MEAN STACK UNIT-04

ANGULAR

What is Angular?

Angular is a popular open-source web application framework that is primarily used to build dynamic single-page applications (SPAs). It is developed and maintained by Google. Angular allows developers to build client-side applications using TypeScript, a superset of JavaScript.

Some key features of Angular include:

- **Component-based architecture**: Angular follows a component-based approach, where the application is divided into reusable and self-contained components. Each component consists of HTML templates, TypeScript code, and CSS styles, which work together to define the component's behavior and appearance.
- **Two-way data binding**: Angular provides a powerful data binding mechanism that allows automatic synchronization of data between the model (component) and the view (HTML). This means that any changes in the model will reflect in the view and vice versa, without the need for manual DOM manipulation.
- **Dependency injection**: Angular has built-in dependency injection, which helps manage the dependencies between different parts of an application. It simplifies the creation and management of objects and promotes modularity, testability, and reusability.
- **Routing**: Angular includes a robust router module that enables developers to define and handle application navigation. It allows the creation of multiple views and provides features like lazy loading, route guards, and parameter passing.
- **Form handling**: Angular provides powerful form handling capabilities with features like form validation, error handling, and data binding. It supports both template-driven forms (where most of the form logic is defined in the template) and reactive forms (where the form logic is defined programmatically using reactive programming).
- **Testing support**: Angular has a comprehensive testing framework that allows developers to write unit tests, integration tests, and end-to-end tests for their applications. This helps ensure the quality and reliability of the codebase.

Angular is widely used by developers for building complex and scalable web applications. It has a large ecosystem of libraries, tools, and community support, making it a popular choice for web development.

Features of Angular: Angular offers several features that contribute to its popularity among developers. Here are some key features of Angular:

- **Component-based architecture**: Angular follows a component-based approach, where the application is divided into reusable and self-contained components. This promotes modular development, code reusability, and maintainability.
- **Two-way data binding**: Angular provides a powerful data binding mechanism that enables automatic synchronization of data between the model (component) and the view (HTML). Changes in the model are reflected in the view and vice versa, without manual DOM manipulation.
- **Dependency injection**: Angular has a built-in dependency injection system that facilitates the management of dependencies between different parts of an application. It helps improve code quality, reusability, and testability by promoting loose coupling and modular design.
- **Templating**: Angular offers a declarative templating language that allows developers to define dynamic and interactive views. Templates in Angular combine HTML with additional syntax and directives to render data and respond to user interactions.
- **Routing**: Angular provides a powerful routing module that enables developers to define and handle application navigation. It allows the creation of multiple views and supports features like lazy loading, route guards, and parameter passing.
- **Forms handling**: Angular provides robust support for building and validating forms. It offers two approaches: template-driven forms, where the majority of the form logic is defined in the template itself, and reactive forms, where the form logic is defined programmatically using reactive programming.
- **Cross-platform development**: Angular supports cross-platform development, allowing developers to build applications that run on various platforms, including web browsers, desktop, and mobile. It achieves this through frameworks like Angular Universal for server-side rendering and Angular Mobile Toolkit for mobile app development.
- **Testing support**: Angular has a comprehensive testing ecosystem that includes tools and libraries for writing unit tests, integration tests, and end-to-end tests. It promotes test-driven development (TDD) and provides utilities for mocking and simulating dependencies.
- **Performance optimization**: Angular offers various features to optimize application performance. It includes features like lazy loading, Ahead-of-Time (AOT) compilation, tree shaking, and code minification, which help reduce the bundle size and improve load times.
- **Extensibility**: Angular is highly extensible and allows developers to add custom functionality through directives, pipes, and services. It also has a rich ecosystem of third-party libraries and packages that can be integrated into Angular applications.

These features make Angular a powerful and versatile framework for building modern, scalable, and maintainable web applications.

Angular Application Setup:

To set up an Angular application, follow these steps:

- Install Node.js and npm (Node Package Manager) if they are not already installed on your system. You can download them from the official Node.js website (https://nodejs.org).
- Open a terminal or command prompt and verify that Node.js and npm are installed by running the following commands:

node -v

npm -v

- Install the Angular CLI (Command Line Interface) globally by running the following command:

npm install -g @angular/cli

- Create a new Angular project by running the following command:

ng new my-angular-app

Replace "my-angular-app" with the desired name for your application. The Angular CLI will set up a new project with the necessary files and dependencies.

- Navigate into the project directory:

cd my-angular-app

- Start the development server by running the following command:

ng serve

This command compiles your Angular application and launches a development server. You can access your application by opening a web browser and navigating to http://localhost:4200/.

By default, the development server will watch for changes in your code and automatically recompile and reload the application in the browser.

- Open your favorite code editor and start modifying the files in the src directory to build your Angular application. The main file is src/app/app.component.ts, which contains the root component for your application.

From this point, you can begin developing your Angular application by creating components, services, and other Angular constructs. You can use the Angular CLI to generate components, services, modules, and more, which will automatically create the necessary files and update the configuration.

For more information on how to develop Angular applications, refer to the Angular documentation: https://angular.io/docs

Components and Modules:

In Angular, components and modules are key building blocks for structuring and organizing your application. They play a crucial role in creating reusable and modular code. Here's an overview of components and modules in Angular:

Components:

A component in Angular represents a self-contained unit that combines the template (HTML), the styling (CSS), and the behavior (TypeScript) of a specific part of the user interface. It encapsulates a section of the UI and can be reused throughout the application.

Components have a lifecycle that includes initialization, rendering, and destruction. They interact with the user through event binding (listening for user actions) and data binding (updating the view based on data changes).

To create a component, you typically use the Angular CLI command:

ng generate component component-name

This command generates the necessary files (HTML template, CSS styles, TypeScript code) for the component and updates the relevant configuration.

Modules:

A module in Angular is a container that consolidates related components, services, directives, and other code into a cohesive unit. It helps manage the complexity of large applications by organizing the codebase into functional areas.

Modules provide a way to group related code, define dependencies, and enable reusability. They encapsulate and isolate functionality, making it easier to maintain and test. In Angular, the application is typically divided into multiple modules, such as a root module and feature modules.

To create a module, you can use the Angular CLI command:

ng generate module module-name

This command generates the necessary files for the module and updates the relevant configuration.

Modules can also import and export other modules, allowing components, services, and other resources to be shared across modules. This promotes modularity and code reuse.

Additionally, Angular has a root module called the AppModule, which serves as the entry point of the application. It imports and configures other modules and specifies the bootstrap component that is loaded when the application starts.

Components and modules work together to structure and organize an Angular application. Components represent individual UI elements, while modules provide a way to group and manage related components and other code. This modular approach helps improve maintainability, reusability, and testability of the application.

Executing Angular Application:

To execute an Angular application, you need to follow these steps:

- Make sure you have set up your Angular application and installed the necessary dependencies as mentioned in the previous response.
- Open a terminal or command prompt and navigate to the root directory of your Angular application.
- Run the following command to start the development server:

ng serve

This command will compile your Angular application and launch a development server. The server will monitor your code for changes, recompile the application as needed, and automatically reload it in the browser.

- Open a web browser and go to http://localhost:4200/ (or the specific URL shown in the terminal) to view your Angular application. The development server provides a live preview of your application, and any changes you make to the code will be reflected instantly in the browser.
- Start interacting with your Angular application in the browser and test its functionality. You can navigate through the application, click on buttons or links, fill out forms, and observe how the application responds.
- While the development server is running, you can continue making changes to your code, and the server will automatically rebuild and reload the application with the updated changes.
- To stop the development server, go back to the terminal or command prompt where it is running and press Ctrl + C to terminate the server.

That's it! You have successfully executed your Angular application using the development server. During development, this is the preferred way to run and test your application as it provides a smooth development experience with automatic reloading. For production deployment, you would typically build the application using the Angular CLI and deploy the generated output to a web server.

Elements of Template:

In Angular, the template is a key part of a component and is responsible for defining the structure and layout of the component's view. The template consists of HTML markup combined with Angular-specific syntax and features. Here are the elements commonly used in an Angular template:

- **Interpolation ({{ }})**: Interpolation allows you to embed dynamic values from the component's data into the template. You can use double curly braces {{ }} to surround an expression that will be evaluated and displayed in the rendered view.
- **Property Binding ([])**: Property binding allows you to bind a component's property to an element's property or attribute in the template. You use square brackets [] to bind a component property to an element property or attribute.
- **Event Binding (())**: Event binding allows you to respond to user actions such as button clicks, mouse events, or form submissions. You use parentheses () to bind a component method to an element's event.
- **Directives**: Angular provides built-in and custom directives that allow you to manipulate the structure and behavior of the DOM elements. Directives can be applied to elements as attributes, classes, or as structural directives that alter the layout of the DOM.
- **Structural Directives (*ngIf, *ngFor, *ngSwitch)**: Structural directives allow you to conditionally add or remove elements from the DOM, iterate over a collection to generate multiple elements, or switch between different elements based on a condition. They are denoted by an asterisk * prefix.
- **Template Reference Variables (#)**: Template reference variables allow you to capture references to HTML elements or Angular directives within the template. You can use these variables to interact with the elements or directives in your component's logic.
- **Two-Way Data Binding ([(ngModel)])**: Two-way data binding combines property binding and event binding to create a bidirectional data flow between the component and the template. It allows you to update the component's property and reflect the changes in the template automatically.
- **Pipes (|)**: Pipes allow you to transform and format data before displaying it in the template. Angular provides several built-in pipes such as date, uppercase, lowercase, and you can also create custom pipes.
- **Template Expression Operators**: Angular templates support various operators for performing operations and evaluations within expressions. These include arithmetic operators (+, -, *, /), comparison operators (==, !=, >, <), logical operators (&&, ||), and more.

