CREDO SYSTEMZ
simplifying IT

Angular - Training

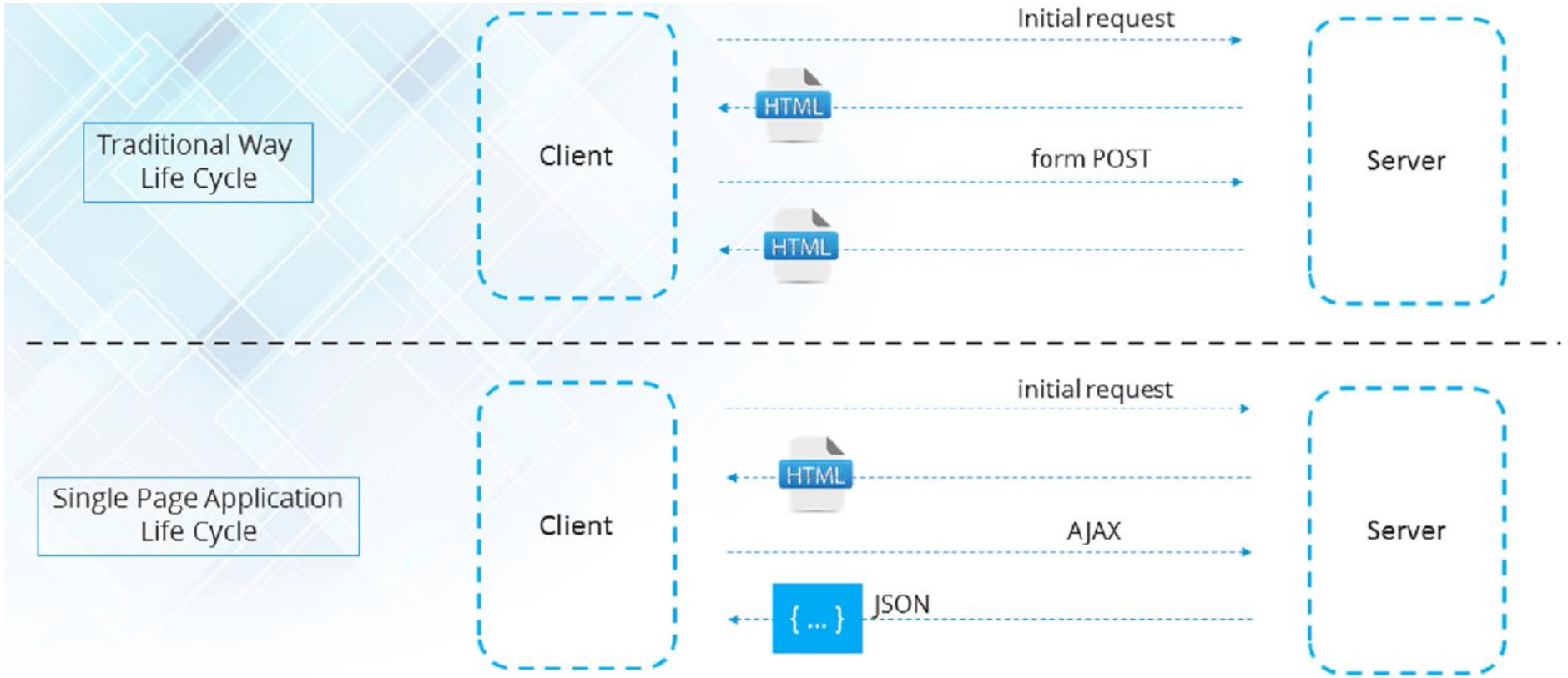
Agenda

- Introduction to Angular
- Angular Environment Setups
- Components In Angular
- Data & Event Bindings
- Directives
- Pipes In Angular
- Angular Forms
- Service And Dependency Injection
- Routing
- Http & Observable In Angular
- Eager, Lazy & Asynchronous Loading
- Angular App Deployment

Angular 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.

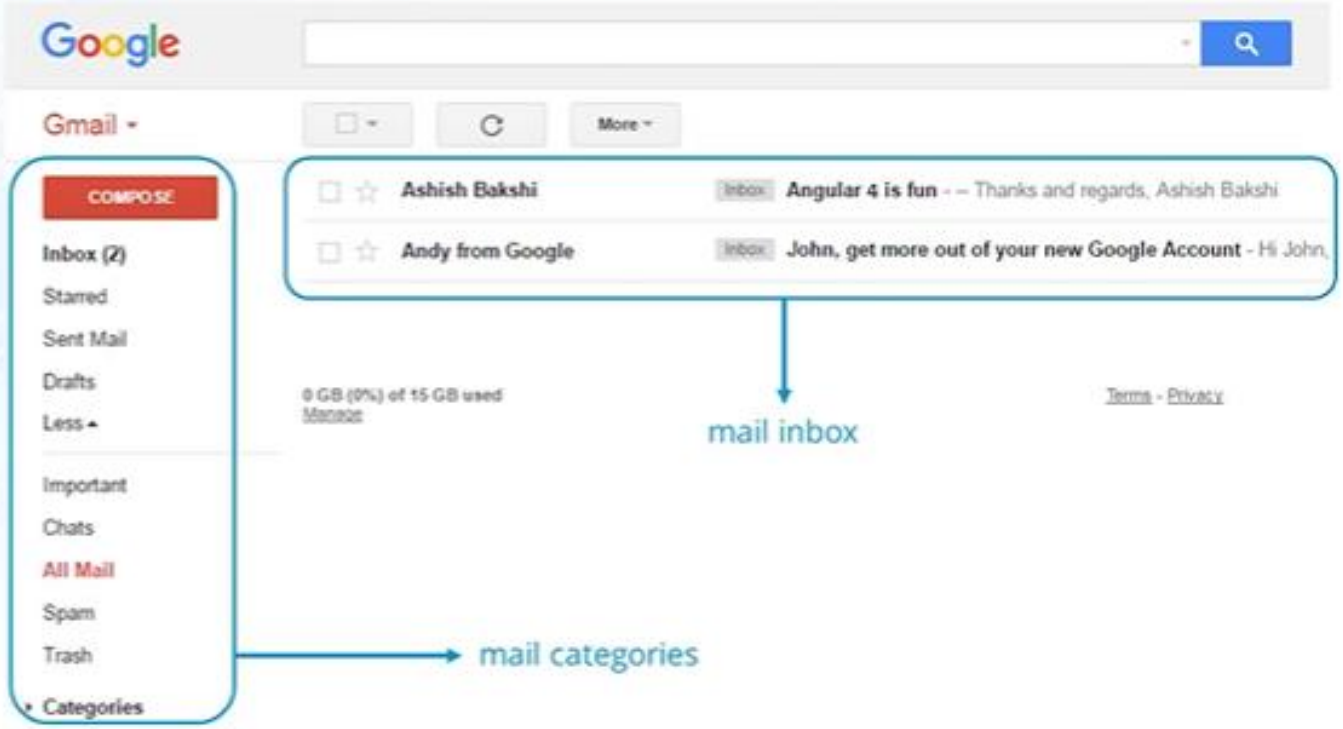
Angular Introduction continue..



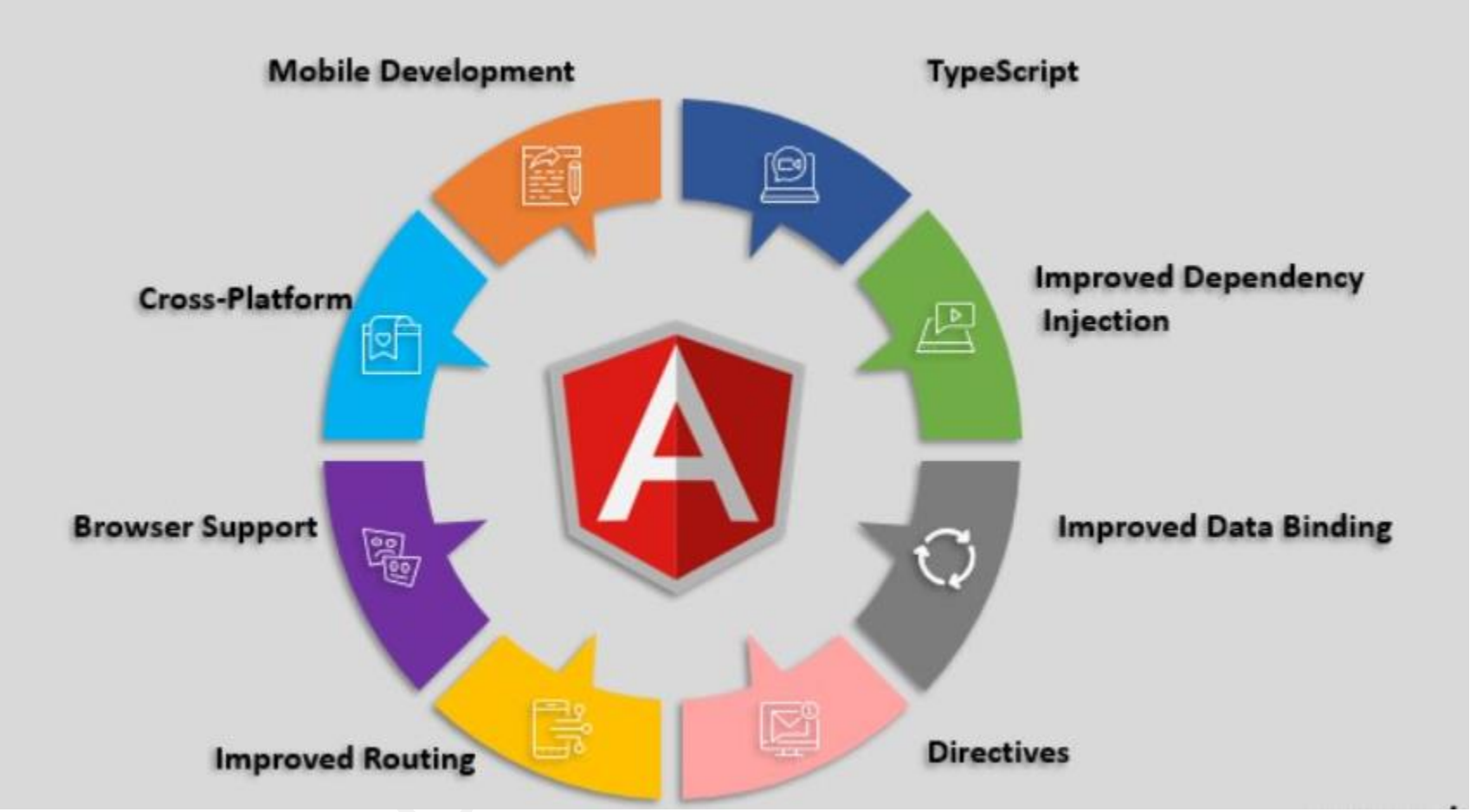
Angular Introduction continue..

A Single Page Application is a web application that requires only a single page load in a web browser.

- Whole page is not reloaded every time
- Your browser fully renders the DOM once
- Later any server interactions is performed by JavaScript which modifies the view



Angular Features



Angular Version History

AngularJS

1.0 1.1 1.2 ... 1.7

Angular

2.0 4.0 5.0 6.0 7.0 ...

Angular Version History continue..

