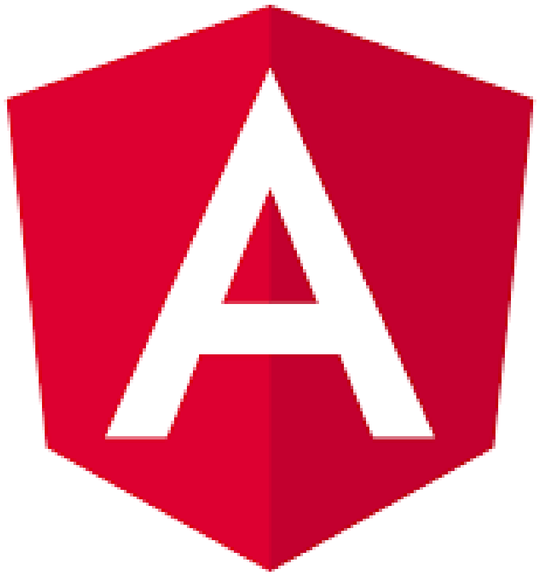
Angular




website=True,
gs):
pes']
[]
s == 'local_bachelor_program_hsc':
domain = [('course_id.is_local_bachelor_prog
elif types == 'local_bachelor_program_a_level':
domain = [('course_id.is_local_bachelor_prog
elif types == 'local_bachelor_program_diploma':
domain = [('course_id.is_local_bachelor_prog
elif types == 'local_masters_program_bachelor':
domain = [('course_id.is_local_masters_progr
elif types == 'international_bachelor_program':
domain = [('course_id.is_international_bache
elif types == 'international_masters_program':
domain = [('course_id.is_international_maste
domain.append(('state', '=', 'applicati
admission_register_list = http
if admission_register
for progr

Introduction

Introduction

- JavaScript framework for writing web applications
- Handles: DOM manipulation, input validation, server communication, URL mangement, etc.
- Uses Model-View-Controller pattern
- HTML Templating approach with two-way binding
- Minimal server-side support dictated
- Focus on supporting for programming in the large and single page applications
- Modules, reusable components, testing, etc

Features of Angular

- Components: Components help to build the applications into many modules. This helps in better maintaining the application over a period of time.
- Typescript: Angular is based on Typescript.This is a superset of JavaScript and is maintained by Microsoft
- Services: Services are a set of code that different components of an application can share. So for example if you had a data component that picked data from a database, you could have it as a shared service that could be used across multiple applications.
- Better event-handling capabilities, powerful templates, and bettersupport for mobile devices.

Components of Angular

- **Modules:** This is used to break up the application into logical pieces of code. Each piece of code or module is designed to perform a single task.
- **Component** This can be used to bring the modules together.
- **Templates** This is used to define the views of an Angular JS application.
- **Metadata** This can be used to add more data to an Angular JS class
- **Service** This is used to create components which can be shared across the entire application.

Angular Environment

- Npm Node package manager that is used to work with the open source repositories. Angular JS as a framework has dependencies on other components. npm can be used to download these dependencies and attach them to your project.
- Git This is the source code software that can be used as a repository
- Editor: There are many editors that can be used for Angular JS development such as Visual Studio code and WebStorm.

Angular CLI

- The Angular CLI is a tool to initialize, develop, scaffold and maintain Angular applications
- Command Line Interface (CLI) can be used to create our Angular JS application.
- It also helps in creating a unit and end-to-end tests for the application.
  - Install NodeJs first. - Go to nodejs.org and download the latest version
    - uninstall (all) installed versions on your machine first
  - npm install -g @angular/cli
  - ng new my-project --skip-tests
  - cd my-project
  - ng serve

Angular Modules

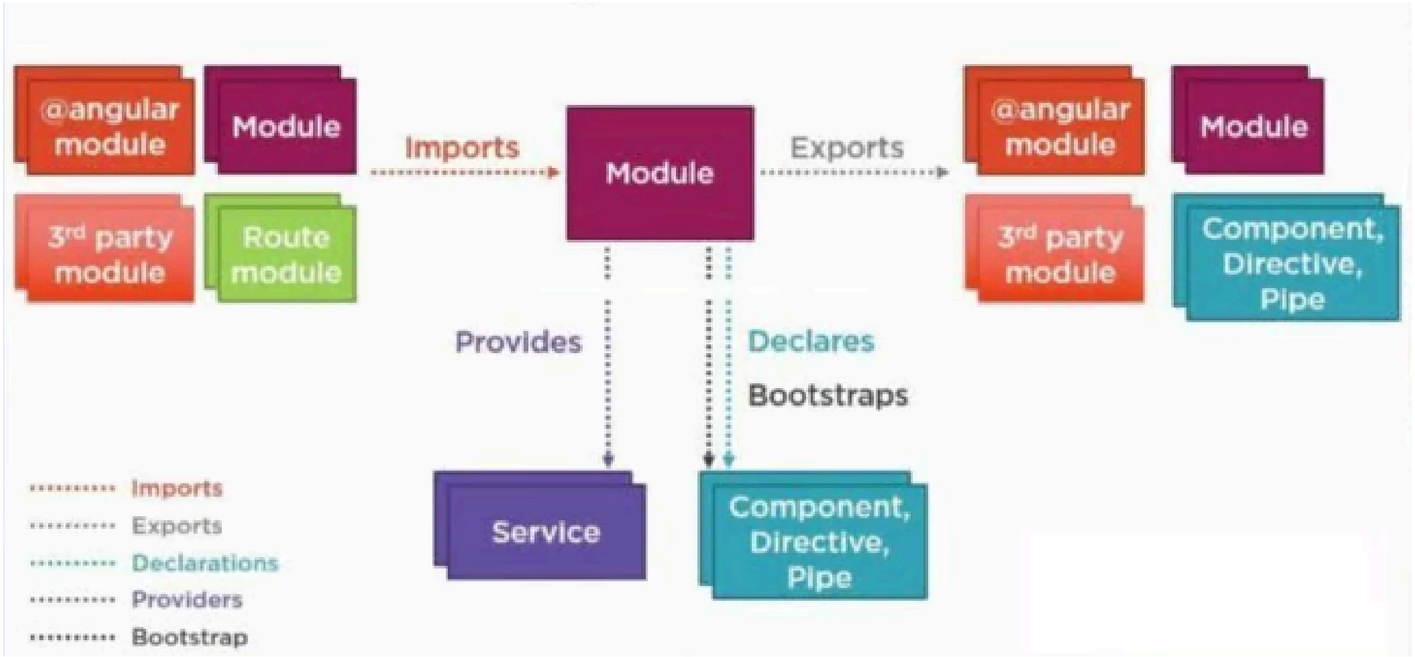
- In Angular, a module is like a container that bundles together parts of an app such as components, services, and directives. It helps organize the application into blocks of functionality.
- Each Angular application starts with at least one module, known as the root module. This is typically called AppModule and it’s where the application begins.
- Modules can also use functionalities from other modules by importing them. This way, Angular apps can be modular and scalable, breaking down features into manageable pieces.
- Key Aspects of Modules
  - Declarations: This section declares which components, directives, and pipes belong to the module.
  - Imports: Here, you can import functionality from other modules. For instance, if you need Angular’s forms features, you import FormsModule .
  - Providers: Services that are to be used within the module are listed here.
  - Bootstrap: The root module uses this to define the root component that Angular should load first.

Angular Modules

- Modules: Modules are used in Angular JS to put logical boundaries in your application.
  - Instead of coding everything into one application, you can instead build everything into separate modules to separate the functionality of your application.
- In Visual Studio code, go to the app.module.ts folder in your app folder. This is known as the root module class.
- A module is made up of the following parts –
  - Bootstrap array - This is used to tell Angular JS which components need to be loaded so that its functionality can be accessed in the application.
  - Export array - This is used to export components, directives, and pipes which can then be used in other modules.
  - Import array – Just like the export array, the import array can be used to import the functionality from other Angular JS modules

Angular Modules

```
import { BrowserModule } from "@angular/platform-browser";
import { NgModule } from "@angular/core";

import { AppRoutingModule } from "./app-routing.module";
import { AppComponent } from "./app.component";

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, AppRoutingModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```



Sample Application Demo

- Create and Execute a sample application

index.html

src > index.html > ...

```
1  <!doctype html>
2  <html lang="en">
3  <head>
4    <meta charset="utf-8">
5    <title>AngularTrainingApp</title>
6    <base href="/">
7    <meta name="viewport" content="width=device-width, initial-scale=1">
8    <link rel="icon" type="image/x-icon" href="favicon.ico">
9  </head>
10 <body>
11   <app-root></app-root>
12 </body>
13 </html>
14
```

Main.ts file

src > TS main.ts > ...

```ts
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));

```

- bootstrapApplication is an Angular function that requires an angular component as an argument, this component is displayed as the start page of the application.

```ts
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { PlatformRef } from '@angular/core';


// Create Browser Platform
const platformRef: PlatformRef = platformBrowserDynamic();

// Bootstrap Application
platformRef.bootstrapModule(AppModule);
```

App Component

src > app > app.component.ts > AppComponent

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

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'angular-training-app';
}
```

- @Component is a decorator adds some metadata defined in the decorator to the class that it decorates
- standalone attribute marks the component as a standalone component, other types of components are module components

src > app > app.component.html > ...

Go to component

```html
<main>
  <p>Hello World</p>
</main>
<router-outlet />
```

- Component HTML that will replace app-root in the main.html

App Component

- The @Component decorator is essential as it associates the given template and style with the class. Metadata typically includes:
  - selector: Defines the custom HTML tag that Angular uses to instantiate this component in the template.
  - templateUrl: Points to an external file that contains the template of the component.
  - styleUrls: Lists the stylesheets that apply to this component.

Inline Template and Styles

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

@Component({
  selector: 'app-inline-label',
  template: `
    <div class="inline-label-container">
      <h3>{{ name }}</h3>
      <p><strong>Address:</strong> {{ address }}</p>
      <p><strong>Phone:</strong> {{ phone }}</p>
    </div>
  `,
  styles: [`
    .inline-label-container {
      border: 1px solid #ccc;
      padding: 10px;
      border-radius: 5px;
      width: 300px;
      margin: 10px 0;
      box-shadow: 0 0 5px rgba(0, 0, 0, 0.1);
    }
    h3 {
      margin: 0;
      color: #333;
    }
    p {
      margin: 5px 0;
      color: #555;
    }
  `]
})
export class InlineLabelComponent {
  @Input() name: string = '';
  @Input() address: string = '';
  @Input() phone: string = '';
}
```

View Encapsulation

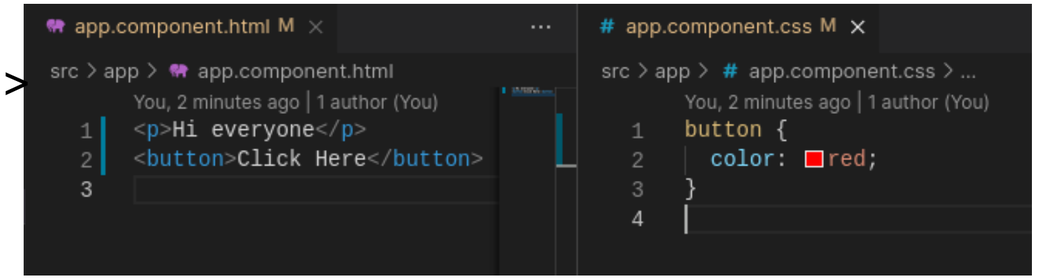
- In Angular, styles defined in a component are scoped to that component.
- This is achieved using ViewEncapsulation, which ensures that styles in one component do not leak or affect the styles of other components.
- **How Angular Scopes Styles**:
  - By default, Angular applies View Encapsulation to components using a mechanism called Shadow DOM emulation.
  - Angular adds unique attributes to the elements within the component's template and the styles are scoped specifically to those elements.

```
@Component({
 selector: 'app-example',
 template: `
  <div class="example-container">
   <p>This is styled text.</p>
  </div>
 `,
 styles: [`
  .example-container {
   background-color: lightgray;
   padding: 10px;
  }
  p {
   color: blue;
  }
 `]
})
export class ExampleComponent { }
```

How View Encapsulation Works

- Angular will modify the component's rendered HTML and CSS to ensure that the styles are only applied to this component's template by adding unique attribute selectors behind the scenes.

```
<div _ngcontent-abc-1 class="example-container">
 <p _ngcontent-abc-1>This is styled text.</p>
</div>
```



- The corresponding styles would only apply to elements that contain the `_ngcontent-abc-1` attribute, ensuring the styles are scoped to the component.


