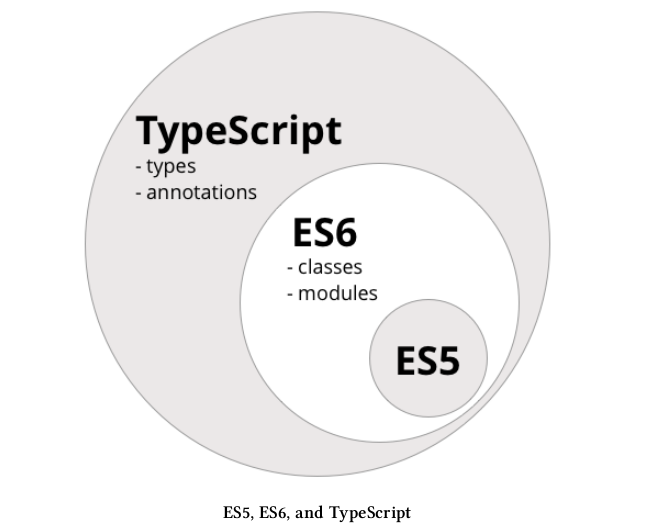
**Steps To Learn Angular:**

**Type Script**

**Why to To Learn?**

Angular built in TypeScript. TypeScript is a superset of ES6.



**What is ES5?**

Regular javascript. ES6 is the next version of ES5.

**How TypeScript Work?** TypeScript uses transcompiler/transpiler that takes TypeScript code as input and output ES5 code that nearly all browsers understand.

Two major ES6-to-ES5 transpilers: traceur by Google and babel by javascript community. TypeScript is an official collaboration between Microsoft and Google.

**Command to Install TypeScript:** sudo npm install -g typescript

**REPL mode:** sudo npm install -g tsun

Five big improvements that TypeScript bring over ES5:

* types
* classes
* decorators
* imports
* language utilities

**Types:** We still use var but now we can optionally provide the variable type.

Example: var fullName: string;

**Built-in types:** string, number, boolean, Array, enum, any and void

Two ways to define array.

Example:

var jobs: Array<string> = [‘apple’, ‘dell’, ‘hp’];

var jobs: string[] = [‘apple’, ‘dell’, ‘hp’];

Enums work by naming numeric values.

Example:

enum Role {Employee, Manager, Admin};

var role: Role = Role.Employee;

The default initial value for an enum is 0

Instead of Employee being 0, it can be 4 by this way.

enum Role {Employee = 4, Manager, Admin};

In this case Manager and Admin will be set to 5 and 6 respectively.

We can even set individual values.

enum Role {Employee = 3, Manager = 5, Admin = 9};

We can also look up the name of a given enum by using its value:

enum Role {Employee, Manager, Admin};

console.log(‘Role: ‘, Role[1]);

any is the default if we omit typing for a given variable.

Example:

var something: any = ‘as string’;

something = 1;

something = [1, 3, 6];

Using void means there’s no type expected. This is usually in functions with no return value. A void value is also a valid any value.

Example:

function setName(name: string): void {

this.fullName = name;

}

**Classes:** To define a class we use the class keyword. Classes may have properties, methods and constructors.

**Methods:** Methods are functions that run in context of an object. To call a method on an object, we first have to have an instance of that object. If a method does not declare an explicit return type, it’s returned value is any type by default. When we return void, no need to have explicit return statement.

**Constructors:** Special method that is executed when a new instance of the class is being created. Constructor methods must be named constructor. They can optionally take parameters but they can’t return any values.

Example: class Vehicle {

}

var v = new Vehicle();

This is same as:

class Vehicle {

constructor() {

}

}

var v = new Vehicle();

So in TypeScript we can define a class this way.

Class Person {

first\_name: string;

last\_name: string;

age: number;

constructor(first\_name: string, last\_name: string, age: number) {

this.first\_name = first\_name;

this.last\_name = last\_name;

this.age = age;

}

greet() {

console.log(“Hello “ , this.first\_name);

}

ageInYears(years: number): number {

return this.age + years;

}

}

var p: Person = new Person(‘Felipe’, ‘Coury’, 45);

p.greet();

**Inheritance:** It is achieved by extends keyword.

Example:

class Report {

data: Array<string>;

constructor(data: Array<string>) {

this.data = data;

}

}

class TabbedReport extends Report {

headers: Array<string>;

constructor(headers: string[], values: string[]) {

super(values);

this.headers = headers;

}

}

**Utilities:** Two important ones are

* fat arrow function syntax
* template strings

Fat arrow => functions are a shorthand notation for writing functions.

