

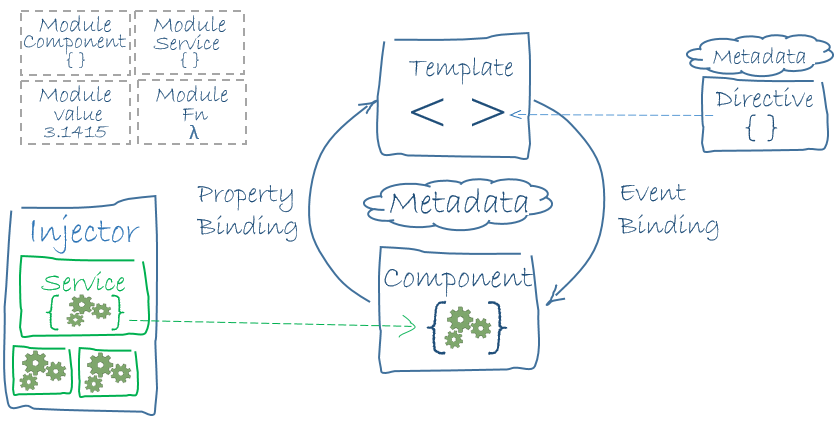
Angular framework

**Angular** (commonly referred to as "**Angular 2+**" or "**Angular v2 and above**") is a [TypeScript](https://en.wikipedia.org/wiki/TypeScript)-based [open-source](https://en.wikipedia.org/wiki/Open-source_software) front-end [web application platform](https://en.wikipedia.org/wiki/Web_framework) led by the Angular Team at [Google](https://en.wikipedia.org/wiki/Google) and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built [AngularJS](https://en.wikipedia.org/wiki/AngularJS).

**Angular** is a JavaScript framework that helps developers build applications. The library provides several features that make it trivial to implement the complex requirements of modern applications, such as data binding, routing, and animations.



Differences between Angular and AngularJS



Architecture of an Angular application. The main building blocks are modules, components, templates, metadata, data binding, directives, services and dependency injection.

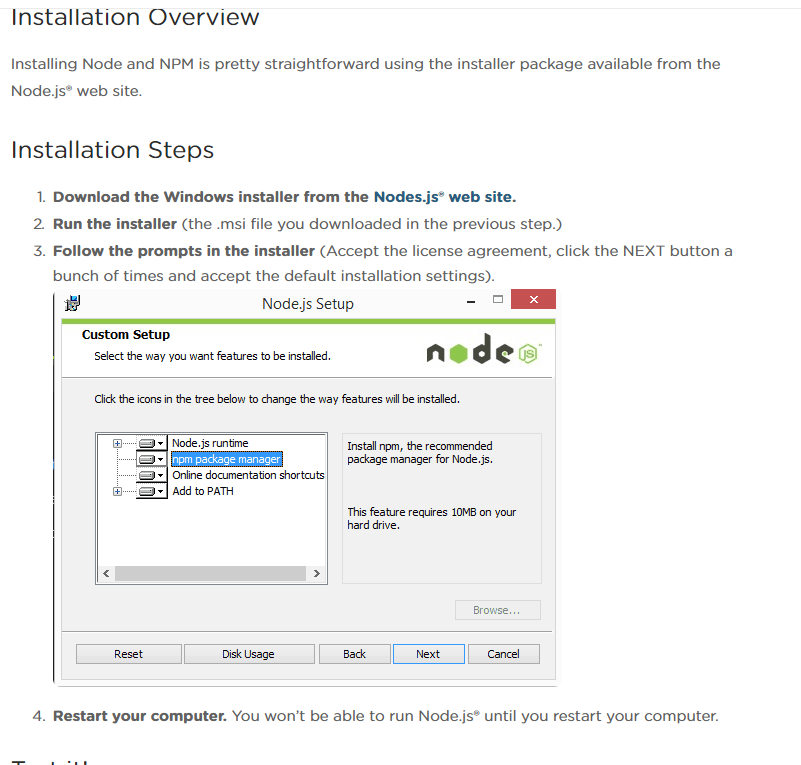
Angular was a ground-up rewrite of AngularJS.

* Angular does not have a concept of "scope" or controllers, instead it uses a hierarchy of components as its primary architectural characteristic.
* Angular has a different expression syntax, focusing on "[ ]" for [property](https://en.wikipedia.org/wiki/Property_(programming)) binding, and "( )" for [event](https://en.wikipedia.org/wiki/Event_(computing)) binding[[7]](https://en.wikipedia.org/wiki/Angular_(application_platform)#cite_note-7)
* Modularity – much core functionality has moved to modules
* Angular recommends the use of Microsoft's [TypeScript](https://en.wikipedia.org/wiki/TypeScript) language, which introduces the following features:
  + Class-based [Object Oriented Programming](https://en.wikipedia.org/wiki/Object-oriented_programming)
  + [Static Typing](https://en.wikipedia.org/wiki/Static_typing)
  + [Generics](https://en.wikipedia.org/wiki/Generic_programming)
* [TypeScript](https://en.wikipedia.org/wiki/TypeScript) is a superset of [ECMAScript 6](https://en.wikipedia.org/wiki/ECMAScript_6) (ES6), and is [backwards compatible](https://en.wikipedia.org/wiki/Backward_compatibility) with [ECMAScript 5](https://en.wikipedia.org/wiki/ECMAScript_5) (i.e.: JavaScript). Angular also includes [ES6](https://en.wikipedia.org/wiki/ECMAScript_6):
  + [Lambdas](https://en.wikipedia.org/wiki/Lambda_(programming))
  + [Iterators](https://en.wikipedia.org/wiki/Iterator)
  + For/Of loops
  + [Python](https://en.wikipedia.org/wiki/Python_(programming_language))-style generators
  + [Reflection](https://en.wikipedia.org/wiki/Reflection_(programming))
* [Dynamic loading](https://en.wikipedia.org/wiki/Dynamic_loading)
* Asynchronous template compilation
* Iterative callbacks provided by RxJS. RxJS limits state visibility and debugging, but these can be solved with reactive add-ons like ngReact or ngrx.

How to install Angular on Windows

<https://angular.io/guide/setup>

* Nodejs - https://nodejs.org/en/
* Npm



HAVE TO RESTART the computer….

