Directives in Angular

Directives are classes that add additional behavior to elements in your Angular applications. You can use Angular's built-in directives or create your own custom directives.

1. **Built-in Directives**:
   - **Structural Directives**: Change the DOM layout by adding or removing elements.
     - `ngIf`: Conditionally include an element in the DOM.
     - `ngFor`: Repeat a part of the DOM tree for each element in a collection.
   - **Attribute Directives**: Change the appearance or behavior of an element, component, or another directive.
     - `ngClass`: Add and remove a set of CSS classes.
     - `ngStyle`: Add and remove a set of HTML styles.

   ```html
   <div *ngIf="isVisible">This div is conditionally visible</div>
   ```

<div *ngFor="let item of items">{{ item }}

<div [ngClass]="{'active': isActive}">This div has conditional classes

<div [ngStyle]="{'color': isRed ? 'red' : 'blue'}">This div has conditional styles

```

2. **Custom Directives**:

- Create a new directive using the Angular CLI: `ng generate directive myHighlight`.

- Implement the directive in `my-highlight.directive.ts`:

```typescript

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

@Directive({

  selector: '[appMyHighlight]'

})

export class MyHighlightDirective {

```
    constructor(private el: ElementRef, private
renderer: Renderer2) {}

    @HostListener('mouseenter') onMouseEnter() {
      this.highlight('yellow');
    }

    @HostListener('mouseleave') onMouseLeave() {
      this.highlight(null);
    }

    private highlight(color: string) {
      this.renderer.setStyle(this.el.nativeElement,
'backgroundColor', color);
    }
  }
  ```

Debugging and Error Handling in Angular

1. **Debugging**:

- Use browser developer tools to inspect elements, view console logs, and debug TypeScript code.

- Use Angular's built-in tools like `ng.probe` to interact with Angular components from the console.

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

@Component({
  selector: 'app-debug-example',
  template: '<p>{{message}}</p>',
})
export class DebugExampleComponent implements OnInit {
  message: string;

  ngOnInit() {
    this.message = 'Debugging in Angular';
    console.log(this.message); // Debugging statement
  }
```

```
}
```

2. **Error Handling**:

- Use Angular's `ErrorHandler` class to handle errors globally.

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

@Injectable()
export class GlobalErrorHandler implements ErrorHandler {
  handleError(error: any): void {
    console.error('An error occurred:', error);
  }
}
```

- Provide the global error handler in the app module:

```typescript
import { NgModule, ErrorHandler } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { GlobalErrorHandler } from './global-error-handler';

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

Life Cycle Hooks in Angular

Life Cycle Hooks are special methods that provide different stages in the life of a component.

1. **ngOnInit**: Called once the component is initialized.
2. **ngOnChanges**: Called when input properties change.
3. **ngDoCheck**: Called during every change detection run.
4. **ngAfterContentInit**: Called after content is projected into the component.
5. **ngAfterContentChecked**: Called after every check of projected content.
6. **ngAfterViewInit**: Called after the component's view and child views are initialized.
7. **ngAfterViewChecked**: Called after every check of the component's view and child views.
8. **ngOnDestroy**: Called once the component is about to be destroyed.

Example:

```typescript
```

```
import { Component, OnInit, OnChanges, DoCheck,
AfterContentInit, AfterContentChecked,
AfterViewInit, AfterViewChecked, OnDestroy, Input }
from '@angular/core';


@Component({
  selector: 'app-lifecycle-demo',
  template: '<p>{{value}}</p>',
})
export class LifecycleDemoComponent implements
OnInit, OnChanges, DoCheck, AfterContentInit,
AfterContentChecked, AfterViewInit,
AfterViewChecked, OnDestroy {
  @Input() value: string;

  ngOnInit() {
    console.log('ngOnInit');
  }

  ngOnChanges() {
    console.log('ngOnChanges');
  }
```

```
ngDoCheck() {
  console.log('ngDoCheck');
}

ngAfterContentInit() {
  console.log('ngAfterContentInit');
}

ngAfterContentChecked() {
  console.log('ngAfterContentChecked');
}

ngAfterViewInit() {
  console.log('ngAfterViewInit');
}

ngAfterViewChecked() {
  console.log('ngAfterViewChecked');
}
```

```
  ngOnDestroy() {
    console.log('ngOnDestroy');
  }
}
```

Pipes in Angular

Pipes transform data in templates. Angular provides built-in pipes like `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, etc., and you can create custom pipes.

1. **Built-in Pipes**:

   ```html
   <p>{{ birthday | date:'fullDate' }}</p>
   <p>{{ name | uppercase }}</p>
   ```

2. **Custom Pipes**:

- Create a new pipe using the Angular CLI: `ng generate pipe exponentiation`.

- Implement the pipe in `exponentiation.pipe.ts`:

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

@Pipe({
  name: 'exponentiation'
})
export class ExponentiationPipe implements PipeTransform {
  transform(value: number, exponent: number): number {
    return Math.pow(value, exponent);
  }
}
```