Example: var data: string[] = [‘Alice Green’, ‘Paul Pfifer’, ‘Louis Blakenship’];

data.forEach((line) => console.log(line));

parentheses are optional when there is only one parameter. It can be used as an expression. Its important feature is that it shares the same this as the surrounding code. This is different than function in javascript.

In javascript:

var nate = {

var name: “Nate”,

printGuitars: function(guitars) {

var self = this;

guitars.forEach(function(g) {

constructor.log(self.name + “ plays a “ + g);

});

}

};

In TypeScript:

var nate = {

var name: “Nate”,

printGuitars: function(guitars) {

var self = this;

guitars.forEach((g) => {

constructor.log(this.name + “ plays a “ + g);

});

}

};

Two great features of template strings are

1. Variable within strings(without being forced to concatenate with +) and

2. Multi-line strings

Examples:

var first\_name = “Nate”;

var last\_name = “Murray”;

var greeting = `Hello ${first\_name} ${last\_name}`;

String interpolation is done by only backticks. Multi-line strings is defined by:

var template = `

<div>

<p>This is great job</p>

<div>

`

There are a variety of other features in TypeScript/ES6 such as:

* Interface
* Generics
* Importing and Exporting Modules
* Decorators
* Destructuring

These will be discussed later. Let’s get back to Angular.

**How Angular Works**

The first thing is that an Angular application is made up of Components. It’s a new way to teach the browser new tags. It’s analogous to directives in AngularJS. So an Angular application is nothing more than a tree of Components. At the root of that tree, the top level Component is the application itself. And that’s what the browser will render when “booting” the app. It renders other Components.

**Component**

Each component is composed of three parts.

* Component Decorator
* A View
* A Controller

Example:

@Component({

selector: ‘app-root’,

template: `

<div>Welcome to Angular</div>

`

})

export class AppComponent {

}

The @Component is called decorator. It adds metadata to the class that follow it(AppComponent). We define metadata by java script object syntax. Which has few properties to demonstrate Component behavior. Here selector value defines the way to use this Component as tag in html and template value will be rendered in that selector. Finally we define the AppComponet class with export keyword so that Component will be available outside of this file.

To bootstrap this application => bootstrap(AppComponent)

**Data Binding**

In AngularJS the default data binding is two-way data binding. In Angular the general flow of data is unidirectional, that means it flows data only in one direction not two. Now it has the advantage of increasing performance, making it easier to debug. In Angular we have four forms of data binding.

* String Interpolation
* Property Binding
* Event Binding
* Two-way Binding

**String Interpolation**

Example: <div>{{name}}</div>

**Property Binding**

In this type we bind properties of an element/directive/component. So data flows in into element/directive/component is called property binding.

Examples:

<input [value]=”name” [ngClass]=”{red: true}”>

the left-hand side square bracket [value] is a property defined by Angular2 for input element says we want to use the *value* input and the right-hand side “name” says that we want to send the value of the expression *name.*

In case of directive also can but does not have to take input. The difference between directives and element properties by ‘ng’ prefix.

@Component({

selector: ‘my-property-binding’,

template: `

<p>Hey {{name}} and I am {{age}} years old!</p>

`,

inputs: [‘name: myName’]

})

export class PropertyBindingComponent {

name = ‘’;

@Input(‘myAge’) age = 20;

}

So we can use *my-property-binding* this way.

<my-property-binding [myName]=”’Nazmul’” [myAge]=”27”></my-property-binding>

In this case we alias *name* and *age* properties by *myName* and *myAge* respectively. If we did not do that

<my-property-binding [name]=”’Nazmul’” [age]=”27”></my-property-binding>

So custom property binding can be done by two ways, which are passing as metadata in *inputs* properties and using @Input decorator in class field. Using @input decorator is preferable. So we can be sure which properties should be filled before initialization of Component.

**Event Binding**

In this type binding, we send data out of elements via outputs.

Examples:

@Component({

selector: ‘product-list’,

templateUrl: ‘./product-list.component.html’

})

export class ProductListComponent {

@Output() onProductSelected: EventEmitter<Product>;

}

We define *onProductSelected* event binding using @Output decorator. So we can use *product-list* this way.