Installation structure:

<https://blog.teamtreehouse.com/install-node-js-npm-windows>

node -v

npm -v

npm install npm@latest -g

git clone https://github.com/angular/quickstart.git quickstart

cd quickstart

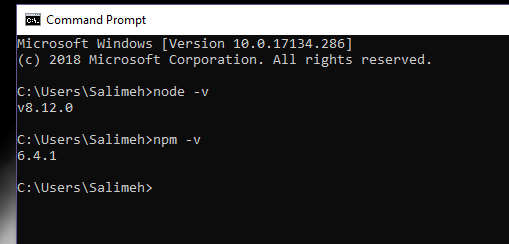
npm install

npm start

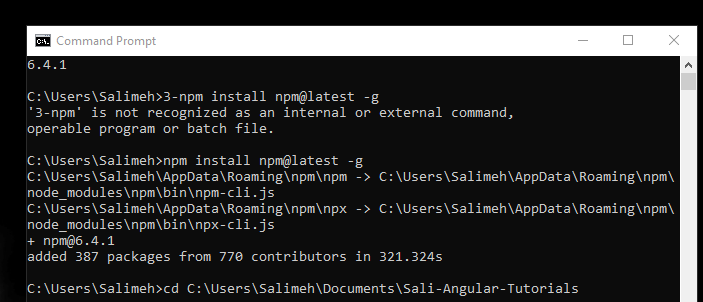
node -v



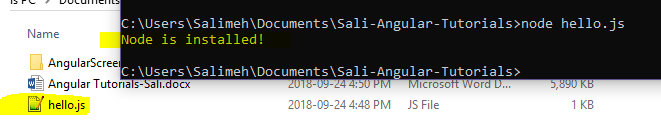
npm -v



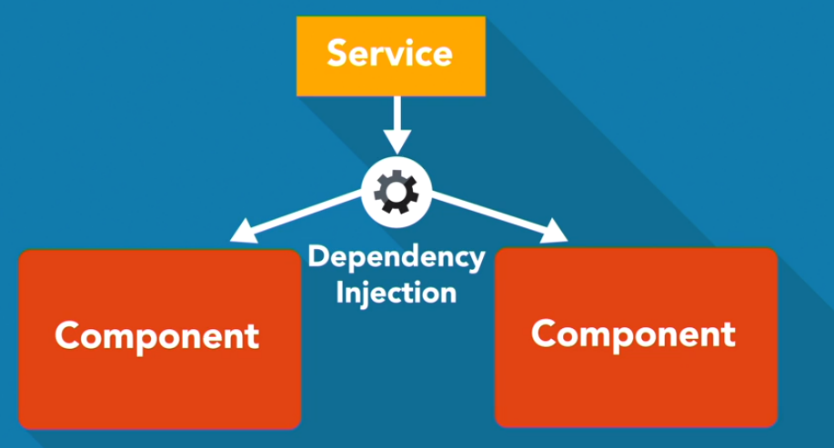
npm install npm@latest -g



hello.js -> in the file : console.log('Node is installed!'); -> cd to the file -> node hello.js



Update data in UI – Reusability – dependency injection



Managing common functionalities through out the website.

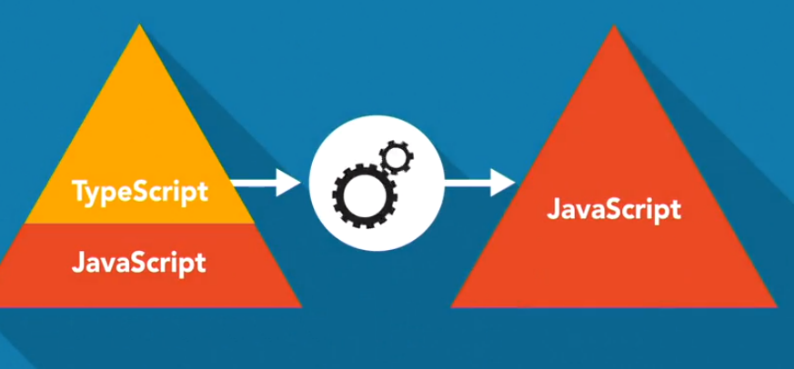
Build upon component model and JavaScript framework.

1. Dependency injection
2. Ultrafast change detection
3. Structure rich template with interactive DOM properties and events
4. Building comprehensive routing

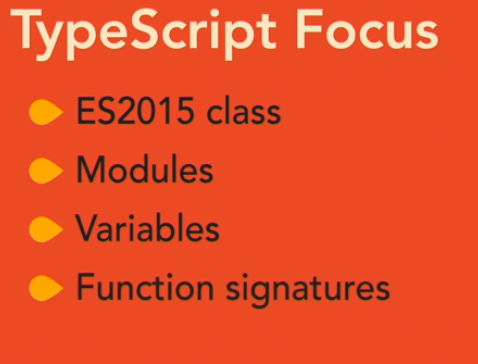
Angular source code is written in TypeScript.

TypeScript is an open-source programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript and adds optional static typing to the language. TypeScript is designed for development of large applications and transcompiles to JavaScript.





TypeScripts benefits:

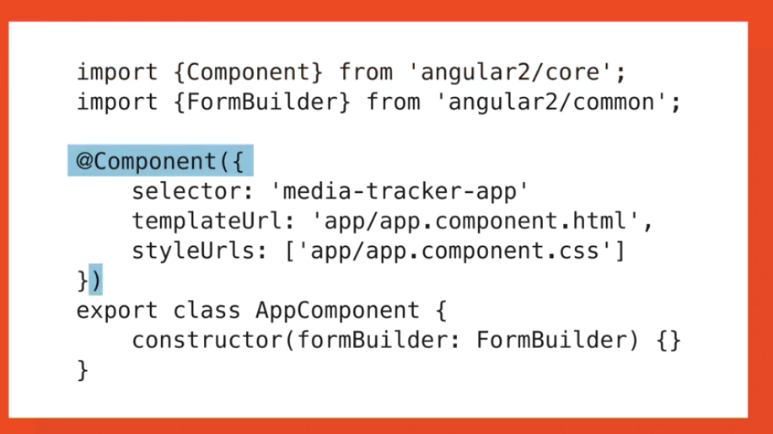


* Simplicity and less code by using framework
* Angular source code increase productivity and learning pace

Using strong typing in javascript is completely optional though which means we can write pure JavaScript in TypeScript file.

