Fundamental Concepts of Angular

Definition of Angular

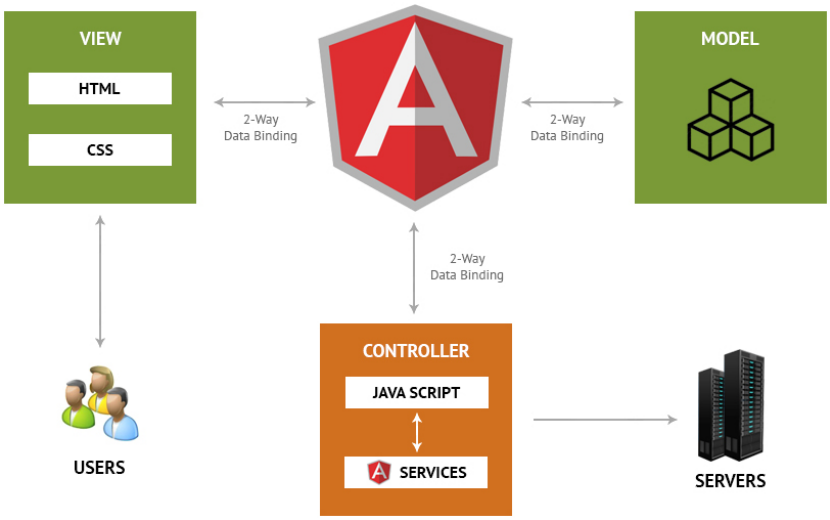
Angular is an open-source framework developed by Google for creating dynamic web applications. It is based on TypeScript and allows applications to be modularly structured through a component-based architecture with services and modules. Angular simplifies data management, API interactions, state management, and navigation in complex web applications.

Figure 1: Angular Logo

Angular Architecture

The architecture of Angular is built on clear separation of responsibilities to ensure scalability and maintainability. The main building blocks of this architecture are: **modules**, **components**, **services**, along with other entities like **directives** and **pipes**.

Figure 2: Angular Architecture



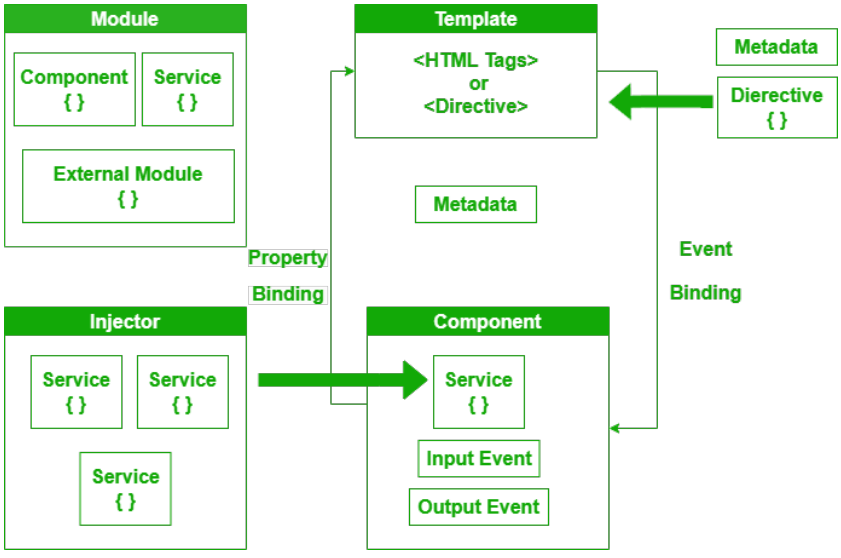
The goal of this section is to understand how these elements interact to form a cohesive Angular application.

1 Overview of Angular Architecture

Angular follows an enhanced **Model-View-Controller (MVC)** pattern, where:

- **Components** manage the view and capture user actions.
- **Services** handle data manipulation and business logic.
- **Modules** structure and organize functionalities.

Figure 3: Angular Architecture



This pattern promotes separation of responsibilities for modular development.

2 Separation of Responsibilities and Data Flow

2.1 Modules: The Global Structure

- Modules define the application's skeleton.
- They group components, services, and other entities.

Relationship with other parts: Modules act as logical containers for grouping related functionalities. For example, an authentication module would contain all components and services related to user management.

