Front-End Java Script Frameworks

DECAP917

Edited by:
Sartaj Singh

Front-End Java Script Frameworks

Edited By
Sartaj Singh

Title: FRONT-END_JAVASCRIPT_FRAMEWORKS

Author's Name: Harjinder Kaur

Published By : Lovely Professional University

Publisher Address: Lovely Professional University, Jalandhar Delhi GT road, Phagwara - 144411

Printer Detail: Lovely Professional University

Edition Detail: (I)

ISBN: 978-81-19929-64-1

Copyrights@ Lovely Professional University

Content

Unit 1:	**Introduction** *Harjinder Kaur, Lovely Professional University*	**1**
Unit 2:	**Angular Projects/Programs** *Harjinder Kaur, Lovely Professional University*	**14**
Unit 3:	**Type Script Fundamentals** *Harjinder Kaur, Lovely Professional University*	**25**
Unit 4:	**Classes and Objects in Angular** *Harjinder Kaur, Lovely Professional University*	**37**
Unit 5:	**Angular Fundamentals** *Harjinder Kaur, Lovely Professional University*	**54**
Unit 6:	**Displaying Data and Handling Events** *Harjinder Kaur, Lovely Professional University*	**67**
Unit 7:	**Directives** *Harjinder Kaur, Lovely Professional University*	**81**
Unit 8:	**Building Re-usable Components** *Harjinder Kaur, Lovely Professional University*	**92**
Unit 9:	**Template-driven Forms** *Harjinder Kaur, Lovely Professional University*	**101**
Unit 10:	**Form Handling in Angular** *Harjinder Kaur, Lovely Professional University*	**121**
Unit 11:	**Consuming HTTP Services** *Harjinder Kaur, Lovely Professional University*	**135**
Unit 12:	**Routing and Navigation** *Harjinder Kaur, Lovely Professional University*	**154**
Unit 13:	**Authentication and Authorization** *Harjinder Kaur, Lovely Professional University*	**163**
Unit 14:	**Deployment** *Harjinder Kaur, Lovely Professional University*	**175**

Unit 01: Introduction

CONTENTS

Objective

Introduction

1.1 Installing and downloading AngularJS

1.2 AngularJS

1.3 Expressions

1.4 Architecture

1.5 Development Environment

1.6 AngularJS Advantages

1.7 AngularJS Disadvantages

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objective

After this unit you will be able to :

- Know the basics of AngularJS
- Understand the Architecture of AnjularJS
- Identify the various components of AngularJS architecture.
- Declare and use expressions

Introduction

Let's first analyze what AngularJS is in detail before going into further detail about it. Why would we want to employ a "framework" and what does that term mean? Would it be wiser not to use one? Even better, why not create our own from scratch?

A framework is described as "a fundamental supporting structure" in the dictionary. Although AngularJS is much more than that, that does a great job of summarizing it. AngularJS has a sizable and supportive community, an ecosystem where new tools and utilities can be found, clever solutions to everyday issues, and, for many, a novel and inspiring way of approaching application structure and design.

If we wanted to make things more difficult for ourselves, we could create our framework. Realistically, this is simply not possible for the majority of us. It practically goes without saying that you need some kind of framework to help you and that this framework most likely shouldn't be your undocumented (or poorly understood) notions and beliefs on how things should be done. A good framework, like AngularJS, has previously been thoroughly examined and studied by others. Remember that someday someone else might inherit your code, join your team, or require the structure and assistance that a framework offers in some other way.

Framework usage is popular; many programmers from many backgrounds rely on it. When creating database-related apps, business application developers employ frameworks like the Microsoft Entity Framework to lessen their suffering and accelerate development.

Example: The LibGDX framework is used by Java programmers to build games.

1.1 Installing and downloading AngularJS

AngularJS installation and download are quick and simple. Go to http://angularjs.org to download AngularJS, then do the following:

1. Make a BeginningAngularJS folder on your computer. Make a subfolder called js inside this folder to store your JavaScript files.

2. Select the Download option from the AngularJS home page. A dialogue box similar to the one in Figure 1 will appear.

Note: The jqLite plugin for AngularJS is a condensed version of jQuery. However, in general, it is preferable to carry out tasks in an "Angular Way."

Figure 1: AngularJS Download

3. Select 1.2.X (legacy) for the branch option and Minified for the build option if you want the 1.2.x-minified version.

4. To begin the download, click the Download button.

5. Insert the downloaded file, angular.min.js, into the js folder you made previously after the download is finished (assuming you did not save it there directly).

6. Now, AngularJS is downloaded and installed in your system.

1.2 AngularJS

An incredibly powerful JavaScript framework is AngularJS. Projects requiring Single Page Applications (SPAs) use it. It adds new characteristics to the HTML DOM and improves responsiveness to user input. Numerous developers worldwide utilize AngularJS, which is open-source and free. The Apache license, version 2.0, governs its use.

Declarative programming should be used to build user interfaces and link software components, but imperative programming is better suited for defining an application's business logic, according to the concept of AngularJS. The framework extended and modified conventional HTML to offer dynamic content with two-way data-binding that made it possible for models and views to be

automatically synced. In order to increase testability and efficiency, AngularJS deemphasized explicit Document Object Model (DOM) manipulation.

Design objectives for AngularJS included:

1. The DOM manipulation should be separated from application logic. The organization of the code has a significant impact on how tough this is.

2. to separate the client side from the server side of an application. As a result, development activity can move on concurrently and both sides can be reused.

3. To provide an application's development process some structure, from UI design to business logic authoring to testing.

Note: AngularJS isn't quite like other frameworks, and it may require you to think a little differently than you are used to

Why AngularJS?

It is an open-source web application framework. It was initially created in 2009 by Adam Abrons and Misko Hevery. Google now takes care of it. It currently runs on version 1.8.3. The following are some of the features because which AngularJS is gaining popularity:

- A powerful framework for building Rich Internet Applications is AngularJS (RIA).
- Developers have the opportunity to create client-side JavaScript apps with AngularJS in a neat Model View Controller (MVC) manner.
- Cross-browser compatibility is a feature of AngularJS-written applications. Each browser-specific JavaScript code is automatically handled by AngularJS.
- Numerous developers worldwide utilize AngularJS, which is open-source and free. The Apache license, version 2.0, governs its use.

AngularJS is a framework for creating complex, fast-performing, and easy web applications.

Code of Angular JS

AngularJS applications are a mix of HTML and JavaScript. The first thing you need is an HTML page. Second, you need to include the AngularJS JavaScript file in the HTML page so we can use AngularJS:

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.5/angular.min.js"></script>
</head>
<body>
.
.
</body>
</html>
```

Features of AngularJS

AngularJS has many features which are depicted in Figure 2. The following are some of the general and core features:

General Features

- AngularJS is an efficient framework that can create Rich Internet Applications (RIA).
- AngularJS provides developers an option to write client-side applications using JavaScript in a clean Model View Controller (MVC) way.
- Applications written in AngularJS are cross-browser compatibility (the capacity for a website to work well regardless of which browser renders it.). AngularJS automatically handles JavaScript code suitable for each browser.
- AngularJS is open-source, completely free, and used by thousands of developers around the world. It is licensed under the Apache license version 2.0.
- Overall, AngularJS is a framework to build large-scale, high-performance, and easy-to-maintain web applications.

Core Features

Data-binding: The automatic synchronization of data between model and display components is known as data-binding.

Scope: The objects in the scope category all refer to the model. Between controller and view, they serve as an adhesive.

Controller: These are JavaScript operations that are tied to a specific scope.

Services: AngularJS has several built-in services, including $http, which is used to create XMLHttpRequests. These are singleton objects that the app only creates once.

Figure 2: AngularJS Features

Filters: The filters are used to pick a portion of an array's contents and then return a new array

Directives: The directives are labels for DOM elements, including elements, attributes, CSS, and more. These can be utilized to develop unique HTML tags that act as new, creative widgets. The built-in directives in AngularJS include ngBind, ngModel, etc.

Templates: These are the rendered views that contain data from the model and controller. These can be several views on the same page using partials or a single file (like index.html).

Routing: Routing is the idea of changing viewpoints.

Model View Whatever − MVW is a design pattern for dividing an application into different parts called Model, View, and Controller, each with distinct responsibilities. AngularJS does not implement MVC in the traditional sense, but rather something closer to MVVM (Model-View-ViewModel).

Deep Linking − Deep linking allows encoding the state of an application in the URL so that it can be bookmarked. The application can then be restored from the URL to the same state. **Dependency Injection** − AngularJS has a built-in dependency injection subsystem that helps the developer to create, understand, and test the applications easily.

1.3 Expressions

HTML is linked to application data through expressions. Double curly braces are used to denote expressions, as in {{expression}}. Expressions function in a manner akin to ngbind directives. Pure JavaScript expressions and AngularJS expressions output the data where they are utilized. Expressions in AngularJS are used to bind application data to HTML. The expressions are resolved by AngularJS and the result is returned to where the expression is written. The expressions in AngularJS are written in double braces:

{{ expression }}.

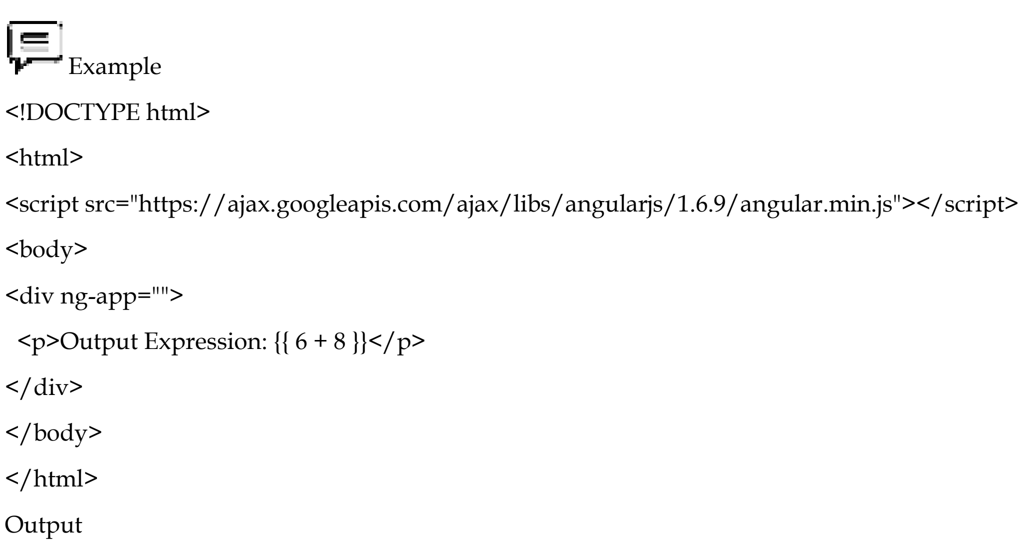
Syntax:

{{ expression }}

They behave similarly to ng-bind directives: ng-bind=" expression".

Expressions are used to bind application data to HTML and are written inside double braces in this manner: {{ expression}}. They behave in the same way as ng-bind directives. AngularJS application expressions are pure JavaScript expressions and output the data where they are used. They have the following qualities:

- Can be written inside double braces: {{ expression }}.
- Can also be written inside a directive: ng-bind="expression".
- AngularJS will resolve the expression, and return the result exactly where the expression is written.
- Much like JavaScript expressions, they can contain literals, operators, and variables.



Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
<div ng-app="">
  <p>Output Expression: {{ 6 + 8 }}</p>
</div>
</body>
</html>
```

Output

Output Expression:14

1.4 Architecture

MVC stands for Model View Controller, it is a software design pattern for designing and developing a web application. Here, we will learn MVC architecture & components and How MVC works in Angular JS.

So, are you ready to explore the Angular Model View Controller (MVC)?

- MVC is nothing but the software design pattern for designing and developing a web application. It is considered the architectural pattern that has existed for a long time in software engineering. Many languages use MVC architecture with little variations in each one of them but conceptually it remains the same.
- In angular, we organize things differently as compared to regular javascript. Therefore, Model View Controller architecture is the most popular and latest way of organizing an application.

AngularJs MVC Architecture & Components

The following Figure shows the various components of MVC:

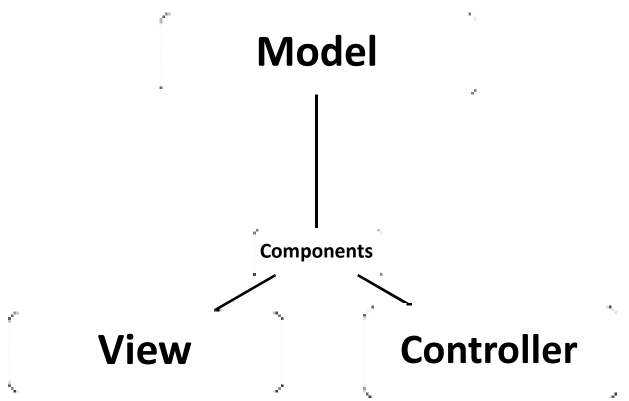


Figure 3: MVC Components

AngularJS MVC divides an application into three parts- Model Part, View Part, and Controller Part. Let's see for what purpose each component of MVC serves:

- **MODEL PART**

The model part consists of a database (use to store data), and application data (application data are variables that are specific to a user. Application and logic (Logic refers to code that is written to perform a certain task).

- **VIEW PART**

The view part is the user interface part of an application. The information that is being displayed to the user in an application is written in this part.

- **CONTROLLER PART**

The interaction between the model part and the view part is controlled by a controller. It is more like software code.

Working on MVC in AngularJS

We can implement MVC patterns through JavaScript and HTML. HTML helps to implement the model part, while JavaScript is for the model and controller part. When a user enters a URL in the browser, according to that specific URL control goes to the server and calls the controller. After that controller uses the appropriate model and appropriate view so that an appropriate response can be made to the user's request. It creates the response and that response is sent back to the user.

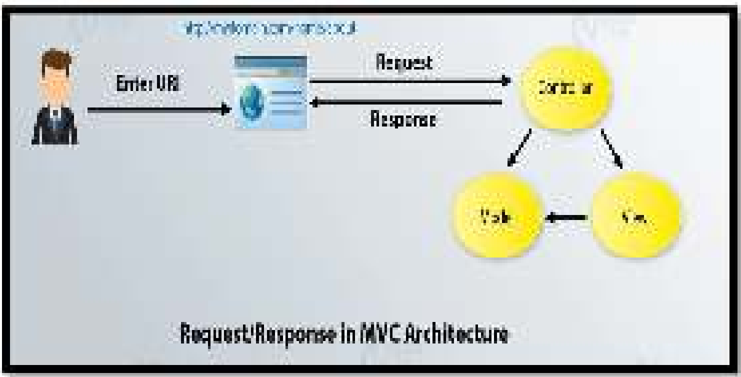


Figure 4:MVC Working

- Model Part In Angular Application
- It responds to the request made from the view part. It can obtain data dynamically means you can get data from a database like MySQL or you can get data from a static JSON (JavaScript Object Notation) file.

```
<script>
  $scope.person = {
                'Name':   'Ram Kumar ',
                  'Address':   'AB Road, Katni',
   }
</script>
```

View Part

- It is the presentation part of an application. Through the view part, a user can see the Model part. Therefore, the model and view parts join together.
- <h1> {{person.name}} </h1>

Controller

- It is the central part of the angular application. Everything starts with a controller in angular. The model part and the view part join with each other. It means whatever happens to the model part automatically affects the view part and whatever happens to the view part automatically affects the model part also. Controller joins the model and views the part.

```
function address($scope)
{
// Model Part is written here
}
```

1.5 Development Environment

To start with the AngularJS environment setup let's open **https://angularjs.org/** and there we can see something like as shown following

Figure 5: AngularJS Development Environment

Once we open the angularjs website it contains three options View on GitHub, Download, and Design Docs & Notes.

View on GitHub - This option will take us to GitHub, where we can see clear instructions to set up the environment for angularjs with documentation.

Download - By clicking on the download button it will open a dialog box as shown following image

Figure 6: AngularJS Download Options

Now let us understand what it contains and what they stand for actually

Branch - This tells about the version as this is easy to understand from what is written there. **1.3. X** is the stable version of the AngularJS framework and **1.4.X** is the latest one.

Build - This tells about in which format we can download the AngularJs library i.e. **Minified**, **Uncompressed**, or **Zip**.

CDN - CDN stands for Content Delivery Network which provides the facility to host your files (static files) on distributed servers rather than hosting them on a single server. There are several

free CDNs are provided by different providers like Microsoft, Google, etc. One benefit of using CDN here is, there will be no need to make extra calls to the server if you have already loaded AngularJs in your browser earlier. It will use the cached one. You can get more information on the web about what is CDN and the benefits of using CDN.

Bower - This command-based method to add AngularJs, first install bower to your system using "**$ npm install -g bower**" (supposing you already have installed npm). And then get AngularJs by the given command "**bower install angular#1.3.14**".Npm- or you can easily use the npm command to add AngularJs.

1.6 AngularJS Advantages

- Dependency Injection: Dependency Injection specifies a design pattern in which components are given their dependencies instead of hard coding them within the component.
- Two-way data binding: AngularJS creates a two-way data binding between the select element and the orderProp model. orderProp is then used as the input for the orderBy filter.
- Testing: Angular JS is designed in a way that we can test right from the start. So, it is very easy to test any of its components through unit testing and end-to-end testing.
- Model View Controller: In Angular JS, it is very easy to develop applications in a clean MVC way. You just have to split your application code into MVC components i.e. Model, View, and the Controller.

1.7 AngularJS Disadvantages

- Not Secure − Being a JavaScript-only framework, applications written in AngularJS are not safe. Server-side authentication and authorization are a must to keep an application secure.
- Not degradable − If the user of your application disables JavaScript, then nothing would be visible, except the basic page.

Example: Application logic is defined in a group of JavaScript classes called a "Controller."

Summary

- A structured JavaScript framework called AngularJS is used to create dynamic one-page applications.
- This Framework uses HTML code templates to carry out specific functions or commands.
- The AngularJS expressions are expressions that link data to HTML are expressed as.
- CDN stands for Content Delivery Network which provides the facility to host your files (static files) on distributed servers rather than hosting them on a single server.
- HTML helps to implement the model part, while JavaScript is for the model and controller part.
- A structural framework for dynamic web applications is AngularJS.
- It allows you to enhance HTML's syntax to express your application components quickly and unambiguously while still allowing you to utilize HTML as your template language.
- An effective framework for developing Rich Internet Applications (RIAs) is AngularJS.
- Developers have the opportunity to create client-side JavaScript apps with AngularJS in a neat Model View Controller (MVC) manner.

Keywords

Modules: A module is a container that houses different application components, according to one definition that contains a collection of functions contained in a JavaScript file that make up the module.

Directives: Directives tell the compiler to change or assign a behavior to a DOM element. Numerous directives are included in Angular JS, including ng-app, ng-controller, ng-view, ng-if, etc

Expressions: Expressions in Angular JS are written with the symbol {{}}, which in HTML denotes a data binding.

Routing: The Routing actions are handled by $routeProvider. It separates the map into different views. It assists in dividing Single Page Applications into various views.

Dependency Injection: Dependency Injection is a design approach that manages the dependencies of different program components. You can create loosely organized architectures using it.

Model: In AngularJS, a model is a basic data type like an integer, text, boolean, object, etc.

View: The Document Object Model (DOM) is what users see in AngularJS's view. Angular expressions that coordinate the model and view any changes can be introduced to the view to display the data from the controller.

