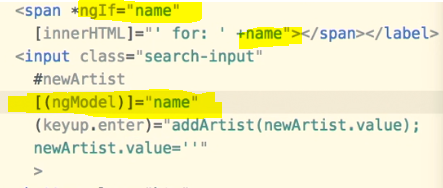
Sometimes, it's convenient to have an element that can both respond to an event and modify a property at the same time. Angular gives you a shortcut to do this that combines both techniques that we've doing so far. The technique is called two-way data binding because it's tracking an event and setting a value at the same time

Angular has a couple of directives called **ngModel** and **ngControl** that allow you to handle **two-way binding.**

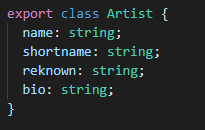
**ngModel**: when we want to work with an input field.

If we have a control, say something like a checkbox or a radio button or a pop-up, we will use ngControl.

We need to import **FormsModule** from Angular forms library to use 2-way data binding.



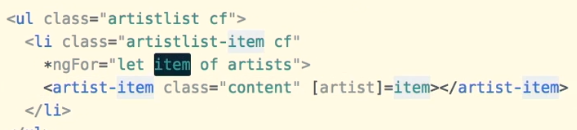
**Creating a new data type (like bean name in Java)**





whenever we are feeding this new component some information, we're going to have to pass it along as a property of this tag.

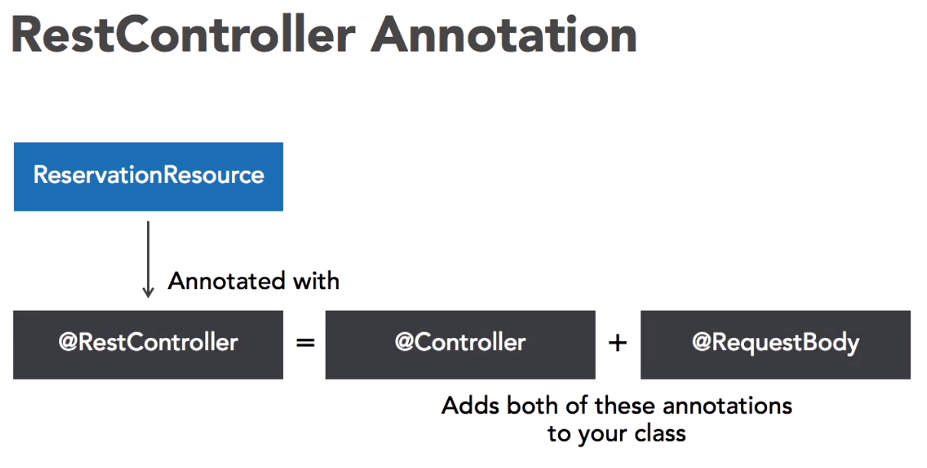
So, we create this tag, we can add a property called, and we will call this artist. And artist is going to be the variable that we feed into this component that is going to have the information for each artist, which will be like their name, their photo, and their renown

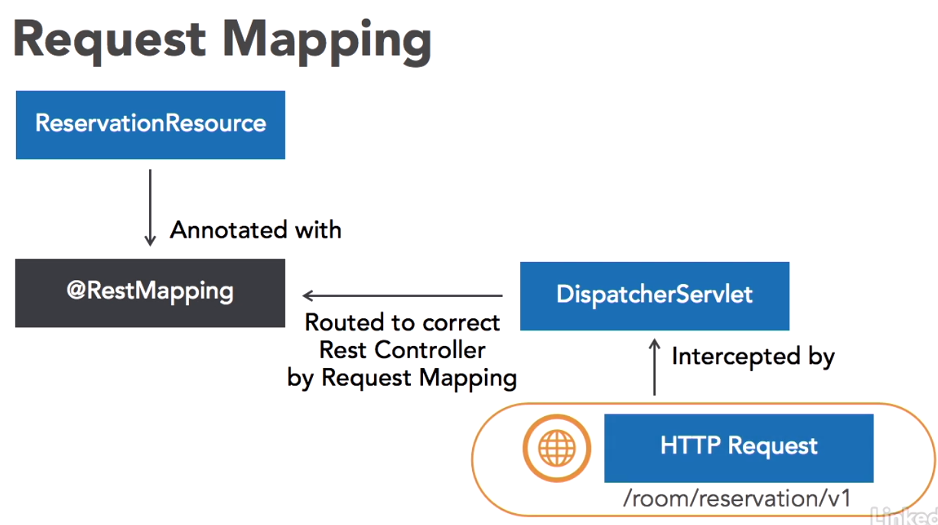


The property artist will be passed to the component artist-item.