Tip: Always start by defining modules before creating associated components and services.

2.2 Components: The User Interface

- Each component represents a part of the user interface (a button, a form, a page).
- A component communicates with services to fetch or process the data to be displayed.

Data Flow: Data flows **from the service to the component**, and user actions flow **from the component to the service**.

Tip: Once the module is configured, begin creating components to design the user interface.

2.3 Services: Business Logic and Data

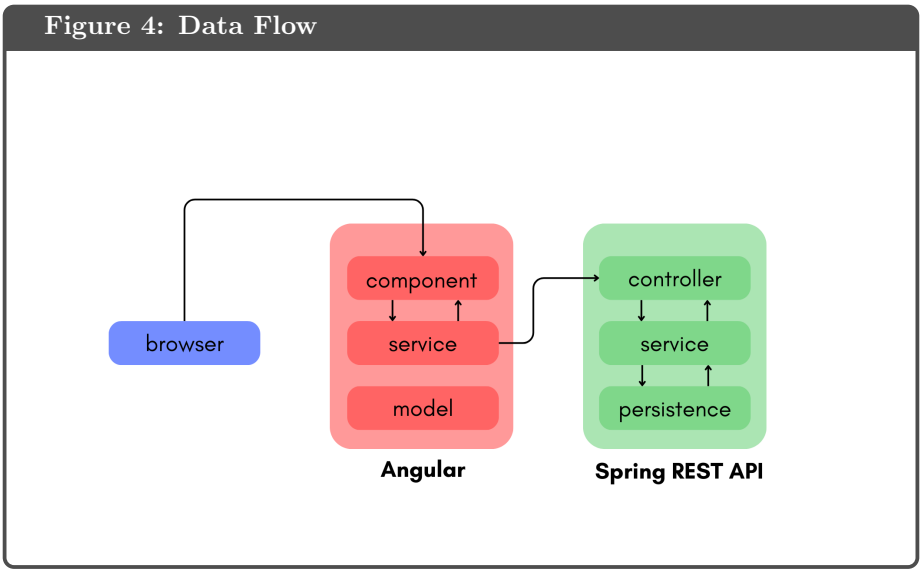
- Services are classes dedicated to data management and business processes.
- They often use the HTTP module to interact with external APIs.

Data Flow: Services retrieve or manipulate data and pass it to components.
Tip: After defining the data model (interfaces/classes), develop services to centralize business logic.

3 Three-Step Data Flow Example

1. **Defining Modules:** Create a module for a specific functionality (e.g., product management).
2. **Creating Services:** Implement a service to fetch product data via an API.
3. **Designing Components:** Develop a component that uses the service to display a product list.

Figure 4: Data Flow



4 Importance of Data Flow Between Parts

The order of development is crucial to avoid errors and inefficiencies:

1. **Module:** Structures functionality and facilitates integration.
2. **Service:** Manages data and processes related to the functionality.
3. **Component:** Displays data to the user and captures user actions.

This flow ensures clean architecture with well-defined responsibilities and maintainable code.

Modules in Angular: Structure and Functionality

In Angular, **modules** play a key role in organizing the application. They allow grouping components, services, directives, pipes, and other modules to make the application modular, reusable, and easily maintainable. Each module is associated with a TypeScript file that serves as the module's entry point, the most important being **AppModule**, the main module.

1. AppModule: The Application Entry Point

The **AppModule** is the base module of any Angular application. It is the module that gets loaded during the application's startup. It groups all the other modules required for the application to function correctly.

AppModule Declaration: AppModule is a typical Angular module containing essential information to bootstrap the application, such as component declarations, necessary module imports, services, and dependency injection.