Typescript focus:

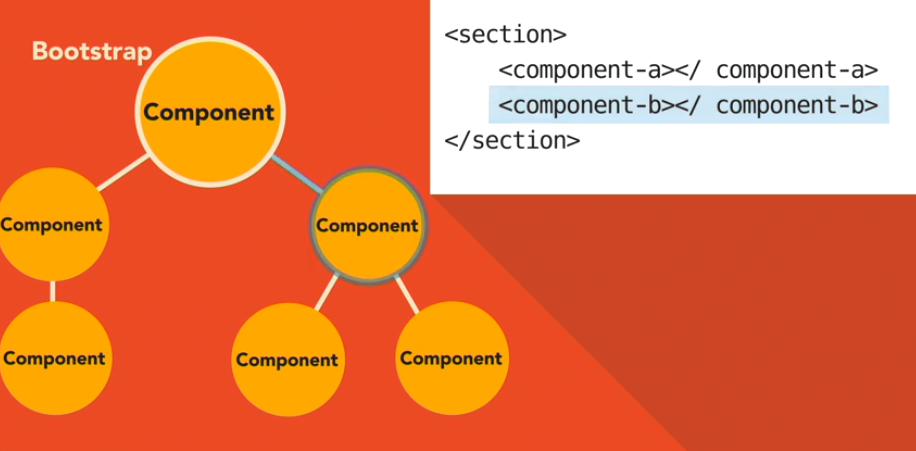
* Writing classes
* Angular decorators
* Parameter type annotations





How angular works:

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. 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.



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.



For example, you can have a MediaItemComponent that can have a property named mediaItem that represents the data for a media item. And that component can also have a method called onDeleteClick that could handle raising the delete media item event. With each component in Angular, you can specify an html template, the markup that will get rendered, and using the component class, and how Angular renders the component, you can display the data for the mediaItem property in your template.



Angular provides an easy syntax known as the template syntax, To wire up to DOM events within your template, so you can wire up the click event on a button to the **onDeleteClick** method.

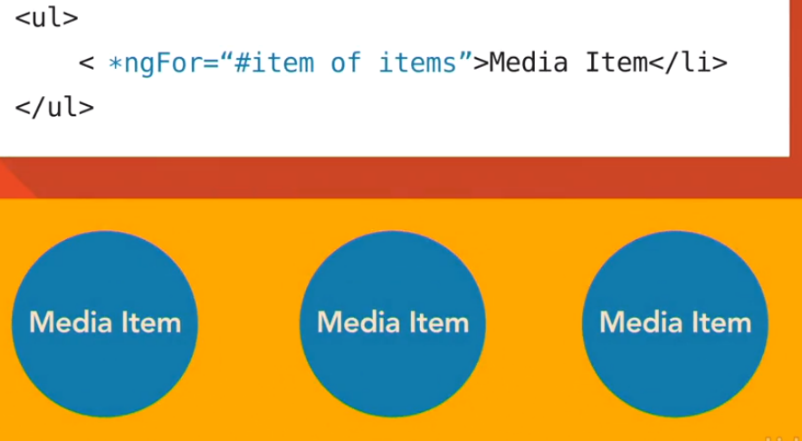
You can even use components within components. This is where the component tree comes into play. Much like how you write html, creating nested elements within elements, you can build out your Angular apps by having components rendering components within their templates. Each component gets configured with a selector, which tells Angular what markup element tag to associate the component class logic with.



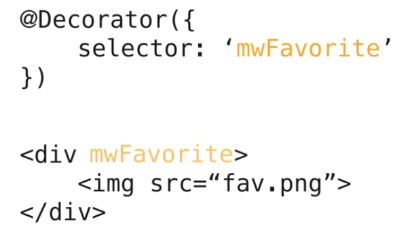
In Angular, a component is a directive with a template. Directives provide functionality and can transform the DOM. There are two types of directives: **structural** and **attribute**. Structural directives modify layout by altering elements in the DOM. Attribute directives change the behavior or appearance of an existing DOM element.



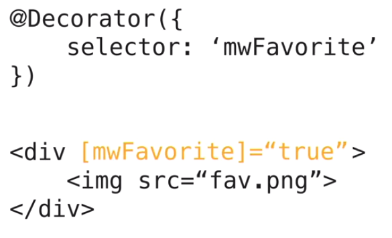
Since directives do not have a template, they are something you can create with the intent of applying them to an existing element, or in some cases a template element to change that element in some way.



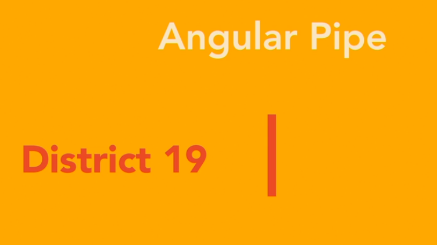
Like a component, a directive gets configured with a selector that Angular will use to find a match and apply the directive. You apply a directive in different ways. You can write an attribute on an element that matches your selector.



Or you can use the template syntax to add a directive in an assignment statement. In addition to creating your own directives, Angular comes with a few directives out of the box to handle common web app constructs, like conditionally rendering elements based on some expression, looping out items to render, or even for things like router links.



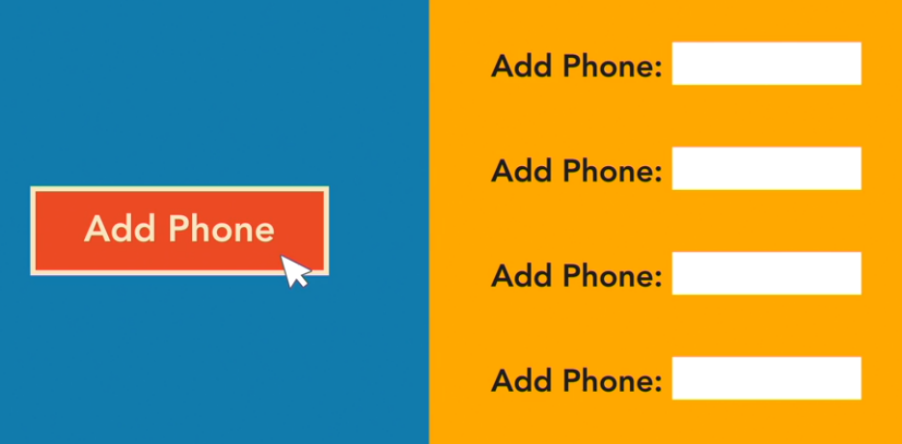
Another tool in the Angular toolbox to display content is the **pipe**. A pipe takes end data, like a string or an array, and runs some logic to transform it to a new output. Angular comes with some common pipes, like date in upper case and lower case. You can also write your own pipes to handle custom scenarios that fit your application needs. Pipes are a great way to change data in a reusable way without having to embed the transform logic within component classes, and without having to modify the data just for display purposes.