For properties we use **brackets**. We’re going to call this property **artist**, because that is the input that this new component that we created expects.

New version of Angular, filters are called Pipes.





**Steps to create Angular and Spring Projects:**

1. Create a Spring boot project
2. Inside the SpringBoot project run: >npm install -g @angular/cli
3. Create a webapp folder inside src/main
4. Go to webapp folder and execute the following command:

>ng new angular2 --skip-git

By default, ng new will initialize your project as a Git Repository. Since we want our Java Spring re: project directory to be a Git Repo, instead of the AngularJS 2 project route, we will be passing in the flag skip-git.

**npm install bootstrap@3**

Twitter Bootstap is a grid based UI component framework. It will provide us with some good looking, out of the box, UI components to use.

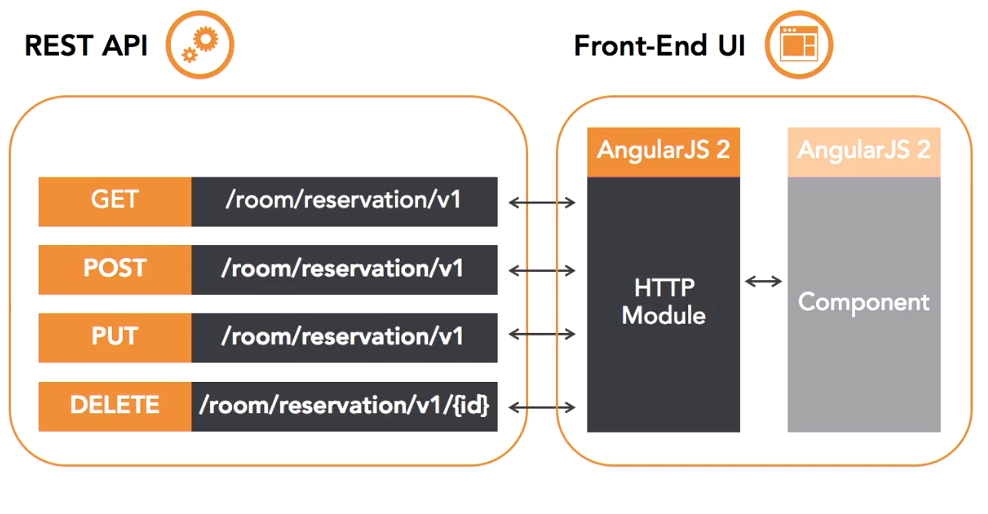
**npm install fuelux**

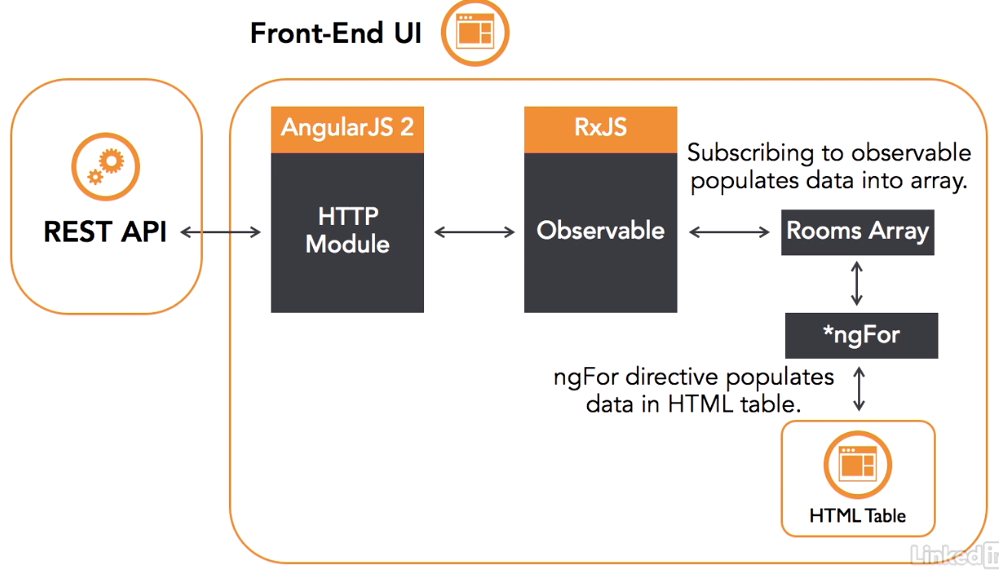
Fuel UX extends Twitter Bootstrap, and adds some additional UI components such as a datepicker