```
@Component({
    encapsulation: ViewEncapsulation.Emulated, // Default mode
    ...
  })
```

```
button[_ngcontent-edj-c16] {
  color: red;
}
```

View Encapsulation Modes

- Angular provides three ViewEncapsulation modes to control how styles are scoped to components:
  - Emulated (Default):
    - Angular’s default behavior, where styles are scoped to the component using an emulation of the Shadow DOM.
    - Styles are encapsulated by adding unique attributes to the elements.
  - None:
    - No style encapsulation; the styles defined in this component will be applied globally across the entire application.
  - Shadow DOM (Native):
    - Uses the actual Shadow DOM available in modern browsers to encapsulate styles. The component will create a real Shadow DOM boundary.
    - Styles defined within the component will only apply inside its Shadow DOM and not affect the rest of the document.

Transpiler

- A transpiler is a special piece of software that translates source code to another source code. It can parse (“read and understand”) modern code and rewrite it using older syntax constructs, so that it’ll also work in outdated engines.
- E.g. JavaScript before year 2020 didn’t have the “nullish coalescing operator” ??. So, if a visitor uses an outdated browser, it may fail to understand the code like height = height ?? 100.
- A transpiler would analyze our code and rewrite **height ?? 100** into
  (height !== undefined && height !== null) ? height : 100.
- Usually, a developer runs the transpiler on their own computer, and then deploys the transpiled code to the server.
- Speaking of names, Babel is one of the most prominent transpilers out there.
- Modern project build systems, such as webpack, provide a means to run a transpiler automatically on every code change, so it’s very easy to integrate into the development process.

Polyfills

- New language features may include not only syntax constructs and operators, but also built-in functions.
- For example, Math.trunc(n) is a function that “cuts off” the decimal part of a number, e.g Math.trunc(1.23) returns 1.
- In some (very outdated) JavaScript engines, there’s no Math.trunc, so such code will fail.
- As we’re talking about new functions, not syntax changes, there’s no need to transpile anything here. We just need to declare the missing function.
- A script that updates/adds new functions is called “polyfill”. It “fills in” the gap and adds missing implementations.

```
if (!Math.trunc) { // if no such function
  // implement it
  Math.trunc = function(number) {
    // Math.ceil and Math.floor exist even in ancient JavaScript engines
    // they are covered later in the tutorial
    return number < 0 ? Math.ceil(number) : Math.floor(number);
  };
}
```

Components


website=True,
gs):
pes']
[]
s == 'local_bachelor_program_hsc':
domain = [('course_id.is_local_bachelor_prog
lif types == 'local_bachelor_program_a_level':
domain = [('course_id.is_local_bachelor_prog
elif types == 'local_bachelor_program_diploma':
domain = [('course_id.is_local_bachelor_prog
elif types == 'local_masters_program_bachelor':
domain = [('course_id.is_local_masters_progr
elif types == 'international_bachelor_program':
domain = [('course_id.is_international_bache
elif types == 'international_masters_program':
domain = [('course_id.is_international_maste
domain.append(('state', '=', 'applicati
admission_register_list = http
if admission_register
for

Components

- A Component is a fundamental part of an angular application
- It is combination of an HTML Template and a component class that controls the portion of the screen

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

@Component({
    selector: 'my-app',
    template: `<h1>Hello {{name}}</h1>`
})
export class AppComponent {
    name = 'Angular';
}
```

- The import statement defines the modules we want to use to write our code. Here we're importing two things: Component, and OnInit.

Components

- A basic Component has two parts:
  - A Component decorator
  - A component definition class
- The selector property tells Angular to display the component inside a custom <app-hello-world> tag in the index.html file
- templateUrl of ./hello-world.component.html - means that we will load our template from the file hello-world.component.html in the same directory as our component.

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-hello-world',
5    templateUrl: './hello-world.component.html',
6    styleUrls: ['./hello-world.component.css']
7  })
8  export class HelloWorldComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15 }
```

Components

- The import statement defines the modules we want to use to write our code. Here we're importing two things: Component, and OnInit.

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-hello-world',
5    templateUrl: './hello-world.component.html',
6    styleUrls: ['./hello-world.component.css']
7  })
8  export class HelloWorldComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15 }
```

Adding a template

- We can define templates in two ways: by using the template key in our @Component object or by specifying a templateUrl URL.
- We could add a template to our @Component by passing the template option
- template string are defined between backticks (` … `) that allows us to do multiline strings.

```
@Component({
  selector: 'app-hello-world',
  template: `
    <p>
      hello-world works inline!
    </p>
  `
})
```

Adding a template

- We can define templates in two ways: by using the template key in our @Component object or by specifying a templateUrl URL.
- We could add a template to our @Component by passing the template option
- template string are defined between backticks (` … `) that allows us to do multiline strings.

```
@Component({
  selector: 'app-hello-world',
  template: `
    <p>
      hello-world works inline!
    </p>
  `
})
```

Adding CSS Styles with styleurls

- We want to use the CSS in the file hello-world.component.css as the styles for this component.
- Angular uses a concept called “style-encapsulation”
- This means that styles specified for a particular component only apply to that component.

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-hello-world',
5    templateUrl: './hello-world.component.html',
6    styleUrls: ['./hello-world.component.css']
7  })
8  export class HelloWorldComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15 }
```

Adding new components

- Use the Angular CLI to add new components to the module
  - ng generate component user-item

- A new component can also be added manually, You need to make the additions to the different files yourself

Adding a class component.ts

- Create a directory within the App called <<component-name>>
- In it create a file called <component-name>.component.ts
- Enter the @Component directive followed by its attributes
- Create a file called <component-name>.component.html
- In app.module.ts
  - add the component name in the declarations
  - add an import for the component
  - import { component-name } from "./server/server.component"
- In the component.html add the selector as a tag

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-hello-world',
5    templateUrl: './hello-world.component.html',
6    styleUrls: ['./hello-world.component.css']
7  })
8  export class HelloWorldComponent implements OnInit {
9
10   constructor() { }
11
12   ngOnInit() {
13   }
14
15 }
```

Adding a component automatically

- Enter the command
  - ng generate component component-name
-

```
C:\Users\Sanjeevan\AngularProjects\my-first-app>ng generate component servers
CREATE src/app/servers/servers.component.html (22 bytes)
CREATE src/app/servers/servers.component.ts (206 bytes)
CREATE src/app/servers/servers.component.css (0 bytes)
UPDATE src/app/app.module.ts (400 bytes)
```

```
import { ServersComponent } from './servers/servers.component';

@NgModule({
  declarations: [
    AppComponent,
    ServersComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

- All files are created and modified as required.

Component Selectors

- There are several types of selectors in Angular, based on CSS selector syntax:
  - Element Selector
    - 'app-hello-world'`
    - Used as a custom HTML element. eg. <app-hello-world></app-hello-world>
  - Attribute Selector
    - `'[appHighlight]'`
    - Selects elements that have a specific attribute. <div appHighlight>
    - Often used for directives, but can also be used for components.
  - Class Selector
    - `'.app-header'`
    - Selects elements by a specific CSS class.
    - This type of selector is also commonly used with directives, but it's possible to use it with components.
  - Wildcard Selector
    - Angular allows for wildcard patterns such as `'*'` to apply to all components if needed, although it’s rare.

Data binding

String Interpolation

- Add a property to the class

```
name: string; // <-- added name property
```

- Initialize the property in the constructor

```
constructor() {
  this.name = 'Felipe'; // set the name
}
```

- Use the value in the Components HTML, the value in the {{ }} should evaluate to a string. It can be a string constant or even a method call returning a string

```
<p>
  Hello {{ name }}
</p>
```

How is UI change detected?

- Angular can detect when component data changes, and then automatically re-render the view to reflect that change.
- Angular at startup time will patch several low-level browser APIs, such as for example addEventListener, which is the browser function used to register all browser events, including click handlers.
- The new version of addEventListener adds more functionality to any event handler: not only the registered callback is called, but Angular is given a chance to run change detection and update the UI.
- This low-level patching of browser APIs is done by a library shipped with Angular called Zone.js.
- The following frequently used browser mechanisms are patched to support change detection:
  - all browser events (click, mouseover, keyup, etc.)
  - setTimeout() and setInterval()
  - Ajax HTTP requests

Property Binding

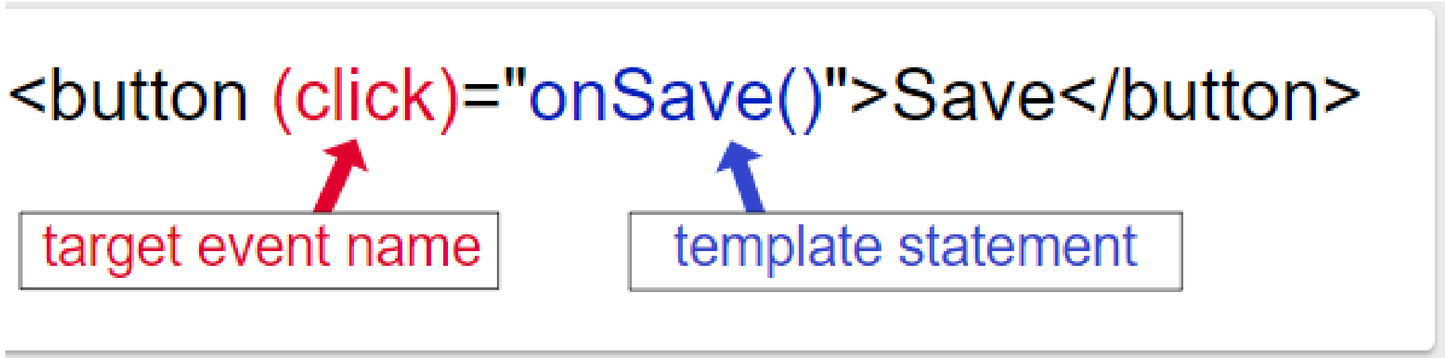
- Property binding in Angular helps you set values for properties of HTML elements or directives.
- Use property binding to do things such as toggle button features, set paths programmatically, and share values between components
- To bind to an element's property, enclose it in square brackets, [], which identifies the property as a target property.
- A target property is the DOM property to which you want to assign a value. It can contain complex expressions.

```
<img alt="item" [src]="itemImageUrl">  // in the html template

// Declare the itemImageUrl property in the class, in this case AppComponent.
itemImageUrl = '../assets/phone.svg';
```

Event Binding

- Event binding lets you listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.
- To bind to an event you use the Angular event binding syntax.
- This syntax consists of a target event name within parentheses to the left of an equal sign, and a quoted template statement to the right. This can be a javascript statement.

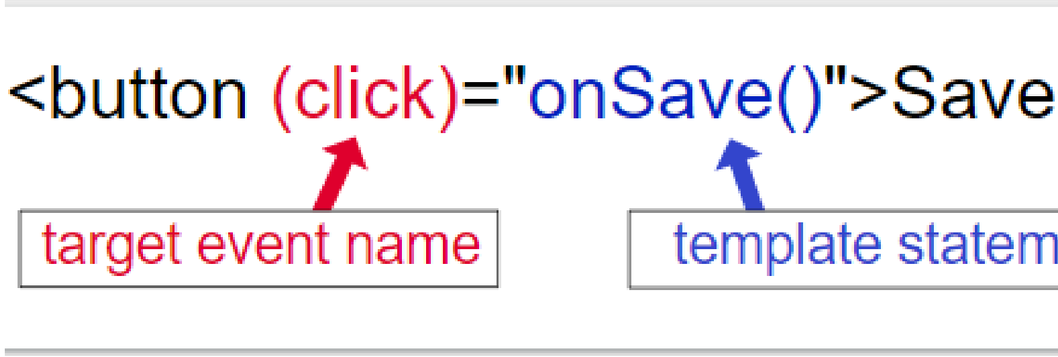
```
<button (click)="onSave()">Save</button>
```



Handling Event

- A common way to handle events is to pass the event object, $event, to the method handling the event.
- The $event object often contains information the method needs, such as a user's name or an image URL.
- The target event determines the shape of the $event object.
- If the target event is a native DOM element event, then $event is a DOM event object, with properties such as target and target.value.

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



Handling Event

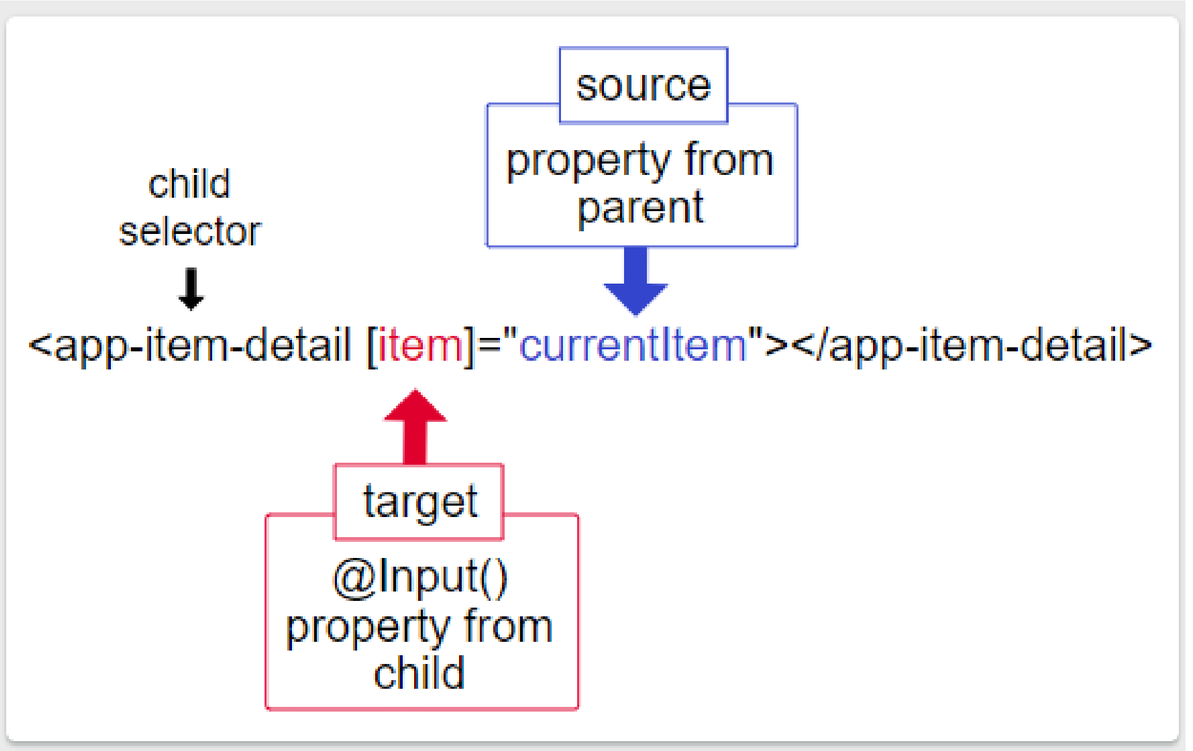
```
<input [value]="name"
        (input)="name=getValue($event)">
```

- The type of $event.target is only EventTarget in the template. In the getValue() method, the target is cast to an HTMLInputElement to allow type-safe access to its value property.