<product-list (onProductSelected)=”productWasSelected($event)”></product-list>

We are saying that we want to listen to the *onProductSelected* output from the *ProductListComponent* component. That is:

(onProductSelected), the left-hand side is the name of the output we want to “listen” on. The right-hand side is the function *productWasSelected* we want to call when something new is sent to the output (onProductSelected). $event is a special variable here that represents the thing emitted on.

**Emitting Custom Events**

To create a custom output event we do three things:

* Specify outputs in the @Component configuration
* Attach an EventEmitter to the output property
* Emit an event from the EventEmitter, at the right time

An EventEmitter does two things.

* Maintain a list of subscribers and
* publish events to them.

Example:

@Component({

selector: ‘my-hobby’,

template: `

<h3>My hobbies are: </h4>

<input (keyup)=”onHobbiesChanged(hobbies.value)” #hobbies>

`,

outputs: [‘hobbiesChanged’]

})

export class MyHobbyComponent {

hobbiesChanged = new EventEmitter<string>();

onHobbiesChanged(hobbies: string) {

this.hobbiesChanged.emit(hobbies);

}

}

We define #hobbies as local variable. So we call the *onHobbiesChanged* method after inputting that will emit *hobbiesChanged*.

**Two-way Binding**

It’s bidirectional.

Example:

<input [(ngModel)]=”name”>

<p>Your Name: {{name}}</p>

**Directives**

Attribute directives are not necessarily enclosed any brackets or parentheses. We use bracket in case of input for binding. So directives are just instructions which we add to an element to change in a some way.

**Custom Directives**

To do that we have to create a class and it’s naming convention is same as component. It also needs to define a decorator in this case @Directive which also takes java script object as argument to set metadata like @Component decorator. Unlike component decorator, we define the name of the selector in selector property surrounded with square bracket because of CSS matching rules.

Example:



Here we define two instance variables in constructor as argument and this is shorthand. ElementRef is used to reference the element that use this custom attribute. To use myHighlight as input, we reference myHighlight attribute value to inputs property in directive decorator.

**Structural Directives**

**NgIf**

Example:

<div class=”container”>

<div \*ngIf=myVar == ‘A’”>Var is A</div>

</div>

**NgSwitch**

Example:

<div class="container" [ngSwitch]="myVar">

<div \*ngSwitchCase="'A'">Var is A</div>

<div \*ngSwitchCase="'B'">Var is B</div>

<div \*ngSwitchCase="'C'">Var is C</div>

<div \*ngSwitchDefault>Var is something else</div>

</div>

**NgFor**

Example: \*ngFor="let item of items"

**NgStyle**

With the NgStyle directive, we can set a given DOM element CSS properties from Angular expressions. The simplest way to use this directive is by doing [style.<cssproperty>]="value"

Example:

<div [style.background-color]="'yellow'">

Uses fixed yellow background

</div>

**NgClass**

Allows us to dynamically set and change the CSS classes for a given DOM element. The way to use this directive is by passing in an object literal. The object is expected to have the keys as the class names and the values should be a truthy/falsy value to indicate whether the class should be applied or not.

.bordered {

border: 1px dashed black;

background-color: #eeeeee;

}

<div [ngClass]="{bordered: false}">This is never bordered</div>

<div [ngClass]="{bordered: true}">This is always bordered</div>

### **NgNonBindable**

### We use ngNonBindable when we want tell Angular not to compile or bind a particular section of our page.

### Example: <div ngNonBindable>{{content}}</div>

**Forms in Angular**

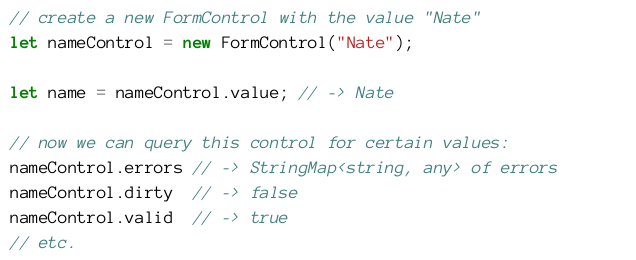
Angular has tools to develop forms. These are:

* FormControl – encapsulate the inputs in our forms and give us objects to work with them
* Validators – validate inputs
* Observers – watch our form for changes and respond accordingly

The two fundamental objects in Angular forms are FormControl and FormGroup.

**FormControl**

It represents a single input field that encapsulate the field’s value and states such as being valid, dirty(changed) or errors.



Like many things in Angular, we have a class (FormControl, in this case) that we attach to the DOM with an attribute (formControl, in this case).