Example:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [  // Declare all components used by this module
    AppComponent
  ],
  imports: [  // Import other required modules
    BrowserModule
  ],
  providers: [],  // Declare services
  bootstrap: [AppComponent]  // Component that bootstraps the
      application
})
export class AppModule { }
```

Listing 1: Using a service in a component

2. Structure of Angular Modules

An Angular module is defined using the `@NgModule` decorator, which takes an object with several key properties. These properties configure the module based on the application's specific needs.

2.1 Declarations

The `declarations` property groups all the **components**, **directives**, and **pipes** used within the module. Every component, directive, or pipe must be declared in a module to be recognized and used.

- **Components:** Represent the user interface.
- **Directives:** Modify the appearance or behavior of a DOM element.
- **Pipes:** Transform the data displayed to the user.

Example:

```
@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent,
    FooterComponent
  ]
})
```

2.2 Imports

The `imports` property contains a list of **external modules** that this module uses. This allows grouping functionalities from other modules, such as the routing module or forms module.

- For example, you can import modules like `FormsModule`, `HttpClientModule`, `RouterModule`, etc.

Example:

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

@NgModule({
  imports: [
    BrowserModule,  // Basic module for the browser
    FormsModule,    // Module for forms
    RouterModule    // Module for routing
  ]
})
```

Listing 2: Using a service in a component

2.3 Providers

The `providers` property allows declaring the **services** you want to inject into the application. This enables Angular to inject these services into components and other services via dependency injection.

- Services can be provided at the module level or for the entire application using `providers` in `@NgModule`.

Example:

```
import { ProductService } from './product.service';

@NgModule({
  providers: [
    ProductService  // Declare the service to be used in the
        application
  ]
})
```

Listing 3: Using a service in a component

2.4 Bootstrap

The `bootstrap` property specifies the **main component(s)** that bootstraps the application. Here, you declare the root component of your application, typically `AppComponent`.

- `AppComponent` is the component used to initialize the application's user interface.

Example:

```
@NgModule({
  bootstrap: [AppComponent]  // Define the component that bootstraps the application
})
```

3. Complete Example of an Angular Module

Here is an example of a complete module with the various parts we have discussed:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';  // For forms
import { RouterModule } from '@angular/router';  // For routing

import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';
import { ProductService } from './services/product.service';

@NgModule({
  declarations: [
    AppComponent,  // Main component
    HeaderComponent,  // Header component
    FooterComponent   // Footer component
  ],
  imports: [
```

```
    BrowserModule,  // Basic module for the browser
    FormsModule,    // Module for forms
    RouterModule    // Module for routing
  ],
  providers: [
    ProductService  // Service for managing products
  ],
  bootstrap: [AppComponent]  // Main component that bootstraps the
      application
})
export class AppModule { }
```

Listing 4: Using a service in a component

4. The Role of `AppModule` in the Information Flow

The **AppModule** acts as the entry point and coordination hub for all the functionalities of the application.

- As soon as the application starts, Angular loads the **AppModule**.
- It **declares** the components, **imports** the necessary modules (such as those for forms, HTTP services, etc.), **provides services** for the application, and **initializes** the **main component (AppComponent)**.

5. Conclusion

The architecture of an Angular module allows for efficient and maintainable structuring and organization of functionalities. The structure of a module — **Declarations**, **Imports**, **Providers**, and **Bootstrap** — is essential for defining how the different parts of the application will interact and be accessible across other modules and components.

Practical Tip: Start by defining your modules to organize the application into independent features. Then, add the necessary components and services, ensuring they are properly injected into the modules.

Components in Angular: Structure and Functionality

In Angular, **components** are the fundamental building blocks for constructing the user interface (UI). Each component represents a **specific part of the UI**, such as a button, a page, or a form. Components enable the division of the UI into modular, reusable, and easily maintainable elements.

Each Angular component is associated with a **Component decorator**, which provides metadata about the component, including the HTML structure, CSS styles, and TypeScript behavior.

1. The Role of a Component in Angular

A **component** serves to:

- **Manage the user interface** of a specific section of the application (e.g., a form, a navigation bar).
- **Handle user interactions** (e.g., clicks, input in form fields).
- **Communicate with other components or services** to retrieve or manipulate data.

In summary, components are the building blocks of an Angular application, and their primary role is to bind the display and business logic.

2. Using the Component Decorator

The Component decorator is a function that configures an Angular component. It is used to define the component's metadata, such as the HTML template, CSS styles, and TypeScript behavior. It also connects a component to a specific module.

The Component decorator takes an object with several key properties:

- **selector**: Declares the name of the custom HTML element representing this component in the DOM.
- **templateUrl** or **template**: A link to the HTML file or inline HTML code used for the component's template.
- **styleUrls** or **styles**: A link to the CSS file or inline CSS code used for the component's styles.
- **providers**: Declares the services used in this component.

Example of the Component Decorator:

```
1
2  import { Component } from '@angular/core';
3
4  @Component({
5    selector: 'app-header',  // Declares a custom HTML element
6    templateUrl: './header.component.html',  // Link to the HTML
         template
7    styleUrls: ['./header.component.css']   // Link to the CSS file
         for the component
8  })
9  export class HeaderComponent {
10   title: string = "Welcome to our website";
11 }
```

Listing 5: Using a service in a component

3. Structure of a Component

An Angular component consists of three main files:

1. **HTML (Template)**: Defines the component's user interface.
2. **CSS (Styles)**: Defines the styles associated with the component's interface.
3. **TypeScript (Class)**: Contains the component's business logic, manages data, and handles user events.

3.1 The HTML File (Template)

The HTML file defines the structure of the user interface. It can include basic HTML tags as well as Angular directives for data binding and event handling.

Example of an HTML Template:

```
<div class="header">
  <h1>{{ title }}</h1>
  <button (click)="onClick()">Click here</button>
</div>
```

- { `title` }: This is a **data binding** that displays the value of the component's `title` property. - `(click)="onClick()"`: This syntax is an **event binding** that links a DOM event (like a click) to a TypeScript method (here, `onClick()`).

3.2 The CSS File (Styles)

The CSS file contains styles specific to the component. Angular uses a style encapsulation system to prevent a component's styles from affecting other components.

Example of a CSS File:

```
.header {
  background-color: #4CAF50;
  color: white;
  padding: 10px;
}
```

3.3 The TypeScript File (Class)

The TypeScript file contains the **component class** that manages the component's properties and methods. This is where you define the business logic, internal state, and behaviors in response to user interactions.

Example of a TypeScript Class:

```
1
2  import { Component } from '@angular/core';
3
4  @Component({
5    selector: 'app-header',
6    templateUrl: './header.component.html',
7    styleUrls: ['./header.component.css']
8  })
9  export class HeaderComponent {
10   title: string = "Welcome to our website";
11
12   onClick() {
13     console.log("The button was clicked!");
14   }
15 }
```

Listing 6: Using a service in a component

- The `title` **property** contains data to be displayed in the HTML template. - The `onClick()` **method** handles the button's click event.

4. Data Flow and Interactions within a Component

Components primarily interact with:

- **Properties and methods** defined in the TypeScript class.
- **The HTML template**, via data binding.
- **Services** for data manipulation or API calls.

4.1 Data Binding

Angular provides several types of data binding:

- **One-way Binding**: From the class to the template (or vice versa).
  - { `value` }: Interpolation to display values in the template.
  - `[property]="value"`: Binds HTML element properties to a component property.
  - `(event)="handler()"`: Binds a DOM event to a method in the class.
- **Two-way Binding**: Uses `[(ngModel)]` to bind form data.

Example of Two-way Binding:

```
<input [(ngModel)]="username">
<p>Welcome, {{ username }}!</p>
```

5. Complete Example of a Component

Here is a complete example of a component:

HTML (Template):

```
<div class="login-form">
  <h2>{{ title }}</h2>
  <input [(ngModel)]="username" placeholder="Enter your name">
  <button (click)="onSubmit()">Log in</button>
</div>
```

Listing 7: HTML example of a login form

CSS (Styles):

```
.login-form {
  margin: 20px;
  padding: 15px;
  border: 1px solid #ccc;
}
input {
  padding: 5px;
  margin-bottom: 10px;
}
button {
  background-color: #4CAF50;
  color: white;
  padding: 10px 20px;
}
```

Listing 8: CSS styling for login form

TypeScript (Class):

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

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent {
  title: string = 'Login';
  username: string = '';

  onSubmit() {
    console.log(`Username: ${this.username}`);
  }
}
```

Listing 9: TypeScript class for login functionality

6. Conclusion

Components are the core building blocks for constructing the user interface in Angular. Each component comprises three essential parts:

- **HTML** for the UI structure,
- **CSS** for styling,
- **TypeScript** for logic and data management.