Self Assessment

1. Which of the following statement is correct for AngularJS?

A. AngularJS is an HTML framework
B. AngularJS is a Java framework
C. AngularJS is a JavaScript framework
D. AngularJS is a SQL framework

2. AngularJS is perfect for _______.

A. Create Single Page Applications
B. Creating a Desktop Application
C. Create Web Services
D. None of these.

3. AngularJS is distributed as a _______.

A. JavaScript file
B. PHP file
C. XML file
D. ASP file

4. AngularJS expressions are written inside _____.

A. { expression }
B. [{ expression }]
C. _expression
D. {{ expression }}

5. Which directive can be used to write AngularJS expressions?

A. ng-expression
B. ng-bind
C. ng-statement
D. ng-bindexpression

6. An AngularJS module defines an ______.

A. expression
B. element
C. application
D. None of the above

7. Which of the following components can be injected as a dependency in AngularJS?

A. factory
B. service
C. value
D. All of the above

8. What is a Model in MVC?

A. the lowest level of the pattern responsible for maintaining data
B. represents server-side data
C. represents data stored in a database
D. None of the mentioned

9. Which of the following is an advantage of AngularJS?

A. AngularJS code is unit testable.
B. AngularJS provides reusable components.
C. AngularJS uses dependency injection and makes use of the separation of concerns.
D. All of the above

10. What is the use of Angular Controllers in the application?

A. Angular controllers are used for controlling the data.
B. Angular controllers are used for displaying the data.
C. Both of the above are correct.
D. None of the above is correct.

11. AngularJS applications are a mix of which of the following technologies?

A. HTML and PHP
B. HTML and JavaScript
C. HTML and TypeScript
D. PHP and JavaScript

12. Which directive initializes an AngularJS application?

A. ng-init

B. ng-model
C. ng-app
D. ng-application

13. Which directive defines the application controller?

A. ng-control
B. ng-controller
C. ng-NewController
D. None of the above

14. It can be added to an HTML page with a _______ tag.

A. style tag
B. php tag
C. using MySQL
D. script tag

15. Which of the following is true about AngularJs?

A. AngularJS is a very powerful JavaScript Framework
B. It extends HTML DOM with additional attributes and makes it more responsive to user actions.
C. AngularJS is open-source, completely free, and used by thousands of developers around the world.
D. All of the above

Answers for Self Assessment

1.	C	2.	A	3.	A	4.	D	5.	B
6.	C	7.	D	8.	A	9.	D	10.	A
11.	B	12.	C	13.	B	14.	D	15.	D

Review Questions

1. Explain the various components of Angularjs architecture.
2. Discuss in detail the working of MVC architecture.
3. Discuss in detail the core features of AngularJS.
4. How we can declare expression on AngularJS give an example.
5. What is the use of AngularJS? Write down the steps used to download and install AngularJS.
6. Write a note on the following:

a) Deep Linking

b) Model View Whatever

Further Readings

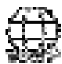
Ruebbelke, L. (2015). *AngularJS in Action*. Simon and Schuster.

Williamson, K. (2015). *Learning AngularJS: a guide to AngularJS development*. " O'Reilly Media, Inc.".

Green, B., & Seshadri, S. (2013). *AngularJS*. " O'Reilly Media, Inc.".

Branas, R. (2014). *AngularJS essentials* (pp. 1-180). Packt Publishing.

Grant, A. (2014). *Beginning AngularJS*. Apress.



Web Links

https://docs.angularjs.org/guide/expression

https://www.javatpoint.com/angularjs-expressions

https://www.knowledgehut.com/blog/web-development/angularjs-expressions

https://intellipaat.com/blog/tutorial/angularjs-tutorial/angularjs-expressions/

Unit 02: Angular Projects/Programs

CONTENTS

Objectives
Introduction
2.1 Your First Angular App
2.2 Structure of Angular Projects
2.3 Webpack
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After this unit you will be able to:

- Know the basic terminology used in Angular projects/programs.
- Understand the creation of first Angular app.
- Learns the concept of webpack.

Introduction

Angular is a popular JavaScript framework developed and maintained by Google. It is used to build dynamic and robust web applications. Angular follows the component-based architecture, where the application is divided into reusable components that encapsulate their own logic and UI.

Here's a brief introduction to Angular projects/programs:

Modules: Angular applications are organized into modules, which are containers for related components, services, directives, and other features. Modules help in organizing the code and managing dependencies. The root module is typically called App Module.

Components: Components are the building blocks of an Angular application. Each component represents a part of the UI and encapsulates its own template, styles, and logic. Components can communicate with each other using inputs and outputs.

Templates: Templates define the structure and layout of the UI for a component. They are usually written in HTML and may include Angular-specific syntax and directives to render dynamic content and bind data.

Services: Services in Angular are used for sharing data, implementing business logic, and performing API calls. They are typically used to encapsulate functionality that is independent of any specific component and can be injected into components and other services.

Dependency Injection: Angular has a built-in dependency injection (DI) system. DI allows you to provide instances of services or other dependencies to components as needed. It promotes code reusability, testability, and modularity.

Routing: Angular provides a powerful routing mechanism for building single-page applications (SPAs). The Angular Router allows you to define routes, associate them with components, and handle navigation between different views within the application.

Forms: Angular offers robust form handling capabilities, including template-driven forms and reactive forms (also known as model-driven forms). Forms in Angular provide features like form validation, data binding, and form submission.

Observables and RxJS: Angular leverages the power of Reactive Extensions for JavaScript (RxJS) to handle asynchronous operations, event streams, and data flow. Observables are used extensively in Angular for handling data streams and handling asynchronous operations.

Build and Development Tools: Angular provides the Angular CLI (Command Line Interface), which is a command-line tool used for scaffolding new projects, generating components, services, and other code files, as well as running development servers and building production-ready bundles.

Testing: Angular encourages and provides strong support for testing. It includes tools like Karma and Jasmine for writing unit tests and Protractor for end-to-end (E2E) testing. Testing in Angular helps ensure the quality and stability of the application. Angular offers a comprehensive set of tools and features that enable developers to build scalable, maintainable, and high-performing web applications.

2.1 Your First Angular App

The following steps you need to follow in order to create your first angular app:

Step 1: Set up the Development Environment

Install Node.js: Angular requires Node.js and npm (Node Package Manager). You can download and install them from the official Node.js website (https://nodejs.org).

Step 2: Install the Angular CLI

Open a terminal or command prompt.

Run the following command to install the Angular CLI globally: npm install -g @angular/cli.

Step 3: Create a New Angular App

Open a terminal or command prompt.

Navigate to the directory where you want to create your Angular app.

Run the following command to create a new Angular app: ng new my-app.

Replace my-app with the desired name for your app.

Step 4: Serve the Angular App

Navigate to the app directory: cd my-app.

Replace my-app with the name of your app.

Run the following command to start the development server and serve the app: ng serve.

Step 5: View Your App

Open a web browser.

Visit http://localhost:4200.

You should see your Angular app running in the browser.

Congratulations! You have created and served your first Angular app. Now you can start building your app by modifying the code in the src directory. The main file to look at is src/app/app.component.ts, where you can define the component's behavior and template.

Note: This is a basic setup to get you started. As you progress, you may want to explore Angular's features and concepts like components, modules, services, and routing to build more

complex applications. The official Angular documentation (https://angular.io/docs) is an excellent resource to learn more about Angular and its various capabilities.

2.2 Structure of Angular Projects

In Angular, projects are structured in a modular and component-based manner. Here is a typical structure of an Angular project:

src Folder: This is the root folder of your project where the source code resides.

app Folder: This folder contains the main application code.

app.component.ts: The main component that serves as the root of the application. It defines the behavior and structure of the main app component.

app.component.html: The HTML template associated with the app component.

app.component.css: The CSS styles specific to the app component.

app.module.ts: The main module that defines the application and its dependencies.

app-routing.module.ts: The routing module that handles navigation between different components.

assets Folder: This folder contains static assets like images, fonts, or configuration files used in the application.

environments Folder: This folder contains environment-specific configuration files like environment.ts and environment.prod.ts, which define environment variables for development and production environments.

index.html: The main HTML file that serves as the entry point for the application. It typically contains the <app-root> tag, which represents the root component.

styles.css: The global CSS file that applies styles to the entire application.

node_modules Folder: This folder contains all the dependencies (libraries, packages) installed via npm. It is generated automatically when you run npm install to install the project dependencies.

angular.json: The Angular project configuration file. It contains various configuration options for the project, including build settings, asset paths, and more.

package.json: The npm package configuration file. It lists the project dependencies, scripts, and other metadata.

tsconfig.json: The TypeScript configuration file. It specifies the compiler options and settings for the TypeScript compiler.

tslint.json: The TSLint configuration file. It defines the linting rules and settings for the project.

karma.conf.js: The configuration file for the Karma test runner. It specifies the test environment and settings for running unit tests.

protractor.conf.js: The configuration file for the Protractor end-to-end testing framework. It defines the settings for running end-to-end tests.

These are the main files and folders you will find in a typical Angular project. As your project grows, you may have additional folders and files for components, services, modules, and other resources based on your application's requirements.

Note: It's important to note that this is a general structure, and you can customize it based on your project's needs. Angular provides a flexible and modular architecture that allows you to organize your code in a way that makes sense for your application.

Example

- src/
  - app/
    - components/

```
      - home/
        - home.component.ts
        - home.component.html
        - home.component.scss
        - home.component.spec.ts
      - about/
        - about.component.ts
        - about.component.html
        - about.component.scss
        - about.component.spec.ts
    - services/
      - data.service.ts
      - auth.service.ts
    - models/
      - user.model.ts
    - shared/
      - header/
        - header.component.ts
        - header.component.html
        - header.component.scss
        - header.component.spec.ts
      - footer/
        - footer.component.ts
        - footer.component.html
        - footer.component.scss
        - footer.component.spec.ts
    - app.component.ts
    - app.component.html
    - app.component.scss
    - app.component.spec.ts
    - app.module.ts
    - app-routing.module.ts
  - assets/
    - images/
    - fonts/
  - environments/
    - environment.ts
    - environment.prod.ts
- index.html
- styles.scss
```

- angular.json

- package.json

- tsconfig.json

In this example:

The src folder is the main folder for source code.

Inside the app folder, you have subfolders such as components, services, models, and shared.

Components are organized within the components folder, with each component having its own folder containing TypeScript, HTML, SCSS, and unit test files.

Services are placed in the services folder.

Models are stored in the models folder.

Shared components, like a header and footer, are organized within the shared folder.

The app.component.ts, app.component.html, and app.component.scss files define the root component of the application.

The app.module.ts file is the main module where components, services, and other dependencies are declared.

The app-routing.module.ts file handles the routing configurations for the application.

The assets folder contains static assets like images and fonts.

The environments folder stores environment-specific configuration files.

The index.html file is the entry point for the application.

The styles.scss file is the main style file for the application.

Configuration files like angular.json, package.json, and tsconfig.json define project settings, dependencies, and TypeScript configurations.

2.3 Webpack

Webpack is not used in AngularJS (Angular version 1.x). Webpack is a module bundler that is commonly used in modern Angular applications (Angular version 2+). In AngularJS, the most common build tool was Grunt or Gulp, which were used for tasks like concatenating and minifying JavaScript files, compiling Sass or Less stylesheets, and more. These build tools focused on task automation and did not include a module bundler like Webpack.

However, if you are working with Angular (version 2+) or newer versions, Webpack is the recommended build tool and module bundler. It can handle a wide range of tasks, including bundling and optimizing JavaScript, CSS, and other assets, handling dependencies, and enabling features like lazy loading. With Webpack, you can configure various loaders and plugins to transform and process different types of files in your Angular application, such as TypeScript files, CSS stylesheets, HTML templates, and more. It helps manage dependencies, tree shaking, and code splitting to optimize the performance of your application.

In an Angular project, the Webpack configuration file (webpack.config.js) allows you to define the entry points, output paths, loaders, and plugins required to build and bundle your application. To set up an Angular project with Webpack, you would typically use the Angular CLI (Command Line Interface) to generate a new project. The Angular CLI internally uses Webpack for bundling and build processes. Overall, while Webpack is not used in AngularJS (Angular version 1.x), it plays a crucial role in modern Angular applications (Angular version 2+) for bundling and optimizing the application's code and assets.

Although AngularJS is not commonly used with Webpack, you can still configure Webpack to bundle and build an AngularJS application. Here's an example of how you can set up Webpack with AngularJS:

Step 1: Initialize a new AngularJS project

Create a new directory for your project and navigate to it in the terminal. Then, initialize a new AngularJS project by creating an index.html file and a main.js file with the following content:

index.html:

Example

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>AngularJS Webpack Example</title>
</head>
<body>
  <div ng-app="myApp">
    <div ng-controller="MainController">
      <h1>{{ greeting }}</h1>
    </div>
  </div>
  <script src="bundle.js"></script>
</body>
</html>
```

main.js

```
var app = angular.module('myApp', []);
app.controller('MainController', function($scope) {
  $scope.greeting = 'Hello, AngularJS with Webpack!';
});
```

Step 2: Install Webpack and necessary dependencies

In the project directory, install Webpack and its related dependencies using npm (Node Package Manager) by running the following commands:

```
npm init -y
npm install webpack webpack-cli angular --save-dev
```

Step 3: Create a Webpack configuration file

Create a webpack.config.js file in the project directory with the following configuration:

webpack.config.js:

```
const path = require('path');
module.exports = {
  entry: './main.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
```