These are some of the key elements you'll commonly find in an Angular template. By utilizing these elements, you can create dynamic and interactive views that respond to user input and reflect changes in the component's data.

Change Detection:

Change detection is a core concept in Angular that tracks changes to the application's data and updates the corresponding views. It ensures that the user interface reflects the most up-to-date state of the

application. Angular employs a mechanism called "change detection" to efficiently identify and propagate changes throughout the application.

Here's how change detection works in Angular:

- Change detection starts at the root component of the application and traverses the component tree, visiting each component.
- During change detection, Angular compares the current state of a component's data with its previous state to detect changes. It does this by performing a deep check of the component's properties, including objects and arrays.
- If Angular detects changes in a component's data, it updates the associated view to reflect the new values. This process involves updating the DOM, rerendering the templates, and updating any bindings or directives applied to the elements.
- After updating a component's view, Angular checks if any child components have been affected by the changes. If so, it recursively performs change detection on those components as well, ensuring that changes propagate throughout the component tree.

Angular's change detection algorithm is optimized to be efficient and performant. It uses a technique called "unidirectional data flow," where changes flow from parent components to child components in a hierarchical manner.

In addition to the default change detection strategy, Angular provides two alternative change detection strategies:

- OnPush Change Detection: With this strategy, change detection is triggered only when the component's input properties change or when an event occurs within the component. It allows for better performance by reducing unnecessary checks.
- Manual Change Detection: In this strategy, change detection is entirely controlled by the developer. Angular does not automatically perform change detection. Instead, developers manually trigger change detection when they explicitly call the detectChanges() method.

By understanding change detection in Angular, developers can optimize their application's performance, minimize unnecessary checks, and ensure that the UI is always in sync with the underlying data.

Structural Directives - ngIf, ngFor, ngSwitch:

Structural directives are a type of directive in Angular that alter the structure of the DOM (Document Object Model) by adding or removing elements based on certain conditions. Angular provides several built-in structural directives, including ngIf, ngFor, and ngSwitch. Here's an overview of these directives:

- **ngIf**:

ngIf is used to conditionally add or remove elements from the DOM based on a given condition. It evaluates an expression and only renders the associated template if the expression evaluates to a truthy value. If the expression is falsy, the template is removed from the DOM.

Example:

```
<div *ngIf="showElement">
  This element is conditionally rendered.
</div>
```

In the example, the element will only be rendered if the showElement property in the component evaluates to true.

- **ngFor**:

ngFor is used for looping over a collection of items and generating multiple elements in the DOM. It iterates over an array or an iterable object and repeats a section of HTML for each item in the collection.

Example:

```
<ul>
  <li *ngFor="let item of items">
    {{ item }}
  </li>
</ul>
```

In the example, each item in the items array is rendered as a <li> element within the <ul> list.

- **ngSwitch**:

ngSwitch is used for conditional rendering based on multiple values. It acts as a switch statement to determine which template to render based on the value of a given expression.

Example:

```
<div [ngSwitch]="color">
  <p *ngSwitchCase="'red'">Red color is selected</p>
```

```
<p *ngSwitchCase="'blue'">Blue color is selected</p>

<p *ngSwitchCase="'green'">Green color is selected</p>

<p *ngSwitchDefault>No color is selected</p>

</div>
```

In the example, the <p> element within each ngSwitchCase block will be rendered based on the value of the color property in the component. If no case matches, the <p> element within ngSwitchDefault will be rendered.

These structural directives provide powerful ways to conditionally render elements, loop over collections, and perform switch-like logic in Angular templates. They enable dynamic and flexible rendering of the DOM based on the application's data and conditions.

Custom Structural Directive:

In Angular, you can create custom structural directives to extend the functionality of the built-in directives or introduce new structural directives specific to your application's requirements. Custom structural directives allow you to manipulate the DOM structure based on custom conditions. Here's how you can create a custom structural directive in Angular:

- Create a new directive using the Angular CLI command:

  ng generate directive directive-name

This command generates the necessary files for the directive, including a TypeScript file and a test file.

- Open the generated TypeScript file (e.g., directive-name.directive.ts) and import the necessary modules and decorators:

  import { Directive, TemplateRef, ViewContainerRef } from '@angular/core';
- Decorate the directive class with the @Directive decorator, providing a selector to use the directive in the template:

```
@Directive({

  selector: '[appDirectiveName]'

})

export class DirectiveNameDirective {
```

```
  constructor(private templateRef: TemplateRef<any>, private viewContainerRef: ViewContainerRef)
{ }

}
```

- In the directive class, inject TemplateRef and ViewContainerRef into the constructor. These parameters allow you to access the template and manipulate the view container.

TemplateRef represents the template associated with the element where the directive is applied.

ViewContainerRef represents the container that holds the view of the element where the directive is applied.

- Implement the desired behavior of your custom structural directive. For example, you can conditionally add or remove elements from the DOM based on a specific condition. You can use methods provided by TemplateRef and ViewContainerRef to manipulate the DOM structure:
  import { Directive, TemplateRef, ViewContainerRef } from '@angular/core';

```
@Directive({
 selector: '[appDirectiveName]'
})
export class DirectiveNameDirective {
 constructor(private templateRef: TemplateRef<any>, private viewContainerRef:
ViewContainerRef) { }

 // Example behavior: Conditionally render the template based on a boolean value
 @Input() set appDirectiveName(condition: boolean) {
  if (condition) {
   this.viewContainerRef.createEmbeddedView(this.templateRef);
  } else {
   this.viewContainerRef.clear();
  }
 }
}
```

- Use your custom structural directive in the template by applying it to the desired element and providing the necessary inputs:

```
<div *appDirectiveName="myCondition">...</div>
```

In the example, the appDirectiveName directive is applied to the element, and the myCondition property in the component is used to control the rendering behavior of the directive.

- Register the custom structural directive in an Angular module by importing and declaring it in the module's @NgModule decorator:

```
import { DirectiveNameDirective } from './directive-name.directive';

@NgModule({
  declarations: [DirectiveNameDirective],
  // ...
})
export class AppModule { }
```

That's it! You have created a custom structural directive in Angular. Now, when you use your custom directive in the template, it will modify the DOM structure based on the specified conditions and logic in the directive implementation.

Attribute Directives - ngStyle, ngClass:

In Angular, attribute directives allow you to manipulate the behavior and appearance of HTML elements by modifying their attributes. Angular provides several built-in attribute directives, including ngStyle and ngClass, which are commonly used to dynamically apply styles and classes to elements based on conditions or data values. Here's an overview of these attribute directives:

- **ngStyle**:

The ngStyle directive allows you to dynamically apply inline styles to HTML elements based on properties of the component. You can bind an object literal or a component property to the ngStyle directive.

Example:

```
<div [ngStyle]="{'color': textColor, 'font-size': fontSize + 'px'}">

  This text will be styled dynamically.

</div>
```

In the example, the ngStyle directive is used to apply the color and font-size styles to the element based on the component's textColor and fontSize properties.

- **ngClass**:

The ngClass directive allows you to conditionally apply CSS classes to HTML elements. You can bind an object literal, a component property that returns an object, or an array of CSS class names to the ngClass directive.

Example:

```
<div [ngClass]="{'active': isActive, 'highlight': shouldHighlight}">

  This div will have dynamically applied CSS classes.

</div>
```

In the example, the ngClass directive is used to conditionally apply the active and highlight CSS classes to the element based on the component's isActive and shouldHighlight properties.

Both ngStyle and ngClass directives support multiple conditions and complex expressions for dynamic styling and class application. You can combine them with other directives, event bindings, or template expressions to create rich and interactive user interfaces.

Note: Remember to import the necessary Angular modules in your component's module (@NgModule) for these directives to work properly. For ngStyle, you need to import the CommonModule, and for ngClass, you need to import the CommonModule or BrowserAnimationsModule (if using animations).