The Component decorator binds these three parts and defines how the component interacts with other components or services, enabling efficient data and event handling.

Practical Tip: When creating a component, start by defining the HTML template (structure), then implement the necessary logic in the TypeScript class, and finally, add styles to ensure the interface is clean and intuitive.

5 Data Binding: One-way and Two-way

In Angular, the **template** of a component is responsible for the user interface, while **data binding** connects the data model (properties and methods in the component class) to the DOM (UI model).

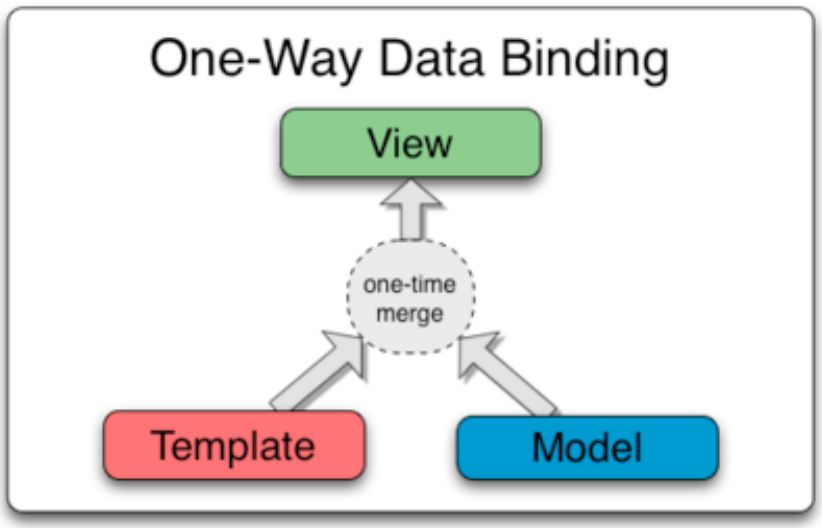
There are several types of data binding that Angular uses to synchronize data between the component and the view effectively.

6 Data Binding: One-way and Two-way

6.1 One-way Binding

In one-way binding, data flows in a single direction: either from the component to the view (display) or from the view to the component (user interactions).

Figure 5: One-way Data Binding



6.1.1 Interpolation (from component to view)

Interpolation binds a value from the TypeScript class to a section in the HTML.

```
<p>{{ title }}</p>
```

In this example, the content of `title` in the component class is displayed inside the `<p>` element.

6.1.2 Property Binding (from component to view)

Property binding links an HTML element property to a component property.

```
<img [src]="imageUrl" alt="Dynamic image">
```

Here, the `src` attribute of the image is bound to the `imageUrl` property in the component.

6.1.3 Event Binding (from view to component)

Event binding associates a DOM event (like a click) with a method in the component class.

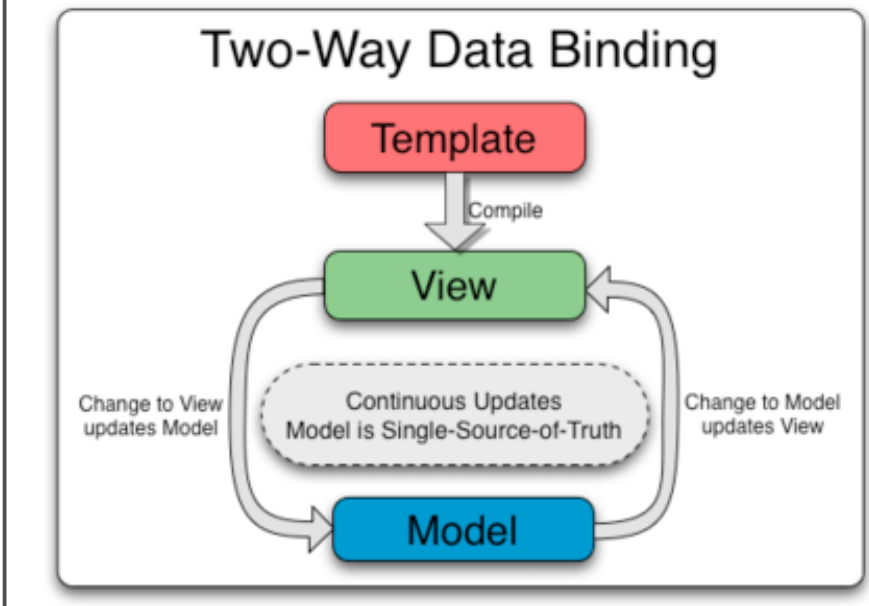
```
<button (click)="onClick()">Click here</button>
```

The `click` event triggers the `onClick()` method in the component when the button is clicked.

