Angular

Mentor as a Service

Angular is a JavaScript framework for building both client and server-side apps.

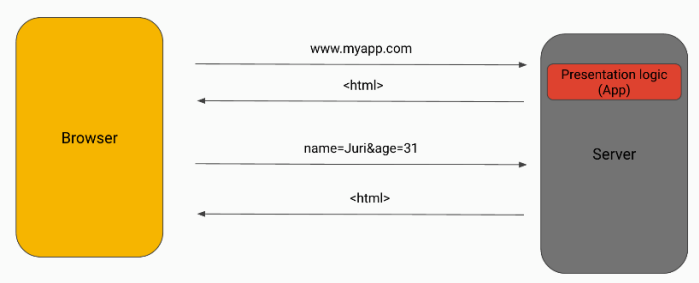
Angular, a front-end framework for building robust, cross-platform, client-side applications.

Angular is component centric framework that is a complete rewrite of AngularJS. It uses Typescript as a default language to create apps in. It also utilizes something called ES2015 imports to refer to external files or modules as they are also called.

The JavaScript and web world has changed a lot in few years. Web Applications could be server side rendered or client side rendered.

Server-side rendered

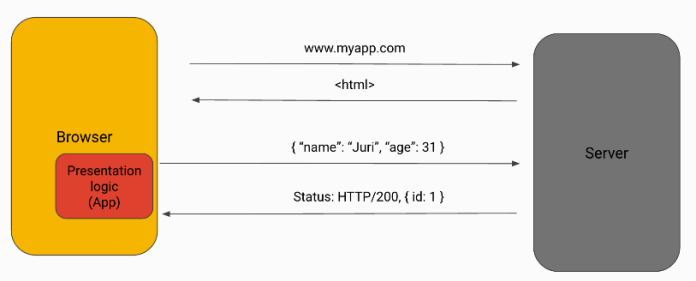
In server-side rendered web applications, most of the application's logic resides on the server, and remains there. The user basically enters the URL, the request gets send to the server, which then produces the final HTML containing the requested data and sends that back to the browser which simply renders it out. When the user interacts with the page, that request gets again sent to the server, which in turn generates a new HTML page and serves it back to the browser.



Client-side rendered

Modern web pages often require to work more like applications do on the desktop, though. People demand for a much better user experience, more interactivity, fast transitions between "pages" and even offline capabilities. That's where the so-called SPAs (Single Page Applications) come into play.

When the user enters the URL, the web server responds with an HTML page, but also with a set of resources (JavaScript files and images) that make up our client-side application. The browser receives that, loads the JavaScript application and "boots it". Now it's the job of that application to dynamically generate the user interface (HTML) based on the data, right from within the browser. After that happens, every new user action doesn't reload the entire web site again, but rather the data for that specific user interaction is send to the server (usually by using the JSON format) and the server in turn responds with the just the amount of data requested by the JavaScript client, again using JSON (normally). The JavaScript application gets the data, parses it and dynamically generates HTML code which is shown to the user.



As you can see, the amount of data that is being exchanged is very optimized. However, a big downside of such type of applications is that the startup time is usually much longer. You might already have figured why: well, because the browser doesn't get the HTML code to show, but rather a bunch of JavaScript files that need to be interpreted, and executed, and which in turn then generates the final HTML to be shown to the user.

Server-side pre-rendering, isomorphic JavaScript or universal JavaScript

When the user enters the URL, the server loads the JavaScript application on the server side, boots it up and then delivers the already (by the JavaScript app) pre-rendered HTML plus the JavaScript app itself back to the client. There, the browser interprets the JS app and runs it, which has then to be intelligent enough to resume where the server has left off.

Angular library gets really, small.

By being able to strip out useless parts with the template/offline compiler, lots of stuff can already be dropped when deploying in production. Furthermore, the goal is to use a bundler that supports tree shaking and thus reduce the size of the final compiled JS files even more by eliminating everything that is not actively being used within your application. Frankly, if you don't use the routing module of angular, it simply won't get included in your final app.

Tree shaking

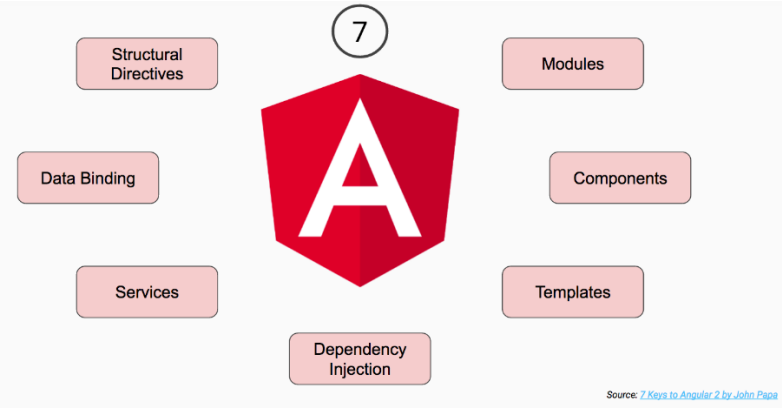
Tree shaking s basically "dead code elimination". By analyzing which JavaScript modules are used and which aren't, compilers that support such approach can eliminate the unused parts and thus produce a much more optimized and smaller bundle. Obviously, a proper module system such as ES2015 modules has to be used for this to work.

Lazy loading, built-in

Finally! Lazy loading has been a hot topic already for Angular 1.x and many other frameworks. When you build a serious application, it might get quite big quick. That said, you cannot force your users to download megabytes over megabytes just to get your app boot up, especially on flaky Internet connections or mobiles. That's why lazy loading needs to be implemented. The idea is to load only those parts the users most heavily use and load other parts on demand when needed. This was particularly hard in Angular 1. Lazy Load being one possibility to achieve this.

Now with Angular 2 this is finally built-in right from the beginning, through the framework's router and so-called "lazy routes".

The 7 Key Concepts behind Angular



Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is itself written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

Modules

Angular framework is built with the help of modules. **Angular modules** are **logical collections of Component, directives, pipes, services, decorators, classes, interfaces, constants, etc**.

Every Angular app has a root module, conventionally named **AppModule**, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

The basic building blocks of an Angular application are **NgModules**, which provide a compilation context for components. **NgModules** collect related code into functional sets; an Angular app is defined by a set of **NgModules**. An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.

Like JavaScript modules, **NgModules** can import functionality from other **NgModules**, and allow their own functionality to be exported and used by other **NgModules**. For example, to use the router service in your app, you import the Router **NgModule**.

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of lazy-loading—that is, loading modules on demand—to minimize the amount of code that needs to be loaded at startup.

Components

Components define views, which are sets of screen elements that Angular can choose among and modify according to your program logic and data. Every app has at least a root component.

Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.

Both components and services are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them.

Templates, directives, and data binding

A template combines HTML with Angular markup that can modify the HTML elements before they are displayed. Template directives provide program logic, and binding markup connects your application data and the document object model (DOM).