**Eclipse TypeScript Plug-in:**

<http://eclipse-update.palantir.com/eclipse-typescript/>

In AngularJS 2 you have an option to use a template driven form or reactive forms. Template driven forms are generally simpler and require less JavaScript. Reactive forms are much easier to unit test because they are model driven. You could test a reactive form using a unit testing framework such as Karma. Testing a template driven form would require an end-to-end testing framework such as Selenium.



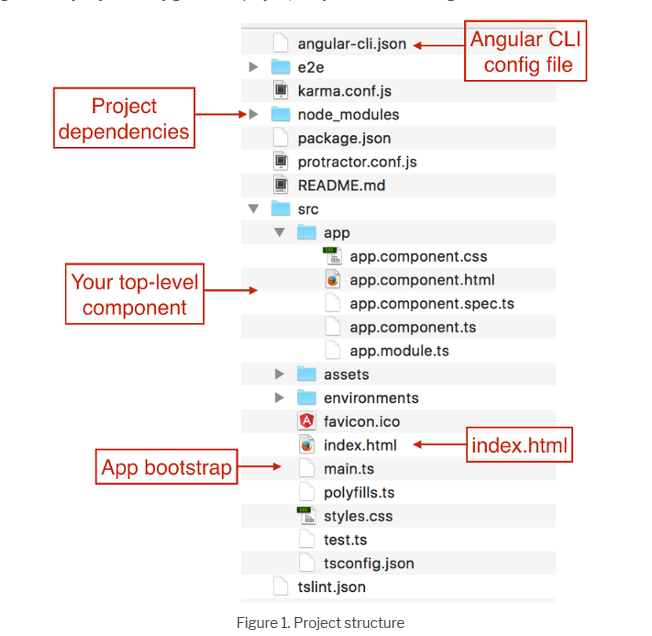


We will be using HTTP module to execute our rest API calls. We will receive what is called a observable in response. An observable is a new type of asynchronous data stream in Angular JS2. We will be using RxJS, a reactive library for JavaScript.

It is a third-party library that is endorsed by Angular JS2 as a framework. An observable is an asynchronous stream of data that is emitted over time. This leads to a better user experience since you can process data as it streams in.

First, let's add a cross origin annotation to our reservation resource. Since our Angular app is running on local host 4200 and our API is running on local host 8080, they are considered different origins.