Angular 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

Angular Environment Setup continue

- 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/

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

  ng serve --port 4201

Angular CLI - Overview

CREDO SYSTEMZ
simplifying IT

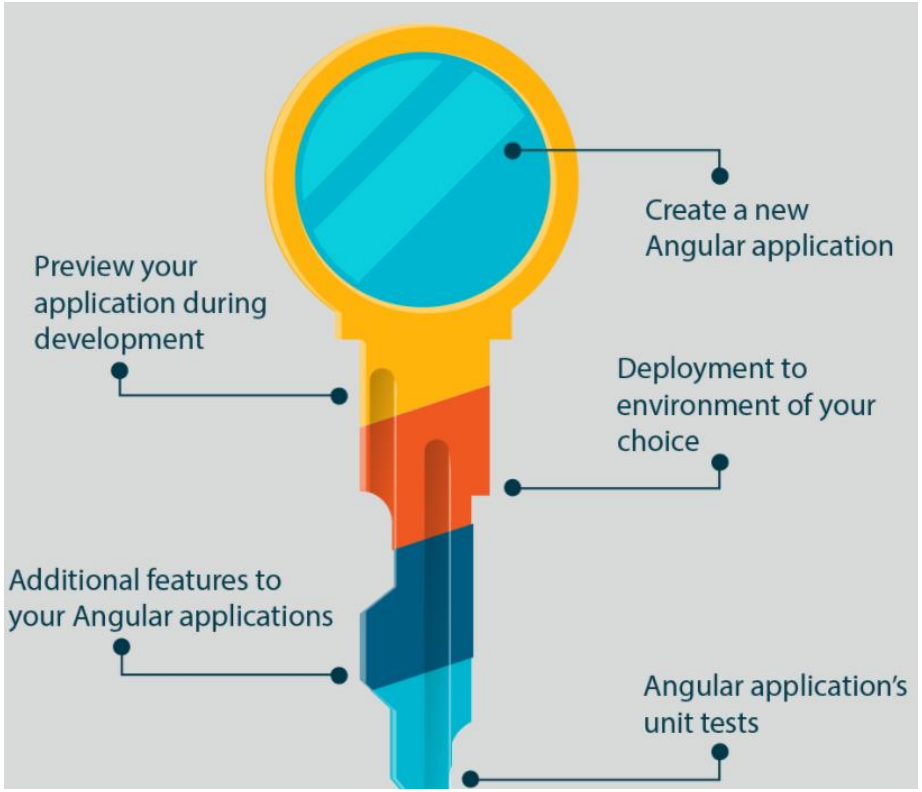
1.Create a separate application folder and add the package definition file (ie. package.json) and other configuration files.

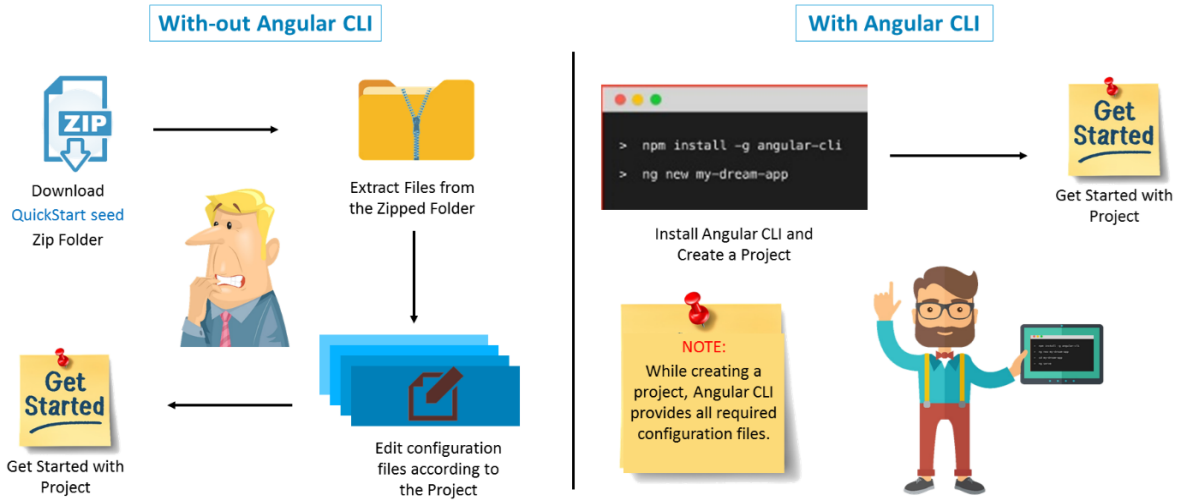
2. Install the packages using NPM

3. Setup the environment.

4. Provides required files from angular program.

5. Create index.html which hosts our application.





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, icons and External CSS & JS files.
- ***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.

Structure of Angular Project continue..

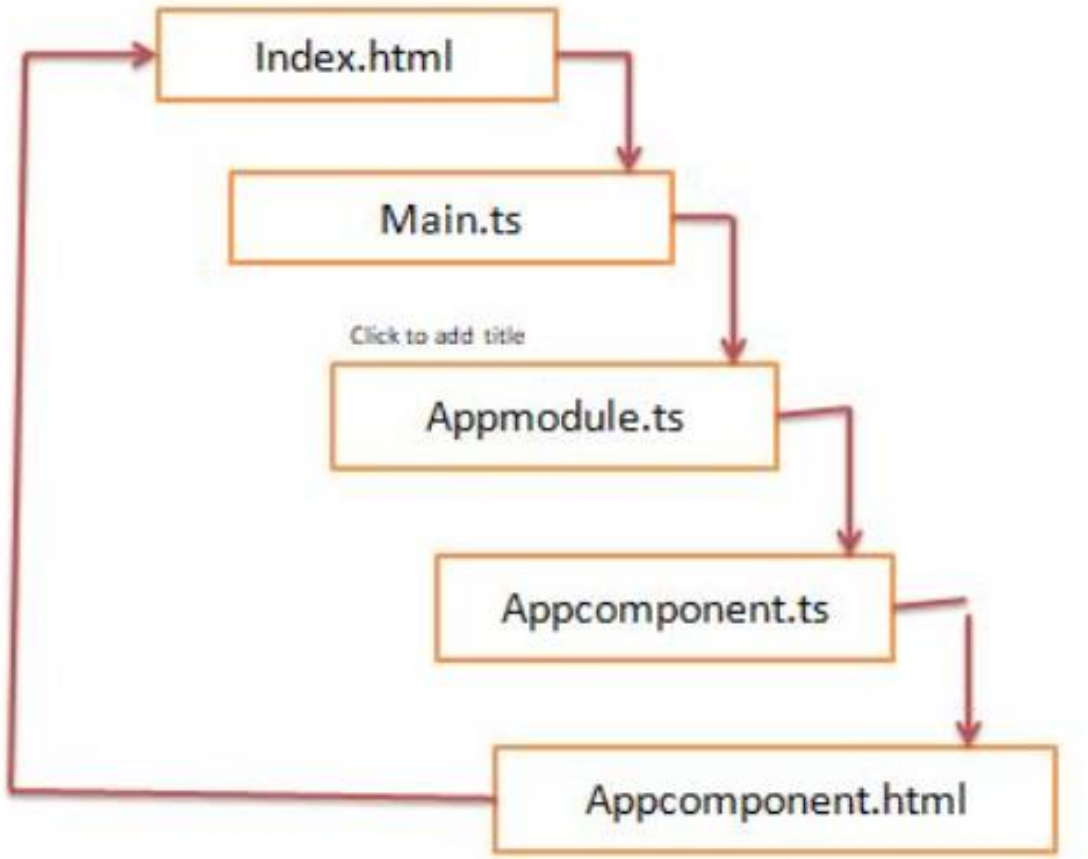
CREDO SYSTEMZ
simplifying IT

- ***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.

Structure of Angular Project continue..

- ***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.