Event binding lets your app respond to user input in the target environment by updating your application data.

Property binding lets you interpolate values that are computed from your application data into the HTML.

Before a view is displayed, Angular evaluates the directives and resolves the binding syntax in the template to modify the HTML elements and the DOM, according to your program data and logic. Angular supports two-way data binding, meaning that changes in the DOM, such as user choices, can also be reflected into your program data.

Services and Dependency Injection

Besides components, Angular always had the concept of Services and Dependency Injection. So does Angular 2. While the component is meant to deal with the UI and related stuff, the service is the place where you put your "business logic" s.t. it can be shared and consumed by multiple components. A service is nothing else than a simple ES6 class:

```
@Injectable()
export class PersonService {
      fetchAllPeople() { ... }
}
```

Router for Navigation (SPA – Single Page Application)

An app's components typically define many views, arranged hierarchically. Angular provides the Router service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

Component based architecture

Component based architectures is the new paradigm for frontend development. The idea is to build **autonomous pieces** with clearly defined responsibilities and which might even be reusable across multiple applications. **Components are 1st class citizens in Angular.**

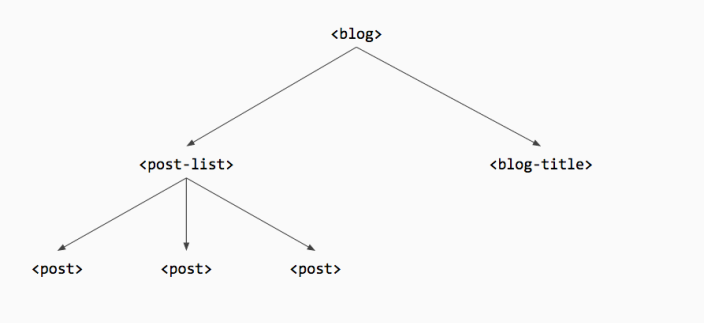
A simple component in Angular 2 looks like as shown below:

```
@Component({selector: 'hello-world',
            template: `<p>Hello, world!</p>`})
class HelloWorldComponent { }
```

In the corresponding HTML you would write this to instantiate it.

```
<hello-world></hello-world>
```

Every Angular application consists of such a component tree, having a top-level "application component" or root component and from there, lots of child and sibling components.



And this could then be mapped to HTML code like as follows.

The component tree is of major importance in Angular 2 and you will always again come across it.

Templates and Data Binding

Obviously, when we write something like **`<hello-world></hello-world>`**, we also need to define somewhere what Angular should render in place. That's where *templates* and *data binding* come into play. As we've seen before, we define the template directly in the **`@Component({})`** annotation by either using the **`template`** or **`templateUrl`** property, depending on whether we want to define it inline or load it from some url.

```
@Component({
      ...
      template: `
      <p>Hello, world!</p>
      `
})
class HelloWorldComponent {}
```

We also need some data binding mechanism to get data into this template and out of it again. Let's look at an example:

```
@Component({
      selector: 'hello-world',
      template: `<p>Hello, {{ who }}</p>`
})
class HelloWorldComponent {   who: string = 'Juri'  }
```

As you can see, the variable who inside the component's class gets bound to into the template. Whenever you change the value of who, the template will automatically reflect that change.

Angular is a JavaScript framework for building both client and server-side apps. While it competes directly with libraries like React and Vue.js, it's most recent release (Angular) makes it preferable for enterprise development. In this article, we'll discuss why Angular is better suited for enterprise environments over React and other front-end libraries.

Frameworks over other JavaScript Libraries

React and **Vue** are libraries. Angular is a framework. The difference? A framework is a highly opinionated way of doing things. It enforces a specific implementation (like MVC) and provides supporting modules out of the box. A library is simply a collection of methods and functions that you can call from within your own implementation.

While libraries provide more freedom and flexibility with architecture and design, they rely on developers to choose their own third-party libraries to make things work. A good example is with routing. While Angular includes its own routing module out of the box, React relies on the developer to choose how they handle routing themselves. This can make it more difficult to onboard new team members who aren't familiar with your specific implementation of React.

By using Angular, larger teams can more easily conform to the same architectural approach. New team members can more quickly jump into existing projects as things are more standardized in the Angular world.

TypeScript

Angular is written in TypeScript, a statically typed superset of JavaScript. TypeScript brings static type checking to front-end development and closely resembles languages like Java, .Net, Scala, etc. This makes it easier for enterprise engineers familiar with Java/.Net to jump into Angular projects. Not to mention the benefits of code readability and maintenance associated with statically typed languages. TypeScript makes it easier for large scale development teams to collaborate.

Testing

Angular projects come preconfigured with testing libraries like Karma and Protractor for unit and functional testing. Using the Angular CLI makes creating components and their corresponding test files a breeze. The Angular framework also leverages Node to quickly implement testing environments for your app. Using preconfigured commands, you can easily run test suites and start dev servers with minimal configuration.

Server-Side Rendering

Angular Universal allows Angular projects to be rendered both client and server side. This adds to the flexibility of the framework in terms of the different types of apps you can build using Angular. Angular also supports native and mobile app development with the Ionic framework.

Long Term Support

With the release of Angular, Angular officially announced long term support for the Angular framework. This means development teams don't have to worry about constantly upgrading or approving software updates to keep using Angular.

Open Source

Unlike React, Angular is completely open source. React has a patent clause that can restrict companies from patenting their own products that use React. For these reasons, many enterprise teams stick with Angular to avoid sticky legal situations.

Angular vs. React

Since the release of Angular, the similarities between Angular and React have greatly increased. While React clearly beats Angular 1 from a performance standpoint, Angular 2 addresses virtually all the issues experienced with its predecessor. With the most recent release of Angular 4, the decisions to use React vs Angular has become largely a matter of personal preference.

Preface: Angular 1 is NOT Dead

Angular 1 is still widely used and even actively maintained by Google. There are still plenty of companies happily using Angular 1.x today. while Angular 1.x is far from dead, Angular is a complete rewrite and addresses most of the major pitfalls experienced with Angular 1.

Angular vs React: Which is Better?

Like all other comparisons online, the answer is "it depends". In fact, the notion of one being "better" or "worse" is an inaccurate assessment. Both have comparable strengths and weaknesses. Both can be used for the same reasons.

Which to use and why?

Choosing Angular vs React really boils down to what your needs are as a development team. When deciding whether to use React or Angular, ask yourself these questions.

Framework or library?

- React and Angular are fundamentally different. React is a UI component library whereas Angular is a full-fledged development framework.
- A library is simply a collection of functions that you can call within your own code. It's something you include within your own implementation. You include a library. A library does not include you!
- The benefits of working with a framework like Angular is that many of the things you want to use are already built in. For example, routing and Ajax HTTP requests are built into Angular. With React, you have to import additional libraries to do the same things.
- The main benefit of using a library like React is that you have a lot more freedom from a design standpoint. Since React is simply a library, you have the choice as to how you handle routing, testing, etc.