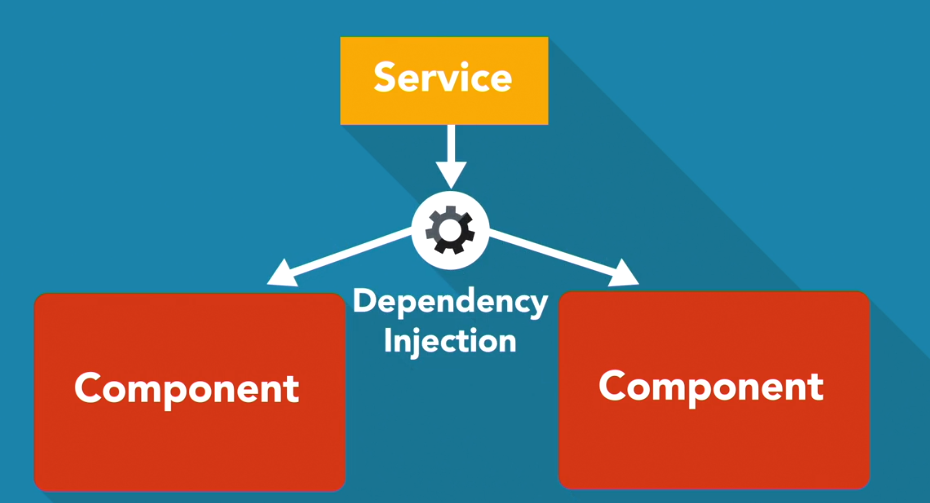




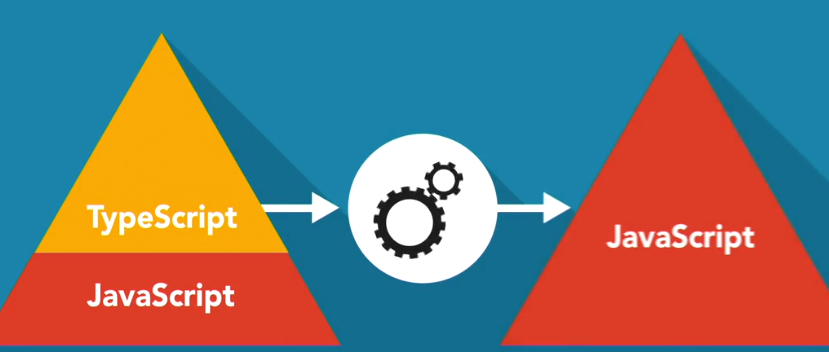
**Angular Essential Training:**

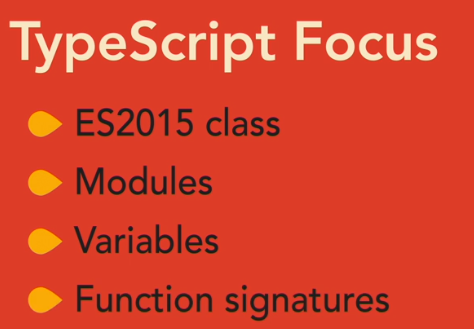
Angular provides a base for building rich client-side applications with a specialization on data binding, so when your web app needs to update content in your UI as a user enters or modifies it, you can easily do that with Angular.

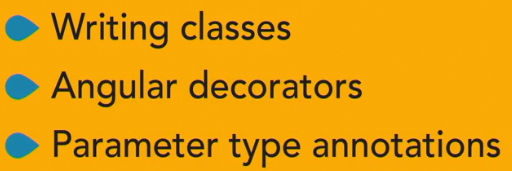
The Angular framework is not just about making the UI shine in this functionality. A clean dependency injection architecture is baked into the framework. So, when you want to write business logic in your client-side code and keep it decoupled from your view logic, Angular dependency injection model makes it easy



**TypeScript**: TypeScript is a typed superset of JavaScript that gets transpiled to plain JavaScript. Some of the advantages of using TypeScript is that you can write ES2015 classes and use Modules









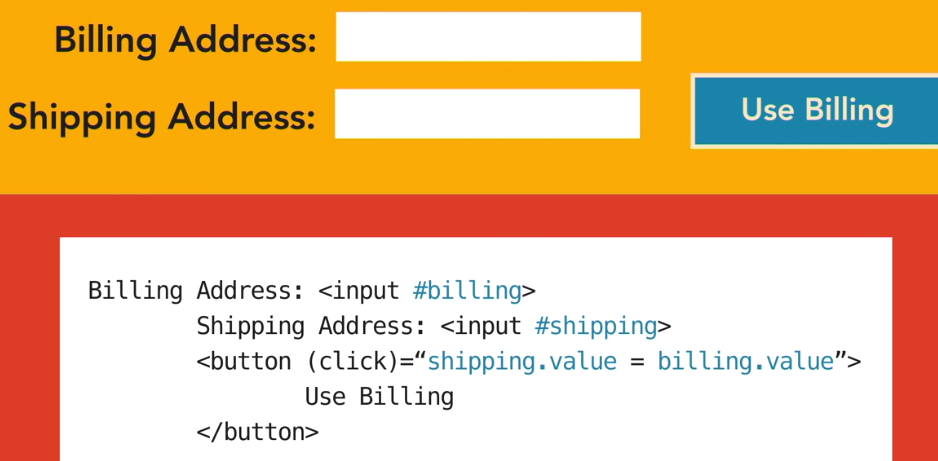
The starting point of an Angular app is the bootstrapping of the initial parent component. Much like the html DOM tree that starts with an html element and then branches down from there, Angular runs on a component tree model.