```
  rules: [
    {
      test: /\.js$/,
      exclude: /node_modules/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env']
        }
      }
    }
  ]
 }
};
```

This configuration sets the entry point as main.js and specifies the output path and filename as dist/bundle.js. It also includes a rule to use Babel with the babel-loader to transpile JavaScript files.

Step 4: Build the AngularJS application.

Run the following command to build the AngularJS application using Webpack:

npx webpack --mode production

This command tells Webpack to bundle the code in production mode.

Step 5: View the bundled application.

Open the index.html file in a web browser, and you should see the output "Hello, AngularJS with Webpack!" displayed on the page.

Summary

- Angular is a popular open-source web development framework maintained by Google. It allows developers to build dynamic, single-page applications (SPAs) and cross-platform mobile applications.
- Angular uses a component-based architecture where the UI is divided into reusable components, each having its own logic and view.
- TypeScript is the primary language used in Angular development. It is a superset of JavaScript that adds static typing and additional features to enhance productivity and maintainability.
- Angular follows the MVC (Model-View-Controller) design pattern, where models represent the data, views handle the UI, and controllers (components) manage the application logic.
- Dependency Injection (DI) is a core feature of Angular that helps manage component dependencies and promotes modular and testable code.
- Angular provides a powerful templating system that allows developers to bind data to the UI, handle user interactions, and create dynamic content.
- Angular CLI (Command Line Interface) is a command-line tool that simplifies the creation, development, and deployment of Angular applications.
- It provides a set of predefined commands for generating components, services, modules, and more.

- Angular offers a comprehensive set of built-in features, such as routing, forms handling, HTTP client for server communication, and internationalization (i18n) support.
- Angular has a vibrant ecosystem with a wide range of third-party libraries and community-created packages available through npm (Node Package Manager).
- Angular has excellent tooling support, including debuggers, code editors with IntelliSense, testing frameworks (e.g., Jasmine, Karma), and performance profiling tools.
- Angular applications can be deployed to various platforms, including web browsers, desktops (using Electron), and mobile devices (using Ionic or NativeScript).
- Angular has a backward compatibility policy that ensures smooth migration and updates between different Angular versions.

Keywords

src Folder: This is the root folder of your project where the source code resides.

Template: Templates in Angular define the structure and layout of the UI.

Service: Services in Angular are used to share data, perform business logic, and provide common functionality across multiple components.

Form: Angular provides robust form handling capabilities, including two-way data binding, form validation, form controls, and form submission handling.

Angular CLI: Angular CLI (Command Line Interface) is a command-line tool that simplifies the development process by providing generators, build tools, and project scaffolding.

Self Assessment

1. What is the purpose of the Angular CLI in creating your first Angular app?

A. To provide a graphical user interface for creating Angular apps.
B. To manage dependencies and package installation.
C. To generate the basic project structure and configuration files.
D. To compile TypeScript code into JavaScript.

2. Which file is considered the entry point of an Angular app?

A. app.module.ts
B. app.component.html
C. index.html
D. main.ts

3. Which command is used to serve an Angular app during development?

A. ng start
B. ng serve
C. ng run
D. ng build

4. Which folder is used to store static assets like images and fonts in an Angular app?

A. src/app

B. src/assets

C. src/styles

D. src/environments

5. Where can you modify the behavior and structure of the main app component in an Angular app?

A. app.module.ts

B. app.component.ts

C. app.component.html

D. styles.css

6. Which folder is used to store the main application code in an Angular project?

A. src/app

B. src/assets

C. src/styles

D. src/environments

7. Which file serves as the entry point for the Angular application?

A. app.module.ts

B. app.component.ts

C. index.html

D. main.ts

8. What is the purpose of the node_modules folder in an Angular project?

A. It contains the compiled JavaScript files of the application.

B. It stores the configuration files for the Angular CLI.

C. It holds the external libraries and dependencies installed via npm.

D. It stores the built CSS stylesheets of the application.

9. Which file is responsible for defining the main module of an Angular application?

A. app.module.ts

B. app.component.ts

C. index.html

D. main.ts

10. What is the purpose of the angular.json file in an Angular project?

A. It stores the metadata and configuration of the Angular application.

B. It defines the routing configuration for the application.

C. It specifies the entry point and output paths for the Webpack bundler.

D. It contains the unit test configuration for the Karma test runner.

11. Which command is commonly used to run the Webpack bundler?

A. npm start
B. npm run dev
C. webpack serve
D. webpack build

12. Which configuration file is commonly used to define Webpack settings?

A. webpack.json
B. package.json
C. webpack.config.js
D. webpack.js

13. What is Webpack primarily used for?

A. Running JavaScript code in the browser
B. Styling web applications
C. Bundling and managing assets for web applications
D. Building server-side applications

14. What role does Webpack play in handling Angular templates and stylesheets?

A. Webpack compiles TypeScript code into JavaScript for Angular.
B. Webpack transpiles Angular templates and stylesheets into browser-compatible formats.
C. Webpack bundles Angular templates and stylesheets into separate files.
D. Webpack ensures that Angular templates and stylesheets are pre-compiled for performance.

15. Which configuration file is commonly used to customize Webpack in an Angular project?

A. angular.json
B. package.json
C. webpack.config.js
D. tsconfig.json

Answers for Self Assessment

1.	C	2.	D	3.	B	4.	B	5.	B
6.	A	7.	C	8.	C	9.	A	10.	A
11.	D	12.	C	13.	C	14.	B	15.	C

Review Questions

1. What is the basic structure of an Angular project? How are files and folders organized?
2. What is the purpose of the "src" folder in an Angular project? What types of files are typically found inside it?
3. How does Angular use Webpack for bundling and building the application?

4. What is a webpack.config.js file, and what are some common configurations used in an Angular project?
5. What is the purpose of the "services" folder in an Angular project? How are services organized and accessed by components?
6. Discuss the various steps required to create your first angular app.

Further Readings

Murray, N., Coury, F., Lerner, A., & Taborda, C. (2018). *Ng-book: The complete guide to Angular*. CreateSpace Independent Publishing Platform.

Wilken, J. (2018). *Angular in Action*. Simon and Schuster.

Hajian, M. (2019). *Progressive web apps with angular: create responsive, fast and reliable PWAs using angular*. Apress.

Seshadri, S. (2018). *Angular: Up and running: Learning angular, step by step*. " O'Reilly Media, Inc.".

Web Links

https://angular.io/quick-start

https://www.simplilearn.com/tutorials/angular-tutorial/angular-hello-world

https://www.freecodecamp.org/news/learn-how-to-create-your-first-angular-app-in-20-min-146201d9b5a7/

https://code.tutsplus.com/tutorials/creating-your-first-angular-app-basics--cms-30092

Unit 03: Type Script Fundamentals

CONTENTS

Objective
Introduction
3.1 Declaring Variables
3.2 Types
3.3 Type Assertion
3.4 Arrow Functions
Summary
Keywords
Self Assessment
Review Questions
Answers: Self-Assessment
Further Readings

Objective

After this unit the students will learn:

- The process of declaring and initialization of variables in typescript.
- Understand the different data types in typescript.
- Learn the concept of type assertion and its various techniques.
- Know the concept of arrow functions.
- Implementation of different types of arrow functions.

Introduction

TypeScript is a statically typed superset of JavaScript developed by Microsoft. It extends the capabilities of JavaScript by adding optional static typing, enhanced tooling, and additional features that make it easier to build large-scale web applications. TypeScript code is transpiled into plain JavaScript, which can then run on any browser or JavaScript runtime.

Here are some key aspects and features of TypeScript:

Static Typing: TypeScript introduces static types, allowing developers to explicitly declare the types of variables, function parameters, and return values. This provides compile-time type checking, enabling early detection of type-related errors and improving code quality. Static typing also enhances tooling support, as editors and IDEs can provide intelligent code completion, refactoring, and error checking based on type information.

Type Inference: TypeScript leverages type inference to automatically determine the types of variables based on their initial values. This means that in many cases, developers don't need to explicitly specify types, making the code cleaner and less verbose. Type inference also allows for seamless integration with existing JavaScript code, as TypeScript can infer types even without explicit type annotations.

Object-Oriented Programming (OOP) Features: TypeScript supports object-oriented programming principles, such as classes, interfaces, inheritance, and access modifiers. It provides a more structured and familiar approach to building applications, making code organization and

maintenance easier. TypeScript's class-based model allows for code reuse, encapsulation, and the implementation of advanced software patterns.

3.1 Declaring Variables

In TypeScript, you can declare variables using the let, const, or var keywords. Here's how you can declare variables using these keywords:

let: It allows you to declare variables that can be reassigned later.

let myVariable: string = "Hello, TypeScript!";

let myNumber: number = 42;

let myBoolean: boolean = true;

const: It declares a variable whose value cannot be reassigned once it is initialized. It is recommended to use const whenever possible to ensure immutability.

const PI: number = 3.14;

const myName: string = "John Doe";

var : It declares variables with function scope or global scope. However, it's generally recommended to use let instead of var in TypeScript.

var x: number = 10;

var y: string = "Hello";

Additionally, TypeScript supports type inference, meaning you can omit the type annotations if the variable's type can be inferred from its initialization value.

Example:

let myVariable = "Hello, TypeScript!"; // Type inference: myVariable is inferred as string

let myNumber = 42; // Type inference: myNumber is inferred as number

TypeScript also supports more advanced type annotations, such as union types, arrays, objects, etc. You can specify those types as needed based on your requirements.

It's worth noting that starting from TypeScript 4.4, you can also use the # symbol to declare private variables within classes or modules.

3.2 Types

Different types of values are supported by the TypeScript language. It gives JavaScript data types so that it can become a strongly typed programming language. Although data types are not supported by JavaScript, we can still leverage JavaScript's data types capability using TypeScript. When an object-oriented programmer wishes to use the type feature in any scripting language or object-oriented programming language, TypeScript is crucial. Before the programme uses the specified values, the Type System verifies their accuracy. It makes sure the code acts as it should.

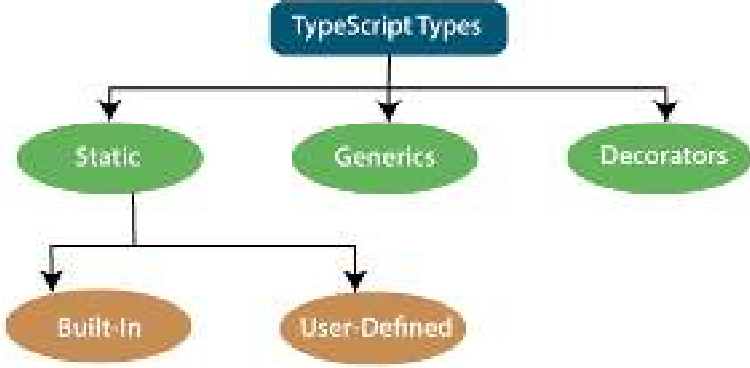


Figure 1: Typescript Types

In TypeScript, you can use various types to explicitly define the data type of variables, function parameters, and function return values. Here are some commonly used types in TypeScript:

Static Types : They are defined in type systems as "at compile time" or "without running a programme." Variables, parameters, and objects have types in statically typed languages that the compiler is aware of at compile time. This data was used by the compiler to carry out the type checking.

- Built-in or Primitive Type
- User-Defined DataType

Built-in Type

Number: All numbers in TypeScript are saved as floating-point values, much like in JavaScript. These numerical values are handled as a data type for numbers. Both integers and fractions can be represented using the numeric data type. Hexadecimal (Base 16), Binary (Base 2), Octal (Base 8), and Decimal (Base 10) literals are also supported by TypeScript.

Syntax:

let identifier: number = value;

String: To represent text in TypeScript, we'll use the string data type. String types operate on text-based data. By encapsulating string literals in single or double quotation marks, we can include them in our scripts. Additionally, it stands for a series of Unicode characters. The expressions were encoded in the form of $ {"expr"}.

Syntax

let identifier: string = " ";

Or

let identifier: string = ' ';

Boolean: Only two values are allowed for the Boolean data type, as opposed to an infinite number for the string and numeric data types. Both of these are "true" and "false." A Boolean value, often known as a truth value, indicates whether or not the condition is true.

Syntax

let identifier: BooleanBoolean = Boolean value;

Void :Functions that don't return any kind of value have a return type called void. When no other data type is available, it is utilised. Because we can only assign undefined or null to void variables, they are useless. Uninitialized variables are denoted by an undefined data type, while undefined variables are denoted by null.

Syntax

let unusable: void = undefined;

Null: A variable with an unknown value is represented by null. It is not really helpful on its own, much like the void. The only value that the Null accepts is null. The TypeScript Null type is defined with the Null keyword, but since we can only give it a null value, it serves no purpose.

Undefined: In TypeScript and JavaScript, all uninitialized variables are represented by the primitive type Undefined. It only has the value "undefined" as a value. The undefined keyword in TypeScript defines the undefined type, but since we can only give it an undefined value, it serves no purpose.

Any Type: It is the "super type" in TypeScript for all data types. To represent any JavaScript value, it is utilised. We can choose whether or not to use type-checking during compilation. A variable may be declared using the data type "Any" if it cannot be represented by any of the fundamental data types. Any type is helpful when we want to omit type-checking at build time but do not know the type of the value (which could come from an API or third-party library).

Syntax

let identifier: any = value;

User-Defined DataType

Array: An array is a group of objects of the same data type. Using arrays of values in TypeScript is like doing so in JavaScript. There are two ways to write an array:

1. To indicate an array of a particular element type, use [] after the element type:

   var list : number[] = [1, 3, 5];

2. The second way uses a generic array type:

   var list : Array**<number>** = [1, 3, 5];

Tuple: A data type called a tuple contains two sets of values of various data kinds. It enables us to express an array in which a fixed number of members have known types but are not identical. For instance, the following could be written to represent a value as a pair of a number and a string:

// Declare a tuple

let a: [string, number];

// Initialize it

a = ["hi", 8, "how", 5]; // OK

Interface: A structure that serves as a contract for our application is called an interface. It specifies the grammar that classes must adhere to, which implies that a class that implements an interface must also implement all of its members. Although it cannot be instantiated, the class that implements it may refer to it. The type-checking interface used by the TypeScript compiler is also referred to as "duck typing" or "structural subtyping."

Class: Classes serve as a model for building objects and are used to build reusable components. It is a logical object that houses functions and variables for carrying out actions. Classes in TypeScript are now supported by ES6. It is distinct from an interface because an implementation is contained within it, whereas an interface does not.

Enums: A set of named constants are defined by enums. Both string-based and numeric-based enums are available in TypeScript. Enums number their elements sequentially by default, but we can override this behaviour by explicitly altering the value of one of the members. Enums are now supported in TypeScript thanks to ES6.

Function: The logical building elements of a programme are called functions. In the same way that JavaScript allows for the creation of named and anonymous functions, TypeScript does the same. Our programme is readable, maintainable, and reusable thanks to functions. The name, return type, and parameters of a function are listed in its declaration.

Generic

Generic is used to create a component which can work with a variety of data type rather than a single one. It allows a way to create reusable components. It ensures that the program is flexible as well as scalable in the long term. TypeScript uses generics with the type variable <T> that denotes types. The type of generic functions is just like non-generic functions, with the type parameters listed first, similarly to function declarations.

Decorator

An accessor, method, property, accessor, and parameter can all have decorators attached to them. A decorator is a specific type of data. For classes and functions, it offers a means to include annotations as well as a meta-programming syntax. Use the "@" symbol when using it.

A decorator is a test feature that might be altered in subsequent releases. We must either activate the experimental Decorators compiler option on the command line or in our tsconfig.json file in order to support the decorator.

3.3 Type Assertion

A feature in TypeScript called type assertion notifies the compiler of the variable type. If TypeScript determines that the assignment is incorrect, we can override the type using a type assertion. When using a type assertion, we must be positive that we are correct because the assignment is always valid. Otherwise, our programme might not run correctly.

Type assertion is the process of explicitly telling the compiler that we plan to treat the object as a separate type. We can consider any as a number or a number as a string using this. Type assertion is widely used while porting code from JavaScript to TypeScript.

Declaratively informing the compiler that we intend to regard the item as a distinct type is known as type assertion. We can regard any as a number or a number as a string thanks to this. When moving code from JavaScript to TypeScript, type assertion is frequently utilised.

Although type assertion functions similarly to typecasting, it does not do type checking or data reorganisation, as C# and Java may. While type assertion has no effect on runtime, type casting has runtime support. Type assertions, on the other hand, are entirely compile-time constructs that give the compiler instructions on how we want our code to be analysed.

Techniques used in Type Assertion

A Typescript technique called type assertion informs the compiler of the variable type. Typecasting recreates code, whereas type assertion does not. By utilising type assertion, you can instruct the compiler not to guess the type of a value. To change a variable from one type to another, such as any to a number, we use a type assertion.

Either the "as" operator or the ">" operator can be used to do type assertion. While type assertion has no effect on runtime, typecasting offers support for runtime.

There are three ways to use Type Assertion in TypeScript, and they are as follows:

- Using “as” operator
- Using “<>” operator
- Using object

Type Assertion using as Operator

Syntax

let variable_any: any = 456

let variable_number: number = variable_any as number

In the above syntax, we used the “as” keyword on any type variable for type assertion.

Example:

let Value: any = "Hello, Everyone!";

let Len: number = (Value as string).length;

console.log(Len)

output

16

Type Assertion using <> Operator

The “<>” operator is another way to perform type assertion in TypeScript.

Syntax

let variable_any: any = 123

let variable_number: number = <number> variable_any

In the above syntax, we used the “<>” operator on any type variable for type assertion.

Example

```
let Value: any = "Hello, All!";
let Len: number = (<string>Value).length;
```

output

11

Type Assertion using Object

Objects are another way to perform the type assertion, unlike the “as” and “<>” operators; the objects can use for multiple type assertions at once.

Syntax

```
interface info {
   name: string,
   value: string
}
let my_obj = <info> { name: 'ABC', value: 'abc'}
```

In the above syntax, we used the object to perform type assertion.

Example

```
interface baby {
  name: string;
  age: number;
}

let babyObject: object = {
  name: "Seerat",
  age: 20,
};
let baby: baby = babyObject as baby;
console.log(baby.name)
console.log(baby.age)
```

output

Seerat

20

3.4 Arrow Functions

As a shortcut syntax for declaring the anonymous function, or for function expressions, TypeScript offers the arrow function. The function keyword is missing. Since -> is a thin arrow and => is a "fat" arrow, we can refer to it as a fat arrow. Another name for it is a Lambda function. The "this" keyword is lexically scoped by the arrow function. The motivation for arrow function is:

- When it is not necessary to continue typing.
- It accurately conveys the meaning of this keyword lexically.
- It expresses arguments' meaning lexically.

- This technique can be used whenever we want our code to be compact and avoid repeatedly using the function.

Syntax

(p1, p2, ..., pN) => expression

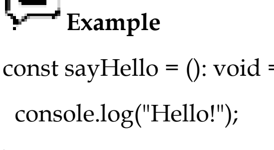
We can split the syntax of an Arrow function into three parts:

- **Parameters:** A function may or may not have parameters.
- **The arrow notation/lambda notation** (=>)
- **Statements:** It represents the function's instruction set.
- If we use the **fat arrow (=>)** notation, there is no need to use the **function** keyword. Parameters are passed in the brackets (), and the function expression is enclosed within the curly brackets {}.
- Parameters are passed in the parenthesis (), and the function expression is enclosed within the curly brackets { }.

Classification of Arrow Functions

- **Arrow Functions Without Parameter**

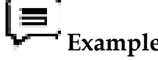
➢ Arrow Functions With No Parameter And No Return Value



Example

```
const sayHello = (): void => {
  console.log("Hello!");
};
sayHello();
 Output: Hello!
```

➢ Arrow Functions With No Parameter But Returns Some Value



Example

```
const getRandomNumber = (): number => {
  return Math.random();
};
const randomNumber = getRandomNumber();
console.log(randomNumber);
 Output: A random number between 0 and 1
```

- **Arrow Functions With Parameter**

➢ Arrow Functions With Parameters But Returns No Value

Example

```
const display = (name: string): void => {
  console.log(`Hi, ${name}!`);
};
greet("Shallu");
Output: Hi, Shallu!
```

- Arrow Function With Parameters And Returns Some Value

Example

```
// Arrow function with explicit return type
const add = (a: number, b: number): number => {
  return a + b;
};
console.log(add(3, 5));
Output: 8
```

- **Arrow Functions in a Class**

Example

```
class MyClass {
  private Name: string;

  constructor() {
    this.Name = "Great Day";
  }

  myArrowFunction = (): void => {
    console.log(this.Name);
  }
}

const myInstance = new MyClass();
myInstance.myArrowFunction();
Output: Great Day
```

Summary

- The var keyword (older syntax) can also be used to declare variables, but it's recommended to use let instead.
- TypeScript supports type inference, allowing you to omit type annotations if the type can be inferred from the initialization value.
- Objects can be represented using {} or specific key-value types using interfaces or type aliases.
- Arrow functions are defined using the "arrow" (=>) syntax. They can have either an implicit return or a block body.
- If the function body is a single expression, it is implicitly returned without using the return keyword.
- A Typescript technique called type assertion informs the compiler of the variable type.
- Although type assertion functions similarly to typecasting, it does not do type checking or data reorganisation, as C# and Java.

Keywords

Implicit return: If the function body is a single expression, it is implicitly returned without using the return keyword.

No arguments object: Arrow functions do not have their own arguments object. Instead, you can use rest parameters to access the function's arguments.

let: Used to declare a mutable variable with block scope.

const: Used to declare a read-only variable with block scope. Once initialized, the value cannot be reassigned.

var: Used to declare a mutable variable with function scope or global scope. It is hoisted to the top of its scope and has less strict scoping rules compared to let and const.

undefined: Represents an uninitialized variable or a variable that has been explicitly assigned the value undefined.

unknown: The unknown type in TypeScript can be used in combination with type assertion to assert a value to a more specific type. This is useful when dealing with dynamically-typed values.

Self Assessment

1. What are variables in Typescript?

A. A variable name should contain alphabets and numeric digits

B. It cannot contain spaces and special characters except underscore (_) and dollar ($) sign

C. A variable name cannot begin with a digit

D. All

2 Which keyword is used to declare a variable in TypeScript?

A. var

B. let

C. const

D. All of the above.

3. What is the difference between "let" and "const" in TypeScript?

A. "let" is used for mutable variables, while "const" is used for immutable variables.

B. "let" is block-scoped, while "const" is function-scoped.

C. "let" allows reassignment, while "const" does not.

D. Both a) and c)

4. Which of the following is NOT a valid data type for variable declaration in TypeScript?

A. number

B. boolean

C. string

D. any

5. What is the purpose of using type annotations in TypeScript variable declarations?

A. To explicitly specify the data type of the variable.

B. To restrict the variable to a specific range of values.

C. To enable type checking and catch potential errors.

D. All of the above

6. How can you declare a variable without specifying its type in TypeScript?

A. By using the "let" keyword.
B. By using the "const" keyword.
C. By using the "var" keyword.
D. TypeScript does not allow variables without type annotations.

7. Which of the following is a correct syntax for an arrow function in TypeScript?

A. function() { }
B. () => { }
C. => function { }
D. -> { }

8. Arrow functions in TypeScript automatically bind to which of the following?

A. Global scope
B. Local scope
C. Lexical scope
D. Prototype scope

9. What is the main advantage of using arrow functions in TypeScript?

A. They provide a concise syntax for writing functions.
B. They have better performance compared to regular functions.
C. They allow for dynamic scoping.
D. They can be used as class constructors.

10. Which of the following is true regarding the 'this' keyword in arrow functions?

A. Arrow functions have their own 'this' context.
B. Arrow functions inherit the 'this' context from their surrounding scope.
C. Arrow functions cannot access the 'this' keyword.
D. Arrow functions require explicit binding of the 'this' keyword.

11. When should you prefer arrow functions over regular functions in TypeScript?

A. When you need access to the 'this' keyword.
B. When you want to create a recursive function.
C. When you need to use the 'arguments' object.
D. When you need to define a method within an object.

12. What is type assertion in TypeScript?

A. Converting one data type to another
B. Checking if a variable holds a specific type

C. Asserting the correctness of a variable's type

D. All of the above

13. Which TypeScript operator is used for type assertion?

A. <>

B. ::

C. as

D. typeof

14. Which of the following is a valid type assertion syntax in TypeScript?

A. (variableName: Type)

B. variableName as Type

C. <Type>variableName

D. All of the above

15. Which statement is true about type assertion in TypeScript?

A. Type assertions are mandatory for all variables

B. Type assertions allow for static type checking at compile time

C. Type assertions can only be used with primitive data types

D. Type assertions are exclusive to TypeScript and not found in other programming languages

Review Questions

1. Explain the concept of variable declaration and initialization in TypeScript.
2. Describe the difference between let, const, and var in TypeScript.
3. Discuss the limitations of arrow functions in TypeScript and situations where traditional function declarations are still necessary.
4. Explain the concept of arrow functions in TypeScript and discuss their advantages over traditional function declarations.
5. Explain the concept of type assertion in TypeScript and discuss its significance in static type checking.
6. Discuss the different ways of using type assertions by giving the example of each type.

Answers: Self-Assessment

1	d	2	d	3	d
4	d	5	d	6	c
7	b	8	c	9	a
10	b	11	a	12	b
13	c	14	d	15	b

Further Readings

Books

Bierman, G., Abadi, M., & Torgersen, M. (2014). Understanding typescript. In ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28 (pp. 257-281). Springer Berlin Heidelberg.

Fenton, S., Fenton, & Spearing. (2014). Pro TypeScript. Apress.

Nance, C. (2014). Typescript essentials. Packt Publishing Ltd.

Cherny, B. (2019). Programming TypeScript: making your JavaScript applications scale. O'Reilly Media.

Maharry, D. (2013). TypeScript revealed. Apress.

Online Links

https://www.typescriptlang.org/docs/handbook/variable-declarations.html

https://www.tutorialsteacher.com/typescript/typescript-variable

https://www.tutlane.com/tutorial/typescript/typescript-variables

https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

https://blog.logrocket.com/assertion-functions-typescript/

Unit 04: Classes and Objects in Angular

CONTENTS

Objectives
Introduction
4.1 Interface
4.2 Classes
4.3 Object Creation
4.4 Constructors
4.5 Access Specifier
4.6 Properties
4.7 Module
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After this unit, you will be able to

- Understand the concept of interface, classes, objects and their implementation.
- Learn the different methods of object initialization and their implementation.
- Know the different types of constructors and their implementation.
- Discover the different types of access specifiers and their implementation.

Introduction

classes and objects play a fundamental role in building applications using object-oriented programming principles. Understanding classes and objects is essential for building components, services, and other parts of an Angular application. By leveraging classes and objects in Angular, you can build scalable, maintainable, and structured applications following the principles of object-oriented programming.

4.1 Interface

A structure that serves as a contract for our application is called an interface. It specifies the grammar that classes must adhere to, which implies that a class that implements an interface must also implement all its members. The interface cannot be instantiated by us, but the class object that implements it can make references to it. Whether an object has a particular structure or not, the TypeScript compiler employs interface for type-checking (also known as "duck typing" or "structural subtyping"). The implementation is not included in the interface; only the declaration of the fields and methods is. It is useless for construction. A class inherits it, and the class that implements the interface defines all its members.

The interface will be removed from the JavaScript file when the Typescript compiler converts it to JavaScript. Therefore, it merely serves to aid in the development stage.

Interface Declaration