Learning curve?

- Both Angular and React have steep learning curves. Angular is written in TypeScript, a superset of JavaScript that brings type checking to JS.
- As a framework, Angular also relies heavily on dependency injection (DI). This can take some getting used to as Angular introduces a lot of magical boilerplate code that's confusing and hard to pick up. Things like annotations, providers, decorators can be hard to understand with Angular 2.
- React takes some getting used to as well. It uses JSX for rendering DOM templates. While JSX allows developers to construct templates directly in JavaScript code, it doesn't exactly match up with classic HTML. JSX syntax can take some getting used to even if your an experienced web developer.
- Additionally, React relies heavily on third party libraries to supplement it's core functionality. For example, you need an external library for AJAX calls (Axios) and routing. While this provides developers with more architectural design freedom, it also makes things harder to standardize with React. This can drastically increase learning curves and the ability for new developers to "jump in" to an existing React code base.
- Both React and Angular require preprocessing or **transpiling** of assets. While bundlers like Webpack make configuration easier, it's equally challenging to set up development environments for React and Angular for first timers. Unlike jQuery or other JavaScript libraries, you can't just throw in a CDN reference and start programming with Angular or React (this is not the case for Angular 1, so there's one thing the original still has going for it!).

How well is it supported?

- React is backed by Facebook. Angular by Google. Next topic.
- The competing strength of these ecosystems keeps both competitive. While Angular has TypeScript, React has Flow for type checking. While Angular has a great CLI, React has create-react-app. React has React Native. Angular has Ionic. Angular Universal allows for server-side rendering, but so does React's next.js.
- The point is that these competitors have strong communities backing them. Each one seems to answer any deficiencies with their own solutions. For these reasons, choosing React vs Angular is largely a personal preference.

Performance?

Ever since Angular abandoned the idea of bidirectional data flow and the digest cycle, the difference between Angular and React performance is negligible. Both React and Angular leverage unidirectional data flow and the virtual DOM to improve rendering. While React may have a slight edge in some circumstances, basing your decision around performance is silly with React and Angular.

Guide - Getting Started (Setup Development Environment)

1. Setup a Node development environment
2. Learn and install the Node Package Manager
3. Setup the Angular CLI
4. Scaffold our first Hello World Angular application
5. Setup an IDE of your choice Visual Code, WebStorm, etc.

The **Angular CLI** makes it easy to create an application that already works, right out of the box. It already follows our best practices!

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

Generating and serving an Angular Project via Development Server

```
ng new PROJECT-NAME
cd PROJECT-NAME
ng serve
```

Generating Components, Directives, Pipes and Services

You can use the ng generate (or just ng g) command to generate Angular components:

```
ng generate component dashboard-component
ng g component dashboard-component # using the alias

# components support relative path generation
# if in the directory src/app/feature/ and you run

ng g component Transflower-component

# your component will be generated in src/app/feature/Transflower-component
# but if you were to run

ng g component ./newer-cmp

# your component will be generated in src/app/newer-cmp
# if in the directory src/app you can also run

ng g component feature/new-cmp
# and your component will be generated in src/app/feature/new-cmp
```

Building Blocks of Angular Architecture

1. Components
2. Structural Directives
3. Attribute Directives

1. Components

- Component directive, is nothing but a simple class which is decorated with the @component decorator.
- In Angular, Normal typescript class will become a Component class once it has been decorated with **@component** decorator
- It is mainly used to specify the html templates.
- It is most commonly used directive in angular project
- If you need more info on angular component, then kindly refer here: need to add

```
import {Component} from 'angular/core'
@Component({
  selector: 'my-app',
  template: `<h3>Push Me</h3>
             <button (click)="btnClick()">Click Me</button>`
  })
export class App {
      btnClick() {
        alert('Hi Transflower, You clicked me!');
    }
}
```

NgStyle

NgStyle directive is like one of data binding technique called style binding in angular, but the main difference is, NgStyle used to set multiple inline styles for html element.

```
import { Component } from '@angular/core';
@Component({
    selector: 'my-app',
    template: ` <button style='color:blue'
                [ngStyle]="ApplyStyles()">Style Applied</button>`
})
export class AppComponent {
    isBold: boolean = true;  fontSize: number = 30;
    isItalic: boolean = true;

    ApplyStyles() {
        let styles = {
            'font-weight': this.isBold ? 'bold' : 'normal',
            'font-style': this.isItalic ? 'italic' : 'normal',
            'font-size.px': this.fontSize
        };
        return styles;
    }
}
```

NgClass

It is like NgStyle attribute but here it is using class attribute of the html element to apply the style.

```
import { Component } from '@angular/core';
@Component({
    selector: 'my-app',
    template: `<button class='colorClass'
                        [ngClass]='applyClasses()'>Transflower
                </button>`,
    styles: [`.boldClass{font-weight:bold; font-size : 30px;}
              .italicsClass{font-style:italic;}
              .colorClass{ color:grey;}`]
})
export class AppComponent {
    applyBoldClass: boolean = true;
    applyItalicsClass: boolean = true;

    applyClasses() {
        let classes = {
            boldClass: this.applyBoldClass,
            italicsClass: this.applyItalicsClass
        };
        return classes;
    }  }
```

2. Attribute Directive

- Examples of built-in attribute directives that ship with Angular are ngStyle and ngClass.
- These directives modify the DOM and we are going to do the same in a simple custom attribute directive of our own.

The naming convention for directives is: name.directive.ts

```
import {Directive, ElementRef, Renderer} from 'angular2/core';
@Directive ({
  selector: '[my-outline]',
  host: {
    '(mouseenter)': 'onMouseEnter()',
    '(mouseleave)': 'onMouseLeave()'
  }
})

export class MyOutline {
  constructor(private _element: ElementRef, private _renderer:Renderer) { }
  onMouseEnter() { this._outlineToggle(true); }
  onMouseLeave() { this._outlineToggle(false); }

  private _outlineToggle(color: string) {
    this._renderer.setElementStyle(this._element.nativeElement,
                                   'border','solid red 1px' );
  }
}
```

MyOutline class declares a constructor that injects the ElementRef service and creates a private _element property which gives us access to the DOM element. ElementRef has a nativeElement property that we could use to add our border directly like so:

```
el.nativeElement.style.border = 'solid red 1px';
```

It is a very simple directive that adds a border to an element when the user hovers over it.

3. Structural Directive

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.