In Angular, a component is a directive with a template. Directives provide functionality and can transform the DOM. There are two types of directives: structural and attribute. Structural directives modify layout by altering elements in the DOM.

 The Angular framework is much more than just an engine for rendering markup templates through components. One of its strengths is the ease of binding data to views, and working with data in those views

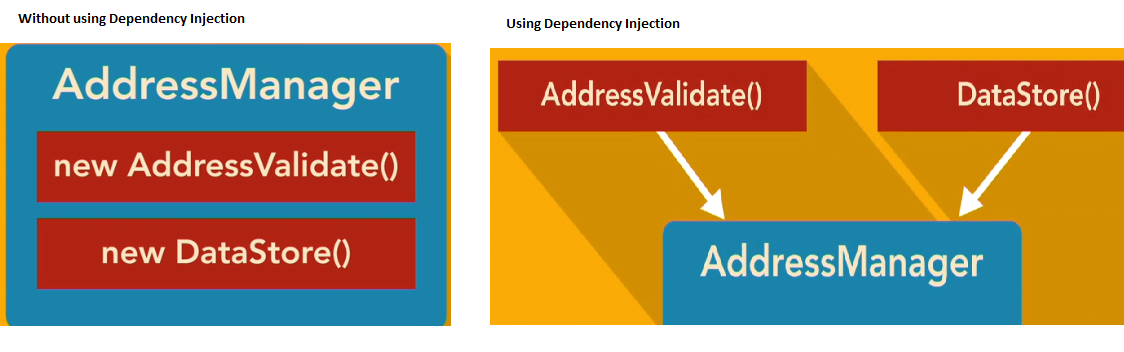
**Interpolation:**





We can also create and use local template variables created in markup using the hash to get a reference to the element

**Dependency Injection:**

Dependency injection, or DI for short, is the concept of inversion of control, or IOC for short, where you architect code in a way that you provide modules with other modules it needs to get some work done instead of having your modules go out and get other modules on their own. DI allows you to write decoupled code that is easier to unit test and to work with.

Ex:



The most common place you use DI is in your class constructors. So, constructors for components, directives, pipes, and services you write can leverage the way Angular does DI. You can simply declare types on your constructor parameters with some help from TypeScript, and Angular will interpret that and make sure you receive an instance of that type when your constructors run.



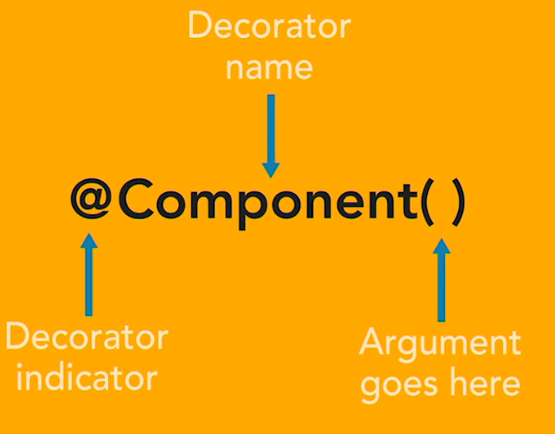
We can process data to and from an API by leveraging the **HTTP protocol**

Angular provides an HTTP module in the framework for abstracting out working with the way XHR and JSONP calls are done via client script. So, we can do things like make GET and POST calls that work with JSON data as simple as passing a URL and a JavaScript object to an Angular HTTP function and subscribing to the results.