Using Services & Dependency Injection in Angular

Services are used to share data and logic across components. **Dependency Injection (DI)** is a design pattern used to implement IoC (Inversion of Control).

1. **Creating a Service**:

   - Generate a service using the Angular CLI: `ng generate service data`.

   - Implement the service in `data.service.ts`:

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

   @Injectable({
     providedIn: 'root',
   })
   export class DataService {
     private data: string = 'Angular Service Data';

     getData(): string {
       return this.data;
   ```

```
  }
}
```

2. **Injecting a Service**:
   - Use the service in a component:

   ```typescript
   import { Component, OnInit } from '@angular/core';
   import { DataService } from './data.service';

   @Component({
     selector: 'app-data-consumer',
     template: '<p>{{data}}</p>',
   })
   export class DataConsumerComponent implements OnInit {
     data: string;

     constructor(private dataService: DataService) {}
   ```

```
  ngOnInit() {
    this.data = this.dataService.getData();
  }
}
```

Forms Programming in Angular

Angular supports two types of forms: **Template-driven** and **Reactive Forms**.

1. **Template-driven Forms**:

```html
<form #form="ngForm" (ngSubmit)="onSubmit(form.value)">
  <input name="name" ngModel required>
  <button type="submit">Submit</button>
</form>
```

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

@Component({
  selector: 'app-template-form',
  templateUrl: './template-form.component.html',
})
export class TemplateFormComponent {
  onSubmit(formData) {
    console.log('Form Data:', formData);
  }
}
```

2. **Reactive Forms**:

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

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


  @Component({
    selector: 'app-reactive-form',
    template: `
      <form [formGroup]="form" (ngSubmit)="onSubmit()">
        <input formControlName="name">
        <button type="submit">Submit</button>
      </form>
    `,
  })
  export class ReactiveFormComponent implements OnInit {
    form: FormGroup;


    ngOnInit() {
      this.form = new FormGroup({
        name: new FormControl('', Validators.required),
      });
```

```
  }

  onSubmit() {
    console.log('Form Value:', this.form.value);
  }
}
```

RXJS and Http Programming in Angular

RxJS is a library for reactive programming using observables, which makes it easy to compose asynchronous or callback-based code.

1. **HttpClient**:
   - Import `HttpClientModule` in your app module.

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

```
@NgModule({
  imports:

[HttpClientModule],
})
export class AppModule {}
```

- Use `HttpClient` in a service:

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

@Injectable({
  providedIn: 'root',
})
export class DataService {
  constructor(private http: HttpClient) {}
```

```
    getData(): Observable<any> {
      return this.http.get('https://api.example.com/data');
    }
  }
  ```

  - Consume the service in a component:

  ```typescript
  import { Component, OnInit } from '@angular/core';
  import { DataService } from './data.service';

  @Component({
    selector: 'app-data-consumer',
    template: '<pre>{{data | json}}</pre>',
  })
  export class DataConsumerComponent implements OnInit {
```

```
    data: any;

    constructor(private dataService: DataService) {}

    ngOnInit() {
      this.dataService.getData().subscribe(data => {
        this.data = data;
      });
    }
  }
  ```

Routing in Angular

Routing allows you to define navigation paths between components.

1. **Defining Routes**:
   - Import `RouterModule` and define routes in your app module:

```typescript
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

2. **Using Router Links**:

- Add navigation links in your template:

```html
<nav>
  <a routerLink="/">Home</a>
  <a routerLink="/about">About</a>
</nav>
<router-outlet></router-outlet>
```

3. **Navigating Programmatically**:
   - Use `Router` service to navigate programmatically:

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

@Component({
  selector: 'app-navigation',
```

```
    template: '<button (click)="goToAbout()">Go to
About</button>',
  })
  export class NavigationComponent {
    constructor(private router: Router) {}

    goToAbout() {
      this.router.navigate(['/about']);
    }
  }
```