**PRINCILPLES**

1 DO Not REPEAT YOURSELF(DRY)

2 SINGLE RESPONSIBILITY PRINCILPLE

**SERVICE**

A class with specific purpose

1 share Data

2 Implement application logic

3 External interaction  
NAming convention -- .service.ts

DI

1

**SOME TASKS NEEDS TO BE DONE.**

**1 Login**

**2 Registration form throught DB**

**3 Api call**

**4 CRUD APP**

**5 REDIRECTION**

**6MULTI STEP FORM**

**7VALIDATION**

**JS Push, Splice., findIndx, arrayPush, find**

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

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

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

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

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

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

//

JS QUESTION

**<https://github.com/lydiahallie/javascript-questions>**

**Installing the bootstrap**

<https://www.techiediaries.com/angular-bootstrap-ui/>

**npm install --save @ng-bootstrap/ng-bootstrap bootstrap font-awesome**

**Check the changes in the files**

**ng serve --open.**

**Generating the module and enabling the routing by default use**

**ng generate module product --routing**

**Generating the class**

**ng generate class product/models/lookup**

**ng generate class product/models/product**

**Generating the services**

**ng generate service product\services\product**

**NOw Generating the component**

**ng generate component product\list**

**//**

**Ng serve --open**

**Handling the wild card Routes**

**When we enter the routes that is not difined in that case we use this route. Here we redirect to the 404 component where we say that routes is not defined.**

**This routes has to be the very last routes in routes section**

**<https://prnt.sc/s2656e>**

**<https://prnt.sc/s266ns>**

**NOW FOR THE PROPER WORKING OF THE ROUTES**

**Topic 1**

**We define routes in the app-routing-module.ts**

**There we use the export and make the an array of the routes which we will display on the app. And import that in the app.module.ts for the proper working.**

**Topic 2**

**HANDLING INVALID ROUTES.**

**USING WILDCARD ROUTES.**

**And setting the default routes.**

**Handling the wild card Routes**

**When we enter the routes that is not difined in that case we use this route. Here we redirect to the 404 component where we say that routes is not defined.**

**This routes has to be the very last routes in routes section**

**<https://prnt.sc/s2656e>**

**<https://prnt.sc/s266ns>**

**TOPIC 3**

npm install -g typescript

**/\*\***

Description  
This method has the same functionality as the global parseInt() function:

Number.parseInt === parseInt; // true

and is part of ECMAScript 2015 (its purpose is modularization of globals).

**/\***

Worth noting is that you shouldn't use parseInt to convert a String into a Number. Use var n = Number('123'); instead, or Math.round()/Math.trunc() if you want to convert a floating point number into an integer. Only use parseInt when converting between numbers of different base.

semantically, you mostly do not want to parse a string for a number, you want to do a type conversion, therefore Number() makes more sense.

parseInt() is much slower to execute (not that it's noticeable).

parseInt() doesn't give an error or any warning as long as the first character is a number. parseInt('3asdfghhj') will simply return 3.

parseInt() coerces the input to a string without error or warning. parseInt([321,'asdf',window]) returns 321.

parseInt() doesn't treat the number Infinity correctly. parseInt(Number.MAX\_VALUE \* 2) returns NaN, but Number(Number.MAX\_VALUE \* 2) correctly returns Infinity

parseInt() doesn't threat negative exponents correctly. parseInt(2e-6) returns 0 and parseInt(2e-7) returns 2.

**\*/**

Also worth nothing that parseInt shouldn't be used with Array::map -- it gets weird since parseInt accepts more than one parameter. Number, however, works well.

["0", "1", "2", "3", "4"].map(parseInt)

[0, NaN, NaN, NaN, NaN]

["0", "1", "2", "3", "4"].map(Number)

[0, 1, 2, 3, 4]

**/\*\***

For most string to number conversions, I think both are interchangeable. However there are differences:

* As mentioned, if a string doesn’t look like a number, using Number will give you NaN.
* parseInt reads as many characters as it can from the string, until it hits a character it cannot convert.

**const** s = '123abxyz';Number(s); *// NaN*parseInt(s); *// 123 (123 is valid base-10, and stops as soon as it hits `a`)*

* Number('') gives 0; parseInt('') gives NaN

**\*\*/**

Now thats clear as crystal! So according to your reply, I guess nothings really better between the two and there is no reason to compare both Number converting stuffs for they do specific jobs and they differ in a unique way! Thanks sir Moderator! Youre cool!

**\*\*/**

# Angular 7 Call Rest API using Get with Params and POST

<https://www.youtube.com/watch?v=rdLJNGZvlAA>

In terminal

:> ng version  
:> ng --version  
:> ng -v

**Angular 2 Components**

We all know that Angular 2 is a component based framework because everything is a component in Angular 2. Basically, components are the basic building blocks of Angular 2 application. It also allows us to create reusable UI templates.

A component in Angular 2 is a class with a template and a decorator. There are basically the following parts of Angular 2 component –

* **Class**  
  It is very similar to the C# class or java class etc. It contains the constructor, variables and methods code which is required for the template/user interface.
* **Decorator**  
  A decorator is used to store the metadata about the class. Basically, the decorator provided by Angular makes a class an Angular component when it is decorated with the component decorator.

Angular 2 provides us basically 4 types of decorators,

1. **Class Decorators**  
   For Ex. @NgModule, @Component & @Directive
2. **Property Decorators**  
   For Ex. @Input & @Output
3. **Method Decorators**  
   For Ex. @HostListener
4. **Parameter Decorators**  
   For Ex. @Inject

I am not going to define the types of decorators here in detail. We will discuss in detail about them in later articles.

Each decorator has a basic configuration using several properties. We are taking a look at some possible configuration properties right here, that you can use when creating a component,

1. **Selector**– This is used for identifying this component in templates.
2. **Template**– This is used for defining HTML template inline for the view.
3. **TemplateUrl**– This is used for defining a URL to an external file containing a template for the view.
4. **Styles**– This is used for defining inline CSS to be applied to the template of this component.
5. **StyleUrls**– This is used for defining URLs to external style sheets to be applied to the templates of this component.
6. **viewProviders**– This is used for defining a list of providers available for this component and its view children