# JAVA FULL STACK DEVELOPMENT PROGRAM

## Session 19: Angular Component & Service

# OUTLINE

- Component Interaction
    - Parent – Child relationship
    - Services
- Component Lifecycle Hook
- Dependency Injection
- Interaction with server using HTTP

# NESTED/CHILD COMPONENT

- The Angular follows component-based Architecture, where each component manages a specific task or workflow. Each component is an independent block of the reusable unit.
- In real life angular application, we need to break our application into a small child or nested components.
    - For example, we can define a pop-up component, so that whenever we need a pop-up (showing errors or overlay), we can reuse it.
    - We can render it using child component's selector into parent component.
- However, how do those component communicate with each other?
    - For example, for different pop-ups, we need to display different error message. So instead of hard coding the message inside the pop-up, we need a way to pass the message from the parent to the root.

# @INPUT

- In Angular, Parent component can pass data to child with input binding
  - To do that the Child component must expose its properties to the parent component.
  - The Child Component does this by using the @Input decorator
- In the Child Component
  - Import the @Input module from @angular/Core Library
  - Mark those property, which you need data from parent as input property using @Input decorator
- In the Parent Component
  - Bind the Child component property in the Parent Component when instantiating the Child

# COMPONENT COMMUNICATION

- Now we can show the pop-up message dynamically by passing data from parent to child component. What if we want pass data from child to parent?

- There are three ways in which parent component can interact with the child component
  - Parent Listens to Child Event
  - Parent uses Local Variable to access the child
  - Parent uses a @ViewChild to get reference to the child component

# LISTEN TO CHILD EVENT

- The Child Component exposes an EventEmitter Property.
  - It is an output property, typically work with @Output decorator
  - When Child Component needs to communicate with the parent it raises the event, the Parent Component listens to that event and reacts to it
- How to Pass data to parent component using @Output
  - Declare a property of type EventEmitter and instantiate it
  - Mark it with a @Output annotation
  - Raise the event passing it with the desired data

# LOCAL VARIABLE

- Parent Template can access the child component *properties* and *methods* by creating the template reference variable
  - Syntax: <child-component *#variable*></child-component>
  - The “child” is called template reference variable, which now represents the child component

app.component.ts

```
1
2         <child-component #child></child-component>` ,
3
```

app.component.ts

```
1
2         <p> current count is {{child.count}} </p>
3         <button (click)="child.increment()">Increment</button>
4         <button (click)="child.decrement()">decrement</button>
5
```

# @VIEWCHILD

- The local variable approach is simple and easy. But it is limited because the parent-child wiring muse be done entirely within the parent template.

- This means the parent component itself does not have access to child resources.

- This means parent component cannot read or write child component values or call child component methods

- In above cases, we use @ViewChild technique to inject child component into parent component

# @VIEWCHILD

- @ViewChild is another technique used by the parent to access the property and method of the child component
- The @ViewChild decorator takes the name of the component/directive as its input. It is then used to decorate a property.
- Decorate parent property with @ViewChild decorator passing it the name of the component to inject
- When angular creates the child component, the reference to the child component is assigned to the child property.

# COMPONENT COMMUNICATION

- @Input: Pass from parent to child
- @Output: with EventEmitter pass from child to parent
  - #local: pass from child to parent
- @ViewChild: pass from child to parent
- All the methods above works between parent and child component, which has a hierarchy architecture, components are not in the sample level
- What if we want to share data between multiple components that are in the same level?

# COMPONENT LIFECYCLE

- Like servlet, thread and other data forms, Angular component has a lifecycle that starts when Angular instantiates the component class and renders the component view along with its child views
- Lifecycle continues with change detection, as Angular will check for any property or data changes and update the view as needed
- Lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM
- Directive has similar lifecycle as directive is just a component without view/template

# COMPONENT LIFECYCLE

- We can implement lifecycle hook interfaces in Angular to act or respond to an event
- Each interface defines the prototype for a single hook method, whose name is the interface name prefixed with ng.
- After our application instantiates a component by calling its constructor, Angular calls the hook method we have implemented at the appropriate point in the life cycle of that instance.

# COMPONENT LIFECYCLE

| | |
|---|---|
| ngOnChanges | Called after a bound input property changes |
| ngOnInit | Called once the component is initialized |
| ngDoCheck | Called during every change detection run |
| ngAfterContentInit | Called after content (ng-content) has been projected into view |
| ngAfterContentChecked | Called every time the projected content has been checked |
| ngAfterViewInit | Called after the component's view (and child views) has been initialized |
| ngAfterViewChecked | Called every time the view (and child views) have been checked |
| ngOnDestroy | Called once the component is about to be destroyed |

# COMPONENT LIFECYCLE

- **ngOnChanges()**
  - Angular invoke ngOnChanges() method when Angular sets or resets data-bound **input** properties
  - Method receives **SimpleChanges** object or current and previous property values
  - This method happens very frequently, any operation we perform in this method will impact performance significantly

# COMPONENT LIFECYCLE

- **ngOnInit()**
  - Used to initialize the directive or component after Angular first displays the data-bound properties and sets the input properties.
  - It is used to perform complex initialization outside of the constructor. Angular components should be cheap and easy to construct.
- Should not fetch data in a component constructor.
- Component will first call ngOnChanges() then call ngOnInit(). It get all the properties first then use those properties to do operations.
- Component will only call ngOnInit() once.