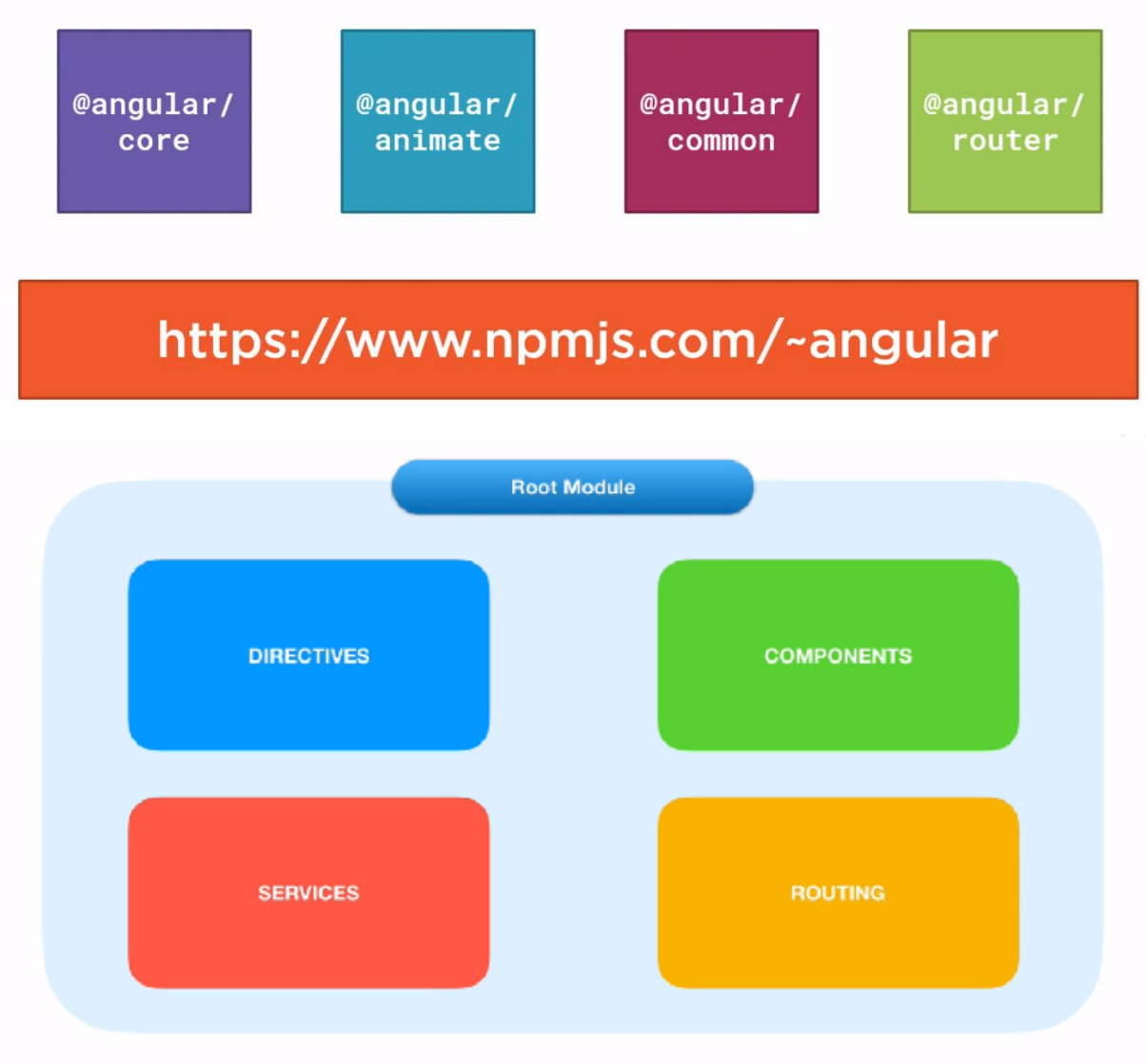
Application Workflow



Main Building Blocks of Angular

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

Modules

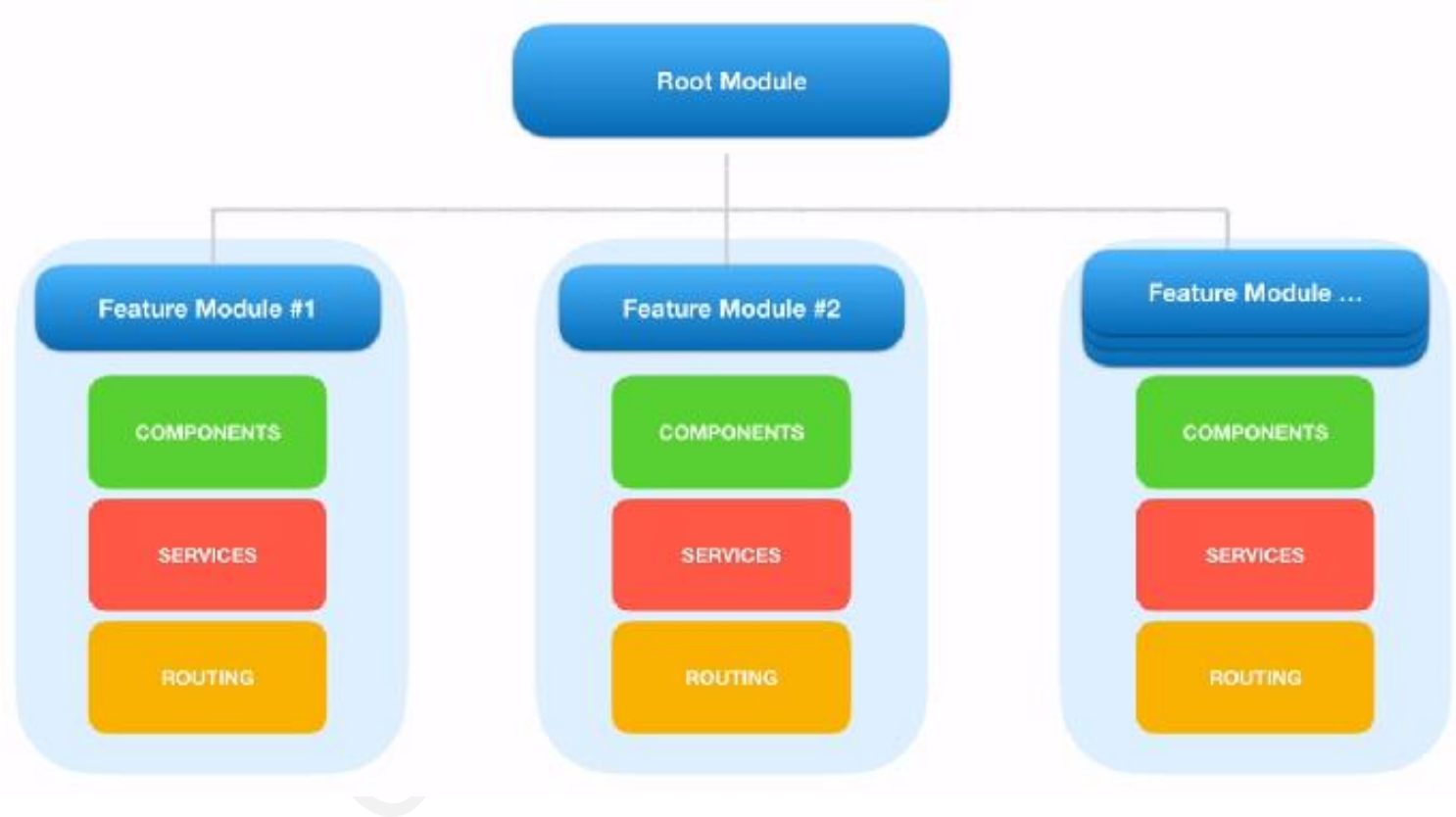


- Bundle of functionalities of our App.
- Every Angular app **contains at least one Angular module**, i.e. the root module.
- Generally, it is named as AppModule.
- We can create multiple Modules if needed.
- Any angular module is a class with **@NgModule decorator**.
- Encapsulation of different similar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

Modules continue..

CREDO SYSTEMZ
simplifying IT

Importing other modules in Root Module



Modules continue..

CREDO SYSTEMZ
simplifying IT

@NgModule 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: 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.

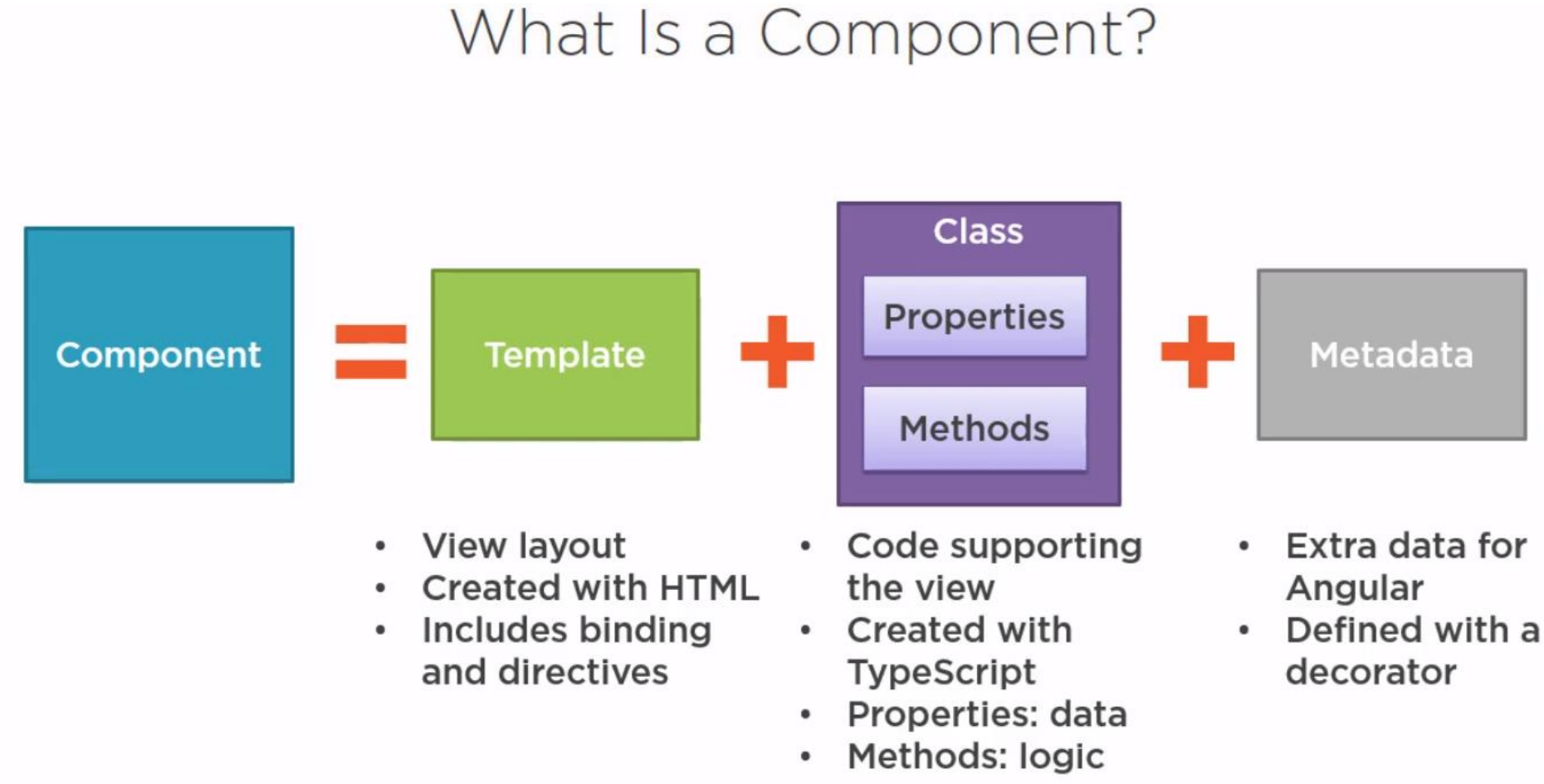
Angular libraries

- 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.

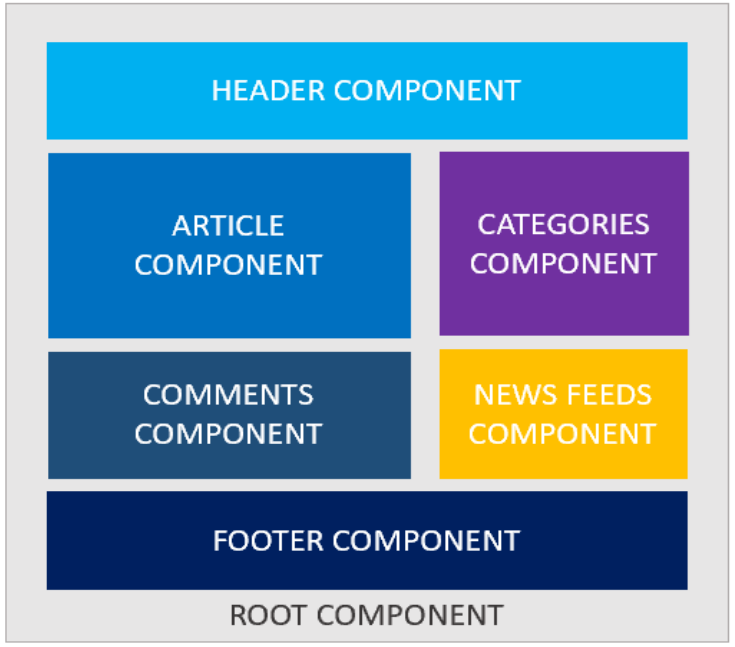
```
import { Component } from '@angular/core';
```

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

Components



Components continue..



- 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.

Components continue..

CREDO SYSTEMZ
simplifying IT

Metadata:

Metadata tells Angular how to process a class.

- 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

- **A TypeScript file**: where we define the state (the data to display) and behavior (logic) of our component.
- **An HTML file**: contains the markup to render in the DOM.
- **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.
- **A spec file**: includes the unit tests.

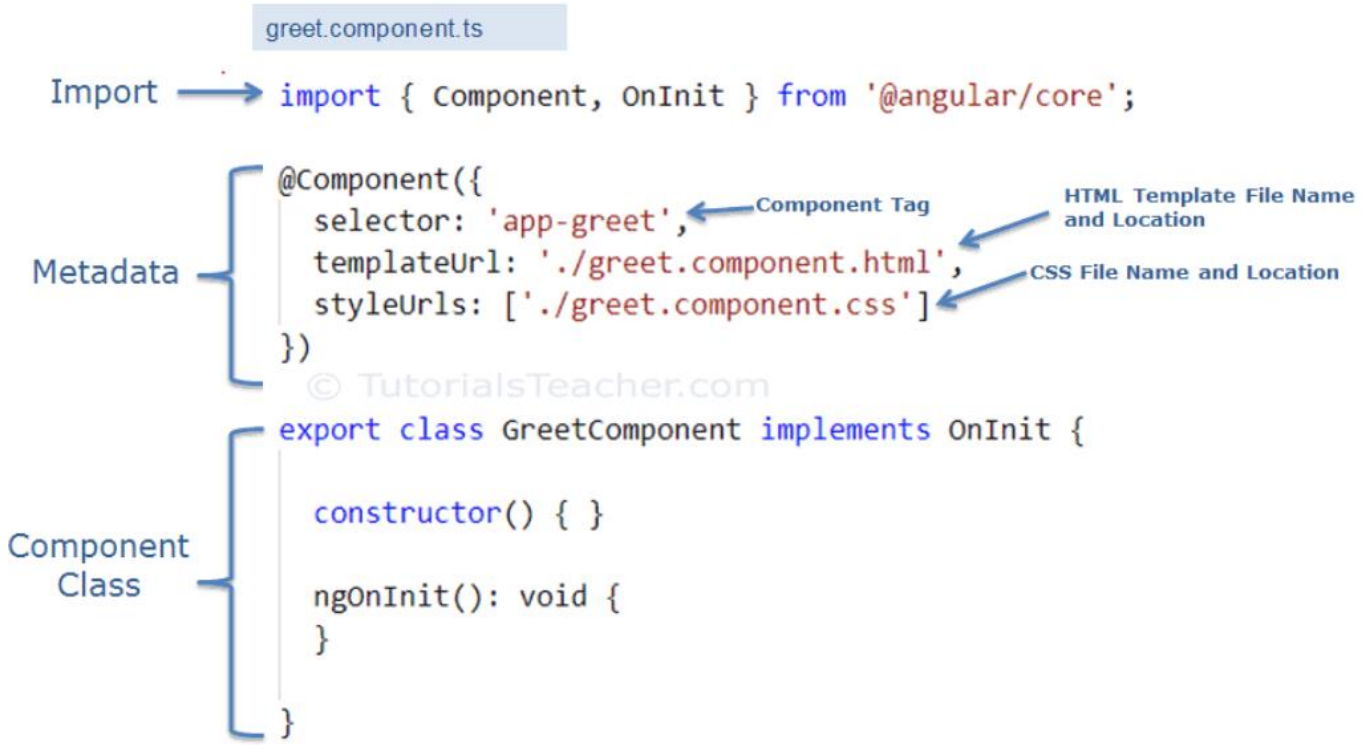
Taken away from Modules & Components

- 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

CREDO SYSTEMZ
simplifying IT

```
ng generate component componentName

ng g c componentName
```