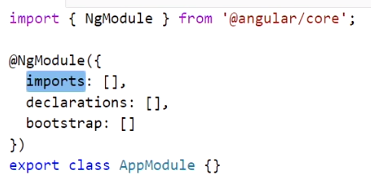
**Decorator:**

Expression that evaluates to a function allowing annotation of classes at design time.  A decorator is an expression that evaluates to a function that makes it possible to annotate and modify classes at design time. Typescript provides support for decorators through it's transpiler.



**Module:**

Starting point of angular application is the module. We can create a module using @NgModule. The NgModule decorator comes from the core scoped package in Angular.



The **NgModule** decorator takes in an object with some known properties to configure the class you decorate as an Angular module. These properties are known as metadata (**imports**, **declarations**, **bootstrap**).

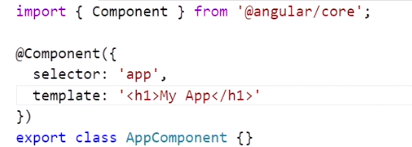
we need to provide support for using the module loading syntax. You do this by using the **export** keyword in front of the class keyword.

**Import**: This property is used to bring in other Angular modules that your module will need.

**Declarations**: This property is used to make components, directives and pipes available to your module that don't come from another module.

**Bootstrap**: This property is used for a root module and will let Angular know which component or components the starting point for the Bootstrap process will be. Basically, the entry point for your app code.

**Component:**



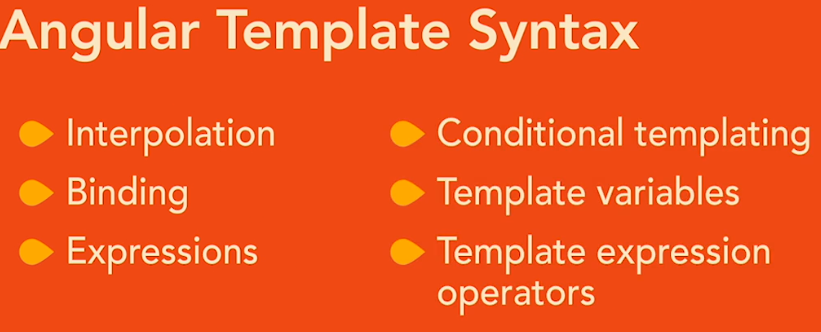
With an Angular root module and a starting component created, the next step to getting the foundation of an Angular app up and running is the code to bootstrap the module.

We can add a file named main.ts to the app folder and put the bootstrap logic in that file.

Angular 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.

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 run the **npm start** command to start up the site and have it reloaded as you change code.

**Interpolation** is a way to get data displayed in the view. We do interpolation by using a pair of matching curly braces in the markup. And the contents of the double curly braces are a JavaScript-like expression that Angular will evaluate and then convert to a string

**Binding**:

**Property Binding:** HTML elements have backing dom properties that track state on elements.  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.

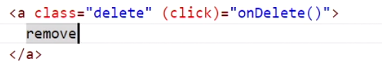
**Approach1**: name configured in the component.ts will be displayed



**Approach2**:



**Event Binding:** Property binding makes use of the square brackets; event binding makes use of the parenthesis.



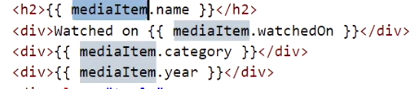
We can add the event named click, wrapped with parenthesis, and set it equal to a statement in quotes that will be evaluated. Let's bind this to a function call onDelete. Angular is expecting the onDelete method to be available in the execution context.

**Input decorator:**

The Input decorator is designed to be used on a class property.







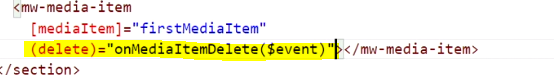
In this syntax, the mediaItem, which is the input property from the mediaItem component, is the binding target, because it's on the left of the equal sign. The first mediaItem property from the app component is the binding source, which is to the right of the equal sign.

The DOM has built-in properties on existing elements. Components you create do not. That is why the Input decorator is used to give component properties that you want to expose for use when using the component.