Custom Attribute Directive:

In Angular, you can create custom attribute directives to modify the behavior or appearance of HTML elements by manipulating their attributes. Custom attribute directives allow you to define custom logic that is applied when the directive is used on an element. Here's how you can create a custom attribute directive in Angular:

- Create a new directive using the Angular CLI command:

```
ng generate directive directive-name
```

This command generates the necessary files for the directive, including a TypeScript file and a test file.

- Open the generated TypeScript file (e.g., directive-name.directive.ts) and import the necessary modules and decorators:

```
import { Directive, ElementRef, Renderer2 } from '@angular/core';
```

- Decorate the directive class with the @Directive decorator, providing a selector to use the directive in the template:

```
@Directive({
  selector: '[appDirectiveName]'
})
export class DirectiveNameDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) { }
}
```

- In the directive class, inject ElementRef and Renderer2 into the constructor. These parameters allow you to access and manipulate the element on which the directive is applied.

ElementRef provides access to the underlying native element.

Renderer2 provides methods to manipulate the element's attributes and styles.

- Implement the desired behavior of your custom attribute directive. For example, you can add or remove attributes, modify styles, or attach event listeners:
  import { Directive, ElementRef, Renderer2 } from '@angular/core';

```
@Directive({
 selector: '[appDirectiveName]'
})
export class DirectiveNameDirective {
 constructor(private el: ElementRef, private renderer: Renderer2) {
  this.renderer.setStyle(this.el.nativeElement, 'color', 'red');
  this.renderer.addClass(this.el.nativeElement, 'custom-class');
 }
}
```

In the example, the appDirectiveName directive sets the color of the element to red and adds the custom-class CSS class to the element.

- Use your custom attribute directive in the template by applying it to the desired element:

```
<div appDirectiveName>
 This element is affected by the custom directive.
</div>
```

In the example, the appDirectiveName directive is applied to the element.

- Register the custom attribute directive in an Angular module by importing and declaring it in the module's @NgModule decorator:

```
import { DirectiveNameDirective } from './directive-name.directive';

@NgModule({
 declarations: [DirectiveNameDirective],
 // ...
})
export class AppModule { }
```

That's it! You have created a custom attribute directive in Angular. Now, when you use your custom directive in the template, it will modify the behavior or appearance of the element to which it is applied based on the logic defined in the directive implementation.

Property Binding:

Property binding is a feature in Angular that allows you to set the value of a property or attribute of an HTML element dynamically based on the data or expressions in your component. It establishes a unidirectional flow of data from the component to the template. With property binding, you can update properties, such as value, disabled, src, href, and many more, of HTML elements dynamically. Here's how you can use property binding in Angular:

- Surround the attribute or property value in square brackets ([]) within the template and assign it to a property or expression in the component.

<input [value]="myProperty">

In the example, the value of the myProperty property in the component is bound to the value property of the <input> element. Any changes to myProperty will be reflected in the input field.

- You can also bind to attributes other than the element's properties. Use attribute binding by prefixing the attribute name with attr. and bind it to a component property or expression.

<a [attr.href]="urlProperty">Go to Link

In this case, the value of the urlProperty property in the component is bound to the href attribute of the <a> element.

- Property binding can also be used with built-in directives like ngStyle and ngClass. You can bind properties to apply dynamic styles or classes to elements.

**<div [ngStyle]="{ 'color': textColor, 'font-size': fontSize + 'px' }"> **
**<div [ngClass]="{ 'active': isActive, 'highlight': shouldHighlight }"> **

In the examples, the textColor, fontSize, isActive, and shouldHighlight properties are bound to the styles and classes applied to the elements.

- You can bind to events as well using event binding syntax. Enclose the event name in parentheses and assign it to a method in the component.

<button (click)="handleClick()">Click Me</button>

In this case, the click event of the <button> element is bound to the handleClick() method in the component.

Property binding allows you to establish dynamic relationships between the component and the template, enabling you to update and manipulate the properties and attributes of HTML elements based on

component data or expressions. It is a powerful tool for creating dynamic and interactive user interfaces in Angular.

Attribute Binding:

Attribute binding is a feature in Angular that allows you to dynamically set the value of an HTML element's attribute based on the data or expressions in your component. Unlike property binding, attribute binding works with HTML attributes that do not have corresponding properties in the DOM API. Attribute binding uses square brackets ([]) to bind a component property or expression to an attribute value. Here's how you can use attribute binding in Angular:

- Surround the attribute name in square brackets ([]) within the template and assign it to a property or expression in the component.

```
<div [attr.data-id]="itemId"></div>
```

In the example, the value of the itemId property in the component is bound to the data-id attribute of the element.

- You can also use attribute binding to dynamically set standard HTML attributes like disabled, readonly, placeholder, etc.

```
<input [attr.disabled]="isDisabled">
```

In this case, the isDisabled property in the component is bound to the disabled attribute of the <input> element. If isDisabled is true, the input will be disabled; otherwise, it will be enabled.

- Attribute binding can be combined with other bindings, such as property binding and event binding, to create more dynamic interactions in your templates.

```
<button [attr.data-action]="action" (click)="handleClick()">Click Me</button>
```

In this example, the action property in the component is bound to the data-action attribute of the <button> element, and the handleClick() method is bound to the click event of the button.

- Attribute binding can also be used with built-in directives like ngClass and ngStyle to dynamically apply classes or styles based on component properties or expressions.

```
<div [ngClass]="{ 'active': isActive, 'highlight': shouldHighlight }"></div>
<div [ngStyle]="{ 'color': textColor, 'font-size': fontSize + 'px' }"></div>
```

In these examples, the isActive, shouldHighlight, textColor, and fontSize properties are bound to the attribute values used by ngClass and ngStyle directives.

Attribute binding allows you to set and manipulate HTML attributes that may not have corresponding properties in the DOM API. It provides a way to dynamically control the attributes of HTML elements based

on the data and expressions in your component, enabling you to create more flexible and interactive templates in Angular.

Style and Event Binding:

Style Binding

Style binding is a feature in Angular that allows you to dynamically set CSS styles of HTML elements based on the data or expressions in your component. It provides a way to apply dynamic styles to elements, such as changing colors, font sizes, visibility, and more. Style binding uses square brackets ([]) to bind a component property or expression to a style property. Here's how you can use style binding in Angular:

- Surround the style property in square brackets ([]) within the template and assign it to a property or expression in the component.

```
<div [style.color]="textColor"></div>
```

In the example, the textColor property in the component is bound to the color style property of the element. The color of the will be dynamically updated based on the value of textColor.

- You can also use style binding to apply multiple styles by binding to an object literal containing style properties and their values.

```
<div [style]="{ 'color': textColor, 'font-size': fontSize + 'px' }"></div>
```

In this case, the textColor and fontSize properties in the component are bound to the color and font-size styles of the element, respectively.

- Style binding can be combined with other bindings, such as attribute binding and event binding, to create more dynamic and interactive styling in your templates.

Event Binding:

Event binding is a feature in Angular that allows you to respond to user interactions, such as button clicks, mouse movements, and key presses, by executing methods in your component. It establishes a connection between the template and the component, enabling you to handle user actions. Event binding uses parentheses (()) to bind an event of an HTML element to a method in the component. Here's how you can use event binding in Angular:

- Enclose the event name in parentheses (()) within the template and assign it to a method in the component.

```
<button (click)="handleClick()">Click Me</button>
```

In the example, the click event of the <button> element is bound to the handleClick() method in the component. When the button is clicked, the method will be executed.

- You can also pass event objects or data to the method by using the $event keyword.

<input (keyup)="handleKeyUp($event)">

In this case, the keyup event of the <input> element is bound to the handleKeyUp() method in the component, passing the event object to the method.

- Event binding can be used with various events, such as click, keyup, mouseover, submit, and many more, to capture and respond to user interactions in your application.

Both style binding and event binding provide powerful ways to create dynamic and interactive user interfaces in Angular. Style binding allows you to apply dynamic styles to elements based on component data or expressions, while event binding enables you to respond to user interactions by executing methods in your component. These features together help you create engaging and responsive applications.

Built in Pipes:

Angular provides a set of built-in pipes that you can use to transform and format data in your templates. Pipes are simple functions that accept an input value and optional parameters, and they return a transformed value. Here are some commonly used built-in pipes in Angular:

DatePipe:

The DatePipe is used for formatting dates. It supports various date formats and options for displaying dates and times.

Example:

** {{ currentDate | date: 'short' }} **

In this example, the currentDate property is formatted using the DatePipe with the 'short' format, which displays the date and time in a short format.

DecimalPipe:

The DecimalPipe is used for formatting numbers with decimal places, thousands separators, and currency symbols.

Example:

** {{ price | number: '1.2-2' }} **

In this example, the price property is formatted using the DecimalPipe with the '1.2-2' format, which displays the number with two decimal places and optional thousands separators.