Component in Action

CREDO SYSTEMZ
simplifying IT

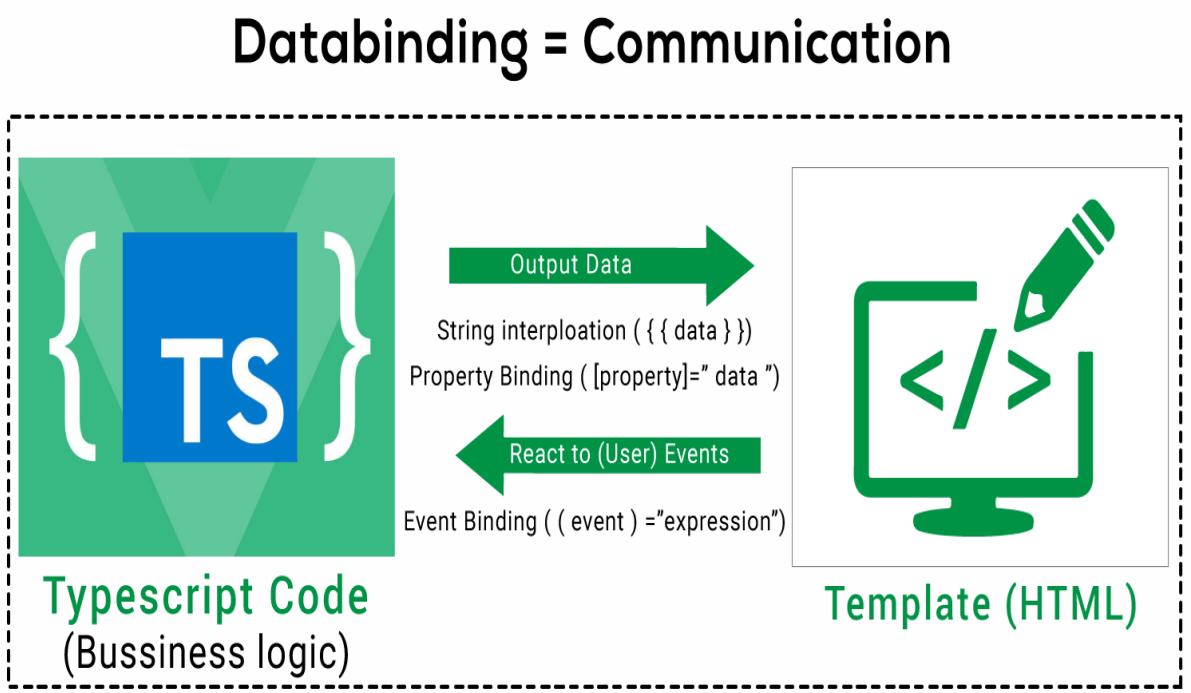
- **Component Class:** GreetComponent is the component class. It contains properties and methods to interact with the view
- **Component Metadata**: The @Component is a decorator used to specify the metadata for the component class, It is a function and can include different configs for the component
- **The import statement gets the required feature from the Angular or other libraries**

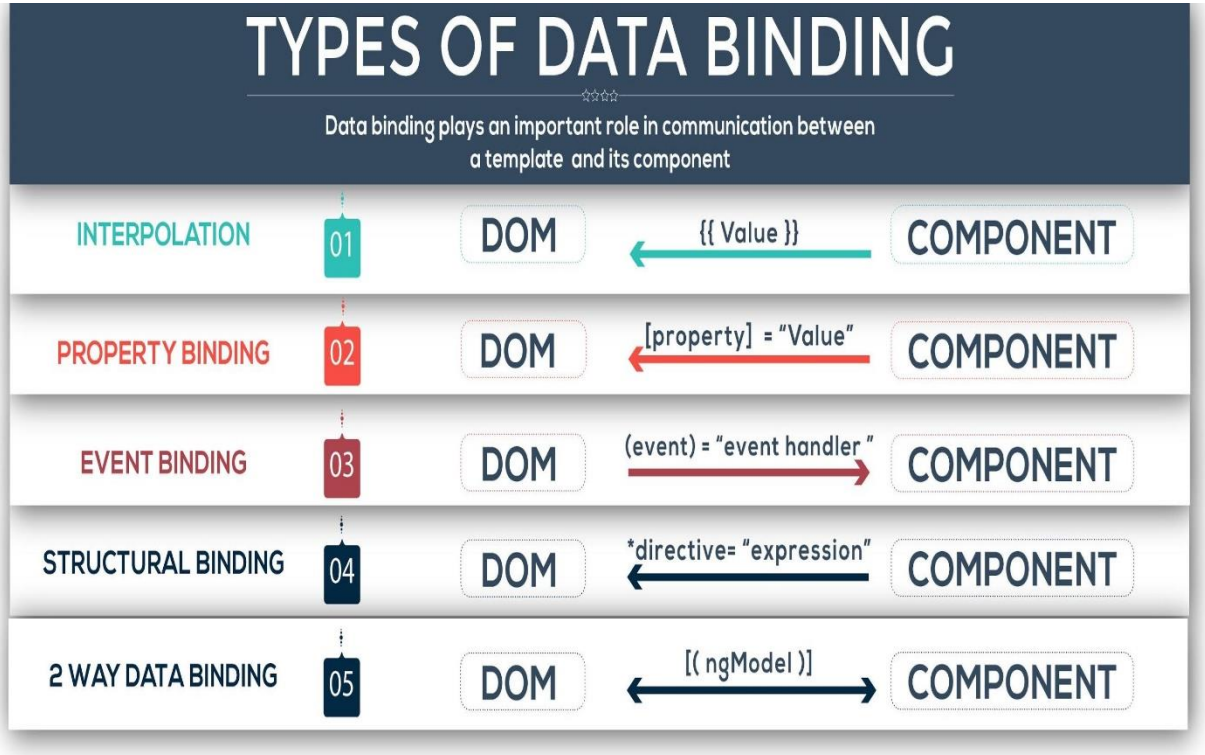
```
export class GreetComponent {
constructor() {
 }
name: string = “Raja";

  displayName(): void {
     alert("Hello " + this.name);
  };
}
```

```
<div>
   Enter Your Name: <input type="text" value={{name}} />
<br />
   <button (click)=" displayName()">Show Name</button>
</div>
```

Data Binding

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





Data Binding continue..

CREDO SYSTEMZ
simplifying IT

Interpolation {{…}}:

Interpolation is used for one-way data binding in Angular. Use variable name inside the double curly braces "{{ }}". This expression is also known as template expression

```
<p> 2 + 2 = {{ 2 + 2 }} </p> <!-- output: "2 + 2 = 4" -->
<p> 2 * 3 = {{ 2 * 3 }} </p> <!-- output:`"2 * 3 = 6" -->
Welcome,  {{firstName}} {{lastName}}
<p>{{ num }} </p>
<p>{{ getCurrentTime() }} </p>
```

```
export class AppComponent {
  firstName= ‘John';
  lastName=”Mathew”
}
```

Property Binding […]:

Property binding is one way from component to view. It lets you set a property of an element in the view to property in the component. Set properties such as class, href, src, textContent, etc using property binding

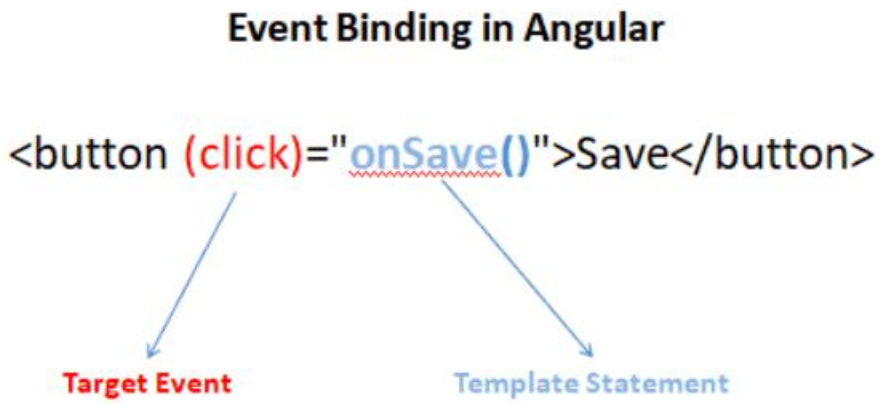
```
//syntax
[binding-target]=”binding-source”

<h1 [innerText]="title"></h1>
<button [disabled]="isDisabled">I am disabled</button>
<img [src]="imageUrl">
```

```
export class AppComponent {
  title="Angular Property Binding Example"
isDisabled= true;
imageUrl="https://angular.io/assets/images/logos/angular/logo-nav@2x.png"
}
```

Data Binding continue..

Event Binding (…):
Event binding allows us to bind events such as keystroke, clicks, hover, touche, etc to a method in component. It is one way from view to component

```
//syntax
(target-event)="TemplateStatement“

<button (click)="onSave()">;Save</button>
```



Data Binding continue..

In the below example, the component listens to the click event on the button. It then executes the clickMe() method and increases the clickCount by one

```
<h1 [innerText]="title"></h1>

<h2>Example 1</h2>
<button (click)="clickMe()">Click Me</button>
<p>You have clicked {{clickCount}}</p>
```

```
clickCount=0
  clickMe() {
    this.clickCount++;
  }
```

Directives

The Angular directive helps us to manipulate the DOM

There are three kinds of directives in Angular:

- **Components** - directives with a template.
- **Structural directives** - change the DOM layout by adding and removing DOM elements.

  (leading with *).
- **Attribute directives** - change the appearance or behavior of an element.

List of Directives in Angular

CREDO SYSTEMZ
simplifying IT

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

[hidden]: It just hides and shows the attached element based on the condition.

Directives continue..

The ngIf is an Angular Structural Directive, which allows us to add/remove DOM Element based on some condition.

