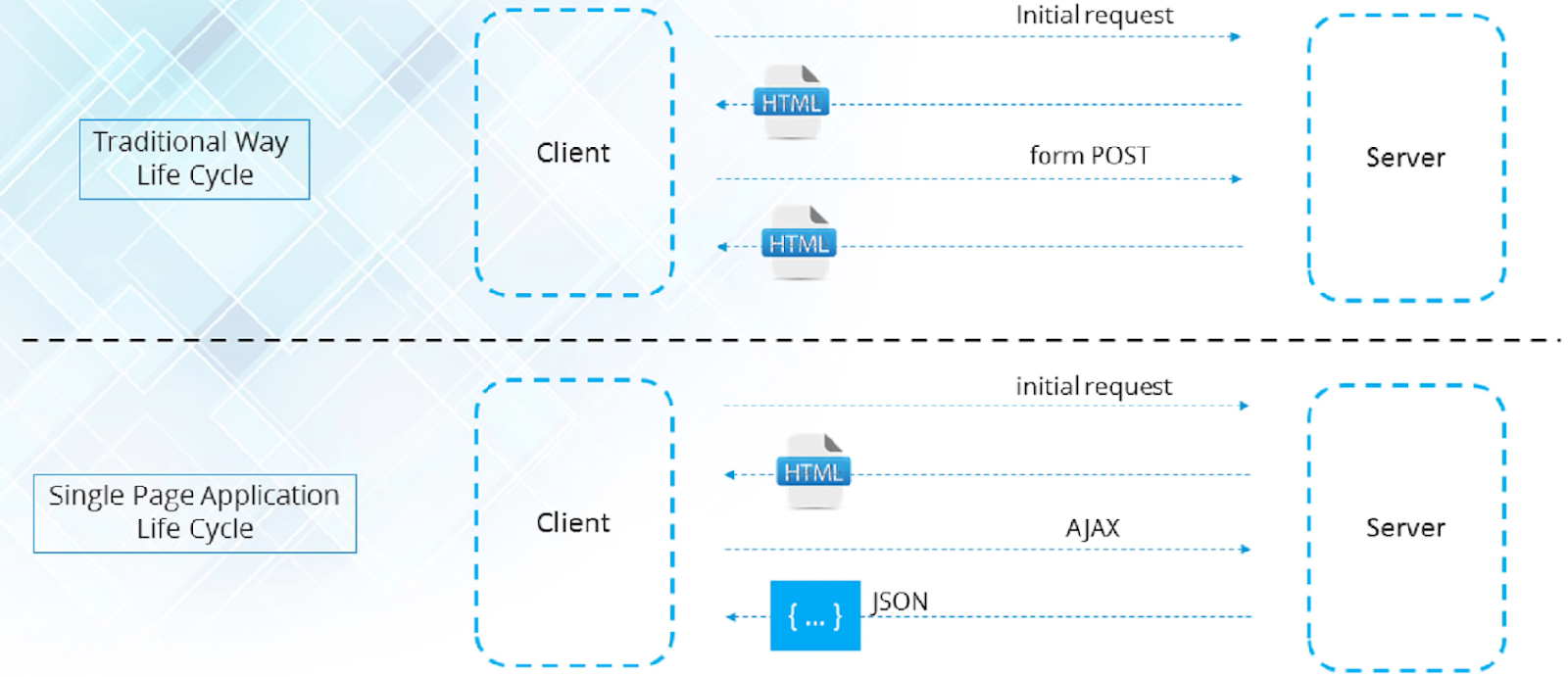
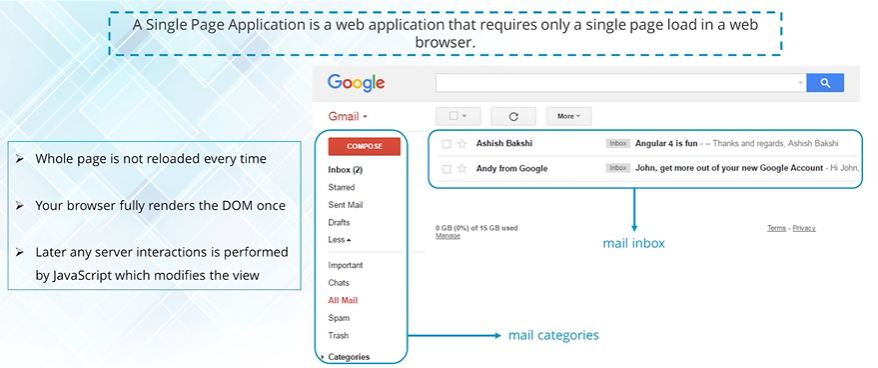
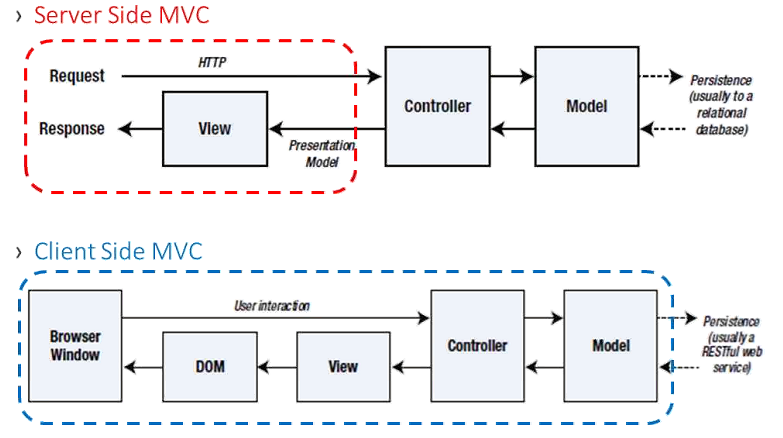
**Introduction**

* Angular is a Client side JavaScript Framework which allows us to create Reactive Client-side Application.
* Great approach for a Single Page Application.
* By default it supports Two-way Data binding.
* By design it comes with Modular Approach, so you can reuse the code wherever you want.
* It supports many inbuilt functionalities.
* By this Angular architecture, we can easily maintainable the code.

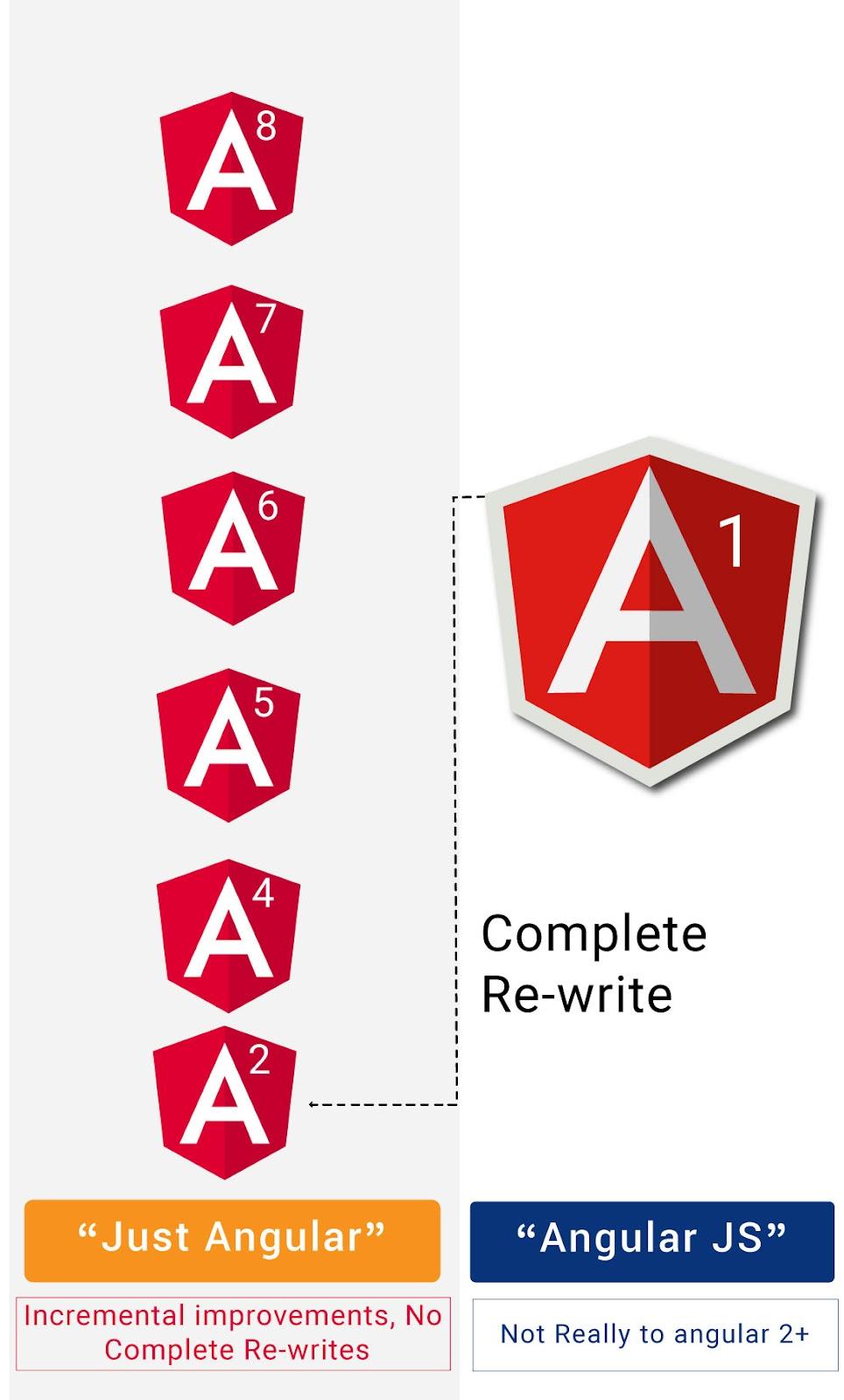




Angular has Model View Whatever structure



**Angular Versions**



****

**Environment Setup**

* Install the Angular CLI by using the following command.

***npm install -g @angular/cli***

* Once you installed just use the below cmd for the verification

***ng --version***

* It will give you the installed version details

If you installed Angular Already means, uninstall to avoid version conflicts

***npm uninstall -g*** ***angular-cli @angular/cli***

***npm cache clean --force***

* Navigate to where you want to create your Angular project by using cd cmd.
* Run the following cmd to create your 1st project.

***ng new your-project-name***

* Once your project setup, run the below cmd

***ng serve -o***

* Now you can run your project in (By default the port is 4200)

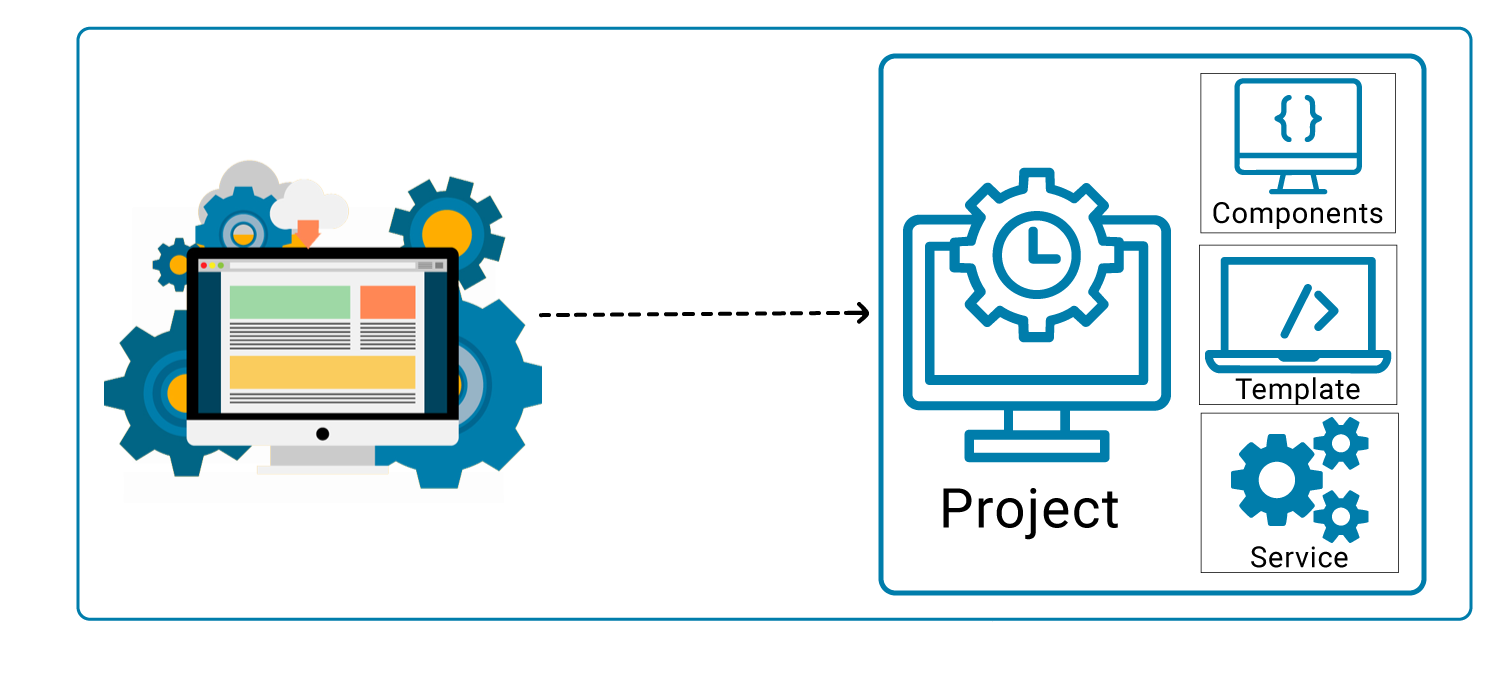
[***http://localhost:4200/***](http://localhost:4200/)

* If you want to change the port, run the following command

***ng serve --port 4201 -o***

First let's understand why should we use Angular CLI and what problems it solves.