- To say in simple words, unlike Attribute Directive which we see above, Structural directive is used to add or remove the Dom Element itself in the Dom Layout, whereas attribute directives are used to just change the attribute or appearance of the Dom element.
- Structural directives are easy to recognize by using an asterisk (*)

Built-in structural directive - NgIf, NgFor, and NgSwitch

- **NgIf** is used to create or remove a part of DOM tree depending on a condition

```
<div *ngIf=false> Say Hello</div>
```

- **NgFor** is used to customize data display. It is mainly used for display a list of items using repetitive loops

```
<table>
      <thead>
          <tr>
              <th>Code</th> <th>Name</th><th>Gender</th>
              <th>Annual Salary</th><th>Date of Birth</th>
          </tr>
      </thead>
      <tbody>
          <tr *ngFor='let employee of employees'>
            <td>{{employee.code}}</td> <td>{{employee.name}}</td>
            <td>{{employee.gender}}</td><td>{{employee.annualSalary}}</td>
            <td>{{employee.dateOfBirth}}</td>
          </tr>
          <tr *ngIf="!employees || employees.length==0">
              <td colspan="5">
                  No employees to display
              </td>
          </tr>
      </tbody>
  </table>
```

- **NgSwitch** is like the JavaScript switch It can display one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM.

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

@Component({
    selector: 'my-app',
    template: `<h2>{{title}}</h2>
               <p *ngIf="showElement">Show Element</p>
               <div [ngSwitch]="inpvalue">
               <p *ngSwitchCase="1">You selected Monday</p>
               <p *ngSwitchCase="2">You selected Tuesday</p>
               <p *ngSwitchCase="3">You selected Wednesday</p>
               <p *ngSwitchDefault>Sorry Invalid selection!!</p>
               </div>`
})
export class AppComponent {
    inpvalue: number = 3;
}
```

Custom Structural Directive

We are now going to create a simple custom variation of the **ngFor** directive.

Our version will take an integer input. This int represents how many times we want to repeat the element that we are attached to.

```
import {Directive, Input} from '@angular/core';
import {TemplateRef, ViewContainerRef} from '@angular/core';

@Directive({ selector: '[repeatMe]' })
export class RepeatMe {
  constructor(private _templateRef: TemplateRef,
              private _viewContainer: ViewContainerRef) {}
  @Input() set repeatMe(count: int) {
    for (var i = 0; i < count; i++) {
      this._viewContainer.createEmbeddedView(this._templateRef);
    }
   }
}
```

Pipes: Pipes are a great way to transform and format data right from your templates.

Currency Pipe

This pipe is used for formatting currencies. Its first argument is an abbreviation of the currency type (e.g. "EUR", "USD", and so on):

```
{{ 1234.56 | currency:'CAD' }}
```

If we want the currency symbol to be printed out; pass second parameter, the string symbol-narrow.

```
{{ 1234.56 | currency:"CAD":"symbol-narrow" }}
```

Date Pipe

This pipe is used for the transformation of dates. The first argument is a format string, like so:

```
<div>
  <h4> Date</h4>
  <div>
    <p ngNonBindable>{{ dateVal | date: 'shortTime' }}</p>
    <p>{{ dateVal | date: 'shortTime' }}</p>
    <p ngNonBindable>{{ dateVal | date:'fullDate' }}</p>
    <p>{{ dateVal | date: 'fullDate' }}</p>
    <p ngNonBindable>{{ dateVal | date: 'd/M/y' }}</p>
    <p>{{ dateVal | date: 'd/M/y' }}</p>
  </div>
</div>
```

Decimal Pipe

This pipe is used for transformation of decimal numbers. The first argument is a format string of the form "{minIntegerDigits}. {minFractionDigits}-{maxFractionDigits}":

```
<div class="card card-block">
  <div class="card-text">
    <h4 class="card-title">DecimalPipe</h4>
    <p ngNonBindable>{{ 3.14159265 | number: '3.1-2' }}</p>
    <p>{{ 3.14159265 | number: '3.1-2' }}</p>

    <p ngNonBindable>{{ 3.14159265 | number: '1.4-4' }}</p>
    <p>{{ 3.14159265 | number: '1.4-4' }}</p>
  </div>
</div>
```

JSON Pipe

This transforms a JavaScript object into a JSON string.

```
<div><div>
    <p ngNonBindable>{{ jsonVal }}</p>  <p>{{ jsonVal }}</p>
    <p ngNonBindable>{{ jsonVal | json }}</p> <p>{{ jsonVal | json }}</p>
</div></div>
```

Custom Pipe

we create a pipe that takes a string and reverses the order of the letters. Here's the code that would go into a reverse-str.pipe.ts file

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'reverseStr'})
export class ReverseStr implements PipeTransform {
  transform(value: string): string {
    let newStr: string = "";
    for (var i = value.length - 1; i >= 0; i--) {
      newStr += value.charAt(i);
    }
    return newStr;
  }
}
```

Finally, in your templates this is how you would use this custom pipe:

```
{{ user.name | reverseStr }}
```

You can also define any number of parameters in the pipe, which enables you to pass parameters to it. For example, here's a pipe that adds a string before and after our provided string:

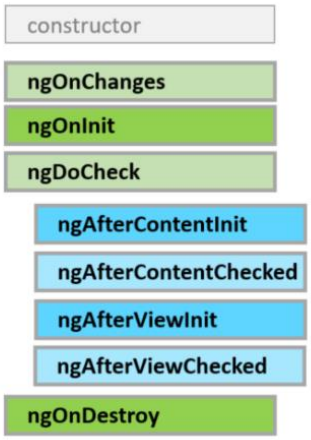
```
@Pipe({name: 'uselessPipe'})
export class uselessPipe implements PipeTransform {
  transform(value: string, before: string, after: string): string {
    let newStr = `${before} ${value} ${after}`;
    return newStr;
  }
}
```

you would call it like that:

```
{{ user.name | uselessPipe:"Mr.":"the great" }}
```

Angular Component Lifecycle

- A component has a lifecycle managed by Angular.
- Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.
- Angular offers lifecycle hooks (interfaces) that provide visibility into these key life moments and the ability to act when they occur.
- Developers can tap into key moments in that lifecycle by implementing one or more of the *lifecycle hook* interfaces in the Angular core library.
- Each interface has a single hook method whose name is the interface name prefixed with ng. For example, the OnInit interface has a hook method named ngOnInit() that Angular calls shortly after creating the component:



After creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

Change Detection

- It is the mechanism by which Angular determines which components need to be refreshed because of changes in the data of the application. Angular can detect when component data changes, and then automatically re-render the view to reflect that change. Let's start with a component to explain how change detection works in angular.