6.2 Two-way Binding

Two-way binding synchronizes data between the model and the view, so any change in one is immediately reflected in the other.

Figure 6: Two-way Data Binding



Angular primarily implements this using the `ngModel` directive, which binds a form field to a property in the component class.

```
<input [(ngModel)]="username">
<p>Username: {{ username }}</p>
```

In this example, the value of `username` is bound to the input field. If the user types something, `username` is automatically updated, and if the `username` property changes in the component, the input field also updates.

7 Using Angular Directives

Directives are classes that modify the behavior of DOM elements. They are used to create reusable behaviors in the user interface.

7.1 Structural Directives

Structural directives modify the structure of the DOM by adding or removing elements. They are typically used to **control visibility** or **repeat elements**.

7.1.1 *ngIf

This directive conditionally displays a DOM element based on an expression. If the expression is true, the element is displayed; otherwise, it is removed from the DOM.

```
<div *ngIf="isVisible">This content is visible if isVisible is true.</div>
```

7.1.2 *ngFor

This directive repeats an element for each item in a collection.

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

7.2 Attribute Directives

Attribute directives modify the appearance or behavior of an element without changing its structure.

7.2.1 ngClass

This directive adds or removes CSS classes from an element based on conditions specified in the expression.

```
<div [ngClass]="{'highlight': isHighlighted}">This text is highlighted if isHighlighted is t
```

7.2.2 ngStyle

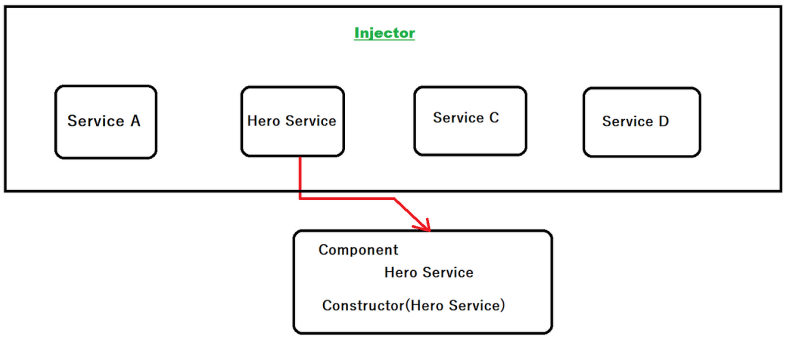
This directive dynamically adds or modifies the styles of an element.

```
<div [ngStyle]="{'color': color, 'font-size': fontSize + 'px'}">This text has dynamic styles
```

8 Services and Dependency Injection

Services play a critical role in Angular for managing **business logic**, accessing data, and performing external operations. They are used to provide reusable functionality throughout the application. Services are commonly used to communicate with APIs, manipulate data, or store information.

Figure 7: Services and Dependency Injection



8.1 Role of Services in Angular

A service is a **class** that encapsulates specific logic or behavior. Services are often used to:

- **Share data between components**: A service can store shared data and allow components to read or modify it.
- **Communicate with APIs**: A service can manage HTTP calls and retrieve data from a server.
- **Manage application state**: A service can hold information that needs to be accessible to multiple components, such as user authentication state or application settings.

8.2 Dependency Injection with `@Injectable`

Dependency injection (DI) is a core concept in Angular that allows injecting objects (such as services) into components or other services. This facilitates managing dependencies without requiring classes to create these objects themselves.

8.2.1 `@Injectable()`

This is a decorator that marks a class as an **injectable service**. It indicates to Angular that this class can be **injected** into other components or services.