CLI stands for Command Line Interface.



1.Create a separate application folder and add the package definition file ( ie. package.json) and other configuration files.

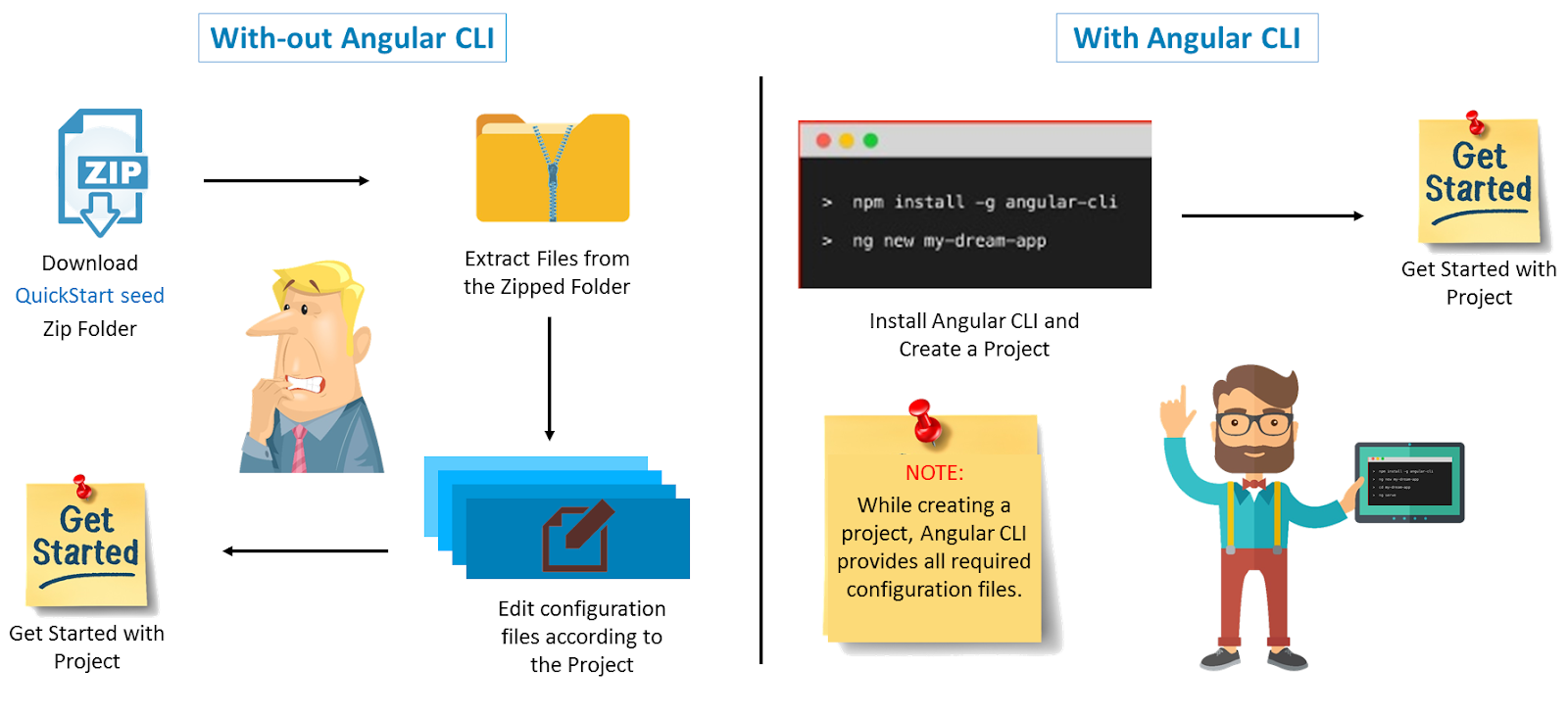
2. Install the packages using NPM

3. Setup the environment.

4. Provides required files from angular program.

5. Create index.html which hosts our application.

6. Reduces the development effort



**Structure of Angular Project**

* **node\_modules** is the place where our all third party library resides, which can be used for development purposes.
* **Src** folder contains the actual source code for development. The src folder has 3 subfolders: App, Assets, Environment.
* **App** folder is the prominent part of Angular. It has all the components and modules of the application.
* **Assets** folder is the place where we can store our images and icons.
* **Environment** folder has two files: environment.prod.ts stores configuration for production environment and environment.ts stores configuration for development environment.
* **favicon.ico**. is the icon file which is displayed on the browser when you run the application.
* **index.html** is the first html file that is loaded when your application is run on browser.
* **Main.ts** is doorway for our application. It is the typescript file. Here we can bootstrap(load) our main module using methods.
* **Pollyfills.ts** is the scripts which eliminates the need to set up everything. In the other words, it makes our application compatible with different browsers. It bridges the gap between our Angular app and browser.
* **style.css** is the file where global styles for our application resides.
* **test.ts** is used for testing purpose.
* **tsconfig.app.json** has the root files and the compiler options.
* **.editorconfig** is used to define standard and consistence coding patterns for team development purpose.
* **.gitignore** is used when exporting your files and folders to github.
* **angular.json:** It is very important configuration file related to your angular application. It defines the structure of your app and includes any settings associated with your application. Here, you can specify environments on this file (development, production). This is the file where we add Bootstrap file to work with Angular.
* **karma.conf.js** is the file which is used for unit test.
* **package.json** is the file which holds metadata for projects such as project name, version and handle dependencies of the project.
* **package-lock.json** : This is an auto-generated and modified file that gets updated whenever npm does an operation related to node\_modules or package.json
* **tsonfig.json** This is a typescript compiler configuration file.
* **tsconfig.app.json:** This is used to override the tsconfig.json file with app specific configurations.
* **tsconfig.spec.json**: This overrides the tsconfig.json file with app specific unit test configurations.
* **tslint.json:** file checks your code for errors. It can be customised according to your own rules.

**Difference between tilde(~) and caret(^) in package.json**

The tilde matches the most recent minor version (the middle number).

~1.2.3 will match all 1.2.x versions but will miss 1.3.0.

This has been the default behaviour of ‘–save’ since the start, and you are probably already comfortable seeing it in your package.json.

The caret, on the other hand, is more relaxed.

It will update you to the most recent major version (the first number). ^1.2.3 will match any 1.x.x release including 1.3.0, but will hold off on 2.0.0.

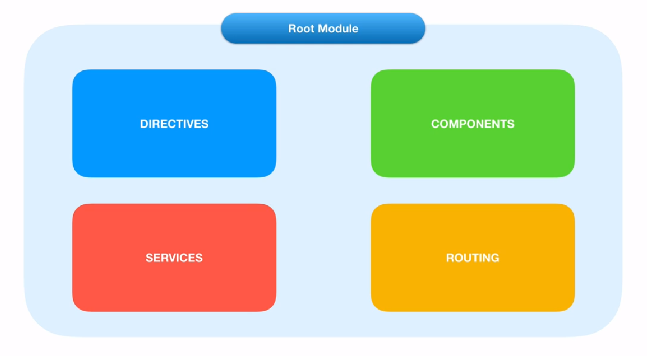
**Main Building Blocks of Angular**

1. Modules
2. Components
3. Templates
4. Metadata
5. Data binding
6. Directives
7. Services
8. Dependency Injection.

**Modules**

****

* Bundle of functionalities of our App.
* Every Angular app contains at least one Angular module, i.e. the **root module**.
* Generally, it is named **as AppModule**.
* We can create multiple Modules if needed.
* Any angular module is a class with **@NgModule** decorator.
* Encapsulation of different similar functionalities.



**Importing other modules in Root Module**



**Module Decorators:**

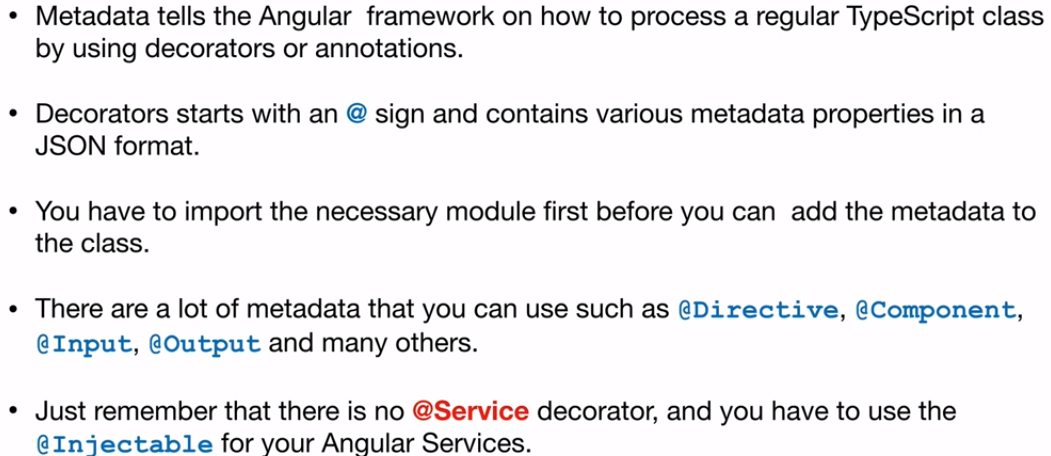
NgModule is a decorator function that takes metadata object whose properties describe the module.

Decorators are basically used for attaching metadata to classes so that it knows the configuration of those classes and how they should work.

The properties are,

* **declarations**: The classes that are related to views and it belongs to this module.
* **imports**: Modules whose classes are needed by the component of this module.
* **providers**: Services present in one of the modules which is to be used in the other modules or components. Once a service is included in the providers it becomes accessible in all parts of that application.
* **exports**: The classes that should be accessible to the components of other modules. (A root module generally doesn’t export it’s class because as root module is the one which imports other modules & components to use them.)
* **bootstrap**: The *root component* which is the main view of the application. This root module only has this property and it indicates the component that is to be bootstrapped.

