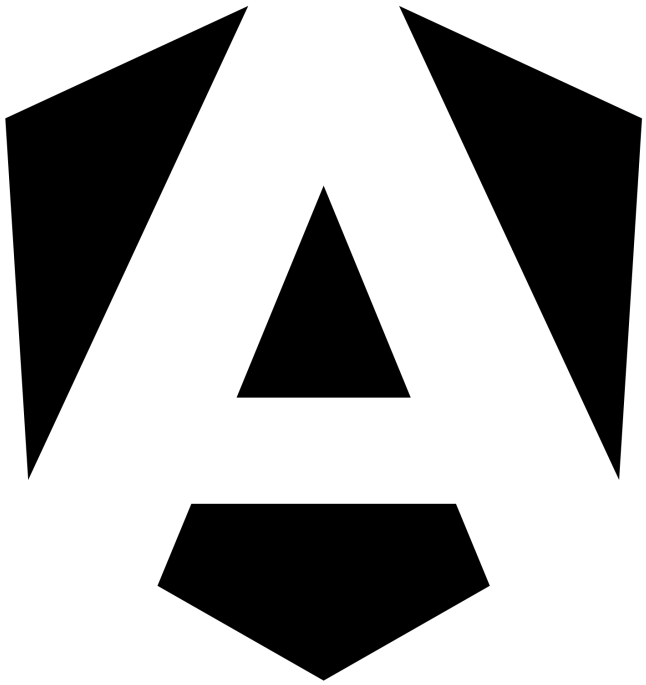

Secure Web Development Series - I

Primer for Secure IAM Development with OIDC and Angular

Navid Mohaghegh

Secure Web Development Series - Primer for Secure AIM Development with OIDC and Angular

Contents

Chapter 1

Introduction

1.1 Introduction

Angular is a platform and framework for building single-page client applications using HTML and TypeScript. It is written in TypeScript and maintained by a dedicated team at Google. Angular provides a comprehensive suite of tools, libraries, and features to simplify the development process and ensure high performance and scalability of web applications.

Angular empowers developers to build applications with a clear and maintainable architecture, thanks to its component-based structure and powerful dependency injection system. Whether you're building small projects or large-scale enterprise applications, Angular's robust ecosystem supports development at any scale.

For more information, visit the official Angular documentation at Angular Docs.

1.2 Prerequisites

Before you begin, ensure you have the following installed:

- Node.js and npm (Node Package Manager) via Node.js Download
- Integrated Development Environment such as Intellij WebStorm or Microsoft Visual Studio Code

1.2.1 Setting Up the Development Environment

To set up your development environment, follow these steps:

1. Install Angular CLI globally using npm:

```
npm install -g @angular/cli

```

2. Create a new Angular project:

```
ng new my-app

```


3. Navigate to the project directory and start the development server:

```
cd my-app
ng serve
```

4. Open your browser and navigate to `http://localhost:4200` to see your application running.

1.2.2 Angular Online Interactive Tutorial

The Angular Online Interactive Tutorial offers a hands-on learning experience designed to introduce the foundational concepts of Angular. This tutorial is ideal for individuals with a basic understanding of HTML, CSS, and JavaScript. It covers key Angular concepts through step-by-step exercises, allowing learners to practice and reinforce their understanding. Each step of the tutorial represents a different Angular concept, making it easy to progress at your own pace. If you encounter difficulties, a "Reveal answer" feature is available to help you move forward. We will follow very similar topics in the following chapters.

Chapter 2

Components in Angular

2.1 Components in Angular

Components are the building blocks of an Angular application. Each component consists of:

- An HTML template that defines the view
- A CSS stylesheet that defines the appearance
- A TypeScript class that defines the behavior

2.1.1 Creating a Component

To create a component, use the Angular CLI:

```
ng generate component my-component
```

This command creates a new directory with four files: HTML, CSS, TypeScript, and a test file.

2.1.2 Updating the Component Class

Define properties and methods in the TypeScript class to manage the component's data and behavior. For example:

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

@Component({
  selector: 'app-root',
  template: `
  Hello Universe
  `,
  styles: `
  :host {
    color: #a144eb;
  }
  `,
  standalone: true,
})
```

```
export class AppComponent {}
```

You compose components to build complex UIs by nesting components within other components. This approach promotes reusability and modularity in your application. For example, a parent component can include child components in its template:

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

@Component({
  selector: 'app-user',
  template: `
  Username: {{ username }}
  `,
  standalone: true,
})
export class UserComponent {
  username = 'youngTech';
}

@Component({
  selector: 'app-root',
  template: `
  <section>
  <app-user />
  </section>
  `,
  standalone: true,
  imports: [UserComponent],
})
export class AppComponent {}
```

Listing 2.1: User and App Angular Components

```
  <!DOCTYPE html>
  <html lang="en">
  <head>
  <meta charset="UTF-8">
  <title>Angular App</title>
  </head>
  <body>
  <app-root></app-root>
  </body>
  </html>
```

Listing 2.2: Main Index HTML File

Please note:

- The `selector` property in an Angular component specifies the custom HTML tag that will represents the component.once mounted in runtime.
- For `AppComponent`, `selector: 'app-root'` means that this component will be identified and instantiated in the HTML using the `<app-root>` tag.
- This is typically the root component of the application, which gets bootstrapped and rendered in the main HTML file (usually `index.html`) of the Angular application.

- The `standalone` property, when set to `true`, indicates that the component is self-contained and does not depend on any Angular modules.
- This allows for more modular and reusable components, as they can be declared and used without being part of a larger Angular module (`NgModule`).
- In the provided code, both `UserComponent` and `AppComponent` are marked as standalone, meaning they can function independently and do not need to be declared in any `NgModule`.

Chapter 3

Angular Directives

Directives are classes that add additional behavior to elements in your Angular applications. Use Angular's built-in directives to manage forms, lists, styles, and what users see. For more information refer to Angular Directives.

3.1 Control Flow in Angular

Control flows in Angular are essential for managing the logic of your application. Angular provides various ways to control the flow of your application using directives. The most commonly used directives for control flow are:

- **`*ngIf`**: Conditionally includes a template based on a boolean expression.
- **`*ngFor`**: Iterates over a collection and renders a template for each item.
- **`*ngSwitch`**: Displays one of a set of possible templates based on the value of an expression.

3.1.1 `*ngIf`

The `*ngIf` directive is used to conditionally include or exclude a template from the DOM based on a boolean expression. Here is an example:

```
<div *ngIf="isLoggedIn">
<p>Welcome back, user!</p>
</div>
<div *ngIf="!isLoggedIn">
<p>Please log in.</p>
</div>
```

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

@Component({
  selector: 'app-login-status',
  templateUrl: './login-status.component.html'
})
export class LoginStatusComponent {
```



```
    isLoggedIn: boolean = false;

    toggleLogin() {
      this.isLoggedIn = !this.isLoggedIn;
    }
  }
```

3.1.2 *ngFor

The *ngFor directive is used to iterate over a collection and render a template for each item. Here is an example:

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

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

@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html'
})
export class ItemListComponent {
  items = ['Item 1', 'Item 2', 'Item 3'];
}
```

3.1.3 *ngSwitch

The *ngSwitch directive is used to display one template from a set of possible templates based on the value of an expression. Here is an example:

```
<div [ngSwitch]="color">
<p *ngSwitchCase="'red'">Color is Red</p>
<p *ngSwitchCase="'green'">Color is Green</p>
<p *ngSwitchCase="'blue'">Color is Blue</p>
<p *ngSwitchDefault>Color is Unknown</p>
</div>
```

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

@Component({
  selector: 'app-color-switch',
  templateUrl: './color-switch.component.html'
})
export class ColorSwitchComponent {
  color: string = 'red';
}
```

3.1.4 Combining Control Flow Directives

You can combine these directives to create more complex control flows. Here is an example:


```
<div *ngIf="items.length > 0; else noItems">
<ul>
<li *ngFor="let item of items">{{ item }}</li>
</ul>
</div>
<ng-template #noItems>
<p>No items available.</p>
</ng-template>
```

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

@Component({
  selector: 'app-conditional-list',
  templateUrl: './conditional-list.component.html'
})
export class ConditionalListComponent {
  items: string[] = [];
}
```

3.2 Other Directives

Angular also provides a variety of structural and attribute directives beyond the commonly used control directives discussed earlier. Here are some other useful Angular directives:

- **`ngClass`**: Dynamically adds or removes CSS classes.
- **`ngStyle`**: Sets multiple inline styles dynamically.
- **`ngModel`**: Implements two-way data binding.
- **`ngTemplateOutlet`**: Embeds a template dynamically.
- **`ngContainer`**: Groups elements without adding extra DOM elements.
- **`ngPlural`**: Displays different messages based on a numeric value.

3.2.1 `ngClass`

The `ngClass` directive allows you to dynamically add or remove CSS classes to an element based on an expression. Here is an example:

```
<div [ngClass]="{ 'active': isActive, 'disabled': isDisabled }">
This div's classes are dynamic!
</div>
```

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

@Component({
  selector: 'app-dynamic-class',
  templateUrl: './dynamic-class.component.html'
})
```



```
export class DynamicClassComponent {
  isActive: boolean = true;
  isDisabled: boolean = false;
}
```

3.2.2 ngStyle

The `ngStyle` directive allows you to set multiple inline styles dynamically. Here is an example:

```
<div [ngStyle]="{ 'color': isActive ? 'blue' : 'black', 'font-size': fontSize +
  'px' }">
This div's style is dynamic!
</div>
```

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

@Component({
  selector: 'app-dynamic-style',
  templateUrl: './dynamic-style.component.html'
})
export class DynamicStyleComponent {
  isActive: boolean = true;
  fontSize: number = 14;
}
```

3.2.3 ngModel

The `ngModel` directive is used for two-way data binding in forms. Here is an example:

```
<input [(ngModel)]="name" placeholder="Enter your name">
<p>Hello, {{ name }}!</p>
```

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

@Component({
  selector: 'app-two-way-binding',
  templateUrl: './two-way-binding.component.html'
})
export class TwoWayBindingComponent {
  name: string = '';
}
```

3.2.4 ngTemplateOutlet

The `ngTemplateOutlet` directive is used to embed a template dynamically. Here is an example:

```
<ng-container *ngTemplateOutlet="myTemplate; context: myContext"></ng-container>

<ng-template #myTemplate let-name="name">
<p>Hello, {{ name }}!</p>
</ng-template>
```



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

@Component({
  selector: 'app-template-outlet',
  templateUrl: './template-outlet.component.html'
})
export class TemplateOutletComponent {
  myContext = { name: 'Angular' };
}
```

3.3 ngContainer

The `ngContainer` directive is a logical container that can be used to group elements without adding extra DOM elements. Here is an example:

```html
<ng-container *ngIf="isVisible">
<p>This paragraph is only visible if isVisible is true.</p>
</ng-container>
```

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

@Component({
  selector: 'app-ng-container',
  templateUrl: './ng-container.component.html'
})
export class NgContainerComponent {
  isVisible: boolean = true;
}
```

3.4 ngPlural

The `ngPlural` directive is used for displaying different messages based on a numeric value. Here is an example:

```html
<ng-container [ngPlural]="value">
<ng-template ngPluralCase="=0">There are no items.</ng-template>
<ng-template ngPluralCase="=1">There is one item.</ng-template>
<ng-template ngPluralCase="other">There are {{ value }} items.</ng-template>
</ng-container>
```

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

@Component({
  selector: 'app-plural',
  templateUrl: './plural.component.html'
})
export class PluralComponent {
  value: number = 2;
}
```

Chapter 4

Data and Event Binding

Data binding and event binding enable communication between the component class and its template. These bindings enable efficient and reactive communication between the component and the view, making Angular applications dynamic and responsive:

- **Interpolation** ({{}}): Binds data from the component to the template.
- **Property Binding** (**`[property]="expression"`**): Binds an element's property to a value.
- **Attribute Binding** (**`[attr.attributeName]="expression"`**): Binds an attribute's value.
- **Class Binding** (**`[class.className]="expression"`**): Adds or removes CSS classes.
- **Style Binding** (**`[style.styleName]="expression"`**): Sets the style of an element.
- **Event Binding** (**`(event)="method()"`**): Listens to events and triggers methods.
- **Two-Way Binding** (**`[(ngModel)]="property"`**): Combines property and event binding for forms.

4.1 Data Binding

Data binding in Angular allows you to synchronize data between the component class and its template. There are several types of data binding:

4.1.1 Interpolation

Interpolation binds data from the component to the template using double curly braces {{}}. Here is an example:

```
<p>{{ message }}</p>
```

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

@Component({
  selector: 'app-interpolation',
  templateUrl: './interpolation.component.html'
})
export class InterpolationComponent {
  message: string = 'Hello, Angular!';
}
```

4.1.2 Property Binding

Property binding binds an element's property to a value from the component. Here is an example:

```html
<input [value]="username">
```

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

@Component({
  selector: 'app-property-binding',
  templateUrl: './property-binding.component.html'
})
export class PropertyBindingComponent {
  username: string = 'AngularUser';
}
```

4.1.3 Attribute Binding

Attribute binding sets the value of an attribute directly. Here is an example:

```html
<button [attr.aria-label]="actionName">Click me</button>
```

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

@Component({
  selector: 'app-attribute-binding',
  templateUrl: './attribute-binding.component.html'
})
export class AttributeBindingComponent {
  actionName: string = 'Perform Action';
}
```

4.1.4 Class Binding

Class binding adds or removes CSS classes. Here is an example:

```html
<div [class.active]="isActive">Active Class Binding</div>
```



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

@Component({
  selector: 'app-class-binding',
  templateUrl: './class-binding.component.html'
})
export class ClassBindingComponent {
  isActive: boolean = true;
}
```

4.1.5 Style Binding

Style binding sets the style of an element. Here is an example:

```html
<p [style.color]="textColor">Styled Text</p>
```

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

@Component({
  selector: 'app-style-binding',
  templateUrl: './style-binding.component.html'
})
export class StyleBindingComponent {
  textColor: string = 'blue';
}
```

4.2 Event Binding

Event binding listens to events in the template and triggers methods in the component. Here is an example:

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

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

@Component({
  selector: 'app-event-binding',
  templateUrl: './event-binding.component.html'
})
export class EventBindingComponent {
  onClick() {
    console.log('Button clicked!');
  }
}
```

In Angular, you can bind to various types of events, similar to standard HTML DOM events. Below we discuss some common types of events you can handle.

4.2.1 Mouse Events

- **click**: Triggered when a mouse button is clicked.