Here is a simple example of a service with dependency injection:

```
1 import { Injectable } from '@angular/core';
2
3 @Injectable({
4   providedIn: 'root' // This indicates that the service is
        available at the application level
```

```typescript
})
export class DataService {
  private data: string[] = [];

  constructor() { }

  getData(): string[] {
    return this.data;
  }

  addData(newData: string): void {
    this.data.push(newData);
  }
}
```

Listing 10: TypeScript service example

8.3 Using the Service in a Component

In the component, you **inject** the service to access it:

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

@Component({
  selector: 'app-data-list',
  templateUrl: './data-list.component.html',
  styleUrls: ['./data-list.component.css']
})
export class DataListComponent {
  data: string[] = [];

  constructor(private dataService: DataService) { } // Inject the service

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

  addData(newData: string): void {
    this.dataService.addData(newData);
  }
}
```

Listing 11: Using a service in a component

In this example, `DataService` is injected into `DataListComponent` via the constructor. You can call the service's methods to retrieve or manipulate data.

9 Conclusion

- **Templates and Data Binding**: Data binding synchronizes the model (data) with the view (user interface). One-way and two-way bindings

facilitate interaction between the model and the view.

- **Angular Directives**: Angular offers powerful directives like `*ngIf`, `*ngFor`, `ngClass`, and `ngStyle`, enabling declarative modifications to the DOM structure or styles.
- **Services and Dependency Injection**: Services encapsulate business logic and shared data. Dependency injection makes it easy to inject services into components, promoting modular and reusable code.

These fundamental Angular concepts are essential for developing robust and well-structured applications.

Routing in Angular

Routing allows navigation between different views in an application. In Angular, routing is configured using the `RouterModule` and enables linking components to specific URLs.

1. Configuring Routes in `app-routing.module.ts`

In Angular, routes are defined in a module file called `app-routing.module.ts`. This module imports the `RouterModule` and uses the `RouterModule.forRoot(routes)` method to configure the various routes.

Here is an example of route configuration:

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

Listing 12: Route Configuration Example

In this example: - The default route (`path: ''`) loads the `HomeComponent`. - The route `/about` loads the `AboutComponent`.

2. Navigating Between Components and Managing Route Parameters

To navigate between components, the `routerLink` directive is used in the template, allowing URLs to be linked to an element.

Example:

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

Managing Route Parameters: Sometimes, parameters need to be retrieved from the URL. This can be done using `ActivatedRoute` in the target component.

Example:

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

@Component({
  selector: 'app-about',
  templateUrl: './about.component.html',
  styleUrls: ['./about.component.css']
})
export class AboutComponent implements OnInit {

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.route.params.subscribe(params => {
      const id = params['id']; // Retrieve the 'id' parameter from
          the URL
      console.log(id);
    });
  }
}
```

Listing 13: Retrieving Route Parameters

Forms in Angular

Forms enable the collection of user input. Angular provides two main approaches for handling forms: **Template-Driven Forms** and **Reactive Forms**.

1. Template-Driven Forms

In this approach, the form and its validations are primarily defined in the HTML template.

Example:

```
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <input name="username" ngModel required />
  <button type="submit" [disabled]="!myForm.valid">Submit</button>
</form>
```

Listing 14: Template-Driven Form Example

In this case, `ngForm` is a directive that creates a form object, and `ngModel` binds the fields to the component class.

2. Reactive Forms

Reactive forms are more powerful and flexible. They are defined in the component and used to create complex forms with advanced validations.

Example:

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

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  loginForm: FormGroup;

  constructor(private fb: FormBuilder) { }

  ngOnInit(): void {
    this.loginForm = this.fb.group({
      username: ['', Validators.required],
      password: ['', [Validators.required, Validators.minLength(6)]]
    });
  }

  onSubmit(): void {
    if (this.loginForm.valid) {
      console.log(this.loginForm.value);
    }
  }
}
```

Listing 15: Reactive Form Example

Here, `FormBuilder` is used to create a reactive form and apply validations to it.

Pipes in Angular

Pipes are used to transform the data displayed in the user interface. They are commonly used to format values, such as dates or currencies.

1. Built-in Pipes

Angular provides several built-in pipes for common transformations: - `date`: Formats a date.

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

- `currency`: Displays a value as currency.

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

2. Creating Custom Pipes

It is possible to create custom pipes for specific transformations required by the application.

Example:

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

@Pipe({
  name: 'reverse'
})
export class ReversePipe implements PipeTransform {
  transform(value: string): string {
    return value.split('').reverse().join('');
  }
}
```

Listing 16: Custom Pipe Example

In the template:

```
<p>{{ 'Hello' | reverse }}</p> <!-- Outputs 'olleH' -->
```

Data Management and HTTP

Angular enables HTTP requests using the `HttpClientModule`, making it easier to consume external APIs.