**Output decorator:**

The different with output is we want to set the property to an EventEmitter object. What we are doing here is exposing an event that can be subscribed to on our custom element, just like a native DOM event. So, we have this EventEmitter that we set the property to. This will allow us to emit the event.





The delete property, which is an EventEmitter, has a method named emit. So we call this.delete.emit.

When Angular evaluates the statement, it will handle getting the emit value and setting it to the variable $event. So, we can just use that in a statement and pass it into the method call. So, this is us in the app component telling Angular "hey, when this media item component emits its delete event, we want to respond to that by calling the app component onMediaItemDelete method." The output decorator allows you to do the same thing with your components that Angular is doing with those native DOM events.

**Directives:**

Angular has two types of directives. **Structural** and **attribute**.

**Structural** directives are applied to normal DOM elements, using an asterisk template syntax. So, on the element, we need to put asterisk ngIf and set that equal to a statement that will evaluate to true or false.



we want to see if mediaItem.watchedOn has a value, so we can put mediaItem.watchedOn in the expression. And if watchedOn has a value, it will evaluate to truth.

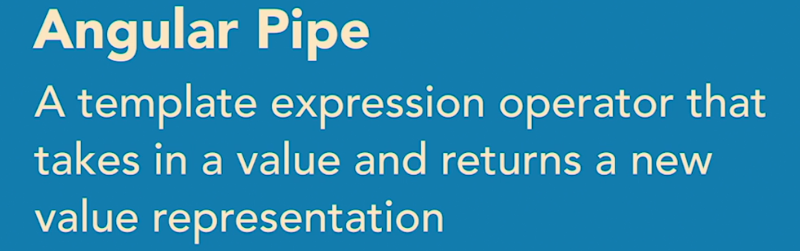
The asterisk is what is referred to as syntactic sugar. Syntactic sugar is a shorthand pattern for writing something that the platform will interpret and convert to the actual syntax. Structural directives work with ng-template elements to modify the DOM.

\*ngFor

**Attribute** directives are designed to change the appearance or behavior of the DOM elements that they are attached to.



 The ngClass directive expects an object structure for its statement value.



**Custom Pipe:**

Angular calls method with the name transform on a pipe class, and it will pass in the pipe argument, or value being piped to it, as the first parameter. If the pipe has parameters, the values after the colons in the template, it will pass those in as parameters in list form, after the first argument. Let's make use of the value parameter. So, we add a parameter named mediaItems.



Returning distinct mediaItmes with comma separated values.

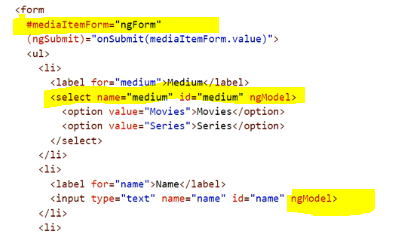
**Angular Form:**





There are two common approaches to building forms in Angular: template driven, where most of the form logic is crafted in the template markup, and model driven, where most of the form logic is crafted in the component class. Overall, the form support build into Angular covers a wide range of input collection scenarios.

Ex: Template driven HTML







In template driven approach that the angular form system is working with a form group.

Model driven forms are built using the ReactiveFormsModule, instead of the FormsModule (Template Driven).

ngOnInit: Angular Life Cycle event.

FormGroup expects to be called with an object structure that has properties named for the controls that will be in the group set to a value of a new control.



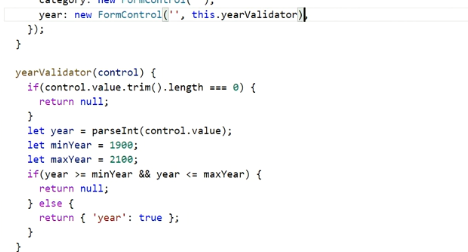


**Validators:**

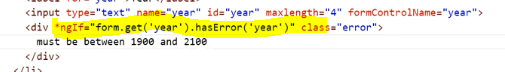


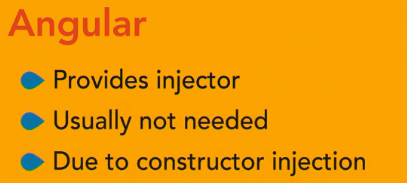
**Custom Validator:**

To build your own custom validators you need to create a function that will receive an object which Angular will pass in as either a FormControl, FormGroup, or FormArray and that function needs to return null when valid and return an object if invalid.