```
getValue(event: Event): string {
    return (event.target as HTMLInputElement).value;

OR
    return (<HTMLInputElement>event.target).value;
}
```

Passing Data to Components

- In Angular 8, to transfer data from the parent component to the child component, we use the @Input decorator.
- The @Input decorator is one of the property decorators in angular.
- Import @Input decorator in child component.
- Place the define @Input in HTML.
- Now, in parent component, need to create a tag with child component selector along with the property binding with the name of define @Input value.
- In parent component, assign a value for defined value.



Passing Data to Components

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

@Component({
    selector: "app-parent",
    templateUrl: "./parent.component.html",
    styleUrls: ["./parent.component.css"]
})

export class ParentComponent {
     constructor() {}
     parentMessage = "Message from the parent to the child";
}
```

```
<div id="parent">
   <p>Parent Component</p>
   <app-child
         [message]='parentMessage'>
   </app-child>
</div>
```

Passing Data to Components

```
import { Component, Input } from "@angular/core";
@Component({
    selector: "app-child",
    templateUrl: "./child.component.html",
    styleUrls: ["./child.component.css"]
})

export class ChildComponent {
   @Input() message = "";
   Constructor() {}
}
```

```
<div id="child">
    <p>Child Component</p>
   <p id="messageFromParent">
       {{ message }}
    </p>
</div>
```

Sending data to parent

- @Output() marks a property in a child component as a doorway through which data can travel from the child to the parent.
- The child component uses the @Output() property to raise an event to notify the parent of the change.
- To raise an event, an @Output() must have the type of EventEmitter, which is a class in @angular/core that you use to emit custom events.

Child Component

```
import { Output, EventEmitter } from '@angular/core';

export class ItemOutputComponent {
    @Output() newItemEvent = new EventEmitter<string>();

     addNewItem(value: string)    {
            this.newItemEvent.emit(value);
     }
}
```

Child Template

```
<label for="item-input">Add an item:</label>
<input type="text" id="item-input" #newItem>
<button type="button"
       (click)="addNewItem(newItem.value)">Add to parent's
                                                  list

</button>
```

Sending data to parent

- The addItem() method takes an argument in the form of a string and then adds that string to the items array.
- In the parent's template, bind the parent's method to the child's event.
- Put the child selector, here <app-item-output>, within the parent component's template, app.component.html.
- The event binding, (newItemEvent)='addItem($event)', connects the event in the child, newItemEvent, to the method in the parent, addItem().
- The $event contains the data that the user types into the <input> in the child template UI.

Parent Component

```
export class AppComponent {
  items = ['item1', 'item2', 'item3', 'item4'];

  addItem(newItem: string) {
      this.items.push(newItem);
  }
}
```

Parent Template

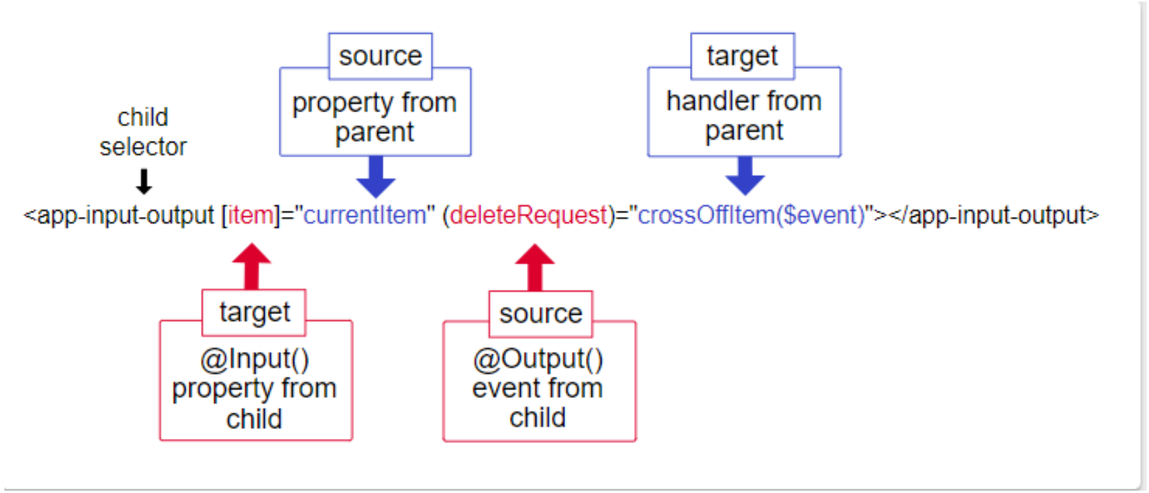
```
<app-item-output
      (newItemEvent)="addItem($event)">
</app-item-output>
```

Using input() and output() together

- The target, item, which is an @Input() property in the child component class, receives its value from the parent's property, currentItem.
- When you click delete, the child component raises an event, deleteRequest, which is the argument for the parent's crossOffItem() method.
- The child selector is <app-input-output> with item and deleteRequest being @Input() and @Output() properties in the child component class.
- The property currentItem and the method crossOffItem() are both in the parent component class.

Parent Template

```
<app-input-output
      [item]="currentItem"
      (deleteRequest)="crossOffItem($event)">
</app-input-output>
```



Two way binding

- Two-way binding gives components in your application a way to share data.
- Use two-way binding to listen for events and update values simultaneously between parent and child components.
- Angular's two-way binding syntax is a combination of square brackets and parentheses, [()].
- The [()] syntax combines the brackets of property binding, [], with the parentheses of event binding, (), as follows.
- For Two-Way-Binding to work, you need to enable the ngModel directive. This is done by adding the FormsModule to the imports[] array in the AppModule.
- You then also need to add the import from @angular/forms in the app.module.ts file:
    - import { FormsModule } from '@angular/forms';

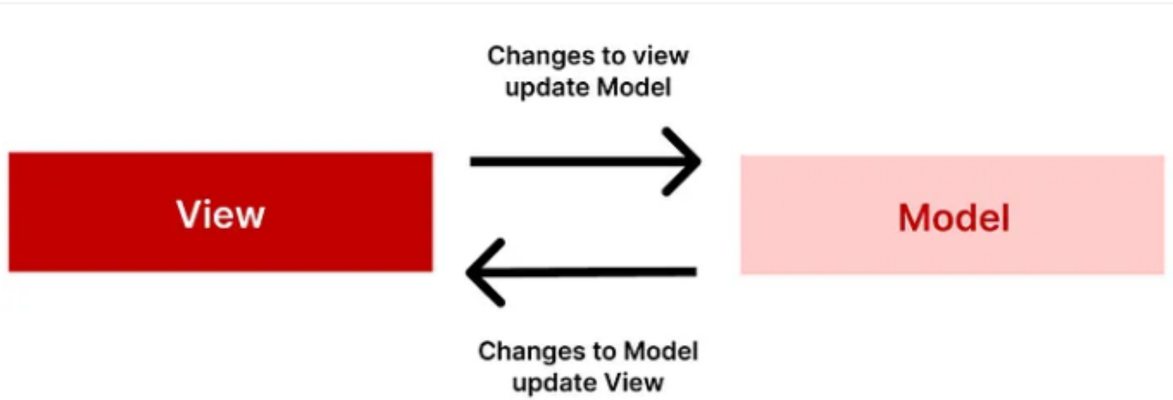
ngModel directive

- The ngmodel directive binds the value of HTML controls (input, select, textarea) to application data.
- With the ng-model directive you can bind the value of an input field to a variable created in Angular.
- The binding goes both ways. If the user changes the value inside the input field, the Angular property will also change its value.

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

@NgModule({
  declarations: [AppComponent, ServersComponent],
  imports: [BrowserModule, FormsModule],
  providers: [],
  bootstrap: [AppComponent],
})
```

Two way binding

```
import { Component } from '@angular/core';
@Component({
      selector: 'app-root',
      template: `<input [(ngModel)]="value">
                        <span>{{value}}</span>`,
})

export class AppComponent {
  value: string = 'Joe';
}
```



Two way binding

- Angular's two-way binding syntax is a combination of square brackets and parentheses, [()].
- The [()] syntax combines the brackets of property binding, [], with the parentheses of event binding, ()
- Clicking the buttons updates the AppComponent.fontSizePx.
- The revised AppComponent.fontSizePx value updates the style binding, which makes the displayed text bigger or smaller.

Parent Template

```
<app-sizer
        app-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></app-sizer>
</app-sizer>
<div [style.font-size.px]="fontSizePx">Resizable Text</div>
```

Parent Component

```
fontSizePx = 16;
```

Two way binding

- The sizerComponent template has two buttons that each bind the click event to the inc() and dec() methods.
- When the user clicks one of the buttons, the sizerComponent calls the corresponding method.
- Both methods, inc() and dec(), call the resize() method with a +1 or -1, which in turn raises the sizeChange event with the new size value.

Child Template

```
<div>
<button type="button" (click)="dec()" title="smaller">-
</button>
<button type="button" (click)="inc()" title="bigger">+
</button>
<span [style.font-size.px]="size">FontSize: {{size}}px</span>
</div>
```

Child Component

```
export class SizerComponent {
    @Input() size!: number | string;
    @Output() sizeChange = new EventEmitter<number>();
    dec() { this.resize(-1); }
    inc() { this.resize(+1); }

    resize(delta: number) {
        this.size = Math.min(40, Math.max(8, +this.size +
                                          delta));
        this.sizeChange.emit(this.size);
        }
}
```

- The ! is the non-null assertion operator. It is a way to tell the compiler "this expression cannot be null or undefined here, so don't complain about the possibility of it being null or undefined

Template References

- Template variables help you use data from one part of a template in another part of the template.
- Use template variables to perform tasks such as respond to user input or finely tune your application's forms.
- A template variable can refer to the following:
  - a DOM element within a template
  - a directive or component
  - a TemplateRef from an ng-template
  - a web component
- Template reference variables will remain null/undefined until the view portion of the component has finished initiating. Make sure that you only attempt to use these variables within the ngAfterViewInit lifecycle hook or after this hook completes!

@ViewChild decorator

- To create a template reference variable, locate the HTML element that you want to reference and then tag it like so: #myVarName.
- a template reference variable is created that references the with id test-div.