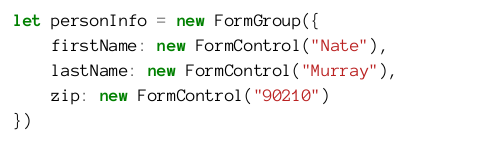


This will create a new FormControl object within the context of our form.

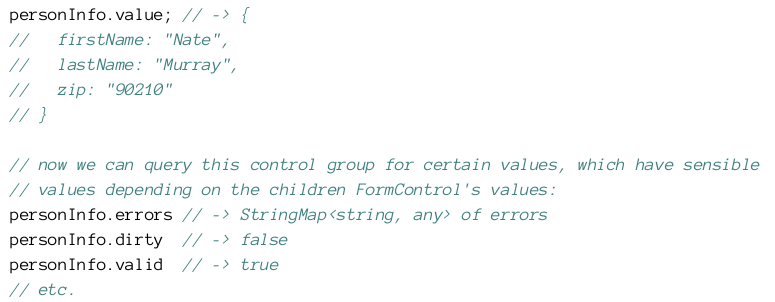
**FormGroup**

To manage multiple FormControls Angular provides FormGroup.

Example:



FormGroup and FormControl have a common ancestor AbstractControl.



There are two ways of using forms in Angular which are FormsModule and ReactiveFormsModule.

FormsModule gives us template driven directives such as:

* ngModel and
* NgForm

ReactiveFormsModule gives us directives like

* formControl and
* ngFormGroup etc

**FormsModule Example:**

****

FormsModule makes NgForm available to our view. It includes the form tag in its selector, that means NgForm will get automatically attached to any <form> tags in our view. Two important pieces of functionality that NgForm gives us:

1. A FormGroup named ngForm

2. A (ngSubmit) output

So ngForm is FormGroup type object. In that example, we specify ngModel with no attribute value for one-way data binding. A FormControl is created with the name sku(because of the name attribute on the input tag) and automatically added to the parent FormGroup.

**NgModel vs. ngModel: what’s the difference?**

PascalCase, like NgModel referring to class and CamalCase(ngModel) referring to object.

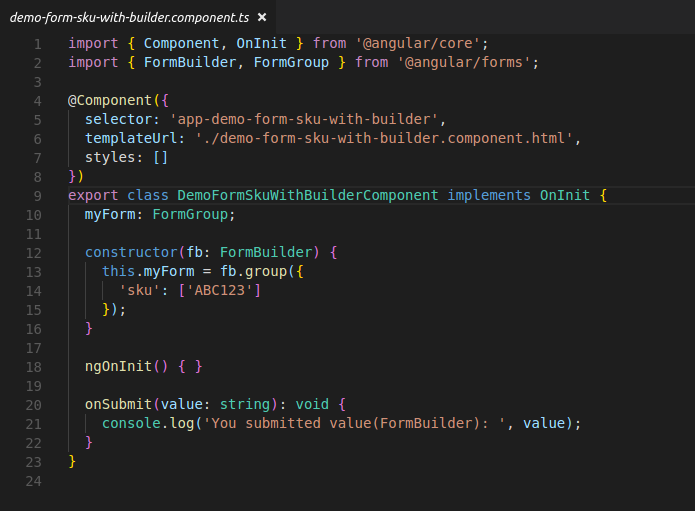
NgModel is used for view and binds the instance variable of input tag, whereas FormControl is used for representing the data and validations in our form.

**Reactive Forms with FormBuilder**

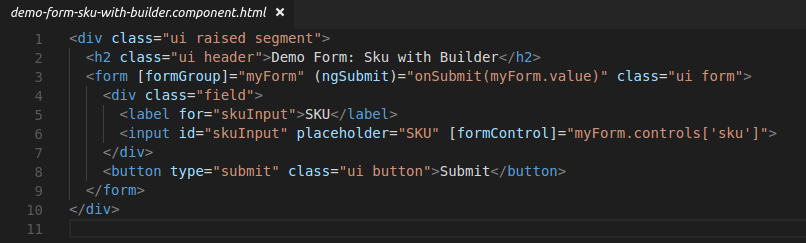
For this component we’re going to be using the formGroup and formControl directives. We have to inject FormBuilder by creating an argument in the constructor of our component class. It provides two main functions.