- **dblclick**: Triggered when a mouse button is double-clicked.
- **mousedown**: Triggered when a mouse button is pressed.
- **mouseup**: Triggered when a mouse button is released.
- **mousemove**: Triggered when the mouse pointer is moving.
- **mouseenter**: Triggered when the mouse pointer enters an element.
- **mouseleave**: Triggered when the mouse pointer leaves an element.
- **mouseover**: Triggered when the mouse pointer moves onto an element.
- **mouseout**: Triggered when the mouse pointer moves out of an element.

4.2.2 Keyboard Events

- **keydown**: Triggered when a key is pressed down.
- **keyup**: Triggered when a key is released.
- **keypress**: Triggered when a key is pressed (deprecated in modern browsers, use `keydown` or `keyup` instead).

4.2.3 Focus Events

- **focus**: Triggered when an element receives focus.
- **blur**: Triggered when an element loses focus.

4.2.4 Form Events

- **submit**: Triggered when a form is submitted.
- **change**: Triggered when the value of an element is changed.
- **input**: Triggered when the value of an input element is changed.

4.2.5 Clipboard Events

- **copy**: Triggered when the content is copied.
- **cut**: Triggered when the content is cut.
- **paste**: Triggered when the content is pasted.

4.2.6 Drag and Drop Events

- **drag**: Triggered when an element is being dragged.
- **dragstart**: Triggered when the user starts dragging an element.
- **dragend**: Triggered when the user stops dragging an element.
- **dragenter**: Triggered when the dragged element enters a drop target.
- **dragleave**: Triggered when the dragged element leaves a drop target.
- **dragover**: Triggered when the dragged element is over a drop target.
- **drop**: Triggered when the dragged element is dropped on a drop target.

4.2.7 Touch Events

- **touchstart**: Triggered when a touch point is placed on the touch surface.
- **touchmove**: Triggered when a touch point is moved along the touch surface.
- **touchend**: Triggered when a touch point is removed from the touch surface.
- **touchcancel**: Triggered when a touch point has been disrupted.

4.2.8 Miscellaneous Events

- **resize**: Triggered when the window is resized.
- **scroll**: Triggered when the document view is scrolled.
- **contextmenu**: Triggered when the right mouse button is clicked (to open a context menu).

4.2.9 Another Example of Event Binding

```
<button (click)="onClick()">Click me</button>
<input (keyup)="onKeyUp(\textdollarevent)" placeholder="Type something">
<div (mouseover)="onMouseOver()">Hover over me</div>
```

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

@Component({
  selector: 'app-event-handling',
  templateUrl: './event-handling.component.html'
})
export class EventHandlingComponent {
  onClick() {
    console.log('Button clicked!');
  }

```



```
    onKeyUp(event: KeyboardEvent) {
      console.log('Key pressed:', (event.target as HTMLInputElement).value);
    }

    onMouseOver() {
      console.log('Mouse is over the div!');
    }
  }
```

4.3 Two-Way Data Binding

Two-way data binding combines property binding and event binding using `[(ngModel)]`. Here is an example:

```
<input [(ngModel)]="name" placeholder="Enter your name">
<p>Hello, {{ name }}!</p>
```

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

@Component({
  selector: 'app-two-way-binding',
  templateUrl: './two-way-binding.component.html'
})
export class TwoWayBindingComponent {
  name: string = '';
}
```


Chapter 5

Forms in Angular

Angular Forms enable users to input and submit their data. Angular forms can be created using two main approaches: Template-Driven and Reactive. Template-driven forms are simple and easy to use, relying on Angular's directives, while reactive forms provide more control and flexibility using reactive patterns and explicit management of form states. Key differences are:

- **Template-Driven Forms**:
  - Simpler to implement.
  - Suitable for simpler forms.
  - Uses Angular directives like `ngModel` and `ngForm`.
  - Less control over form validation and handling.
- **Reactive Forms**:
  - More powerful and flexible.
  - Suitable for complex forms.
  - Uses reactive patterns and `FormGroup`, `FormControl`.
  - Offers more control over validation and form state.

5.1 Template-Driven Forms

Template-driven forms rely heavily on Angular's directives to create forms. They are simpler to use but offer less control compared to reactive forms. Here is an example:

```
<form #userForm="ngForm" (ngSubmit)="onSubmit(userForm)">
<div>
<label for="name">Name:</label>
<input type="text" id="name" name="name" ngModel required>
</div>
<div>
<label for="email">Email:</label>
<input type="email" id="email" name="email" ngModel required>
```



```
  </div>
  <button type="submit" [disabled]="!userForm.form.valid">Submit</button>
  </form>
```

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

@Component({
  selector: 'app-template-driven-form',
  templateUrl: './template-driven-form.component.html'
})
export class TemplateDrivenFormComponent {
  onSubmit(form: any) {
    console.log('Form Submitted!', form);
  }
}
```

5.2 Reactive Forms

Reactive forms provide more control and flexibility by using an explicit and immutable approach to managing the state of a form at a given point in time. Here is an example:

```
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">
<div>
<label for="name">Name:</label>
<input type="text" id="name" formControlName="name">
</div>
<div>
<label for="email">Email:</label>
<input type="email" id="email" formControlName="email">
</div>
<button type="submit" [disabled]="!userForm.valid">Submit</button>
</form>
```

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

@Component({
  selector: 'app-reactive-form',
  templateUrl: './reactive-form.component.html'
})
export class ReactiveFormComponent implements OnInit {
  userForm: FormGroup;

  ngOnInit() {
    this.userForm = new FormGroup({
      name: new FormControl('', [Validators.required]),
      email: new FormControl('', [Validators.required, Validators.email])
    });
  }

  onSubmit() {
    console.log('Form Submitted!', this.userForm.value);
  }
}
```


5.3 Form Validation

Both approaches support form validation, but the methods to implement it differ slightly.

5.3.1 Template-Driven Forms Validation Example

```
<input type="email" id="email" name="email" ngModel required email>
<div *ngIf="email.invalid && (email.dirty || email.touched)">
<div *ngIf="email.errors.required">Email is required.</div>
<div *ngIf="email.errors.email">Invalid email format.</div>
</div>
```

5.3.2 Reactive Forms Validation Example

```
<input type="email" id="email" formControlName="email">
<div *ngIf="userForm.get('email').invalid && userForm.get('email').touched">
<div *ngIf="userForm.get('email').errors.required">Email is required.</div>
<div *ngIf="userForm.get('email').errors.email">Invalid email format.</div>
</div>
```

Chapter 6

Services and Dependency Injection

6.1 Services and Dependency Injection

Angular services are singleton objects that can be used to organize and share code across your app.

6.1.1 Creating an Injectable Service

Use the `@Injectable` decorator to make a class injectable. For example:

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

@Injectable({
  providedIn: 'root',
})
export class MyService {
  // Service code here
}
```

6.1.2 Dependency Injection

Angular's dependency injection system provides services to components or other services. For example, you can inject a service into a component:

```
constructor(private myService: MyService) {}
```

Chapter 7

Routing

Learn how to navigate between different views or components in your application using Angular's routing module. For example, to define routes:

```
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 { }
```

Chapter 8

Pipes

Pipes allow you to transform data in your templates. Learn how to use built-in pipes and create custom pipes.

8.1 Using Built in Pipes

Angular provides several built-in pipes, such as `DatePipe UpperCasePipe CurrencyPipe`:

```
<p>{{ today | date }}</p>
<p>{{ amount | currency }}</p>
```

8.2 Creating a Custom Pipe

You can create custom pipes to implement your own data transformations. For example:

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

@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

Chapter 9

Services and Dependency Injection

Services and Dependency Injection (DI) help to write modular, maintainable, and testable code.

- **Services**: Handle business logic, data fetching, and other reusable operations. They are used to share data and functions among different parts of your application.
- **Dependency Injection**: A design pattern where a class requests dependencies from external sources. Angular's DI system provides dependencies to components and services automatically.

9.1 Services in Angular

Services are classes that handle business logic, data fetching, and other reusable operations. They are used to share data and functions among different parts of your application.To create a service, use the Angular CLI execute:

```
ng generate service my-service
```

This command generates a service file `my-service.service.ts`. Here is an example of a dummy service:

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

@Injectable({
  providedIn: 'root'
})
export class MyService {
  private data: string[] = [];

  constructor() {}

  addData(item: string) {
    this.data.push(item);
  }

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

```
}
```

9.2 Dependency Injection (DI)

Dependency Injection is a design pattern in which a class requests dependencies from external sources rather than creating them itself. DI is used to implement IoC (Inversion of Control), allowing the creation of dependent objects outside of a class and providing those objects to a class in various ways at run-time. Benefits of Dependency Injection are:

- Promotes code reusability and maintainability
- Decouples the creation of objects from their dependencies
- Facilitates testing by allowing dependencies to be mocked or stubbed

To create an injectable service, use the `@Injectable` decorator. For example:

```

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

@Injectable({
  providedIn: 'root',
})
export class MyService {
  constructor() {}

  getData() {
    return [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' },
    { id: 3, name: 'Item 3' },
    ];
  }
}
```

Angular's DI system provides dependencies to components and services automatically. To use a service in a component, you inject it through the component's constructor. Here is an example:

```


import { Component, OnInit } from '@angular/core';
import { MyService } from './my-service.service';

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.css']
})
export class MyComponent implements OnInit {
  items: any[];

  constructor(private myService: MyService) {
    // get data using the service method
```



```
    this.items = this.myService.getData();
  }

  ngOnInit(): void {
    // Any additional initialization logic can go here
  }
}
```

```
<div>
<input #itemInput type="text" placeholder="Enter item">
<button (click)="addItem(itemInput.value)">Add Item</button>
</div>
<ul>
<li *ngFor="let item of items">{{ item }}</li>
</ul>
```

9.3 Providing Services

Angular services can be provided in different ways:

9.3.1 Provided in Root

The service is available application-wide and a single instance is created.

```
@Injectable({
  providedIn: 'root'
})
export class MyService { ... }
```

9.3.2 Provided in Module

The service is available to all components declared in the module.

```
@NgModule({
  ...
  providers: [MyService]
})
export class AppModule { ... }
```

9.3.3 Provided in Component

A new instance of the service is created for each component.

```
@Component({
  ...
  providers: [MyService]
})
export class MyComponent { ... }
```


Chapter 10

@Input and @Output Decorators

The `@Input` decorator allows a parent component to pass data to a child component. This is used to bind a property in the parent component to a property in the child component. The `@Output` decorator allows a child component to emit events that a parent component can listen to. This is used to send data from the child component back to the parent component via event binding. Below are some examples.

In this example, the `parentProperty` in the `ParentComponent` is passed to the `ChildComponent` via the `childProperty` input property:

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

@Component({
  selector: 'app-child',
  template: `
  <p>{{ childProperty }}</p>
  `
})
export class ChildComponent {
  @Input() childProperty: string;
}
```

Now bind the property in the parent component:

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

@Component({
  selector: 'app-parent',
  template: `
  <app-child [childProperty]="parentProperty"></app-child>
  `
})
export class ParentComponent {
  parentProperty = 'Hello from Parent';
}
```

Let's have another example focused on `@Output()`: Here the `ChildComponent` emits an event through the `dataEmitter` property, which the `ParentComponent` listens to via the `(dataEmitter)` event binding. The parent component's `receiveData` method is called when the event is emitted, allowing data to flow from the child to the parent.

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

```
2
3   @Component({
4     selector: 'app-child',
5     template: `
6     <button (click)="sendData()">Send Data</button>
7     `
8   })
9   export class ChildComponent {
10    @Output() dataEmitter = new EventEmitter<string>();
11
12    sendData() {
13      this.dataEmitter.emit('Hello from Child');
14    }
15  }
```

Now listen to the event in the parent component:

```
1   import { Component } from '@angular/core';
2
3   @Component({
4     selector: 'app-parent',
5     template: `
6     <app-child (dataEmitter)="receiveData($event)"></app-child>
7     <p>{{ receivedData }}</p>
8     `
9   })
10  export class ParentComponent {
11    receivedData: string;
12
13    receiveData(data: string) {
14      this.receivedData = data;
15    }
16  }
```


Chapter 11

Routing

Angular Routing is a mechanism for navigating between different views or components. Angular's Router enables navigation from one view to the next as users perform application tasks. You need to take below steps to have proper routing in Angular:

- **Setting Up Routes**: Define routes in `app-routing.module.ts` and use `RouterModule` to configure them.
- **Route Parameters**: Pass and access parameters in routes using Angular's `ActivatedRoute`.
- **Route Guards**: Protect routes using guards like `CanActivate`.
- **Lazy Loading**: Load modules on demand to improve application performance. We will discuss this in details later on.

Here is an example. Ensure to enable routes while generating a new Angular application:

```
ng new my-angular-app
cd my-angular-app
ng generate component home
ng generate component about
ng generate component contact
```

Now import the `RouterModule` and define routes in `app-routing.module.ts`:

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

const routes: Routes = [
{ path: '', redirectTo: '/home', pathMatch: 'full' },
{ path: 'home', component: HomeComponent },
{ path: 'about', component: AboutComponent },
{ path: 'contact', component: ContactComponent }
];

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



```
})
export class AppRoutingModule { }
```

Now include `AppRoutingModule` in the `AppModule`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';

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

Now define navigation links in `app.component.html`:

```
<nav>
<a routerLink="/home">Home</a>
<a routerLink="/about">About</a>
<a routerLink="/contact">Contact</a>
</nav>
<router-outlet></router-outlet>
```

11.1 Route Parameters

Routes can have parameters, allowing you to pass data in the URL. Here is an example to define a route with parameters in `app-routing.module.ts`:

```
const routes: Routes = [
{ path: 'profile/:id', component: ProfileComponent }
];
```

And this is how to access the route parameters in your component:

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

@Component({
  selector: 'app-profile',
  templateUrl: './profile.component.html',
  styleUrls: ['./profile.component.css']
```



```
})
export class ProfileComponent implements OnInit {
  id: string;

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.id = this.route.snapshot.paramMap.get('id');
  }
}
```

11.2 Route Guards

Route guards determine if a route can be activated or deactivated, protecting routes from unauthorized access. Here is an example to use guards:

```
ng generate guard auth
```

Now implement the guard logic in `auth.guard.ts`:

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

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  constructor(private router: Router) {}

  canActivate(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean |
   UrlTree> | boolean | UrlTree {
    const isAuthenticated = false; // Replace with actual authentication check
    if (isAuthenticated) {
      return true;
    } else {
      this.router.navigate(['/login']);
      return false;
    }
  }
}
```

And finally apply the guard to routes in `app-routing.module.ts`

```
const routes: Routes = [
{ path: 'profile/:id', component: ProfileComponent, canActivate: [AuthGuard] }
];
```

11.3 Lazy Loading for Routes

Lazy loading allows you to load feature modules only when they are needed, reducing the initial load time. Please note we have a dedicated chapter ro lazy loading , however, here is a quick example focused on routes:

```
ng generate module feature --route feature --module app.module
```

Now configure lazy loading in `app-routing.module.ts`

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


Chapter 12

Pipes

Angular Pipes allow you to transform data directly in your templates. They are similar to filters in other frameworks. Angular comes with several built-in pipes, and you can also create custom pipes to handle specific data transformations. Here are popular pipes that you may encounter:

- **Built-in Pipes**: Angular provides several built-in pipes for common data transformations like date formatting, text case transformation, currency formatting, etc.
- **Custom Pipes**: You can create custom pipes to handle specific transformations that are not covered by built-in pipes.
- **Pipe Parameters**: Pipes can accept parameters to modify their behavior.
- **Pure and Impure Pipes**: Pure pipes are stateless and efficient, while impure pipes can handle more complex scenarios but may affect performance.

12.1 Built-in Pipes

Angular provides a variety of built-in pipes that cover common data transformation needs:

- **DatePipe**: Formats dates.
- **UpperCasePipe** and **LowerCasePipe**: Transforms text to upper or lower case.
- **CurrencyPipe**: Formats numbers as currency.
- **DecimalPipe**: Formats numbers with decimal points.
- **PercentPipe**: Formats numbers as percentages.
- **JsonPipe**: Converts an object into a JSON string.
- **SlicePipe**: Creates a slice of an array.

Here are some examples:

- **DatePipe**:

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

```

- **UpperCasePipe**:

```
    <p>{{ 'hello world' | uppercase }}</p>

```

- **CurrencyPipe**:

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

```

12.2 Creating Custom Pipes

Sometimes, built-in pipes are not enough for your needs, and you need to create custom pipes. Custom pipes are simple to create and use. Here are the steps needed:

```
ng generate pipe custom
```

Next, edit the generated `custom.pipe.ts` file:

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

@Pipe({
  name: 'custom'
})
export class CustomPipe implements PipeTransform {

  transform(value: string, ...args: any[]): string {
    // Example transformation: reverse the input string
    return value.split('').reverse().join('');
  }

}
```

And finally use the custom pipe in your template:

```
<p>{{ 'Angular' | custom }}</p> <!-- Output: ralugnA -->
```

12.3 Pipe Parameters

Pipes can accept parameters to influence their behavior. These parameters are passed after the pipe name, separated by colons. Here are some examples using the built-in `SlicePipe` with parameters:

```
<p>{{ 'Angular' | slice:1:4 }}</p> <!-- Output: ngu -->
```


12.4 Pure and Impure Pipes

- **Pure Pipes**: Only re-evaluate when the input data changes. They are stateless and perform well. By default, all pipes are pure.
- **Impure Pipes**: Re-evaluate on every change detection cycle. They can handle more complex transformations but may impact performance.

To create an impure pipe, set the `pure` property to `false` in the `@Pipe` decorator:

```
@Pipe({
  name: 'impurePipe',
  pure: false
})
export class ImpurePipe implements PipeTransform {
  transform(value: any): any {
    // transformation logic
  }
}
```


Chapter 13

Reactive Extensions for JavaScript in Angular Framework

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using Observables, which makes it easier to compose asynchronous or callback-based code. In Angular, RxJS is used extensively for handling asynchronous operations like HTTP requests, user inputs, and more.

13.1 Basic Concepts

- **Observable**: Represents a stream of data that can be observed over time.
- **Observer**: An object with `next`, `error`, and `complete` methods to handle data from the Observable.
- **Subscription**: Allow you to subscribe to an Observable and handle its data. Subscription represents the execution of an Observable, allowing you to unsubscribe and stop receiving data.
- **Operators**: Functions that enable functional programming and composition in observables, like `map`, `filter`, `merge`, etc.

Let's create a simple Observable that emits a sequence of numbers and subscribes to it. Here is an Observable code:

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

const numberObservable = new Observable<number>(observer => {
  let count = 0;
  const interval = setInterval(() => {
    observer.next(count++);
    if (count > 5) {
      observer.complete();
    }
  }, 1000);

  return () => clearInterval(interval);
```



```
});
```

This is how we Subscribe to the Observable:

```
const subscription = numberObservable.subscribe({
  next: num => console.log('Number:', num),
  complete: () => console.log('Observable complete'),
  error: err => console.error('Error:', err)
});

// Unsubscribe after 7 seconds to clean up
setTimeout(() => subscription.unsubscribe(), 7000);
```

13.2 Using RxJS Operators

RxJS operators are used to transform, filter, and combine observables. Here are some examples that filters out odd numbers and multiplies each remaining number by 10:

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

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

numbers.pipe(
filter(num => num % 2 === 0),
map(num => num * 10)
).subscribe(result => console.log('Result:', result));
```


Chapter 14

Angular HTTP Client

The Angular HTTP client, provided by the `HttpClient` module, is a powerful and flexible way to communicate with backend services over HTTP. It allows you to perform various HTTP requests, such as `get`, `post`, `put`, `delete` and more. Angular uses RxJS operators to process responses and handle errors.

14.1 Setting Up the Angular HTTP Client

First, you need to import the `HttpClientModule` in your Angular module to enable the HTTP client functionality.

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

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

To use the `HttpClient`, you need to inject it into your service or component.

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable({
  providedIn: 'root',
})
export class DataService {
  private apiUrl = 'https://api.example.com/data';

  constructor(private http: HttpClient) {}

  getData(): Observable<any> {
```

```
    return this.http.get<any>(this.apiUrl).pipe(
    catchError(this.handleError)
    );
  }

  postData(data: any): Observable<any> {
    return this.http.post<any>(this.apiUrl, data).pipe(
    catchError(this.handleError)
    );
  }

  private handleError(error: HttpErrorResponse) {
    // Handle the error here
    return throwError('An error occurred');
  }
}
```

For HTTP GET Requests:

```
this.dataService.getData().subscribe(
data => console.log(data),
error => console.error(error)
);
```

For HTTP POST Requests:

```
const newData = { name: 'New Item' };
this.dataService.postData(newData).subscribe(
data => console.log(data),
error => console.error(error)
);
```

14.2 Handling HTTP Responses

You can handle responses using RxJS operators like `map`, `catchError`, `tap`, etc. The `catchError` operator is useful for handling errors:

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

getData(): Observable<any> {
  return this.http.get<any>(this.apiUrl).pipe(
  map(response => response.data), // Process the response
  catchError(this.handleError)
  );
}
```

14.3 HTTP Interceptors

HTTP interceptors are used to modify HTTP requests or responses. You can use them for adding authentication tokens, logging, error handling, etc.

Here is how to create a HTTP interceptor:


```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/
  common/http';
import { Observable } from 'rxjs';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any
  >> {
    const authToken = 'your-auth-token';
    const authReq = req.clone({
      headers: req.headers.set('Authorization', `Bearer ${authToken}`)
    });

    return next.handle(authReq);
  }
}
```

Ensure to add the provider in AppModule for the Interceptor:

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

@NgModule({
  providers: [
  { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
  ]
})
export class AppModule {}
```

For more information, refer to the official guide on HTTP client at Angular HTTP Client Guide.

Chapter 15

WebSocket Client

WebSockets provide a full-duplex communication channel over a single, long-lived connection, allowing for real-time data transfer between the client and server. In Angular, you can use WebSockets with RxJS to handle asynchronous streams of data.

First, ensure that you have Angular and RxJS set up in your project. If not, you can create a new Angular project using Angular CLI.

```
ng new websocket-example
cd websocket-example
npm install rxjs
```

You need to create a service that will handle the WebSocket connection. This service will use RxJS to manage the WebSocket connection and provide a stream of messages. Below is a sample websocket.service.ts:

```
import { Injectable } from '@angular/core';
import { Observable, Subject, webSocket } from 'rxjs';
import { WebSocketSubject } from 'rxjs/webSocket';

@Injectable({
  providedIn: 'root'
})
export class WebsocketService {
  private socket$: WebSocketSubject<any>;

  constructor() {
    this.socket$ = webSocket('ws://localhost:8080'); // Replace with your
   WebSocket server URL
  }

  public connect(): Subject<any> {
    return this.socket$;
  }

  public sendMessage(message: any): void {
    this.socket$.next(message);
  }

  public close(): void {
    this.socket\textdollar.complete();
  }
}
```

In this example:

- `webSocket` from RxJS creates a WebSocket subject.
- `connect` returns the WebSocket subject as an observable.
- `sendMessage` sends a message to the WebSocket server.
- `close` closes the WebSocket connection.

Next, use the WebSocket service in a component to connect to the WebSocket server and handle incoming messages. In app.component.ts perform:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { WebsocketService } from './websocket.service';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-root',
  template: `
  <div>
  <h1>WebSocket Example</h1>
  <button (click)="sendMessage()">Send Message</button>
  <ul>
  <li *ngFor="let message of messages">{{ message }}</li>
  </ul>
  </div>
  `,
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit, OnDestroy {
  messages: string[] = [];
  private subscription: Subscription;

  constructor(private websocketService: WebsocketService) {}

  ngOnInit() {
    this.subscription = this.websocketService.connect().subscribe(
    message => {
      this.messages.push(message);
    },
    err => console.error(err),
    () => console.warn('Completed!')
    );
  }

  sendMessage() {
    this.websocketService.sendMessage('Hello from Angular');
  }

  ngOnDestroy() {
    this.subscription.unsubscribe();
    this.websocketService.close();
  }
}
```


In this example:

- `ngOnInit` subscribes to the WebSocket connection and adds incoming messages to the `messages` array.
- `sendMessage` sends a message to the WebSocket server.
- `ngOnDestroy` unsubscribes from the WebSocket connection and closes it when the component is destroyed.

Chapter 16

Deferrable Views and Services

Deferrable Views in Angular refer to a strategy to improve the performance and user experience of an application by loading parts of the user interface on demand, rather than loading everything upfront. This technique is particularly useful in large applications where certain components or views are not needed immediately and can be loaded later based on user interaction. We need to first understand Lazy Loading and Defer Directives:

- **Lazy Loading**: Loading modules or components on demand rather than at the initial load time.
- **Defer Directive**: A hypothetical directive that could be used to defer the loading of a component until it is needed.
- **Deferrable Views**: Strategy to improve performance by loading parts of the UI on demand via techniques such as Lazy Loading.

16.1 Lazy Loading

Lazy loading is a common technique used to implement deferrable views. In Angular, you can lazy load modules by configuring the Angular router. Here is an example to set up lazy loading which `featureA` and `featureB` modules will only be loaded when the user navigates to their respective routes.

```
ng generate module featureA --route featureA --module app.module
ng generate module featureB --route featureB --module app.module
```

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
{ path: 'featureA', loadChildren: () => import('./featureA/featureA.module').
  then(m => m.FeatureAModule) },
{ path: 'featureB', loadChildren: () => import('./featureB/featureB.module').
  then(m => m.FeatureBModule) },
{ path: '', redirectTo: '/home', pathMatch: 'full' },
{ path: '**', redirectTo: '/home' }
];
```

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

16.2 Using the Defer Directive

While Angular does not have a built-in `defer` directive, you can achieve a similar effect by creating custom directives or using existing libraries. Below is an example of how you might implement a custom directive to defer the loading of a component:

```
ng generate directive defer
```

Next, edit `defer.directive.ts` to create a directive that defers the initialization of a component until it becomes visible.

```
import { Directive, Input, TemplateRef, ViewContainerRef, OnInit, AfterViewInit
  } from '@angular/core';

@Directive({
  selector: '[appDefer]'
})
export class DeferDirective implements AfterViewInit {
  @Input() appDefer: boolean;

  constructor(private tpl: TemplateRef<any>, private vcr: ViewContainerRef) { }

  ngAfterViewInit() {
    if (this.appDefer) {
      const observer = new IntersectionObserver(entries => {
        entries.forEach(entry => {
          if (entry.isIntersecting) {
            this.vcr.createEmbeddedView(this.tpl);
            observer.unobserve(entry.target);
          }
        });
      });
      observer.observe(this.vcr.element.nativeElement);
    } else {
      this.vcr.createEmbeddedView(this.tpl);
    }
  }
}
```

In order to use this directive in HTML templates, apply the directive to defer the rendering of a component. Basically the `app-heavy-component` will only be instantiated and rendered when it becomes visible in the viewport.

```
<ng-container *appDefer="true">
<app-heavy-component></app-heavy-component>
</ng-container>
```


16.3 Best Practices

- **Identify Deferrable Views**: Determine which parts of your application can be deferred without affecting the initial user experience.
- **Combine Techniques**: Use a combination of lazy loading and custom defer directives to optimize the performance.
- **Monitor Performance**: Continuously monitor the performance and user experience to ensure that deferring views are improving the application.

16.4 Optimizing Lazy Loaded Modules

16.4.1 Preloading Strategies

Use preloading strategies to load modules in the background after the initial load:

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

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

For more information, refer to the official guide on Lazy Loading at Angular Lazy Loading Guide.

Chapter 17

Testing in Angular

Testing is a crucial part of developing robust Angular applications. We usually try to isolate tests, use mocks, keep tests fast, write self-descriptive tests, for our best to have high test coverage, and integrate tests into our CI/CD pipeline. Angular provides a comprehensive testing framework that includes tools for unit testing, integration testing, and end-to-end (E2E) testing. Angular projects generated with Angular CLI come pre-configured with Jasmine, Karma, and Protractor for testing. Here's a brief overview of each:

- **Unit Testing**: Use Jasmine and Karma to write and run unit tests for components, services, and other units of code.
- **E2E Testing**: Use Protractor to write and run end-to-end tests that simulate user interactions.
- **Jasmine**: A behavior-driven development framework for writing unit tests.
- **Karma**: A test runner that runs tests in multiple browsers.
- **Protractor**: An end-to-end test framework for Angular applications.

You can run tests using the Angular CLI commands:

```
ng test          # Runs unit tests
ng e2e           # Runs end-to-end tests
```

17.1 Unit Testing with Jasmine and Karma

Unit tests in Angular are typically written using Jasmine. They focus on testing individual components, services, and other units of code. As an example consider a simple `CounterComponent`:

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

@Component({
  selector: 'app-counter',
  template: `
  <button (click)="increment()">Increment</button>
```

```
    <span>{{ count }}</span>
    `
  })
  export class CounterComponent {
    count = 0;

    increment() {
      this.count++;
    }
  }
```

The corresponding unit test might look like this:

```
  import { ComponentFixture, TestBed } from '@angular/core/testing';
  import { CounterComponent } from './counter.component';

  describe('CounterComponent', () => {
    let component: CounterComponent;
    let fixture: ComponentFixture<CounterComponent>;

    beforeEach(async () => {
      await TestBed.configureTestingModule({
        declarations: [ CounterComponent ]
      })
      .compileComponents();
    });

    beforeEach(() => {
      fixture = TestBed.createComponent(CounterComponent);
      component = fixture.componentInstance;
      fixture.detectChanges();
    });

    it('should create', () => {
      expect(component).toBeTruthy();
    });

    it('should increment count', () => {
      component.increment();
      expect(component.count).toBe(1);
    });
  });
```