CurrencyPipe:

The CurrencyPipe is used for formatting currency values with the specified currency code and symbol.

Example:

```
<p>{{ amount | currency: 'USD' }}</p>
```

In this example, the amount property is formatted using the CurrencyPipe with the 'USD' currency code, which displays the value with the USD currency symbol.

PercentPipe:

The PercentPipe is used for formatting numbers as percentages.

Example:

```
<p>{{ discount | percent }}</p>
```

In this example, the discount property is formatted using the PercentPipe, which displays the value as a percentage.

SlicePipe:

The SlicePipe is used for slicing arrays or strings and returning a subset of the original data.

Example:

```
<ul>
  <li *ngFor="let item of items | slice:0:3">{{ item }}</li>
</ul>
```

In this example, the SlicePipe is used within an ngFor directive to display the first three items from the items array.

These are just a few examples of the built-in pipes available in Angular. There are many more, including UpperCasePipe, LowerCasePipe, AsyncPipe, and JsonPipe, among others. Pipes provide a convenient way to transform and format data in your templates without modifying the underlying component properties. You can also chain multiple pipes together to perform complex data transformations.

Passing Parameters to Pipes:

In Angular, you can pass parameters to built-in pipes and custom pipes to customize their behavior. This allows you to apply specific transformations or formatting based on dynamic values. Here's how you can pass parameters to pipes:

Built-in Pipes:

When using a built-in pipe, you can pass parameters by separating them with colons (:) within the pipe syntax.

Example:

```
<p>{{ currentDate | date: 'fullDate' }}</p>
```

In this example, the date pipe is used with the 'fullDate' parameter, which formats the currentDate property as a full date.

Custom Pipes:

When using a custom pipe, you need to define the parameter(s) within the pipe implementation and pass the value(s) when using the pipe in the template.

Step 1: Create a custom pipe using the Angular CLI:

```
ng generate pipe myCustomPipe
```

Step 2: Open the generated pipe file (e.g., my-custom-pipe.pipe.ts) and define the parameter(s) in the transform method.

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


@Pipe({
  name: 'myCustomPipe'
})
export class MyCustomPipe implements PipeTransform {
  transform(value: any, param1: any, param2: any): any {
    // Custom transformation logic using the value and parameters
```

```
    return transformedValue;
  }
}
```

Step 3: Use the custom pipe in the template and pass the parameter(s) as arguments.

```
<p>{{ someValue | myCustomPipe: param1: param2 }}</p>
```

In this example, the someValue property is transformed using the myCustomPipe, and the param1 and param2 values are passed as parameters to the pipe.

Note: You can have multiple parameters in a custom pipe by separating them with colons (:) within the pipe syntax.

By passing parameters to pipes, you can customize their behavior and perform specific transformations or formatting based on dynamic values. This flexibility allows you to create reusable and adaptable pipes that can be used in different scenarios within your Angular application.

Nested Components Basics:

In Angular, nested components refer to the concept of creating a component hierarchy, where one component is used within another component. This allows for better code organization, reusability, and separation of concerns. To understand nested components in Angular, let's go through the basics:

- **Component Creation**:

To create a new component in Angular, you can use the Angular CLI (Command Line Interface) by running the command ng generate component component-name.

This command will generate the necessary files for the component, including a TypeScript file (.ts), an HTML template file (.html), a CSS file (.css), and a spec file (.spec.ts) for testing.

- **Parent-Child Relationship**:

In Angular, components can have a parent-child relationship, where one component acts as the parent and another as the child.

The parent component can contain the child component in its template and provide data to the child component.

- **Component Communication**:

To communicate between parent and child components, Angular provides two mechanisms: input properties and output events.

Input properties allow the parent component to pass data to the child component. The child component uses the @Input decorator to define an input property.

Output events allow the child component to emit events to the parent component. The child component uses the @Output decorator along with an EventEmitter to define an output event.

- **Using Nested Components**:

To use a nested component, you need to include its selector in the template of the parent component.

For example, if you have a parent component called ParentComponent and a child component called ChildComponent, you can include the child component in the parent's template like this: <app-child></app-child>, assuming the selector of the child component is app-child.

- **Passing Data to Child Components**:

To pass data from the parent component to the child component, you can use input properties.

In the parent component's template, you can bind a property to the child component's input property using square brackets ([]).

For example, if the child component has an input property called data, you can pass data from the parent component like this: <app-child [data]="parentData"></app-child>, where parentData is a property in the parent component.

- **Emitting Events from Child Components**:

To emit events from the child component to the parent component, you can use output events.

In the child component's TypeScript file, you define an output event using the @Output decorator and an EventEmitter. Then, you can emit the event using the emit() method.

In the parent component's template, you can listen to the child component's output event using parentheses (()).

For example, if the child component has an output event called childEvent, you can listen to it in the parent component like this: <app-child (childEvent)="handleEvent($event)"></app-child>, where handleEvent() is a method in the parent component.

By using these techniques, you can create a hierarchy of nested components in Angular, allowing for a modular and reusable application structure.

Passing data from Container Component to Child Component:

To pass data from a container component (parent) to a child component in Angular, you can use input properties. Input properties allow the parent component to bind data to the child component. Here's how you can do it:

- **Define an Input Property in the Child Component**:

In the child component's TypeScript file, declare an input property using the @Input decorator. This property will receive the data from the parent component.

For example, in the child component's TypeScript file:

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  @Input() data: string; // Define an input property called 'data'
}
```

- **Bind the Property in the Parent Component's Template**:

In the parent component's HTML template, bind the desired property to the child component's input property using square brackets ([]).

For example, in the parent component's template:

```
<app-child [data]="parentData"></app-child>
```

In this example, parentData is a property in the parent component that you want to pass to the child component's data input property.

- **Pass the Data from the Parent Component**:

In the parent component's TypeScript file, define the parentData property and assign the desired value to it.

For example, in the parent component's TypeScript file:

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

@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html',
  styleUrls: ['./parent.component.css']
})
export class ParentComponent {
  parentData: string = 'Hello from parent component!';
}
```

- **Access the Data in the Child Component**:

In the child component, you can access the data received from the parent component via the input property (data in the example).

For example, in the child component's HTML template:

```
<p>{{ data }}</p> <!-- Display the value of the 'data' property -->
```

By following these steps, you can pass data from a container component to a child component in Angular using input properties. The child component will receive the data and can use it as needed in its template or logic.

Passing data from Child Component to Container Component:

To pass data from a child component to a container component (parent) in Angular, you can use output events. Output events allow the child component to emit events that the parent component can listen to and react accordingly. Here's how you can do it:

- **Define an Output Event in the Child Component**:

In the child component's TypeScript file, declare an output event using the @Output decorator and an instance of EventEmitter.

For example, in the child component's TypeScript file:

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

@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent {
  @Output() childEvent: EventEmitter<string> = new EventEmitter<string>(); // Define an output event called 'childEvent'
  sendDataToParent() {
    const dataToSend = 'Hello from child component!';
    this.childEvent.emit(dataToSend); // Emit the event with the data
  }
}
```

- **Emit the Event from the Child Component**:

In the child component's TypeScript file, create a method that will emit the output event when triggered. Use the emit() method on the childEvent EventEmitter instance to emit the event and pass the desired data.

For example, in the child component's TypeScript file:

```
sendDataToParent() {
  const dataToSend = 'Hello from child component!';
```

```
    this.childEvent.emit(dataToSend); // Emit the event with the data
  }
```

- **Handle the Event in the Container Component**:

In the container component's HTML template, listen to the child component's output event using parentheses (()). Assign a handler method in the container component's TypeScript file to handle the emitted event.

For example, in the container component's template:

```
<app-child (childEvent)="handleChildEvent($event)"></app-child>
```

- **Define the Event Handler in the Container Component**:

In the container component's TypeScript file, define the event handler method that will receive the emitted event from the child component. This method can then access the emitted data and perform any necessary actions.