**Angular Metadata**

****

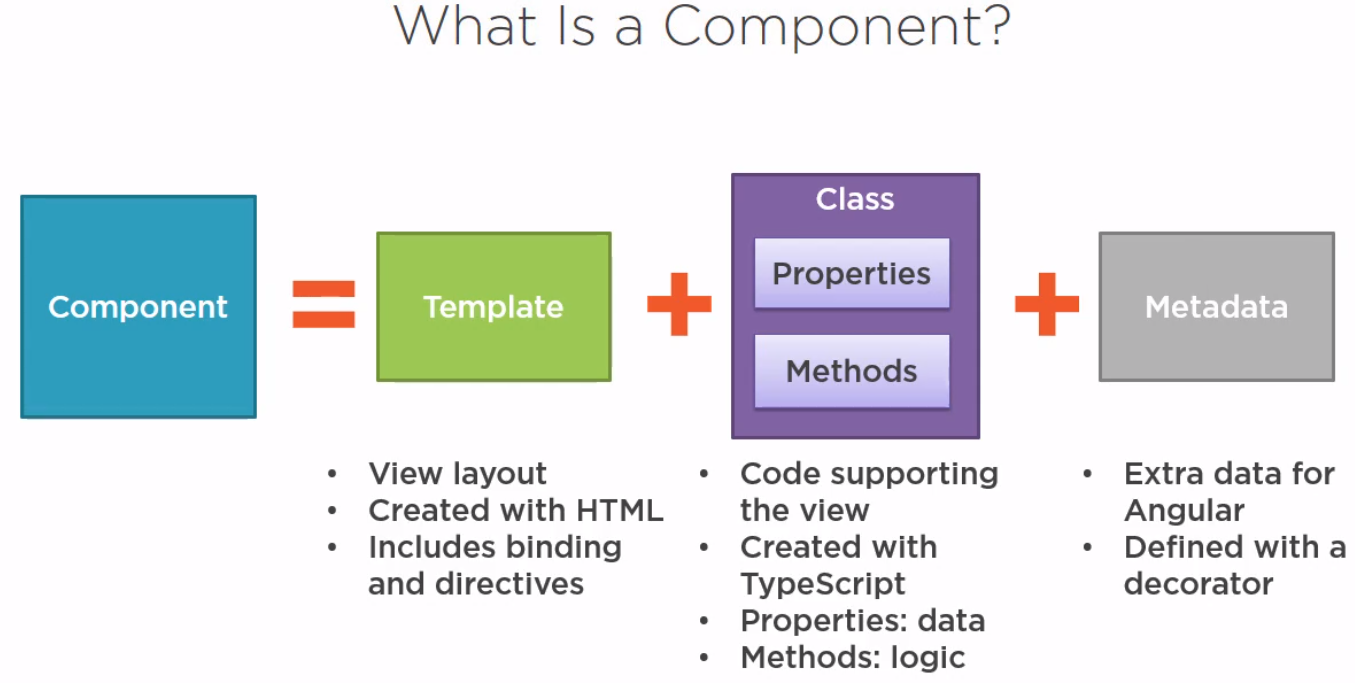
**Angular libraries**

* Angular gives us a collection of JavaScript modules (library modules) which provide various functionalities.
* Each Angular library has **@angular** prefix, like @angular/core, @angular/compiler, @angular/compiler-cli, @angular/http, @angular/router.
* You can install them using the **npm** package manager and import parts of them with JavaScript import statements.

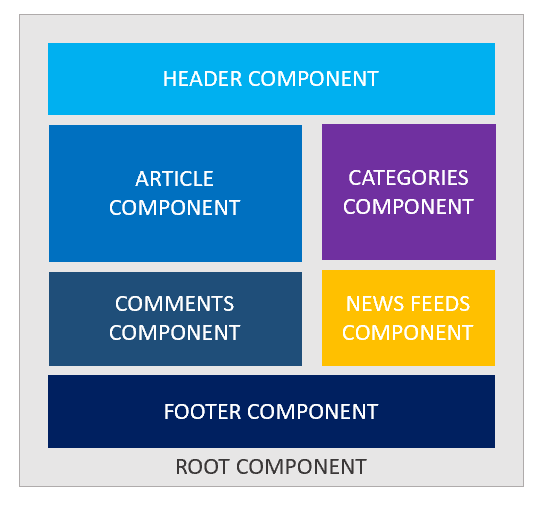
| import { Component } from '@angular/core'; |
| --- |

* In the above example, Angular’s Component decorator is imported from the @angular/core library.

**Components**

****

* A component controls one or more section on the screen called a **view**.
* For example, if we build shopping cart Application, we can have components like App Component (the bootstrapped component), list products, product description, add to cart, update cart, etc.,



* Component fetch and update data from **services**. Transforms the DOM using **Directives** and Redirecting the user to another component by using **Routing**.
* Inside the component, you define a component’s **presentation** logic i.e. how does it support the view—inside a class.
* Every app has a main component which is bootstrapped inside the main module, i.e AppComponent.

| import { Component } from '@angular/core';   @Component({ selector:'app-root', templateUrl:'./app.component.html', styleUrls: ['./app.component.css'] })   export class AppComponent{ title = 'app works!'; } |
| --- |

## 

## **Metadata:**

Metadata tells Angular how to process a class.

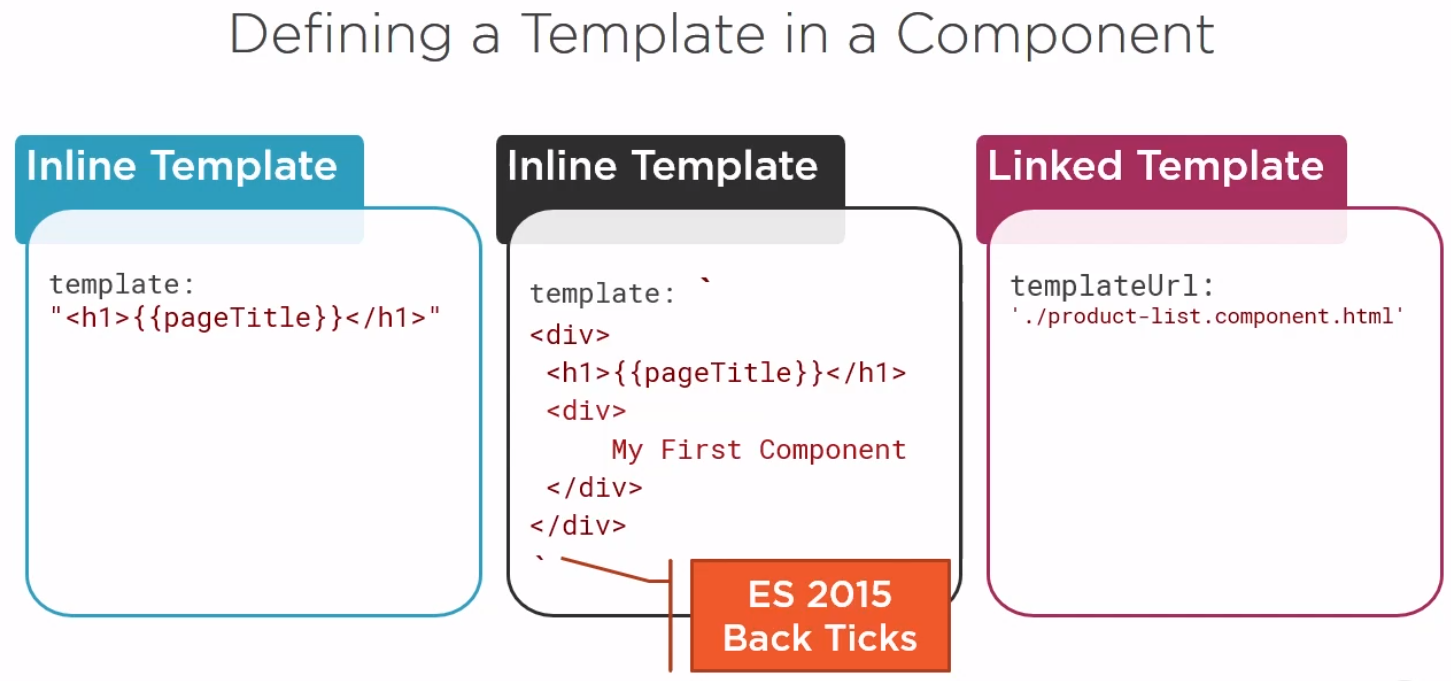
| import { Component } from '@angular/core';   @Component({ selector:'app-root', templateUrl:'./app.component.html', styleUrls: ['./app.component.css'] })  ng generate component componentName |
| --- |

* Here is the ***@Component*** decorator, which identifies the class immediately below it as a component class.
* The ***@Component*** decorator takes the required configuration object which Angular needs to create and present the component and its view.

The most important configurations of @Component decorator are,

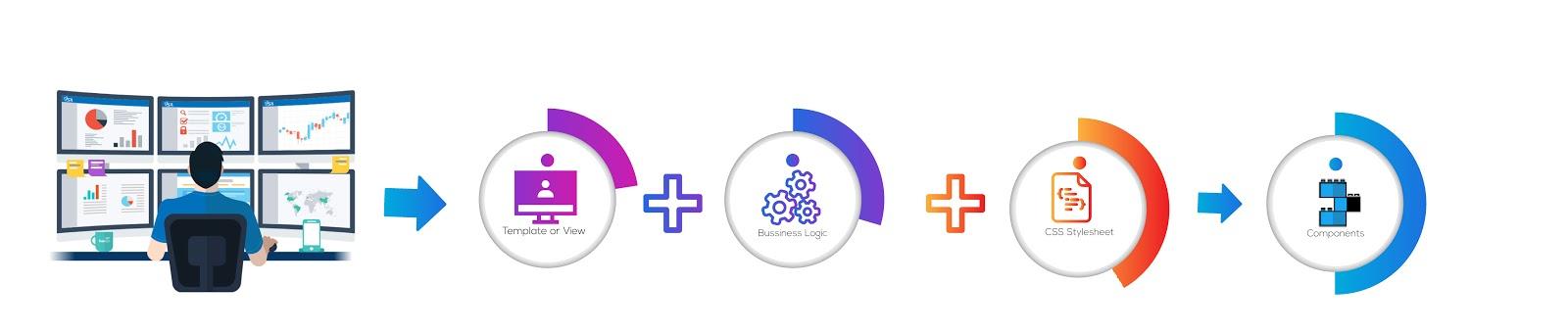
* **selector**: Selector tells Angular to create and insert an instance of this component where it finds ***<product-desc>*** tag. For example, if an app’s HTML contains ***<product-desc></product-desc>***, then Angular inserts an instance of the Product Description view between those tags.
* **templateUrl**: It contains the path of this component’s HTML template.
* **styleUrls**: We can specify the styes which is related to current component.

The template, metadata, and component together describe a view.



**An Angular Component in Action**

* app.component.css
* app.component.html
* app.component.spec.ts
* app.component.ts
* app.module.ts
* **A CSS file**: where we define all the styles for that component. These styles will only be scoped to this component and will not leak to the outside.
* **An HTML file**: contains the markup to render in the DOM.
* **A spec file**: includes the unit tests.
* **A TypeScript file**: where we define the state (the data to display) and behavior (logic) of our component.



**Note:**

* Angular Server serves the index.html and renders the application view in browser.
* The index.html contains all the JS, styles, compilation files which is required for run the application.
* The index.html holds the root component (AppComponent) which bootstrapped from the root model and get loads very 1st.
* The index.html should have only one component that is root component by the directive **<app-root></app-root> (template directive).**
* The selector property value is used to create and attach the instance of class in the innerHtml of the element.
* The root component having multiple nested child components.