1. control – creates a new FormControl

2. group – creates a new FormGroup



FormGroup is created by calling fb.group()..group takes an object of key-value pairs that specify the FormControls in this group. In this case, we’re setting up one control sku with default value(“ABC123”). To use myForm in <form> tag, we will do [formGroup]=”myForm” in that tag.



To use myForm on onSubmit method, we bind existing FormControl to input tag.

**Adding Validations**

Validators are provided by the Validators module and the simplest is Validators.required which simply says that the designated field is required or else the FormControl will be considered invalid. To use validators we need to do two things:

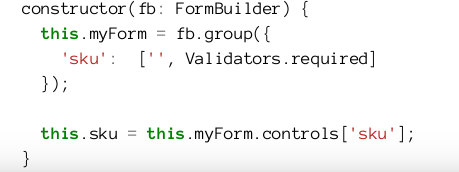
1. Assign a validator to the FormControl object

2. Check the status of the validator in the view and take action accordingly

To assign a validator to a FormControl object we simply pass it as the second argument to our FormControl constructor. There are two ways we can access the validation value in the view.

1. Explicitly assign the FormControl sku to an instance variable of the class – which is more verbose, but gives us easy access to the FormControl in the view.

2. Lookup the FormControl sku from the myForm in the view.



**Custom Validations**

Sometimes we need custom validations.

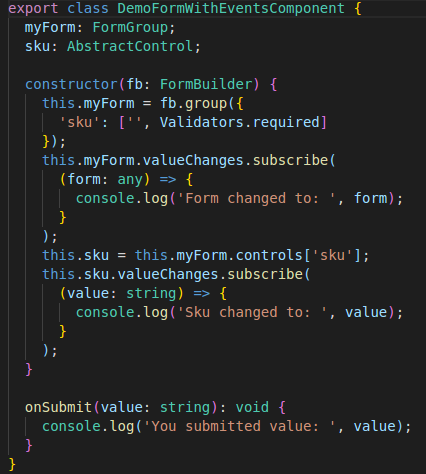
Example:



This validator will return an error code invalidSku if the input (the control.value) does not begin with 123. Validators.compose adds multiple validators to a single field.

**Watching For Changes**

Both FormGroup and FormControl have an EventEmitter that we can use to observe changes. EventEmitter is an Observable for watching for changes. First we get access to the EventEmitter by calling control.valueChanges, then we add an observer using the .subscribe method.

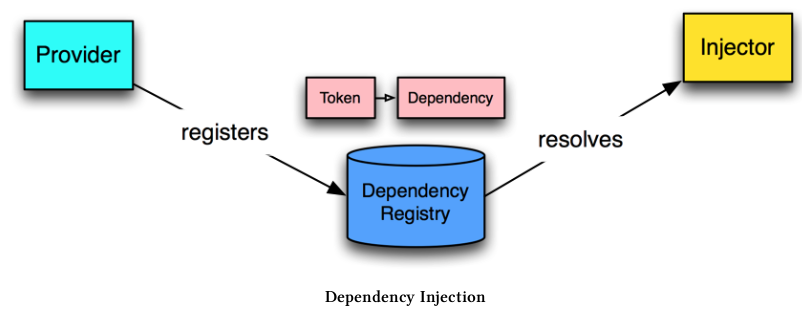


**Dependency Injection Parts**

To register a dependency we have to bind it to something that will identify that dependency. This identification is called the dependency token. For instance, if we want to register the URL of an API, we can use the string API\_URL as the token. Similarly, if we are registering a class, we can use the class itself as its token as we will see below.

Dependency injection in Angular has three pieces:

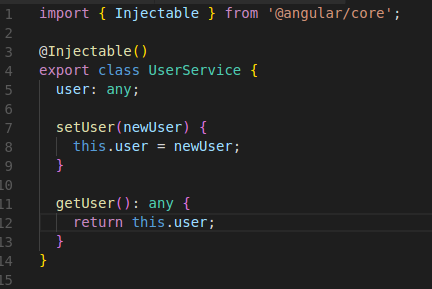
* Provider: maps a token to a list of dependencies. It tells Angualr how to create an object, given a token.
* Injector: holds a set of bindings and is responsible for resolving dependencies and injecting them when creating object.
* Dependency: that is what’s being injected