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

@Component ({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  foods = ['Bacon', 'Lettuce', 'Tomatoes'];
  addFood(food) {
    this.foods.push(food);
  }
}
```

And the template looks like this:

```
<input #newFood type="text" placeholder="Enter a new food">
<button (click)="addFood(newFood.value)">Add food</button>
<app-child [data]="foods"></app-child>
```

Let us explore child.component.ts:

```
import {Component, Input} from '@angular/core';
@Component({
            selector: 'app-child',
            templateUrl: './child.component.html'
})
export class ChildComponent {
  @Input() data: string[];
}

// Child.component.html
<ul>  <li *ngFor="let item of data">{{ item }}</li> </ul>
```

Now let's set the change detection strategy in the child component to OnPush:

```
import { Component, Input, ChangeDetectionStrategy } from '@angular/core';
@Component({ selector: 'app-child',
             templateUrl: './child.component.html',
             changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChildComponent {  @Input() data: string[];}
```

With this, things don't seem to work anymore. The new data still gets pushed into our foods array in the parent component, but Angular don't see a new reference for the data input and therefore doesn't run change detection on the component.

To make it work again, the trick is to pass a completely new reference to our data input. This can be done with something like this instead of Array.push in our parent component's addFood method:

```
addFood(food) {  this.foods = [...this.foods, food]; }
```

With this variation, we are not mutating the foods array anymore, but returning a completely new one. Et voilà, things are working again in our child component! Angular detected a new reference to data, so it ran it's change detection on the child component.

ChangeDetectorRef

When using a change detection strategy of OnPush, other than making sure to pass new references every time something should change, we can also make use of the ChangeDetectorRef for complete control.

ChangeDetectorRef.detectChanges()

We could for example keep mutating our data, and then have a button in our child component with a refresh button like this:

```
import { Component,Input,
         ChangeDetectionStrategy,ChangeDetectorRef } from '@angular/core';
@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChildComponent {
  @Input() data: string[];
  constructor(private cd: ChangeDetectorRef) {}
  refresh() { this.cd.detectChanges(); }
}
```

And now when we click the refresh button, Angular runs change detection on the component.

ChangeDetectorRef.markForCheck()

Let's demonstrate with example using a RxJS Behavior Subject:

```
import { Component } from '@angular/core';
import { BehaviorSubject } from 'rxjs/BehaviorSubject';
@Component({ ... })
export class AppComponent {
  foods = new BehaviorSubject(['Bacon', 'Letuce', 'Tomatoes']);
  addFood(food) { this.foods.next(food); }
}
```

And we subscribe to it in the OnInit hook in our child component:

```
import { Component, Input, ChangeDetectionStrategy,
         ChangeDetectorRef,OnInit } from '@angular/core';

import { Observable } from 'rxjs/Observable';

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChildComponent implements OnInit {
  @Input() data: Observable<any>;
  foods: string[] = [];

  constructor(private cd: ChangeDetectorRef) {}

  ngOnInit() {
    this.data.subscribe(food => {
      this.foods = [...this.foods, ...food];
    });
  }
}
```

As here new data mutates our data observable, so Angular doesn't run change detection. The solution is to call ChangeDetectorRef's markForCheck when we subscribe to our observable:

```
ngOnInit() {
  this.data.subscribe(food => {
    this.foods = [...this.foods, ...food];
    this.cd.markForCheck();
  });
}
```

markForCheck instructs Angular that this input should trigger change detection when mutated.

ChangeDetectorRef.detach() and ChangeDetectorRef.reattach()

Yet another powerful thing you can do with ChangeDectorRef is to completely detach and reattach change detection manually with the detach and reattach methods.

How does the default change detection mechanism work?

By default, Angular Change Detection works by checking if the value of template expressions have changed. This is done for all components. By default, Angular does not do deep object comparison to detect changes, it only considers properties used by the template

Angular Forms

- Forms are the mainstay of business applications. You use forms to log in, submit a help request, place an order, book a flight, schedule a meeting, and perform countless other data-entry tasks.
- Developing forms requires design skills, as well as framework support for two-way data binding, change tracking, validation, and error handling,

As Frontend Application Developer (Angular), we:

- Build an Angular form with a component and template.
- Use ngModel directive to create two-way data bindings for reading and writing input-control values.
- Track state changes and the validity of form controls.
- Provide visual feedback using special CSS classes that track the state of the controls.
- Display validation errors to users and enable/disable form controls.
- Share information across HTML elements using template reference variables.

Angular makes the process easy by handling many of the repetitive, boilerplate tasks you'd otherwise wrestle with yourself.

Template Driven Forms

Template driven forms are simple forms which can be used to develop forms. These are called template-driven as everything that we are going to use in a application is defined into the template that we are defining along with the component. Let's see step by step how we can develop and use these forms in the application.

```
<div>
 <form #regForm='ngForm' (ngSubmit)="Register(regForm)" >
  <h2>Registration page</h2>
   <div >
   <input type="text" placeholder="First Name" name="firstname" ngModel>
   <input type="text" placeholder="Last Name" name="lastname" ngModel >
   <input type="email" id="email" placeholder="Email" name="email"ngModel>
   <br/>
   <button type="submit" id="register">Register</button>
   </div>
 </form>
 </div>
```

NgForm

It is the directive which helps to create the control groups inside form directive.

It is attached to the <form> element in HTML and supplements form tag with some additional features.

NgModel

Event handler for submitting for look like as shown below code snippet.

```
import { Component, OnInit } from '@angular/core';
import { NgForm } from '@angular/forms';
@Component({ …})
export class FormdemobasicsComponent implements OnInit {
  Register(regForm:NgForm){  console.log(regForm);  }
}
```

Template of Component:

```
<div >
<form #form="ngForm" (ngSubmit)="registerUser(form)">
 <h2 class="text-center">Registration page</h2>
 <input type="text" placeholder="First Name" name="firstname"
        required ngModel>
 <input type="text" placeholder="Last Name" name="lastname"
        required ngModel>
 <input type="email" id="email" placeholder="Email" name="email"
        email required ngModel #email="ngModel">
 <span  *ngIf="email.touched && !email.valid ">
         Please enter the Email Value
 </span>
 <br/>
 <button type="submit" class="btn btn-default" id="register"
         [disabled]="!form.valid" >Register</button>
</form>
</div>
```

Reactive Forms

- For explicit management of data between the UI and the Model reactive forms are used.
- Using Reactive forms, we create the tree of Angular form controls and bind them in the native form controls.

1. Import **ReactiveFormsModule** in the application module and add it in import array.
2. import the **FormGroup**, **FormControl** and **FormBuilder** classes to the component.

FormControl : It tracks the value of the controls and validates the individual control in the form.

FormGroup: Tracks the validity and state of the group of FormControl Instance or moreover, we can say the FormGroup to be a collection of FormControls.

FormBuilder

```
import { Component, OnInit } from '@angular/core';
import {FormBuilder,FormGroup,FormControl,Validators,NgForm} from
                                                          '@angular/forms'
@Component({
  selector: 'app-reactive-demo',
  templateUrl: './reactive-demo.component.html',
  styleUrls: ['./reactive-demo.component.css']
})
export class ReactiveDemoComponent implements OnInit {
signupForm:FormGroup;
FirstName:string="";  LastName:string="";
Email:string="";      Password:string="";

constructor(private frmbuilder:FormBuilder){
    this.signupForm=frmbuilder.group({
    fname:new FormControl(),  lname:new FormControl(),
    Emailid:new FormControl(),userpassword:new FormControl()
    });
}

ngOnInit() {   }
PostData(signupForm:NgForm)  { console.log(signupForm.controls); }

}
```

Component Template would look like as shown below:

```
<div>
  <form [formGroup]='signupForm' (ngSubmit)="PostData(signupForm)">
   <input type="text" formControlName='fname' placeholder="Your First name">
   <input type="text" formControlName='lname' placeholder="Your Last name">
   <input type="text" formControlName='Emailid' placeholder="Your Email id">
   <input type="Password" formControlName='userpassword'
          placeholder="Your password">
   <input type="Submit" value="Post Data" >
  </form>
</div>
```

Validations with Reactive forms

Now, the most important part of any form is to add the validations in the application. Let's see how we can add the validation in the form. For that, let's check the **FormControl** in the component.

When we do the fname:new FormControl(), the first argument in the **FormControl** is the initial value and the second value is the Validations that we can add there.

The only change in the component would be like as shown below.