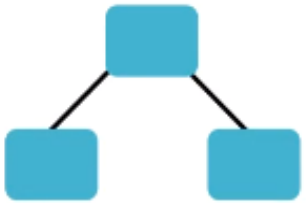
```
<p [hidden]="condition">
   content to render, when the condition is true
</p>
```

```
<p *ngIf="condition">
   content to render, when the condition is true
</p>
```

```
<div *ngIf="condition; then thenBlock else elseBlock">
   This content is not shown
</div>
<ng-template #thenBlock>
   content to render when the condition is true.
</ng-template>
<ng-template #elseBlock>
   content to render when condition is false.
</ng-template>
```

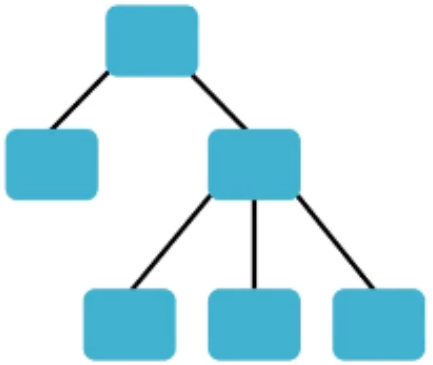
[hidden]

For small element trees



*ngIf

For large element trees



Directives continue..

CREDO SYSTEMZ
simplifying IT

***ngFor**

Angular ngFor directive iterates over a collection of data like an array, list, etc and create an HTML element for each of the items from an HTML template

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:

```
<html-element *ngFor="let <item> of <items>;”>
   <html-Template>item</html-Template>
</html-element>
```

Directives - *ngFor

CREDO SYSTEMZ
simplifying IT

- * → Angular Template Syntax
- **<item> of <items>;**

  <item> → is the Template input variable. It represents the currently iterated item from the <items>

  <items> → is a collection, which we need to show it to the user
- The scope of the **item** is within the <html-element>..</html-element>. You can access it anywhere within that, but not outside of it.

```
<ul >
   <li *ngFor="let fruit of fruits">{{fruit}}</li>
</ul>
```

```
fruits:string[] = ['Apple','Orange','Grapes','Pineapple']
```

```
<ul>
   <li *ngFor="let fruit of fruits; let i = index;">
     {{i}}. {{fruit}}
   </li>
</ul>
```

Directives - *ngFor

- Iterating object using *ngFor

```
const movies = [
   {id: 1, name:'Forrest Gump'},
   {id: 2, name:'Schindler's List'},
   {id: 3, name:'The Godfather'},
   {id: 4, name:'Goodfellas'},
   {id: 5, name:'The Dark Knight'}
];
```

```
<table>
    <thead>
      <th>Movie Name</th>
      <th>Index</th>
    </thead>
    <tbody>
      <tr *ngFor="let movie of movies">
        <td>{{movie.name}}</td>
      </tr>
    </tbody>
  </table>
```


Directives - ngStyle

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

This sets the background color of the div to green

```
<div [ngStyle]="{'background-color':'green'}"></<div>
```

ngStyle becomes much more useful when the value is dynamic

```
<div [ngStyle]="{'background-color':gender === ‘Male' ? ‘blue' : ‘pink' }"></<div>
```

ngStyle with Function - We set the color of the text according to the value that’s returned from the getColor function

```
<ul *ngFor="let person of people">
  <li [ngStyle]="{'color':getColor(person.country)}"> {{ person.name }}
({{ person.country }})
  </li>
</ul>
```

```
people: any[] = [
   {
    "name": “John Mathew",
    "country": 'UK'
   },
   {
    "name": “Thomas Peet",
    "country": 'USA'
   } ];
```

Directives - NgClass

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

The value is true this will set the class text-success onto the element the directive is attached to

```
[ngClass]="{'text-success':true}"
```

The value can also be an expression like below

```
[ngClass]="{'text-success':person.country === 'UK'}"
```

```
<h4>NgClass</h4>
<ul *ngFor="let person of people">
  <li [ngClass]="{
    'text-success':person.country === 'UK',
    'text-primary':person.country === 'USA',
    'text-danger':person.country === 'HK'
  }">{{ person.name }} ({{ person.country }})
  </li>
</ul>
```

Angular 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 (for lowercase-ing Strings)
- Uppercase (for uppercase-ing Strings)
- Slice (for slicing Strings)
- Json (JavaScript object into a JSON string)
- Currency (for formatting currencies)
- Date (for parsing Date objects)

Angular Pipes continue..

CREDO SYSTEMZ
simplifying IT

```
this.todayDate = new Date();
this.amount = 100;
this.message = “Pipes in Angular";
jsonval = {name:'Rox', age:'25', address:{a1:'Mumbai', a2:'Karnataka'}};
months = ["Jan", "Feb", "Mar", "April", "May", "Jun", "July", "Aug", "Sept", "Oct", "Nov", "Dec"];
```

```
<h2>Date Pipes</h2>
   Full Date : {{todayDate}}<br />
   Short Date : {{todayDate | date:'shortDate'}}<br />
   Medium Date : {{todayDate | date:'mediumDate'}}<br />
   Full Date : {{todayDate | date:'fullDate'}}<br />
   Time : {{todayDate | date:'HH:MM'}}<br />
   Time : {{todayDate | date:'hh:mm:ss a'}}<br />
```

```
<h2>Number Pipes</h2>
   No Formatting : {{amount}}<br />
   2 Decimal Place : {{amount |number:'2.2-2'}}
```

```
<h2>String Related Pipes</h2>
   Actual Message : {{message}}<br />
   Lower Case : {{message | lowercase}}<br />
   Upper Case : {{message | uppercase}}<br />
```

```
<h1>Json Pipe</h1>
    <b>{{ jsonval | json }}</b>
```

```
<h1>Slice Pipe</h1>
    <b>{{months | slice:2:6}}</b>
    // here 2 and 6 refers to the start and the end index
```

Angular Pipes continue..

CREDO SYSTEMZ
simplifying IT

```
  // Chained pipes
<div>
  Date with uppercase :- {{presentDate | date:'fullDate' | uppercase}} <br/>
  Date with lowercase :- {{presentDate | date:'medium' | lowercase}} <br/>
</div>
```

```
Fruits = ["Apple","Orange","Grapes","Mango","Kiwi","Pomegranate"];

<div>
    <h3>Start index:- {{Fruits | slice:2}}</h3>
    <h4>Start and end index:- {{Fruits | slice:1:4}}</h4>
    <h5>Negative index:- {{Fruits | slice:-2}}</h5>
    <h6>Negative start and end index:- {{Fruits | slice:-4:-2}}</h6>
  </div>
```

```
decimalNum1: number = 0.8178;
 <p> {{decimalNum1 | percent:'2.2'}} </p>
```

Routing

CREDO SYSTEMZ
simplifying IT

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**.

Routing continue..

CREDO SYSTEMZ
simplifying IT

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.

4. RouterLink

```
<a routerLink="/users">User list</a>
```

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

5. RouterLinkActive

- RouterLinkActive is a directive for adding or removing classes from an HTML element.
- When the user navigates to any route, the router **adds the "active" class** to the activated element. And when the user navigates away the class will be removed

```
<a routerLink="/users“ routerLinkActive="active">User list</a>
```

Routing continue..

CREDO SYSTEMZ
simplifying IT

6. 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.

```
<div>
  <ul>
     <li><a routerLink="/" >Home</a></li>
     <li><a routerLink="/products" >Products</a></li>
     <li><a routerLink="/about" >About Us</a></li>
     <li><a routerLink="/contact" >Contact Us</a></li>
  </ul>
</div>
<router-outlet></router-outlet>
```

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'products', component: ProductComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent, },
  { path: '', redirectTo: 'HomeComponent', pathMatch: 'full' },
];
```

- **redirectTo** property within the routes array to tell angular route service if the users navigate to the empty URL they should be redirected to the home URL rather than the empty URL

Routing continue..

CREDO SYSTEMZ
simplifying IT

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');
```

Wild Card Route

if user navigates to a path that does not exist, Ideally we should also handle such scenario and show a page not found

```
{ path: '**', component: COMPONENT_NAME}
```

Routing continue..

CREDO SYSTEMZ
simplifying IT

Pass parameters to Angular Route

We can define parameter by adding forward slash followed colon and a placeholder (id) 1.

```
{ path: 'product/:id', component: ProductDetailComponent }
```

```
<a [routerLink]="['/Product', '2']">{{product.name}} </a>
```

Reading Route Parameters

The ActivatedRoute service provides a **params** Observable which we can subscribe to get the route parameters,

```
import { ActivatedRoute } from '@angular/router';
constructor(private route: ActivatedRoute) {}

ngOnInit() {
this.route.params.subscribe(params => {
  console.log(params)
  this.id = params['id'];
});
 }
```

Component Interaction

CREDO SYSTEMZ
simplifying IT

Data sharing between the angular components is possible with the following ways

- Sharing Data from Parent to Child: **@Input**
- Sharing Data from Child to Parent: **@Output & EventEmitter**
- Sharing Data from Child to Parent: **@ViewChild**
- Sharing Data Between Various Components: **Service**

@Input Decorator

Angular provides a decorator called @Input decorator which can be used to pass data from Parent to Child. It is a most common methods of sharing data from Parent to Child in Angular application

```
import { Component, Input  } from '@angular/core';
constructor(private route: ActivatedRoute) {}

export class ChildComponent {
   @Input() count: number;
}
```

```
<child-component [count]=Counter></child-
component>

export class AppComponent {
  title = 'Component Interaction';
  Counter = 5;
}
```

Component Interaction

CREDO SYSTEMZ
simplifying IT

@ViewChild

ViewChild is used to access DOM elements in the template from the containing component class or parent component class

```
export class ChildComponent {
 count = 0;

 increment() {
  this.count++;
 }
 decrement() {
  this.count--;
 }
}
```

```
import { ChildComponent } from './child.component';
@ViewChild(ChildComponent) child: ChildComponent;

 increment() {
  this.child.increment();
 }

 decrement() {
  this.child.decrement();
 }
```

Component Interaction

CREDO SYSTEMZ
simplifying IT

@Output & EventEmitter

Child component

- Declare a property of type EventEmitter and instantiate it
- Mark it with a @Output Decorator
- Raise the event passing it with the desired data

```
export class ChildComponent {
   @Input() count: number;

   @Output() countChanged: EventEmitter<number> =   new
EventEmitter();

   increment() {
      this.count++;
      this.countChanged.emit(this.count);
    }
   decrement() {
      this.count--;
      this.countChanged.emit(this.count);
   }
}
```

Parent Component

- Bind to the Child Component using Event Binding and listen to the child events
- Define the event handler function

```
<child-component [count]=ClickCounter
(countChanged)="countChangedHandler($event)"></child-
component>`


countChangedHandler(count: number) {
  this.ClickCounter = count;
  console.log(count);
}
```

Template Driven Forms

CREDO SYSTEMZ
simplifying IT

- 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>**

Template Driven Forms

CREDO SYSTEMZ
simplifying IT

- 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.

- Add the ngModel directive
- Add the name attribute.
- **<input name="foo" ngModel>**

Template Driven Forms

CREDO SYSTEMZ
simplifying IT

```
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">
 <p>
  <label for="firstname">First Name</label>
  <input type="text" id="firstname" name="firstname">
 </p>
<p>
  <label for="email">Email </label>
  <input type="text" id="email" name="email">
 </p>
 <p>
 <label for="country">country </label>
 <select name="country" id="country">
  <option selected="" value=""></option>
  <option [ngValue]="c.id" *ngFor="let c of countryList">
   {{c.name}}
  </option>
 </select>
 </p>
 <p>
  <button type="submit">Submit</button>
 </p>
</form>
```

```
countryList:country[] = [
  new country("1", "India"),
  new country('2', 'USA'),
  new country('3', 'England')
 ];

onSubmit(contactForm) {
  console.log(contactForm.value);
 }
```

```
<pre>Value : {{contactForm.value | json }} </pre>
<pre>Valid : {{contactForm.valid}} </pre>
<pre>Touched : {{contactForm.touched  }} </pre>
<pre>Submitted : {{contactForm.submitted  }} </pre>

<input type="text" name="firstname" #fname="ngModel"
ngModel>
<pre>Value    : {{fname.value}} </pre>
<pre>valid    : {{fname.valid}} </pre>
<pre>invalid  : {{fname.invalid}} </pre>
<pre>touched  : {{fname.touched}} </pre
```

Template Driven Forms - Validations

CREDO SYSTEMZ
simplifying IT

- Validations in Template-driven forms are provided by the Validation directives. The Angular Forms Module comes with several built-in validators

Required Validation:

```
<input type="text" id="firstname" name="firstname" required
[(ngModel)]="contact.firstname">
```

Minlength Validation

```
<input type="text" id="firstname" name="firstname" required minlength="10"
[(ngModel)]="contact.firstname">
```

Email Validation

```
<input type="text" id="email" name="email" required email [(ngModel)]="contact.email">
```

Disable Submit Button

```
<button type="submit" [disabled]="!contactForm.valid">Submit</button>
```

Reactive Forms

CREDO SYSTEMZ
simplifying IT

Reactive forms are forms where we define the structure of the form in the component class.