17.1.1 Testing Services

Services are typically tested using mocks and spies. Here's an example of a simple service and its test:

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

  @Injectable({
    providedIn: 'root'
  })
  export class DataService {
    getData(): string {
      return 'data';
    }
```



```
}
```

The corresponding unit test might look like this:

```
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();
  });

  it('should return data', () => {
    expect(service.getData()).toBe('data');
  });
});
```

17.2 End-to-End (E2E) Testing with Protractor

E2E tests simulate user interactions with the application. Protractor is the default E2E test framework for Angular. Here's an example of a simple Protractor test:

```
import { browser, by, element } from 'protractor';

describe('workspace-project App', () => {
  it('should display welcome message', () => {
    browser.get('/');
    expect(element(by.css('app-root h1')).getText()).toEqual('Welcome to app!');
  });
});
```

Now, run E2E tests using the Angular CLI:

```
ng e2e
```

Please ensure to:

- **Isolate Unit Tests**: Ensure unit tests are isolated and test a single piece of functionality.
- **Use Mocks and Spies**: Mock dependencies and use spies to avoid testing external code.
- **Keep Tests Fast**: Unit tests should be fast to run; avoid making HTTP requests or using real services.
- **Write Descriptive Tests**: Test names should clearly describe what the test is verifying.

- **Test Coverage**: Aim for high test coverage but ensure tests are meaningful and cover critical paths.
- **Continuous Integration**: Integrate tests into your CI/CD pipeline to ensure tests run on every commit.

Chapter 18

Advanced Topics: Hierarchical Injectors and Injection Tokens

Angular's injectors are organized in a hierarchy, allowing child components to inherit services from parent components or modules. Injection tokens are unique identifiers for dependencies, useful for injecting non-class-based values or for handling multiple providers for a single dependency. These concepts enhance Angular's flexibility and modularity, allowing for complex dependency management across large applications.

18.1 Hierarchical Injectors

In Angular, injectors are used to manage the creation and delivery of services (dependencies) to components and other services. Angular uses a hierarchical dependency injection system, meaning injectors are organized in a tree structure that parallels the component tree:

- **Root Injector**:
  - The root injector is created when the application is bootstrapped. It can provide services to any component or service throughout the application.
  - Services provided at the root level are singleton by default, meaning there is only one instance of the service for the entire application.
- **Child Injectors**:
  - Every component in Angular has its own injector, which can provide services to that component and its child components.
  - If a child component requests a service, Angular first checks the child component's injector. If the service is not found, Angular then checks the parent injector, and so on up to the root injector.

Below example, assumes that ChildComponent the child of ParentComponent:

- If `ParentComponent` provides a service, `ChildComponent` can use that service, but `AnotherComponent` cannot.

- If the service is provided in `AppComponent` (or at the root level), all components (`ParentComponent`, `ChildComponent`, and `AnotherComponent`) can use it.

18.2 Injection Tokens

Injection tokens are a way to uniquely identify dependencies in Angular's DI system. They are particularly useful when you need to inject non-class dependencies or when multiple providers might match a single token:

- **Defining an Injection Token**:
  - You can create an injection token using the `InjectionToken` class from `@angular/core`.
  - This token can be used to inject values that are not class-based (e.g., configuration objects, constants).
- **Providing an Injection Token**:
  - You provide a value for the injection token in the providers array of a module, component, or directive.
- **Injecting Using an Injection Token**:
  - You inject the value associated with the token using Angular's DI system, typically in a constructor.

Here is an example to define an injection token for a configuration object:

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

// Define an injection token
export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');

// Define the configuration object interface
export interface AppConfig {
  apiUrl: string;
  timeout: number;
}

// Provide the configuration object
export const AppConfigValue: AppConfig = {
  apiUrl: 'https://api.example.com',
  timeout: 3000,
};
```

Now, provide this token in your module:

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

@NgModule({
```



```
    declarations: [AppComponent],
    imports: [BrowserModule],
    providers: [{ provide: APP_CONFIG, useValue: AppConfigValue }],
    bootstrap: [AppComponent]
})
export class AppModule {}
```

Finally, inject the configuration object in a component:

```
import { Component, Inject } from '@angular/core';
import { APP_CONFIG, AppConfig } from './app.config';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(@Inject(APP_CONFIG) private config: AppConfig) {
    console.log(this.config.apiUrl);  // Outputs: https://api.example.com
  }
}
```


Chapter 19

Advanced Topics: Server-Side Rendering

Server-Side Rendering (SSR) in Angular is a technique used to render the initial HTML of an Angular application on the server, and then send it to the client. This enhances the performance, improves the SEO, and provides a better user experience, especially for slow network conditions.

19.1 Benefits of SSR

- **Improved Performance**: Initial load time is reduced since the server sends a fully rendered page.
- **SEO Optimization**: Search engines can crawl and index the fully rendered HTML, improving the discoverability of the application.
- **Faster Time-to-Interactive**: The user can see the content faster, leading to a better user experience.

19.2 Setting Up Angular Universal for SSR

To set up SSR in an Angular application, you need to use Angular Universal and add it to an existing project:

```
1 ng add @nguniversal/express-engine
```

This command will configure your project for SSR with Angular Universal using an Express server. It will add necessary dependencies and files to your project. After running the above command, you will notice several changes and additions in your project:

- **server.ts**: The main server file that uses Express to serve your Angular application.
- **main.server.ts**: The main entry point for the server-side application.

- **angular.json**: Updated build configuration to include server and browser targets.

Here is an example for server.ts:

```
import 'zone.js/dist/zone-node';
import { ngExpressEngine } from '@nguniversal/express-engine';
import * as express from 'express';
import { join } from 'path';

import { AppServerModule } from './src/main.server';
import { APP_BASE_HREF } from '@angular/common';
import { existsSync } from 'fs';

const app = express();
const PORT = process.env.PORT || 4000;
const DIST_FOLDER = join(process.cwd(), 'dist/my-angular-app/browser');

app.engine('html', ngExpressEngine({
  bootstrap: AppServerModule,
}));

app.set('view engine', 'html');
app.set('views', DIST_FOLDER);

app.get('*.*', express.static(DIST_FOLDER, {
  maxAge: '1y'
}));

app.get('*', (req, res) => {
  res.render('index', { req, providers: [{ provide: APP_BASE_HREF, useValue: req
  .baseUrl }] });
});

app.listen(PORT, () => {
  console.log(`Node Express server listening on http://localhost:\textdollar{
  PORT}`);
});
```

Please note we have to build and running the application in production mode for both frontend client and server-side SSR targets:

```
ng build --prod
ng run your-project-name:server
```

Now serve the application using the Angular Universal Express server:

```
node dist/your-project-name/server/main.js
```

19.3 Angular Universal Transfer State and Preboot

As we mentioned, Angular Universal is a technology for SSR in Angular. It pre-renders the application on the server and sends the HTML to the client. To optimize performance, Angular Universal allows you to transfer the state from the server to the client. This avoids making duplicate HTTP requests on the client side. Also, Preboot captures and replays events that occur during server-side rendering to improve the user experience.

Below setup demonstrates how to implement Angular Universal with Transfer State and Preboot to optimize performance and enhance user experience. By transferring the state from the server to the client, you avoid duplicate HTTP requests, and by capturing and replaying events with Preboot, you make the application interactive as quickly as possible. We first need to add Angular Universal to our existing Angular project:

```
ng add @nguniversal/express-engine
```

Now to enable Transfer State, first create a data service to fetch data from an API.

```
ng generate service data
```

```
% data.service.ts:
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
import { TransferState, makeStateKey } from '@angular/platform-browser';

const DATA_KEY = makeStateKey('data');

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

  fetchData(): Observable<any> {
    if (this.transferState.hasKey(DATA_KEY)) {
      const data = this.transferState.get(DATA_KEY, null);
      this.transferState.remove(DATA_KEY);
      return data;
    } else {
      return this.http.get('https://api.example.com/data').pipe(
      map(data => {
        this.transferState.set(DATA_KEY, data);
        return data;
      })
      );
    }
  }
}
```

Use the `DataService` in a component to fetch and display data.

```
ng generate component data-view
```

```
% data-view.component.ts:
import { Component, OnInit } from '@angular/core';
import { DataService } from '../data.service';

@Component({
  selector: 'app-data-view',
  templateUrl: './data-view.component.html'
})
export class DataViewComponent implements OnInit {
  data: any;
```



```
11
12    constructor(private dataService: DataService) {}
13
14    ngOnInit() {
15      this.dataService.fetchData().subscribe(data => {
16        this.data = data;
17      });
18    }
19  }
```

```
1  % data-view.component.html:
2  <div *ngIf="data">
3  <pre>{{ data | json }}</pre>
4  </div>
```

To enable Preboot to capture and replay events that occur during server-side rendering:

```
1  npm install preboot
```

Add Preboot to your server configuration.

```
1  % server.ts:
2  import 'zone.js/node';
3
4  import { ngExpressEngine } from '@nguniversal/express-engine';
5  import { provideModuleMap } from '@nguniversal/module-map-ngfactory-loader';
6  import { enableProdMode } from '@angular/core';
7  import * as express from 'express';
8  import { join } from 'path';
9  import { readFileSync } from 'fs';
10 import { APP_BASE_HREF } from '@angular/common';
11 import { ROUTES } from './static.paths';
12 import 'localstorage-polyfill';
13 import 'globalthis/auto';
14
15 import { AppServerModule } from './src/main.server';
16 import { ngExpressEngine } from '@nguniversal/express-engine';
17 import { provideModuleMap } from '@nguniversal/module-map-ngfactory-loader';
18 import { APP_BASE_HREF } from '@angular/common';
19 import { join } from 'path';
20 import { existsSync } from 'fs';
21
22 import { enableProdMode } from '@angular/core';
23 import 'zone.js/dist/zone-node';
24
25 // Faster server renders w/ Prod mode (dev mode never needed)
26 enableProdMode();
27
28 // Express server
29 const app = express();
30
31 const PORT = process.env.PORT || 4200;
32 const DIST_FOLDER = join(process.cwd(), 'dist/browser');
33
34 // Our Universal express-engine (found @ https://github.com/angular/universal/
     tree/master/modules/express-engine)
35 app.engine('html', ngExpressEngine({
36   bootstrap: AppServerModule,
37 }));
```



```
38
39  app.set('view engine', 'html');
40  app.set('views', DIST_FOLDER);
41
42  // Example Express Rest API endpoints
43  // app.get('/api/**', (req, res) => { });
44  // Serve static files from /browser
45  app.get('*.*', express.static(DIST_FOLDER, {
46    maxAge: '1y'
47  }));
48
49  // All regular routes use the Universal engine
50  app.get('*', (req, res) => {
51    res.render('index', { req });
52  });
53
54  app.listen(PORT, () => {
55    console.log(`Node Express server listening on http://localhost:\textdollar{
      PORT}`);
56  });
```

Now add Preboot configuration to the main server module:

```
1   % app.server.module.ts:
2   import { NgModule } from '@angular/core';
3   import { ServerModule } from '@angular/platform-server';
4   import { ModuleMapLoaderModule } from '@nguniversal/module-map-ngfactory-loader'
      ;
5   import { AppComponent } from './app.component';
6   import { AppModule } from './app.module';
7   import { HTTP_INTERCEPTORS } from '@angular/common/http';
8   import { TransferStateInterceptor } from './transfer-state.interceptor';
9   import { PrebootModule } from 'preboot';
10
11  @NgModule({
12    imports: [
13    AppModule,
14    ServerModule,
15    ModuleMapLoaderModule,
16    PrebootModule.withConfig({ appRoot: 'app-root' })
17    ],
18    providers: [
19    {
20      provide: HTTP_INTERCEPTORS,
21      useClass: TransferStateInterceptor,
22      multi: true
23    }
24    ],
25    bootstrap: [AppComponent]
26  })
27  export class AppServerModule {}
```

```
1   % transfer-state.interceptor.ts:
2   import { Injectable } from '@angular/core';
3   import {
4     HttpEvent,
5     HttpInterceptor,
6     HttpHandler,
7     HttpRequest,
```



```
    HttpResponse
} from '@angular/common/http';
import { Observable, of } from 'rxjs';
import { tap } from 'rxjs/operators';
import { TransferState, makeStateKey } from '@angular/platform-browser';

@Injectable()
export class TransferStateInterceptor implements HttpInterceptor {
  constructor(private transferState: TransferState) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any
  >> {
    if (req.method !== 'GET') {
      return next.handle(req);
    }

    const storedResponse = this.transferState.get(makeStateKey<HttpResponse<any
  >>(req.url), null);
    if (storedResponse) {
      return of(new HttpResponse({
        body: storedResponse.body,
        status: storedResponse.status,
        statusText: storedResponse.statusText,
        headers: storedResponse.headers,
        url: storedResponse.url
      }));
    } else {
      return next.handle(req).pipe(
      tap(event => {
        if (event instanceof HttpResponse) {
          this.transferState.set(makeStateKey<HttpResponse<any>>(req.url), event
  );
        }
      })
      );
    }
  }
}
```

Ensure to have Preboot included in the Main HTML:

```
% src/index.html:
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Angular Universal Transfer State and Preboot Example</title>
<base href="/">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
<app-root></app-root>
<preboot></preboot>
</body>
</html>
```

And finally build and Run the Project


```
npm run build:ssr
npm run serve:ssr
```


Chapter 20

Signals

20.1 Introduction to Signals

Signals allow you to reactively manage state changes and ensure your application responds to these changes efficiently.

20.1.1 Basic Concepts

A signal represents a piece of state that can change over time. You can subscribe to signals to perform actions when the state changes.

20.2 Using Signals in Angular

20.2.1 Creating Signals

Create signals using the `signal` function:

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

const count = signal(0);

function increment() {
  count.set(count() + 1);
}
```

20.2.2 Subscribing to Signals

Subscribe to signals to react to state changes:

```
count.subscribe(newValue => {
  console.log('Count changed to', newValue);
});
```

For more details, refer to the official guide on Signals at Angular Signals Guide.

Chapter 21

Search Engine Optimization (SEO)

21.1 Introduction to SEO

SEO is crucial for driving organic traffic to your web applications. By optimizing your content and website structure, you can enhance the user experience and increase the likelihood of your site appearing in search results.

21.2 Best Practices for SEO in Angular

21.2.1 Server-Side Rendering (SSR)

Using Angular Universal for SSR can improve SEO by rendering your application on the server and providing fully rendered HTML to search engines. This allows search engines to crawl and index your content more effectively.