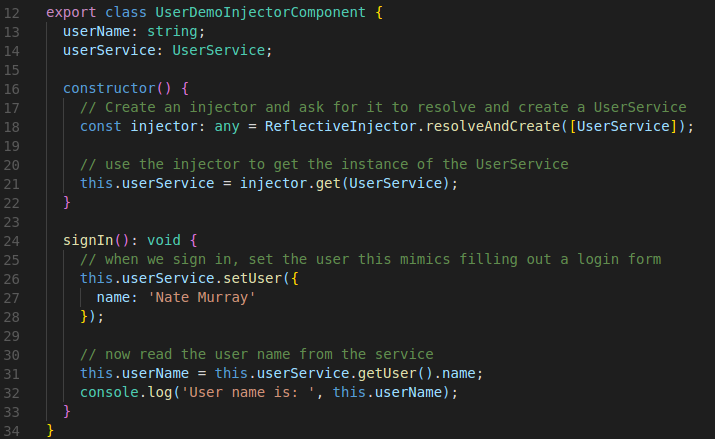
**Playing with an Injector**

Angular uses an injector to resolve a dependency and create the instance. This is done for us behind the scene. Let’s manually use the injector in out component to resolve and create a service.

Here is a basic UserService that stores the user object as a property.



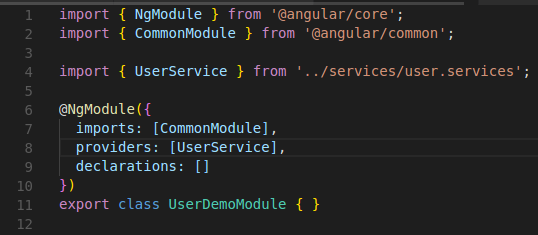
Now let’s implement this functionality in our component by using the injector directly.



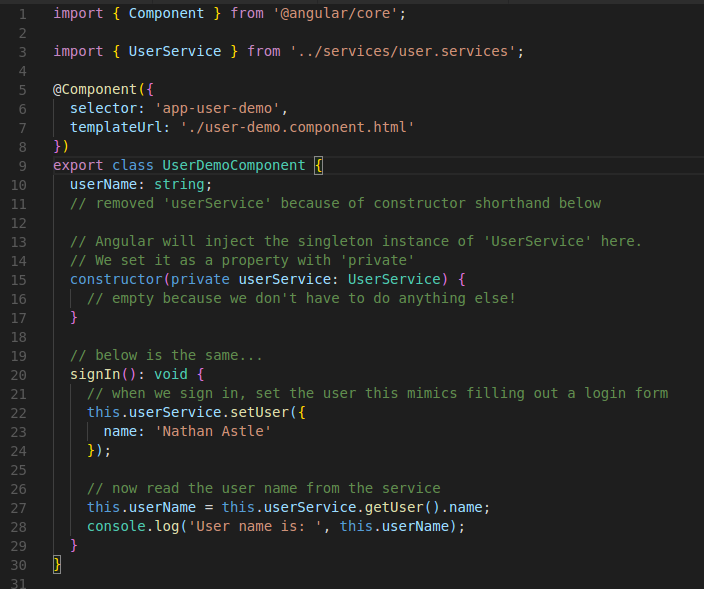
Here static method resolveAndCreate is responsible for creating a new injector. The parameter we pass in is an array with all the injectible things we want this new injector to know.

**Providing Dependencies with NgModule**

We use NgModule to register what we’ll inject–these are called providers and use decorators(generally on a constructor) to specify what we’re injecting. By doing these two steps Angular will manage creating the injector and resolving the dependencies.



Our UserService is injected as singleton across our app. Now we can inject UserService into our component like this:



Our constructor takes userService: UserService as argument. When this component is created on our page Angular will resolve and inject the UserService singleton. So every class that injects the UserService will receive the same singleton.

**Providers**

There are several ways we can configure resolving injected dependencies in Angular. For instance we can:

* Inject a (singleton) instance of a class(as we’ve seen)
* Inject a value
* Call any function and inject the return value of that function

**Using a Class**

providers: [UserService]

This tells Angular that we want to provide a singleton instance of UserService whenever UserService is injectd. But the class itself is actually shorthand notation for the following, equivalent configuration:

providers: [{ provide: UserService, useClass: UserService }]

The object configuration takes two keys. provide is the token that we use to identify the injection and the second useClass is how and what to inject. The token and the injected thing aren’t required to have the same name. As we’ve seen above, in this case the injector will create a singleton behind the scenes and return the same instance every time we inject it. Of course, the first time it is injected, the singleton hasn’t been instantiated yet, so when creating the userService instance for the first time, the DI system will trigger the class constructor method.