- Import ReactiveFormsModule
- Create Form Model in component class using Form Group, Form Control & Form Arrays
- Create the HTML Form resembling the Form Model.
- Bind the HTML Form to the Form Model

```
import { ReactiveFormsModule } from '@angular/forms';

imports: [
   BrowserModule,
   AppRoutingModule,
   ReactiveFormsModule
 ],
```

Reactive Forms

CREDO SYSTEMZ
simplifying IT

- Template-driven approach, we used ngModel & ngModelGroup directive on the HTML elements. The FormsModule created the FormGroup & FormControl instances from the template.
- In Reactive Forms approach, It is our responsibility to build the Model using **FormGroup, FormControl and FormArray**
- **FormControl** encapsulates the state of a single form element in our form. It stores the value and state of the form element
- **FormGroup** represents a collection of form Controls. It can also contain form groups and form arrays
- we need to import FormGroup, FormControl & Validator from the **@angular/forms**

```
import { FormGroup, FormControl, Validators } from '@angular/forms'
```

Reactive Forms

CREDO SYSTEMZ
simplifying IT

- **FormGroup**

```
contactForm = new FormGroup({})
```

- **FormControl**

```
firstname: new FormControl()
```

```
contactForm = new FormGroup({
  firstname: new FormControl(),
  lastname: new FormControl(),
  email: new FormControl(),
  gender: new FormControl(),
  country: new FormControl()
})
```

- we have created an instance of a FormGroup (contactForm). contactForm is our top-level FormGroup.
- Under the contactForm, we have five FormControl instances each representing the properties
- firstname, lastname, email, gender & country

Reactive Forms – Bind to HTML

CREDO SYSTEMZ
simplifying IT

- we need to associate our model to the Template

```
<form [formGroup]="contactForm">

<input type="text" id="firstname" name="firstname" formControlName="firstname">
<input type="text" id="lastname" name="lastname" formControlName="lastname">
```

- Submit Form

```
<form [formGroup]="contactForm" (ngSubmit)="onSubmit()">
```

Reactive Forms – Validations

CREDO SYSTEMZ
simplifying IT

- **Disabling the Browser validation**

```
<form [formGroup]="contactForm" (ngSubmit)="onSubmit()" novalidate>
```

- **Adding in Built-in Validators**

Required Validator:
firstname: new FormControl('',[Validators.required]),

Minlength Validator:
firstname: new FormControl('',[Validators.required,Validators.minLength(10)]),

Maxlength Validator:
lastname: new FormControl('',[Validators.maxLength(15)]),

Pattern Validator:
lastname: new FormControl('',[Validators.maxLength(15), Validators.pattern("^[a-zA-Z]+$")]),

Email Validator:
email:new FormControl('',[Validators.email,Validators.required]),

Reactive Forms – Validations

CREDO SYSTEMZ
simplifying IT

```
 contactForm = new FormGroup({
    firstname: new FormControl('',[Validators.required,Validators.minLength(10)]),
    lastname: new FormControl('',[Validators.required, Validators.maxLength(15), Validators.pattern("^[a-zA-Z]+$")]),
    email:new FormControl('',[Validators.email,Validators.required]),
    gender: new FormControl('',[Validators.required]),
country: new FormControl('',[Validators.required]),
})


<button type="submit" [disabled]="!contactForm.valid">Submit</button>
```

```
<div
    *ngIf="!contactForm.controls.firstname?.valid && (contactForm.controls.firstname?.dirty
    ||contactForm.controls.firstname?.touched)">
      First Name is not valid
  </div>
```

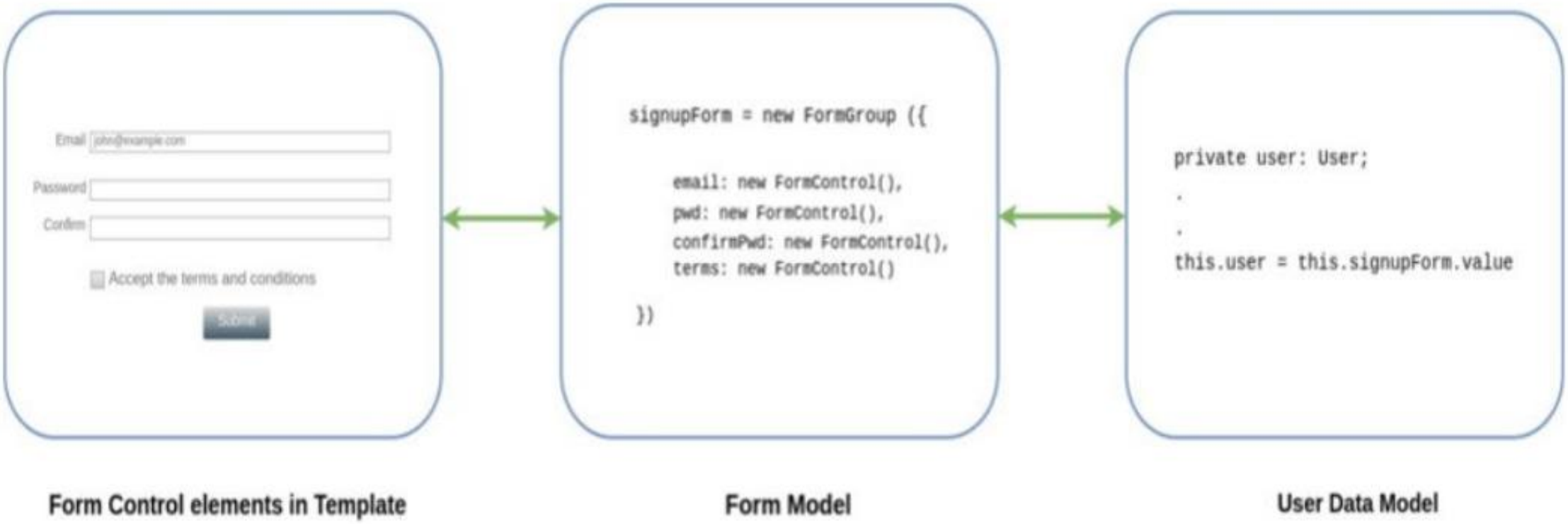
Reactive Forms – SetValue & PatchValue

- The setValue method sets a new value to the form controls of this FormGroup
- The patchValue patches the new value of this FormGroup in best possible way
- patchValue is that the object passed to setValue should match the structure of FormGroup object

```
this. contactForm.setValue({
  firstname: 'Mahesh',
  lastname: 'Kumar',
  email: 'Mahesh.kumar@gmail.com',
  gender: 'M',
  country: 'IND'
});
```

```
this. contactForm. patchValue({
  lastname: 'Kumar Raju',
  email: 'Mahesh.kumar12@gmail.com',
});
```

Reactive Forms – Model Integration



Route Guard in Angular

CREDO SYSTEMZ
simplifying IT

We use the Angular Guards to control, whether the user can navigate to or away from the current route. One of the common scenario, where we use Route guards is authentication. We want our App to stop the unauthorized user from accessing the protected route.

Types of Route Guards:

- **CanActivate** - This guard decides if a route can be activated
- **CanDeactivate** -This Guard decides if the user can leave the component (navigate away from the current route)
- **Resolve** - This guard delays the activation of the route until some tasks are complete
- **CanLoad** - The CanLoad Guard prevents the loading of the Lazy Loaded Module
- **CanActivateChild** - This guard determines whether a child route can be activated.

Create Route Guard in Angular

CREDO SYSTEMZ
simplifying IT

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

```
ng g guard auth
```

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

```
import { CanActivate } from '@angular/router';
```

3. **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()**

```
canActivate(): boolean {
  return true;
}
```

Return value from the Guard
The guard method must return either a True or a False value.
If it returns true, the navigation process continues. if it returns false, the navigation process stops.

Route Guard continue..

CREDO SYSTEMZ
simplifying IT

4. **Register the Guard as Service in Module**

We need to register them with the Providers array of the Angular Module as shown below

```
providers: [AuthGuard]
```

5. **Update the Routes**

we need to add the guards to the routes array as shown below

```
{ path: 'products', component: ProductsComponent, canActivate:[AuthGuard] },
```

```
canActivate() {
 console.log("OnlyLoggedInUsers");
   if (this.user==='Admin') { (
     return true;
   } else {
     window.alert("You don't have permission to view this page");
     return false;
   }
}
```

localStorage and sessionStorage

The localStorage and sessionStorage objects, part of the web storage API, are two great tools for saving key/value pairs locally.

- The data is saved locally only and can't be read by the server, which eliminates the security issue that cookies present.
- It allows for much more data to be saved (10mb for most browsers).
- The syntax is straightforward.
- It's also supported in all modern browsers

localStorage vs sessionStorage:

- **sessionStorage** - the data is persisted only until the window or tab is closed
- **localStorage** - the data is persisted until the user manually clears the browser cache or until your web app clears the data

Creating, Reading, and Updating Storage

CREDO SYSTEMZ
simplifying IT

- Create entries for the localStorage object by using the setItem() method

```
localStorage.setItem(key, 'Value');
```

- Read entries, use the getItem() method

```
let myItem = localStorage.getItem(key);
```

- Updating an entry is done with the setItem()

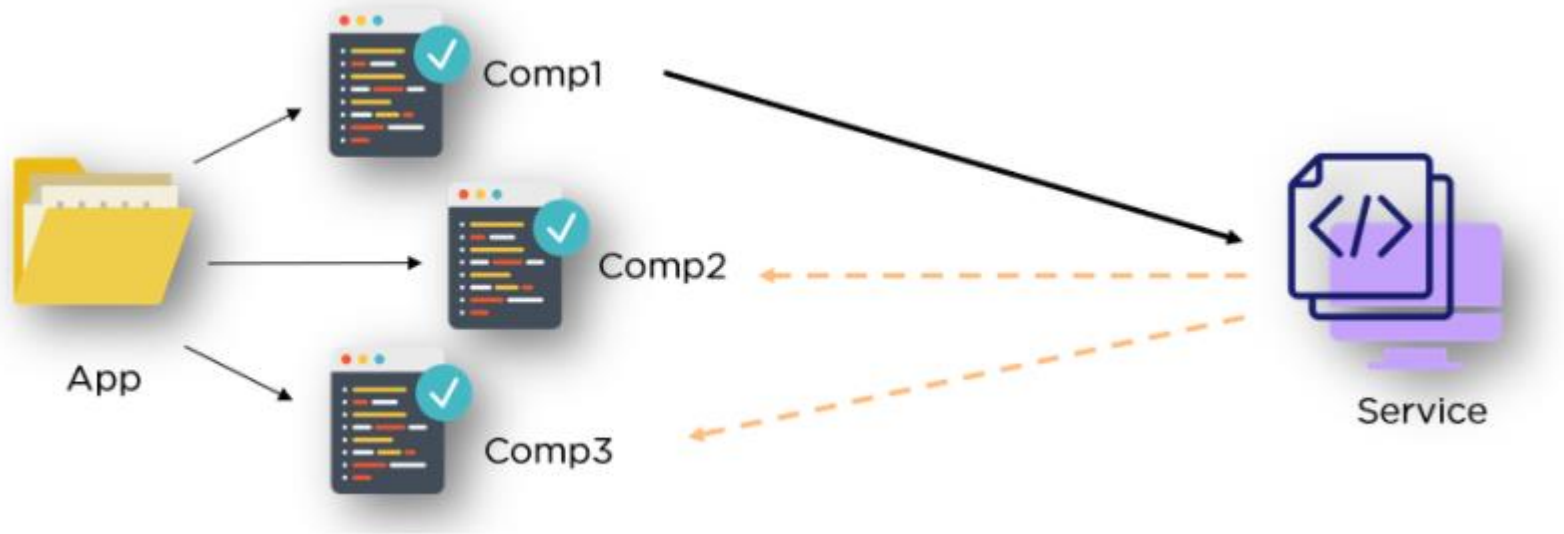
```
localStorage.setItem(key, 'New Value');
```

- Deleting and Clearing Entries

```
localStorage.removeItem(key);
localStorage.clear();
```

Services

- The main objective of the service is to organize and share business logic with different components of an Angular application.
- 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.