1. Using `HttpClientModule`

To consume REST APIs, you first need to import the `HttpClientModule` in the `app.module.ts` file:

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

@NgModule({
  imports: [HttpClientModule],
  // other configurations...
})
export class AppModule { }
```

Listing 17: Importing HttpClientModule

Then, in your service or component, you can use `HttpClient` to make HTTP requests.

Example of a GET request:

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

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

```
  templateUrl: './data.component.html',
  styleUrls: ['./data.component.css']
})
export class DataComponent {

  constructor(private http: HttpClient) {}

  fetchData(): void {
    this.http.get('https://api.example.com/data').subscribe(data =>
        {
      console.log(data);
    });
  }
}
```

Listing 18: HTTP GET Request Example

2. Consuming REST APIs and Error Handling

When consuming APIs, it is important to handle errors using `catchError` and to process asynchronous responses.

Example of error handling:

```
import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';

this.http.get('https://api.example.com/data')
  .pipe(catchError(error => {
    console.error('Error occurred:', error);
    return throwError(error);
  }))
  .subscribe(data => {
    console.log(data);
  });
```

Listing 19: Error Handling in HTTP Requests

RxJS and Observables

Observables are a core part of Angular and are used to handle asynchronous data streams, such as HTTP requests, user events, or data changes.

1. Introduction to RxJS

RxJS (Reactive Extensions for JavaScript) is a library for composing asynchronous and event-based programs using observables.

An **observable** is a data stream that can be observed. A **subscriber** can subscribe to this observable to receive the emitted values.

Basic example with an observable:

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

const observable = new Observable(subscriber => {
  subscriber.next('Hello');
  subscriber.complete();
});

observable.subscribe(value => console.log(value));  // Outputs '
    Hello'
```

Listing 20: Basic Observable Example

2. Using Observables to Handle Data Streams

Observables allow you to manage data streams reactively, for example, handling data from an API, user events, etc.

Example with an HTTP request:

```
this.http.get('https://api.example.com/data')
  .subscribe(data => {
    this.data = data;
  });
```

Listing 21: Using Observables for HTTP Requests

Lazy Loading and Optimization

Lazy Loading allows Angular modules to be loaded on demand, reducing the initial load time of the application.

1. Deferred Module Loading

Lazy loading is configured in the routes. Instead of loading all modules at the start, only the necessary ones for the current view are loaded.

Example:

```
const routes: Routes = [
  { path: 'dashboard', loadChildren: () => import('./dashboard/
      dashboard.module').then(m => m.DashboardModule) }
];
```

Listing 22: Lazy Loading Example

2. Performance Optimization

Several techniques can improve the performance of an Angular application, such as: - **Reducing bundle size** by using Lazy Loading. - **Using ChangeDetectionStrategy.OnPush** to minimize change detection cycles. - **Minification and compression of files** during deployment.

Testing and Deployment

Unit testing ensures that the code works as expected, while deployment involves making the application available to users on a server or cloud platform.

1. Unit Testing with Jasmine/Karma

Angular uses **Jasmine** for unit testing and **Karma** for running tests in a browser environment.

Example of a unit test:

```
import { TestBed } from '@angular/core/testing';
import { DataService } from './data.service';

describe('DataService', () => {
  let service: DataService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(DataService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });
});
```

Listing 23: Unit Test Example

2. Deploying to a Server or Cloud Platform

Once the application has been developed and tested, it needs to be deployed to a server. The deployment process typically involves: - Building the production version of the application with `ng build --prod`. - Deploying the generated files (located in the `dist/` folder) to a server or a cloud platform such as **Firebase**, **AWS**, or **Heroku**.

Conclusion

This course covers the essential concepts of Angular for creating robust and responsive applications. From routing to data management, forms, and performance optimization, Angular provides a comprehensive toolkit for modern web application development.