**Using a Value**

We can use DI is to provide a value, much like we might use a global constant. For instance, we might configure an API Endpoint URL depending on the environment.

providers: [{ provide: ‘API\_URL’, useValue: ‘<http://my.api.com/v1>’ }]

So the component class will be..

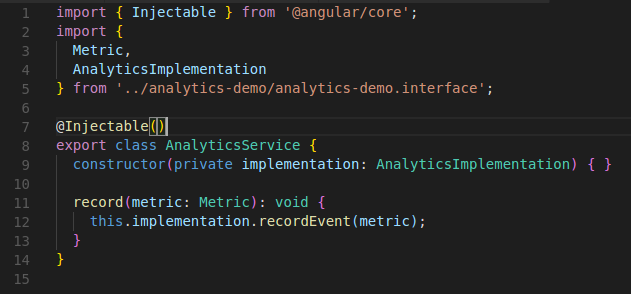
export class AnalyticsDemoComponent {

constructor(@Inject(‘API\_URL’) apiUrl: string) { }

}

If a service’s constructor requires argument, we have to implement providers using a factory which is a function that can return any object when injected.

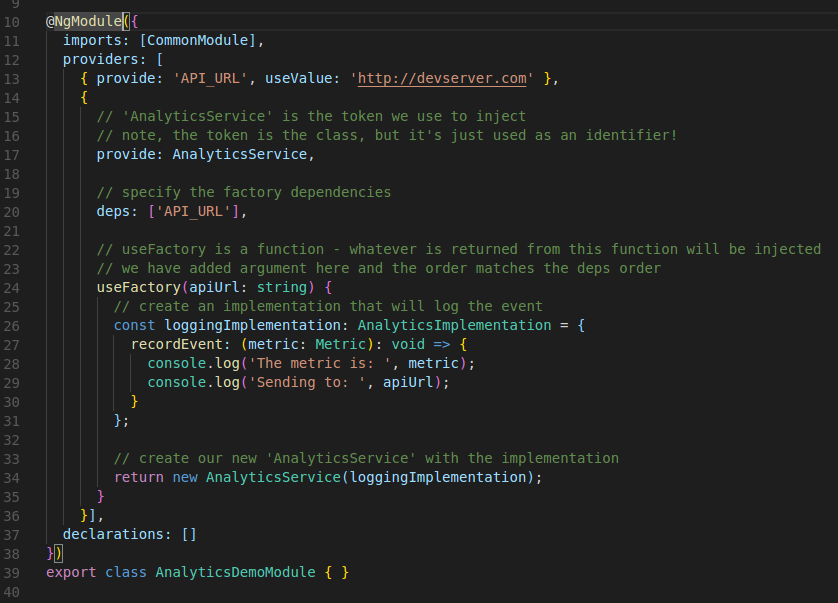
Example: Let’s see AnalyticsService..



Syntax of providers should be..

providers: [{ provide: token\_name, useFactory: () => … }]

So let’s configure the module..



useFactory takes a function and whatever this function returns will be injected. We are using the class AnalyticsService as the identifying token of what we are going to inject. This factory function takes an argument for factory dependency. So we added a new key:deps. deps is an array of injection tokens and these tokens will be resolved and passed as arguments to the factory function. Factory function must takes arguments in the order that matches the deps order.

**HTTP**

Angular comes with its own HTTP library which we can use to call out external APIs. Here HTTP requests are asynchronous. There are generally three approaches to dealing with asynchronous code:

1. Callbacks

2. Promises

3. Observables

In Angular, the preferred method of dealing with async code is using Observables.

**import from @angular/http**

In our app.module.ts we are going to import HttpModule which is a convenience collection of modules.

import { HttpModule } from ‘@angular/http’;

To make an HTTP request is straightforward: we call this.http.request and pass the URL to which we want to make a GET request. It returns an Observable. We can subscribe to changes(akin to using **then** from a **Promise**) using **subscribe**.

Example: http.request(‘<http://jsonplaceholder.typicode.com/posts/1>’)

.subscribe((res: Response) => { })

When our http.request returns(from the server) the stream will emit a Response object.

.subscribe can also handle failures and stream completion by passing a function to the second and third arguments respectively. While http.request can make any kind of request(POST, DELETE, GET, PUT etc.), http.get is a shorthand for GET request.

* http.put, http.post and http.patch take a URL and a body. In the body we pass json format by using JSON.stringify method, which takes an Object as argument.
* http.delete, http.get and http.head take only URL.