```
this.signupForm= frmbuilder. group({

fname:['',Validators.compose([Validators.required,Validators.maxLength(15),Va
lidators.minLength(1)])],
  lname:['',[Validators.required,Validators.maxLength(19)]],
  Emailid:['',[Validators.required,Validators.email]],
  userpassword:['',Validators.required]
})
```

We can set more than one validations for the controls by using Validators.compose.

```
<div>
  <form [formGroup]='signupForm' (ngSubmit)="PostData(signupForm)">
   <input type="text" formControlName='fname'
          name='fname' id='fname' placeholder="Your First name">
   <div *ngIf="signupForm.controls['fname'].touched &&
        !signupForm.controls['fname'].valid">
     <span *ngIf="signupForm.controls['fname'].hasError('required') ">
           First name is required
     </span>
     <span *ngIf="signupForm.controls['fname'].hasError('minlength') ">
           Min length is 1
     </span>
     <span *ngIf="signupForm.controls['fname'].hasError('maxlength') ">
           max length is 15
     </span>
   </div>
    <input type="text" formControlName='lname' placeholder="Your Last name">
    <div *ngIf="signupForm.controls['lname'].touched &&
         !signupForm.controls['lname'].valid">
       <span *ngIf="signupForm.controls['lname'].hasError('required') ">
             Last name is required
       </span>
       <span *ngIf="signupForm.controls['lname'].hasError('minlength') ">
            Min length is 1
       </span>
       <span *ngIf="signupForm.controls['lname'].hasError('maxlength') ">
            max length is 15
        </span>
    </div>
     <div><input type="text" formControlName='Emailid'
                placeholder="Your Email id">
    </div>

    <div *ngIf="signupForm.controls['Emailid'].touched &&
         !signupForm.controls['Emailid'].valid">
     <span *ngIf="signupForm.controls['Emailid'].hasError('required') ">
           Email ID is needed
     </span>
     <span *ngIf="signupForm.controls['Emailid'].hasError('email') ">
           Invalid Email ID
     </span>
    </div>
    <div>
      <input type="Password" formControlName='userpassword'
             placeholder="Your password">
    </div>
    <div>
     <input type="Submit" value="Post Data" [disabled]='!signupForm.valid' >
    </div>
  </form>
```

Single Page Applications (SPA) [Navigation]

It's just one index.html file, with a CSS bundle and a JavaScript bundle. While a SPA is running, only data gets sent over the wire, which takes a lot less time and bandwidth than constantly sending HTML.

User Interface (UI) Routing

In traditional websites, routing is handled by a router on the server:

1. T user clicks a link in the browser, causing the URL to change
2. The browser sends an HTTP request to server
3. The server reads the URL from the HTTP request and generates an appropriate HTTP response
4. The server sends the HTTP response to the browser.

In modern JavaScript web applications, routing is often handled by a JavaScript router in the browser.

JavaScript router

A JavaScript router does two things:

1. update the web application state when the browser URL changes
2. update the browser URL when the web application state changes.

JavaScript routers make it possible for us to develop single-page applications (SPAs).

Angular Router

Angular Router is an official Angular routing library, written and maintained by the Angular Core Team. It's a JavaScript router implementation that's designed to work with Angular and is packaged as **@angular/router**.

Angular Router takes care of the duties of a JavaScript router:

- It activates all components to compose a page when a user navigates to a certain URL.
- It lets users navigate from one page to another without page reload.
- It updates the browser's history, so the user can use the back and forward buttons when navigating back and forth between pages.

In addition, Angular Router allows us to:

- Redirect a URL to another URL
- Resolve data before a page is displayed
- Run scripts when a page is activated or deactivated
- Lazy load parts of our application.

How Angular Router Works?

When a user navigates to a page, Angular Router performs the following steps in order:

1. It reads the browser URL the user wants to navigate to.
2. It applies a URL redirect (if one is defined).
3. It figures out which router state corresponds to the URL.
4. It runs the guards that are defined in the router state.
5. It resolves the required data for the router state.
6. It activates the Angular components to display the page.
7. It manages navigation and repeats the steps above when a new page is requested.

To accomplish its tasks, Angular Router introduces the following terms and concepts:

- **Router Service**: the global Angular Router service in our application.
- **Router Configuration**: definition of all possible router states our application can be in.
- **Router State**: the state of the router at some point in time, expressed as a tree of activated route snapshots.
- **Activated Route Snapshot**: provides access to the URL, parameters, and data for a router state node.
- **Guard**: script that runs when a route is loaded, activated or deactivated.
- **Resolver**: script that fetches data before the requested page is activated.
- **Router Outlet**: location in the DOM where Angular Router can place activated components.

An Angular application that uses Angular Router only has one router service instance: It's a singleton. Whenever and wherever you inject the Router service in your application, you'll get access to the same Angular Router service instance.

Enabling Routing

To enable routing in our Angular application, we need to do three things:

1. Create a routing configuration that defines the possible states for our application
2. Import the routing configuration into our application
3. Add a router outlet to tell Angular Router where to place the activated components in the DOM.

Route.ts

```
export const childRoutes: Routes = [
  { path: '', redirectTo: 'main', pathMatch: 'full' },
  { path: 'main', component: MainComponent },
  { path: 'interest', component: InterestComponent },
  { path: 'flower', component: FlowerComponent },
  { path: ':id', component: ByIdComponent }
];

export const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full' },
  {path: 'home', component: HomeComponent },
  {path: 'about', component: AboutComponent },
  {path: 'contact',component: ContactComponent },
  {path:'dashborad', component:DashboardComponent},
  {path: 'products', component: ProductsComponent, children: childRoutes },
  {path: 'protected', component: ProtectedComponent,
   canActivate: [LoggedInGuard]}
];
```

appmodule.ts