# COMPONENT LIFECYCLE

- ngDoCheck
  - Detect and act upon changes that Angular can't or won't detect on its own.
  - It is used to monitor changes that occur where ngOnChanges() won’t catch.
- ngDoCheck vs. ngOnChanges
  - ngOnChanges will be notified of changes if the Input is a primitive type or your Input reference changes
  - If the model reference doesn’t change, but some property of the Input model changes, we may implement the ngDoCheck lifecycle hook to construct your change detection logic manually

# COMPONENT LIFECYCLE

- ngAfterContentInit()
  - This Life cycle hook is called after the Component's content has been fully initialized.
  - The Angular Component can include the external content from the child Components by adding them using the <ng-content></ng-content> element.
    - This hook is fired after these projected contents are initialized.
  - This is a component only hook and Called Only once.

# COMPONENT LIFECYCLE

- ngAfterContentChecked()
  - This life cycle hook is called after the Components Content is checked by the Angular's Change detection module.
- It is called immediately after ngAfterContentInit and after every subsequent ngDoCheck().
- This is a component only hook

# COMPONENT LIFECYCLE

- ngAfterViewInit()
  - Similar to ngAfterContentInit, but invoked after Angular initializes the component's views and all its child views.
- This is called once after the first ngAfterContentChecked.
- A component-only hook.

# COMPONENT LIFECYCLE

- ngAfterViewChecked()
  - The Angular fires this hook after it checks the component's views and child views.
  - This event is fired after the ngAfterViewInit and after that for every subsequent ngAfterContentChecked hook.
  - This is a component only hook.

# COMPONENT LIFECYCLE

- ngOnDestroy()
  - This hook is called just before the Component/Directive instance is destroyed by Angular.
  - We can Perform any cleanup logic for the Component here.
  - This is the correct place where you would like to Unsubscribe Observables object and detach event handlers to avoid memory leaks.

# COMPONENT INTERACTION

- With CSS selector, it is possible to include one component inside another component
- Angular provides us few ways to component communication, which two or more components share information
  - Parent to child
  - Child to Parent
  - Services

# ANGULAR DI FRAMEWORK

- Angular 2 Dependency Injection Framework helps us to implement the Dependency injection Pattern.
- This framework consists of four main parts:
  - Consumer: The Component that needs the Dependency
  - Dependency: The Service that is being injected.
  - Provider: Maintains the list of Dependencies. It provides the instance of dependencies to the injector
  - Injector: Responsible for injecting the instance of the Dependency to the Consumer

# SERVICE

- Services allow for greater separation of concerns for your application and better modularity by allowing you to extract common functionality out of components.
- Service is a piece of reusable code with a Focused Purpose. A code that we will use it in many components across our application.
- Services may have their associated properties and the methods, which can be included in the component. Services are injected, using DI (Dependency Injection).
- Services share data or functions between different parts of angular application.
- Use cases:
  - Features that are independent of components such a logging services
  - Share logic and data across components (calculation)
  - Encapsulate external interactions like data access (ex. service & dao)

# SERVICE

- Service is a class with @Injectable() decorator
- @Injectable decorator is an essential gradient in every Angular server definition
- The decorator marked our service so Angular knows it can be injected. But Angular cannot inject the service until we configure a **dependency injector**.
- Normally we do not need to configure dependency injectors, Angular creates one for us during the execution, which is the root injector.

# SERVICE

- @Injectable() decorator has the **providedIn** metadata parameter which allows us to specify the provider(injector) of the service

- By default, it will be the root injector

- Both @NgModule and @Component decorators have the providers metadata option, where we can configure providers for module-level and component-level injectors.

# ANGULAR PROVIDER

- Provider can be configured at two level
  - NgModule
    - The service (dependency) is shared with all instance of components in the application
- Component or Directive
  - The service (dependency) is shared across each instance of components and its children.

# ANGULAR PROVIDER

- Angular Provider Syntax
  - providers: [{provide: service, useClass: serviceClass}]
- The above syntax has two properties.
  - Token: The First property Provide is known as the Token. The token is used by the injector to locate the provider
  - Provider: The second property useClass is Provider. It tells how and what to inject.
- The injector maintains an internal collection of token-provider map. The token acts as a key to that collection. Injector uses the Token (key) to locate the provider.

# ANGULAR PROVIDER

- The Angular-Dependency Framework provides several types of providers.
  - Class Provider(useClass)
    - The Class Provider is used, when you want to provide an instance of the class.

```
providers :[{ provide: ProductService, useClass: ProductService }]
```

- Value Provider (useValue)
  - Value Provider is used, when you want to provide a Simple value
  - This property can be used when you want to provide URL of Service class, Application wide Configuration Option etc.

```
const APP_CONFIG = { serviceURL: "www.serviceUrl.com\api", IsDevleomentMode: true
};
providers: [{ provide: AppConfig, useValue: APP_CONFIG }]
```

# OBSERVABLE

- Observables provide support for passing messages between parts of our application

- They are used frequently in Angular and are the recommended technique for event handling, asynchronous programming, and handling multiple values

- It is a design pattern. It has an object, which is called **subject**, and maintain a list of dependents called **observers**, and notify the observers automatically of any state changes.