```
<div id="test-div" #myTestDiv>2    </div>
```

Accessing Template Elements using the ViewChild decorator

- Within your Angular component, use the ViewChild decorator that Angular provides in order to bind to the previously created template reference variable.
- For HTML elements, use the ViewChild decorator to create a new ElementRef as shown below:

```
@ViewChild('myTestDiv') myTestDiv: ElementRef;
```

- This ElementRef gives your component direct access to the underlying HTML element when you use its nativeElement field like this

```
const divEl: HTMLDivElement = this.myTestDiv.nativeElement;
```

Template Refs for Child Components

- create a template reference variable on a child component like this:

```
<app-test-component #myTestComp></app-test-component>
```

- With your child component referenced, you can gain access to it in your Angular component class like this:

```
@ViewChild('myTestComp') myTestComp: TestComponent;
```

- With your child component successfully captured, you can access it like this

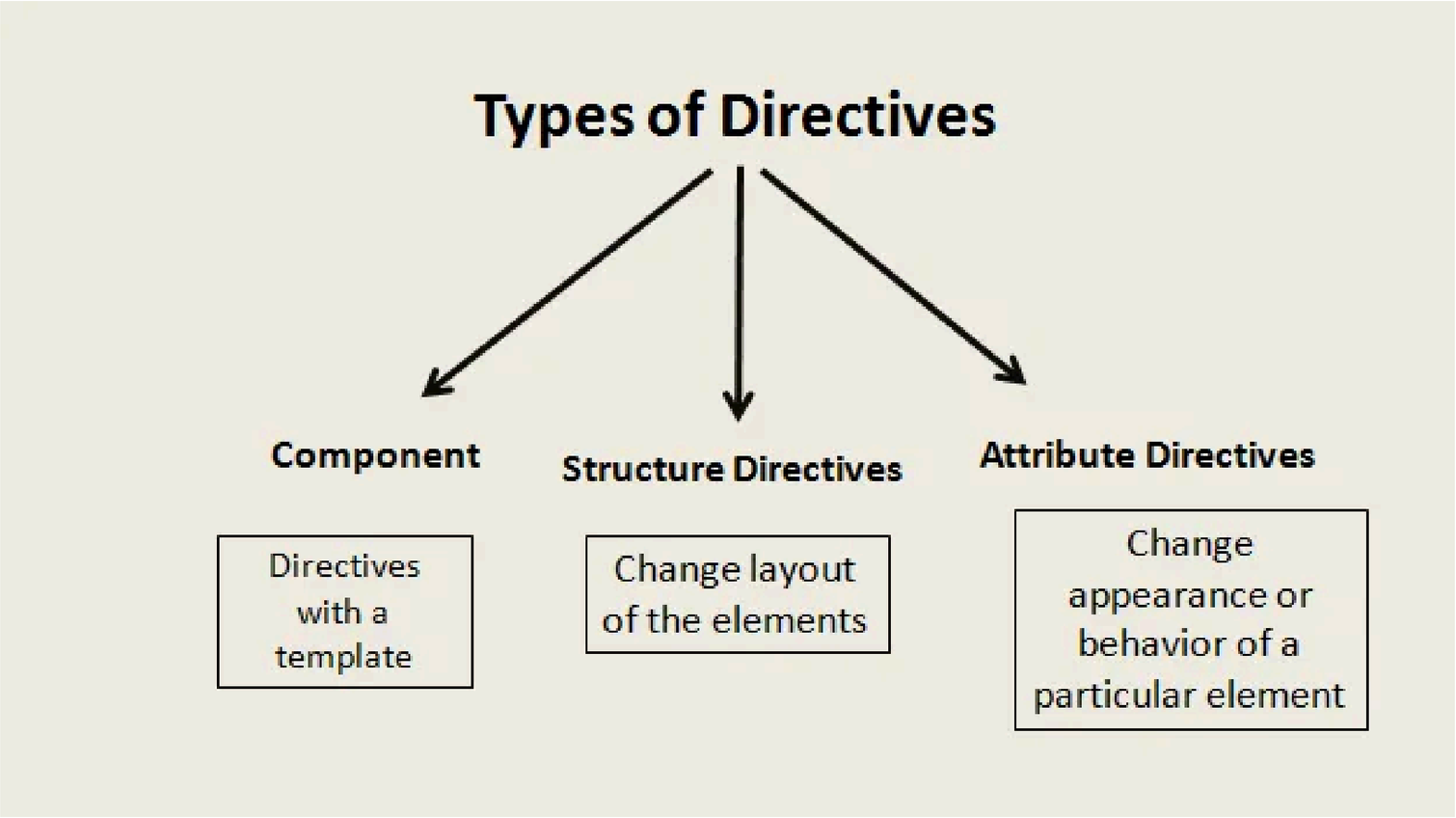
```
ngAfterViewInit(): void {
    // We can access the TestComponent now that this portion of the view tree has been initiated.
    this.myTestComp.saveTheWorld();
}
```

Directives


website=True,
gs):
pes']
[]
s == 'local_bachelor_program_hsc':
domain = [('course_id.is_local_bachelor_prog
elif types == 'local_bachelor_program_a_level':
domain = [('course_id.is_local_bachelor_prog
elif types == 'local_bachelor_program_diploma':
domain = [('course_id.is_local_bachelor_prog
elif types == 'local_masters_program_bachelor':
domain = [('course_id.is_local_masters_progr
elif types == 'international_bachelor_program':
domain = [('course_id.is_international_bache
elif types == 'international_masters_program':
domain = [('course_id.is_international_maste
domain.append(('state', '=', 'applicati
admission_register_list = http
if admission_register
for progr

Directives in Angular

- Directives are custom HTML attributes which tell angular to change the style or behavior of the Dom elements.



Component Directives

- Directives must be declared in Angular Modules in the same manner as components.
- These form the main class having details of how the component should be processed,used at run-time. directive in Angular is a reusable component
- The other two directive types, attribute and structural, do not have templates.
- In the next example, there is a class called ChangeTextDirective and a constructor, which takes the element of type ElementRef, which is mandatory. The element has all the details to which the Change Directive is applied.

Component Directives

Example: change-text. directive

```
import { Directive } from '@angular/core';
@Directive({
   selector: '[changeText]'
})

export class ChangeTextDirective {
   constructor() { }
}
```

app.component.html

```
<div style="text-align:center">
   <span changeText >Welcome to {{title}}.</span>
</div>
```

change-text.directive.ts

```
import { Directive, ElementRef} from '@angular/core';
@Directive({
   selector: '[changeText]'
})

export class ChangeTextDirective {
   constructor(Element: ElementRef) {
      console.log(Element);
      Element.nativeElement.innerText="Text is changed by changeText
Directive. ";
   }
}
```

Structural Directives

- Structural Directives are responsible for changing the structure of the DOM. They work by adding or removing the elements from the DOM, unlike Attribute Directives which just change the element’s appearance and behavior.
- You can easily differentiate between the Structural and Attribute Directive by looking at the syntax. The Structural Directive’s name always starts with an asterisk(*) prefix, whereas Attribute Directive does not contain any prefix.
- 3 most popular structural directives:
  - **NgIf**,
  - **NgFor**,
  - **NgSwitch**.

*ngif

- The ngIf directive is used when you want to display or hide an element based on a condition. The condition is determined by the result of the expression that you pass into the directive

```
<div *ngIf="false"></div>      <!-- never displayed -->
<div *ngIf="a > b"></div>      <!-- displayed if a is more than b -->
<div *ngIf="str == 'yes'">     </div> <!-- displayed if str is the string "yes" -->
<div *ngIf="myFunc()">         </div> <!-- displayed if myFunc returns truthy -->
```

*ngif else

- Create an <ng-template #template-reference> with a template reference
- Between opening and closing <ng-template> tags, define the element tags that you wish to display in the else
- Write the if - else as follows
- <p *ngIf="servercreated; else noServer">The server name is {{servername}}
- <ng-template #noServer>
- No Server Created
- </ng-template>

ng-template

- ng-template is a template element used to define sections of code to render later or conditionally.
- ng-template defines a template that is not rendered by default. It's a way to define a block of HTML or components that will be conditionally or dynamically inserted into the DOM.
- However, it does not create an actual DOM element unless used with other directives (like ngIf, ngFor, or ngTemplateOutlet).
- It is often used with Angular directives like ngIf, ngFor, and ngSwitch.

```
<ng-template ngFor let-item [ngForOf]="items" let-i="index">
  <div>
    <p>Index: {{ i }}</p>
    <p>Item: {{ item }}</p>
  </div>
</ng-template>
```

ngSwitch

- The idea behind this directive is the same: allow a single evaluation of an expression, and then display nested elements based on the value that resulted from that evaluation.
- Once we have the result then we can:
  - Describe the known results, using the ngSwitchCase directive
  - Handle all the other unknown cases with ngSwitchDefault

```
<div class="container" [ngSwitch]="myVar">
<div *ngSwitchCase="'A'">Var is A</div>
<div *ngSwitchCase="'B'">Var is B</div>
<div *ngSwitchDefault>Var is something else</div>
</div>
```

ngSwitch

- Having the ngSwitchDefault element is optional. If we leave it out, nothing will be rendered when myVar fails to match any of the expected values.
- You can also declare the same *ngSwitchCase value for different elements, so you’re not limited to matching only a single time.

```
<div class="ui raised segment">
    <ul [ngSwitch]="choice">
    <li *ngSwitchCase="1">First choice</li>
    <li *ngSwitchCase="2">Second choice</li>
    <li *ngSwitchCase="3">Third choice</li>
    <li *ngSwitchCase="4">Fourth choice</li>
    <li *ngSwitchCase="2">Second choice, again</li>
    <li *ngSwitchDefault>Default choice</li>
    </ul>
</div>
```

Attribute Directives - ngStyle

- With the ngStyle directive, you can set a given DOM element CSS properties from Angular expressions.

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
    Uses fixed white text on blue background
</div>
```

- Notice that in the ng-style specification we have single quotes around background-color but not around color.
- Why is that?
    - The argument to ng-style is a JavaScript object and color is a valid key, without quotes.
    - With background-color, however, the dash character isn't allowed in an object key, unless it's a string so we have to quote it.

Attribute Directives - ngStyle

- ngStyle with object property from variable

```
<div>
    <span [ngStyle]="{color: color}">
        {{ color }} text
    </span>
</div>
```

- style from variable

```
<div [style.background-color]="color"
    style="color: white;">
    {{ color }} background
</div>
```

ngClass

- The NgClass directive, represented by a ngClass attribute in your HTML template, allows you to dynamically set and change the CSS classes for a given DOM element.
- The first way to use this directive is by passing in an object literal.
- The object is expected to have the keys as the class names and the values should be a truthy/falsy value to indicate whether the class should be applied or not.

```
.bordered {
    border: 1px dashed black;
    background-color: #eee;
}
```

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

ngClass

- To make the ngClass directive dynamic we add a variable as the value for the object value,

```
.bordered {
    border: 1px dashed black;
    background-color: #eee;
}
```

```
<div [ngClass]="{bordered: isBordered}">
        Using object literal. Border {{ isBordered ? "ON" : "OFF" }}
</div>
```

ngFor

- The role of this directive is to repeat a given DOM element (or a collection of DOM elements) and pass an element of the array on each iteration.
- The syntax is *ngFor="let item of items".
  - The let item syntax specifies a (template) variable that’s receiving each element of the items array;
  - The items is the collection of items from your controller.

```
<h4 class="ui horizontal divider header">
    Simple list of strings </h4>

<div class="ui list" *ngFor="let c of cities">
      <div class="item">{{ c }}</div>
</div>
```

```
this.cities = ['Miami', 'Sao Paulo', 'New York'];
```

ngFor - Getting an index

- There are times that we need the index of each item when we're iterating an array.
- We can get the index by appending the syntax let idx = index to the value of our ngFor directive, separated by a semi-colon.
- When we do this, ng2 will assign the current index into the variable we provide

```
this.cities = ['Miami', 'Sao Paulo', 'New York'];

<div class="ui list" *ngFor="let c of cities; let num = index">
    <div class="item">{{ num+1 }} - {{ c }}</div>
</div>
```

Custom Attribute Directive

- An attribute directive in Angular is used to change the appearance or behavior of a DOM element without altering its structure.
- Unlike structural directives, attribute directives simply change the attributes or styles of an element.