21.2.2 Meta Tags and Descriptions

Adding meta tags and descriptions to your Angular application helps search engines understand the content of your pages. Use the Angular `Meta` service to dynamically update meta tags:

```
import { Meta } from '@angular/platform-browser';

constructor(private meta: Meta) {
  this.meta.addTags([
  { name: 'description', content: 'This is an example description' },
  { name: 'keywords', content: 'Angular, SEO, example' }
  ]);
}
```

21.2.3 Structured Data

Implementing structured data using JSON-LD can help search engines understand the context of your content. Use the `Meta` service to add structured data to your pages:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "WebSite",
  "name": "Example",
  "url": "https://www.example.com"
}
</script>
```

21.2.4 Canonical URLs

Use canonical URLs to prevent duplicate content issues and consolidate your content's ranking signals. The `Meta` service can also be used to set the canonical URL:

```
import { Meta } from '@angular/platform-browser';

constructor(private meta: Meta) {
  this.meta.addTag({ rel: 'canonical', href: 'https://www.example.com' });
}
```


Chapter 22

Progressive Web Apps (PWA)

Progressive Web Apps combine the best of web and mobile apps. They are built and enhanced with modern APIs to deliver capabilities and reliability while being available via URLs.

22.1 Adding PWA Capabilities to Your Angular App

22.1.1 Installing PWA Support

Add PWA support to your Angular application using the Angular CLI:

```
ng add @angular/pwa
```

This command configures your application with a service worker and updates your manifest file.

22.1.2 Configuring the Service Worker

The service worker caches assets and API responses to enable offline access. Customize the service worker configuration in the `ngsw-config.json` file.

22.2 Building and Deploying PWAs

22.2.1 Building for Production

Build your application for production:

```
ng build --prod
```

Ensure the service worker is properly configured and registered in your application.

22.2.2 Deploying to a Web Server

Deploy the built application to a web server (e.g., Firebase Hosting, Netlify) to make it accessible to users.

For more information, refer to the official guide on Progressive Web Apps at Angular PWA Guide.

Chapter 23

Angular CLI

23.1 Commands and Resources

23.1.1 Installation and Setup

1. **Install Node.js and npm:**
   - Download Node.js
   - `sudo npm install -g @angular/cli`
   - `sudo npm install -g sass`
   - `sudo npm install -g n`
   - `sudo n latest`
2. **Create a New Angular Project:**
   - `ng new angular422 --version=18.1.2 --style=scss --skip-git --routing --package-manager=npm --defaults`
   - `cd angular422; npm install;`

23.1.2 Add Features to Your Project

1. **Add Universal Server-Side Rendering:**
   - `ng add @nguniversal/express-engine`
2. **Add Progressive Web App (PWA) Support:**
   - `ng add @angular/pwa`
3. **Add Angular Material:**
   - `ng add @angular/material`

23.1.3 Generate Angular Elements

1. **Directives and Pipes:**
   - `ng generate directive directive_100`
   - `ng generate pipe pipe_100`
2. **Components, Services, and Modules:**
   - `ng generate component binding-example`
   - `ng generate service data`
   - `ng generate module user`
   - `ng generate component user/binding-example`
3. **Angular Material Components:**
   - `ng generate @angular/material:dashboard --name myDashboard`
   - `ng generate @angular/material:nav --name myNav`
   - `ng generate @angular/material:table --name myTable`

23.1.4 Install Dependencies

1. **General Packages:**
   - `npm install --save preboot`
   - `npm install --save jquery`
   - `npm install --save-dev jquery`
   - `npm install --save rxjs`
   - `npm install --save luxon`
   - `npm install --save lodash`
   - `npm install --save ng-lazyload-image`

23.1.5 Build and Serve

1. **Server-Side Rendering (SSR):**
   - `npm run build:ssr`
   - `npm run serve:ssr`
2. **Build for Production:**
   - `ng build --prod`
3. **Serve Static Files:**
   - `npx http-server ./dist/your-project-name`

23.1.6 Useful Links

- Angular Material Getting Started Guide

23.1.7 Angular CLI Commands

1. **General Generate Commands:**

    - `ng generate <schematic>`: Run the provided schematic.
    - `ng generate app-shell`: Generates an application shell for running a server-side version of an app.
    - `ng generate application [name]`: Generates a new basic application definition in the "projects" subfolder of the workspace.
    - `ng generate class [name]`: Creates a new, generic class definition in the given project.
    - `ng generate component [name]`: Creates a new, generic component definition in the given project.
    - `ng generate config [type]`: Generates a configuration file in the given project.
    - `ng generate directive [name]`: Creates a new, generic directive definition in the given project.
    - `ng generate enum [name]`: Generates a new, generic enum definition in the given project.
    - `ng generate environments`: Generates and configures environment files for a project.
    - `ng generate guard [name]`: Generates a new, generic route guard definition in the given project.
    - `ng generate interceptor [name]`: Creates a new, generic interceptor definition in the given project.
    - `ng generate interface [name] [type]`: Creates a new, generic interface definition in the given project.
    - `ng generate library [name]`: Creates a new, generic library project in the current workspace.
    - `ng generate module [name]`: Creates a new, generic NgModule definition in the given project.
    - `ng generate pipe [name]`: Creates a new, generic pipe definition in the given project.
    - `ng generate resolver [name]`: Generates a new, generic resolver definition in the given project.
    - `ng generate service [name]`: Creates a new, generic service definition in the given project.

- `ng generate service-worker`: Pass this schematic to the "run" command to create a service worker.
- `ng generate web-worker [name]`: Creates a new, generic web worker definition in the given project.

23.2 Angular Material Commands and Resources

23.2.1 Installation and Setup

1. **Add Angular Material to Your Project:**
   - `ng add @angular/material`

23.2.2 Generate Angular Material Components

1. **Generate Dashboard:**
   - `ng generate @angular/material:dashboard --name myDashboard`
2. **Generate Navigation:**
   - `ng generate @angular/material:nav --name myNav`
3. **Generate Table:**
   - `ng generate @angular/material:table --name myTable`
4. **Generate Address Form:**
   - `ng generate @angular/material:address-form --name myAddressForm`
5. **Generate Tree:**
   - `ng generate @angular/material:tree --name myTree`
6. **Generate Drag and Drop:**
   - `ng generate @angular/material:drag-drop --name myDragDrop`
7. **Generate Expansion Panel:**
   - `ng generate @angular/material:expansion --name myExpansion`

23.2.3 Useful Links

- Angular Material Getting Started Guide
- Angular Material Components
- Angular Material Documentation
- Angular Material Examples on StackBlitz

23.2.4 Commonly Used Components

1. **Button:**
    - Button Documentation
    - `<button mat-button>Basic</button>`
    - `<button mat-raised-button>Raised</button>`
2. **Toolbar:**
    - Toolbar Documentation
    - `<mat-toolbar>My Toolbar</mat-toolbar>`
3. **Icon:**
    - Icon Documentation
    - `<mat-icon>home</mat-icon>`
4. **Card:**
    - Card Documentation
    - `<mat-card>Simple Card</mat-card>`
5. **Input:**
    - Input Documentation
    - `<mat-form-field>`
    - `<mat-label>Input</mat-label>`
    - `<input matInput>`
    - `</mat-form-field>`
6. **Table:**
    - Table Documentation
    - `<table mat-table [dataSource]="dataSource">`
    - `<ng-container matColumnDef="position">`
    - `<th mat-header-cell *matHeaderCellDef> No. </th>`
    - `<td mat-cell *matCellDef="let element"> element.position </td>`
    - `</ng-container>`
    - `<tr mat-header-row *matHeaderRowDef="displayedColumns"></tr>`
    - `<tr mat-row *matRowDef="let row; columns: displayedColumns;"></tr>`
    - `</table>`

23.3 Examples

- `ng generate app-shell`: Generates an application shell for running a server-side version of an app.
  - **Example:** `ng generate app-shell`
  - **Explanation:** This command creates an application shell that allows for server-side rendering (SSR) of the Angular application, improving performance and SEO.
- `ng generate application [name]`: Generates a new basic application definition in the "projects" subfolder of the workspace.
  - **Example:** `ng generate application my-app`
  - **Explanation:** This command generates a new Angular application named `my-app` within the `projects` directory of the workspace. It's useful for creating multiple applications within a single workspace.
- `ng generate class [name]`: Creates a new, generic class definition in the given project.
  - **Example:** `ng generate class my-class`
  - **Explanation:** This command generates a new TypeScript class named `my-class`. Classes are used to define entities with properties and methods.
- `ng generate component [name]`: Creates a new, generic component definition in the given project.
  - **Example:** `ng generate component my-component`
  - **Explanation:** This command generates a new Angular component named `my-component`. Components are the building blocks of Angular applications, consisting of HTML templates, CSS styles, and TypeScript logic.
- `ng generate config [type]`: Generates a configuration file in the given project.
  - **Example:** `ng generate config karma`
  - **Explanation:** This command generates a configuration file of the specified type (e.g., Karma configuration for unit testing) in the project.
- `ng generate directive [name]`: Creates a new, generic directive definition in the given project.
  - **Example:** `ng generate directive my-directive`
  - **Explanation:** This command generates a new Angular directive named `my-directive`. Directives are used to extend HTML by adding new behavior to elements.
- `ng generate enum [name]`: Generates a new, generic enum definition in the given project.

  - **Example:** `ng generate enum my-enum`
  - **Explanation:** This command generates a new TypeScript enum named `my-enum`. Enums are used to define a set of named constants.

- `ng generate environments`: Generates and configures environment files for a project.
  - **Example:** `ng generate environments`
  - **Explanation:** This command generates environment configuration files, such as `environment.ts` and `environment.prod.ts`, to manage different settings for development and production environments.

- `ng generate guard [name]`: Generates a new, generic route guard definition in the given project.
  - **Example:** `ng generate guard my-guard`
  - **Explanation:** This command generates a new route guard named `my-guard`. Guards are used to control access to routes in Angular applications.

- `ng generate interceptor [name]`: Creates a new, generic interceptor definition in the given project.
  - **Example:** `ng generate interceptor my-interceptor`
  - **Explanation:** This command generates a new HTTP interceptor named `my-interceptor`. Interceptors are used to inspect and transform HTTP requests and responses.

- `ng generate interface [name] [type]`: Creates a new, generic interface definition in the given project.
  - **Example:** `ng generate interface my-interface`
  - **Explanation:** This command generates a new TypeScript interface named `my-interface`. Interfaces define the shape of objects and ensure type safety.

- `ng generate library [name]`: Creates a new, generic library project in the current workspace.
  - **Example:** `ng generate library my-library`
  - **Explanation:** This command generates a new library project named `my-library` within the current Angular workspace. Libraries are reusable modules that can be shared across multiple projects.

- `ng generate module [name]`: Creates a new, generic NgModule definition in the given project.
  - **Example:** `ng generate module my-module`
  - **Explanation:** This command generates a new NgModule named `my-module`. Modules are used to organize an application into cohesive blocks of functionality.

- `ng generate pipe [name]`: Creates a new, generic pipe definition in the given project.
  - **Example:** `ng generate pipe my-pipe`
  - **Explanation:** This command generates a new Angular pipe named `my-pipe`. Pipes are used to transform data in templates.
- `ng generate resolver [name]`: Generates a new, generic resolver definition in the given project.
  - **Example:** `ng generate resolver my-resolver`
  - **Explanation:** This command generates a new resolver named `my-resolver`. Resolvers are used to pre-fetch data before navigating to a route.
- `ng generate service [name]`: Creates a new, generic service definition in the given project.
  - **Example:** `ng generate service my-service`
  - **Explanation:** This command generates a new service named `my-service`. Services are used to encapsulate business logic and data access.
- `ng generate service-worker`: Pass this schematic to the "run" command to create a service worker.
  - **Example:** `ng add @angular/pwa`
  - **Explanation:** This command adds PWA (Progressive Web App) support to your Angular project, including a service worker for offline capabilities.
- `ng generate web-worker [name]`: Creates a new, generic web worker definition in the given project.
  - **Example:** `ng generate web-worker my-worker`
  - **Explanation:** This command generates a new web worker named `my-worker`. Web workers are used to run background tasks without blocking the main thread.

23.3.1 App Shell

Explanation: An Angular app shell is a minimal version of your application that is rendered on the server. It helps improve the perceived load time by displaying a static shell of the app quickly.

```
import { NgModule } from '@angular/core';
import { ServerModule } from '@angular/platform-server';
import { AppComponent } from './app.component';
import { AppModule } from './app.module';

@NgModule({
  imports: [
  AppModule,
  ServerModule,
```



```
    ],
    bootstrap: [AppComponent],
})
export class AppServerModule {}
```

Listing 23.1: App Shell Example

23.3.2 Application

Explanation: An Angular application is a standalone project created within a workspace, which includes all necessary configurations and components.

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

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

Listing 23.2: Application Example

23.3.3 Class

Explanation: A class is a blueprint for creating objects, providing initial values for state and implementations of behavior.

```
export class Person {
  constructor(public name: string, public age: number) {}
}
```

Listing 23.3: Class Example

23.3.4 Component

Explanation: A component controls a patch of screen called a view and defines the logic and data binding for that view.

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

@Component({
  selector: 'app-hello-world',
  template: '<h1>Hello, World!</h1>',
})
export class HelloWorldComponent { }
```


Listing 23.4: Component Example

23.3.5 Config

Explanation: Configuration files are used to define settings and parameters for different environments and tools.

```
export const environment = {
  production: false,
  apiUrl: 'http://localhost:3000/api'
};
```

Listing 23.5: Config Example

23.3.6 Directive

Explanation: Directives add behavior to an existing DOM element or component instance.

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef, renderer: Renderer2) {
    renderer.setStyle(el.nativeElement, 'backgroundColor', 'yellow');
  }
}
```

Listing 23.6: Directive Example

23.3.7 Enum

Explanation: Enums are a way to define a set of named constants.

```
export enum Color {
  Red = 'RED',
  Green = 'GREEN',
  Blue = 'BLUE'
}
```

Listing 23.7: Enum Example

23.3.8 Environments

Explanation: Environment files allow you to define different configurations for different environments (e.g., development, production).

```
// environment.ts
export const environment = {
  production: false,
  apiUrl: 'http://localhost:3000/api'
};

// environment.prod.ts
export const environment = {
  production: true,
  apiUrl: 'https://api.example.com'
};
```

Listing 23.8: Environments Example

23.3.9 Guard

Explanation: Guards determine if a route can be activated or deactivated.

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

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  canActivate(
  next: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): boolean {
    return true; // Add your authentication logic here
  }
}
```

Listing 23.9: Guard Example

23.3.10 Interceptor

Explanation: Interceptors inspect and transform HTTP requests and responses.