- Similar to publish/subscribe design pattern.

# OBSERVABLE

- Observables are declarative, which means when we define a function to publish the values, the function will not execute until a consumer subscribes to it.
- Think of shopping online, you want to trace the item price, so you click the watch button. The item will be added to your watch list. Whenever there is a price change on this item, you will receive an email with details. Only users who watched the item will receive the email.
- Observables can deliver multiple values over time — If we subscribe the item, we will keep receiving emails about the price change. The seller decides when to change the price, all we need to do is subscribe and wait.

# OBSERVABLE

- Observable is similar to Promise, we can use it for asynchronization implementation
- But why choose Observable over Promise?
  - Observables do not mutate the server response. Instead, we can use a series of operators to transform values as needed
  - For HTTP requests, we can also use **unsubscribe()** method to cancel the subscription
  - Failed requests can be retried easily
  - Observable handle data like a stream, we can allow zero to multiple event to come through where promise only handles sing event
  - Observable is lazy, nothing happens until we subscribe the event, easy to maintain. Whereas Promise is eager, it executes immediately.

# OBSERVABLE

- RxJS is a great tool for managing data with help of the Observer pattern.
  - Instead of keeping state in a variable, it stores data in a stream

```
pp > components > http-ex > observables > TS observables.compon
import { Component, OnInit } from '@angular/core';
import { Observable, Subscription } from 'rxjs';
```

- Observable instance begins publishing values only when someone subscribes to it.
  - Eg. subscribe(success, error, complete)
- The Subscriber Operator is the glue that connects an observer to an Observable.
- We need to first create an Observable object and defines it's behavior.

```
test: Observable<any> = new Observable<any>(function subscribe(subscriber) {
  subscriber.next("Landon");
  subscriber.next("Fan");
  subscriber.next("Zack");
  // subscriber.error("This is an error test");
  subscriber.complete();
});
```

# OBSERVABLE

- Observer has three basic methods
  - Next(): send value like a number, string, etc.
  - Complete(): will not send any value, but indicates the observable as completed
  - Error(): will send an error and stop execution

# OBSERVABLE

- The subscribe method has three arguments. Each specifies the action to be taken when a particular event occurs
  - Success: This event is raised whenever observable returns a value. We use this event to assign the response to the repos
  - Failure: This event occurs, when observable is failed to generate the expected data or has encountered some other error
- Completed: This event fires, when the observables complete its task. We disable the loading indicator here.

```
this.subscriber = this.test.subscribe(
  success => console.log(success),
  error => console.log(error),
  () => console.log("Observable is complete")
);
```

# OBSERVABLE

- We can also create our own observables to change the data at runtime
    - Subject — Subjects are observables themselves but what sets them apart is that they are also observers
        - It means that a subject can emit data
        - It supports multiple subscriptions
- BehaviorSubject — When you subscribe to a behavior subject, it will give you the last emitted value right away.

# INTERACTING WITH HTTP

- In order to start making HTTP calls from our Angular app
  - we need to import the angular/http module and register for HTTP services.
    - It supports both XHR and JSONP requests exposed through the HttpModule and JsonpModule respectively.
- HttpModule serves the purpose to perform HTTP requests.
- Asynchronous HTTP requests can be implemented using callbacks, promises or observables

# IMPORT IN MODULE

```
import { ViewChildComponent } from './components/component-communication-ex/view-child/view-child.component';
import { ServiceExComponent } from './components/service-ex/service-ex.component';
import { SubOneComponent } from './components/service-ex/sub-one/sub-one.component';
import { SubTwoComponent } from './components/service-ex/sub-two/sub-two.component';
import { HttpExComponent } from './components/http-ex/http-ex.component';
import { ObservablesComponent } from './components/http-ex/observables/observables.component';
import { HttpComponent } from './components/http-ex/http/http.component';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent,
    ComponentCommunicationExComponent,
    LifeCycleComponent,
    InputComponent,
    OutputComponent,
    LocalVariableComponent,
    ViewChildComponent,
    ServiceExComponent,
    SubOneComponent,
    SubTwoComponent,
    HttpExComponent,
    ObservablesComponent,
    HttpComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

# INTERACTING WITH HTTP

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class HttpService {

  endPoint: string = "localhost:8080";

  constructor(private http: HttpClient) { }

  getData(): Observable<Object> {
    return this.http.get(this.endPoint);
  }
}
```

```
export class HttpComponent implements OnInit {

  constructor(private httpService: HttpService) { }

  ngOnInit(): void {
  }

  getData() {
    this.httpService.getData().subscribe(
      (data) => {
        console.log(data);
      }
    );
  }

}
```