## **Create Component through CLI**

| ng generate component componentName  ng g c componentName |
| --- |

**Use your selector as an Attribute:**

To use your selector as an Attribute, simply make changes in selector property as

| selector : '[attributeName]' |
| --- |

Now in your html just use it as Attribute for an element

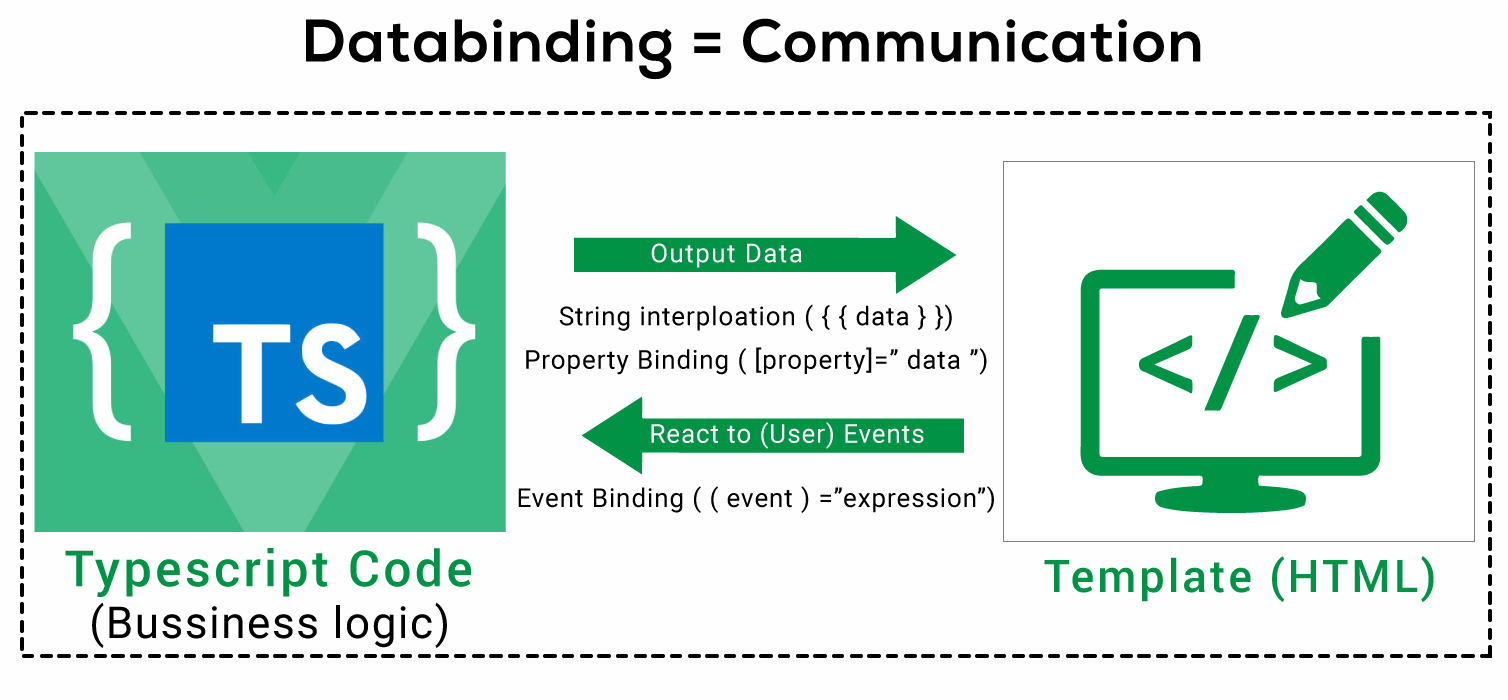
| <div attribute></div> |
| --- |

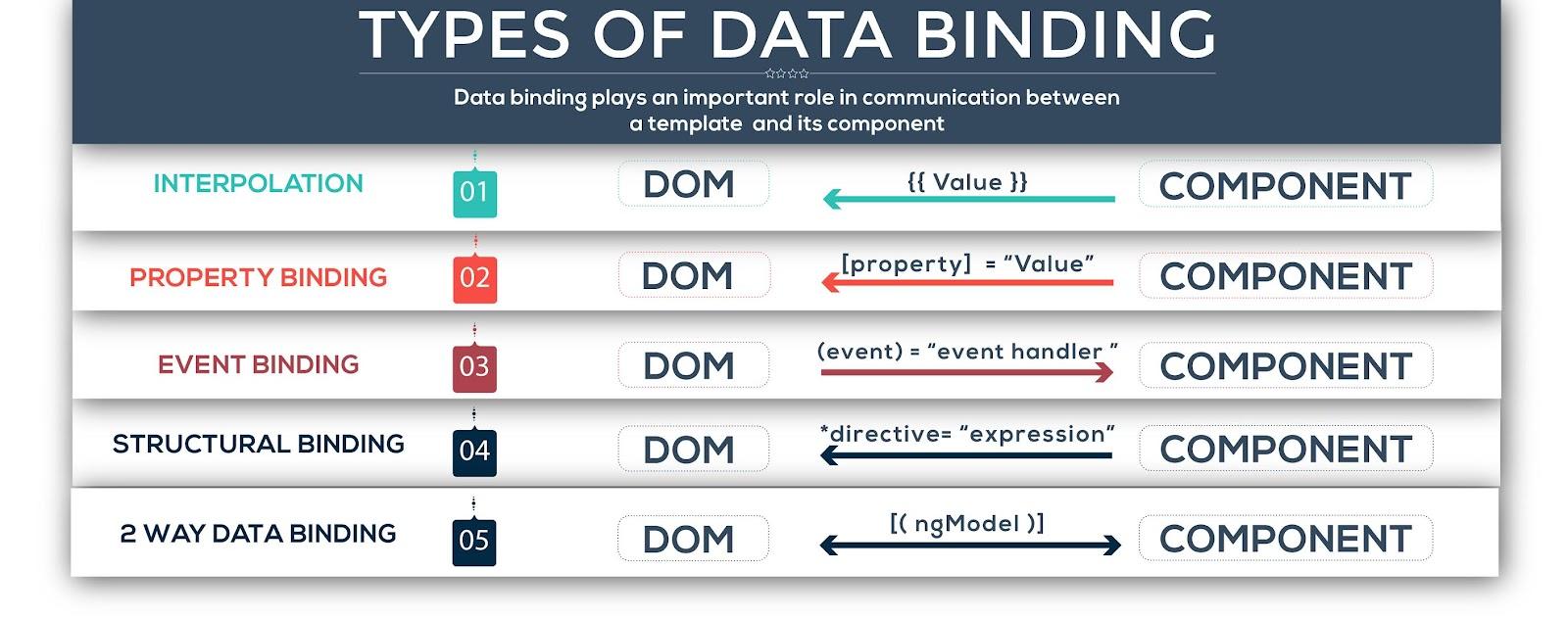
**Use your selector as a Class:**

| selector : '.className' |
| --- |

## **Data Binding**

## Which means Projection of the Model or Communication between component & DOM.





The combination of both is called Two way data binding.

| ( [(ngModel)]="data" ) |
| --- |

## **Interpolation {{…}}**

It’s used to weave string value into the HTML element. Use variable name inside the double curly braces "**{{ }}**".

Example:

| <**h1**>Interpolation Example<**;**/h1>  Name: {{name}}  To pay: {{amount \* 10}}  <**img** src="{{imageUrl}}"> |
| --- |

**Property Binding […]**

It’s used to set the property of the element to a component property value. Use square brackets “[]” to set property binding.

Example

| <**h1**>Property Binding Example</**h1**> <**img** [src]="imageUrl">  <**button** [disabled]="isUnchanged">Cancel is disabled</**button**> |
| --- |

**Event Binding (…)**

It’s used to set the event of HTML element.

Example

| <**h1**>Event Binding Example</**h1**>  <**button** (click)="onSave()">;Save</**button**> |
| --- |

**Two-way Binding [(…)]**

It’s also known as Banana in the box "[()]" – the combination of property binding [ ] and event-binding ( ), that said you can display the property and update the same whenever the user makes any changes. It can be done using ngModel.

Example

| <input [(ngModel)]="name" > |
| --- |

**Class Binding:**

You can add and remove CSS class names from an element's class attribute with a class binding.

Syntax:

| [class.className]="propertyValue"; |
| --- |

If the propertyValue becomes true the specific class will be apply or else no.

**Style Binding:**

You can set inline styles with a style binding.

Syntax:

| [style.stylePropertyName]="propertyValue ? ‘value1’ : ‘value2’" |
| --- |

(**condition**) ? expression on **true** : expression on **false**

## **Directives**

Directives are Instructions to the DOM.

There are three kinds of directives in Angular:

1. **Components** - directives with a template.
2. **Structural directives** - change the DOM layout by adding and removing DOM elements. (leading with **\***).
3. **Attribute directives** - change the appearance or behavior of an element.

## **List of Directives in Angular**

1. **\*ngIf**

* The **NgIf** directive is used when you want to display or remove an element based on a condition.
* If the condition is false the element the directive is attached to will be removed from the DOM.

*The syntax is: \*ngIf="<condition>"*

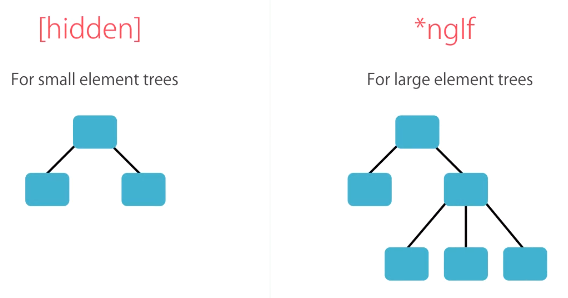
**Difference Between ngIf and hidden.**

**ngIf :** It just remove the element and re-attach the element based on the condition.

**<ng-template>** is an Angular virtual element used to render HTML templates.

We use **<ng-template>** with angular \*ngIf directive to display else template.

1. **Hidden :** It just hides and shows the attached element based on the condition.



1. **\*ngFor**

It’s point is to repeat a given HTML template once for each value in an array, each time passing it the array value as data binding.

By default \*ngFor has the following local variables,

**index**: number: The index of the current item in the iterable.

**first**: boolean: True when the item is the first item in the iterable.

**last**: boolean: True when the item is the last item in the iterable.

**even**: boolean: True when the item has an even index in the iterable.

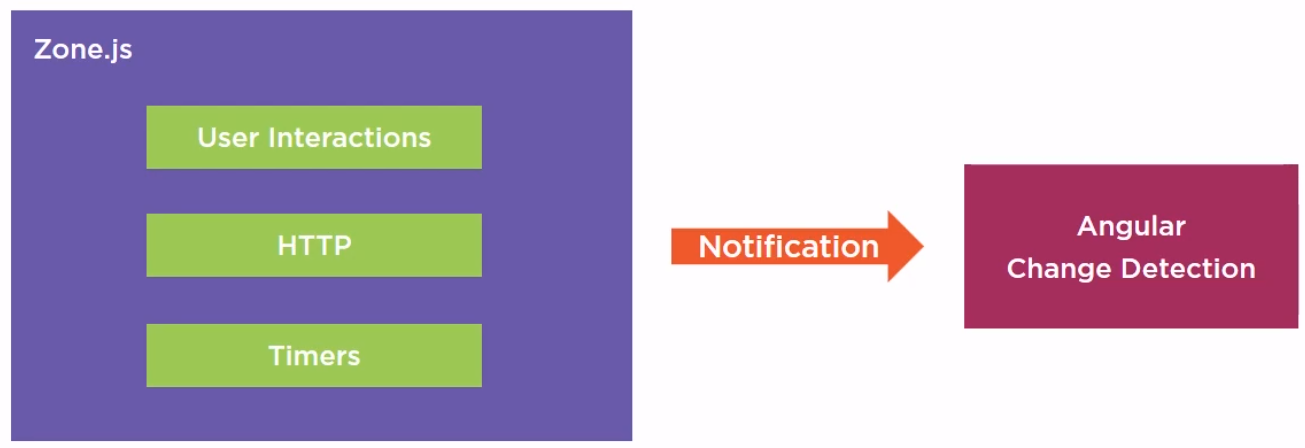
**odd**: boolean: True when the item has an odd index in the iterable.

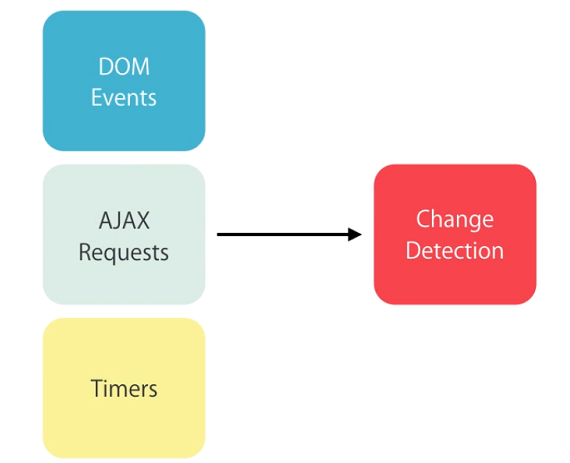
Syntax:

| <li \*ngFor="let user of users; index as i; first as isFirst"> |
| --- |

**ngFor and Change Detection**

**Change Detection**

****



1. ngClass

The NgClass directive allows you to set the CSS class dynamically for a DOM element.

1. ngStyle

The NgStyle directive lets you set a given DOM elements style properties.