```
interface interface_name {
        // variables' declaration
        // methods' declaration
}
```

- An interface is a keyword which is used to declare a TypeScript Interface.
- An interface_name is the name of the interface.
- An interface body contains variables and methods declarations.

Example

```
interface stu {
   name: String;
   Roll: number;
}
let stuinfo = (type: stu): void => {
  console.log('My Name is  ' + type.name + ' having ' + type.Roll + ' number');
};
let s1 = {name: 'Hello', Roll: 22}
stuinfo(s1);
```

Output

```
C:\Users\Harjinder Kaur\OneDrive\Desktop>tsc s1.ts

C:\Users\Harjinder Kaur\OneDrive\Desktop>node s1.js
My Name is  Hello having 22 number
```

Array Type Interface

In Angular, you can use interfaces to define the structure of objects, including arrays. When you want to specify the shape of an array, you can create an array type interface that describes the elements inside the array. This is helpful when you want to maintain consistency and type safety in your code. We can also use interfaces to describe the array type. The following example helps us to understand the Array Type Interface.

Example

```
interface nameArray {
   [index:number]:string
}
// use of the interface
let myNames: nameArray;
myNames = ['Ram', 'Rohit', 'Sharan'];


// Array which return number
interface ageArray {
```

```
    [index:number]:number
}
 var myAges: ageArray;
myAges =[10, 18, 25];
for(let i=0;i<3;i++)
{
console.log('My age is '+myAges[i]+' and name is '+myNames[i])
 }
```

Output

```
C:\Users\Harjinder Kaur\OneDrive\Desktop>tsc a1.ts

C:\Users\Harjinder Kaur\OneDrive\Desktop>node a1.js
My age is 10 and name is Ram
My age is 18 and name is Rohit
My age is 25 and name is Sharan
```

Interface in a class

TypeScript also allows us to use the interface in a class. A class implements the interface by using the **implements** keyword. This allows you to create a local interface specific to that class, encapsulating the structure of certain properties or methods. It can be helpful when you want to provide more context and encapsulation for a specific class's properties or methods.

Example

```
interface Person {
    FName: string;
    LName: string;
    age: number;
    FullName();
    GetAge();
}
// implementing the interface
class Student implements Person {
    FName: string;
    LName: string;
    age:number;
FullName() {
        return this.FName + ' ' + this.LName;
    }
    GetAge() {
        return this.age;
    }
    constructor(firstN: string, lastN: string, getAge: number) {
        this.FName = firstN;
        this.LName = lastN;
        this.age = getAge;
```

```
  }
}
// using the class that implements interface
let myStudent = new Student('AnKit', 'Sharma', 15);
let fullName = myStudent.FullName();
let Age = myStudent.GetAge();
console.log("Name of Person: " +fullName + '\nAge: ' + Age);
```

Output

```
C:\Users\Harjinder Kaur\OneDrive\Desktop>tsc a1.ts

C:\Users\Harjinder Kaur\OneDrive\Desktop>node a1.js
Name of Person: AnKit Sharma
Age: 15
```

Interface Inheritance

In Angular (which uses TypeScript), interface inheritance allows you to create new interfaces based on existing ones, inheriting their properties and methods. This is a fundamental concept in Object-Oriented Programming (OOP) that promotes code reuse and provides a way to create more specialized interfaces from more general ones.

We can inherit the interface from the other interfaces. In other words, Typescript allows an interface to be inherited from zero or more **base types**. The base type can be a **class** or **interface**. We can use the "**extends**" keyword to implement inheritance among interfaces.

Syntax

```
child_interface extends parent interface{
}
```

Example

```
interface Person {
  name:string
}
interface Detail {
   age:number
   gender:string
}
interface Student extends Person, Detail {
   Stid:number
}
let O1 = <Student>{};
O1.name = "Rahul"
O1.age = 21
O1.gender = "Male"
O1.Stid = 43
console.log("Name: "+O1.name);
```

```
console.log("Student Code: "+O1.Stid);
```

Output

```
C:\Users\Harjinder Kaur\OneDrive\Desktop>tsc a1.ts

C:\Users\Harjinder Kaur\OneDrive\Desktop>node a1.js
Name: Rahul
Student Code: 43
```

4.2 Classes

In Angular, a class is a fundamental concept used to create reusable and maintainable components, services, and other building blocks of an application. A class in Angular is typically a TypeScript class that represents a blueprint for creating objects with specific properties and methods. Classes are the essential building blocks for creating reusable components in object-oriented programming languages like Java. It is a collection of items with similar characteristics. A class is an OOP template or blueprint for building objects. It makes sense. Since TypeScript is an object-oriented version of JavaScript, it enables features of object-oriented programming including classes, interfaces, polymorphism, databinding, etc. Classes were not supported in JavaScript ES5 or older versions. This feature is supported by TypeScript starting with ES6. Because TypeScript is based on the ES6 version of JavaScript, it comes with built-in support for using classes. Today, a lot of programmers create class-based object-oriented languages and compile them into JavaScript, which is compatible with all the main operating systems and browsers.

Class Declaration

```
class <class_name>{
   field;
   method;
}
```

- **Fields:** It is a variable declared in a class.
- **Methods:** It represents an action for the object.
- **Constructors:** It is responsible for initializing the object in memory.
- **Nested class and interface:** It means a class can contain another class

Example

```
//Defining a Student class.
class Student {
   //defining fields
   Roll: number;
   Name:string;
   Course:String;
    //creating method or function
   display():void {
      console.log("Student Roll is: "+this.Roll)
      console.log("Student Name is: "+this.Name)
     console.log("Student Course is: "+this.Course)
   }
}
//Creating an object or instance
```

```
let obj = new Student();
obj.Roll = 101;
obj.Name = "Suraj Nehra";
obj.Course="MCA";
obj.display();
```

Output

```
C:\Users\Harjinder Kaur\OneDrive\Desktop>tsc a1.ts

C:\Users\Harjinder Kaur\OneDrive\Desktop>node a1.js
Student Roll is: 101
Student Name is: Suraj Nehra
Student Course is: MCA
```

4.3 Object Creation

A class creates an object by using the new keyword followed by the class name. The new keyword allocates memory for object creation at runtime. All objects get memory in heap memory area. We can create an object as below.

Syntax

let object_name = new class_name(parameter)

new keyword: it is used for instantiating the object in memory.

The right side of the expression invokes the constructor, which can pass values.

Object Initialization

Object initialization means storing of data into the object. There are three ways to initialize an object. These are:

- By reference variable: Object initialization means storing of data into the object. There are three ways to initialize an object.
- By method: A method is similar to a function used to expose the behavior of an object. Advantages : Code reusabiltity and code optimization
- By Constructor: A constructor is used to **initialize** an object. In TypeScript, the constructor method is always defined with the name "**constructor**." In the constructor, we can access the member of a class by using **this** keyword.

Example

```
//Defining a Student class.
class Student {
   //defining fields
   id: number;
   name:string;
       //defining constructor
  constructor(id: number, name:string) {
     this.id = id;
     this.name = name;
  }
 //creating method or function
```

```
  display():void {
    console.log("Function displays Student ID is: "+this.id)
    console.log("Function displays Student ID is: "+this.name)
  }
}
//Creating an object or instance
let obj = new Student(11, "Deepak Sharma")
//access the field
console.log("Reading attribute value of Student as: " +obj.id,)
console.log("Reading attribute value of Student as: " +obj.name)
//access the method or function
obj.display()
```

Output

```
C:\Users\Harjinder Kaur\OneDrive\Desktop>tsc a1.ts

C:\Users\Harjinder Kaur\OneDrive\Desktop>node a1.js
Reading attribute value of Student as: 11
Reading attribute value of Student as: Deepak Sharma
Function displays Student ID is: 11
Function displays Student ID is: Deepak Sharma
```

Note: It is not necessary to always have a constructor in the class.

4.4 Constructors

A constructor is a special method available in a class that is automatically called when an instance of the class is created. It is used to initialize the object and perform any necessary setup or configuration for the class. A constructor is a unique function found inside a class that is only called once during object creation. Because classes are distinct, so too are the calls to the constructor functions.

Functions and class constructors have a lot in common. You can include parameters with default values and type annotations. A method called an Angular constructor is used to initialize an Angular application. The Angular environment is set up by the constructor, which is called when the application is first formed. Dependencies can be injected, default values for properties can be provided, and any further initialization code can be run in the constructor.

Syntax

```
class classname {
constructor() {}
}
```

Example

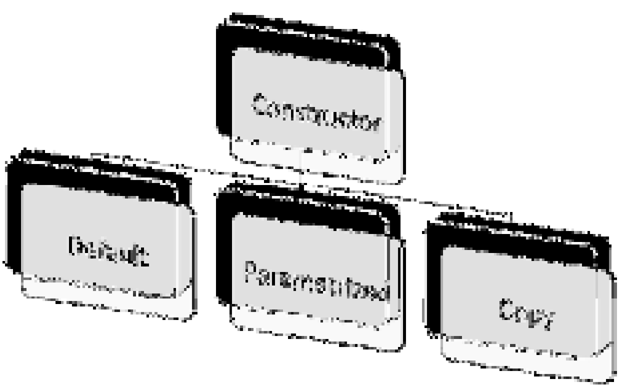
```
class Student {
  name: string;
  roll: number;
  constructor(name: string, roll: number) {
    this.name = name;
    this.roll = roll;
```

```
  }
  getName(): string {
    return this.name;
  }
getRoll(): number {
    return this.roll;
  }
}
let o1 = new Student('Es', 39);
 // 'Espresso'
console.log(o1.getName());
console.log(o1.getRoll());
```

Output

```
C:\Users\Harjinder Kaur\OneDrive\Desktop>tsc a1.ts

C:\Users\Harjinder Kaur\OneDrive\Desktop>node a1.js
Anil
23
```

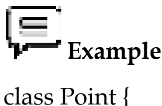
Types of Constructors



Constructor Types

Default Constructor

a default constructor is a constructor that is automatically generated by the TypeScript compiler when you don't explicitly define any constructor within a class. This default constructor is provided to the class if you haven't written any custom constructors. The default constructor is usually an empty constructor that doesn't take any parameters or perform any specific actions. Its main purpose is to ensure that the class can be instantiated and provide a way to initialize the object later using setter methods or property assignments.

 **Example**

```
class Point {
```

```
  x: number;
  y: number;

  constructor(x = 10, y = 20) {
    this.x = x;
    this.y = y;
  }
}
//Using default Values
let p1 = new Point();
console.log(p1)  //Point: {   "x": 10,   "y": 20 }

let p2 = new Point(100,200);
console.log(p2)  //Point: {   "x": 100,   "y": 200 }
```

Output

```
C:\Users\Harjinder Kaur\OneDrive\Desktop>tsc a1.ts

C:\Users\Harjinder Kaur\OneDrive\Desktop>node a1.js
Point { x: 10, y: 20 }
Point { x: 100, y: 200 }
```

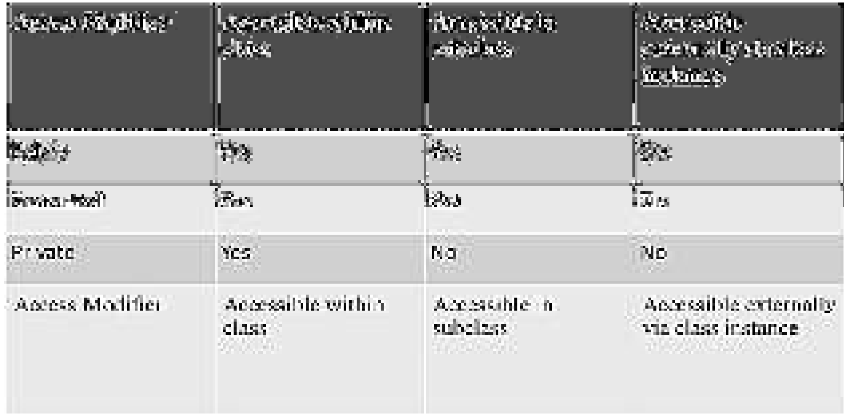
4.5 Access Specifier

We can use access modifiers at the class level in Typescript just like we can in other programming languages. It grants the class member immediate access control. Functions and properties make up these class members. Class members can be used within the class itself, outside the class, or in a child or derived class.

The access modifier makes the class members more secure and shields them from unauthorised use. It can also be used to manage a class's data members' visibility. TypeScript automatically sets the public access modifier to every member of the class when an access modifier does not need to be set for the class.

The following are the different types of access specifiers:

1. **Public** - By default, members (properties and methods) of the TypeScript class are public - so you don’t need to prefix members with the public keyword. Public members are accessible everywhere without restrictions even from the multiple level sub-classes without any compile errors.
2. **Private** - A private member cannot be accessed outside of its containing class. Private members can be accessed only within the class and even their sub-classes won't be allowed to use their private properties and attributes.
3. **Protected** - A protected member cannot be accessed outside of its containing class. Protected members can be accessed only within the class and by the instance of its sub/child class.



[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
Private	Yes	No	No
Access Modifier	Accessible within class	Accessible in subclass	Accessible externally via class instance

Accessibility of Access Specifiers

4.6 Properties

According to what we have previously learned and understood, class objects cannot be used to access secret and protected members outside of the class. Using the get and set keywords OR the getter() and setter() methods, properties enable us to access private members outside of the class. Basically, you may fetch a field's value and assign a new value to it once you have access to it somewhere.

A concept called property is comparable to a field but has additional unique characteristics. Three categories of property exist:

Read-only Property: Permits access to its value but prevents the addition of new values.

Write-only Property: Enables changing its value. Accessing this property will return an undefined value, nonetheless.

Read/Write Property: Sets a new value for it and provides access to the current value.

The TypeScript design principle states that you should utilise properties instead of fields for communication and define all the class's fields as private.

The following are the methods of the Typescript accessor property:

- **getter:** This method comes when you want to access any property of an object. A getter is also called an accessor.
- **setter:** This method comes when you want to change any property of an object. A setter is also known as a mutator.

Getter

The Getter syntax allows you to define a property that allows access to its value, but cannot set a new value for it unless you also define a Setter for this property.

Syntax:

```
get property_name(): data_type {
    // code ...
    return a_value;
}
```

Setter

The Setter syntax allows you to define a property, allowing you to assign a new value to it. However, without the Getter, trying to access this property will get underfined value.

Syntax:

```
set property_name(new_value: data_type) {
    // code ...
```

```
    }
    // Or:
    [private, protected, public] set property_name(new_value: data_type) {
        // code ...
}
```

Example

```
// Implementation of getter and setter method
class employee {
  // creating private class members
  private emp_Name: string = "Shubham Vora";
  private age_of_emp: number = 22;
  private role: string = "Content Writer";
  // getters to get the name of the employee
  public get name() {
    return this.emp_Name;
  }
  public set name(new_value: string) {
    this.emp_Name = new_value;
  }
}
// creating the object of the Student class
let man = new employee();

// Call the getter without paranthesis
let name_Value = man.name;
// Print the name
console.log("The name of the employee is " + name_Value);
//   update the employee name using the setters
man.name = "Rohit Sharma";
console.log(
  "The name of the employee after updating it using the setters is " + man.name
);
```

Output

```
C:\Users\Harjinder Kaur\OneDrive\Desktop>tsc a1.ts

C:\Users\Harjinder Kaur\OneDrive\Desktop>node a1.js
The name of the employee is Shubham Vora
The name of the employee after updating it using the setters is Rohit Sharma
```

4.7 Module

A group of connected variables, functions, classes, interfaces, etc. can be created as a module. Instead of the global scope, it operates in the local scope. In other words, a module's defined variables, functions, classes, and interfaces cannot be directly accessed from outside the module. The export keyword can be used to construct a module, while the import keyword can be used in other modules. A module loader is used by modules to import another module. All module's dependencies must be found and executed at runtime in order for the module to be executed. The CommonJS module loader for Node.js and require.js for Web applications are the two most popular module loaders used in JavaScript.

Categories of Module

- Internal Module
- External Module

Internal Module

Internal modules were in the earlier version of Typescript. It was used for logical grouping of the classes, interfaces, functions, variables into a single unit and can be exported in another module. The modules are named as a namespace in the latest version of TypeScript. So today, internal modules are obsolete. But they are still supported by using namespace over internal modules.

Syntax

```
module Module_Name {
 export function functionname(arguments) {
Set of statements;
  }
```

External Module

External modules are also known as a module. When the applications consisting of hundreds of files, then it is almost impossible to handle these files without a modular approach. External Module is used to specify the load dependencies between the multiple external js files. If the application has only one js file, the external module is not relevant. ECMAScript 2015(ES6) module system treats every file as a module.

Module declaration

- We can declare a module by using the export keyword. The syntax for the module declaration is given below.