For example, in the container component's TypeScript file:

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

@Component({
  selector: 'app-container',
  templateUrl: './container.component.html',
  styleUrls: ['./container.component.css']
})
export class ContainerComponent {
  handleChildEvent(data: string) {
    console.log('Received data from child component:', data);
    // Perform actions with the received data
  }
}
```

By following these steps, you can pass data from a child component to a container component in Angular using output events. The child component emits an event with the desired data, and the container component listens to the event and handles it accordingly in the event handler method.

Shadow DOM:

In Angular, the concept of Shadow DOM refers to the encapsulation of styles and DOM elements within a component. It allows for scoping styles and isolating the component's structure and behavior from the rest of the application. Angular provides native support for Shadow DOM through the View Encapsulation feature. Here's how Shadow DOM works in Angular:

- **View Encapsulation Modes**:

Angular provides three view encapsulation modes: Emulated (default), Native, and None.

Emulated: This mode is the default and emulates Shadow DOM behavior by encapsulating component styles and DOM elements within the component's template. It uses CSS selectors to apply component-specific styles. This mode provides style scoping but doesn't use the browser's native Shadow DOM implementation.

Native: This mode leverages the browser's native Shadow DOM support. It encapsulates the component's template and styles within a Shadow DOM subtree, isolating it from the external DOM. It provides true style encapsulation and prevents style leaking from the component to other parts of the application.

None: This mode disables view encapsulation completely. Component styles are not scoped to the component's template, and they can affect the entire application. Use this mode cautiously as it can lead to style conflicts and reduced encapsulation benefits.

- **Setting View Encapsulation Mode**:

By default, Angular uses the Emulated view encapsulation mode. However, you can specify a different mode for a component by setting the encapsulation property in the component's metadata.

For example, to use the Native view encapsulation mode, you would specify it in the component's metadata:

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

@Component({

 selector: 'app-example',

 templateUrl: './example.component.html',
```

```
  styleUrls: ['./example.component.css'],
  encapsulation: ViewEncapsulation.Native
})
export class ExampleComponent {
  // Component logic
}
```

- **Style Encapsulation**:

With view encapsulation, component styles defined in the component's CSS file or inline styles are scoped to the component's template.

In the default Emulated mode, Angular adds a unique attribute to the component's host element, and the component's styles are transformed with the attribute selector to apply only to that component.

In the Native mode, the component's styles are encapsulated within the Shadow DOM subtree, and they don't leak out or affect other parts of the application.

Using view encapsulation with Shadow DOM in Angular provides several benefits, including:

Style scoping: Component styles are scoped to the component's template, preventing unintended style conflicts with other components.

Code organization: Styles and template structure are encapsulated within the component, improving code modularity and maintainability.

Reusability: Components with encapsulated styles can be easily reused without worrying about style interference.

Isolation: DOM elements and behavior within the component are isolated, minimizing unintended side effects on other parts of the application.

Note that while Angular supports Shadow DOM through the Native view encapsulation mode, it also provides a robust emulation of Shadow DOM with the default Emulated mode, allowing for consistent behavior across browsers.

Component Life Cycle:

In Angular, components have a lifecycle that consists of various stages or hooks that are executed at different points during the component's creation, rendering, and destruction. These lifecycle hooks allow you to perform actions at specific moments in the component's lifecycle. Here are the main lifecycle hooks available in Angular:

ngOnChanges:

This hook is called when one or more of the component's input properties change.

It receives a SimpleChanges object that contains information about the previous and current values of the input properties.

Example:

```
ngOnChanges(changes: SimpleChanges) {

 // Perform actions based on input property changes

}
```

ngOnInit:

This hook is called once, after the component and its child components have been initialized and their input properties have been set.

It is commonly used to initialize data, make API calls, or perform other setup tasks.

Example:

```
ngOnInit() {

 // Perform initialization tasks

}
```

ngDoCheck:

This hook is called during every change detection cycle of the component.

It is used to detect and respond to changes that Angular cannot detect automatically, such as changes to properties of an object or changes in complex data structures.

Example:

```
ngDoCheck() {
  // Perform custom change detection logic
}
```

ngAfterContentInit:

This hook is called after the component's content (e.g., child components, projected content) has been initialized.

It is commonly used to interact with the component's content or child components after they have been created.

Example:

```
ngAfterContentInit() {
  // Perform actions after content initialization
}
```

ngAfterContentChecked:

This hook is called after the component's content has been checked for changes.

It is called after every change detection cycle and can be used to perform additional actions based on the component's content.

Example:

```
ngAfterContentChecked() {
  // Perform actions after content has been checked
}
```

ngAfterViewInit:

This hook is called after the component's view (DOM) has been initialized.

It is commonly used to interact with the component's view or perform actions that require access to the rendered DOM.

Example:

```
ngAfterViewInit() {
  // Perform actions after view initialization
}
```

ngAfterViewChecked:

This hook is called after the component's view has been checked for changes.

It is called after every change detection cycle and can be used to perform additional actions based on the component's view.

Example:

```
ngAfterViewChecked() {
  // Perform actions after view has been checked
}
```

ngOnDestroy:

This hook is called just before the component is destroyed and removed from the DOM.

It is commonly used to clean up resources, unsubscribe from observables, or perform other cleanup tasks.

Example:

```
ngOnDestroy() {
  // Perform cleanup tasks before component destruction
}
```

These lifecycle hooks allow you to control the behavior and perform actions at different stages of a component's lifecycle. By leveraging these hooks, you can initialize data, interact with the DOM, handle changes, and clean up resources when necessary.

Template Driven Forms: In Angular, Template-driven forms provide an approach to creating forms using Angular's template syntax. With Template-driven forms, the form controls and validation logic are

primarily defined in the component's template rather than in the component class. Here's an overview of how to use Template-driven forms in Angular:

- **Import FormsModule**:

To use Template-driven forms, you need to import the FormsModule from @angular/forms in your application module (e.g., app.module.ts).

Example:

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

@NgModule({
  imports: [
    FormsModule
  ],
  // Other module configurations
})
export class AppModule { }
```

Create a Form in the Template:

In the component's template, define the form using the <form> element and add form controls using various input elements such as <input>, <select>, and <textarea>.

Use the ngModel directive to bind form controls to properties in the component class for two-way data binding.

Example:

```
<form (ngSubmit)="onSubmit()">
  <div>
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" [(ngModel)]="user.name" required>
```

```
    </div>
    <div>
      <label for="email">Email:</label>
      <input type="email" id="email" name="email" [(ngModel)]="user.email" required>
    </div>
    <button type="submit">Submit</button>
  </form>
```

- **Handle Form Submission**:

Add an event handler method in the component class to handle the form submission. In the example above, we bind the (ngSubmit) event of the <form> element to the onSubmit() method.

Example:

```
export class MyComponent {
  user: any = {};
  onSubmit() {
    // Form submission logic
  }
}
```

Form Validation:

Use HTML5 validation attributes such as required, minlength, maxlength, etc., to specify validation rules for form controls.

Angular automatically updates the form's validity and provides visual feedback such as error messages based on these attributes.

Example:

```
<div>
```

```
 <label for="password">Password:</label>

 <input type="password" id="password" name="password" [(ngModel)]="user.password" required minlength="8">

 <div *ngIf="password.invalid && (password.dirty || password.touched)">

  <div *ngIf="password.errors.required">Password is required.</div>

  <div *ngIf="password.errors.minlength">Password should have at least 8 characters.</div>

 </div>

</div>
```

Template-driven forms provide a simpler approach for creating forms in Angular, especially for simple and straightforward forms. However, they may have limitations when it comes to complex form scenarios and custom form validation. For more advanced form requirements, Reactive Forms offer more flexibility and control.

Model Driven Forms or Reactive Forms:

In Angular, Reactive Forms (also known as Model-driven Forms) provide a more flexible and powerful approach for building forms compared to Template-driven forms. Reactive Forms are based on the Reactive Programming paradigm and allow you to define form controls and validation logic programmatically in the component class. Here's an overview of how to use Reactive Forms in Angular:

- **Import ReactiveFormsModule**:

To use Reactive Forms, you need to import the ReactiveFormsModule from @angular/forms in your application module (e.g., app.module.ts).

Example:

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

@NgModule({

 imports: [

  ReactiveFormsModule

 ],
```

```
  // Other module configurations
})
export class AppModule { }
```

- **Create a Form in the Component**:

In the component class, import the necessary form-related classes from @angular/forms, such as FormGroup, FormControl, and Validators.

Define a FormGroup instance to represent the entire form and create FormControl instances for each form control.

Assign the form controls to the form group using key-value pairs.

Example:

```
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';
@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.css']
})
export class MyComponent {
  myForm: FormGroup;
  constructor() {
    this.myForm = new FormGroup({
      name: new FormControl('', Validators.required),
      email: new FormControl('', [Validators.required, Validators.email]),
      password: new FormControl('', [Validators.required, Validators.minLength(8)])
```

```
    });
  }
  onSubmit() {
    // Form submission logic
  }
}
```

- **Link Form Controls to the Template**:

In the component's template, bind the form controls to the corresponding input elements using the formControlName directive.

Example:

```
<form [formGroup]="myForm" (ngSubmit)="onSubmit()">
 <div>
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" formControlName="name">
 </div>
 <div>
  <label for="email">Email:</label>
  <input type="email" id="email" name="email" formControlName="email">
 </div>
 <div>
  <label for="password">Password:</label>
  <input type="password" id="password" name="password" formControlName="password">
 </div>
```

```
  <button type="submit" [disabled]="myForm.invalid">Submit</button>

</form>
```

- **Form Validation and Error Handling**:

Use the Validators class from @angular/forms to define validation rules for form controls.