Custom Attribute Directive

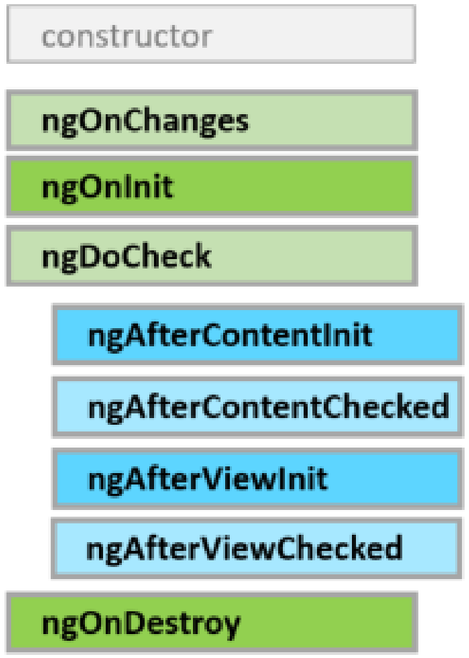
- @Directive({ selector: '[appHighlight]' }): This marks the class as a directive. The selector specifies that this directive will be used as an attribute (appHighlight) on an HTML element.
- ElementRef: A reference to the DOM element where the directive is applied. We use it to manipulate the element's styles.
- @HostListener('mouseenter') and @HostListener('mouseleave'): These decorators listen to mouse events. When the user hovers over the element, we change its background color; when they stop hovering, we reset the color.
- highlight(color: string | null): A helper method to apply the background color to the element.

```
// highlight.directive.ts
import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input() defaultColor: string = 'yellow'; // Default highlight color
  @Input() highlightColor: string = 'lightblue'; // Color on hover

  constructor(private el: ElementRef) {}

  // Change background color when the mouse enters the element
  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || this.defaultColor);
  }

  // Reset background color when the mouse leaves the element
  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string | null) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

Content Projection with ng-content

- Content projection in Angular allows you to insert content into a component from the outside, making components more reusable and flexible.
- This is achieved by using <ng-content> inside the component's template, where projected content from the parent will be inserted.
- There are three types of content projection in Angular:
  - Single-slot content projection (Basic content projection)
    - <ng-content></ng-content>) projects content from the parent into one location.
  - Multi-slot content projection
    - allows projecting different parts of the parent content into multiple locations using CSS selectors (<ng-content select="..."></ng-content>).
  - Conditional content projection (using ng-content with selectors)
    - allows displaying default content when no content is projected or handling different content conditions using Angular directives and lifecycle hooks.

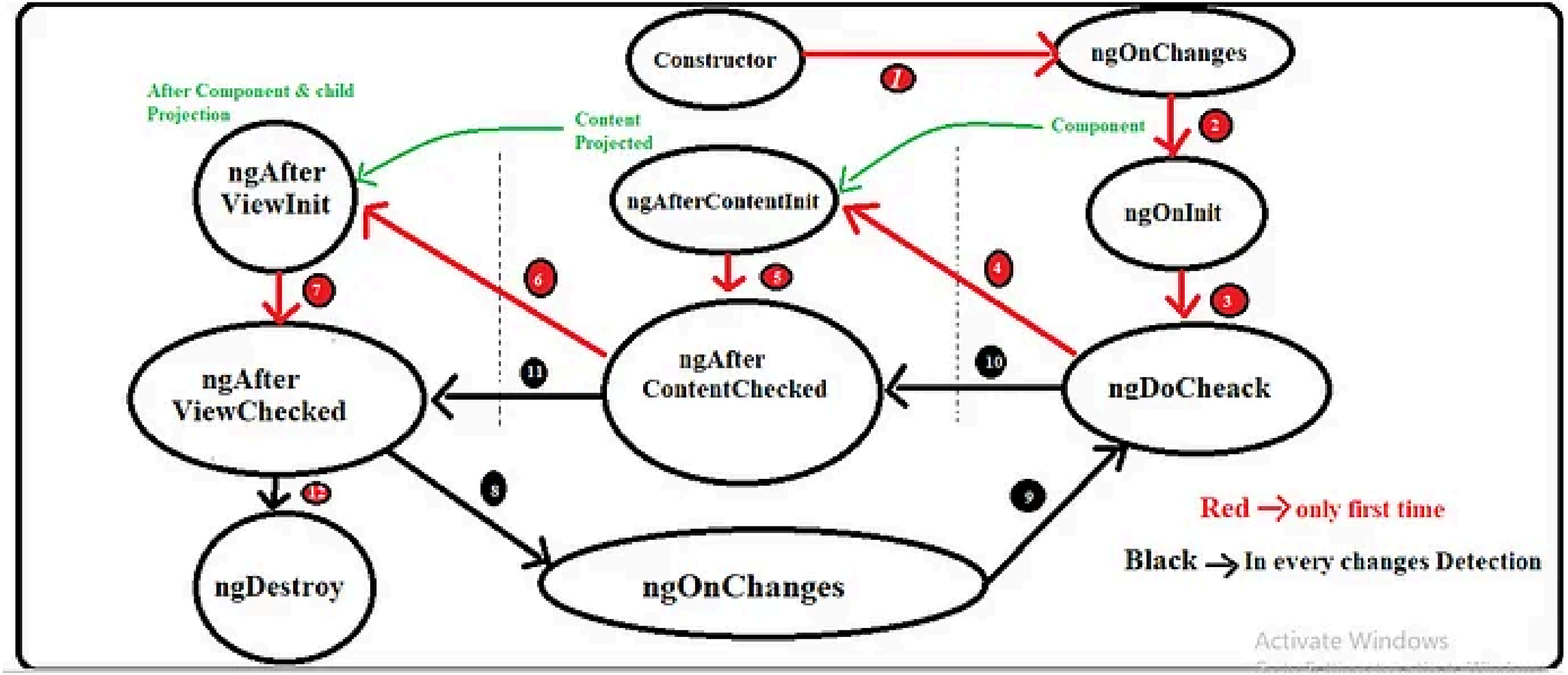
Angular Components Lifecycle hooks



Lifecycle of an Angular component

- A component instance in Angular has a lifecycle that starts when it instantiates the component class and renders the component view with its child views.
- The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed.
- The lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM.
- Directives have a similar lifecycle, as Angular creates, updates, and destroys instances during execution.
- Angular applications can use lifecycle hook methods to tap into key events in the lifecycle of a component or directive to initialize new instances, initiate change detection when needed, respond to updates during change detection, and clean up before the deletion of instances.

Lifecycle of an Angular component



Lifecycle of an Angular component

- Angular calls these hook methods in the following order:
  - **ngOnChanges**: Once when the component is created and always when an input/output binding value changes.
  - **ngOnInit**: Only once when the component is created, after the first ngOnChanges.
  - **ngDoCheck**: Called after ngOnChanges or ngOnInit, Developer's custom change detection.
  - **ngAfterContentInit**: Called only once, After component content initialized. @ViewChild references are valid
  - **ngAfterContentChecked**: Once after ngAfterContentInit and then every time ngDoCheck After every check of component content.
  - **ngAfterViewInit**: Once after ngAfterContentChecked, After a component's views are initialized.
  - **ngAfterViewChecked**: Once after ngAfterViewInit and then after ngAfterContentChecked, After every check of a component's views.
  - **ngOnDestroy**: Just before the component/directive is destroyed.

Implementing a lifecycle hook

ngOnInit :-

- When first time the data-bound properties are displayed and here we set the input property values.
- This event gets its call only after ngOnChanges event and after the constructor & It is called only for once.
- With this hook, you can initialize logic to your component.

ngDoCheck :-

- This is for the detection and to act on changes that Angular can’t or won’t detect on its(or Called whenever angular change detection run)
- This hook comes on demand instantly after ngOnInit, and this hook has its duty of execution even if there is no change in the property of a component.

ngAfterContentInit :-

- This method is implemented as soon as Angular makes any content projection within a component view.
- This is called in response after Angular projects external content into the component’s view.
- External child components can be included by Angular using this method within the <ng-content> </ng-content> tag. In the total lifecycle of a component, this hook gets call only for once.

Implementing a lifecycle hook

ngAfterContentChecked :-

- This is called in response after Angular checks the content projected into the component (This is the changes detection check for the contents projected).
- It gets its call after ngAftercontentInit and also gets executed after every execution ngDoCheck. It plays a big role in the initialization of the child component.

ngAfterViewInit :-

- This lifecycle method gets its call after ngAfterContentChecked and finds its use only on components.
- This is very much similar to ngAfterContentInit, and it gets invoked only after all the component view and its child view.

ngAfterViewChecked :-

- This Once the default changes detection run and content projected change detection run then this event fires .
- When something is awaited from the child component, this component can be helpful.
- It gets call Called after the ngAfterViewInit and every subsequent ngAfterContentChecked.

ngOnDestroy :-

- This is the clean-up phase just use before Angular destroys the directive/component (On the destruction of a component, ngOnDestroy is invoked by the Angular).

Angular Forms

Forms in Angular

- Angular provides two different approaches to handling user input through forms: reactive and template-driven.
  - capture user input events from the view
  - validate the user input
  - create a form model and data model to update
  - and provide a way to track changes.
- Reactive forms
  - Provide direct, explicit access to the underlying form's object model.
  - They are more robust, scalable, reusable, and testable.
- Template-driven forms
  - Rely on directives in the template to create and manipulate the underlying object model.
  - They are useful for adding a simple form to an app
  - They don't scale as well as reactive forms.

Form Foundation Classes

- FormControl
  - Tracks the value and validation status of an individual form control.
- FormGroup
  - Tracks the same values and status for a collection of form controls.
- FormArray
  - Tracks the same values and status for an array of form controls.
- ControlValueAccessor
  - Creates a bridge between Angular FormControl instances and built-in DOM elements.

Template Driven Forms

- HTML form controls (such as <input> or <select>) in the component template can be placed in form-controls, and bind them to data model properties, using directives like ngModel.
- With template-driven forms, you don’t create Angular form control objects.
- They are created by Angular directives using information from your data bindings.
- You don’t have to push and pull data values around because Angular handles that for you through the ngModel directive.
- Angular updates the mutable data model according to user changes as they occur.
- directives are; ngModel, required, maxlength.
- In template-driven forms we specify directives to bind our models, values, validations and more,
- So we are actually letting the template do all the work on the background.

Template Driven Forms

- Steps to create template driven forms
  - Add FormsModule to app.module.ts.
  - Create a class for the User model.
  - Create initial components and layout for the signup form.
  - Use Angular form directives like ngModel, ngModelGroup, and ngForm.
  - Add validation using built-in validators.
  - Display validation errors meaningfully.
  - Handle form submission using ngSubmit.

Submiting the Form

- In the form tag add a template reference to ngForm
- Bind to onSubmit event use the reference

```
<form class="myForm" (ngSubmit)="onSubmit(f)" #f="ngForm">
```

- In the Component TS file add the event handler for submit

```
onSubmit(form: NgForm) {
    console.log('Submitted', form.form);
}
```

Usage of ngModel

```
<input type="text" id="phone" ngModel name="phone"
#phone="ngModel"
placeholder="Mobile">

<pre>{{ phone.value }}</pre>

<input type="text" ngModel name="name" required />
```

- The angular ngmodel does not save a value for a variable but rather a reference to it.

- Use ngModel to create two-way data bindings for reading and writing input-control values

Registering the Controls

- Bind form controls to data properties using the ngModel directive

```
<input type="text" ngModel name="name" required />
```

- Perform various steps depending on Form and element states

```
▼ <body _ngcontent-ng-c4195326977>
  ▼ <form _ngcontent-ng-c4195326977 novalidate class="myForm ng-untouched ng-pristine
    ng-invalid">
```

- Enable and Disable elements based on the state

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

Add Validations to the Form

- Add Validation to the Form elements

```
<p>
 <label>  Email
     <input   type="email"
              ngModel
              name="email"
              required
              email
             #email="ngModel"/>
 </label>
</p>
<p *ngIf="!email.valid && email.touched">Email entered is incorrect</p>
```

Built-in Validations

- Required - The required validator returns true only if the form control has non-empty value entered. Let us add this validator to all fields
- Minlength - This Validator requires the control value must not have less number of characters than the value specified in the validator.

```
<input name="userName" [ngModel]="user.userName" minlength="5" #uname="ngModel">
```

- Maxlength - This Validator requires that the number of characters must not exceed the value of the attribute.
- Pattern - This Validator requires that the control value must match the regex pattern provided in the attribute. For example, the pattern ^[a-zA-Z]+$ ensures that the only letters are allowed (even spaces are not allowed).
- Email - This Validator requires that the control value must be a valid email address.

Setting Default Values

- To set default values for an element- Use property binding on ngModel to bind to a variable in the component.

```
<label
   >Name
   <input type="text" [ngModel]="defaultName" name="name" required />
</label>
```

Grouping Elements with ngModelGroup

- Creates and binds a FormGroup instance to a DOM element.

```
<fieldset ngModelGroup="connect">
   <p>
    <label
     >Phone
     <input type="tel" ngModel name="phone" required />
    </label>
   </p>
</fieldset>
```

Setting and Patching form values

- In our example we have a template reference, which is an instance of #f="ngForm"
- We can get the reference to the #f in the app.component.ts, using the viewchild
- Once we have the reference, we can use the setValue method of the ngForm to set the initial value
- setValue will set the value of all components.

```
this.form_variable.setValue(object);
```

- To set only a single element value, Use patchValue.

- You can reset the form to empty value using the reset or resetForm method of the ngForm. These also resets the form status like dirty, valid, pristine & touched, etc

```
this.form_variable.reset();
```

Reactive Forms

Elite
IT Academy

Creating Reactive Forms

- With reactive forms, you build your own representation of a form in the component class.
- some of the advantages of reactive forms:
  - Using custom validators
  - Changing validation dynamically
  - Dynamically adding form fields
- Setting up the reactive form
  - In the Application module file, import ReactiveFormsModule instead of FormsModule
  - In the component controller file import FormsGroup from @angular/forms
- Create a FormGroup variable in the controller i.e myform:FormGroup

Using the FormGroup

- The FormGroup takes part in creating reactive form.
- FormGroup is used with FormControl and FormArray.
- The role of FormGroup is to track the value and validation state of form control.
- Create an instance of FormGroup with the instances of FormControl.