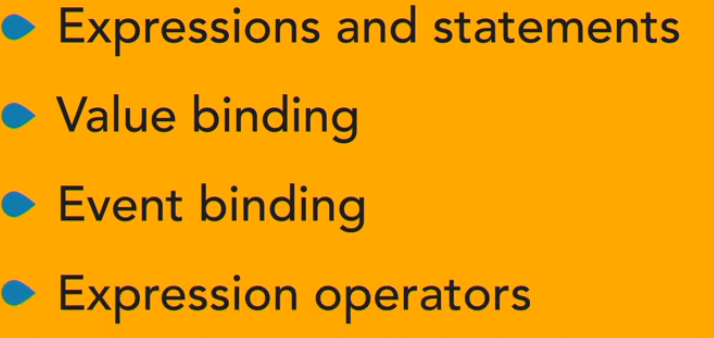
- The Angular framework is much more than just an engine for rendering markup templates through components. One of its strengths is the ease of binding data to views and working with data in those views. The most common way of displaying data in a view template is via interpolation, where you use a set of curly braces around a component property to tell Angular to render the content of that property. You can also use directives, both built-in Angular framework ones, and ones you create to help display data.



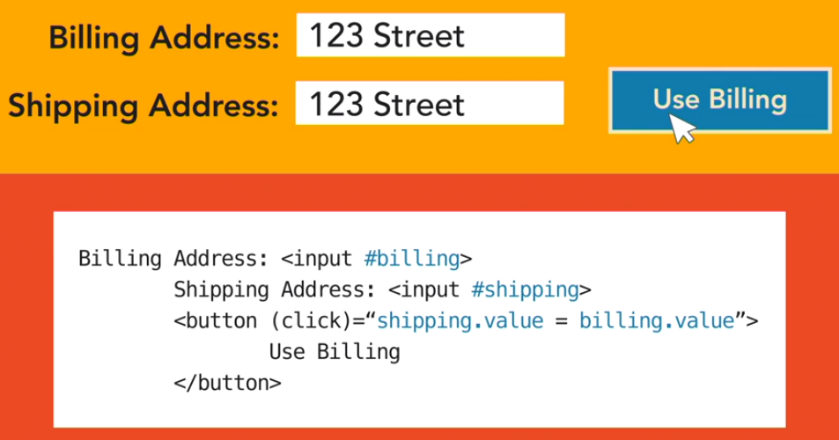
Directives give you the power Client side to add logic to your views like how you would on the Server side. The template syntax in Angular is robust and allows you to accomplish a lot of things when it comes to working with data in your views. You can wire up click events to DOM elements that modify data that you've displayed elsewhere, and Angular will handle the update of that data visually. There are many elements to the template syntax.



In addition to interpolation and built-in directives, the syntax has constructs and patterns such as template expressions and statements, a binding syntax for property, attribute, class and style bindings, event binding, and template expression operators.

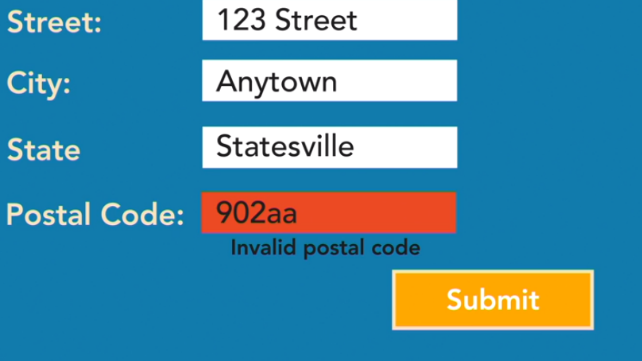


You can also create and use local template variables created in markup using the **hash** to get a reference to the element, and then use that from any sibling or child element in the view.

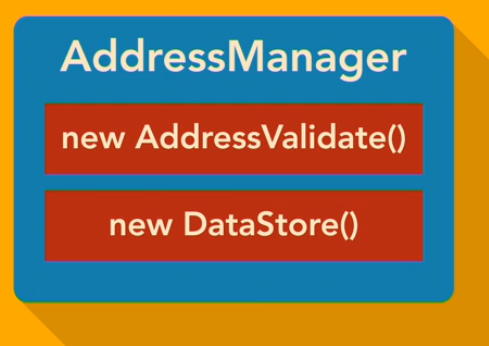


This allows you to wire up simple interactions or display related data from within your markup, without needing to write any script code.

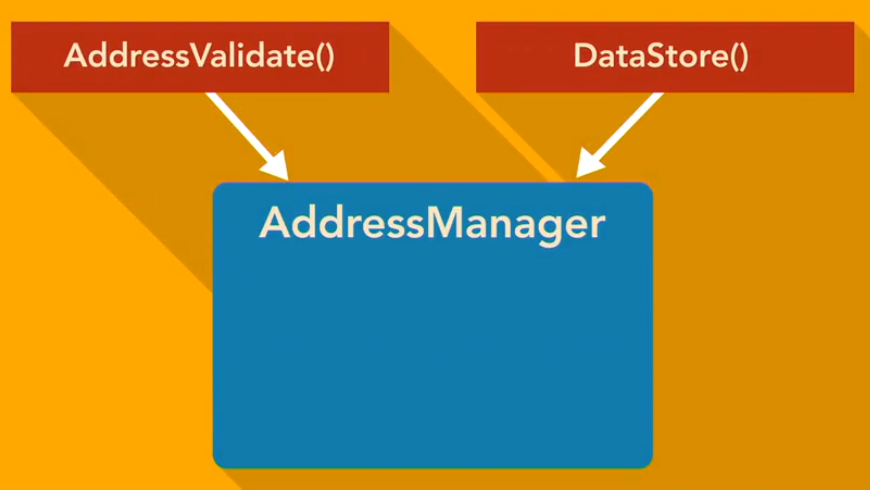
And when it comes to collecting data from the user, Angular has a form modular loaded with directives and services for helping you build html forms. It provides things like data binding for both setting and getting data, change tracking, validation, and error handling.



Angular’s most powerful features as a framework, is how it brings **dependency injection** to JavaSript. **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.

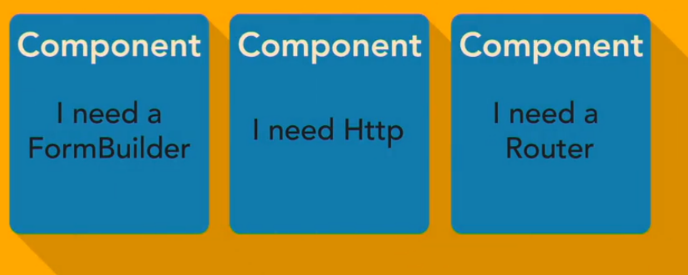


DI allows you to write decoupled code that is easier to unit test and to work with.



In the Angular world, this allows you to write these modular components and even services within your applications and simply tell Angular what you want to use and where want to use them.