1. ngSwitch

This directive allows us to render different elements depending on a given condition.

| **Note**: We can’t have two structural directives, directives starting with a \*, attached to the same element. |
| --- |

**NgNonBindable**

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

*<pre ngNonBindable>{{ name }}</pre>*

**Install Bootstrap npm**

| npm install bootstrap@3 --save |
| --- |

After successful Installation, make necessary changes in angular.json.

***angular.json => architect => build => style[];***

**Key Event Filtering:**

Syntax:

| <input type="text" (keyup.enter)="OnKeyUp($event)"> |
| --- |

Notice **$event**? It's an optional, and it is used to capture the data which is emitted by the event.

If you want to invoke a function only for inputs means just use ***(input)***.

**Pipes**

Angular Pipe takes in data as input and **transforms** it to a desired output before the view. Angular comes with some inbuilt pipes as follows,

* Lowercase
* Uppercase
* Titlecase
* Slice
* Json
* Number - {minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}
* Percent
* Currency
* Date

Pre-defined format options

**'short'**: equivalent to 'M/d/yy, h:mm a' (6/15/15, 9:03 AM).

**'medium'**: equivalent to 'MMM d, y, h:mm:ss a' (Jun 15, 2015, 9:03:01 AM).

**'long'**: equivalent to 'MMMM d, y, h:mm:ss a z' (June 15, 2015 at 9:03:01 AM GMT+1).

**'full'**: equivalent to 'EEEE, MMMM d, y, h:mm:ss a zzzz' (Monday, June 15, 2015 at 9:03:01 AM GMT+01:00).

**'shortDate'**: equivalent to 'M/d/yy' (6/15/15).

**'mediumDate'**: equivalent to 'MMM d, y' (Jun 15, 2015).

**'longDate'**: equivalent to 'MMMM d, y' (June 15, 2015).

**'fullDate'**: equivalent to 'EEEE, MMMM d, y' (Monday, June 15, 2015).

**'shortTime'**: equivalent to 'h:mm a' (9:03 AM).

**'mediumTime'**: equivalent to 'h:mm:ss a' (9:03:01 AM).

**'longTime'**: equivalent to 'h:mm:ss a z' (9:03:01 AM GMT+1).

**'fullTime'**: equivalent to 'h:mm:ss a zzzz' (9:03:01 AM GMT+01:00).

## **Services**

* Angular services are **singleton** objects which get instantiated only once during the lifetime of an application.
* They contain methods and properties that maintain data throughout the life of an application, i.e. data does not get refreshed and is available all the time.
* The main objective of the service is to organize and share business logic with different components of an Angular application.

**Why Should We Use Services in Angular?**

* Angular service methods can be invoked from any component of Angular, like Controllers, Directives, etc.
* This helps in dividing the web application into small, different logical units which can be reused.

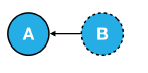
| **Note**: We use the ***@Injectable*** class decorators to automatically resolve and inject all the other classes. This only works if each parameter has a TypeScript type associated with it, which the DI framework uses as the token. |
| --- |

**Dependency Injection:**

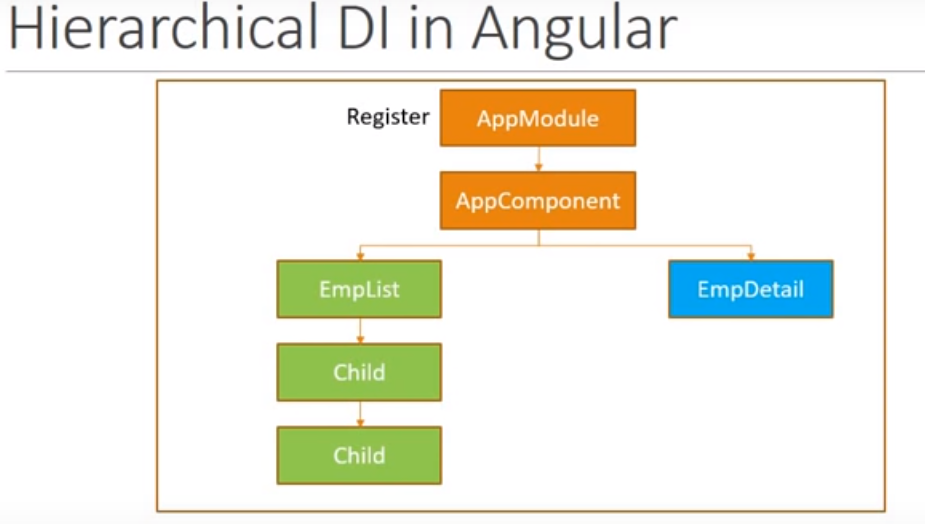
**What is a dependency?**

When module **A** in an application needs module **B** to run, then module **B** is a

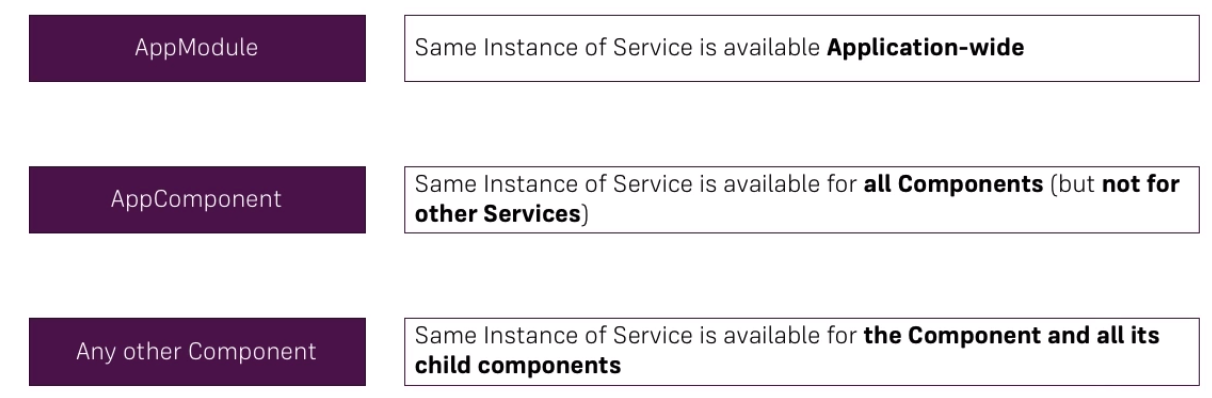
dependency of module **A**.



* Dependency Injection is a way of architecting an application so code is easier to re-use, easier to test and easier to maintain.
* Angular comes with a Dependency Injection (DI) framework of it’s own and it’s used throughout Angular code.



* If we want a dependency to be shared across our entire application we would configure it on our NgModule.
* If we want a separate instance of a dependency to be shared across each instance of a component and it’s children we configure it on the components providers property.



**Ways to implement Dependency Injection in Angular:**

Below are the ways to implement DI in Angular.

* Using @Injectable() in the service
* Using @NgModules() in the module
* Using Providers in the component

**Custom Pipes**

* Every pipe is decorated with ***@Pipe*** where we define the name of our custom pipe. Every pipe will implement ***PipeTransform*** interface.
* This interface provides ***transform()*** method and we have to override it in our custom pipe class.
* ***transform()*** method will decide the input types, number of arguments and its types and output type of our custom pipe.
* We perform the following steps to create a custom pipe.

Step 1: Create a pipe by using ***ng g p pipeName***

Step 2: Decorate the class using @Pipe.

Step 3: Implement **PipeTransform** interface.

Step 4: Override transform() method.

Step 5: Configure the class in application module with @NgModule.

Step 6: Ready to use our custom pipe anywhere in application.

On the basis of change detection, angular provides two types of pipes.

**Pure pipe**: A pure pipe is only called when Angular detects a change in the value or the parameters passed to a pipe.

**Impure pipe**: An impure pipe is called for every change detection cycle no matter whether the value or parameters changes.

This is relevant for changes that are not detected by Angular

* when you pass an array or object that got the content changed (but is still the same instance).
* when the pipe injects a service to get access to other values, Angular doesn't recognize if they have changed.

By default, All the pipes become pure pipe.

**Practicals:**

1. Ordinal
2. TitleOnlyCase (The Vitamins are in My Fresh Indian Raisins).
3. CleanWord
4. camelCase
5. Reverse
6. **CreditCard \*\*\*\* \*\*\*\* \*\*\*\* 6789**
7. **FilterByPropertywise**
8. **CreditCardImg master / visa**

**Creating Custom Directives**

**Angular Custom Attribute Directive:**

Find the steps to create custom attribute directive.

1. Create a class decorated with **@Directive()**.

2. Assign the attribute directive name using selector metadata of **@Directive()** decorator enclosed with bracket [] .

3. Use **ElementRef** - it is a service, which gives us access to the DOM Object.

The *ElementRef* gives the directive direct access to the DOM element upon which it’s attached.

4. Use **@Input()** decorator to accept user input from our directive.

5. Use **@HostListener()** decorator to listen events in custom attribute directive.

6. Configure custom attribute directive class in application module in the declarations metadata of **@NgModule** decorator.

Angular allows us to create Custom directives.

Syntax:

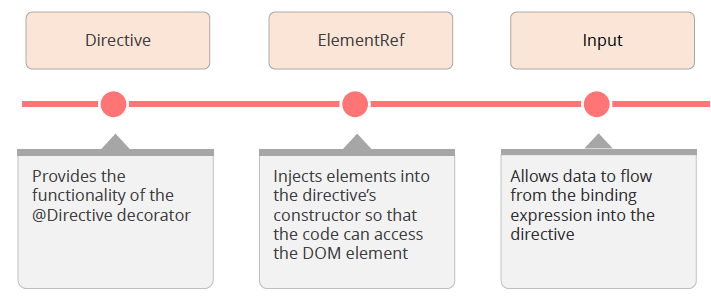
| ng g d directiveName |
| --- |

**HostListener - decorator**

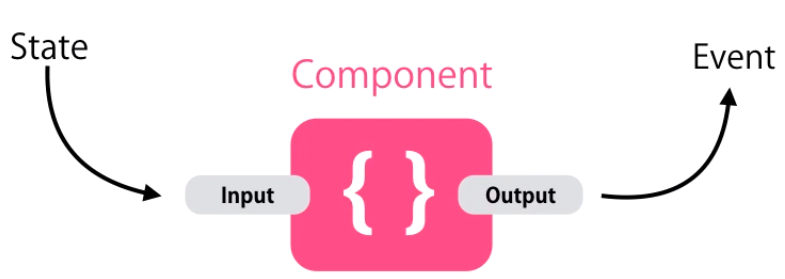
* This decorator allows us to subscribe the events which is raised from the DOM element.
* Listen to the event occurs on the element and act accordingly.
* This is a function decorator that accepts an event name as an argument.
* When that event gets fired on the host element it calls the associated function.

So if we add this function to our directive class:

| @HostListener('mouseover') onHover() {  console.log("hover"); } |
| --- |



**Components Interaction**



Pass data from parent to child with input property binding by using @Input decorator.

**@Input**

***It Allows this component’s property from outside***

Defines input variable in component communication. It is used to communicate from parent to child component using **property binding**.

* As the name implies it is used to **Input**ting the data.
* Enables the component to accept Input from the parent component.
* Use the @Input decorator.

**@Output**

Defines output variable in component communication. It is used to communicate from child to parent component using **custom event binding**.

* As the name implies it is used to **Output**ting the data.
* Enables the component to send or emit an output to another component..
* Use the @Output decorator.

**How to add an @Output Property?**

1. Import the **@Output** decorator & **EventEmitter** from the @angular/core library.
2. Declare a new variable with an @Output decorator and a value of new EventEmitter Object.
3. Add the new variable as a data binding target, enveloped inside a parenthesis.
4. Send out (emit) the output data by calling the emit() method of new variable.