All of the http methods also take an optional last argument: RequestOptions. This object encapsulates:

* method
* headers
* body
* mode
* credentials
* cache
* url
* search

Let’s say we want to craft a GET request that uses a special X-API-TOKEN header. We can create a request with this header like so:



**Routing**

Routing means splitting the application into different areas usually based on rules that are derived from the current URL in the browser.

There are two modes of routing. Which are hash-based routing and HTML5 based routing. In Angular HTML5 is the default mode. Examples:

An about page, using hash-base is /#/about and using HTML5 base is /about

There are three main components that we use to configure routing in Angular:

* Routes describes the routes our application supports
* RouterOutlet is a “placeholder” component that shows Angular where to put the content of each route
* RouterLink directive is used to link to routes

In order to use the router in Angular, we import constants from the @angular/router package:

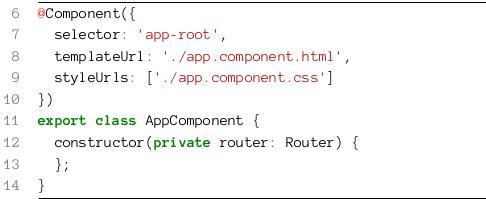
import { RouterModule, Routes } from ‘@angular/router’;

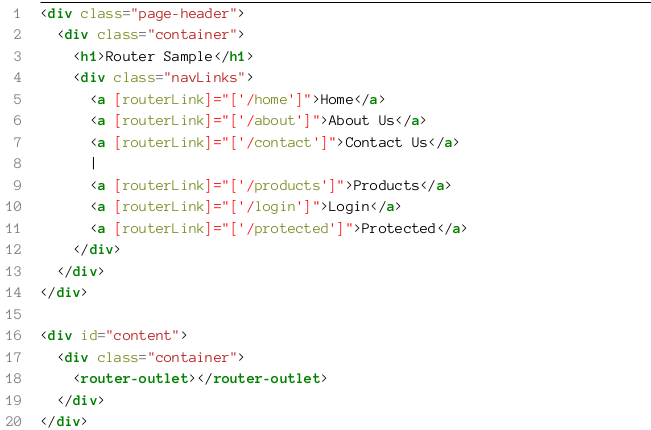
To define routes for our application, create a Routes configuration and then use RouterModule.forRoot(routes) to provide our application with the dependencies necessary to use the router. First, let’s look at the routes definitions:



path specifies the URL this route will handle, component is what ties a given route path to a component that will handle the route and the optional redirectTo is used to redirect a given path to an existing route.

In order to describe to Angular where in our page we want to render the contents for each route, we use the RouterOutlet directive.





When we visit /home, that’s where HomeComponent template will be rendered. The same happens for the other components. The left-hand side the [routerLind] that applies the directive to the current element(in our case a tags). On the right-hand side we have an array with the route path as the first element, like “[‘/home’]” or “[‘/about’]” that will indicate which route to navigate to when we click the element.

<base href=”/”> declares the base HTML tag. This tag is traditionally used to tell the browser where to look for images and other resources declared using relative paths. We can declare the application base path programmatically, when configuring our NgModule by using the APP\_BASE\_HREF provider.

{ provide: APP\_BASE\_HREF, useValue: ‘/’ }

The default routing strategy is PathLocationStrategy, which is what we call HTML5 routing. While using this strategy, routes are represented by regular paths, like /home or /contact. We can change the location strategy used for our application by binding the LocationStrategy class to a new, concrete strategy class which is HashLocationStrategy. If we were to refresh the page, instead of asking the server for the root URL which is what is being served, instead we’d be asking for /about or /contact. Because there is no known page at /about or /contact the server would return a 404. This hash based strategy the server understand as being the / path.

We can specify that a route takes a parameter by putting a colon : in front of the path segment like this:

*route*/:param

To add a parameter to our router configuration, we specify the route path like this:

const routes: Routes = [{ path: ‘product/:id’, component: ProductComponent }];

Then when we write the ProductComponent, we add the ActivatedRoute as one of the constructor arguments:

export class ProductComponent {

id: string;

constructor(private route: ActivatedRoute) {

route.params.subscribe(params => { this.id = params[‘id’]’ });

}

}

**Nested Routes**

We will start by describing the products route on the app.module.ts file:

const routes: Routes = [{ path: ‘products’, component: ProductComponent, children: childRoutes }];