```
export interface Interfacename {
  //code declarations
}
```

- We can use the declare module in other files by using an import keyword, which looks like below. The file/module name is specified without an extension.
- import { class/interface name } from 'module_name';

Example

```
// Module Creation
// creation of module
export class Largest{
   constructor(private x?: number, private y?: number){
   }
```

```
   larges(){
if(this.x>this.y)
{
      console.log("largest is " +this.x );
   }
else
{
console.log("largest is " +this.y );
}
}
}
export class Smallest{
   constructor(private x?: number, private y?: number){
   }
   smalles(){
if(this.x<this.y)
{
      console.log("smallest is " +this.x );
   }
else
{
console.log("smallest is " +this.y );
}
}
}
//accessing of module
import {Largest,Smallest} from './Largest';
let addObject = new Largest(10, 20);
let o2=new Smallest(25,15);
addObject.larges();
o2.smalles();
```

Output

largest is 20

smallest is 15

Summary

- Classes in Angular are TypeScript constructs used to define blueprints for creating objects with specific properties and methods.
- Angular uses classes to implement various components, services, models, and other building blocks of an application.
- Components, services, and directives in Angular are implemented as classes.

- Classes provide a way to follow the principles of Object-Oriented Programming (OOP), making code more modular, reusable, and easier to maintain.
- Decorators like @Component, @Injectable, and @Directive are often used with classes to provide additional metadata and functionality.
- In Angular, objects are instances of classes that represent real-world entities or data structures.
- Each instance of a class is created using the new keyword, and it holds its own set of properties and behavior defined by the class.
- Objects can be created from various classes, including components, services, and models.
- Components in Angular are used to create UI elements, and each component is an object of its class.
- Constructors are special methods available in classes used to initialize objects when they are created.
- If you don't define any constructor in a class, TypeScript automatically generates a default constructor for you.
- The default constructor is empty and can be used to create instances of the class without any additional setup.

Keywords

class: The class keyword is used to define a new class in TypeScript.

constructor: The constructor keyword is a special method within a class that is automatically called when an object is created. It is used to initialize the object and perform setup tasks.

new: The new keyword is used to create a new instance of a class . It calls the constructor to initialize the object.

extends: The extends keyword is used to create a subclass that inherits properties and methods from a parent class.

implements: The implements keyword is used to make a class adhere to an interface.

Self Assessment

1. Which keyword is used to define a class in Angular?

A. interface
B. component
C. class
D. new

2. In Angular, the this keyword refers to:

A. The current instance of the class
B. The parent class
C. The child class
D. The parent component

3. The constructor in a class is used for:

A. Defining class properties
B. Creating a new instance of the class

C. Initializing the object and setting up dependencies
D. Performing heavy computations

4. What does the extends keyword in TypeScript do?

A. Creates a new class
B. Allows for class inheritance, where one class inherits from another
C. Extends the class to implement additional interfaces
D. Enables Angular-specific functionality

5. How do you create an object (instance) of a class in Angular?

A. Using the object keyword
B. By calling the create() method of the class
C. Using the new keyword followed by the class name and parentheses
D. There is no need to create an object in Angular.

6. What is a parameterized constructor in Angular?

A. A constructor that takes parameters in Angular components.
B. A constructor used to pass parameters between components and services.
C. A constructor that has a fixed set of predefined parameters in all Angular classes.
D. A constructor that is automatically generated and doesn't accept any parameters.

7. How is a parameterized constructor defined in an Angular component?

A. constructor() {}
B. constructor(params: any) {}
C. constructor(params: any) { this.params = params; }
D. params: any;

8. Which keyword is used to define a parameter in a constructor?

A. params
B. this
C. constructor
D. private

9. What is the benefit of using a parameterized constructor in Angular?

A. It allows for dependency injection and passing data to the component.
B. It automatically initializes all properties in the component.
C. It prevents other methods from accessing the component's properties.
D. It ensures that the component is instantiated only with specific parameters.

10. What is an interface in Angular?

A. An interface is a class that defines the structure of objects in Angular.

B. An interface is a blueprint for creating components in Angular.
C. An interface is a TypeScript construct used to define the structure of objects.
D. An interface is a special type of component used to handle user interactions.

11. What is the main purpose of using interfaces in Angular?

A. To improve the performance of the application.
B. To enforce encapsulation and data hiding in classes.
C. To facilitate communication between components and services.
D. To maintain consistency and type safety for object structures.

12. Can an interface have methods in Angular?

A. Yes, an interface can have methods in addition to properties.
B. No, interfaces can only define properties, not methods.
C. Yes, but methods in interfaces are optional.
D. No, interfaces are used solely for class properties.

13. How are interfaces used in Angular components?

A. Interfaces are used to define the template structure for components.
B. Interfaces are used to define the dependencies required by components.
C. Interfaces are used to define the properties and methods of components.
D. Interfaces are used to enforce component lifecycle hooks.

14. Which keyword is used to implement an interface in an Angular class?

A. extends
B. implements
C. interfaces
D. implement

15. Which access specifier is commonly used for class members that need to be accessed from the template in Angular components?

A. public
B. private
C. protected
D. readonly

Answers for Self Assessment

1.	C	2.	A	3.	C	4.	B	5.	C
6.	A	7.	C	8.	A	9.	A	10.	C
11.	D	12.	A	13.	C	14.	B	15.	A

Review Questions

1. Explain the concept of interfaces in Angular. How are interfaces used to enhance the structure, type safety, and maintainability of Angular applications?
2. Explain the concept of constructors in Angular in detail. What is the purpose of a constructor, and how is it different from other class methods?
3. Explain the different types of questions by giving the example of each type.
4. Explain the concept of access modifiers in Angular in detail.
5. Discuss the significance of public, private, and protected access modifiers and provide practical examples of their usage in Angular.
6. Explain the concept of properties in Angular in detail. What are properties, and how are they used to store and manage data in Angular components, services, and models?

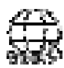
Further Readings

Wilken, J. (2018). Angular in action. Simon and Schuster.

Deeleman, P. (2016). Learning angular 2 (pp. 39-40). Packt Publishing.

Green, B., & Seshadri, S. (2013). AngularJS. " O'Reilly Media, Inc.".

Nayrolles, M., Gunasundaram, R., & Rao, S. (2017). Expert Angular. Packt Publishing Ltd.



Web Links

https://codecraft.tv/courses/angular/es6-typescript/classinterface/

https://ng2.codecraft.tv/es6-typescript/classinterface/

https://angular.io/guide/class-binding

https://angular-book.dev/ch03-04-classes.html

Unit 05: Angular Fundamentals

CONTENTS

Objectives
Introduction
5.1 Building Blocks of Angular Apps
5.2 Components
5.3 Generating Components Using Angular CLI
5.4 Templates Directives
5.5 Angular Services
5.6 Dependency Injection
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After this unit student will be able to:

- Know the different building blocks and their functionality.
- Learn the usage of template directives.
- Understand the various angular services.
- Identify the need of dependency injection.

Introduction

Angular is a powerful open-source web application framework maintained by Google and a community of developers. It is built on top of TypeScript, a superset of JavaScript, and is designed to simplify and accelerate the development of modern single-page applications (SPAs). With its robust features and comprehensive toolset, Angular has become one of the most popular frameworks for building dynamic and interactive web applications. Angular's strong architecture, efficient data handling, and excellent performance make it a popular choice for building scalable and maintainable web applications. Its extensive ecosystem, active community, and continuous development ensure that Angular remains a cutting-edge framework for modern web development.

5.1 Building Blocks of Angular Apps

Angular applications are built using various building blocks that work together to create a robust and structured web application. These building blocks include:

Modules: Angular applications are organized into modules, which are containers for related components, directives, services, and other code. Each module typically represents a feature or a

cohesive set of functionalities within the application. Angular apps have at least one root module (usually named AppModule), and they may have multiple feature modules.

Components: Components are the basic building blocks of the user interface in Angular. They represent specific parts of the application's UI and encapsulate their own logic, templates, and styles. Components are reusable and can be composed together to build complex user interfaces.

Templates: Templates define the structure of the user interface for a component. They are written in HTML and can include Angular-specific syntax and directives. Templates bind to component properties and display dynamic data.

Directives: Directives extend the behavior of HTML elements. They can be structural (changing the DOM layout) or attribute-based (changing the behavior/appearance of elements). Angular comes with built-in directives, and you can create custom directives to suit your application's needs.

Services: Services are used to share data, perform business logic, and manage communication with external APIs. They are injectable across different components and modules, allowing them to be shared and reused throughout the application.

Dependency Injection: Angular relies heavily on dependency injection to provide components with the services they need. This pattern promotes loose coupling and makes it easy to manage and test the application's components.

Routing: Angular provides a powerful routing system that enables navigation within the application. The router allows you to map URLs to specific components, load components lazily, and pass data between different routes.

Forms: Angular offers support for building complex forms, including template-driven forms and reactive forms. These forms handle user input validation, form submission, and data binding.

Pipes: Pipes are used to transform data in templates. They allow you to format, filter, and manipulate data before displaying it in the view.

HTTP Client: The HTTP client module in Angular facilitates communication with backend APIs and services. It enables making HTTP requests and processing responses.

Observables: Angular leverages RxJS to work with asynchronous data streams. Observables are used extensively, especially when working with HTTP requests and reactive forms.

Interceptors: HTTP interceptors allow you to intercept HTTP requests and responses globally. They can be used for tasks like adding authentication headers or handling errors centrally.

By understanding and effectively using these building blocks, developers can create scalable, maintainable, and feature-rich applications using the Angular framework.

5.2 Components

In AngularJS, a Component is a special kind of directive that uses a simpler configuration which is suitable for a component-based application structure. This makes it easier to write an app in a way that's like using Web Components or using the new Angular's style of application architecture.

Advantages of Components:

- Simpler configuration than plain directives.
- Promote sane defaults and best practices.
- Optimized for component-based architecture.
- Writing component directives will make it easier to upgrade to angular.

When not to use Components:

- For directives that need to perform actions in compile and pre-link functions because they aren't available.
- When you need advanced directive definition options like priority, terminal, multi-element.

- When you want a directive that is triggered by an attribute or css class, rather than an element.

In AngularJS (version 1.x), the main components that form the foundation of the application are:

Modules: AngularJS applications are organized into modules, which are containers for related components, services, filters, directives, etc. Modules help in managing the application's dependencies and provide a way to encapsulate different parts of the app.

Controllers: Controllers are responsible for defining the behavior and business logic of a particular view. They act as a bridge between the model and the view, handling user interactions and updating the model as needed.

Views/Templates: Views or templates define the presentation and structure of the user interface. They are written in HTML with AngularJS-specific directives and expressions to bind data from the model to the view.

Directives: Directives are a core feature of AngularJS and allow you to extend HTML with custom behaviors. They can be used to create reusable UI components or add behavior to existing elements.

Services: Services in AngularJS are singletons that are used to share data, business logic, and communication with external resources across different parts of the application. Services are often injected into controllers or other services to be used.

Filters: Filters allow you to format or filter data before displaying it in the view. They can be applied to expressions in templates to modify how data is presented.

Routing: While not built into the core of AngularJS, routing is commonly achieved using external libraries like ngRoute or ui-router. Routing enables navigation between different views and is essential for building single-page applications (SPAs).

Scopes: Scopes are objects that represent the model of an AngularJS application. They act as a glue between the view and the controller, providing data binding and change detection.

Dependency Injection: AngularJS uses dependency injection to provide components with their dependencies, such as services, filters, and other components. This promotes modularity and testability.

Forms: AngularJS provides features to work with HTML forms efficiently. It includes form validation, two-way data binding, and handling form submission.

$http Service: The $http service is used for making AJAX requests to the server to fetch or submit data. It is the primary way to interact with remote APIs.

$routeProvider and $stateProvider: When using routing, these services ($routeProvider for ngRoute and $stateProvider for ui-router) are used to configure the application's routes, specifying which templates and controllers to use for each route.

These components work together to create a dynamic, data-driven, and interactive web application in AngularJS. However, it's important to note that AngularJS is an older version of the Angular framework, and it's recommended to use the latest version of Angular (Angular 2+) for new projects, as it provides more powerful features and better performance.

5.3 Generating Components Using Angular CLI

In Angular, you can use the Angular CLI (Command Line Interface) to generate new components easily. The Angular CLI provides a set of commands to scaffold various parts of your Angular application, including components, services, modules, and more.

To generate a new component using the Angular CLI, follow these steps:

Install the Angular CLI: If you haven't installed the Angular CLI yet, you can do so globally using npm by running the following command in your terminal:

npm install -g @angular/cli

Create a New Angular Project: If you don't have an existing Angular project, you can create one using the Angular CLI with the following command:

ng new my-angular-app

Replace my-angular-app with your desired project name.

Navigate to the Project Directory: Change into the project directory that was just created:

cd my-angular-app

Generate a New Component: Now, you can use the generate component (or its shorthand g c) command to create a new component. The syntax for generating a component is as follows:

ng generate component component-name

Or use the shorthand:

ng g c component-name

Replace component-name with the desired name for your new component. The component name should be in kebab-case, e.g., my-component.

Using CLI with Angular Ivy (Optional): By default, the Angular CLI uses the Ivy rendering engine to build your app. If you are using Ivy, your components will be generated with the new templateUrl and styleUrl syntax. If you want to generate components using the older template and styles syntax, you can add the --view-encapsulation flag to the generate command:

After running the generate component command, the Angular CLI will create a new folder with the specified component name inside the src/app directory. It will generate the necessary files for the component, including the component class, template, and styles. Now, you have a new component ready to use in your Angular application. You can include it in other components or templates, and it's fully integrated into your project's structure.

5.4 Templates Directives

Directives in Angular are described as classes that can change or add additional behaviour to the template's elements. In Angular, directives are used to manipulate the Document Object Model (DOM), whether by adding or removing items or by altering the appearance of existing ones. Template directives are an essential part of extending HTML with custom behaviors and logic. Directives are used to manipulate the DOM, attach event listeners, create reusable components, and more. They allow you to extend HTML with new syntax and functionalities specific to your application.

AngularJS provides various built-in directives, and you can also create your own custom directives. Here are some commonly used template directives in AngularJS:

ngApp: This directive is used to define the root element of the AngularJS application. It initializes the AngularJS application and identifies the scope for the application.

```
<div ng-app="myApp">
  <!-- Your application content here -->
</div>
```

ngController: The ngController directive associates a controller with a portion of the DOM. It specifies which controller will handle the logic and data for a specific portion of the view.

```
<div ng-controller="MyController">
  <!-- Your view content here -->
</div>
```

ngModel: The ngModel directive is used for two-way data binding between the model and the view. It keeps the data in the view and the controller in sync.

```
<input type="text" ng-model="username">
```

ngClick: This directive allows you to define an expression that will be executed when an element is clicked.

```
<button ng-click="doSomething()">Click Me</button>
```

ngIf: The ngIf directive conditionally renders elements based on a given expression.

```
<div ng-if="showMessage">This message will be shown if 'showMessage' is truthy</div>
```

ngRepeat: The ngRepeat directive is used for rendering a collection of items in the DOM.

```
<ul>
  <li ng-repeat="item in items">{{item.name}}</li>
</ul>
```

ngStyle: The ngStyle directive allows you to dynamically apply CSS styles to an element based on expressions in your scope.

```
<div ng-style="{ 'color': textColor, 'font-size': fontSize }">Dynamic Styling</div>
```

ngClass: The ngClass directive is used to apply CSS classes to an element based on conditions in the scope.

```
<div ng-class="{ 'bold': isBold, 'italic': isItalic }">Styled Text</div>
```

These are some of the commonly used template directives in AngularJS. AngularJS directives give you the power to create dynamic and interactive templates by enhancing the standard HTML syntax with custom behaviors and logic.

Example

app.component.html

```
<h1 appCustomDirective>
  Template Directive Example
</h1>
```

app.component

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'p1';
}
```

custom.directive.ts

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

@Directive({
  selector: '[appCustomDirective]' // Selector to use the directive as an attribute
})
export class CustomDirective {
  constructor(private elementRef: ElementRef) {
    // Access the DOM element that the directive is applied to (the host element)
    this.elementRef.nativeElement.style.backgroundColor = 'Green';
```

```
    this.elementRef.nativeElement.style.fontSize = '24px';
    this.elementRef.nativeElement.style.padding = '10px';
  }
}
```

Output

localhost:4200

Template Directive Example

5.5 Angular Services

In Angular, services are a key building block used for sharing data, business logic, and functionalities across different parts of an application. Services are a way to separate concerns and promote reusability, maintainability, and testability in your codebase.

Here are the key aspects of Angular services:

Creating a Service: To create a service, you can use the @Injectable decorator along with the Injectable class from the @angular/core module. The decorator marks the class as a service, and it can then be injected into other components, directives, or services.

```
import { Injectable } from '@angular/core';
@Injectable()
export class MyService {
  // Service logic and data go here
}
```

Injecting a Service: You can inject a service into a component, directive, or another service by declaring it as a constructor parameter. Angular's dependency injection system will automatically provide an instance of the service when the component or service is created.

```
import { Component } from '@angular/core';
import { MyService } from './my.service';
@Component({
  selector: 'app-my-component',
  template: '...',
})
export class MyComponent {
  constructor(private myService: MyService) {
    // Use the myService instance in the component
  }
}
```

Singleton Behavior: Angular services are singleton objects by default. This means that there will be only one instance of the service created and shared across the entire application. Whenever a component or other part of the app requests the service, they will receive the same instance.

Providing a Service: When you declare a service in the providers array of a module, you are providing that service at the module level. This makes the service available throughout the module and its components.

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

```
import { MyService } from './my.service';
@NgModule({
  providers: [MyService],
})
export class MyModule {
  // Module configuration
}
```

Hierarchical Injectors: Angular services follow a hierarchical injection pattern. If a service is provided at the module level, it will be available to all components within that module. However, if a component declares the same service in its providers array, it will have its own instance of the service, creating a new child injector.

Async Services:

Services can handle asynchronous operations such as making HTTP requests, working with timers, or using observables.

Example, a service might fetch data from a remote API and provide it to the components.

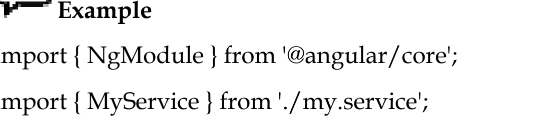
Testing Services: Since services are separate from components, they are easy to test in isolation. You can write unit tests for services to verify their functionality without involving the entire application.

Angular services play a crucial role in keeping the application lean, modular, and maintainable. They promote the DRY (Don't Repeat Yourself) principle by enabling code reuse and providing a clean separation of concerns.

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
<div ng-app="myApp" ng-controller="myCtrl">
<p>The url of this page is:</p>
<h3>{{myUrl}}</h3>
</div>
<p>This example uses the built-in $location service to get the absolute url of the page.</p>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope, $location) {
   $scope.myUrl = $location.absUrl();
});
</script>
</body>
</html>
```

Output



5.6 Dependency Injection

Dependency Injection (DI) is a fundamental concept in the Angular framework that enables the efficient management and sharing of dependencies among different parts of an application. DI allows you to create loosely coupled components, services, and other objects, making your code more modular, maintainable, and testable.

Here's how dependency injection works in Angular:

Provider Registration: In Angular, dependencies are registered with the Dependency Injection (DI) container, which maintains a registry of various services, components, and other objects used in the application. The registration is done using providers. A provider is a recipe for creating a particular dependency.

For example, you register a service as a provider in the module's providers array:

Example

```
import { NgModule } from '@angular/core';
import { MyService } from './my.service';
@NgModule({
  providers: [MyService], // Register MyService as a provider
})
export class MyModule { }
```

Dependency Resolution: When a component, directive, or service requires a dependency, it defines the required dependencies as constructor parameters. The Angular DI system takes care of resolving and providing these dependencies when an instance of the class is created.

For example, in a component, you can inject a service as follows:

Example

```
import { Component } from '@angular/core';
import { MyService } from './my.service';
@Component({
  selector: 'app-my-component',
  template: '...',
})
export class MyComponent {
  constructor(private myService: MyService) {
    // Use the myService instance in the component
  }
}
```

Hierarchical Dependency Injection: Angular follows a hierarchical injection pattern. When a component requests a dependency, the DI system first looks for the dependency in the component's injector. If the dependency is not found there, it continues searching up the injector hierarchy until it reaches the root injector, which is usually at the application level. This hierarchical resolution ensures that dependencies can be provided at different levels in the application and shared accordingly.