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler } from '@angular/common/http'
  ;

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const clonedRequest = req.clone({
      headers: req.headers.set('Authorization', 'Bearer YOUR_TOKEN')
    });
    return next.handle(clonedRequest);
  }
}
```

Listing 23.10: Interceptor Example

23.3.11 Interface

Explanation: Interfaces define the shape of an object, enforcing type safety.

```
export interface Person {
  name: string;
  age: number;
}
```

Listing 23.11: Interface Example

23.3.12 Library

Explanation: Libraries are reusable modules that can be shared across multiple projects.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MyComponent } from './my-component/my-component.component';

@NgModule({
  declarations: [MyComponent],
  imports: [CommonModule],
  exports: [MyComponent]
})
export class MyLibraryModule { }
```

Listing 23.12: Library Example

23.3.13 Module

Explanation: Modules are used to group related components, directives, pipes, and services.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MyComponent } from './my-component/my-component.component';

@NgModule({
  declarations: [MyComponent],
  imports: [CommonModule],
  exports: [MyComponent]
})
export class MyModule { }
```

Listing 23.13: Module Example

23.3.14 Pipe

Explanation: Pipes are used to transform data in templates.

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

@Pipe({
  name: 'capitalize'
})
```



```
export class CapitalizePipe implements PipeTransform {
  transform(value: string): string {
    return value.charAt(0).toUpperCase() + value.slice(1);
  }
}
```

Listing 23.14: Pipe Example

23.3.15 Resolver

Explanation: Resolvers pre-fetch data before navigating to a route.

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

@Injectable({
  providedIn: 'root'
})
export class DataResolver implements Resolve<Observable<string>> {
  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable
  <string> {
    return of('Resolved Data');
  }
}
```

Listing 23.15: Resolver Example

23.3.16 Service

Explanation: Services are used to encapsulate business logic and data access.

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

@Injectable({
  providedIn: 'root'
})
export class DataService {
  getData(): string {
    return 'Some data';
  }
}
```

Listing 23.16: Service Example

23.3.17 Service Worker

Explanation: Service workers enable Progressive Web App (PWA) features such as offline support.

```
// No TypeScript code needed. Add the service worker using the Angular CLI
// ng add @angular/pwa
```

Listing 23.17: Service Worker Example

23.3.18 Web Worker

Explanation: Web workers allow you to run background tasks without blocking the main thread.

```
// app.worker.ts
addEventListener('message', ({ data }) => {
  const response = `Worker response to \textdollar{data}`;
  postMessage(response);
});
```

Listing 23.18: Web Worker Example

Chapter 24

Client Side Security

JWT (JSON Web Token) and JWKS (JSON Web Key Set) are powerful tools for secure and scalable authentication in modern web applications. JWTs are used for representing claims securely between parties. They consist of three parts: a header, a payload, and a signature. The header typically consists of the type of token (JWT) and the signing algorithm being used. The payload contains the claims, which are statements about an entity (typically, the user) and additional data. The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

JWKS, on the other hand, provides a standardized way to publish public keys for verifying JWT signatures. A JWKS endpoint is a URL where a set of keys is published. Each key can be used to validate the signature of a JWT. This mechanism allows servers to rotate keys and clients to automatically discover new keys, ensuring secure and seamless operations without downtime or the need for manual intervention.

OAuth (Open Authorization) and OIDC (OpenID Connect) further enhance the security and scalability of web authentication. OAuth 2.0 is an authorization framework that enables third-party applications to obtain limited access to a service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. It allows users to authorize third-party applications to access their information without sharing their credentials. Common use cases include single sign-on (SSO) and delegated access to resources on behalf of the user.

OIDC (OpenID Connect) is an identity layer on top of OAuth 2.0. It allows clients to verify the identity of the end-user based on the authentication performed by an authorization server and to obtain basic profile information about the end-user. OIDC introduces the concept of an ID Token, which is a JWT that contains user authentication information, such as the user's identifier, the time the token was issued, and any other claims necessary for the client to understand who the user is. It simplifies the process of federated authentication by providing a standardized identity protocol, making it easier to implement and ensuring interoperability. In below sections we discuss each in more details.

24.1 JSON Web Token (JWT)

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure, enabling the claims to be digitally signed or integrity-protected with a Message Authentication Code (MAC) and/or encrypted. A JWT is composed of three parts and each part is encoded in Base64Url and separated by dots:

SomeHeader.SomePayload.SomeSignature

24.1.1 HS256 Overview

Before we start, let's review HMAC, or **Hash-based Message Authentication Code**, which is a method used to verify the integrity and authenticity of a message using a cryptographic hash function along with a secret key. The purpose of HMAC is to provide a means of checking whether a message has been tampered with, as well as ensuring the authenticity of the message's origin.

1. **Hash Function**: HMAC can be used with any cryptographic hash function (e.g., MD5, SHA-1, SHA-256), though SHA-256 is often preferred for its security.
2. **Secret Key**: HMAC is symmetric, meaning the same secret key is used for both generating and verifying the code. Only parties that have the key can generate or verify an HMAC, which provides mutual trust.
3. **Construction**: HMAC combines the message with the secret key and applies the hash function to produce a code. This output code is unique to the message and key, and if either is changed, the HMAC result changes.
4. **Verification**: The recipient, with access to the shared secret key, can recompute the HMAC on the received message. If the computed HMAC matches the one sent with the message, the message is validated as genuine.

HS256, which is HMAC with SHA-256 is a specific implementation of HMAC that uses SHA-256 as the hashing function. It can be used for:

1. **Signing a Message**:
    - The sender creates a hash of the message, combined with the secret key, using SHA-256.
    - This produces a unique code (or signature) based on the message content and the secret key.
2. **Validating a Message**:
    - The receiver uses the same SHA-256 hashing function and the shared secret key on the received message.

- If the generated HMAC matches the provided signature, it verifies the message's authenticity and integrity.

HS256 is commonly used in **JSON Web Tokens (JWTs)** for secure communication between a client and server. In a JWT, the payload is signed with HS256, allowing the server to verify that the token was created by a trusted source and has not been altered. It provides a symmetric key mechanism where both parties share a secret key, which is then used to generate and verify signatures, ensuring message integrity and authenticity.

24.1.2 RS256 Overview

RS256 is a cryptographic signing algorithm that combines RSA (Rivest-Shamir-Adleman) encryption with SHA-256 hashing to create a secure digital signature. It's an asymmetric signing algorithm commonly used in JSON Web Tokens and other protocols where secure verification of message authenticity is needed. RS256 is particularly favored for scenarios where the verifying party should not have access to the signing key, as it uses separate keys for signing and verification.

Unlike HMAC, RS256 is an asymmetric algorithm, meaning it uses two different keys:

- **Private Key**: Used by the sender to generate a digital signature.
- **Public Key**: Used by the receiver to verify the digital signature.

The private key must remain secure, as it is used to sign the message, while the public key is distributed to others for verification. This separation allows third parties to verify the authenticity of a message without having access to the private key, enhancing security.

As we know, RSA is a widely used public-key cryptosystem, chosen in RS256 because of its robustness in securing data. It provides the foundation for the asymmetric nature of the algorithm.

RS256 uses the SHA-256 hashing function to compute a fixed-length digest of the message content. The digest is then encrypted with the RSA private key to create a signature. SHA-256 is chosen here for its balance of security and efficiency. It can be used for:

Signing a Message:

1. The sender hashes the message with SHA-256, producing a fixed-length hash.
2. The sender then uses the RSA private key to encrypt this hash. The encrypted hash becomes the digital signature.
3. The message and the signature are then sent together to the receiver.

Validating a Message:

1. The receiver uses the sender's public key to decrypt the signature, retrieving the original SHA-256 hash created by the sender.
2. The receiver then hashes the received message independently using SHA-256.

3. If the newly computed hash matches the decrypted hash, the message is authentic and has not been altered in transit.

RS256 can offer strong security for applications requiring verifiable, tamper-resistant messages, with an extra layer of trust facilitated by the use of separate signing and verification keys for scenarios like secure authentication. In this context:

- A server signs a JWT containing user information with its private key.
- Clients can then use the server's public key to verify the token's authenticity, ensuring that it was indeed generated by the server and hasn't been tampered with.

24.1.3 JWT Header

The header typically consists of two parts: the type of token and the signing algorithm (e.g., HMAC SHA256).

```
{
  "alg": "HS256",
  "typ": "JWT"
}

```

24.1.4 JWT Payload

The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}

```

24.1.5 JWT Signature

To create the signature part, you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that:

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

```

24.1.6 HS256 vs. RS256

As we discussed, RS256 and HS256 are both cryptographic signing algorithms often used in JSON Web Tokens and some other secure communication protocols. The choice between RS256 and HS256 depends on specific requirements for **security**, **performance**, and **key management complexity**.

- **HS256** is ideal for performance-critical applications with controlled access to a shared key.
- **RS256** is preferred in distributed systems that require strong security and public verification.

Below we want to do a comparison based on their security model, efficiency, use cases.

Security Model

- **HS256 (HMAC with SHA-256)**
  - **Symmetric**: HS256 is symmetric, meaning the same secret key is used for signing and verifying. This requires both the sender and receiver to share the secret key, which increases risk if the key is exposed.
  - **Pros**: Simpler to implement since only one key is required.
  - **Cons**: Key management is challenging in distributed systems, as all parties need to protect the shared key.
- **RS256 (RSA with SHA-256)**
  - **Asymmetric**: RS256 uses a private key for signing and a public key for verification. Only the sender holds the private key, while the public key can be distributed for verification.
  - **Pros**: More secure for distributed systems; only the private key needs protection.
  - **Cons**: Requires managing a key pair, which adds complexity.

Efficiency

- **HS256**
  - **Performance**: Faster, as symmetric operations are quicker than asymmetric.
  - **Efficiency**: Ideal for high-speed processing in low-latency environments.
  - **Resource Usage**: Requires less computational power, making it suitable for embedded systems or devices with limited resources.
- **RS256**
  - **Performance**: Slower due to computationally intensive RSA operations.
  - **Efficiency**: Less efficient for systems needing fast verification of many signatures.
  - **Resource Usage**: Higher resource usage, which may affect real-time systems or hardware-constrained environments.

Key Length and Signature Size

- **HS256**
  - **Key Length**: Typically uses a 256-bit secret key.
  - **Signature Size**: Fixed at 256 bits (32 bytes) with SHA-256, regardless of message length.
- **RS256**
  - **Key Length**: Commonly 2048 or 4096 bits, depending on security needs.
  - **Signature Size**: Depends on RSA key length (e.g., 256 bytes for a 2048-bit key).

Use Cases

- **HS256**
  - **Best Suited For**: High-performance applications like IoT, embedded systems, or real-time processing where key sharing is manageable.
  - **Examples**: Often used in internal API authentication where the secret key can be securely shared.
- **RS256**
  - **Best Suited For**: Distributed systems or applications requiring public verification without exposing the private key.
  - **Examples**: Used in OAuth 2.0 and JWTs in open systems where verification by multiple parties is necessary.

Feature	HS256 (Symmetric)	RS256 (Asymmetric)
Key Management	Simple, single shared secret key	Requires key pair (private public keys)
Security	Secure, but depends on secret key secrecy	More secure for distributed verification
Performance	Faster, low computational overhead	Slower, higher computational overhead
Resource Usage	Lightweight	Heavier resource consumption
Key Length	Shorter (e.g., 256 bits)	Longer (e.g., 2048+ bits)
Signature Size	Fixed and small (32 bytes)	Larger (depends on RSA key size)
Use Cases	Best for internal or high-performance apps	Best for distributed or public verification

24.2 JSON Web Key Set (JWKS)

JSON Web Key Set (JWKS) is a JSON structure that represents a set of JSON Web Keys (JWK). This data structure can be used to represent one or more keys, which can be used for various purposes, including signature verification. A JWKS typically contains a `keys` property, which is an array of JWK objects. Here is an example:

```
{
  "keys": [
  {
    "kty": "RSA",
    "kid": "1234ABCD",
    "use": "sig",
    "n": "0vx7agoebGcQSuuPiLJXZptN4nrh9m5dI5ySTwh8dG...",
    "e": "AQAB"
  }
  ]
}

```

Please note:

- `kty`: Key Type (e.g., "RSA")
- `kid`: Key ID, used to match a specific key (e.g., "1234ABCD")
- `use`: Public Key Use, `sig` (signature) or `enc` (encryption) (e.g., "sig")
- `n`: Modulus (base64url-encoded) for RSA
- `e`: Exponent (base64url-encoded) for RSA

24.3 How JWT and JWKS Work Together

Using JWKS, services can dynamically retrieve and rotate public keys without downtime. This enhances security and flexibility, which is particularly useful in distributed systems and microservices architectures. Tokens such as JWT can be verified as well

24.3.1 Token Issuance and Verification

When a client authenticates and requests a token, the server creates a JWT signed with its private key. Further, when the client makes a request with the JWT, the server can verify the token's signature using the corresponding public key found in the JWKS. Here is a typical workflow:

1. Resource server fetches the JWKS from the authorization server.
2. Finds the public key with the matching `kid`.
3. Verifies the JWT's signature using this key. If valid, processes the request.

24.3.2 Client Authentication

Client authenticates with the server, while, server issues a JWT to the client, signed with the server's private key. Here is a example of client making a request to a resource server with the JWT in the Authorization header.

```
GET /protected/resource HTTP/1.1
Host: resource.server.com
Authorization: Bearer eyJhbGciOiJSUzI1NiIsImt...

```

24.3.3 Key Rotation and Public Key Distribution (PKD)

The JWKS allows for easy rotation and invalidation of keys without impacting clients, as they can dynamically fetch the latest set of keys. This helps with Public Key Distribution (PKD). For instance, the authorization server provides a well-known JWKS endpoint where it publishes its public keys. These keys are used by clients and resource servers to verify the signatures of JWTs. The JWKS allows for smooth key rotation. The authorization server can further update the keys in the JWKS endpoint, and clients can periodically fetch the latest keys without service disruption.