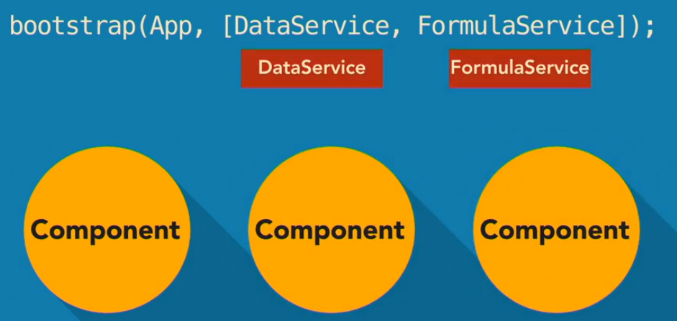
Angular will handle constructing instances of those and sending them to your code where needed. The most common place you use DI is in your class constructors.



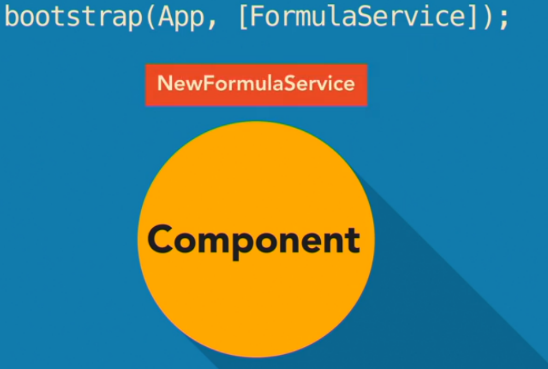
So, constructors for components, directives, pipes, and services you write can leverage the way Angular does DI. You can simply declare types on your constructor parameters with some help from TypeScript, and Angular will interpret that and make sure you receive an instance of that type when your constructors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



You can also leverage **DI** through things like component metadata properties for directives and providers. You can even do some DI at the bootstrap phase of an Angular app, setting up your dependency graph when your app starts up and getting that delivered through all aspects of your app.

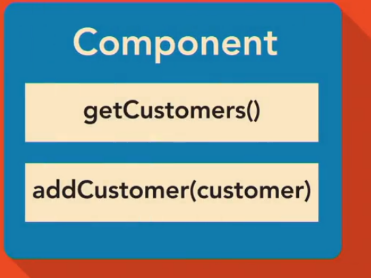


And one of the powers of DI is the ability to replace a dependency at any phase of application code. Angular has support for this. So, you can even do things like set up a dependency for a service at bootstrap and then replace that dependency with a different version of it for a specific component.

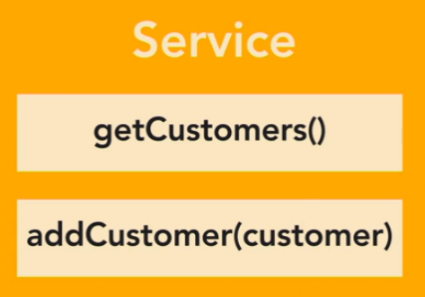


Angular services:

Services in Angular are more of an implied pattern. There is nothing specific in the Angular framework that defines some code as a service. We typically refer to a JavaScript class or function that we have written to encapsulate some logic as a service for our application. When you write your components in Angular, it preferred to craft the classes for these in a way that the class logic only consists of brokering data to and from the view and adding functionality to the view.



So, if you don't put application business logic in your components and directives, then where do you put it? The answer is services.



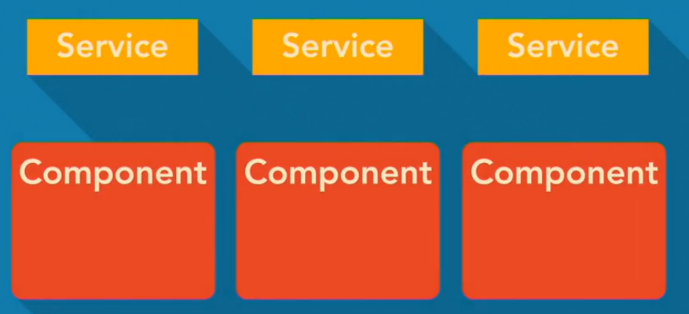
So, let's say you had a component that needs to display a media item. You can write a JavaScript class that handles finding the record data and returning it as an object.



This would be a service. And then, using Angular's DI framework, you can specify that your media item component is going to use your service. And from within your component logic, you can request the media item record from your service and make it available to your view template.



These services that you write can also leverage DI. So you can create constructors and specify parameter types with the help of some TypeScript, and Angular will provide your service instance with the appropriate dependencies. There is nothing in the services that you write that make them any sort of Angular-type service. They are just plain old JavaScript logic code.

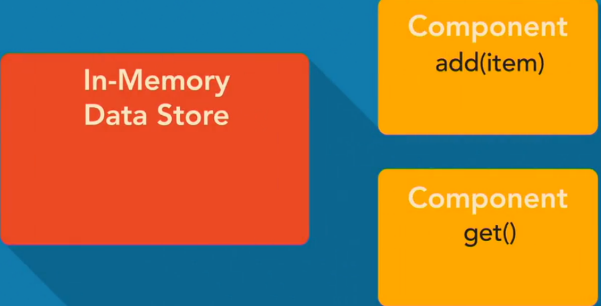


And with the power of DI, they are given other code to work with without needing to know if that code is Angular-specific code. So when you leverage writing services in your app code, you get the benefit of writing JavaScript logic that is in no way coupled to the Angular framework.

This is one of the nice advantages of Angular. The framework is designed to use your code. This allows you to write modular decoupled code that is easier to maintain and reuse.

Data Persistance:

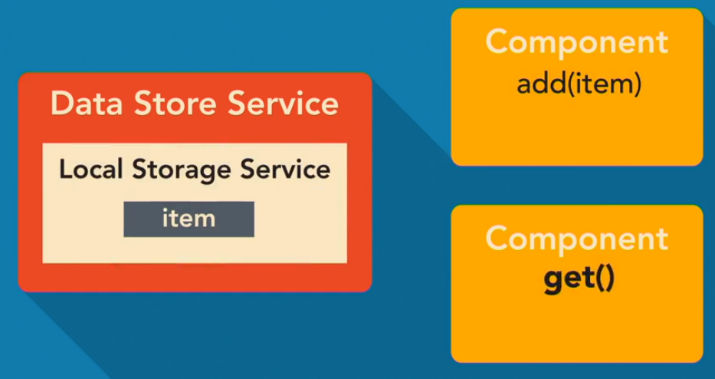
When you built client-side apps, your intent is typically to work with some kind of data in a read-write fashion. So you end up needing a way to pull and process data from Javascript in the client. This is handled in a couple of different ways with Angular, depending on the source of your data. If you are only concerned with storing data for the time in which the user is using your application, you could store that data in memory.