Singleton Behavior:

By default, Angular services are singletons, meaning that there is only one instance of the service created and shared throughout the application. Once a service is instantiated, subsequent requests for that service will receive the same instance.

Providing Services: Services can be provided at different levels in the application. You can provide a service at the module level, component level, or even within a specific provider hierarchy.

Providing at the module level: The service is available throughout the entire module and its components.

Providing at the component level: The service is available only within the component and its children, creating a separate instance for each component instance.

Providing at the component hierarchy: The service is shared among a group of components and their children.

Testing with Dependency Injection: Because of the DI system, Angular components and services can be easily tested in isolation by providing mock or fake dependencies during unit tests. This allows for effective unit testing without relying on the entire application environment.

Dependency Injection is a core design pattern in Angular that promotes maintainability, reusability, and testability. By leveraging DI, Angular encourages the use of small, single-purpose services and components, leading to more modular and robust applications.

Summary

- Template directives in Angular are powerful tools that extend the functionality of HTML elements and attributes.
- Template directives allow you to add custom behaviors and logic to your application's templates, making them dynamic and interactive.
- There are three types of directives in Angular: structural directives, attribute directives, and component directives.
- Structural directives alter the layout and structure of the DOM by adding or removing elements based on conditions.
- Common structural directives include ngIf, ngFor, and ngSwitch.
- Dependency Injection (DI) is a design pattern used in Angular to manage and provide dependencies to different parts of an application.
- DI is a software design pattern that promotes loose coupling and separation of concerns in an application.
- Angular maintains a DI container, which is a registry of various services and dependencies used in the application.

Keywords

Modules: Organize related components, directives, services, and other code into cohesive units.

Components: Basic building blocks that represent specific parts of the user interface.

Templates: Define the structure of the user interface using HTML and Angular-specific syntax.

Directives: Extend HTML with custom behaviors and manipulate the DOM.

Services: Share data, business logic, and communication with external APIs across components.

Dependency Injection: Mechanism for providing dependencies to components and services.

Routing: Enable navigation within the application and map URLs to components.

Forms: Handle user input, validation, and form submission.

HTTP Client: Communicate with backend APIs and services using HTTP requests.

Observables: Handle asynchronous data streams, especially in the context of HTTP requests and reactive forms.

Self Assessment

1. Which building block in Angular is responsible for defining the structure and presentation of specific parts of the user interface?

A. Modules
B. Components
C. Services
D. Directives

2. Which Angular building block is used to extend HTML with custom behaviors and manipulate the DOM?

A. Modules
B. Components
C. Services
D. Directives

3. In Angular, which building block is used for sharing data, business logic, and communication with external APIs across different parts of the application?

A. Modules
B. Components
C. Services
D. Directives

4. What is the role of the Angular Router building block?

A. Handle user input and form submission
B. Organize related components, directives, and services
C. Communicate with backend APIs using HTTP requests
D. Enable navigation within the application and map URLs to components

5. Which building block in Angular is used to handle asynchronous data streams and is commonly used when working with HTTP requests and reactive forms?

A. Modules
B. Components
C. Services
D. Observables

6. Template directives in Angular are used to:

A. Manipulate the DOM and add custom behaviors to HTML elements

B. Define the structure of the user interface using HTML and CSS
C. Manage HTTP requests and handle asynchronous data streams
D. Create reusable services and components for the application

7. Which type of template directive is responsible for altering the layout and structure of the DOM based on conditions?
A. Structural directives
B. Attribute directives
C. Component directives
D. Custom directives

8. The ngIf directive is an example of which type of template directive in Angular?
A. Structural directive
B. Attribute directive
C. Component directive
D. Custom directive

9. What does the ngStyle directive do in Angular?
A. Applies dynamic styles to an element based on expressions in the scope
B. Iterates over a collection and generates elements for each item
C. Conditionally adds or removes elements from the DOM
D. Extends HTML with custom behaviors and logic

10. How are attribute directives applied to HTML elements in Angular templates?
A. As custom HTML elements
B. As HTML attributes
C. As components with their own selectors
D. As special HTML tags

11. What is Dependency Injection (DI) in Angular?
A. A technique to inject CSS styles into Angular components.
B. A mechanism to resolve circular dependencies in Angular modules.
C. A design pattern used to manage and provide dependencies to different parts of an application.
D. A way to inject JavaScript libraries into Angular applications.

12. In Angular, what is the role of the Dependency Injection (DI) container?
A. It stores all the CSS styles used in the application.
B. It maintains a registry of services and components used in the application.
C. It handles routing and navigation in the application.
D. It provides a centralized database for storing application data.

13. How does Angular resolve dependencies when using Dependency Injection (DI)?
A. By automatically providing all dependencies to each component in the application.
B. By using circular references to inject dependencies into each other.

C. By using a hierarchical injection pattern, searching for dependencies in parent injectors if not found in child injectors.
D. By storing all dependencies in global variables accessible by all components.

14. In Angular, where are dependencies typically injected?
A. As URL parameters in route definitions.
B. As query parameters in HTTP requests.
C. As constructor parameters in components and services.
D. As inline scripts within HTML templates.

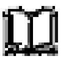
15. What is the benefit of using Dependency Injection (DI) in Angular?
A. It reduces the need for writing unit tests in the application.
B. It allows developers to avoid using services and keep components more self-contained.
C. It promotes code reusability and modularity by creating loosely coupled components and services.
D. It improves the performance of the application by reducing the number of HTTP requests.

Answers for Self Assessment

1.	B	2.	D	3.	C	4.	D	5.	D
6.	A	7.	A	8.	A	9.	A	10.	B
11.	C	12.	B	13.	C	14.	C	15.	C

Review Questions

1. Explain the concept of Modules in Angular and how they contribute to building Angular applications.
2. Discuss the significance of Services in Angular and their role in sharing data and functionality across components.
3. Explain the concept of Template Directives in Angular and how they extend the behavior and capabilities of HTML.
4. Discuss the difference between Structural Directives and Attribute Directives in Angular, providing examples of each.
5. Explain the concept of Dependency Injection (DI) in Angular and its significance in building scalable and maintainable applications.
6. Discuss the Benefits of using Dependency Injection in Angular.
7. Discuss the hierarchical injection pattern in Angular's Dependency Injection system and how it affects the resolution of dependencies in a multi-level component hierarchy.



Further Readings

Jonna, S., & Varaksin, O. (2017). *Angular UI Development with PrimeNG*. Packt Publishing Ltd.

Padmanabhan, P. (2018). *Java EE 8 and Angular: A practical guide to building modern single-*

page applications with Angular and Java EE. Packt Publishing Ltd.

Uluca, D. (2018). *Angular 6 for Enterprise-Ready Web Applications: Deliver production-ready and cloud-scale Angular web apps*. Packt Publishing Ltd.

Moreno, Z. (2015). *AngularJS Deployment Essentials*. Packt Publishing Ltd.

Panda, S. (2014). *AngularJS: Novice to Ninja: Elegant, Powerful, Testable, Extendable*. SitePoint Pty Ltd.

Web Links

https://angular.io/docs

https://angular.io/tutorial

https://www.udemy.com/topic/angular/

https://angular.io/guide/dependency-injection-overview

https://angular.io/guide/structural-directives

Unit 06: Displaying Data and Handling Events

CONTENTS

Objectives
Introduction
6.1 Property Binding
6.2 Attribute Binding
6.3 Style Binding
6.4 Template Variables
6.5 Two-Way Binding
6.6 Pipes
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After this unit you will be able to:

- Know the basic concepts and implementation of of property and angular binding.
- Understanding of template variables.
- Implementation of style binding and know the basics of pipes.

Introduction

Displaying data and handling events are fundamental concepts in web development and programming in general. Displaying data involves showing information on a web page or user interface. In the context of web development frameworks like Angular, React, and Vue.js, data is often displayed dynamically by binding it to the user interface elements. This ensures that any changes to the data are automatically reflected in the UI. Angular uses data binding techniques to interpolate values ({{ value }}), property bind using square brackets ([property]="value"), and apply directives like ngFor to iterate through arrays and display data.

Events are interactions initiated by the user, like clicking a button, submitting a form, or pressing a key. Handling events involves writing code to respond to these interactions. Events can trigger actions, update data, and manipulate the user interface. Angular components can bind to events using event binding syntax ((event)="handlerFunction()"). Event handlers in Angular are methods defined in the component class.

6.1 Property Binding

Understanding the distinction between an HTML attribute and a Document Object Model (DOM) is necessary before we can comprehend property binding. The DOM properties correspond to the current values of the attributes, which are constants defined in HTML and used to initialise them. HTML properties don't change. To give a specific DOM property a value, utilise property binding.

Properties are immediately encased in square brackets and given a value in single quotes when utilising property binding. The component class typically defines these values. The binding of properties is one-way. This suggests that data only moves from the component to the template in a single path. The view can be updated by changing the component's defined values.

Property binding is a concept commonly associated with Angular, the newer framework. However, I'll provide information about both.

AngularJS (Angular 1.x): AngularJS is an older JavaScript framework that introduced the concept of data binding to web applications. In AngularJS, property binding is achieved using double curly braces {{ expression }} or the ng-bind directive.

Using double curly braces:

```
<div>{{ someVariable }}</div>
```

Using the ng-bind directive:

```
<div ng-bind="someVariable"></div>
```

In both cases, someVariable is a property from your controller's scope, and its value will be automatically updated in the DOM whenever the value in the scope changes.

Angular (Angular 2+): In modern Angular (not AngularJS), property binding is a fundamental concept used to pass data from the component class to the template (DOM). It uses square brackets [] around the property name in the template.

Example, if you have a component property named message:

```
@Component({
  selector: 'app-example',
  template: '<div>{{ message }}</div>', // Interpolation
})
export class ExampleComponent {
  message = 'Hello, Angular!';
}
```

You can use property binding to achieve the same result:

```
<div [textContent]="message"></div>
```

In this case, the message property value from the component class is bound to the textContent property of the div element.

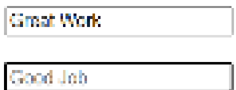
Example

```
//Setting Properties of an HTML Tag
//app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
 export class AppComponent {
  title = 'Great Work';
```

```
  WriteHere = 'Good Job';
}
//app.component.html
<h1>Implementation of property binding</h1>
<!-- Setting attributes of input using property binding -->
<input [value]='title'>
<br />
<br />
<input [placeholder]='WriteHere'>
<br />
<br />
```

Output

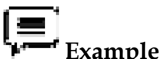
Implementation of property binding



6.2 Attribute Binding

Attribute binding in Angular is a technique that allows you to dynamically set HTML attributes on elements based on component data. It's a way to manipulate the attributes of HTML elements by binding them to values from your component's properties. Attribute binding is denoted by square brackets [] in Angular templates. Here's a simple explanation of how attribute binding works in Angular:

Binding to HTML Attributes: You can bind an attribute of an HTML element to a property in your component. This is useful when you want to update attributes based on dynamic data. For example, you might want to change the href attribute of a link based on a URL from your component.

 **Example**

```
<a [href]="dynamicUrl">Click me</a>
```

In this example, dynamicUrl is a property in your component class, and its value will be used as the href attribute for the <a> element.

Conditional Attribute Binding: Attribute binding is commonly used to conditionally apply attributes based on certain conditions. For instance, you can apply a CSS class conditionally to an element.

```
<div [class.highlight]="isActive">This div is highlighted</div>
```

Here, the highlight class will be added to the <div> element only when the isActive property in your component is true.

Binding to Non-Standard Attributes: Sometimes, you might want to bind to attributes that are not standard HTML attributes. For example, if you're working with custom directives, you can use attribute binding to set their values.

Example

```
<custom-directive [customAttribute]="value"></custom-directive>
```

In this case, customAttribute is a property that's expected by the custom-directive, and its value will be dynamically set based on your component's data.

Attribute binding is a powerful way to dynamically control the behavior and appearance of your HTML elements based on the state of your Angular components. It promotes flexibility and reusability in your templates by allowing you to easily manipulate attributes without resorting to manual JavaScript manipulation. You can directly change the value of an HTML Element Attribute using Angular's Attribute Binding feature. In order to dynamically link an element's attribute with a component's properties, attribute binding is employed.

Example

// Implementation of attribute binding

app.component.html

```
<table>
 <thead>
   <tr>
     <th colspan="2">
       {{pageHeader}}
     </th>
   </tr>
 </thead>
 <tbody>
   <tr>
     <td>Name</td>
     <td>{{Name}}</td>
   </tr>
      <tr>
     <td>Roll</td>
     <td>{{Roll}}</td>
   </tr>
        <tr>
     <td>Age</td>
     <td>{{Age}}</td>
   </tr>
 </tbody>
</table>
```

app.component.ts

```
import { Component } from '@angular/core';
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
```

```
    pageHeader: string = 'Details';
    Name: string = 'Seerat';
    Roll: number = 18;
    Age: number = 22;
}
```

Output

Details

Name Seerat

Roll 18

Age 22

6.3 Style Binding

Using Angular style binding, it is simple to apply CSS styles to HTML components. A view element's style is set using style binding. Angular's style binding allows us to set an HTML element's inline styles. A conditionally added style can also be used to create a dynamically styled element.

Example

```
<h3 [style.color] = "'red'"
            [style.text-align] = "'center'"
 [style.font-size.px]="'32'" >
Implementation of style binding
</h3>
```

Output

Implementation of style binding

6.4 Template Variables

In AngularJS, you can use template variables to access and manipulate data within templates and expressions. These template variables are typically created using the ngModel directive or other directives like ngRepeat, ngForm, ngController, etc. Here's how you can use template variables in AngularJS:

ngModel Directive: You can use ngModel to create a template variable that references the current input element's value. This is commonly used for forms and input elements.

```
<input type="text" ng-model="username" placeholder="Enter your username">
<p>Your username is: {{ username }}</p>
```

In this example, the ng-model directive creates a template variable called username, which is then displayed using double curly braces {{ username }}.

ngRepeat Directive: ngRepeat can be used to iterate over a collection and create template variables for each item in the collection.

Example

```
<ul>
  <li ng-repeat="item in items">{{ item }}</li>
</ul>
```

Here, ng-repeat creates a template variable called item for each element in the items array.

ngForm Directive: When using forms, you can create a form-level template variable using ngForm.

Example

```
<form name="myForm">
  <input type="text" ng-model="user.name" required>
  <span ng-show="myForm.$invalid">Form is invalid!</span>
</form>
```

In this case, ngForm creates a template variable named myForm, which can be used to access form validation information.

ngController Directive: If you're using controllers, you can create a controller-level template variable using ngController.

```
<div ng-controller="MyController as myCtrl">
  <p>{{ myCtrl.message }}</p>
</div>
```

Here, ngController creates a template variable called myCtrl, which allows you to access properties and methods defined on the controller.

Example

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
</head>
<body ng-controller="GreetingController">
   <label for="name">Enter your name:</label>
   <input type="text" id="name" ng-model="user.name">

   <p ng-if="user.name">Hello, {{ user.name }}!</p>
</body>
<script>
   var app = angular.module('myApp', []);
   app.controller('GreetingController', function($scope) {
      $scope.user = {}; // Initialize an empty object to hold user data
   });
```

```
</script>
</html>
```

Output

Template variables are a powerful feature in AngularJS that allows you to work with data and elements in your templates effectively. You can use them to bind data, perform validation, and control the flow of your application within the template.

6.5 Two-Way Binding

two-way data binding is a powerful feature that allows you to automatically keep the model and view in sync. This means that when you change the data in the model (JavaScript code), the corresponding view (HTML) updates automatically, and when you make changes in the view, the model updates as well. Two-way data binding simplifies the development of interactive and responsive web applications.

Here's how two-way data binding works in AngularJS:

Using ng-model: The ng-model directive is used to establish two-way data binding between an HTML input element and a model property.

```
<input type="text" ng-model="myModel">
<p>{{ myModel }}</p>
```

In this example, any changes made in the input field will be automatically reflected in the paragraph (<p>) below, and vice versa.

Controller Initialization: In your AngularJS controller, you initialize the model property. Any changes to this property will be automatically reflected in the view, and any changes in the view will update the model.

```
app.controller('MyController', function($scope) {
  $scope.myModel = 'Initial value';
});
```

Two-way Data Binding Flow: When the page loads, AngularJS initializes the view with the initial value from the model. As the user interacts with the input field, AngularJS updates the model ($scope.myModel) whenever the input value changes.

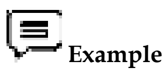
Any updates to $scope.myModel in the controller will immediately propagate to the view.

Binding to Other Elements: Two-way data binding isn't limited to input fields. You can use it with other form elements like checkboxes, radio buttons, and text areas, as well as with custom directives.

```
<input type="checkbox" ng-model="isChecked">
<p>{{ isChecked ? 'Checked' : 'Not Checked' }}</p>
```

In this example, the checkbox state (isChecked) is bound to the paragraph, and any changes to the checkbox will update the paragraph text.

Two-way data binding in AngularJS helps you build interactive and dynamic web applications with ease. It simplifies the synchronization between the model and the view, reducing the need for manual DOM manipulation and event handling.



Example

```
<!DOCTYPE html>
```

```
<html ng-app="myApp">
<head>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
</head>
<body ng-controller="TwoWayBindingController">
  <label for="name">Enter your name:</label>
  <input type="text" id="name" ng-model="user.name">

  <p>{{ user.name ? 'Welcome, ' + user.name + '!' : 'Please enter name.' }}</p>
</body>
<script>
  var app = angular.module('myApp', []);

  app.controller('TwoWayBindingController', function($scope) {
    $scope.user = {}; // Initialize an empty object to hold user data
  });
</script>
</html>
```

Output

Enter your name: Happy

Welcome, Happy!

6.6 Pipes

pipes were not a built-in feature like they are in Angular (Angular 2 and later). Instead, AngularJS had a concept known as "filters" that served a similar purpose. Filters were used to format and transform data in templates, allowing you to display data in a more user-friendly way.

Here's how you could use filters in AngularJS:

Built-in Filters: AngularJS provided several built-in filters that you could use in your templates. Some common built-in filters included:

{{ expression | filter:arguments }}: This syntax allowed you to apply a filter to an expression in your template.

Example:

{{ dateValue | date:'yyyy-MM-dd' }}

Here, the date filter formats the dateValue variable as per the specified format.

{{ text | uppercase }} and {{ text | lowercase }}: These filters converted the text to uppercase or lowercase, respectively.

Custom Filters: You could also create custom filters to apply your own data transformations. Custom filters were defined using the filter method of your AngularJS module.