```
@NgModule({
  declarations: [ ],
  imports:[BrowserModule,
           RouterModule.forRoot(routes)
  ],
  providers: [
              {provide:APP_BASE_HREF,useValue:'/'},
              {provide:LocationStrategy,useClass:HashLocationStrategy}
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

routeappcomponent.ts

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

@Component({
            selector: 'router-app',
            template: `<div class="page-header">
                        <div class="container">
                            <h1>Nested Router</h1>
                            <div class="navLinks">
                              <a [routerLink]="['/home']">Home</a>
                              <a [routerLink]="['/about']">About us</a>
                              <a [routerLink]="['/contact']">Contact Us</a>
                              <a [routerLink]="['/products']">Products</a>
                            </div>
                        </div>
                      </div>

                      <div>
                        <div>
                            <router-outlet></router-outlet>
                        </div>

                    </div> `
          })
export class RootComponent {
          query: string;
}
```

Angular Services & Dependency Injection

Angular application is built from components – organized in modules with hierarchy. The component represents a user interface element with properties, methods, input and output parameters and more. When we want interaction between components we use input and output parameters but if the interaction is not between parent and child it can be complex to make it use the parent. this is where Angular services help

A Service is a functionality that can be shared with different Components. Whenever we find some common aspects in our components, we usually separate it out as a Service.

Dependency Injection is a coding pattern in which a class takes the instances of objects it needs which is called dependencies from an external source rather than creating them itself.

In Angular, services are Singleton. It internally implements the Service Locator Pattern. This means each service register itself under one container as a single instance. Angular provides a built in Injector which acts as a container to hold the single instances of all registered services. Angular takes care of creation of Service instance and registering it to the Service container.

The @Injectable () decorator

To inject the service into component, Angular provides an Injector decorator: **@Injectable ().**

We broadly have the following steps to create a Service:

1) Create the service class
2) Define the metadata with a decorator
3) Import what we need.

1. IProfile interface

```
export interface IProfile{
  Name: string;
  Age: number;
}
```

2. Profile Class

```
import {Component, Injectable} from '@angular/core'
import {IProfile} from './iprofile'

@Component({
  selector: 'profile',
  template: `
    <div>
      <h2>Name : {{MyProfile.Name}}</h2>
      <h2>Age : {{MyProfile.Age}}</h2>
    </div>
  `
})
export class Profile{
  MyProfile : IProfile;
  constructor(){ }
}
```

3. ProfileService class

```
import {Injectable} from '@angular/core'
import {IProfile} from './iprofile'
@Injectable()
export class ProfileService{
  getProfiles(): IProfile[]{
    return [{ Name:'Ravi Tambade', Age: 42},
            { Name:'Omkar Hinge',  Age: 24}
           ];
 }
}
```

Registering a Service:

A Service can be registered in any component. However, it should be registered at root level component only. This is because it we register at two different nested level component, then the service will have two entire different instances in both components. However, we can register at any nested level component just in case if we require to use the service in that component or it's child components only.

```
import {ProfileService} from './profile.service';
.
.
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ App ],
  providers:[ProfileService],
  bootstrap: [ App ]
})
```

Injecting a Service in component:

We can inject our service as a parameter to our constructor class.

```
constructor(private _profileService: ProfileService){ }
```

Modify the Profile class as below:

```
import {Component, Injectable} from '@angular/core'
import {ProfileService} from './profile.service';
import {IProfile} from './iprofile'

@Component({
  selector: 'profile',
  template: `
    <div>
      <h2>Name : {{MyProfile.Name}}</h2>
      <h4>Age : {{MyProfile.Age}}</h4>
    </div>
  `,
})
export class Profile implements OnInit{  ProfileList : IProfile[];

  MyProfile : IProfile;
  constructor(private _profileService: ProfileService){  }

  ngOnInit(){
    this.ProfileList = this._profileService.getProfiles();
    this.MyProfile = this.ProfileList[0];
  }
}
```

Consuming RESTful Service with HttpClient and RxJS

- REST or HTTP based services are the primary requirements of single page applications to retrieve the data and gel into the web application.
- Angular offers its inbuilt http client service which wraps the major functions for requesting the data from server where REST service is hosted.
  Angular provides **HttpClient** for this purpose which is packaged under @angular/common/http.

Before we can use HttpClient, we must import **HttpClientModule** in our main application module.

```
import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {HttpClientModule} from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule,
  ],
})
export class MyAppModule {}
```

RxJS

HttpClient is best consumed when used with Observables.

An Observable is simply a collection of future events which arrives asynchronously over time.

An Observable is a class provided by RxJs. To learn more about RxJS, please visit **http://reactivex.io/rxjs/**

Observables are pretty much like promise or callbacks but the main advantage is they are lazy loaded. Observables will not be called till the time any subscriber invoked on it.

Building our example app

Creating our survey data from our Http service as JSON.

Create a survey interface

```
export interface ISurvey{
 question:string;
 choices: string[];
}
```

Our Survey Data

```
 [
    {   "question": "What is your favourite programming language?",
        "choices": ["Swift","Python","Objective-C","Ruby",”Java”,”C#”]},
    {   "question": "What is your source of Learning?",
        "choices": ["Books","Online Tutorials","Transflower boot comps"]
    }
]
```

Our Survey Service

```
@Injectable()
export class SurveyService { ..  }
```

Injecting HttpClient into Survey service

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

export class SurveyService {
  constructor(private _http:HttpClient){  ..  }
```

Adding Observable method in Survey service

```
import {Observable} from 'rxjs/Observable'
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/map';

getSurveyQuestion(): Observable<ISurvey[]>{
    return this._http
      .get<ISurvey[]>('./src/survey.json')
      .do(data =>console.log('All : ' + JSON.stringify(data)))
      .catch(this.handleError);
  }
```

Injecting our Survey Service:

```
import {ISurvey} from './isurvey'
import {SurveyService} from './survey.service';
export class Survey {
  MySurvey : ISurvey[] = [];
  constructor(private _surveyService: SurveyService){ ...}
}
```

Calling our observable method from SurveyService

```
export class Survey implements OnInit{
  tflSurvey : ISurvey[] = [];
  constructor(private _surveyService: SurveyService){ }

  ngOnInit(){
    this._surveyService.getSurveyQuestion().subscribe(
      data => {    console.log(data);
                   this.tflSurvey = data;
               },
      err => { console.log(err);});
  }
}
```

Component Template which will be rendered as soon as data gets populated in our ISurvey [].

```
Component({
  selector: 'survey',
  template: `<div>
              <h2>Question : </h2>
               <div *ngIf="tflSurvey && tflSurvey.length">
               <div *ngFor="let survey of tflSurvey; let idx = index"><br/>
                <div>{{survey.question}}</div>
                <div *ngFor="let choice of survey.choices">
                 <input type="radio" name="radioGroup{{idx}}"/>{{choice}}
                </div>
               </div>
              </div>
             </div>`
})
export class Survey implements OnInit{  ...  }
```

The RxJS Retry operator

If you want to retry your http requests on case you receive error in your http response, then there is a retry operator (provided by RxJs) that you can use to resubscribe to the observable to re-invoke it for several times.
Following code will retry 3 times as http service continues to receive any http error.

```
import 'rxjs/add/operator/retry';
http.get('/api/questions').retry(3).subscribe(...);
```

CRUD Operation with Angular HTTPClient

HTTP is the messaging system between the client and the server. The client sends the request and the server responds with the proper message. Angular HTTP Client is the toolkit which enables us to send and receive data over RESTful HTTP endpoints.