```
userForm = new FormGroup({
    name: new FormControl(),
    age: new FormControl('20')
    }
);
```

```
<form [formGroup]="userForm" (ngSubmit)="onFormSubmit()">
  Name: <input formControlName="name"  placeholder="Enter Name">
  Age: <input formControlName="age"  placeholder="Enter Age">
  <button type="submit">Submit</button>
</form>
```

- Using setValue()
  - this.userForm.setValue({name: 'Mahesh', age: '20' });
- Using patchValue()
  - this.userForm.patchValue({name: 'Mahesh'});

Using the FormGroup

- FormGroup Get Value
  - Suppose userForm is the instance of FormGroup - To get the value of form control named as name after form submit,

```
this.userForm.get('name').value
```

- If we want to get all values of the form then we can write code as given below
  - this.userForm.value
- FormGroup reset() - To reset the form we need to call reset() method on FormGroup instance.
  - this.userForm.reset();

Initializing the Reactive Form

- Implement ngOnInit and initialize the form object as follows

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';

export class AppComponent implements OnInit {
  myForm: FormGroup = new FormGroup({
      name: new FormControl('Sammy'),
      email: new FormControl(''),
      message: new FormControl('')
    });
  }
```

```
  onSubmit() {
      console.log(this.myForm);
  }
```

Reactive Form View

- User the formGroup directive and formControl name to sync with the model, and ngSubmit

```
<form [formGroup]="myForm" (ngSubmit)="onSubmit()">
 <div>    <label>    Name:
    <input formControlName="name" placeholder="Your name">
  </label>
 </div>
 <div>    <label>   Email:
    <input formControlName="email" placeholder="Your email">
  </label>
 </div>
 <div>    <label>      Message:
    <input formControlName="message" placeholder="Your message">
  </label>
 </div>  <button type="submit">Send</button>
</form>
```

Adding Validations to a Form

- You can use the pre-defined validators as follows

```
name: ['Sammy', Validators.required],
email: ['', [Validators.required, Validators.email]],
message: ['', [Validators.required, Validators.minLength(15)]],
```

- To get access to the elements state values like .valid or .touched we can access the element data via the formGroup object and using the get method.

```
<span
    *ngIf="!myForm.get('name').valid && myForm.get('name').touched">
    Please enter a Valid Username
</span>
```

Creating a nested FormGroup

- To have nested form groups, Implement the nesting in the FormGroup Object

```
myForm: FormGroup;
ngOnInit() {
      this.myForm = new FormGroup({
          'userData: new FormGroup({
                  name: new FormControl('Sammy'),
                  email: new FormControl('')
           }),
           message: new FormControl('')
        });
}
```

Synchronize with the View

- The same structure needs to be implemented in the View

```
<form [formGroup]="myForm" (ngSubmit)="onSubmit(myForm)">
 <div formGroupName="userData">
    <div>
       <label>    Name:
           <input formControlName="name" placeholder="Your name">
    </label>
 </div>
 <div>    <label>   Email:
    <input formControlName="email" placeholder="Your email">
  </label>
 </div>
  </div>
   .... remaining view
</form>
```

Synchronize with the View

- Any formGroup.get methods need to be also changed if required.

```
<span
    *ngIf="!myForm.get('userData.name').valid && myForm.get('userData.name').touched">
    Please enter a Valid Username
</span>
```

Using the Form Array

- A Form Array can be used to hold a collection of FormControls, FormGroups or FormArray instances
- The FormArrayName syncs a nested FormArray to a DOM element.
- It is used under formGroup in HTML template.
- It accepts the string name of FormArray registered in the FormGroup created in the TypeScript.
- The selector of FormArrayName is formArrayName.

```
<div formArrayName="classmates">
        <div *ngFor="let cm of classmates.controls; index as i">
                <input [formControlName]="i">
        </div>
</div>
```

Methods of Form Array

- at(): Returns the AbstractControl instance for the given index. The AbstractControl is the base class for FormControl, FormGroup and FormArray classes.
- push(): Inserts the new AbstractControl at the end of array.
- insert(): Inserts a new AbstractControl at the given index in the array.
- removeAt(): Removes the control at the given index in the array.
- setControl(): Sets a new control replacing the old one.
- setValue(): Sets the value to every control of this FormArray.
- patchValue(): Patches the value to available controls starting from index 0.
- reset(): Resets the values.
- getRawValue(): The aggregate value of the array.
- clear(): Removes all controls in this FormArray.

Observables

Elite
IT Academy

RxJs

- Reactive Extensions for JavaScript (RxJS) is a reactive streams library that allows you to work with asynchronous data streams.
- RxJS can be used both in the browser or in the server-side using Node.js.
  - Asynchronous, in JavaScript means we can call a function and register a callback to be notified when results are available, so we can continue with execution and avoid the Web Page from being unresponsive.
  - This is used for ajax calls, DOM-events, Promises, WebWorkers and WebSockets.
  - Data, raw information in the form of JavaScript data types as: Number, String, Objects (Arrays, Sets, Maps).
  - Streams, sequences of data made available over time. As an example, opposed to Arrays you don’t need all the information to be present in order to start using them.

RxJs

- In RxJS, you represent asynchronous data streams using observable sequences or also just called observables. Observables are very flexible and can be used using push or pull patterns.
  - When using the push pattern, we subscribe to the source stream and react to new data as soon as is made available (emitted).
  - When using the pull pattern, we are using the same operations but synchronously. This happens when using Arrays, Generators or Iterables.

RxJs

- Make sure you install RxJS by running
  - npm install --save rxjs
- In addition, also install the rxjs-compat package:
  - npm install --save rxjs-compat

```
import { interval, Subscription } from 'rxjs';
export class ObservableComponent implements OnInit, OnDestroy {
 private mySubscription?: Subscription;

 ngOnInit(): void {
  this.mySubscription = interval(1000).subscribe((count) => {
   console.log(count);
  });
 }

 ngOnDestroy(): void {
  this.mySubscription?.unsubscribe()
 }

}
```

RxJs

- Make sure you install RxJS by running
  - npm install --save rxjs
- In addition, also install the rxjs-compat package:
  - npm install --save rxjs-compat

```
import { interval, Subscription } from 'rxjs';
export class ObservableComponent implements OnInit, OnDestroy {
 private mySubscription?: Subscription;

 ngOnInit(): void {
  this.mySubscription = interval(1000).subscribe((count) => {
   console.log(count);
  });
 }

 ngOnDestroy(): void {
  this.mySubscription?.unsubscribe()
 }

}
```

Building a custom observable

```
import { interval, Subscription, Observable } from 'rxjs';

export class ObservableComponent implements OnInit, OnDestroy {
  private myObservable?: Subscription;

 ngOnInit(): void {
  const customObserverable = Observable.create((observer: any) => {
      let count = 0;
      setInterval(() => {   observer.next(count);  count++; }, 1000);
  });

  this.myObservable = customObserverable.subscribe((data: any) => {
   console.log(data);
  });
 }
}
```

- observer.next - publishes its value using the next method

Errors and Completion

```
const customObserverable = Observable.create((observer: any) => {
  let count = 0;
  setInterval(() => {
   observer.next(count);
   count++;
   if (count > 3) {
    observer.complete();
   }
   if (count > 5) {
    observer.error(new Error('Count exceeded value'));
   }
  }, 1000);
 });
```

```
this.myObservable = customObserverable.subscribe(
  (data: any) => {
   console.log(data);
  },
  (error: any) => {
   console.log(error);
  },
  () => {
   console.log('Complete');
  }
 );
```

Operators

```
import { map, filter } from 'rxjs/operators';

this.myObservable = customObserverable
   .pipe(
    filter((data: any) => {
     return data % 2 == 0;
    }),
    map((data) => {
     return 'Round ' + data;
    })
   )
.subscribe( ...)
```

- Types of Operators
  - Creation Operators
  - Join Creation Operators
  - Transformation Operators
  - Filtering Operators
  - Join Operators
  - Multicasting Operators
  - Utility Operators
  - Conditional and Boolean Operators
  - Mathematical and Aggregate Operators

Operators - Creation

- defer
- empty
- from
- fromEvent
- fromEventPattern
- generate
- interval
- of
- range

```
import { of } from 'rxjs';
//emits any number of provided values in sequence
const source = of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
const subscribe = source.subscribe(val => console.log(val));

import { range } from 'rxjs';
//emit 1-10 in sequence
const source = range(1, 10);
const example = source.subscribe(val => console.log(val));
```

Promise in Angular

- A promise in angular is a JavaScript object which produces a value after an async operation is executed successfully.
- If the operation does not execute successfully, then it generates an error.
- A promise in angular is a class-based object, it is created using the new keyword and its constructor function.
- It contains different types of methods that gives our objects some power.
- A promise in Angular is defined by passing a callback function also known as the executor function or executor code as an argument to the Promise constructor. The executor function usually takes two arguments: resolve and reject.
- An async operation is defined inside the executor function and intended result or error if any occurred is handled by the resolve and reject handlers respectively.
- Promise objects have a .then and a .catch methods which handle fulfilled results and errors if any occurred.
- They receive the outcome of async operations as data or error from the resolve and reject handler in the promise constructor.

Promise in Angular

```
const myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('this is a promise');
      }, 300);
    });

myPromise.then((value) => {
      console.log(value);    // expected output: "this is a promise"
});

console.log(promise1); // expected output: [object Promise]
```

Custom Validators

- Angular's @angular/forms package comes with a Validators class that supports useful built-in validators like required, minLength, maxLength, and pattern.
- However, there may be form fields that require more complex or custom rules for validation. In those situations, you can use a custom validator.
- When using Reactive Forms in Angular, you define custom validators with functions.

```
import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms';

export function ValidateURL(): ValidatorFn {
 return (control: AbstractControl): ValidationErrors | null => {
  console.log('displaying error: ', control.value);
  const check = !control.value.startsWith('https');
  return check ? { invalidUrl: true } : null;
 };
}
```

Using the Custom Validator

- Create the form controls with the custom validator

```
myForm: FormGroup;
ngOnInit() {
    this.myForm = new FormGroup({
        name: new FormControl('Sammy'),
        email: new FormControl(''),
        url: new FormControl('', [Validators.required, ValidateUrl()]
 });
}
```

- Binding the this variable

```
url: new FormControl('', [Validators.required, this.ValidateUrl.bind(this)]
```

Using the Custom Validator

- Create the form controls with the custom validator

```
myForm: FormGroup;
ngOnInit() {
    this.myForm = new FormGroup({
        name: new FormControl('Sammy'),
        email: new FormControl(''),
        url: new FormControl('', [Validators.required], ValidateUrl() });
}
```

- Binding the this variable

```
url: new FormControl('', [Validators.required, this.ValidateUrl.bind(this)]
```

Async Custom Validators

- The validate() functions must return a Promise or an observable,
- The observable returned must be finite, meaning it must complete at some point.
- To convert an infinite observable into a finite one, pipe the observable through a filtering operator such as first, last, take, or takeUntil.
- synchronous validation happens after the synchronous validation, and is performed only if the synchronous validation is successful.
- This check lets forms avoid potentially expensive async validation processes (such as an HTTP request) if the more basic validation methods have already found invalid input.
- After asynchronous validation begins, the form control enters a pending state.
- Inspect the control's pending property and use it to give visual feedback about the ongoing validation operation.

Update On Option

- Every time the value of a form control changes, Angular reruns our validators.
- This can lead to serious performance issues.
- To alleviate this problem, the v5 release of Angular has introduced the updateOn property on the AbstractControl.
- This means that FormControl, FormGroup, and FormArray, all have this property.
- The updateOn option allows us to set the update strategy of our form controls by choosing which DOM event trigger updates.
- The possible values for the updateOn property are:
  - change, the default: corresponds to the DOM input event of the <input/> element;
  - blur: corresponds to the DOM blur event of the <input/> element;
  - submit: corresponds to the DOM submit event on the parent form.

```
emailFormControl = new FormControl('', {
    validators: [Validators.required, Validators.email],
    updateOn: 'blur'
});
```

Routing

Elite
IT Academy

Why Client-side Routing?

Elite IT Academy

- Defining routes in our application is useful because we can:
  - separate different areas of the app;
  - maintain the state in the app;
  - protect areas of the app based on certain rules;
- Routing lets us define a URL string that specifies where within our app a user should be.
- With client-side routing we're not necessarily making a request to the server on every URL change.
- With our Angular apps, we refer to them as "Single Page Apps" (SPA) because our server only gives us a single page and it's our JavaScript that renders the different pages.