```
angular.module('myApp', [])
  .filter('customFilter', function () {
```

```
    return function (input) {
        // Perform custom transformation on input
        return transformedValue;
    };
});
```

In your template:

{{ inputValue | customFilter }}

You would replace customFilter with the name of your custom filter, and it would apply the transformation to inputValue.

Chaining Filters: You could also chain multiple filters together to achieve complex transformations.

Example:

{{ text | uppercase | limitTo:10 }}

In this case, the text would first be converted to uppercase and then limited to the first 10 characters.

Please note that AngularJS (version 1.x) is now considered a legacy framework, and Angular (versions 2 and later) has a different approach to data formatting and manipulation using pipes. If you are working with Angular 2+ or later versions, you should use pipes instead of filters.

Example

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
</head>
<body ng-controller="FilterExampleController">
  <h2>Filter Example</h2>

  <label for="text">Enter some text:</label>
  <input type="text" id="text" ng-model="inputText">

  <p>Filtered Text: {{ inputText | customFilter }}</p>
</body>
<script>
  var app = angular.module('myApp', []);

  app.controller('FilterExampleController', function($scope) {
    $scope.inputText = ''; // Initialize an empty input field
  });
  app.filter('customFilter', function() {
```

```
        return function(input) {
            // Custom filter: Convert input text to uppercase
            return input ? input.toUpperCase() : '';
        };
    });
</script>
</html>
```

Output

Filter Example

Summary

- Property binding is a technique for dynamically setting the properties of HTML elements based on component data or expressions.
- Property binding primarily deals with HTML element properties, representing the current state of the element in the DOM.
- Property binding is used when you want to modify an element's properties, such as value, innerHTML, or checked, based on component data or conditions.
- One-Way Binding is one-way data flow from the component to the DOM attribute. Changes in the component update the attribute, but changes in the DOM do not affect the component data.
- Attribute binding is a mechanism for setting HTML attributes dynamically based on component data or expressions.
- Attribute binding primarily deals with HTML attributes, not properties. Attributes are the initial values specified in the HTML markup.
- Template variables are identifiers that you can assign to HTML elements or Angular components within your template. They are prefixed with a hash (#) symbol.
- To declare a template variable, you place the # symbol in front of an HTML element or Angular component, like <input #myInput>.
- Template variables give you direct access to DOM elements, allowing you to read values, set properties, or even trigger events programmatically.
- You can also use template variables to access child components or directives and interact with their public methods and properties.

Keywords

Property Binding: Property binding is a technique that allows you to set the value of an HTML element's property based on a value from your component's data or model.

Dynamic Binding: Dynamic binding refers to the ability to change the property value dynamically based on changes in the component's data or state.

One-way Data Flow: Property binding enforces a one-way data flow from the component to the DOM. This means that changes in the component's data automatically update the associated property in the DOM, but changes in the DOM do not affect the component's data.

Interpolation: Interpolation is a method of property binding in which you embed a variable or expression directly within the HTML template using double curly braces {{ }}.

Style Binding: Style binding is the process of dynamically setting CSS styles for HTML elements based on the values of component properties or expressions.

ngStyle (Angular): In Angular, ngStyle is a built-in directive that allows you to bind CSS styles to an HTML element based on conditions or component properties.

Conditional Styles: Style binding allows you to apply CSS styles conditionally based on component logic or data, making it useful for dynamic styling.

Dynamic Styling: Dynamic styling refers to the ability to change the styles of an element in real-time based on user interactions or changes in component data.

Inline Styles: Inline styles are CSS styles defined directly within an HTML element's style attribute. Style binding is often used to create inline styles.

Object Syntax: In style binding, you can use an object syntax to define the styles you want to apply, where keys are CSS properties, and values are the corresponding values or expressions.

Class Binding: Class binding is another binding technique that allows you to add or remove CSS classes from an element based on component data or conditions. It can work in conjunction with style binding to achieve complex styling.

Dynamic Styling in Components: Style binding is commonly used in components to style elements based on the component's state, making user interfaces more interactive and responsive.

Self Assessment

1. In web development, what is the primary purpose of property binding?

A. To set HTML attributes dynamically.

B. To modify HTML element properties dynamically.

C. To define custom CSS styles for elements.

D. To create two-way data binding.

2. Which front-end framework commonly uses property binding with square brackets [ ] syntax?

A. React

B. Angular

C. Vue.js

D. jQuery

3. When using property binding, which direction does the data flow?

A. From the DOM to the component.

B. Bidirectionally, both from the DOM to the component and vice versa.

C. From the component to the DOM.

D. Data doesn't flow; it remains static.

4. Which of the following is an example of property binding?

A. <img [src]="imageUrl">

B. Link

C. <input (value)="inputValue">

D. <div {background-color}="bgColor">

5. What kind of values can you bind to an HTML element property using property binding?

A. Only primitive values (e.g., strings, numbers).

B. Only JavaScript functions.

C. Any valid JavaScript expression or component property.

D. Only CSS styles.

6. In web development, what is the primary purpose of attribute binding?

A. To set HTML attributes dynamically.

B. To modify HTML element properties dynamically.

C. To define custom CSS styles for elements.

D. To create two-way data binding.

7. Which front-end framework commonly uses attribute binding with square brackets [ ] syntax?

A. React

B. Angular

C. Vue.js

D. jQuery

8. When using attribute binding, which direction does the data flow?

A. From the DOM to the component.

B. Bidirectionally, both from the DOM to the component and vice versa.

C. From the component to the DOM.

D. Data doesn't flow; it remains static.

9. Which of the following is an example of attribute binding?

A. <img [src]="imageUrl">

B. Link

C. <input (value)="inputValue">

D. <div {background-color}="bgColor">

10. What kind of values can you bind to an HTML attribute using attribute binding?

A. Only primitive values (e.g., strings, numbers).

B. Only JavaScript functions.

C. Any valid JavaScript expression or component property.

D. Only CSS styles.

11.What are template variables primarily used for in web development?

A. Defining CSS styles for HTML elements.

B. Interacting with and manipulating DOM elements or components in templates.

C. Managing API requests.

D. Creating two-way data binding.

12. How are template variables declared in HTML templates?

A. With a dollar sign ($).

B. With an exclamation mark (!).

C. With a hash symbol (#).

D. With a percentage sign (%).

13. What can you do with template variables in Angular templates?

A. Access component methods only.

B. Access component properties only.

C. Access and manipulate DOM elements or components.

D. Modify CSS styles directly.

14. In which part of the code is the scope of a template variable typically limited?

A. To the entire application.

B. To the entire component.

C. To the template where it is defined.

D. To a shared global scope.

15. What is a common use case for template variables in event handling?

A. To declare new variables within components.

B. To add new HTML elements to the DOM.

C. To directly access and manipulate DOM elements based on user interactions.

D. To create custom CSS animations.

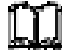
Answers for Self Assessment

1.	B	2.	B	3.	C	4.	A	5.	C
6.	A	7.	B	8.	C	9.	B	10.	C
11.	B	12.	C	13.	C	14.	C	15.	C

Review Questions

1. Describe what property binding is in Angular, its purpose, and how it differs from other forms of binding. Provide examples of property binding.
2. Explain the concept of one-way data flow in property binding. How does data flow from the component to the DOM elements?
3. Discuss the scenarios where you might use property binding to bind input element properties like value, disabled, and checked to component data.

4. Describe what template variables are in Angular and how they are declared in HTML templates.
5. Discuss the scope and lifespan of template variables. How long do they persist, and where can they be accessed within an Angular application? Provide examples to illustrate their scope.
6. Discuss how template variables can be used to access and manipulate DOM elements directly from the template.



Further Readings

Arora, C., & Hennessy, K. (2016). *Angular 2 By Example*. Packt Publishing.

Kotaru, V. K. (2020). *Angular for Material Design*. Apress.

Dayley, B., Dayley, B., & Dayley, C. (2017). *Learning Angular: A Hands-On Guide to Angular 2 and Angular 4*. Addison-Wesley Professional.

Freeman, A. (2017). *Pro Angular* (p. 778). Apress.

Kasagoni, S. K. (2017). *Building Modern Web Applications Using Angular*. Packt Publishing.

Salehi, S. (2017). *Angular Services*. Packt Publishing Ltd.

Web Links

https://angular.io/guide/property-binding

https://www.knowledgehut.com/blog/web-development/property-binding-in-angular

https://www.telerik.com/blogs/understanding-angular-property-binding-and-interpolation

https://www.pluralsight.com/guides/how-to-use-template-reference-variables-in-angular

https://itnext.io/working-with-angular-5-template-reference-variable-e5aa59fb9af

Unit 07: Directives

CONTENTS

Objectives

Introduction

7.1 Commonly used Angular Directives

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit, you will be able to

- know the commonly used angular directives
- learn the implementation of common angular directives

Introduction

AngularJS directives are a powerful feature of the AngularJS framework that allows you to extend HTML with custom functionality and create reusable components. Directives are essentially markers on a DOM element that tell AngularJS's HTML compiler to attach a specific behavior to that element or transform it in some way.

Directives in AngularJS come in three different types:

Attribute Directives: These are the most common type of directives and are used as attributes on HTML elements. They are denoted by the "ng-" prefix. Attribute directives modify the behavior or appearance of an element.

Example:

```
<div ng-app="myApp">
  <input type="text" ng-model="name" my-directive>
</div>
```

In this example, my-directive is an attribute directive that can modify the behavior of the <input> element.

Element Directives: These directives create new HTML elements or components. They are represented by custom HTML tags and can be used as standalone components.

Example:

```
html
<my-directive></my-directive>
```

Here, my-directive is an element directive that defines a custom HTML element.

Class Directives: These directives are applied to elements using CSS classes. They modify the behavior or appearance of elements based on the presence of a particular class.

Example:

```
html

<div class="my-directive">Content</div>
```

In this case, the my-directive class directive is applied to the <div> element.

Directives can be used to manipulate the DOM, handle events, create custom templates, interact with data, and much more. They provide a way to encapsulate complex functionality into reusable components, making AngularJS applications more modular and maintainable.

Creating a custom directive in AngularJS involves defining a JavaScript function that specifies the directive's behavior and registering it with AngularJS using the directive() function.

Example:

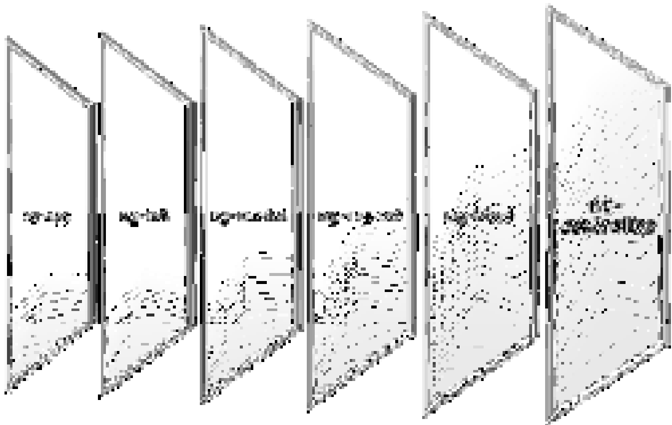
```
angular.module('myApp', [])
 .directive('myDirective', function() {
  return {
   restrict: 'A', // Restrict the directive to attribute usage
   link: function(scope, element, attrs) {
    // Directive logic and behavior
   }
  };
 });
```

In this example, we define a directive called myDirective using the directive() function. The restrict option restricts the directive to attribute usage (A). The link function contains the logic and behavior of the directive.

AngularJS directives provide a powerful way to extend HTML and build reusable components. They play a crucial role in creating dynamic and interactive web applications using the AngularJS framework.

7.1 Commonly used Angular Directives

There are several commonly used Angular directives that are frequently employed in Angular applications. Here are some of the most widely used directives:



ng-app: The ng-app directive is one of the fundamental directives in AngularJS. It is used to bootstrap an AngularJS application by defining the root of the application's scope and activating the AngularJS framework within a specific portion of the HTML document.

The ng-app directive is typically placed on the <html> or <body> tag to define the scope of the entire document. It tells AngularJS to initialize and process the AngularJS directives and expressions within that scope.

Syntax:

```
<element ng-app="modulename">
content
</element>
```

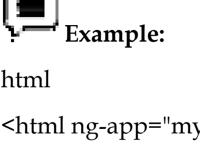
Example:

```
<!DOCTYPE html>
<html>

<head>
        <title>
                AngularJS ng-app Directive
        </title>
        <script src=
"https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js">
        </script>
</head>
<body style="text-align: center">
        <div ng-app="" ng-init="name='Implementation of ng-app'">
<b>{{ name }} </b>
        </div>
</body>
</html>
```

Output

Implementation of ng-app

The value of the ng-app directive can be an AngularJS module name, indicating that the application should use that specific module to bootstrap. The module must be defined in the JavaScript code using the angular. module function.

 **Example:**

```
html
<html ng-app="myApp">
<!-- ... -->
</html>
javascript
angular.module('myApp', []);
```

In this case, the ng-app directive specifies that the AngularJS application should use the "myApp" module to bootstrap. The corresponding module is defined in the JavaScript code using angular.module('myApp', []).

By using the ng-app directive, AngularJS takes control over the specified portion of the HTML document and processes any AngularJS-specific directives, expressions, and functionality within that scope. It allows you to create dynamic and interactive web applications using AngularJS's powerful features.

ng-init: The ng-init directive is used in AngularJS to initialize a variable or evaluate an expression in the scope of an element. It allows you to set an initial value or perform an action when the element is processed by AngularJS.

The ng-init directive is typically used on an HTML element, such as a <div> or an <input>, and it takes an expression as its value.

Syntax:

```
<element ng-init="expression" ></element>
```

Example:

```
<!DOCTYPE html>
<html>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></script>
<body>
<div ng-app="" ng-init="mydata='Implementation of ng-init '">
<h1>{{mydata}}</h1>
</div>
</body>
</html>
```

Output

Implementation of ng-init

You can also use more complex expressions and perform actions using ng-init.

Example:

```
html
<div ng-init="numbers = [1, 2, 3]; total = numbers[0] + numbers[1] + numbers[2]">
  The total is {{ total }}
</div>
```

In this case, the ng-init directive initializes the numbers array with three values and then calculates the total by summing the elements of the array. The calculated value is displayed using {{ total }}.

While ng-init can be useful for initializing simple values or performing quick calculations, it is generally recommended to avoid using it for complex logic. Instead, it's better to initialize variables and perform more complex actions in the controller or using custom directives.

Note: that starting from AngularJS version 1.4, the use of ng-init is discouraged in favor of initializing variables and expressions directly in the controller or component bindings. It helps to separate the initialization logic from the template, making the code more maintainable and easier to test.

ng-bind: The AngularJS ng-bind directive replaces the content of an HTML element with the value of a given variable, or expression. If you change the value of the given variable or expression, AngularJS changes the content of the specified HTML element as well.

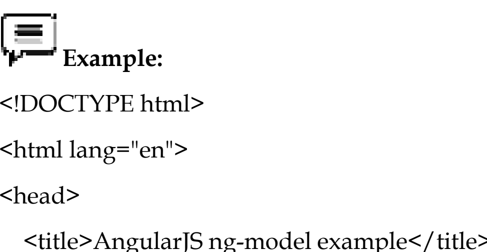
Syntax

```
<element ng-bind="expression"></element>
```

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>AngularJS ng-model example</title>
   <meta charset="UTF-8">
   <script src=
"https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.min.js">
            </script>
</head>
<body data-ng-app>
   <input type="text" data-ng-model="m1">
   <span ng-bind="m1.toLowerCase()"></span>
</body>
</html>
```

Output



ng-repeat: Ng-repeat is one of the directives in angularjs. It is used to loop the given object or array. When we need to display the set of HTML in repeated mode with some data or collection then we can use the ng-repeat directive.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>AngularJS ng-model example</title>
```

```
   <meta charset="UTF-8">
   <script src=
"https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.min.js">
         </script>
</head>
<body ng-app="myApp" ng-controller="myCtrl">
<h1 ng-repeat="i in records">{{i}}</h1>
<script>
var app = angular.module("myApp", []);
app.controller("myCtrl", function($scope) {
   $scope.records = [
      "Apple",
      "Banana",
      "Cherry",
      "grapes",
   ]
});
</script>
</body>
</html>
```

Output

Apple

Banana

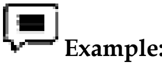
Cherry

grapes

ng-controller: The AngularJS ng-controller directive adds a controller class to the view (your application). It is the key aspect that specifies the principles behind the Model-View-Controller design pattern. It facilitates you to write code and make functions and variables, which will be parts of an object, available inside the current HTML element. This object is called scope.

Syntax

```
<element ng-controller="expression"></element>
```

 **Example:**

```
<!DOCTYPE html>
<html>
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.4.8/angular.min.js"></script>
<body>
<div ng-app="myApp" ng-controller="myCtrl">
Name: {{fName}}
```

```
Age:{{age}}
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.fName = "Harry";
  $scope.age = 66;
});
</script>
</body>
</html>
```

Output

Name: Harry Age:66

Summary

- In AngularJS, the "ng-app" directive is used to bootstrap an AngularJS application.
- When the "ng-app" directive is applied, AngularJS initializes and compiles the specified element and its children, allowing AngularJS-specific features like data binding, directives, controllers, and services to be used within that application.
- The "ng-bind" directive is used to bind a model or expression to the content of an HTML element. It ensures that the data is dynamically updated in the element whenever the model changes.
- The "ng-bind" directive in AngularJS is a convenient way to dynamically bind data to the content of HTML elements, ensuring that the element's content stays synchronized with the underlying data model.
- When using "ng-bind," if the expression or model value is undefined or null, it prevents the display of unwanted output, ensuring that an empty string is shown instead.
- The "ng-repeat" directive is used to repeat a section of HTML code for each item in a collection. It dynamically generates HTML elements based on the data in the collection.
- You can nest "ng-repeat" directives to create multiple levels of iteration, allowing you to handle complex data structures and generate hierarchical HTML structures.
- To improve performance and prevent unnecessary DOM manipulation, you can use the "track by" expression with "ng-repeat".

Keywords

Directive: Directives are defined using the @Directive decorator in Angular. The decorator allows you to specify various properties and behaviors of the directive, such as selectors, inputs, outputs, and lifecycle hooks.

Selector: The selector is used to identify the elements or components to which a directive should be applied. It can be specified as an attribute, class, element, or combination of these.

Binding: Directives can use bindings to communicate with the element/component they are applied to. Bindings allow you to pass data or trigger actions between the directive and the template.

Dependency Injection: Directives can make use of Angular's dependency injection system to obtain references to other services or dependencies they require.

Built-in Directives: Angular provides a set of built-in directives that cover common scenarios. Examples include ngIf, ngFor, ngSwitch, etc. These directives allow you to conditionally render elements, iterate over collections, and switch between different views.

Custom Directives: Angular also allows you to create custom directives tailored to your specific needs. Custom directives enable you to encapsulate complex behaviors or reusable UI components.

Directive Communication: Directives can communicate with other directives or components using inputs, outputs, or shared services. This enables building complex interactions between different parts of an Angular application.

Self Assessment

1. What is the purpose of the "ng-app" directive in AngularJS?

A. It defines a new AngularJS module.
B. It bootstraps an AngularJS application.
C. It creates a new AngularJS component.
D. It establishes two-way data binding.

2. Where is the "ng-app" directive typically applied?

A. Inside AngularJS controllers.
B. On HTML elements that need data binding.
C. On services to define dependencies.
D. At the root element of an HTML page.

3. Does the "ng-app" directive support multiple applications on a single page?

A. Yes, you can use multiple "ng-app" directives.
B. No, only one "ng-app" directive is allowed per page.
C. It depends on the version of AngularJS.
D. It requires additional configuration to support multiple applications.

4. Which of the following statements about the "ng-app" directive is true?

A. It is specific to AngularJS and is not used in Angular.
B. It is used to bind data to the content of an HTML element.
C. It allows for dynamic generation of HTML elements based on a collection.
D. It is used for two-way data binding between a model and an input field.

5. In AngularJS, what is the alternative syntax to the "ng-app" directive for binding data to HTML elements?

A. ng-bind
B. ng-model
C. ng-repeat
D. {{ double curly braces }}

6. What is the purpose of the "ng-bind" directive in AngularJS?

A. It binds a model to an input field.

B. It binds data to the content of an HTML element.
C. It creates a new AngularJS component.
D. It establishes two-way data binding.

7. How does the "ng-bind" directive differ from using double curly braces "{{ }}"?
A. "ng-bind" allows two-way data binding, while double curly braces only support one-way binding.
B. "ng-bind" is used for expressions, while double curly braces are used for simple model values.
C. There is no difference; they are two different ways to achieve the same result.
D. "ng-bind" is specific to AngularJS, while double curly braces are used in Angular.