You can access the form controls in the component class and check their validity, errors, and other properties.

Example:

```
onSubmit() {
  if (this.myForm.valid) {
    // Form submission logic
  } else {
    // Handle form errors or display validation messages
  }
}
get name() {
  return this.myForm.get('name');
}
get email() {
  return this.myForm.get('email');
}
// ...
```

Reactive Forms provide advanced features such as dynamic form control manipulation, custom validators, cross-field validation, and more. They offer greater control and flexibility compared to Template-driven.

Custom Validators in Reactive Forms:

In Angular, you can create custom validators to implement custom validation logic for your Reactive Forms. Custom validators are functions that take a form control as an input and return an object with validation errors if the validation fails, or null if the validation is successful. Here's how you can create custom validators in Angular:

- **Create a Custom Validator Function**:

Define a function that takes a form control as an argument and returns an object with validation errors or null.

The function should have the following signature: (control: AbstractControl): ValidationErrors | null.

Example:

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

function customValidator(control: AbstractControl): ValidationErrors | null {

 // Perform custom validation logic

 if (/* validation fails */) {

  return { customError: true };

 }

 return null; // Validation successful

}
```

- **Register the Custom Validator**:

You can register the custom validator function at the form control level or at the form group level, depending on your requirements.

To register the custom validator at the form control level, you can pass the validator function as the second argument to the FormControl constructor or use the setValidators() method to add it later.

Example:

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

```
const myControl = new FormControl('', customValidator);

// or

myControl.setValidators(customValidator);
```

- **Using the Custom Validator in Template and Component**:

In the template, you can access the validation errors returned by the custom validator using the errors property of the form control.

In the component class, you can access the form control and check its validity, errors, and other properties.

Example:

```
<input type="text" id="myInput" name="myInput" formControlName="myControl">

<div *ngIf="myControl.errors?.customError">Custom validation failed.</div>
```

By creating custom validators, you can implement your own validation rules and apply them to form controls in your Reactive Forms. These validators can be used alongside built-in validators provided by Angular's Validators class, giving you the flexibility to enforce complex validation requirements for your forms.

Custom Validators in Template Driven forms:

In Angular, you can also create custom validators for Template-driven forms. Custom validators in Template-driven forms are functions that you can define in the component class and use in the template to perform custom validation logic. Here's how you can create custom validators in Template-driven forms:

- **Create a Custom Validator Function**:

Define a function in the component class that takes a form control or an abstract control as an argument and returns an object with validation errors if the validation fails, or null if the validation is successful.

The function should have the following signature: (control: AbstractControl): ValidationErrors | null.

Example:

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

function customValidator(control: AbstractControl): ValidationErrors | null {

  // Perform custom validation logic
```

```
    if (/* validation fails */) {
      return { customError: true };
    }
    return null; // Validation successful
  }
```

- **Use the Custom Validator in the Template**:

In the template, you can apply the custom validator by adding it to the ngModel directive of the form control using the validators property.

Example:

```
<input type="text" id="myInput" name="myInput" [(ngModel)]="myModel" #myInput="ngModel" [validators]="customValidator">

<div *ngIf="myInput.errors?.customError">Custom validation failed.</div>
```

- **Handling the Validation Result**:

You can access the validation errors returned by the custom validator using the errors property of the form control.

In the example above, we check if the customError validation error exists on the myInput form control and display an error message if it does.

Note that the form control should have the ngModel directive and a reference variable (#myInput="ngModel") for accessing the validation errors.

Custom validators in Template-driven forms provide a way to implement custom validation logic directly in the template without the need for explicit form control creation in the component class. However, if you have more complex validation requirements or need to perform validation logic in the component class, Reactive Forms with custom validators might be a better option.

Dependency Injection:

Dependency Injection (DI) is a core feature of the Angular framework that helps manage the dependencies between different components and services. DI allows you to declare the dependencies of a class or component in a way that the framework can automatically resolve and provide those dependencies when

the class or component is instantiated. This approach promotes modularity, testability, and reusability of code.

In Angular, there are three main participants in the dependency injection process:

- **Injectable Classes**:

Injectable classes are classes that have dependencies and can be injected with the required dependencies.

To make a class injectable, you can decorate it with the @Injectable() decorator. This decorator marks the class as a provider of services or dependencies.

Example:

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

@Injectable()

export class MyService {

  // Class implementation

}
```

- **Providers**:

Providers are responsible for creating and managing instances of injectable classes.

In Angular, providers are typically defined at the module level or component level.

Providers can be registered in the providers array of an Angular module or in the providers metadata property of a component.

Example:

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

import { MyService } from './my.service';

@NgModule({

  providers: [MyService]

})
```

```
export class AppModule { }
```

- **Dependency Injection**:

Angular's DI system takes care of injecting dependencies into classes or components that declare their dependencies.

To inject a dependency, you can declare a constructor parameter with the type of the dependency.

The Angular DI system will automatically resolve and provide an instance of the dependency when creating an instance of the class.

Example:

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

import { MyService } from './my.service';

@Component({

 selector: 'app-my-component',

 template: '...',

 providers: [MyService]

})

export class MyComponent {

 constructor(private myService: MyService) {

  // Use myService instance

 }

}
```

The Angular DI system uses hierarchical injection, meaning that dependencies can be injected at the component level, and if not found, it will check the parent components and providers defined at the module level. This allows for flexibility in managing dependencies across the application.

By utilizing dependency injection, you can easily manage the dependencies of your components and services, decouple code modules, promote code reusability, and enable better testing and maintenance of your Angular applications.

Services Basics:

In Angular, services are used to encapsulate and provide functionality that can be shared across multiple components. Services act as a centralized place to manage data, perform business logic, and interact with external resources such as APIs or databases. Here are the basics of using services in Angular:

- **Creating a Service**:

To create a service in Angular, you can generate a new service file using the Angular CLI command: ng generate service my-service.

Alternatively, you can manually create a new TypeScript file for your service and define a class that provides the desired functionality.

Example:

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

@Injectable({

  providedIn: 'root'

})

export class MyService {

  // Service functionality and data

}
```

- **Providing the Service**:

By default, Angular services are registered with the root injector, making them available to the entire application.

The providedIn metadata option set to 'root' in the @Injectable() decorator ensures that the service is automatically provided at the root level.

You can also provide the service at a specific module or component level by adding it to the providers array of the respective module or component metadata.

Example:

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

import { MyService } from './my.service';

@NgModule({

 providers: [MyService]

})

export class MyModule { }
```

- **Consuming the Service**:

To use a service in a component, you need to inject it into the component's constructor.

Angular's dependency injection system will automatically provide an instance of the service when creating the component.

You can then access the service instance and use its methods and properties within the component.

Example:

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

import { MyService } from './my.service';

@Component({

 selector: 'app-my-component',

 template: '...',

 providers: [MyService]

})

export class MyComponent {

 constructor(private myService: MyService) {

  // Use myService instance
```

```
  }
}
```

Services are commonly used for tasks such as fetching data from APIs, performing CRUD operations, sharing data between components, implementing authentication and authorization, and more. They help in separating concerns and keeping your components lean and focused on their specific responsibilities.

Remember to provide services at the appropriate level (root, module, or component) based on your application's needs. By utilizing services effectively, you can create more modular, reusable, and maintainable Angular applications.

RxJS Observables:

RxJS (Reactive Extensions for JavaScript) is a powerful library for reactive programming in JavaScript. In Angular, RxJS is extensively used for handling asynchronous operations, such as handling HTTP requests, managing data streams, and event handling. RxJS provides the Observable pattern, which is a core concept in reactive programming. Here's an overview of using RxJS Observables in Angular:

- **Importing RxJS**:

To use RxJS in Angular, you need to import the necessary operators and classes from the rxjs package.

The most commonly used import is Observable which represents a stream of values over time.

Example:

```
import { Observable } from 'rxjs';
```

- **Creating Observables**:

You can create an Observable using the Observable constructor or by using helper methods such as of, from, interval, etc.

An Observable emits values over time, and you can subscribe to it to receive those values.

Example:

```
const myObservable = new Observable(observer => {
  observer.next('Hello');
  observer.next('World');
```

```
  observer.complete();
});
myObservable.subscribe(value => {
  console.log(value);
});
```

- **Operators**:

RxJS provides a wide range of operators that allow you to transform, filter, combine, and manipulate data streams emitted by Observables.

Operators like map, filter, mergeMap, tap, etc., enable you to perform various operations on emitted values.

You can chain multiple operators together to create complex data transformation pipelines.