24.4 JWKS with Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is a form of public key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC keys are smaller, which results in faster computation, lower power consumption, and smaller memory requirements. Similar to RSA structure, a JWKS for ECC contains an array of JWK objects, each representing an elliptic curve key. The keys will have parameters specific to ECC, such as `crv` (curve), `x`, and `y` coordinates. Here is an example:

```
{
  "keys": [
  {
    "kty": "EC",
    "kid": "1",
    "use": "sig",
    "crv": "P-256",
    "x": "f83OJ3D2xFMB2TcQvV5G9QtxHWEwYI6NL_dwPmg6rmI",
    "y": "x_FEzRu9kiPSt9I1gTYV8ZOmNbebcfiZJ-2V0y8BEp8",
    "alg": "ES256"
  }
  ]
}

```

Please note:

- `kty`: Key Type (e.g., "EC" for elliptic curve)
- `kid`: Key ID, used to match a specific key (e.g., "1")

- `use`: Public Key Use, "sig" (signature) or "enc" (encryption) (e.g., "sig")
- `crv`: Curve, indicates the name of the curve (e.g., "P-256")
- `x`: X coordinate of the elliptic curve point (base64url-encoded)
- `y`: Y coordinate of the elliptic curve point (base64url-encoded)
- `alg`: Algorithm, specifies the algorithm used with the key (e.g., "ES256" for ECDSA using P-256 and SHA-256)

24.5 Role of JWKS and JWT in OIDC

We will discuss OIDC in more details in the subsequent sections. For now, we can say OpenID Connect (OIDC) is an identity layer on top of the OAuth 2.0 protocol, enabling clients to verify the identity of an end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user.

The primary use of JWT in OIDC is for the ID Token, which is a security token that contains identity claims about the end-user. The ID Token is digitally signed using the server's private key. OIDC also uses JWTs for access tokens, which are used to authorize access to protected resources. Payloads typically contain claims about the user and other metadata, such as:

- `iss` (Issuer): URL of the issuing authorization server.
- `sub` (Subject): Identifier for the user.
- `aud` (Audience): Intended audience for the token.
- `exp` (Expiration time): When the token expires.
- `iat` (Issued at): When the token was issued.
- `nonce`: A random value to mitigate replay attacks.
- `Signature`: Ensures the token's integrity and authenticity.

Please note, for user authentication, the user logs in to an OIDC-compliant identity provider. The identity provider authenticates the user and issues a JWT ID Token to the client. The client receives the ID Token and needs to verify its authenticity. The client retrieves the JWKS from the identity provider's well-known configuration endpoint (e.g., `https://auth.server.com/.well-known/openid-configuration`), which includes the URL of the JWKS endpoint. The client finds the public key corresponding to the `kid` in the JWT header from the JWKS. The client uses the public key to verify the JWT's signature, ensuring the token's integrity and authenticity.

To access protected resources, the client includes the JWT access token in requests to the resource server. The resource server retrieves the JWKS from the identity provider. The resource server verifies the access token's signature using the appropriate public key from the JWKS. If the token is valid, the resource server processes the request.

24.6 SaaS Providers and Their Role on OAuth 2.0 and OIDC

OAuth (Open Authorization) and OIDC (OpenID Connect) further enhance the security and scalability of web authentication. OAuth 2.0 is an authorization framework that enables third-party applications to obtain limited access to an service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the service, or by allowing the third-party application to obtain access on its own behalf. It allows users to authorize third-party applications to access their information without sharing their credentials. Some common use cases include single sign-on (SSO) and delegated access to resources on behalf of the user. It defines several roles:

- **Resource Owner**: The user who authorizes access to their resources.
- **Client**: The application requesting access to the resources.
- **Authorization Server**: The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.
- **Resource Server**: The server hosting the protected resources, which accepts and validates access tokens.

OIDC (OpenID Connect) is an identity layer on top of OAuth 2.0. It allows clients to verify the identity of the end-user based on the authentication performed by an authorization server and to obtain basic profile information about the end-user. OIDC introduces the concept of an ID Token, which is a JWT that contains user authentication information, such as the user's identifier, the time the token was issued, and any other claims necessary for the client to understand who the user is. It simplifies the process of federated authentication by providing a standardized identity protocol, making it easier to implement and ensuring interoperability. Key features include:

- **ID Token**: A JWT that contains user authentication information.
- **UserInfo Endpoint**: An endpoint from which the client can retrieve additional user profile information.
- **Discovery Document**: A JSON document that describes the OIDC provider's configuration, including endpoints and supported features.

Together, JWTs, JWKS, OAuth, and OIDC form a robust framework for secure, scalable, and flexible authentication and authorization in modern web applications. For client security of single-page applications (SPAs), there are many SaaS providers, such as IBM AppID, that can be quickly integrated into your Angular applications. These providers offer robust authentication and authorization services, allowing developers to focus on building their applications rather than worrying about the complexities of security. Behind the scenes, such providers utilize OpenID Connect (OIDC) and, of course, JWT, JWKS, etc., to ensure secure and scalable authentication. These services often come with client SDKs that simplify the process of integrating authentication and authorization into your application. They provide comprehensive documentation and sample code to get you started quickly.

24.6.1 Using IBM AppID SDK

Client SDKs provided by SaaS providers simplify the integration process by abstracting the complexities of OAuth and OIDC. Here's a high-level overview of how to use IBM AppID SDK in an Angular application:

Use package managers like npm to install the SDK. For example, for IBM AppID:

```
npm install --save ibmcloud-appid
```

Next, initialize the SDK with your application credentials and configuration details. This typically includes client ID, discovery URL, and redirect URIs.

```
import { AppID } from 'ibmcloud-appid-js';

const appID = new AppID();
appID.init({
 clientId: 'YOUR_CLIENT_ID',
 discoveryEndpoint: 'YOUR_DISCOVERY_URL',
 redirectUri: 'YOUR_REDIRECT_URI'
});
```

Now you can use the SDK to handle user authentication. This often involves redirecting the user to the authentication server, handling the callback, and storing the tokens. In your application code, your login button can add a call to `signin`. The `signin` will trigger a pop-up window opens where a user is prompted to enter their credentials. After a successful authentication, the screen closes and user is authenticated.

```
appID.signin()
.then(tokens => {
 console.log('Access Token:', tokens.accessToken);
 console.log('ID Token:', tokens.idToken);
})
.catch(error => {
 console.error('Error during sign-in:', error);
});
```

Please note, when SSO for Cloud Directory is enabled, you can automatically obtain new tokens for a user without them having to reauthenticate by using `silentSignin` and check for resulting access token. If there is no token obtained by `silentSignin`, you can fallback to regular`signin` logic :

You may also want to implement route guards to protect your application routes. These guards check for a valid authentication state before allowing access to certain parts of your application.

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
```



```
@Injectable({
 providedIn: 'root'
})
export class AuthGuard implements CanActivate {
 constructor(private router: Router, private appID: AppID) {}

 canActivate(): boolean {
  if (this.appID.isAuthenticated()) {
   return true;
  } else {
   this.router.navigate(['/login']);
   return false;
  }
 }
}
```

For more details, please take a look at:

- IBM Cloud App ID Client SDK for JavaScript
- IBM Cloud App ID Server SDK for Node.js
- IBM Cloud App ID Single Page Application Documentation
- OAuth 2.0 for Browser-Based Apps IETF Draft

In the following sections, we will go deeper into OAuth and OIDC concepts.

24.7 OAuth 2.0 for Browser-Based Apps

The IETF draft of *OAuth 2.0 for Browser-Based Apps* outlines best practices for implementing OAuth 2.0 in browser-based applications. It uses OAuth 2.0 authorization code flow with Proof Key for Code Exchange (PKCE) to enhance security. It handles authorization and refresh tokens, protecting against common vulnerabilities like Cross-Site Request Forgery (CSRF), and how to better secure storage of tokens. It also covers architecture patterns for JavaScript applications with or without a backend. The goal is to provide guidelines to securely manage OAuth flows within browser environments.

In the OAuth 2.0 authorization framework, the typical flow involves the user granting the application permission via the authorization server, which then issues an access token. The application uses this token to access resources from the resource server on behalf of the user. Here are some common terminology that we use:

- **User (Resource Owner - RO)**: The individual who owns the data and authorizes the application to access their resources.
- **Application (Client)**: The application requesting access to the user's resources. It can be a web app, mobile app, or other types of software.

- **Authorization Server - AS**: The server responsible for authenticating the user, obtaining their consent, and issuing access tokens to the client application. It manages the OAuth 2.0 flows.
- **Resource Server - RS**: The server hosting the protected resources. It verifies the access token and serves the requested resources to the client.

24.7.1 Typical Interactions in OAuth Authorization Code Flow with PKCE

Here are some of the typical interactions that happens during OAuth 2.0 Authorization Code Flow with Proof Key for Code Exchange to better protect authorization codes in public clients, such as browser-based or mobile applications:

1. **Client Initiation**: The client generates a unique code verifier and a corresponding code challenge. The code challenge is derived from the code verifier using a hashing algorithm (e.g., SHA-256).
2. **Authorization Request**: The client initiates an authorization request to the authorization server, including the code challenge and the method used to generate it.
3. **Authorization Response**: The authorization server authenticates the user and redirects them back to the client address with an authorization code.
4. **Token Request**: The client sends the authorization code along with the code verifier to the token endpoint of the authorization server.
5. **Token Response**: The authorization server validates the code verifier against the code challenge and, if valid, issues an access token to the client. This means possible intercepted authorization codes cannot be used without the code verifier, thus enhancing security.
6. **Accessing Resources**: After the client receives the access token, it can use this token to make authenticated requests to the resource server on behalf of the user of the application. The access token is typically included in the HTTP headers of the requests. The resource server then validates the access token with the authorization server to ensure its legitimacy. If the token is valid, the resource server processes the request and returns the requested resources to the client. If the access token expires, the client can use a refresh token, if issued, to obtain a new access token without requiring the user to reauthenticate. Access token format is often set to JWT and its lifetime is typically short-lived to enhance security. The refresh token is used to obtain a new access token when the current access token expires and it is typically a simple opaque string, which is longer-lived and do require secure storage.

24.7.2 OpenID Connect (OIDC)

OpenID Connect (OIDC) standard aims to add an identity layer on top of the OAuth 2.0 protocol. While OAuth 2.0 is primarily designed for authorization (granting access to resources), OIDC provides authentication, allowing clients to verify the identity of the end-user based on the authentication performed by an authorization server. OIDC also allows for single sign-on (SSO) capabilities and more secure handling of user identities via additional tokens such as:

- **ID Token**: A JWT that is used to authenticate the user and provide information about their identity. ID Token contains claims about the user information (e.g., sub, name, email).
- **UserInfo Endpoint**: An endpoint to fetch additional user attributes.

OIDC allows clients to verify the identity of the end-user based on the authentication performed by an authorization server, as well as to obtain basic profile information about the end-user. The **OIDC Scope** is a mechanism for defining what access privileges are being requested by the client application. When a client requests authorization from an end-user, it specifies one or more scopes. These scopes (typically simple strings) define the level of access the client application is requesting. Common OIDC scopes include:

- `openid`: Required scope for OIDC, which signifies that the application wants to use OIDC to verify the user's identity.
- `profile`: Requests access to the user's default profile claims, such as name, family name, given name, etc.
- `email`: Requests access to the user's email address.
- `address`: Requests access to the user's address information.
- `phone`: Requests access to the user's phone number.
- `offline_access`: Requests a refresh token to obtain access tokens without user interaction.

The **OIDC Context** refers to the environment and conditions under which authentication and authorization occur. This can include several factors, such as:

- **Authentication Context Class Reference (ACR)**: This specifies the requirements for the authentication method used by the identity provider. For example, it may require multi-factor authentication (MFA) or a specific level of assurance.
- **Claims Request**: The claims that are requested by the client can be specified in a more granular way, beyond just using scopes. This allows the client to ask for specific pieces of information about the user.
- **Authorization Request Parameters**: Additional parameters in the authorization request can provide context, such as `prompt` (to force user interaction), `max_age` (maximum allowable time since the user was last authenticated), and `ui_locales` (preferred user interface languages).

The **OIDC Roles** defines several roles involved in the authentication and authorization process:

- **End-User**: The human user who wants to access a resource and authenticate their identity.
- **Relying Party (RP)**: Also known as the client, this is the application that wants to authenticate the end-user and access resources on their behalf. It relies on the identity provider to verify the user's identity.
- **Identity Provider (IdP)**: Also known as the OpenID Provider (OP), this is the service that authenticates the end-user and issues identity tokens, access tokens, and sometimes refresh tokens to the relying party.
- **Resource Server**: This is the server hosting the protected resources that the relying party wants to access. It validates the access token issued by the identity provider.

To recap we can say that: **Scopes** define what access is being requested by the client application. **Context** encompasses the conditions and parameters surrounding the authentication and authorization process. **Roles** include the end-user, relying party, identity provider, and resource server, each playing a specific part in the OIDC framework.

OIDC Standard Endpoints

In OIDC, there are several standard endpoints exposed by the authorization server:

- **Authorization Endpoint**: Used by the client to obtain authorization from the resource owner via user agent redirection.
  - **URL Format**: `https://example.com/oauth2/authorize`
  - **Includes**: client_id, response_type, redirect_uri, scope, state, code_challenge, code_challenge_met (for PKCE).
- **Token Endpoint**: Used by the client to exchange an authorization code for an access token, refresh token, and optionally an ID token.
  - **URL Format**: `https://example.com/oauth2/token`
  - **Includes**: client_id, client_secret, grant_type, code, redirect_uri, code_verifier (for PKCE).
- **UserInfo Endpoint (OIDC)**: Used by the client to retrieve user profile information.
  - **URL Format**: `https://example.com/oauth2/userinfo`
  - **Includes**: access_token in the Authorization header.
- **Token Revocation Endpoint**: Used by the client to notify the authorization server that a previously obtained token is no longer needed.
  - **URL Format**: `https://example.com/oauth2/revoke`

  - **Includes**: token, token_type_hint.

- **Token Introspection Endpoint**: Used by the client or resource server to check the active state of an access token or refresh token.
  - **URL Format**: `https://example.com/oauth2/introspect`
  - **Includes**: token, token_type_hint.

- **Discovery Endpoint (OIDC)**: Provides metadata about the authorization server to simplify the configuration of clients.
  - **URL Format**: `https://example.com/.well-known/openid-configuration`
  - **Includes**: JSON document with URLs for all the other endpoints, supported scopes, response types, grant types, etc.

24.8 OIDC + OAuth 2.0 Sequence Diagram

Please note:

- **User Action**: A user attempts to log in to the single-page application.
- **SDK Generates Code Verifier**: The App ID SDK creates a code verifier for the authorization request, a plain text version of the code challenge. The client sends the code challenge and the challenge method used to encode the challenge with the authorization request.
- **Authorization Request**: The authentication flow is started by App ID in a new window.
- **User Authentication**: The user chooses an identity provider to authenticate with and completes the sign-in process.
- **Grant Code Received**: The App ID SDK on the application receives the grant code.
- **Token Request**: The SDK makes an XHR request to the App ID token endpoint along with the grant code and the code verifier to obtain access and identity tokens.

24.8.1 Why Authorization Code + PKCE?

Please keep in mind that with SPAs, most of the security comes from the API's server, so you should also secure your API server. From a security perspective, single-page applications alone cannot store secrets securely. The Authorization Code + PKCE flow tries to add a variation of the basic OAuth 2.0 Authorization Code flow that uses a one-time code verifier and challenge instead of a secret to address this concern.

- **Preventing Attacks**: The verifier and challenge ensure the same entity calls the authorization and token endpoints, preventing attackers from requesting tokens without knowing the code verifier.

- **Implicit Flow Issues**: The Implicit flow is no longer recommended due to security flaws, such as token interception through URL fragments and susceptibility to redirect URI attacks, where attackers could access user tokens.

24.8.2 Example of OIDC Authorization Code with PKCE Flow

Here is an example that shows a one-time code verifier and challenge in the OIDC Authorization Code with PKCE flow:

Step 1: Client Creates a Code Verifier

The client generates a random string as the code verifier.