Configuring Routes

- In Angular we configure routes by mapping paths to the component that will handle them.
- There are three main components that we use to configure routing in Angular:
  - Routes describes the routes our application supports
  - RouterOutlet is a “placeholder” component that shows Angular where to put the content of each route
  - RouterLink directive is used to link to routes

Configuring Routes

```
import { RouterModule, Routes } from '@angular/router';
 // In Appmodule.ts
const routes: Routes = [
    // basic routes
    { path: '', redirectTo: 'home', pathMatch: 'full' },
    { path: 'home', component: HomeComponent },
    { path: 'contactus', redirectTo: 'contact' },
    { path: 'protected', component: ProtectedComponent, canActivate: [ LoggedInGuard ] },
    {path: 'products', component: ProductsComponent, children: childRoutes }
    { path: '**', redirectTo: 'home' },
];
```

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

Installing our routes

- Now that we have our Routes routes, we need to install it. To use the routes in our app we do two things to our NgModule:
    - Import the RouterModule
    - Install the routes using RouterModule.forRoot(routes) in the imports of our NgModule

```
imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot(routes), // <-- routes
    ProductsModule
],
```

Using router-outlet

- When we change routes, we want to keep our outer “layout” template and only substitute the “inner section” of the page with the route’s component
- In order to describe to Angular where in our page we want to render the contents for each route, we use the RouterOutlet directive.
- The router-outlet element indicates where the contents of each route component will be rendered.
- The <router-outlet> </router-outlet> acts as a placeholder for components. Angular dynamically adds the component for the route to be activated into this.

```
<router-outlet></router-outlet>
```

Using router-link

- To add links to one of the routes, use the routerLink directive in HTML.
- This directive accepts an array. The first parameter is the name of the route, and the second parameter is the parameters that you want to pass with the route.

```
<a [routerLink]="['component-one', routeParams]"></a>
```

- If you use the routerLink directive without the brackets, you'll need to pass the route as a string.

```
<a routerLink="/component-one"></a>
```

Using navigate method

- The navigate method is used to programatically navigate to different pages in the application.
- To use the router in other components, you can inject the router via the constructor

```
constructor (private router: Router
             private route: ActivatedRoute) {}
```

- To navigate to another page, use the navigate method on the router

```
this.router.navigate(['/servers']);
this.router.navigate(['/servers'], {relativeTo: this.route]);
```

Passing parameters to route

- Parameters can be passed and retrieved through the url to different pages

{ path: ="users/:id/:name", component: UserComponent}

- To retrieve the parameters.

constructor (private route: ActivatedRoute){}

this.route.snapshot.params['id']
this.route.snapshot.params['name']

so with a url
localhost:4200/userid./1/MyName => id = 1, name="MyName";
<a routerLink="['/users', 10, 'abc']"> Add User

Subscribing to route.params

- You can subscribe to a observable when you wish to check when the route parameters change.

```
this.route.params.subscribe( (params: Params) => {
    this.id = params['id'];
    this.name = params['name'];
}
```

- Angular will clean up the subscription whenever the component is destroyed.

Passing query Paramameters

- You can pass query parameters and fragements as follows

  localhost:4200/mypage/1/edit?allowEdit=1#loading

  this.router.navigate(['/mypage', id, 'edit'] {queryParams: {allowEdit: '1'},
  fragments: 'loading'})

  <a [routerLink]="'/servers', 5, 'edit']",
  [queryParams]="{allowEdit: '1'}"
  [fragment]='loading'
  href="#'> Test

Retrieving query Paramameters

- To retrieve the query params you can access the snapshot object

  this.params = this.route.snapshot.queryParams

  this.fragments = this.route.snapshot.fragment

- To preserve the query params you can set the queryParameterHandling attribute to 'preserve'

  this.router.navigate(['edit'], { relativeTo: this.route, queryParamsHandling='preserve']

Creating Child Routes

- Create child routes for those routes that have the same url but with route parameters or Query Parameters or fragments.

```
import { RouterModule, Routes } from '@angular/router';
 // In Appmodule.ts
const routes: Routes = [
    // basic routes
    { path: '', redirectTo: 'home', pathMatch: 'full' },
    { path: 'home', component: HomeComponent , children: [
        { path: '/:id/:name', component: UsersComponent },
    ]},
];
```

Then add one more <router-outlet> tag.

Creating Child Routes

- Create child routes for those routes that have the same url but with route parameters or Query Parameters or fragments.

```
import { RouterModule, Routes } from '@angular/router';
 // In Appmodule.ts
const routes: Routes = [
    // basic routes
    { path: '', redirectTo: 'home', pathMatch: 'full' },
    { path: 'home', component: HomeComponent , children: [
        { path: '/:id/:name', component: UsersComponent },
    ]},
];
```

Then add one more <router-outlet> tag.

Using Route Guards

```
{ path: 'home', component: HomeComponent , children: [
  {path: ':id/:name', component: UsersComponent },
]},
];
```

Then add one more <router-outlet> tag.

- Route Guards help us to prevent users to access a certain area of the application that is not permitted or has access to
- Four types of Route guards are available in Angular.
  - CanActivate
  - CanActivateChild
  - CanDeactivate
  - CanLoad
- note that all Route guards have interfaces that need to be implemented in the component class.
  - eg. “CanActivate” is an interface that has a method called “canActivate”
- Suppose we have multiple Route guards applied to a specific route then all route guards should return true to navigate. If any route guard returns false then the navigation will be canceled.

Using Route Guards

```
{ path: '', redirectTo: 'home', pathMatch: 'full' },
{ path: 'home', component: HomeComponent , children: [
  { path: ':id/:name', component: UsersComponent },
]},
];
```

Then add one more <router-outlet> tag.

- Route Guards help us to prevent users to access a certain area of the application that is not permitted or has access to
- Four types of Route guards are available in Angular
  - CanActivate
  - CanActivateChild
  - CanDeactivate
  - CanLoad
- note that all Route guards have interfaces that need to be implemented in the component class.
  - eg. “CanActivate” is an interface that has a method called “canActivate”
- Suppose we have multiple Route guards applied to a specific route then all route guards should return true to navigate. If any route guard returns false then the navigation will be canceled.

```
const routes: Routes = [

  {path:'home',component:HomeComponent},
  {path:'member',component:MemberComponent,canActivate:[MemberAuthGuard,DetailAuthGaurd]}
];
```

Creating an Auth service

ng g s “Auth” ->Creates the service

```
import { Injectable } from '@angular/core';
    @Injectable({
          providedIn: 'root'
})

export class AuthService {
     constructor() { }
  get MemberAuth()
  {    //Login for Member Auth
    return false;
  }
}
```

Creating a RouteGuard

ng g g "memberAuth" -> creates a Guard

```
import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class MemberAuthGuard implements CanActivate {
      canActivate(route: ActivatedRouteSnapshot,state: RouterStateSnapshot):
                        Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree
       {return true;}}
```

Configure the route guard

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { MemberComponent } from './member/member.component';
 import { MemberAuthGuard } from './member-auth.guard';

const routes: Routes = [
        {path:'home',component:HomeComponent},
       {path:'member',component:MemberComponent,canActivate:[MemberAuthGuard]}
        ];

@NgModule({
     imports: [RouterModule.forRoot(routes)],
      exports: [RouterModule]})
      export class AppRoutingModule { } export const RoutingComponents = [HomeComponent,MemberComponent];
```

Call Auth service

```
import { Injectable } from '@angular/core';
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';
import { AuthService } from './auth.service;

@Injectable({ providedIn: 'root' }) export class MemberAuthGuard implements CanActivate {
      constructor(private _authservice :AuthService){}
      canActivate(route: ActivatedRouteSnapshot,state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean |
UrlTree> | boolean | UrlTree
{
       if(this._authservice.MemberAuth){   return true;}else{
          window.alert("You dont have access!!! Please connect to Administrator ");
       return false;
      }
}
}
```

HTTP CLIENT MODULE

Elite
IT Academy

What is HTTP Client

- HttpClient is a built-in service class available in the @angular/common/http package.
- It has multiple signature and return types for each request.
- It uses the RxJS observable-based APIs, which means it returns the observable and what we need to subscribe it.
- This API was developed based on XMLHttpRequest interface exposed by browsers
- Features
  - Provides typed request and response objects
  - Contains testability features
  - Intercepts request and response
  - Supports RxJS observable-based APIs
  - Supports streamlined error handling
  - Performs the GET, POST, PUT, DELETE operations

Using HTTP Client

- Import or configure the HttpClientModule into the app.module.ts
- You can directly use the HttpClient in your component, but its best to access it via the service.
- We are creating a new service with the help of angular-cli command
  - ng generate service service-name.
- Inject the HttpClient in the service created in the previous step.
- The HttpClient service makes use of observables for all transactions.
- You must import the RxJS observable and operator symbols that appear in the example snippets.
  - import { Observable, throwError } from 'rxjs';
  - import { catchError, retry } from 'rxjs/operators';
- In this step we are going to fetch the data from the server with the help of HTTP GET request. For that, we are adding one method in the service name as getPosts—that method we are calling in the component.

HTTP Get

- We have added the method getPosts and placed HTTP GET request, and passed the one parameter with the request that is nothing but the End-point-url.
- The HTTP GET request and its request and response objects includes the following.

```
get<T>(
    url: string,
     options?: { headers?: [HttpHeaders];
                 context?: [HttpContext];
                 observe?: "body";
                 params?:  [HttpParams];
                 reportProgress?: boolean;
                 responseType?:  "json";
                 withCredentials?: boolean;
               }
):  Observable<T>
```

- The HTTP GET request has around 15 different types of methods to use.

HTTP Get

```
this.http.get<any>('https://api.npms.io/v2/search?q=scope:angular').subscribe(data => {
      this.totalAngularPackages = data.total;
      })
```

- Get with Strongly typed response

```
ngOnInit() {
      this.http.get<SearchResults>('https://api.npms.io/v2/search?q=scope:angular').subscribe(data =>
      {
         this.totalAngularPackages = data.total;
       })
}

interface SearchResults {
    total: number;
    results: Array<object>;
}
```

HTTP Get

- Get with Strongly typed response

```
ngOnInit() {
    this.http.get<any>('https://api.npms.io/v2/invalid-url').subscribe({
     next: data => {
          this.totalAngularPackages = data.total;
    },
    error: error => {this.errorMessage = error.message;
         console.error('There was an error!', error);
      }
   })
}
```

Using the GetPost method

- In the code we have injected the service first in constructor
- Then we call the getPosts method and subscribe to it.
- Whenever we get the response from this subscribe method, it will be a list of object containing id, title, path, as shown below:

[
{id: 1, title: "Angular Localization Using ngx-translate", path: "https://www.telerik.com/blogs/angular-localization-using-ngx-translate"},
{id: 2, title: "How to Use the Navigation or Shadow Property in Entity Framework Core", path: "https://www.telerik.com/blogs/how-to-use-the-navigation-or-shadow-property-in-entity-framework-core"},
{id: 3, title: "Getting Value from appsettings.json in .NET Core", path: "https://www.telerik.com/blogs/how-to-get-values-from-appsettings-json-in-net-core"},
{id: 4, title: "Embedding Beautiful Reporting into Your ASP.NET MVC Applications", path: "https://www.telerik.com/blogs/embedding-beautiful-reporting-asp-net-mvc-applications"},
{id: 5, title: "Select Tag Helper in ASP.NET Core MVC", path: "https://www.telerik.com/blogs/select-tag-helper-asp-net-core-mvc"}
]

HTTP POST method

- This request is used to send data from the application to the server, by using the signature below:

HTTP PUT method

- This request is used to send data from application to server for update

```
ngOnInit() {
  const body = { title: 'Angular PUT Request Example' };

   this.http.put<any>(url, body).subscribe(data => this.postId = data.id);