Example:

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

const numbersObservable = of(1, 2, 3, 4, 5);

numbersObservable
  .pipe(
    filter(num => num % 2 === 0),
    map(num => num * 2)
  )
  .subscribe(result => {
    console.log(result); // Output: 4, 8
```

```
});
```

- **Subscribing to Observables**:

To receive values emitted by an Observable, you need to subscribe to it.

The subscribe method takes one or more callback functions as arguments to handle the emitted values, errors, and completion.

Example:

```
const myObservable = of('Hello', 'World');

myObservable.subscribe(
  value => {
    console.log(value); // Output: Hello, World
  },
  error => {
    console.error(error);
  },
  () => {
    console.log('Observable completed');
  }
);
```

- **Unsubscribing from Observables**:

When subscribing to an Observable, the subscription object is returned. You can use this object to unsubscribe and stop receiving further values.

Unsubscribing is important to prevent memory leaks and unnecessary processing when the component or subscription is no longer needed.

Example:

```
import { Subscription } from 'rxjs';

const myObservable = /* create Observable */;

const subscription: Subscription = myObservable.subscribe(/* handle values */);

// Unsubscribe when no longer needed

subscription.unsubscribe();
```

RxJS Observables provide a powerful way to handle asynchronous operations, compose data transformations, and manage data streams in Angular applications. Understanding the basics of Observables and operators can greatly enhance your ability to handle reactive programming scenarios in Angular.

Server Communication using HttpClient:

In Angular, you can communicate with a server using the built-in HttpClient module, which provides methods to send HTTP requests and handle the server's responses. Here's an overview of how to perform server communication using HttpClient in Angular:

- **Import the HttpClient Module**:

To use HttpClient, you need to import the HttpClientModule in your Angular module.

Example:

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

@NgModule({

 imports: [

  HttpClientModule

 ]

})

export class AppModule { }
```

- **Inject and Use HttpClient**:

In the component or service where you want to perform server communication, inject the HttpClient service by declaring it as a dependency in the constructor.

The HttpClient service provides methods like get, post, put, delete, etc., for making HTTP requests.

Example:

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

constructor(private http: HttpClient) { }
```

- **Sending GET Requests**:

To send a GET request, use the get method of the HttpClient service, specifying the URL you want to request.

The get method returns an Observable that emits the server's response.

Example:

```
this.http.get('/api/data').subscribe(response => {
  console.log(response);
});
```

- **Sending POST Requests**:

To send a POST request, use the post method of the HttpClient service, specifying the URL and the data you want to send.

The post method also returns an Observable that emits the server's response.

Example:

```
const data = { name: 'John', age: 30 };

this.http.post('/api/data', data).subscribe(response => {
  console.log(response);
});
```

- **Handling Responses**:

The responses from the server are typically in JSON format, and HttpClient automatically parses them into JavaScript objects by default.

You can also specify the response type explicitly using generics (<T>) when calling the HTTP methods to receive a typed response.

Example:

```
this.http.get<User>('/api/users/1').subscribe(user => {
  console.log(user.name);
});
```

- **Error Handling**:

You can handle errors in the subscribe method by providing an error callback.

The error callback will be triggered if there is an error in the HTTP request or if the server responds with an error status code.

Example:

```
this.http.get('/api/data').subscribe(
  response => {
    console.log(response);
  },
  error => {
    console.error(error);
  }
);
```

By using HttpClient in Angular, you can easily perform HTTP requests and handle server responses in a concise and efficient manner. The HttpClient module also provides additional features like request headers, request options, interceptors, and more, allowing you to customize and enhance your server communication logic.

Communicating with different backend services using Angular HttpClient:

In Angular, you can use the HttpClient module to communicate with different backend services, such as RESTful APIs, GraphQL APIs, WebSocket servers, and more. The HttpClient provides methods for making HTTP requests to various endpoints and handling the server's responses. Here's an overview of how to communicate with different backend services using HttpClient in Angular:

- **RESTful APIs**:

For communicating with RESTful APIs, you can use the HttpClient methods like get, post, put, delete, etc.

Make sure to provide the appropriate URL and data as needed by the API.

Example:

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

constructor(private http: HttpClient) { }

// GET request to retrieve data

this.http.get('/api/data').subscribe(response => {

  console.log(response);

});

// POST request to send data

const data = { name: 'John', age: 30 };

this.http.post('/api/users', data).subscribe(response => {

  console.log(response);

});
```

- **GraphQL APIs**:

To communicate with GraphQL APIs, you can use the HttpClient to send POST requests with the GraphQL query or mutation.

Make sure to provide the appropriate GraphQL endpoint URL and the GraphQL request body containing the query/mutation.

Example:

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

constructor(private http: HttpClient) { }

const query = `
  query {
    getUser(id: 1) {
      name
      email
    }
  }
`;

this.http.post('/graphql', { query }).subscribe(response => {
  console.log(response);
});
```

WebSocket Servers:

To communicate with WebSocket servers, the HttpClient module does not provide direct support. Instead, you can use the WebSocket object from the browser's standard API or utilize third-party WebSocket libraries like rxjs/webSocket.

Example using the rxjs/webSocket library:

```
import { webSocket } from 'rxjs/webSocket';

const socket = webSocket('ws://localhost:8080');

socket.subscribe(
  message => {
    console.log(message);
```

```
  },
  error => {
    console.error(error);
  },
  () => {
    console.log('WebSocket connection closed');
  }
);
```

- **Other Backend Services**:

Depending on the backend service you want to communicate with, you might need to use specific protocols or libraries.

For example, if you are working with a server-sent events (SSE) server, you can use the EventSource API provided by the browser.

If you are working with a socket.io server, you can utilize the ngx-socket-io library, which provides Angular-friendly wrappers for socket.io functionality.

Always refer to the documentation or specifications of the specific backend service you are using to determine the appropriate approach for communication.

By leveraging the HttpClient module in Angular, you can easily communicate with different backend services by making appropriate HTTP requests and handling the responses. Adjust your code according to the specific requirements and protocols of the backend service you are working with.

Routing Basics:

In Angular, routing is a mechanism for navigating between different views or components in a single-page application (SPA). It allows you to define routes for different URLs and map them to corresponding components. Here's an overview of routing basics in Angular:

- **Set up the Router**:

To use routing in your Angular application, you need to import the RouterModule and Routes modules from @angular/router in your Angular module.

Example:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
const routes: Routes = [
  // Define your routes here
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

- **Define Routes**:

In the routes array, you define your application's routes using the path and component properties.

The path property represents the URL path, and the component property specifies the component to render for that URL.

Example:

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'products', component: ProductsComponent },
  { path: 'products/:id', component: ProductDetailComponent },
  { path: '**', component: NotFoundComponent } // Wildcard route for handling unknown URLs
];
```

- **Add the Router Outlet**:

In your application's main template file (usually app.component.html), add the <router-outlet></router-outlet> tag.

The <router-outlet> tag acts as a placeholder where the routed components will be rendered.

Example:

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

Navigation:

To navigate to a specific route or URL, you can use the Router service and its navigate method.

Provide the target route path as a parameter to the navigate method.

Example:

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

constructor(private router: Router) { }

// Navigating programmatically

this.router.navigate(['/about']);

// Navigating with parameters

this.router.navigate(['/products', productId]);
```

- **Accessing Route Parameters**:

You can define route parameters in the route path by prefixing a segment with a colon (:).

To access the route parameters in a component, inject the ActivatedRoute service and access the params Observable.

Example:

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

constructor(private route: ActivatedRoute) { }

// Accessing route parameters
```

```
this.route.params.subscribe(params => {
  const productId = params['id'];
  // Do something with the productId
});
```

- **Route Guards**:

Angular provides route guards that allow you to control access to routes based on certain conditions.

Route guards include CanActivate, CanActivateChild, CanDeactivate, Resolve, etc.

You can implement custom route guards by creating a class that implements the corresponding guard interface.