**EventEmitter:**

* ***EventEmitter*** is a class in angular framework. It has **emit**() method that emits custom events.
* We can use **EventEmitter** in custom event binding.
* To create a custom event we need to create an instance of ***EventEmitter*** annotated by ***@Output()***.
* We can receiving emitting custom function arguments by ***$event***.
* To achieve it first we need to import it in our component file as given below

| import {Component, EventEmitter, Input, Output} from '@angular/core'; |
| --- |

* And then initialize it using @Output decorator as follows,

| @Output() customEventname = new EventEmitter(); |
| --- |

* Using emit() method of EventEmitter class we can emits parent component event.

| this.customEventname.emit(); |
| --- |

**View Encapsulation**

View encapsulation defines whether the styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies:

* **Emulated** (default) - styles from main CSS (style.css) propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
* **Native / Shadowdom** - styles from main CSS do not propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.

*Native*: (Uses browser's native Shadow DOM. Check for browser support before enabling it.)

*ShadowDom:* Uses browser’s native Shadow DOM v1 for better cross-browser support and is a shared standard across the browsers.

* **None** - styles from the component propagate back to the main HTML and therefore are visible to all components on the page.

**ng-container directive**

We can’t use more than one structural directive in a single element. In order to avoid having to create that extra div, we can instead use ng-container directive:

| <**ng-container** \*ngIf="lessons">  <**div** class="lesson" \*ngFor="let lesson of lessons" \*ngIf=””>  <**div** class="lesson-detail">  {{lesson | json}}  </**div**>  </**div**> </**ng-container**> |
| --- |

**Angular Component Lifecycle Hooks**

A component in Angular has a life-cycle, a number of different phases it goes through from start to end.

We can hook into those different phases to get some works done on the particular phase.

**Constructor:**

This is invoked when Angular creates a component calling new on the class.

**ngOnChanges -** OnChanges**: Called whenever @input() property value changes**

Invoked event **ngOnChanges** every time there is a change in one of the input properties [inputProperty] of the component.

**ngOnInit -** OnInit**: Called once the component Initialised**

The ngOnInit method of a component is called directly after the constructor and after the ngOnChange is triggered for the first time. It is the perfect place for initialisation work. Invoke only once.

**ngDoCheck -** DoCheck**: Called during every change detection runs.**

This is fired each time anything that can trigger change detection has fired (e.g. click handlers, http requests, route changes, etc…). This lifecycle hook is mostly used for debug purposes;

**ngAfterContentInit -** AfterContentInit **: Called after content has been projected into view**

Invoked after Angular performs any content projection into the components view. Invoke only once.

**ngAfterContentChecked -** AfterContentChecked: called after every check of component (ngDoCheck) content.

**ngAfterViewInit -** AfterViewInit: **Called after the component's view (and its child view) has been initialised.**

Called after ngAfterContentInit when the component’s view has been initialised. Invoked when the component’s view has been fully initialised. Invoke only once.

**ngAfterViewChecked -** AfterViewChecked

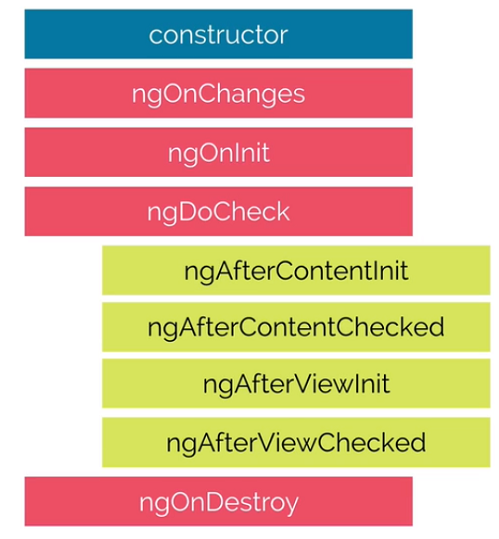
Invoked each time the view of the given component has been checked by the change detection mechanism of Angular. called after every check of component (ngAfterContentChecked) content.

***ngDoCheck => ngAfterContentChecked => ngAfterViewChecked***

**ngOnDestroy -** OnDestroy

This method will be invoked just before Angular destroys the component.

The hooks are executed in this order:



**Template Driven Forms**

* In Template driven Form Approach, everything is defined in the template.
* In template driven we use directives to create the model.
* The directives we need to build template driven forms are in the FormsModule.

**Form Setup:**

* Import FormsModule and add it to our NgModule as an import.

| import {FormsModule} from '@angular/forms'; |
| --- |

* One of the directives pulled in via the FormsModule is called ngForm.
* So just by adding FormsModule to our NgModule imports our template form is already associated with an instance of the ngForm directive.
* This instance of ngForm is hidden but we can expose it with a local template reference variable attached to the form element like,

| <form #f="ngForm"> ... </form> |
| --- |

* Now we can use the variable **f** in our template and it will point to our instance of the ngForm directive.
* To create **Form control,** we need to do two things to each template.

1. Add the **ngModel** directive
2. Add the **name** attribute.

| <input name="foo" ngModel> |
| --- |

**Angular Form States:**

| **States** | **Data Type** | **Description** |
| --- | --- | --- |
| valid | Boolean | Returns true if the form/input element has been valid |
| invalid | Boolean | Returns true if the form/input element has been Invalid |
| touched | Boolean | Returns true if the input element has been touched |
| untouched | Boolean | Returns true if the input element yet not touched |
| dirty | Boolean | Returns true if the input element or form has been modified |
| pristine | Boolean | Returns true if the input element or form yet not modified (original state) |
| submitted | Boolean | Returns true if the form has been submitted |
| errors | Object | Returns an object with existing errors in the input elements of form. The key names are error names and the values always true or object. |

**Reactive Form Approach**

Reactive Form created programmatically and synchronised with DOM.

**Form Setup**

1. Import ReactiveFormsModule

| import { ReactiveFormsModule } from '@angular/forms'; |
| --- |

2. Create a property which holds our form with the type of FormGroup

| signUpForm : FormGroup; |
| --- |

**What is FormGroup?**

FormGroup in Angular is a collection of different FormControls and it is used to manage the value of different inputs and implement validations as well.

**What is FormControl?**

FormControl is a class that is used to get and set values and validation of a form control such as <input> and <select> tag.

**Creating a Form from code**

| this.signUpForm = new FormGroup({  username : new FormControl(''),  email : new FormControl(''),  gender : new FormControl(''),  }); |
| --- |

**Sync HTML & Our Form**

* To override the default form behaviour**,** attach **[formGroup]** Directive in your form element.
* Setup the Form Control by using **formControlName=”the specific property name”**

| <form [formGroup]="signUpForm">  <div class="form-group">  <label for="exampleInputName">Name</label>  <input type="text" class="form-control" id="exampleInputName" aria-describedby="nameHelp" placeholder="Enter Name"  formControlName="username">  </div> </form |
| --- |

**Adding Validations:**

First, we need to import validators like this.

| import{ Validators} from '@angular/forms'; |
| --- |

Then we can use different validators using our form controls as explained below.

| this.loginForm = new FormGroup({ username : new FormControl('', [Validators.required, Validators.minLength(5)]) }) |
| --- |

Then Access the states in templates like

| <**span** \*ngIf="signUpForm.get('username').erros?.required && loginForm.get('username').touched"> Please Enter Username  </**span**> |
| --- |

**Creating Dynamic Form controls:**

1. Add a new form level like below

'passengers' : new FormArray([])

1. Sync the Form Array with the DOM container

**<div class="form-group" formArrayName="passengers">**

1. Call a method in an Event Binding to dynamically add controls

**<input type="button" value="Add Passenger" class="btn btn-primary" (click)="addPassengers()">**

**addPassengers() {**

**const passenger = new FormControl(null, Validators.required);**

**(<FormArray>this.loginForm.get('passengers')).push(passenger);**

**}**

**getPassengers(){**

**return (<FormArray>this.loginForm.get('passengers')).controls;**

**}**

1. Attach the \*ngFor Directive in the DOM where we want to clone the element based on the FormArray controls.

**<div \*ngFor="let passenger of getPassengers(); index as i;">**

**<label for="exampleInputName">Passenger Name</label>**

**<input type="text" class="form-control" aria-describedby="nameHelp" placeholder="Enter Name" [formControlName]="i">**

**</div>**

**What is FormArray?**

The FormArray is a way to manage the collection of Form Controls in Angular. The controls can be a FormGroup, FormControl, or another FormArray.

We can group Form Controls in Angular forms in two ways.

One is using the FormGroup and the other one is FormArray. The difference is how they implement it. In FormGroup controls become a property of the FormGroup. Each control is represented as a key-value pair. While in FormArray, the controls become part of an array

Because it is implemented as an Array, it makes it easier to dynamically add controls.

**What is AbstractControl?**

It provides some of the shared behaviour that all controls and groups of controls have, like running validators, calculating status, and resetting state. It also defines the properties that are shared between all subclasses, like value, valid, and dirty. It shouldn't be instantiated directly.

**Custom Validation**

Validators at their core are just functions, they take as input a FormControl instance and returns either ***null*** if it’s valid or an error ***object*** if it’s not.

function domainEmailValidator(control : FormControl) : {[key:string]:boolean}

{

let email : string = control.value;

if(email && email.indexOf("@") != -1)

{

let domain = email.split("@")[1];

console.log(domain);

if(domain != "csz.com")

{

return {emailDomain:true};

}

}

return null;

}

**Routing**

Routing Makes your application as SPA. To use Routing in our application, we have to follow the following steps,

1. **Import RouterModule and Routes**

| import { RouterModule, Routes } from '@angular/router'; |
| --- |

* RouterModule is a separate module in angular that provides required services and directives to use routing and navigation in angular application.
* Routes defines an array of roots that map a path to a component.

2. **Create Array of Routes**

| const routes: Routes = [  { path: 'pathName', component: componentName },  { path: '', redirectTo: '/manage-book ', pathMatch: 'full' } ] |
| --- |

* The path property describes the URL this route will handle.
* The component property is the name of the component we want to display when the URL in the browser matches this path.

3. **Using RouterModule.forRoot()**

| imports: [ RouterModule.forRoot(routes) ] |
| --- |

Now we need to import RouterModule.forRoot(routes) using imports metadata of @NgModule.

Here argument routes is our constant that we have defined above as array of Routes.

The forRoot() method returns an NgModule and its provider dependencies.

Creates a routing module that includes the router directives, the route configuration and the router service

4. **RouterLink**

| <**a** routerLink="/users">User list</**a**> |
| --- |

RouterLink is a directive that is used to bind a route with clickable HTML element.

5. **RouterOutlet**

| <router-outlet></router-outlet> |
| --- |

RouterOutlet is a directive provided by Angular which is used to load the different components based on the router state.

Whenever the user clicks on the link, at a time the activated component will be rendered and added to HTML DOM inside the router-outlet directive.

**Navigating to other links programmatically**

To Navigate to other pages through programmatically, we need to follow the below steps,

1. Import Router class from ‘@angular/router’
2. Create dependency injection on the current class’s constructor function.

| constructor(private router : Router) |
| --- |

3. By using the router object we can navigate to next link like,

| this.router.navigateByUrl(‘/home’); |
| --- |

**Passing Parameter to Routes**

Sometimes our url may have some parameters like this

| product/2 |
| --- |

The product number will be different for each products. So we can’t write route for each and every product. Here is the **Dynamic Routing**.

We need to handle Dynamic route for the above url.

| {path: 'product/:pid', component: ProductComponent} |
| --- |

Here the :pid will receive the url value dynamically.

**Fetching Route Params**

To fetch the route params in our class file we need to follow the below steps,

1. Import ActivatedRoute class from ‘@angular/router’

The ActivatedRoute class has many inbuilt methods which are related to the currently activated route.