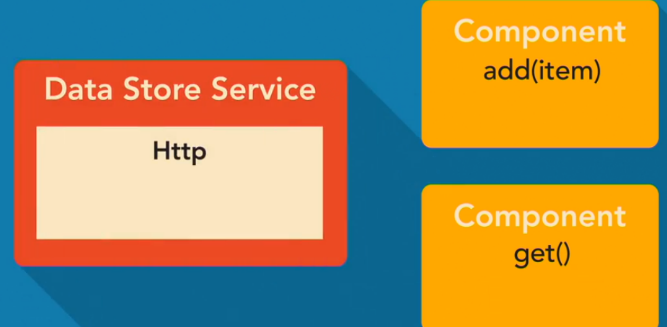
Angular's dependency injection makes this trivial to do. You can create a Javascript class or object to store your data, provide it to your app as something that can be injected in, and then do constructor injection where needed to bring in the instance of that object.

From there, you can do read-write updates to that object and have the changes available throughout.

If you were looking to work with data stored in browser storage, such as local storage, you will find the need to write your own Javascript code to do so, and then use it with the services pattern, leveraging Angular's dependency injection to work with it throughout your application.

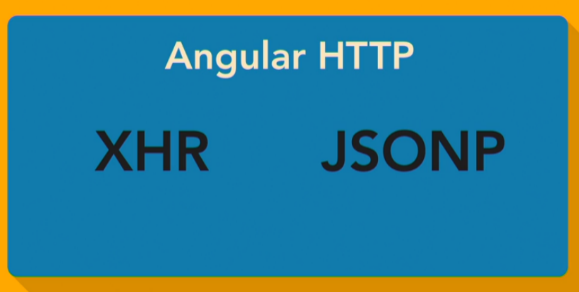


If you are looking to work with data from the server via an API, Angular has some built in framework stuff to help you with that. One way you can process data to and from an API is by leveraging the HTTP protocol.



You can do this in two ways. One, by using the **XML HTTP request** or **XHR** and the other by using **JSONP**.

Angular provides an HTTP module in the framework for abstracting out working with the way XHR and JSONP calls are done via client script.

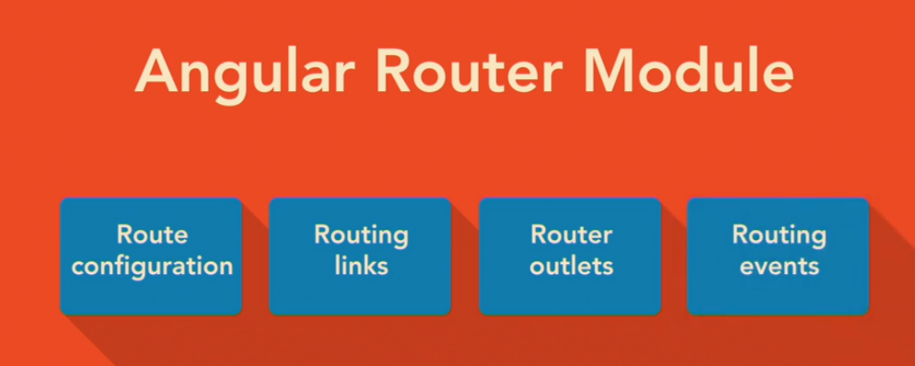


So you can do things like make **GET and POST** calls that work with JSON data as simple as **passing a URL** and a Javascript object to an Angular HTTP function and subscribing to the results.

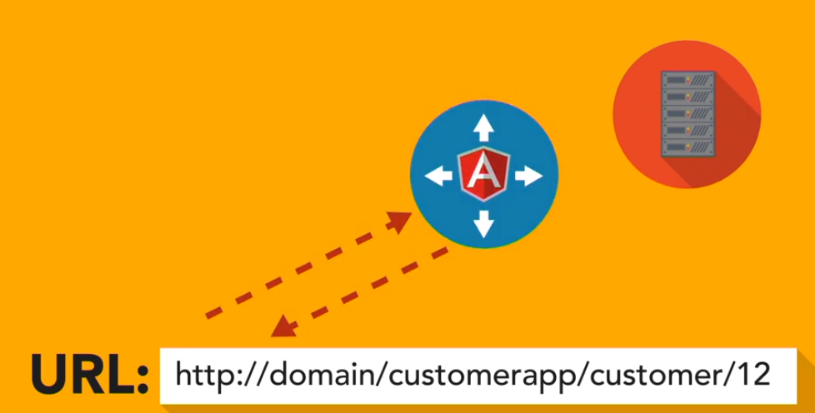
Routing:

Client-side web applications can take on many shapes, from small widgets, to interactive forms, to full-blown multi tool experiences. When you start needing to build a client-side solution that supports tool navigation, like, say, a customer management system that allows a user to view, edit, and create customers, and the same for orders, you need some sort of routing solution. A server delivered solution allows you to handle routing based on unique URL requests to the server.

But on the client side, you want to be able to not send URL requests to the server, and instead handle them in the client, adjusting the UI, and the data display accordingly. Angular provides a router module out of the box for doing just that. It supports configuring route pass to components, route params, so you can have variables in the URL, a directive for working with links that do routing, a directive for specifying where in a template the routed component will display, creating child routes, and routing lifecycle hooks for responding to routing events.



The Angular router also handles all the navigation interception. It will interpret a URL request and run it through its route logic to find a match and alter the component tree accordingly. It also handles history state, altering the way a browser handles it by default, so that back and forward actions by the user will result in angular route changes. Using the router gives your apps a familiar feel to the standard client server website experience users have grown accustomed to, but it also provides you with a way to load different components through a set of configuration instructions and links, rather than a bunch of conditional logic scripting.



NgModule and the route module

Decorator:

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



Typescript provides support for decorators through it's transpiler. The syntax for using a decorator is the at symbol, followed by the decorator name, and then a pair of parentheses. What you put in the parentheses depends upon the decorator. 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** 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, you need to decorate it with the **NgModule** decorator.

Angular exports the NgModule decorator from its core scoped package. To use the decorator from that package, you need to import it using the ES2015 module loading sytax that TypeScript supports. You start with the keyword import, followed by a space and a pair of curly braces. Inside the curly braces, you list the types that you want to import. We want to import the NgModule decorator, so I will put that inside of the curly braces. You can import more than one type from a module by adding a comma after each.

So for the string of the module name, you put the module name for the Angular core module in single quotes and you end the import statement with a semi-colon. Now that we've imported the NgModule decorator, we can use it. When you use it in code, you prefix it with the at symbol, then NgModule and a pair of parentheses. You don't put a semi-colon after this expression because it's not a statement but rather a piece of code that will be applied to the class or, in some cases, the property that it comes before.

So from here, you need to follow it up with the class definition for the Angular module. Since this is going to be our app root module, let's name it AppModule. So we type class, space, AppModule, and follow that up with a pair of curly braces. Since we are building this module in its own file, and will want to import it in another file, 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.