}

// Put request with error handling
ngOnInit() {
      const body = { title: 'Angular PUT Request Example' };
      this.http.put<any>('https://jsonplaceholder.typicode.com/invalid-url', body).subscribe({
            next: data => {this.postId = data.id;},
           error: error => {
                  this.errorMessage = error.message;
                  console.error('There was an error!', error);
           }
       });
}
```

HTTP DELETE method

- This request is used to delete the data based on the parameter,

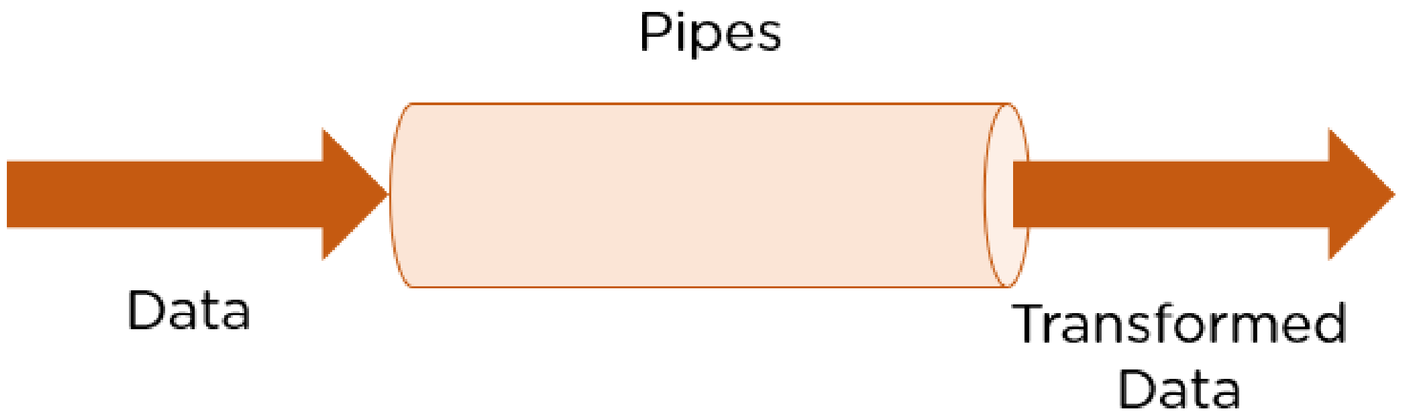
```
ngOnInit() {
        this.http.delete(url).subscribe(() =>
                        this.status = 'Delete successful');
}
```

Pipes

Elite
IT Academy

What are Angular Pipes?

- Angular Pipes transform the output. You can think of them as makeup rooms where they beautify the data into a more desirable format.
- They do not alter the data but change how they appear to the user.



What are Angular Pipes?

- Technically, pipes are simple functions designed to accept an input value, process, and return a transformed value as the output.
- Angular supports many built-in pipes.
- However, you can also create custom pipes that suit your requirements. Some salient features include:
  - Pipes are defined using the pipe “|” symbol.
  - Pipes can be chained with other pipes.
  - Pipes can be provided with arguments by using the colon (:) sign

- Some commonly used predefined Angular pipes are:
  - DatePipe: Formats a date value.
  - UpperCasePipe: Transforms text to uppercase.
  - LowerCasePipe: Transforms text to lowercase.
  - CurrencyPipe: Transforms a number to the currency string.
  - PercentPipe: Transforms a number to the percentage string.
  - DecimalPipe: Transforms a number into a decimal point string.

Creating Custom Pipes

- Angular makes provision to create custom pipes that convert the data in the format that you desire. Angular Pipes are TypeScript classes with the @Pipe decorator.
- The decorator has a name property in its metadata that specifies the Pipe and how and where it is used.

- Pipe implements the PipeTransform interface.
- It receives the value and transforms it into the desired format with the help of a transform() method.

```
/**
 * Transforms text to uppercase.
 *
 * @stable
 */
@Pipe({name: 'uppercase'})
export class UpperCasePipe implements PipeTransform {
  transform(value: string): string {
    if (!value) return value;
    if (typeof value !== 'string') {
      throw invalidPipeArgumentError(UpperCasePipe, value);
    }
    return value.toUpperCase();
  }
}
```

Steps to create a Custom Pipe

- Here are the general steps to create a custom pipe:
  - Create a TypeScript Class with an export keyword.
  - Decorate it with the @Pipe decorator and pass the name property to it.
  - Implement the pipe transform interface in the class.
  - Implement the transform method imposed due to the interface.
  - Return the transformed data with the pipe.
  - Add this pipe class to the declarations array of the module where you want to use it.
- Alternatively, you can use the following command,
  - ng g pipe <nameofthepipe>

Pure Pipes

- Pure pipes in angular are the pipes that execute when it detects a pure change in the input value.
- A pure change is when the change detection cycle detects a change to either a primitive input value (such as String, Number, Boolean, or Symbol) or object reference (such as Date, Array, Function, or Object).

```
<div> {{ user | myPipe }} </div>
```

- myPipe will execute if it detects a change in the user's object reference.
- myPipe will not execute even if the property has changed because the object reference is still the same.
- A single instance of pipe is used all over the component.
- A pure must use a pure function.
- A pure function does not depend on any state, data, or change during the execution. In other words, given the same arguments, a pure function should always return the same output.
- By default, pipes in angular are pure pipes. Custom pipes can be defined as pure pipes by turning the pure flag to true of the @Pipe decorator.

```
@Pipe({
name: 'purePipe',
pure: true
})
export class PurePipe {}
```

Impure Pipes

- Impure pipes in angular are the pipes that execute when it detects an impure change in the input value.
- An impure change is when the change detection cycle detects a change to composite objects, such as adding an element to the existing array.
- Now, myPipe will execute on every change.
- Impure pipes are required because angular ignores changes to composite objects.
- Impure pipes execute every time angular detects any changes regardless of the change in the input value.
- It uses the impure function.
- An impure function depends on any state, data, or change during the execution and may not return the same result if the same inputs are passed into the respective function.

```
@Pipe({
name: 'impurePipe',
pure: false
})
export class ImpurePipe {}
```

Services

Elite
IT Academy

Dependency Injection

Elite IT Academy

- Dependency Injection (DI) is a system to make parts of our program accessible to other parts of the program
- We can configure how that happens.
- The major benefit of using Dependency Injection is that the client component needn’t be aware of how to create the dependencies.
- All the client component needs to know is how to interact with those dependencies.

Creating & using a Service

```
export class PriceService {
    constructor() { }

    calculateTotalPrice(basePrice: number, state: string) {
        // e.g. Imgine that in our "real" application we're
        // accessing a real database of state sales tax amounts
         const tax = Math.random();

         return basePrice + tax;
    }
}
```

```
import { PriceService } from './price.service';

export class Product {
    service: PriceService;
    basePrice: number;

    constructor(basePrice: number) {
            this.service = new PriceService();
            this.basePrice = basePrice;
    }

    totalPrice(state: string) {
          return this.service.calculateTotalPrice(
                  this.basePrice, state);
    }
}
```

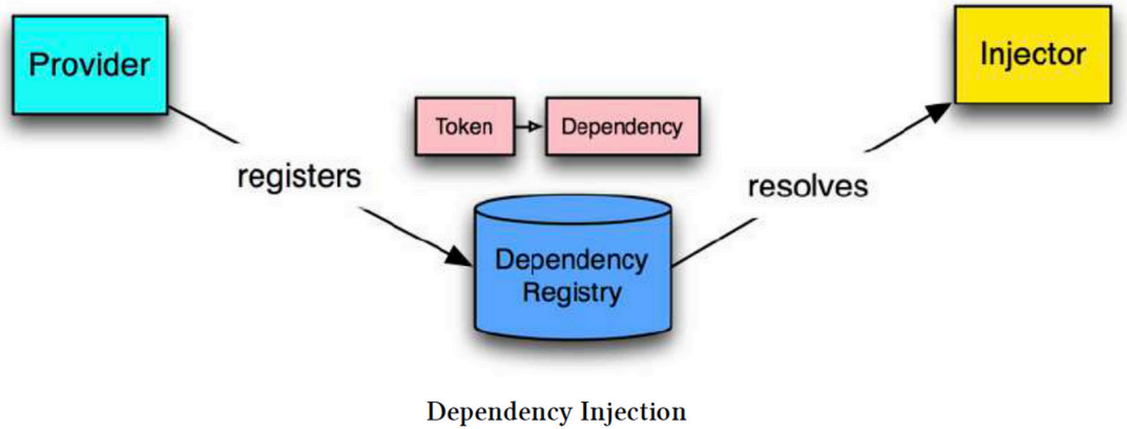
Creating & using a Service

- Within Angular's DI system, instead of directly importing and creating a new instance of a class, instead we will:
  - Register the "dependency" with Angular
  - Describe how the dependency will be injected
  - Inject the dependency
- Dependency injection in Angular has three pieces:
  - the Provider (also often referred to as a binding) maps a token (that can be a string or a class) to a list of dependencies.
  - It tells Angular how to create an object, given a token.
  - The Injector that holds a set of bindings and is responsible for resolving dependencies and injecting them when creating objects
  - the Dependency that is what's being injected

Creating & using a Service

- use NgModule to register what we'll inject
  - these are called providers and
  - use decorators (generally on a constructor) to specify what we're injecting

```
providers: [
    UserService // <-- added right here
],
```



Dependency Injection

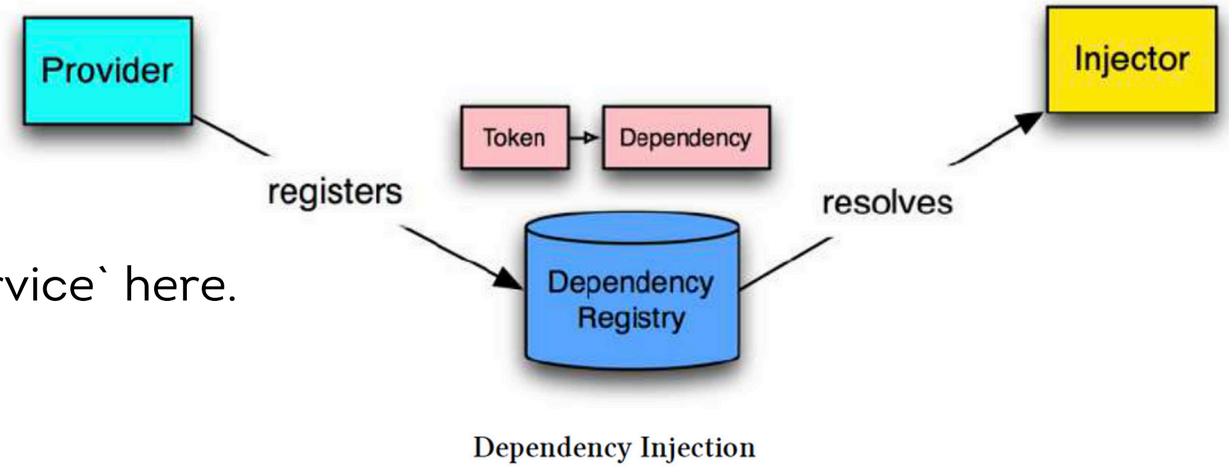
Terms used

- Injectable - any class decorated with @Injectable, for example a service.
- injector - an Angular class that is capable of providing Injectables to classes below it. (This includes all components and modules.)
- injector scope/level - the scope of all class instances that live "below" a specific injector.
- injector hierarchy - a proritized tree of injector scopes, organized in platform -> root -> module -> component order.
- Injectable is provided - an instance of the Injectable will be given to classes below this specific injector level, whenever they request it.
- Injectable is injected - a class constructor has requested to be given some instance of the service, so Angular will try to give it the nearest instance that can be found in the injector hierarchy".
- tree-shaking - an optimization that happens automatically thanks to the Angular compiler. When it detects that some code is not being used, that code is removed from the final compilation of the app (or compilation of a given lazy-loaded module).

Injecting a Service

Now we can inject UserService into our component like this:

```
import { UserService } from '../services/user.service';

// Angular will inject the singleton instance of `UserService` here.
// We set it as a property with `private`.
constructor(private userService: UserService) {
// empty because we don't have to do anything else!
}

signIn(): void {
    // when we sign in, set the user
    // this mimics filling out a login form
        this.userService.setUser({
        name: 'Nate Murray'});
```



Dependency Injection

Using Injectable

- Services can be loaded lazily by Angular (behind the scenes) and redundant code can be removed automatically.
- This can lead to a better performance and loading speed - though this really only kicks in for bigger services and apps in general.
- In the service component

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

@Injectable()
export class UserService {
    user: any;

    setUser(newUser) {  this.user = newUser;}
    getUser(): any {  return this.user;}
}
```

Using Injectable

- Instead of adding a service class to the providers[] array in AppModule , you can set the following config in @Injectable() :
  - @Injectable({providedIn: 'root'})
- providedIn: 'root' is the easiest and most efficient way to provide services since Angular 6:
- The service will be available application wide as a singleton with no need to add it to a module's providers array (like Angular <= 5).
- If the service is only used within a lazy loaded module it will be lazy loaded with that module
- If it is never used it will not be contained in the build (tree shaked).

Angular Signals

Elite
IT Academy

What are Signals

- A new primitive type called “Signal” has been introduced in Angular v16.
- It’s designed to store a value similar to a regular variable.
- Signals contain values that change over time; when you change a signal's value, it automatically updates anything that uses it.
- Signals store both primitive data types and objects.
- Signals always have a value, signals are side effect free, and signals are reactive, keeping their dependents in sync.

Settable Signals

- The signal() function produces a specific type of signal known as a Settable Signal.
- In addition to being a getter function, SettableSignals have an additional API for changing the value of the signal (along with notifying any dependents of the change).
- These include the
  - .set operation for replacing the signal value,
  - .update for deriving a new value, and
  - .mutate for performing internal mutation of the current value.
- These are exposed as functions on the signal getter itself.

```
const counter = signal(0);
counter.set(2);
counter.update(count => count + 1);
```

- The signal value can be also updated in-place, using the dedicated .mutate method:

```
const todoList = signal<Todo[]>([]);
todoList.mutate(list => {
  list.push({
     title: 'One more task', completed: false
    });
});
```

Appendix

Elite
IT Academy

Keyboard Events

- You can bind to keyboard events using Angular's binding syntax.
- You can specify the key or code that you would like to bind to keyboard events.
- They key and code fields are a native part of the browser keyboard event object.
- By default, event binding assumes you want to use the key field on the keyboard event.
- You can also use the code field.

```
<input (keydown.code.shiftleft.altleft.keyt)="onKeydown($event)" />
```