**Error Handling:**





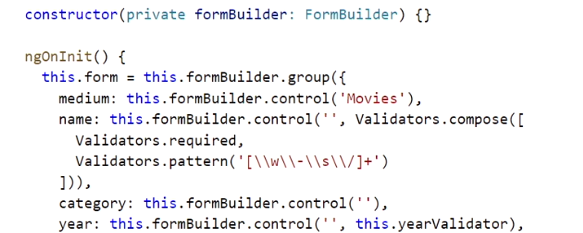


Angular has a component tree, and we can register things at a component level, those singletons of things become available from the point in the component tree that they were registered on down. So, something registered at Bootstrap is available in the entire component tree. Something registered at a component is available in that component and its children and their children. And finally, if you create your own classes to encapsulate some logic, referred to as services, it is extremely simple to tell Angular that you want it to do constructor injection for service class as well.

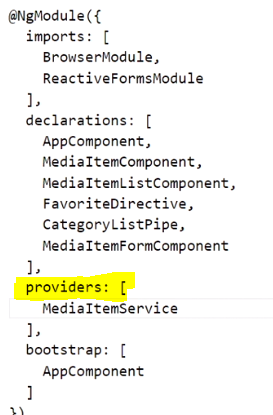
Services provide an architectural way to encapsulate business logic in a reusable fashion, allowing you to keep that logic out of your components, directives, and pipe classes.

**FormBuilder** class to help in the creation of the FormModel to see constructor injection in action. The formBuilder is a class provided in the Angular platform that has a group method on it for building a form group.

**Building a form using FormBuilder:**



To make services available to angular modules, you need to provide them. The NG module metadata has support for a property named providers. So, we can add a providers property onto the object literal that is being passed into the NG module decorator.



# Angular Constructor vs ngOnInit

The constructor is not actually part of Angular 2. It is feature of Typescript class, which is **called when class is instantiated**. Constructor is used to create new instance of a class. It can’t be used to decide when Angular has finished the initialization of component.

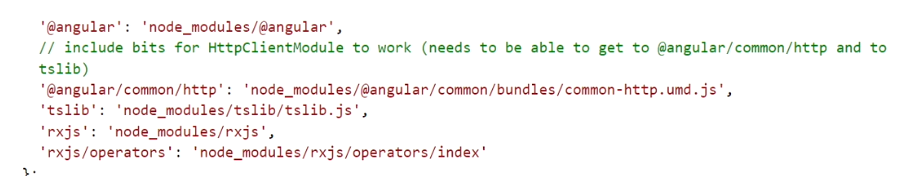
In Angular, constructor can be used to initialize the fields and angular DI [Dependency Injection] looks for constructor parameters to find providers and resolve dependencies and passes them as constructor arguments.

Since constructor is called by JavaScript engine not by Angular, **ngOnInit** lifecycle hook was created in Angular. ngOnInit signals that Angular has finished initializing and setting up component.

The ngOnInit is called after the constructor is executed. In constructor Angular initializes and resolves all class members so in ngOnInit you can initialize work and logic of the component. ngOnInit guarantees that your bindings are readily available.

**HTTP Module:**

Angular has a module named HttpClient module for working with HTTP calls from the client.

**Systemjs.config.js**

The HttpClient module works with observables and RxJS. So, the RxJS bundle is being included as well in the map and packages object literals.

Angular has a class named HTTP Client, that is a service for making HTTP request calls and returning HTTP responses

**Routing:**

Angular has a module bundle named Router for handling client-side routing. To make use of routing in your Angular application, you need to do a handful of things, the first of which is setting up the base href in the dom. The router makes use of the browser's history push state for navigation and URL interaction.



Angular provides a way to get information about the **currently activated route** through a service of type activated route. Getting to that information is just a constructor injection away