```
code_verifier = "dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk"
```

Step 2: Client Creates a Code Challenge

The client creates a code challenge from the code verifier using SHA-256.

```
code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))
code_challenge = "E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM"
```

Step 3: Client Initiates Authorization Request

The client sends the user to the authorization server with the following parameters:

```
GET /authorize?response_type=code
&client_id=client123
&redirect_uri=https://client.example.com/cb
&scope=openid
&code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
&code_challenge_method=S256
```

Step 4: User Authenticates

The user authenticates and the authorization server redirects back to the client with an authorization code.

```
https://client.example.com/cb?code=AUTH_CODE
```

Step 5: Client Requests Access Token

The client exchanges the authorization code for an access token by sending the code verifier along with the code to the token endpoint.


```
POST /token
Host: authorization-server.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=AUTH_CODE
&redirect_uri=https://client.example.com/cb
&client_id=client123
&code_verifier=dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

Step 6: Authorization Server Verifies and Responds

The authorization server verifies the code verifier against the code challenge and, if valid, issues an access token.

```
{
 "access_token": "ACCESS_TOKEN",
 "token_type": "Bearer",
 "expires_in": 3600,
 "id_token": "ID_TOKEN"
}
```

24.8.3 Custom Identity Providers in IBM AppID

You can use your own custom identity provider in your app! Users can sign in using their existing credentials and your application can exchange the authenticated user information for App ID tokens. Your identity provider can still be integrated with App ID even if it is proprietary and non-OIDC or SAML compliant. For more information, see the docs. App ID uses a public key to validate your JSON Web Token (JWT). If you don't already have one, you can use OpenSSL to create a public and private RSA key pair. Your application uses the private key to sign a JWT assertion that contains authenticated user information. App ID then validates the authentication with your pre-configured public key.

24.9 Sample of .well-known/openid-configuration

```
{
 "clientId": "d4314a22-a010-44a8-a835-355f4ea894b9",
 "tenantId": "f84ea474-f78c-4708-9f9d-d1f0bb83b521",
 "name": "angular422",
"oAuthServerUrl": "https://us-south.appid.cloud.ibm.com/oauth/v4/f84ea474-f78c-4708-9f9d-d
 "profilesUrl": "https://us-south.appid.cloud.ibm.com",
"discoveryEndpoint": "https://us-south.appid.cloud.ibm.com/oauth/v4/f84ea474-f78c-4708-9f9
 "type": "singlepageapp",
 "scopes": [
 "admin",
```



```
  "staff"
  ]
}
```

24.9.1 Server Side Node.js OpenID Connect Example with Passport.js

Below sample code provides a basic server-side application using OpenID Connect for authentication.

```
// Import dependencies:
// Run: npm install passport passport-openidconnect express express-session
  openid-client
const express = require("express");
const session = require("express-session");
const passport = require("passport");
const { Issuer, Strategy } = require("openid-client");

// Replace these values with your OpenID Provider details
const clientID = "YOUR_CLIENT_ID";
const clientSecret = "YOUR_CLIENT_SECRET";
const issuerURL = "https://YOUR_ISSUER_URL";  // OpenID Provider URL
const redirectURI = "http://localhost:3000/auth/callback";  // Callback URL

const app = express();

// Express session configuration
app.use(session({
  secret: "YOUR_SESSION_SECRET",
  resave: false,
  saveUninitialized: true,
}));

// Passport initialization
app.use(passport.initialize());
app.use(passport.session());

// Serialize and deserialize user
passport.serializeUser((user, done) => done(null, user));
passport.deserializeUser((user, done) => done(null, user));

// Configure OpenID Connect Strategy
(async () => {
  // Discover the OpenID configuration
  const issuer = await Issuer.discover(issuerURL);

  // Create a client using the discovered issuer information
  const client = new issuer.Client({
    client_id: clientID,
    client_secret: clientSecret,
    redirect_uris: [redirectURI],
    response_types: ["code"],
  });

  // Configure the Passport strategy
  passport.use("oidc", new Strategy({ client }, (tokenSet, userinfo, done) => {
    return done(null, { ...userinfo, tokenSet });
  }));
})();
```



```
49
50  // Authentication routes
51  app.get("/auth", passport.authenticate("oidc"));
52
53  app.get("/auth/callback",
54  passport.authenticate("oidc", { failureRedirect: "/" }),
55  (req, res) => {
56    res.redirect("/protected");
57  }
58  );
59
60  // Protected route
61  app.get("/protected", ensureAuthenticated, (req, res) => {
62    res.send(`Hello, ${req.user.name}. You have accessed a protected route!`);
63  });
64
65  // Logout route
66  app.get("/logout", (req, res) => {
67    req.logout((err) => {
68      if (err) { return next(err); }
69      res.redirect("/");
70    });
71  });
72
73  // Middleware to ensure authentication
74  function ensureAuthenticated(req, res, next) {
75    if (req.isAuthenticated()) {
76      return next();
77    }
78    res.redirect("/auth");
79  }
80
81  // Start server
82  app.listen(3000, () => {
83    console.log("Server is running on http://localhost:3000");
84  });
```

Listing 24.1: Node.js OpenID Connect Example with Passport.js

Discovery and Client Setup

- `Issuer.discover` automatically fetches configuration data from the OpenID Provider (OP).
- The client is configured with credentials like `client_id` and `client_secret`, as well as the redirect URI.

Passport.js Strategy

- A new `Strategy` (e.g., how to deal with specifics of OIDC) is created using the `openid-client` client instance. Passport uses this strategy to handle the OpenID Connect flow.
- The callback in the strategy processes the `tokenSet` and `userinfo`, storing them in the session.

Routes Setup

- `/auth`: Initiates the OpenID Connect flow.
- `/auth/callback`: Handles the callback from the OpenID Provider and redirects the user to the `/protected` route upon success.
- `/protected`: An example of a protected route accessible only to authenticated users.
- `/logout`: Logs out the user by clearing the session.

Middleware

- `ensureAuthenticated` is a helper middleware to check if the user is authenticated before accessing protected routes.

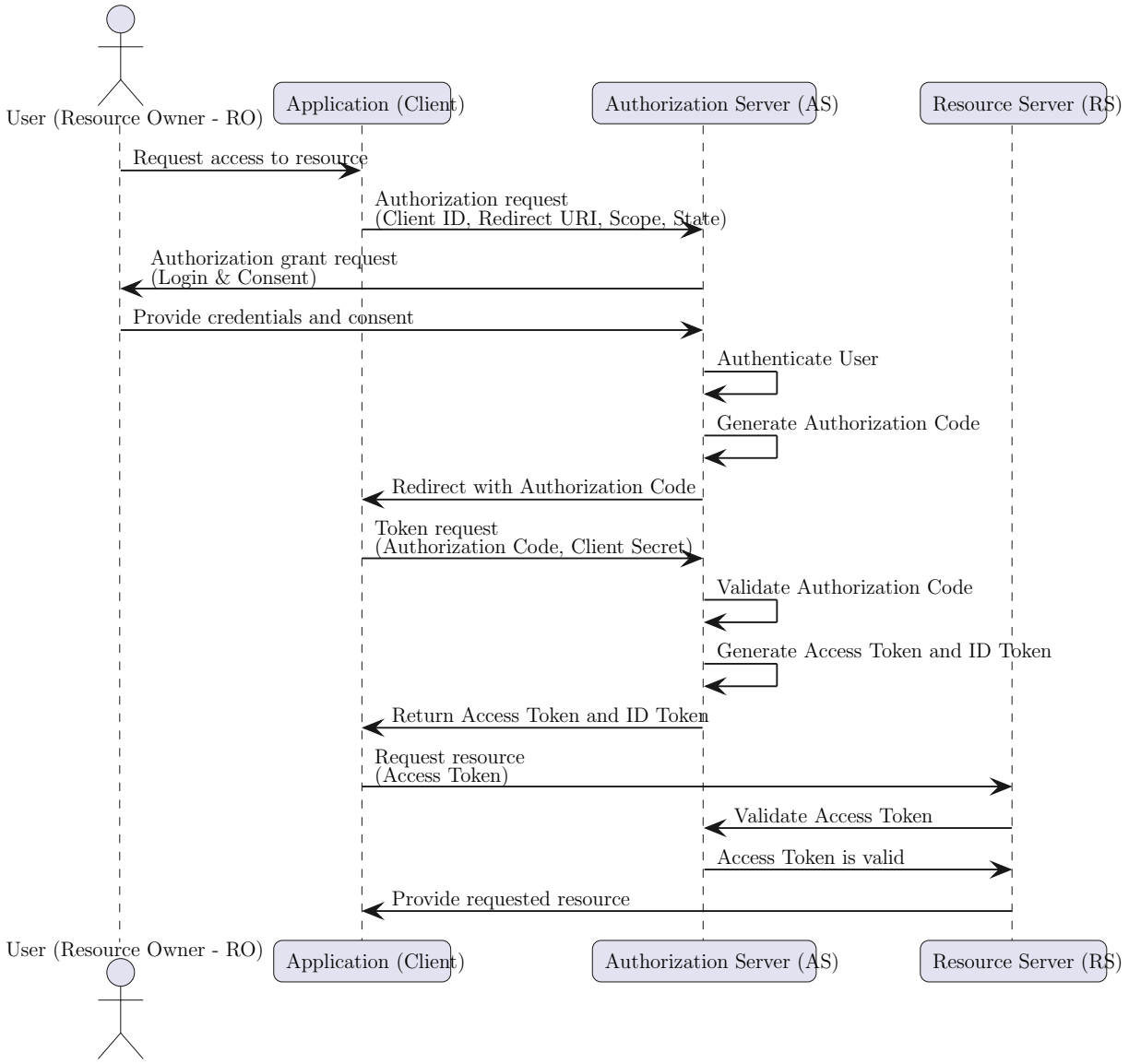

Figure 24.1: Typical Interactions in OAuth Authorization Code Flow

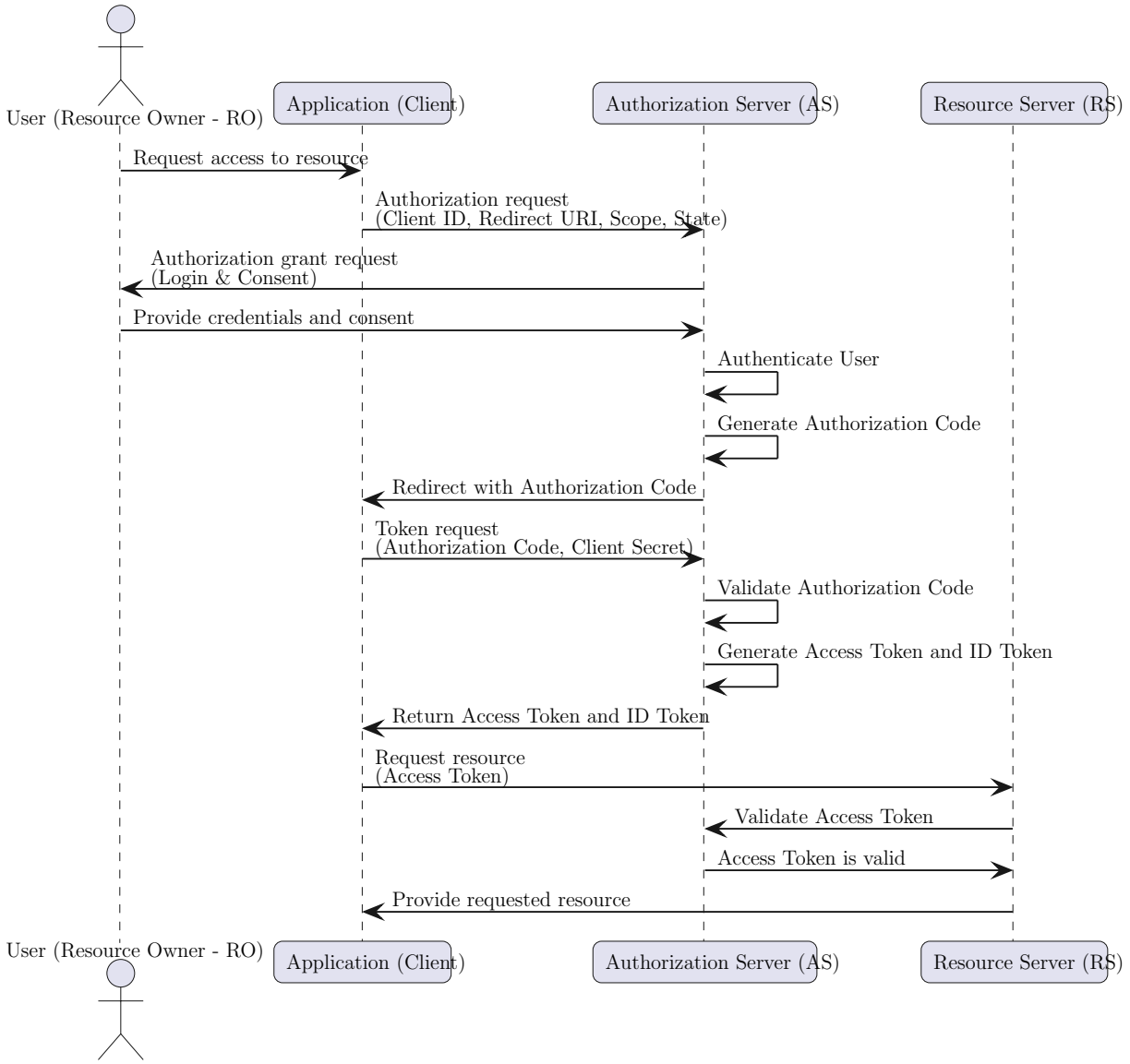

Figure 24.2: Sample SPA with PKCE

Chapter 25

Next Steps

Explore the following resources to deepen your understanding of Angular:

- Angular Tutorials
- Angular Playground
- Dependency Injection Guide
- HTTP Client Guide
- Angular API Reference
- Angular SEO Guide.