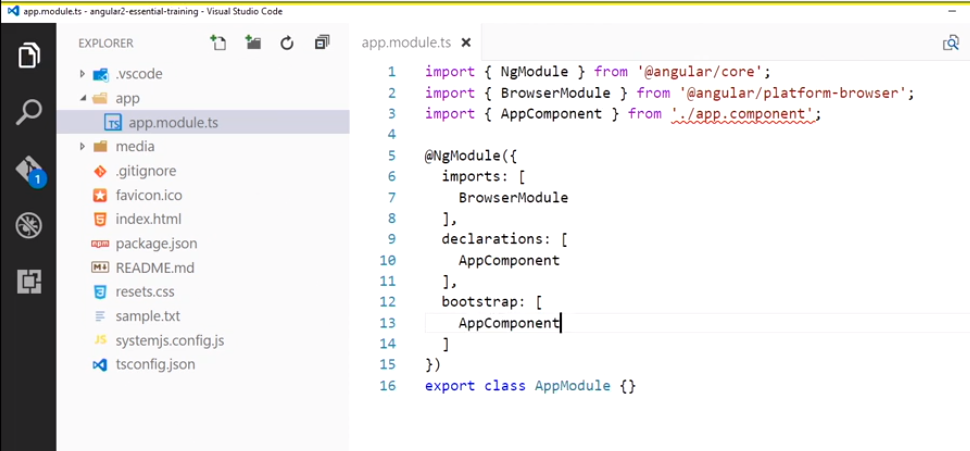
The NgModule decorator takes in an object with some known properties to configure the class you decorate as an Angular module. These properties are known as metadata. I'm going to set this object literal up with some properties here and then I will discuss what each are used for. So for the Angular root module, I am making use of the imports, declarations and Bootstrap metadata properties. All of these can be set up as an array. The import property is used to bring in other Angular modules that your 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. The **Bootstrap** 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. Let's fill these out with some types. Since we are building a browser-based app, we will want to make use of the browser module that the Angular platform has available.

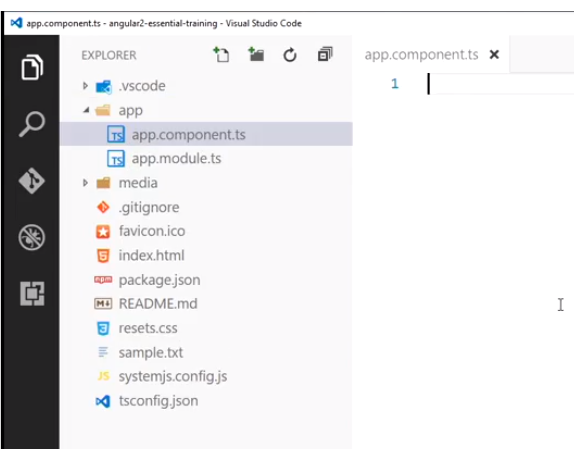
The browser module contains core directives, pipes and more for working with the DOM and can be found in the platform browser scoped package. So we need an import statement for that. And then we can add the browser module to the array for the import metadata property. Our app is going to need a starting component. I will cover creating that in the next video. But for now, let's add some code here for it.

The 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. So we can write the 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. That will be a dot, slash, followed by the name of the file without the extension.

The extension is not needed due to the way the project is configured for system js module loading. Now that's outside the scope of this course so I won't dive into that in detail. But from here we want to 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 as well. Since this app module is being used as the root module, Angular will use the AppComponent as a target for Bootstrapping the app.



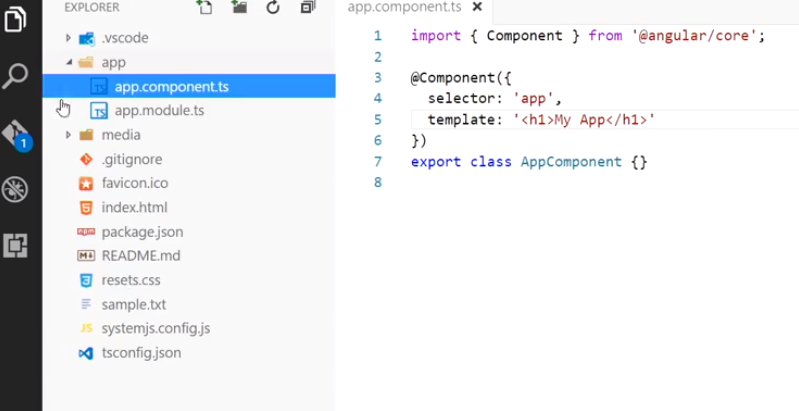
- Let's build the first component, the app component, in a file named app.component.ts, inside the app directory.



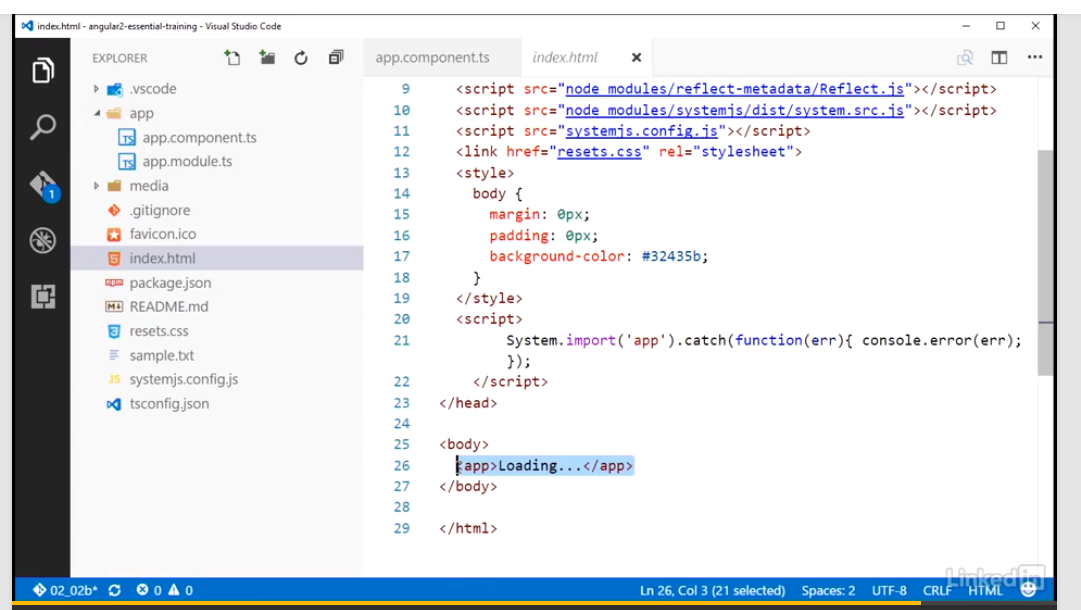
To build an Angular component, you need to use the component decorator on a class. The component decorator comes from the core scope package in Angular. So, we can add an import statement for the component decorator from the Angular core scope package. And then we can call the decorator, using the app component syntax with the parentheses.