Features of Services

A Service is a Class

Decorated with @Injectibe

They share the same piece of code

Hold the business logic

Interact with the backend

Share data among components

Services are singleton

Registered on modules or components

Why Services

CREDO SYSTEMZ
simplifying IT

Why Should We Use Services in Angular?

- Features that are independent of components such a logging services
- Share logic or data across components
- Encapsulate external interactions like data access

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.

Create Service:

```
ng generate service Products
```

Registration of Angular Service

```
providers: [ProductsService],
```

Injecting Services into Component

CREDO SYSTEMZ
simplifying IT

Instantiated the productService in the Component directly as shown below

```
prdSrc:any;
this.prdSrc = new ProdsService();
```

Directly instantiating the service, as shown above, has many **disadvantageous**

- The ProductService is **tightly coupled to the Component**. If we change the ProductService class definition, then we need to update every code where service is used
- If we want to change ProductService with BetterProductService, then we need to search wherever **the ProductService is used and manually change it**
- Makes Testing difficult. We may need to **provide mockProductService for testing and use the ProductService** for Production

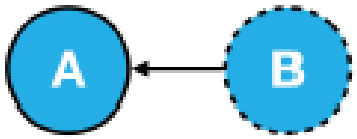
Services continue..

CREDO SYSTEMZ
simplifying IT

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.
- The Angular dependency injection is now the core part of the Angular. It allows us to inject dependencies into the Component, Directives, Pipes, or Services.
- Dependency Injection (DI) is a technique in which a class receives its dependencies from external sources rather than creating them itself.

Services continue..

If we used ng generate or VSCode “Generate Service”, it will add the following code

```
@Injectable({ providedIn: 'root' })
```

- The providedIn allow us to specify how Angular should provide the dependency in the service class itself instead of in the Angular Module.
- Use the ProvidedIn root option, when you want to register the application-level singleton service.
- The root option registers the service in the Root Module Injector of the Module Injector tree. This will make it available to the entire application.

Services continue..

CREDO SYSTEMZ
simplifying IT

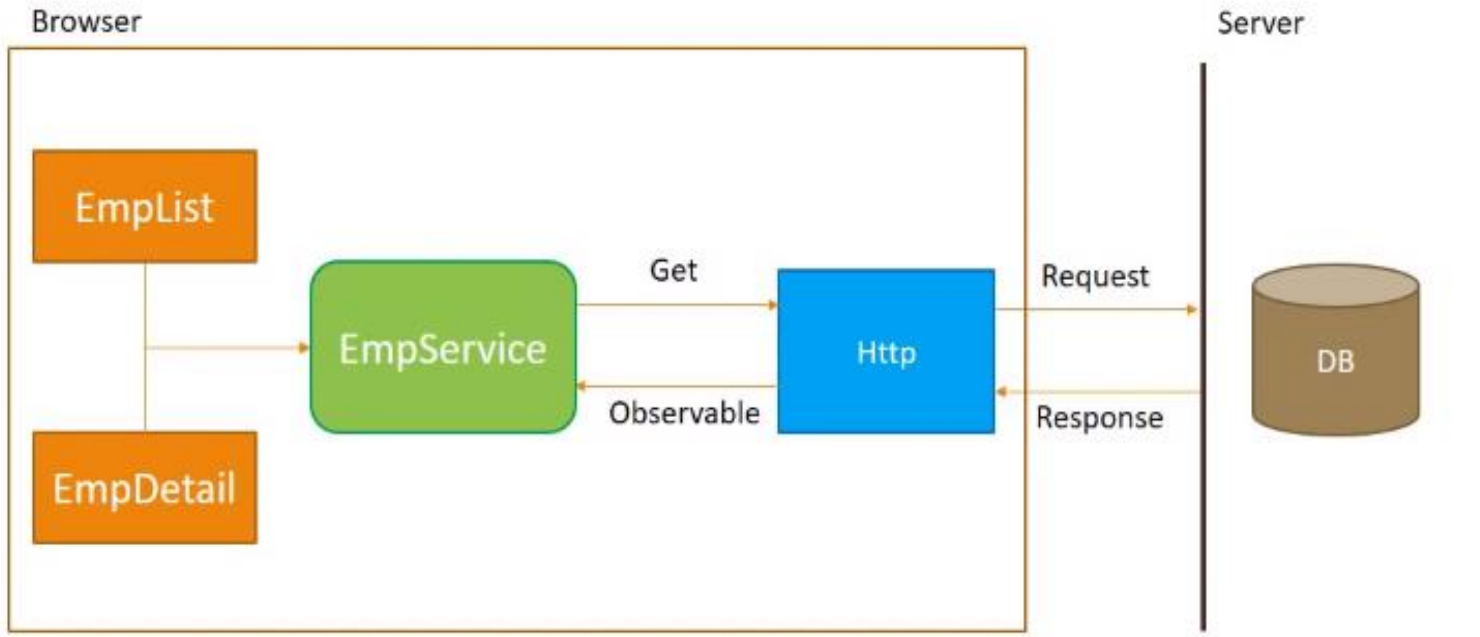
Injecting service into the component

```
constructor(private productService:ProductService) {
  }

getProducts() {
     this.products=this.productService.getProducts();
  }
```

- Our Component **does not create the instance** of the ProductService. It just asks for it in its Constructor.
- The responsibility of Creating the ProductService falls on the **creator of the Component class**
- Our Component is now **loosely coupled** to the ProductService

CREDO SYSTEMZ
simplifying IT

Http Mechanism



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.

Http Implementation

CREDO SYSTEMZ
simplifying IT

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';
```

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

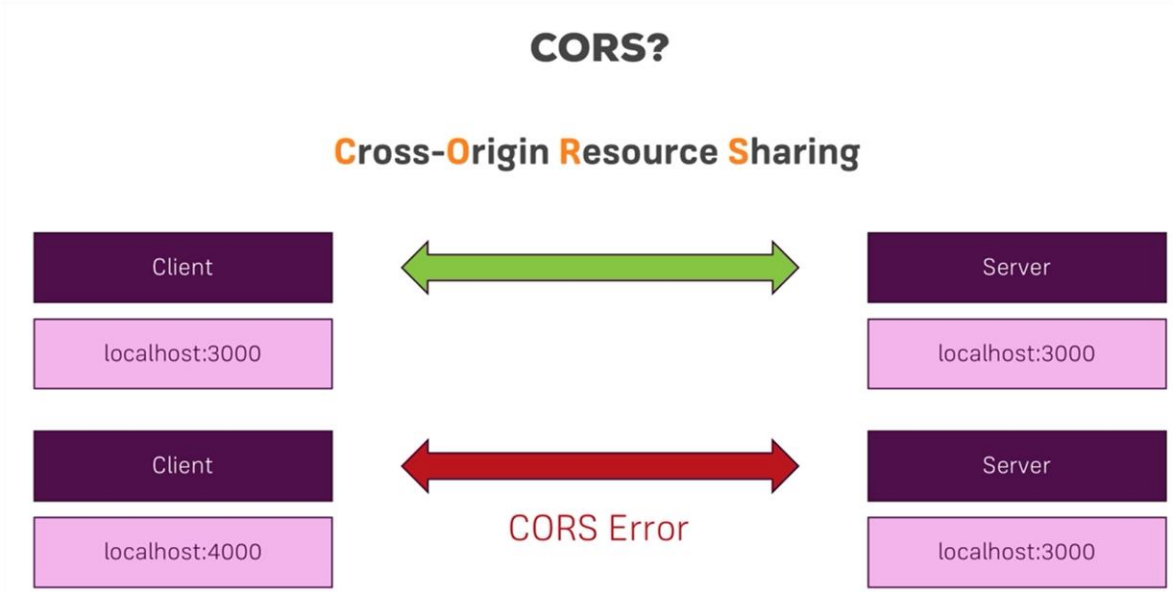
```
imports: [
  BrowserModule,
  HttpClientModule
],
```

Http Implementation

CREDO SYSTEMZ
simplifying IT

3. Create a service to handle the HTTP Request

```
baseURL: string = "http://api.icndb.com/";
 geProds(userName: string): Observable<any> {
   return this.http.get(this.baseURL + 'jokes/random/')
 }
```



Http Implementation

CREDO SYSTEMZ
simplifying IT

4. Consuming service in Components

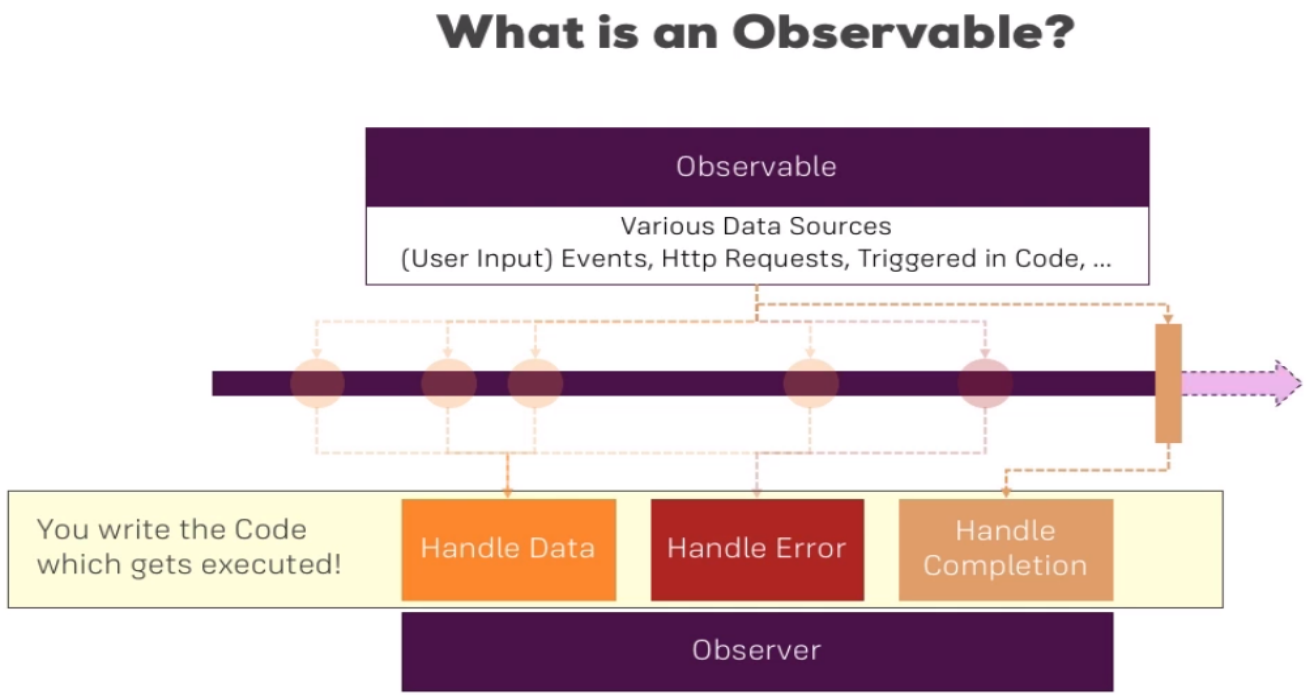
```
import { HttpClient } from '@angular/common/http‘