2. Create DI for imported ActivatedRoute class.

| constructor(private route : ActivatedRoute) |
| --- |

3. Now we can get the route params through route object.

| this.route.params.subscribe((params)=> {  params['pid'] }); |
| --- |

The **pid** should be same as which we used in dynamic routing **(:pid)**

**Unsubscribe the above subscribe**

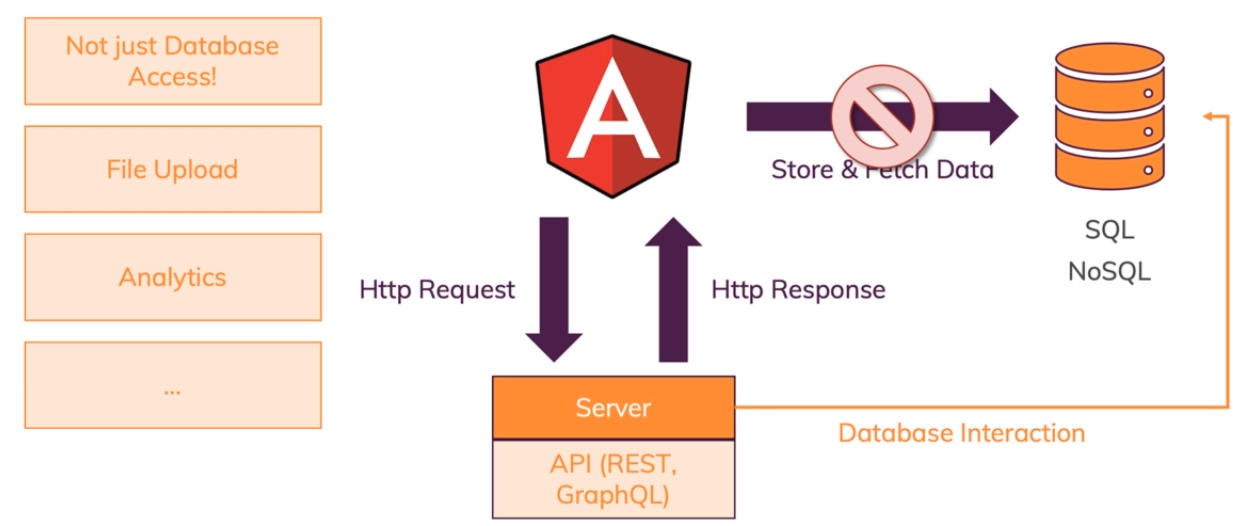
Unsubscribe the subscription is very important, if not the subscription will be active and always in memory.

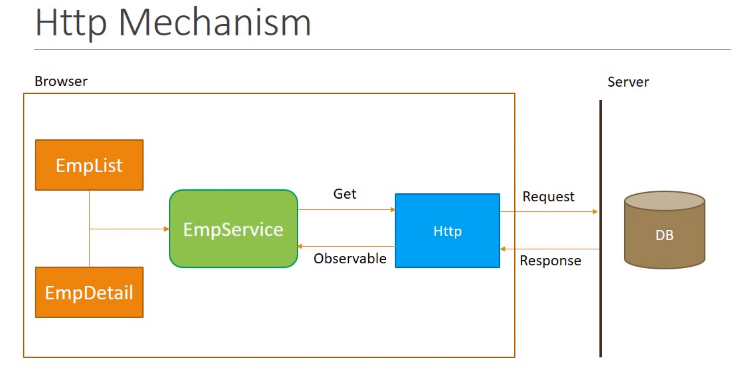
We need to unsubscribe the subscription while our current component gets destroyed.

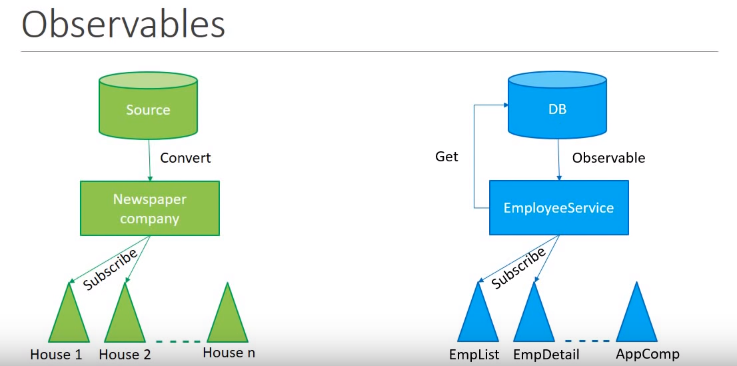
So implement OnDestroy interface in the current class and the interface method is

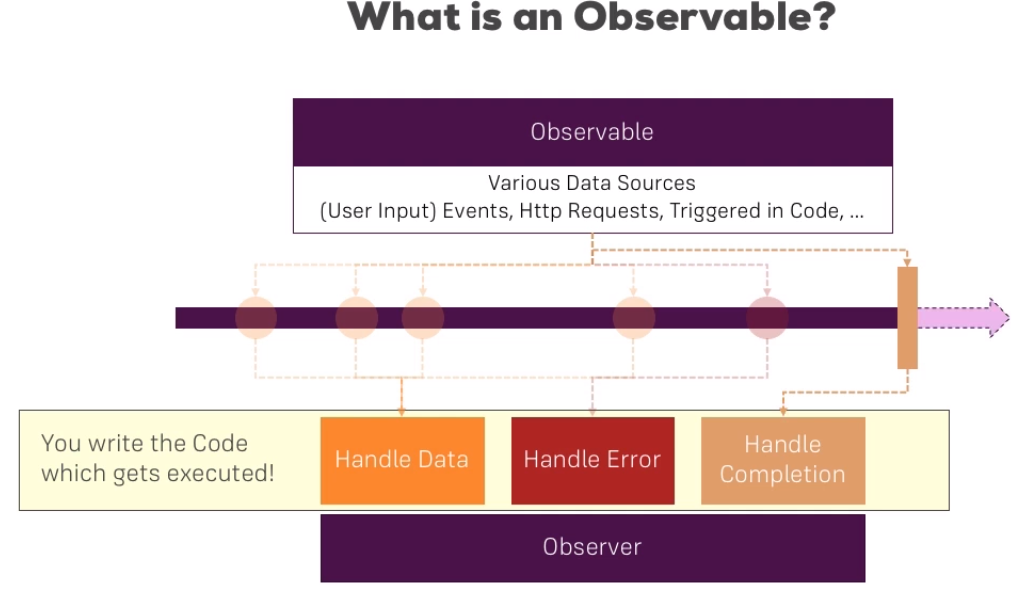
ngOnDestroy.

**Http**

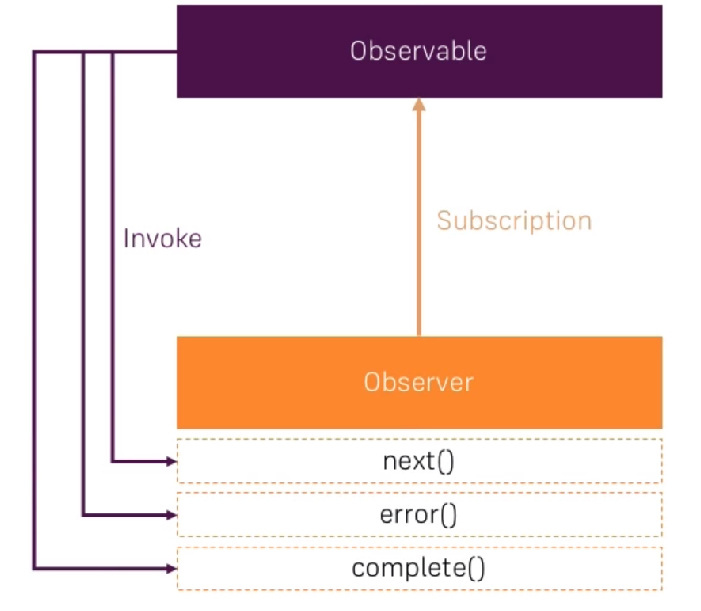
****

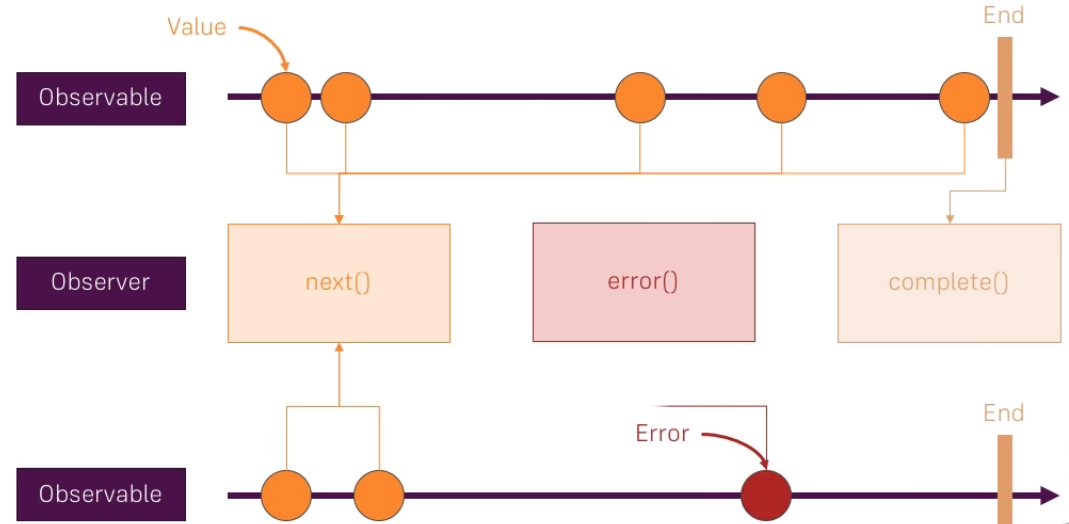
****

****

****

**Observable is a wrapper around data stream**

****



* Observables is a way to populate the data from the different external resources **asynchronously**.
* The main purpose of using Observables is to observe the behavior of a data or event.

The **Subscribe** method is how you connect an observer to an Observable. Your observer implements some subset of the following methods:

**onNext**

An Observable calls this method whenever the Observable emits an item. This method takes a parameter as the item emitted by the Observable.

**onError**

An Observable calls this method to indicate that it has failed to generate the expected data or has encountered some other error. It will not make further calls to onNext or onCompleted. The onError method takes as its parameter an indication of what caused the error.

**onCompleted**

An Observable calls this method after it has called onNext for the final time, if it has not encountered any errors.

**Using HTTP in Angular**

To enable Http service in our Angular Application, we need to follow the below steps,

1. **Setup and Configure Angular HttpClient**

Register the HttpClientModule in our root module.

| import { HttpClientModule } from '@angular/common/http'; |
| --- |

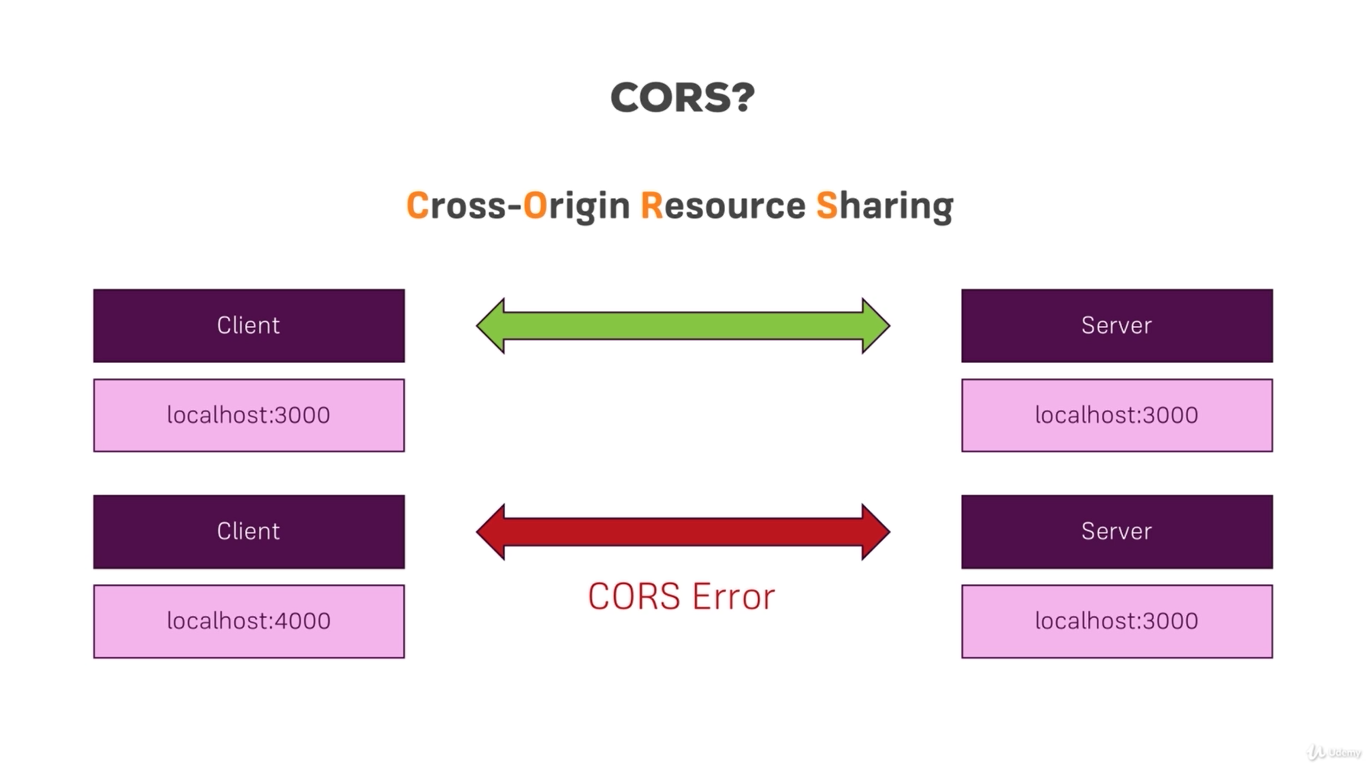
Add that module in `**@NgModule**` imports.