Then, we need to export a class for the component. We will name this class AppComponent. So, the class definition reads as: export class AppComponent, and a pair of curly braces. Now, just like the NG module decorator, the component decorator takes in a metadata object, with some known properties, to configure a class you decorate as an angular component. To decorate a component, you need to provide two metadata properties at a minimum.

Selector, and template, or template url. We will pass in an object literal, inside the component decorator parentheses. And for now, we will set the selector property to the stringed value of app. And the template property to a string with an HTML H1 tag. The selector property is what Angular will use to locate a custom HTML element, and apply the component too.



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. You will see both in action in the next video, when I cover how to kick an Angular application off with the bootstrap call.

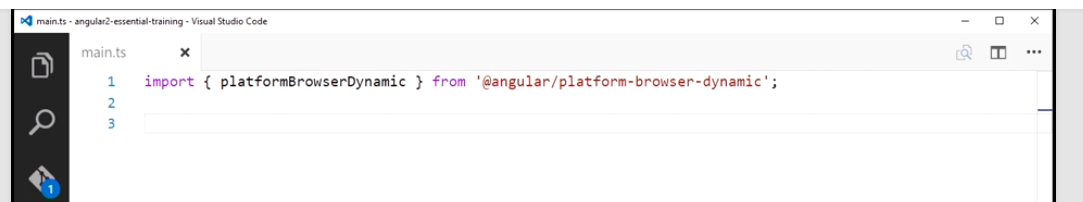


Bootstraping the module for the browser

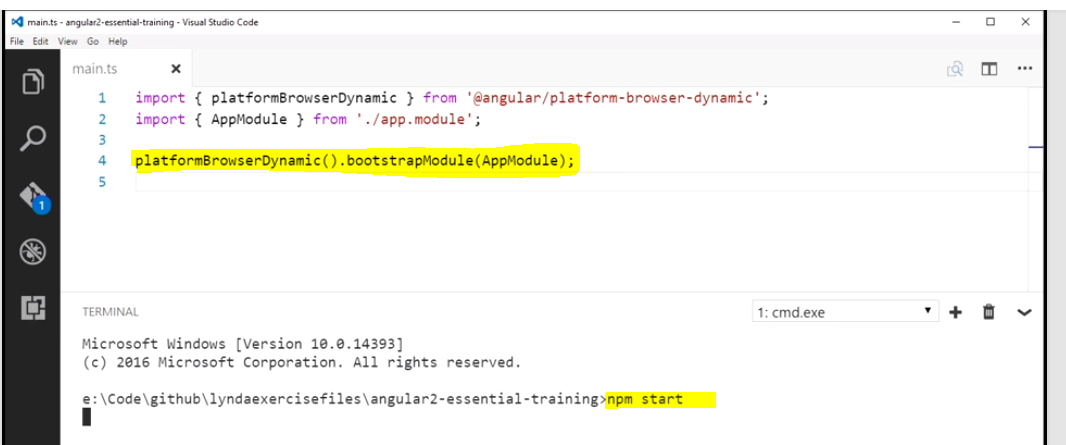
With an Angular root module and a starting component created, the next step to getting the foundation of an Angular app up and running is the code to bootstrap the module. We can add a file named main.ts to the app folder and put the bootstrap logic in that file. Angular has support for running on multiple 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.

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. That is the function you will use to bootstrap your root module on the platform.

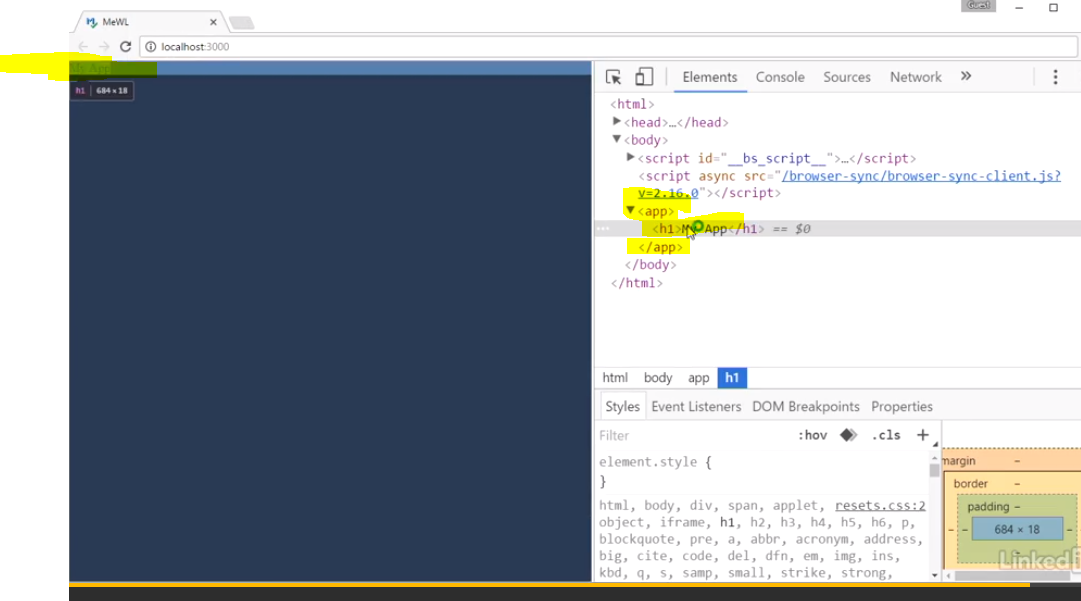
Note that earlier we were importing class types, and here we are importing a function. The module-loading syntax supports importing all kinds of exported things, from class types and functions to constants, variables, and even JSON file data. Okay, so now we can make the call to the platformBrowserDynamic function which will return an instance of a platform object. That platform object has a method named bootstrapModule. So off of that platformBrowserDynamic call, we can call bootstrapModule.



This function is expecting a root module and we have one already created from earlier named AppModule. So we can go up and write an import statement to load that type into this file, and then we can pass the AppModule type into the bootstrapModule function call. And with that, we have 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.



I'll do that from within Visual Studio code here. And we can run the npm start command to kick off the TypeScript build and web server and we'll watch for changes. And over in the browser, we can see the content from the app component template is displayed.



Building Media Watch List App – Client-side app