Example:

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    // Check authentication status or any other conditions
    // Return true to allow access, or navigate to another route using UrlTree
    // Return false to deny access
```

```
  return true;
 }
}
```

Routing in Angular enables you to navigate between different components based on URL paths. By defining routes, using the <router-outlet> tag, and leveraging the Router service, you can implement navigation within your Angular application. Additionally, route parameters and route guards provide additional flexibility and control over your application's routing behavior.

Router Links:

In Angular, router links are used to navigate between different routes within your application. They provide a declarative way to create links that trigger route navigation. Here's an overview of how to use router links in Angular:

- **Import the Router Module**:

To use router links, you need to import the RouterModule module from @angular/router in your Angular module.

Example:

```
import { NgModule } from '@angular/core';
import { RouterModule } from '@angular/router';
@NgModule({
 imports: [
  RouterModule.forRoot([...])
 ]
})
export class AppModule { }
```

- **Using Router Links**:

To create a router link, use the routerLink directive and provide the target route path as its value.

You can use the routerLink directive on anchor (<a>), button, or any other clickable element.

Example:

```
<!-- Router link for a route with a static path -->

<a routerLink="/about">About</a>

<!-- Router link for a route with a dynamic parameter -->

<a [routerLink]="['/products', productId]">Product Details</a>

<!-- Router link with additional options -->

<a [routerLink]="['/products']" [queryParams]="{ category: 'electronics' }">Electronics</a>
```

- **Active Router Links**:

To style router links based on the currently active route, you can use the routerLinkActive directive.

The routerLinkActive directive adds a CSS class to the element when its associated route is active.

You can specify the CSS class using the routerLinkActive directive's active property.

Example:

```
<a routerLink="/about" routerLinkActive="active">About</a>

<a routerLink="/products" routerLinkActive="active">Products</a>
```

- **Router Link Parameters**:

You can pass parameters to the router link by binding an array to the routerLink directive.

The array contains the route path segments and any route parameters.

Example:

```
<!-- Router link with a dynamic parameter -->

<a [routerLink]="['/products', productId]">Product Details</a>

<!-- Router link with multiple parameters -->

<a [routerLink]="['/users', userId, 'edit']">Edit User</a>
```

Query Parameters:

You can add query parameters to a router link using the queryParams binding.

The queryParams property accepts an object that represents the query parameters.

Example:

<a [routerLink]="['/products']" [queryParams]="{ category: 'electronics' }">Electronics

By using router links in Angular, you can create navigation elements that automatically handle route navigation when clicked. Router links provide a convenient and declarative way to create navigation within your Angular application. You can navigate to static routes, pass dynamic parameters, add query parameters, and even style the active router links.

Route Guards:

Route guards in Angular are used to control access to routes based on certain conditions. They allow you to implement logic that determines whether a user is allowed to navigate to a particular route or not. Angular provides several types of route guards that you can use to secure your application's routes. Here's an overview of the different types of route guards in Angular:

- **CanActivate**:

The CanActivate route guard determines whether a user can activate a route and navigate to it.

It is used to protect routes based on certain conditions, such as user authentication or authorization.

Implement the CanActivate interface and its canActivate method, which returns a boolean or an observable/promise that resolves to a boolean.

Example:

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

import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree } from '@angular/router';

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root'
```

```
})
export class AuthGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    // Implement your logic here
    // Return true to allow access, or navigate to another route using UrlTree
    // Return false to deny access
    return true;
  }
}
```

CanActivateChild:

The CanActivateChild route guard is similar to CanActivate but specifically used for child routes.

It allows you to protect child routes of a route, ensuring that the user meets certain criteria before accessing the child routes.

Implement the CanActivateChild interface and its canActivateChild method, which returns a boolean or an observable/promise that resolves to a boolean.

Example:

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

```
  providedIn: 'root'
})
export class AuthGuard implements CanActivateChild {
  canActivateChild(
    childRoute: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    // Implement your logic here
    // Return true to allow access, or navigate to another route using UrlTree
    // Return false to deny access
    return true;
  }
}
```

CanDeactivate:

The CanDeactivate route guard is used to determine whether a user can deactivate a route and leave it.

It is commonly used to show confirmation prompts or save form data before leaving a route.

Implement the CanDeactivate interface and its canDeactivate method, which returns a boolean or an observable/promise that resolves to a boolean.

Example:

```
import { Injectable } from '@angular/core';
import { CanDeactivate } from '@angular/router';
import { Observable } from 'rxjs';
export interface CanComponentDeactivate {
```

```
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}
@Injectable({
  providedIn: 'root'
})
export class ConfirmDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {
  canDeactivate(component: CanComponentDeactivate): Observable<boolean> | Promise<boolean> |
boolean {
    return component.canDeactivate ? component.canDeactivate() : true;
  }
}
```

Resolve:

The Resolve route guard is used to fetch data from a remote server or perform other tasks before activating a route.

It ensures that the route is only activated when the necessary data is available.

Implement the Resolve interface and its resolve method, which returns an observable/promise of the resolved data or the resolved data itself.

Example:

```
import { Injectable } from '@angular/core';
import { Resolve, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { Observable } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
```

```
export class DataResolver implements Resolve<string> {
  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<string> | Promise<string> | string {
    // Implement your data fetching logic here
    // Return the resolved data
    return 'Resolved Data';
  }
}
```

To use a route guard, you need to define it in the canActivate, canActivateChild, canDeactivate, or resolve property of the route configuration object. You can apply route guards at the individual route level or at the module level. By implementing route guards, you can add an extra layer of security and control to your application's routes.

Asynchronous Routing:

Asynchronous routing in Angular allows you to load the route components and their dependencies asynchronously, improving the initial load time of your application. It is achieved by lazy loading modules or using the loadChildren property in the route configuration. Here's how you can implement asynchronous routing in Angular:

- **Lazy Loading Modules**:

Lazy loading allows you to load modules and their associated components only when they are needed, rather than loading them all upfront.

To lazy load a module, create a separate module file for the module you want to load lazily.

Configure the route for the module with the loadChildren property in the route configuration.

Example:

```
const routes: Routes = [
```

```
 { path: 'admin', loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule) },

 // other routes

];
```

- **Preloading Lazy-Loaded Modules**:

By default, lazy-loaded modules are loaded on-demand when the associated route is accessed.

However, you can also preload the lazy-loaded modules to improve the user experience.

Use the PreloadAllModules strategy from @angular/router in the AppRoutingModule to preload all lazy-loaded modules.

Example:

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

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

const routes: Routes = [

 // route configurations

];

@NgModule({

 imports: [

  RouterModule.forRoot(routes, {

   preloadingStrategy: PreloadAllModules

  })

 ],

 exports: [RouterModule]

})

export class AppRoutingModule { }
```

- **Custom Preloading Strategy**:

Angular allows you to implement a custom preloading strategy to control how lazy-loaded modules are preloaded.

Create a custom preloading strategy by implementing the PreloadingStrategy interface.

Implement the preload method, which takes a route and a preload function and returns an observable.

Example:

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

import { PreloadingStrategy, Route } from '@angular/router';

import { Observable, of } from 'rxjs';

@Injectable({

 providedIn: 'root'

})

export class CustomPreloadingStrategy implements PreloadingStrategy {

 preload(route: Route, preload: () => Observable<any>): Observable<any> {

  if (route.data && route.data['preload']) {

   return preload();

  }

  return of(null);

 }

}
```

- **Configuring Route-level Preloading**:

You can configure individual routes to be preloaded using the data property in the route configuration.

Set the preload key to true for the routes you want to preload.

Example:

```
const routes: Routes = [

  { path: 'admin', loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule), data: { preload: true } },

  // other routes

];
```

By lazy loading modules and preloading them, you can optimize the initial load time of your Angular application. Lazy loading and preloading ensure that only the necessary modules are loaded when required, reducing the initial bundle size and improving the overall performance.

Nested Routes:

Nested routes in Angular allow you to define a hierarchical structure for your application's routes. This enables you to have child routes within a parent route, creating a nested navigation structure. Here's how you can implement nested routes in Angular:

- **Configure Parent Route**:

Define a parent route in your route configuration using the children property.

The children property should be an array of child route configurations.

Example:

```
const routes: Routes = [

  { path: 'products', component: ProductsComponent, children: [

    { path: 'details/:id', component: ProductDetailsComponent },

    { path: 'edit/:id', component: EditProductComponent }

  ]},

  // other routes

];
```

- **Add <router-outlet>**:

In the parent component's template where you want to render the child routes, add the <router-outlet> tag.

The <router-outlet> tag acts as a placeholder for rendering the child routes.

Example:

```
<h1>Products</h1>

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

- **Navigate to Child Routes**:

To navigate to the child routes, you can use the routerLink directive on an anchor or button element.

Specify the path to the child route relative to the parent route.

Example:

```
<!-- Navigate to product details -->

<a routerLink="details/1">View Details</a>

<!-- Navigate to product edit -->

<a routerLink="edit/1">Edit Product</a>
```

- **Accessing Route Parameters**:

If a child route has route parameters, you can access them using the ActivatedRoute service in the child component.

Import the ActivatedRoute service and inject it into the child component's constructor.

Use the params observable from the ActivatedRoute to subscribe to route parameter changes.

Example:

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

import { ActivatedRoute } from '@angular/router';

@Component({

 selector: 'app-product-details',
```

```
  templateUrl: './product-details.component.html',
  styleUrls: ['./product-details.component.css']
})
export class ProductDetailsComponent implements OnInit {
  productId: string;
  constructor(private route: ActivatedRoute) {}
  ngOnInit() {
    this.route.params.subscribe(params => {
      this.productId = params['id'];
      // Perform any required operations with the route parameter
    });
  }
}
```

By using nested routes in Angular, you can organize your application's routes in a hierarchical manner, making the navigation structure more intuitive and manageable. Child routes can have their own components and route parameters, allowing you to create more complex and interactive applications.

THANK YOU

HAPPY LEARNING