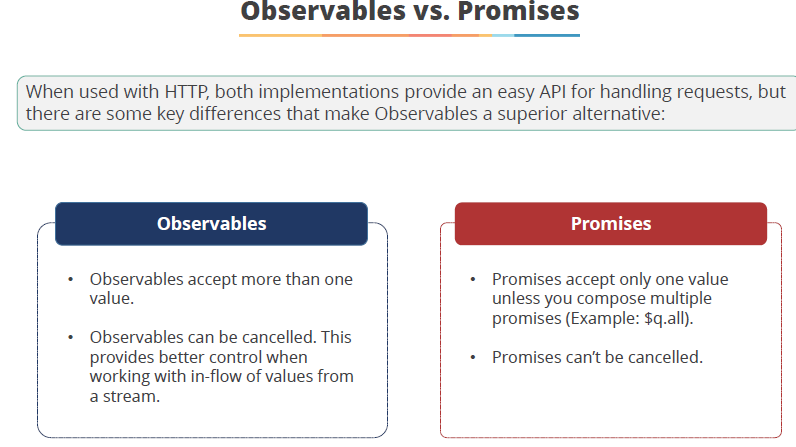
| imports: [  BrowserModule,  HttpClientModule ], |
| --- |

That it's, now you can use the Angular HttpClient in your application.

**2. Create object for HttpClient through DI on the required classes or services.**

| import { HttpClient } from '@angular/common/http';  constructor(private http: HttpClient) { } |
| --- |

****

****

**Create Route Guard in Angular**

Guarding routes means whether we can visit the route or not. We can use the route guard in angular by using the simple steps,

1. **Created Route Guard by using the below command.**

| ng g guard auth |
| --- |

1. Once generated you can find auth.guard.ts, then import the class file in root module and register in providers (*providers[AuthGuard]*).
2. **Using *CanActivate* interface.**

CanActivate is an Angular interface. It is used to force user to login into application before navigating to the route. *CanActivate* interface has a method named as *canActivate*()

4. **Use canActivate property of Route**

Use *canActivate* property of *Route* interface to guard the route and assign service class implementing CanActivate interface, for example AuthGuard. Now find the canActivate property used in route declarations.

For example,

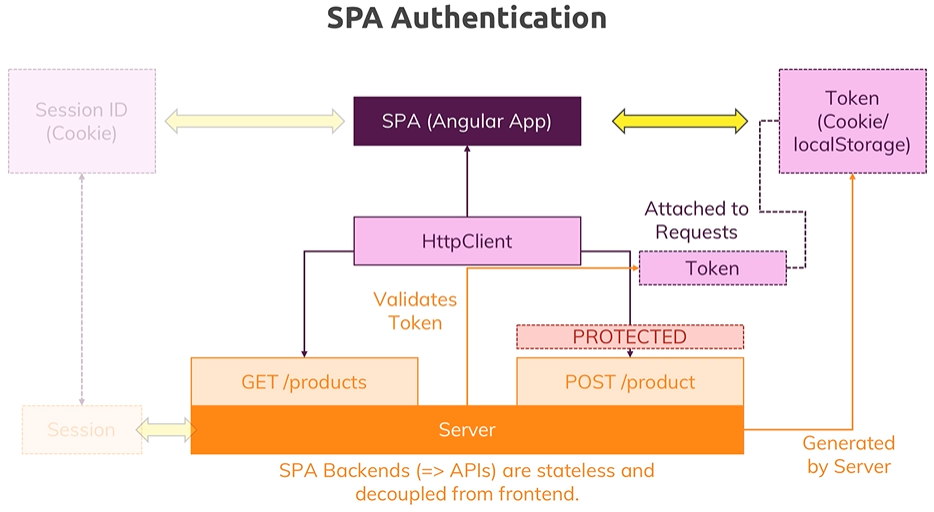
| {  path: 'cart',  component: CartComponent,  canActivate: [ AuthGuard, AdminGurd ] } |
| --- |

If *canActivate()* method from AuthGuard returns **true** only when route can be navigated. In case of false value, navigation can be redirected to login page.

Till now we have used Client side Authentication, but it’s not a complete solution. We should validate with server side.

To Achieve Server side authorization, we need to send the client side token which is stored in Local storage on every request.

**Understanding SPA Authentication**

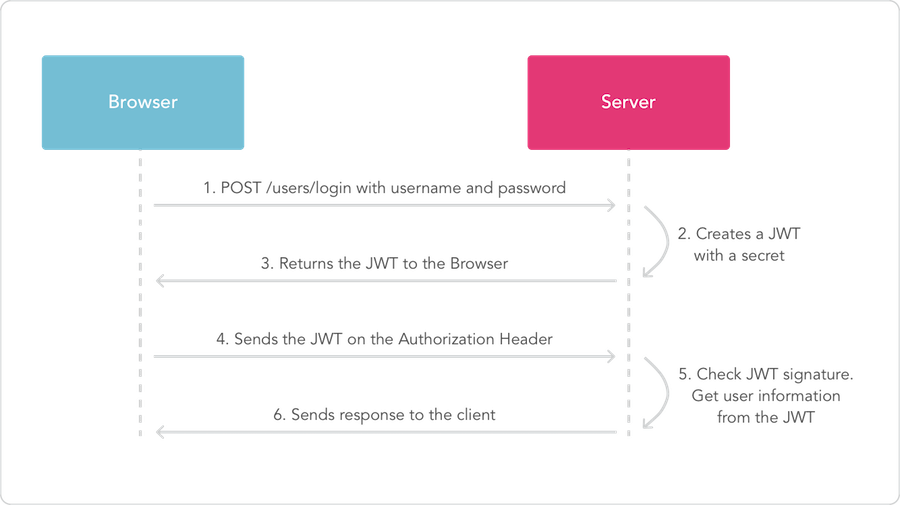
****

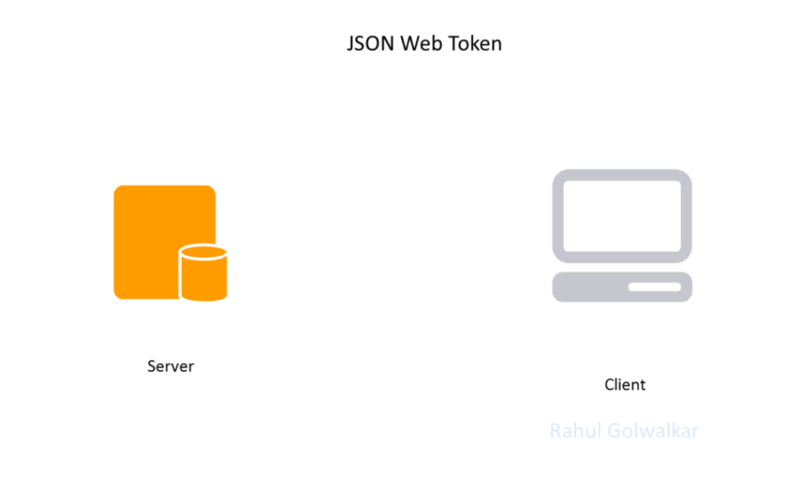
**Angular - JWT - jsonwebtoken NPM**

JWT defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

**When should you use JSON Web Tokens?**

* Authorization
* Information Exchange





**What is the JSON Web Token structure?**

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

* Header
* Payload
* Signature

Therefore, a JWT typically looks like the following.

xxxxx.yyyyy.zzzzz

**Header**

The header typically consists of two parts: the type of the token, which is JWT, and the hashing algorithm being used, such as HMAC SHA256 or RSA.

For example:

| {  "alg": "HS256",  "typ": "JWT" } |
| --- |

Then, this JSON is Base64Url encoded to form the first part of the JWT.

**Payload**

The second part of the token is the payload, which is statement about an entity (typically, the user) and additional data.

An example payload could be:

| {  "sub": "1234567890",  "name": "John Doe",  "admin": true } |
| --- |

The payload is then Base64Url encoded to form the second part of the JSON Web Token.

**Signature**

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

| HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  secret) |
| --- |

**HTTP interceptors**

HTTP interceptors are used to automatically attach authentication information to requests.

A HTTP interceptor is a piece of code that gets activated for every single HTTP request received by your application. Interceptors are very useful when you need to perform some common processing for every HTTP request.

This can take several different forms but most often involves attaching a JSON Web Token (or other form of access token) as an Authorization header.

**Create an Interceptor**

The goal is to include the JWT which is in local storage as the Authorization header in any HTTP request that is sent. The first step is to create an interceptor.

**Send client side token on every http request**

1. **Create a new interceptor Service**

| ng g s token-interceptor |
| --- |

**2. Implement *HttpInterceptor* Interface**

Implement the ***HttpInterceptor*** on the newly created service class. This interface has the method ***intercept.***

The ***intercept*** method has 2 default parameters. Those are **HttpRequest**, **next.**

**3. Override the *intercept* method.**

| intercept(req:HttpRequest<any>, next:HttpHandler)  {  var tokenizedReq = req.clone({  setHeaders : {  Authorization: (this.userSer.getToken()) ? this.userSer.getToken() : ''  }  });  return next.handle(tokenizedReq);  } |
| --- |

**4. Register the token-interceptor service in Root Module**

In Providers, inject the below object

| {  provide: HTTP\_INTERCEPTORS,  useClass: TokenInterpectorService,  multi:true  } |
| --- |

Once completed the above 4 steps, we can easily transfer our client side token to Server side.

**Useful APIs**

**Weather:**

<http://api.openweathermap.org/data/2.5/weather?q=chennai&appid=3a3eb62e70b9745f96cb7c04245a9cb8>

**Live News:** (408b4153b994422d8638da72f2d3ac5b)

<https://newsapi.org/v2/top-headlines?country=in&apiKey=408b4153b994422d8638da72f2d3ac5b>

**With News Category:**

i. business, ii. entertainment, iii. health, iv. science, v. sports, vi. Technology

<https://newsapi.org/v2/top-headlines?country=in&category=sports&apiKey=408b4153b994422d8638da72f2d3ac5b>

**Send Customized Header:**

| this.http.get("https://developers.zomato.com/api/v2.1/categories", {  headers : new HttpHeaders({  'user-key' : 'e5e735840413b5ee58d8bb60dfea2e0f'   }), params : queryParams }) |
| --- |

**Set Query Params:**

| let queryParams = new HttpParams(); queryParams = queryParams.append("paramName1", "paramValue1"); queryParams = queryParams.append("paramName2", "paramValue2"); |
| --- |

**File Upload - client side setup**

**FormData()**

**JavaScript Provides the object FormData**

* FormData is basically data format which allows text value and BLOB (Binary large object)
* The FormData provides a way to easily construct a set of key/value pairs representing form fields and their values.
* It uses the same format of encoding type were set to "**multipart/form-data**".

**FormData.append()**

The append() method of the FormData interface appends a new value onto an existing key inside a FormData object, or adds the key if it does not already exist.

**Syntax:**

There are two versions of this method: a two and a three parameter version:

| formData.append(name, value);  formData.append(name, value, filename); |
| --- |

**Parameters:-**

**name:**

The name of the field(property) whose data is contained in value.

**value:**

The field's value. (including File).

**filename Optional:**

The filename reported to the server, when File is passed as the second parameter.