In Angular 4.3, this Angular HTTP API was provided, which is the extension to the existing API that provides some new features and added-in its own package of the **@angular/common/http**.

Client-Side Setup

To make the HTTP client module available in the application, we must make sure it is included and configured properly in the application. Let's see, step-by-step, how we can do this.

1. Import the HTTP client module into the application module or root module so that our root module is app.module.ts.

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {HttpClientModule} from '@angular/common/http';
import {FormsModule } from '@angular/forms'
import {AppComponent } from './app.component';
@NgModule({
    declarations: [ AppComponent],
    imports: [ BrowserModule, HttpClientModule, FormsModule],
    providers: [],
    bootstrap: [AppComponent]
})
export class AppModule {}
```

Now that we have imported the HTTP Client in our application, we can use them in our component easily.

Let's add one component to our Angular application.

```
<div class="container">
 <h3>Employee List</h3>
  <table class="table table-condensed">
    <thead>
      <tr>
        <td></td><td></td><td></td><td></td>
        <td><a (click)="ShowRegForm(e)">Add New</a></td>
      </tr>
      <tr>
        <th>ID</th><th>First Name</th><th>Last Name</th><th>Email</th>
        <th>Edit</th>
      </tr>
    </thead>
    <tbody>
      <tr class="success" *ngFor="let e of employeelist ">
        <td> {{e.id}}</td>
        <td>{{e.fname}}</td>
        <td>{{e.lname}}</td>
        <td>{{e.email}}</td>
        <td><a (click)="ShowRegForm(e)">Edit</a></td>
        <td><a (click)="ShowRegFormForDelete(e)">Delete</a></td>
      </tr>
    </tbody>
  </table>
</div>
<hr>
<form #regForm="ngForm">
<div class="container" *ngIf="editCustomer">
 <h3>{{FormHeader}}</h3>
  <table class="table table-condensed">
    <tbody>
      <tr>
        <td>First Name</td>
        <td><input type="text" name="fname" [(ngModel)] ></td>
      </tr>
      <tr>
        <td>Last Name</td>
        <td><input type="text" name="lname" [(ngModel)]></td>
      </tr>
      <tr>
        <td> Email</td>
        <td><input type="text" name="email" [(ngModel)]></td>
      </tr>
      <tr>
        <td><input type="hidden" name="id" [(ngModel)]></td>
        <td><input type="button" value="Save" (click)="Save(regForm)"></td>
      </tr>
    </tbody>
  </table>
 </div>
</form>
```

1. Constructor and Injecting the Data Service

```
constructor(private dataservice: EmployeeDataService) { }
```

2. Invoking the Data Service

```
this.dataservice.getEmployee().subscribe(
                                    (tempdate) => {this.;}),
                                    err => {console.log(err);}
    }
   (employee) {
       this.;
       if (employee!) {
           this.SetValuesForEdit(employee)
       } else {
           this.ResetValues();
       }
   }
   (employee) {
       this.;
       if (employee!) {
           this.SetValuesForDelete(employee)
       }
   }
   (employee) {this..fname;
               this..lname;
               this..email;
               this..id;
               this.FormHeader = "Delete"
   }

//Function to set the values for edit form
  (employee) {this..fname;
              this..lname;
              this..email;
              this..id;
              this.FormHeader = "Edit"
  }

//Function to reset the values
 ResetValues() {this.fname = "";
                this.lname = "";
                this.email = "";
                this.id = "";
                this.FormHeader = "Add"
}
```

```
//Common function for the Operation

Save(regForm: NgForm) {
    this.GetDummyObject(regForm);
    switch (this.FormHeader) {
        case "Add":this.Addemployee(this.Dummyemployee); break;
        case "Edit":this.UpdateEmployee(this.Dummyemployee); break;
        case "Delete":this.DeleteEmployee(this.Dummyemployee); break;
        default: break;
    }
}

GetDummyObject(regForm: NgForm): employee {
    this.Dummyemployee = new employee
    this.Dummyemployee..value.email;
    this.Dummyemployee..value.fname;
    this.Dummyemployee..value.lname;
    this.Dummyemployee..value.id;
    return this.Dummyemployee;
}

Addemployee(e: employee) {
    this.dataservice.AddEmployee(this.Dummyemployee).subscribe(res => {
        this.employeelist.push(res);
        alert("Data added successfully !! ")
        this.;
    }), err => {console.log ("Error Occured " + err);}
}

UpdateEmployee(e: employee) {
    this.dataservice.EditEmployee(this.Dummyemployee).subscribe(
      res => {this.;
              this.dataservice.getEmployee().subscribe(res => {this.;});
     alert("Employee data Updated successfully !!")
    });
}

DeleteEmployee(e: employee) {
    this.dataservice.DeleteEmployee(this.Dummyemployee).subscribe(res => {
        this.;
        this.dataservice.getEmployee().subscribe(res => {this.;});
        alert("employee Deleted successfully !! ")
    });
}
```

```
export interface employee {ID: string;Fname: string;
                           Lname: string; Email: string;}

@Injectable()
export class EmployeeDataService {
    employees: Observable <employee[]>;
    newemployee: Observable <employee>;

    constructor(private http: HttpClient) {   }

    GetEmployee() {return this.http.get <employee[]> (ROOT_URL +
                  '/Employees')}
    AddEmployee(emp:employee) {
      const headers = new HttpHeaders().set('content-type',
                                            'application/json');
    var body = {Fname: emp.Fname,
                Lname: emp.Lname,
                Email: emp.Email }
    return this.http.post <employee> (ROOT_URL + '/Employees', body,
               { headers })
    }

    EditEmployee(emp: employee) {
        const params = new HttpParams().set('ID', emp.ID);
        const headers = new HttpHeaders().set('content-type',
                                              'application/json');
        var body = { Fname: emp.Fname,
                     Lname: emp.Lname,
                     Email: emp.Email,
                     ID: emp.ID }
     return this.http.put <employee>(ROOT_URL + '/Employees/' + emp.ID,
                                              body, {headers, params})
    }

    DeleteEmployee(emp: employee) {
      const params = new HttpParams().set('ID', emp.ID);
      const headers = new HttpHeaders().set('content-type',
                                            'application/json');
     var body = { Fname: emp.Fname,
                  Lname: emp.Lname,
                  Email: emp.Email,
                  ID: emp.ID  }
     return this.http.delete<employee>(ROOT_URL+'/Employees/'+emp.ID)
    }
}
```