constructor(private http: HttpClient) {  }

public getProducts() {
 this.ProdService.getProds()
     .subscribe(
      (response) => {                    //next() callback
        console.log('response received')
        this.prods = response;
      },
      (error) => {                       //error() callback
        console.error('Request failed with error')
      },
      () => {                            //complete() callback
        console.error('Request completed')      //This is actually not needed
      })
}
```

Observable

- To perform asynchronous programming in Angular application, we can use Observable.

- When we send and receive data over HTTP, we need to deal it asynchronously because fetching data over HTTP may take time.

- The Observable represents any set of values over any amount of time.



Observable is a wrapper around data stream

Observable

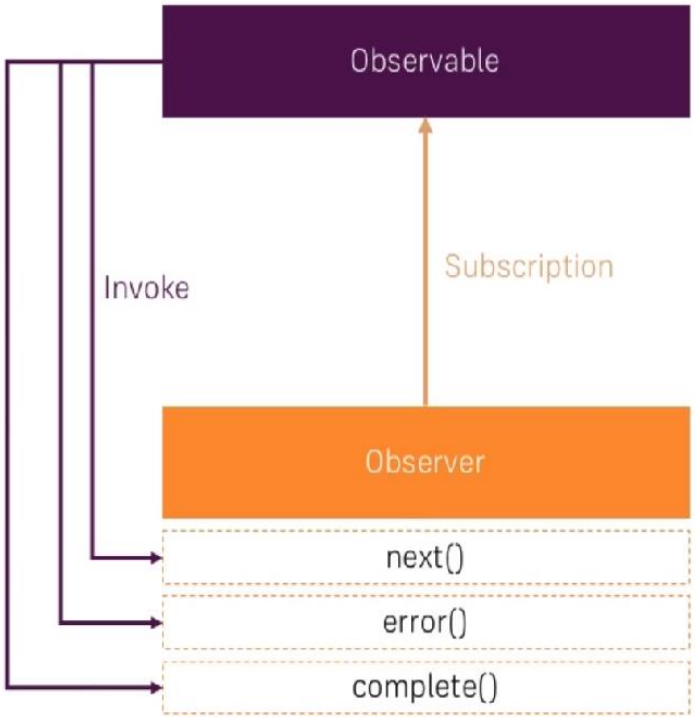
CREDO SYSTEMZ
simplifying IT

When we subscribe to any observable, we optionally pass the three callbacks.
next(), error() & complete()

1. Next() - callback is where we get the result of the observable.
In above example the list of products as a result.

2. error() - The observable can also result in an error. It will invoke the error() callback and pass the error object

3. complete() - When the observable completes, it will call the complete() callback



Http Get

CREDO SYSTEMZ
simplifying IT

- The HttpClient.get sends the HTTP Get Request to the API endpoint and parses the returned result
- Default body of the response is parsed as JSON.
- We can also specify other type explicitly using the observe & responseType options

```
get(url: string,
    options: {
        headers?: HttpHeaders | { [header: string]: string | string[]; };
        params?: HttpParams | { [param: string]: string | string[]; };
        observe?: "body|events|response|";
        responseType: "arraybuffer|json|blob|text";
        reportProgress?: boolean;
        withCredentials?: boolean;}
    ): Observable<>
```

Http Get - options

- **headers -** It allows you to add HTTP headers to the outgoing requests
- **params** - Allows us to Add the URL parameters or Get Parameters to the Get Request
- **observe** - The HttpClient.get method returns the body of the response parsed as JSON. The allowed options are *response, body & events*
- **reportProgress** - set this to true, if you want to get notified of the progress of the Get Request
- **responseType** - Json is the default response type. In case of different type of response, then we can use this parameter. The Allowed Options are *arraybuffer, blob, JSON, and text*.
- **withCredentials** - It is of boolean type. If the value is true then HttpClient.get will request data with credentials (cookies)

Http Post

- The HttpClient.post() sends the HTTP POST request to the endpoint
- Similar to the get(), we need to subscribe to the post() method to send the request
- Post method parsed the body of the response as JSON, we can also specify other type explicitly using the observe & responseType options

```
post(url: string,
    body: any,
    options: {
      headers?: HttpHeaders | { [header: string]: string | string[]; };
      observe?: "body|events|response|";
      params?: HttpParams | { [param: string]: string | string[]; };
      reportProgress?: boolean;
      responseType: "arraybuffer|json|blob|text";
      withCredentials?: boolean;
    }
): Observable
```

Http Post - Example

- In addPerson method, we send an HTTP POST request to insert a new person in the backend
- we need to set the 'content-type': 'application/json' in the HTTP header for sending data as JSON
- The JSON.stringify(person) converts the person object into a JSON string
- Finally, we use the http.post() method using URL, body & headers as shown below

```
addPerson(person:Person): Observable<any> {
  const headers = { 'content-type': 'application/json'}
  const body=JSON.stringify(person);
  console.log(body)
  return this.http.post(this.baseURL + 'people', body,{'headers':headers})
}
```

```
addPerson() {
  this.apiService.addPerson(this.person)
    .subscribe(data => {
      console.log(data)
      this.refreshPeople();
    })
}
```

```
export class Person {
 id:number
 name:string
}
```

Service

Component

Model

Http Headers & Params

CREDO SYSTEMZ
simplifying IT

HttpHeaders(): We add HTTP Headers using the HttpHeaders helper class to use HttpHeaders in your app, you must import it into your component or service

```
import { HttpHeaders } from '@angular/common/http';

const headers= new HttpHeaders()
  .set('content-type', 'application/json')
  .set('Access-Control-Allow-Origin', '*');

this.httpClient.get(this.baseURL + 'users/' + userName + '/repos', { 'headers': headers })
```

HttpParams(): We add the URL parameters using the helper class HttpParams. The HttpParams is passed as one of the arguments to HttpClient.get method.

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

this.httpClient.get(this.baseURL + 'users/' + userName + '/repos',{params})
```

Sample API's

CREDO SYSTEMZ
simplifying IT

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

Movie Api:
http://www.omdbapi.com/?t=theri&apikey=c429066e

Jokes Api:
http://api.icndb.com/jokes/random

Users:
https://jsonplaceholder.typicode.com/users

Interceptor

CREDO SYSTEMZ
simplifying IT

- The Interceptor helps us to modify the HTTP Request by intercepting it before the Request is sent to the back end
- Interceptor can be useful for adding custom headers to the outgoing request, logging the incoming response, etc
- The Angular HTTP interceptors sit between our application and the backend

1. Implement the Interceptor, you need to create an injectable service, which implements the **HttpInterceptor interface**

```
@Injectable() export class AppHttpInterceptor implements HttpInterceptor {
}
```

2. class must implement the method **Intercept**

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
   //do whatever you want with the HttpRequest
   return next.handle(req);
}
```

3. This class is then provided in the Root Module using the **HTTP_INTERCEPTORS** token

```
providers: [ {
      provide: HTTP_INTERCEPTORS,
      useClass: AppHttpInterceptor,
      multi: true
   } ],
```

Interceptor - HttpInterceptor Interface

CREDO SYSTEMZ
simplifying IT

- The interface contains a single method Intercept with the following signature

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>
```

HttpRequest - first argument

- The HttpRequest is an outgoing HTTP request which is being intercepted
- It contains URL, method, headers, body, and other request configuration

HttpHandler - second argument

- The HttpHandler dispatches the HttpRequest to the next Handler using the method HttpHandler.handle
- The next handler could be another Interceptor in the chain or the Http Backend

Interceptor - Examples

```
import {Injectable} from "@angular/core";
import {HttpEvent, HttpHandler, HttpInterceptor,HttpRequest} from "@angular/common/http";
import {Observable} from "rxjs/Observable";

@Injectable()
export class AppHttpInterceptor implements HttpInterceptor {
   constructor() {
   }

   intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
      console.log(req);
      return next.handle(req);
   }
}
```

Import these modules

Class which implements **HttpInterceptor** Interface

Intercept method with HttpRequest & HttpHandler

Interceptor - Setting the new headers

CREDO SYSTEMZ
simplifying IT

```
import {Injectable} from "@angular/core";
import {HttpEvent, HttpHandler, HttpInterceptor,HttpRequest} from "@angular/common/http";
import {Observable} from "rxjs/Observable";

@Injectable()
export class AppHttpInterceptor implements HttpInterceptor {
   constructor() {
   }

   intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
      console.log(req);
      req = req.clone({ headers: req.headers.append('Content-Type', 'application/json')});
      req = req.clone({ headers: request.headers.set('Authorization', "Bearer " + this.token)});
      console.log(req);
      return next.handle(req);
   }
}
```

Angular lifecycle hooks

- A component in Angular has a life-cycle, a number of different phases it goes through from birth to death
- Once Angular loads the components, it needs to check the input properties, evaluate the data bindings & expressions, render the projected content etc to start the process of rendering the view
- Angular also removes the component from the DOM, when it is no longer needs it.
- The Angular life cycle hooks are nothing but callback function, which angular invokes when a certain event occurs during the component's life cycle

Angular lifecycle hooks

This is list of life cycle hooks, which angular invokes during the component life cycle, Angular invokes them when a certain event occurs

- ngOnChanges
- ngOnInit
- ngDoCheck
- ngAfterContentInit
- ngAfterContentChecked
- ngAfterViewInit
- ngAfterViewChecked
- ngOnDestroy

Life Cycle of a component begins, when Angular creates the component class. First method that gets invoked is class Constructor.

Angular lifecycle hooks

CREDO SYSTEMZ
simplifying IT

ngOnChanges –

- The Angular invokes ngOnChanges life cycle hook whenever **any data-bound input property of the component or directive changes**.
- Input properties are those properties, through which parent communicates with the child component
- The change detector checks if such input properties of a component are changed by the parent component. If it is then it raises the ngOnChanges hook

```
ngOnChanges() {
   console.log("AppComponent:ngOnChanges");
 }
```

Angular lifecycle hooks

CREDO SYSTEMZ
simplifying IT

ngOnInit–

- The Angular raises the ngOnInit hook, after it **creates the component and updates its input properties**. It raises it after the ngOnChanges hook
- This hook is **fired only once** and immediately after its creation
- We can add **any initialisation logic** for our component in this method

```
ngOnInit() {
  console.log("AppComponent:OnInit");
 }
```

Angular lifecycle hooks

CREDO SYSTEMZ
simplifying IT

ngDoCheck – The Angular invokes the ngDoCheck hook event during every change detection cycle. This hook is invoked even if there is no change in any of the properties.

ngAfterContentInit– ngAfterContentInit Life cycle hook is called after the Component's projected content has been fully initialized

ngAfterContentChecked– ngAfterContentChecked Life cycle hook is called during every change detection cycle after Angular finishes checking of component's projected content

ngAfterViewInit– ngAfterViewInit hook is called after the Component's View & all its child views are fully initialized

ngAfterViewChecked– The Angular fires this hook after it checks & updates the component's views and child views

ngOnDestroy– This hook is called just before the Component/Directive instance is destroyed by Angular. You can Perform any cleanup logic for the Component here.