8. Where is the "ng-bind" directive typically applied?
A. Inside AngularJS controllers.
B. On HTML elements that need data binding.
C. On services to define dependencies.
D. At the root element of an HTML page.

9. Can you use multiple instances of the "ng-bind" directive in a single element?
A. Yes, you can use multiple "ng-bind" directives within the same element.
B. No, only one "ng-bind" directive is allowed per element.
C. It depends on the version of AngularJS.
D. It requires additional configuration to support multiple "ng-bind" directives.

10. Which of the following statements about the "ng-bind" directive is true?
A. It is specific to AngularJS and is not used in Angular (Angular 2+).
B. It is used to repeat HTML code for each item in a collection.
C. It is used to bind a model to an input field.
D. It allows for dynamic generation of HTML elements based on a collection.

11. What is the purpose of the "ng-controller" directive in AngularJS?
A. It defines a new AngularJS module.
B. It binds a model to an input field.
C. It creates a new AngularJS controller.
D. It establishes two-way data binding.

12. Where is the "ng-controller" directive typically applied?
A. Inside AngularJS services.
B. On HTML elements to define a controller's scope.
C. In CSS stylesheets apply specific styles.
D. At the root element of an HTML page.

13. Can you use multiple "ng-controller" directives within a single HTML element?

A. Yes, you can use multiple "ng-controller" directives within the same element.
B. No, only one "ng-controller" directive is allowed per element.
C. It depends on the version of AngularJS.
D. It requires additional configuration to support multiple "ng-controller" directives.

14. How does the "ng-controller" directive relate to the AngularJS scope?
A. It creates a new isolated scope for the controller.
B. It automatically inherits the parent scope of the element.
C. It establishes two-way data binding between the controller and the view.
D. It defines dependencies for the controller.

15. Which of the following statements about the "ng-controller" directive is true?
A. It is specific to AngularJS and is not used in Angular (Angular 2+).
B. It is used to bind a model to an input field.
C. It is used to repeat HTML code for each item in a collection.
D. It allows for dynamic generation of HTML elements based on a collection.

Answers for Self Assessment

1.	B	2.	D	3.	B	4.	A	5.	D
6.	B	7.	B	8.	B	9.	B	10.	A
11.	C	12.	B	13.	B	14.	B	15.	A

Review Questions

1. What is the purpose of the "ng-app" directive in AngularJS and where is this directive typically applied give an example.
2. "The "ng-repeat" directive in AngularJS provides several useful features for iterating over collections." Justify the statement with the help of an example.
3. Explain the syntax and usage of the "ng-repeat" directive, and provide an example of how to use it to iterate over an array and display the array items as a list.
4. Explain the purpose and functionality of the ng-bind directive, and provide an example that demonstrates its usage in binding a model value to an element's content.
5. Explain the purpose and functionality of the ng-controller directive, and provide an example that demonstrates its usage to bind a controller to an HTML element and access controller properties and methods.
6. With the help of appropriate example explain the significance of using the ng-module directive.

Further Readings

Wilken, J. (2018). *Angular in action*. Simon and Schuster.

Arora, C., & Hennessy, K. (2016). *Angular 2 By Example*. Packt Publishing.

Dayley, B., Dayley, B., & Dayley, C. (2017). *Learning Angular: A Hands-On Guide to Angular 2*

and Angular 4. Addison-Wesley Professional.

Arora, C., & Hennessy, K. (2018). *Angular 6 by Example: Get up and running with Angular by building modern real-world web apps*. Packt Publishing.

Kaufman, N., & Templier, T. (2016). *Angular 2 Components*. Packt Publishing Ltd.

Web Links

https://angular.io/guide/built-in-directives

https://www.simplilearn.com/what-are-directives-in-angular-article

https://blog.bitsrc.io/directives-in-angular-6160ce805416

https://www.tutorialspoint.com/angular8/angular8_directives.htm

https://www.tektutorialshub.com/angular/angular-directives/

Unit 08: Building Re-usable Components

CONTENTS

Objectives

Introduction

8.1 component API

Summary

Keywords

Self Assessment

Review Questions

Answers for Self Assessment

Further Readings

Objectives

After this unit the students will:

- Know the process of creating new components.
- Understand the concept od component api.
- Implementation of code to make a component re-usable.

Introduction

Building reusable components is a fundamental concept in Angular development, and it's one of the key advantages of using a component-based framework. Reusable components allow you to create modular and maintainable code by encapsulating functionality, UI, and logic into self-contained units. Here's how you can create reusable components in Angular:

Create a New Component:

Use the Angular CLI command to generate a new component:

sh

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

Component Structure:

Inside the generated component folder (e.g., my-component), you'll find files like my-component.component.ts, my-component.component.html, my-component.component.css, and more. These files define the component's TypeScript code, template, and styling.

Inputs and Outputs:

Define @Input and @Output properties in your component to make it more flexible and interactive.

@Input: Use this decorator to allow data to flow into your component from its parent component. This allows you to pass data to your component when you use it in templates.

```
@Input() data: any;
```

@Output: Use this decorator to emit events from your component to its parent component. This enables communication from the child component to the parent component.

```
@Output() buttonClicked: EventEmitter<void> = new EventEmitter<void>();
```

Template and Styles:

Design the UI and styling of your component in its HTML and CSS files. The template file can use the data passed through @Input to render dynamic content.

Component Logic:

Implement the necessary logic in the component's TypeScript file. This includes handling user interactions, responding to events, and processing data.

Reusing the Component:

To use your reusable component, simply include it in the template of another component using its selector:

<app-my-component [data]="someData" (buttonClicked)="handleButtonClick()"></app-my-component>

Module Declaration:

Make sure to declare your reusable component in an Angular module's declarations array.

Considerations:

- Design your component to be as self-contained as possible, reducing its dependencies on external factors.
- Plan for flexibility. Make sure your component can handle various scenarios and accept different data inputs.
- Use Angular features like services for shared logic that multiple components might need.
- Testing:
- Write unit tests for your components to ensure their functionality and interactions are working correctly.

Documentation:

Provide clear documentation for your reusable components, including usage instructions, available inputs/outputs, and any specific considerations. By following these steps, you can create modular, reusable components in Angular that enhance code maintainability, readability, and scalability.

8.1 component API

In Angular, the term "Component API" refers to the public interface of an Angular component that allows interaction with the component from the outside, typically from other components or services. This API is defined by the component's properties, methods, inputs, and outputs that are designed to be used by other parts of your application. Let's delve into the different aspects of a component API:

Inputs (@Input):

Inputs allow you to pass data from a parent component to a child component. By using the @Input decorator, you define properties in the child component that can receive values from the parent component.

@Input() data: any;

The data property can then be set by the parent component in its template:

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

Outputs (@Output):

Outputs allow a child component to send events to its parent component. You define properties with the @Output decorator and an EventEmitter instance to emit events.

@Output() buttonClicked: EventEmitter<void> = new EventEmitter<void>();

In the child component, you can emit the event when a button is clicked:

```
onClick() {
  this.buttonClicked.emit();
}
```

In the parent component's template, you can handle the event:

```
<app-child-component (buttonClicked)="handleButtonClick()"></app-child-component>
```

Methods:

Components can have public methods that allow external code to invoke specific actions on the component. These methods can be called from other components or services.

```
export class MyComponent {
  public doSomething() {
    // Perform some action
  }
}
```

Other parts of your application can be called doSomething() on an instance of MyComponent.

Properties:

Public properties in a component can be accessed and modified from external code.

```
export class MyComponent {
  public title: string = 'Hello'.
}
```

Other parts of your application can read and change the value of the title property.

Lifecycle Hooks:

Angular provides lifecycle hooks that allow you to react to different stages of a component's lifecycle, such as initialization, change detection, and destruction. These hooks provide opportunities to interact with a component's behavior at specific points.

```
export class MyComponent implements OnInit, OnDestroy {
  ngOnInit() {
    // Component initialization logic
  }

  ngOnDestroy() {
    // Component cleanup logic
  }
}
```

Services:

While not strictly part of the component API, services are used to provide reusable functionality across components. Components can use services to delegate specific tasks, making the component's API cleaner and more focused.

By designing and exposing a well-defined component API, you enable better separation of concerns and create components that are easier to understand, maintain, and reuse throughout your Angular application.

The view will be developed in accordance with the model. When a new choice is picked, the component emits an output event called selection. Events are what a component produces as its

output. They communicate any important changes to the component's internal state to the outside world.

Creating re-usable Components

1. Create an application using the following cli command:
   ng new app1
2. Create simple component using cli command. in component file we will write code as like bellow.
   ng g component Post
3. Update src/app/post/post.component.ts

Example

```
import { Component, OnInit, Input } from '@angular/core';
 @Component({
 selector: 'my-post',
 templateUrl: './post.component.html',
 styleUrls: ['./post.component.css']
})
export class PostComponent{
 @Input() post:any;
}
```

4. Write the following code in src/app/post/post.component.html

```
<div>
   <p><strong>{{ post.id }}</strong>. {{ post.title }}</p>
</div>
```

5. In the app component update the following files:

src/app/app.component.ts

```
import { Component } from '@angular/core';
 @Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
 title = 'appComponent';
 posts = [
   {
     id: 1,
     title: 'Example of creating new component'
   },
   {
     id: 2,
```

```
      title: 'how to make angular component re-usable?'
    }
  ];
}
```

6. Reuse the post component.

Update the following file

src/app/app.component.html

```
<h3>Code for the creation of reusable components in Angular</h3>


<my-post *ngFor="let post of posts" [post]="post"></my-post>
```

7. Now you can run this app using ng serve to view the following output.

Output

Code for the creation of reusable components in Angular

1. Example of creating new component

2. how to make angular component re-usable?

Note: Once you’ve created your reusable Angular components, you can use an open-source toolchain such as Bit to “harvest” them from any codebase and share them on bit.dev. This would let your team reuse and collaborate on components to write scalable code, speed up development, and maintain a consistent UI.

Summary

- Inputs and Outputs facilitate communication and data flow between components.
- Components can have public methods that perform specific actions when invoked.
- Public properties can be accessed and modified from external code.
- Methods and properties provide a way to interact with component behavior.
- Component APIs enable communication and modularity in Angular applications.
- Inputs, Outputs, methods, and properties provide a way to implement data binding between components.
- Component APIs promote encapsulation, allowing components to hide internal details.

Keywords

@Input: Decorator to define a property as an input that can be set by a parent component.

@Output: Decorator to define a property as an output that emits events to be captured by a parent component.

Component Lifecycle: The sequence of events a component goes through from creation to destruction.

Bindings: Mechanisms for connecting component properties to template elements.

Template: The HTML representation of a component's UI.

Selector: A CSS selector used to identify where a component will be used in the template.

Directive: A feature that allows you to attach behavior to an HTML element.

Encapsulation: The concept of keeping the internal implementation of a component hidden and exposing only necessary functionality through its API.

Lifecycle Hooks: Methods that allow you to tap into various stages of a component's lifecycle.

Self Assessment

1. What is the purpose of the @Input decorator in Angular components?
A. It defines a property as an output for emitting events.
B. It defines a property as an input for receiving data from parent components.
C. It defines a property as a method to be called by other components.
D. It defines a property as a method for Angular's change detection.

2. Which Angular decorator is used to emit events from a component?
A. @EmitEvent
B. @Output
C. @EventEmitter
D. @Emit

3. In Angular, which lifecycle hook is called after a component's inputs have been initialized?
A. ngAfterViewInit
B. ngAfterContentInit
C. ngOnInit
D. ngOnChanges

4. Which of the following is not a valid view encapsulation strategy in Angular?
A. ShadowDom
B. Emulated
C. Native
D. None

5. Which Angular directive is used for content projection in component templates?
A. ng-projection
B. ng-slot
C. ng-content
D. ng-template

6. What is the purpose of Angular services in the context of components?
A. They define component templates.
B. They provide shared functionality and data to components.
C. They encapsulate component styles.
D. They handle component events.

7. Which of the following is a valid way to pass data from a parent component to a child component?
A. Using ViewChild
B. Using EventEmitter

C. Using ng-content
D. Using @Input

8. Which lifecycle hook is called just before a component is destroyed and removed from the DOM?
A. ngOnDestroy
B. ngDestroy
C. ngRemoved
D. ngCleanup

9. Which Angular directive is used to define template-driven forms?
A. ng-form
B. ng-model
C. ng-template
D. ngSubmit

10. Which Angular feature allows you to dynamically load and render components?
A. ng-include
B. ng-dynamic
C. Component Loader
D. Dynamic Component Loader

11. What is the purpose of the ngOnInit lifecycle hook in Angular?
A. To perform initialization logic for a component.
B. To handle changes in component inputs.
C. To clean up resources before a component is destroyed.
D. To render dynamic content in the component's template.

12. What is the purpose of template variables in Angular?
A. To store component data.
B. To reference template elements or components.
C. To define methods in the component.
D. To encapsulate component styles.

13. Which Angular feature allows you to define and apply custom behavior to elements?
A. ng-directive
B. ng-style
C. ng-class
D. Directive

14. What is the primary benefit of using view encapsulation strategies in Angular?
A. They prevent data leakage from components.
B. They simplify component API design.
C. They improve component rendering performance.
D. They allow components to share data seamlessly.

15. What is the primary role of the @Output decorator in Angular components?
A. To define properties for receiving input from parent components.
B. To emit events that can be captured by parent components.
C. To define methods that can be called by child components.
D. To create dynamic templates for components.

Review Questions

1. Explain the concept of a component's API in Angular. How does the component API facilitate communication and interaction between different parts of an Angular application?
2. Provide examples of how @Input, @Output, methods, and properties contribute to the component API's functionality.
3. Discuss the significance of services in the context of the component API. How do services contribute to modular and maintainable Angular applications?
4. Explain the importance of component reusability in Angular development. How does building reusable components contribute to maintainability, scalability, and code efficiency?
5. Describe the key strategies and techniques for achieving component reusability in Angular.

Answers for Self Assessment

1.	B	2.	B	3.	D	4.	C	5.	C
6.	B	7.	D	8.	A	9.	D	10.	D
11.	A	12.	B	13.	D	14.	A	15.	B

Further Readings

Wilken, J. (2018). *Angular in action*. Simon and Schuster.

Holmes, S. (2019). *Getting MEAN with Mongo, Express, Angular, and Node*. Simon and Schuster.

Moiseev, A., & Fain, Y. (2018). *Angular Development with TypeScript*. Simon and Schuster.

Moiseev, A., & Fain, Y. (2018). *Angular Development with TypeScript*. Simon and Schuster.

Web Links

https://blog.bitsrc.io/create-customisable-reusable-angular-components-with-transclusion-6343fb251e09

https://angular.io/guide/component-overview

https://www.tutorialsteacher.com/angular/angular-component

https://www.knowledgehut.com/blog/web-development/components-in-angular

Unit 09: Template-driven Forms

CONTENTS

Objectives

Introduction

9.1 Building a Bootstrap Form

9.2 Types of Forms

9.3 Ng Model

9.4 Adding Validations in Bootstrap Form

9.5 Specific Validation Errors

9.6 Styling Invalid Input Fields

9.7 Cleaner Templates ng Form

9.8 Ng Model Group

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit, you will be able to:

- Learn the creation of different types of forms in bootstrap.
- Know and implement the various form validations.
- Understand the various methods used for Styling Invalid Input Fields

Introduction

Template-driven forms are a way of handling form elements in web applications, particularly in the context of Bootstrap, a popular front-end framework. Template-driven forms are a part of Angular, a powerful JavaScript framework, but they can be used independently of Angular as well. In a template-driven form, the form structure and validation rules are defined directly in the HTML template, making it easier and faster to create simple forms with less coding. Here's a brief introduction to using template-driven forms in Bootstrap:

Setting up Bootstrap: Before getting started, ensure that you have included Bootstrap in your project. You can include it by adding the necessary CSS and JS files to your HTML file or by using a package manager like npm if you are using a build tool like Webpack.

HTML Form Structure: Start by creating the HTML form structure using Bootstrap classes for styling. You can use various form components like input, textarea, select, etc., and add Bootstrap classes to style them properly.

Example

html

Copy code

```
<form>
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control" id="name" name="name" [(ngModel)]="formData.name" required>
  </div>
    <div class="form-group">
    <label for="email">Email</label>
    <input type="email" class="form-control" id="email" name="email" [(ngModel)]="formData.email" required>
  </div>
  <!-- Additional form elements here -->
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

ngModel Two-Way Data Binding: Notice the [(ngModel)] directive in the input fields. It's part of Angular's two-way data binding feature, which allows you to bind the form fields directly to properties in your component. In this example, we are binding the name and email fields to formData.name and formData.email respectively.

Form Validation: Bootstrap provides styling for various form validation states, such as valid, invalid, touched, etc. You can leverage these styles to enhance user experience when handling form validation. Angular will automatically add and remove these classes based on the form's validation status.

Handling Form Submission: To handle the form submission, you can listen to the submit event of the form element in your component and perform the necessary actions, such as sending the form data to a server or performing client-side validation.

Example

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-your-component',
  templateUrl: './your-component.component.html',
  styleUrls: ['./your-component.component.css']
})
export class YourComponent {
  formData: any = {}; // Initialize an object to store form data.

  onSubmit() {
    // Perform actions when the form is submitted, e.g., sending data to the server.
    console.log(this.formData);
  }
}
```

Bootstrap CSS Classes: Remember to apply appropriate Bootstrap CSS classes to enhance the overall look and feel of your form. You can use various classes like form-group, form-control, btn, etc.

Note: Remember, template-driven forms are great for simpler forms with basic validation needs. For more complex forms or custom validation requirements, you might want to consider using Reactive Forms, another approach provided by Angular. However, template-driven forms can be a good starting point to get a form up and running quickly.

9.1 Building a Bootstrap Form

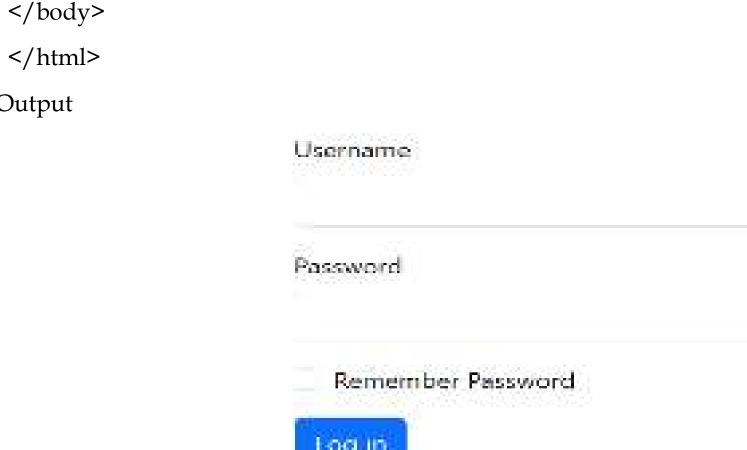
Users can submit information on a form, such as their name, email address, password, etc., and the information can subsequently be transmitted to a server for processing. In order to develop a range of forms with different styles, layouts, and unique components, Bootstrap offers classes. Bootstrap form controls expand Form styles were restarted with classes. Use these classes for uniform rendering across browsers and devices with customized display. Use an appropriate type attribute on all inputs (such as email for email addresses or the number for numerical data) if you want to use more recent input controls like email verification, number selection, and other features.

Example

```
<!DOCTYPE html>
 <html lang="en">
 <head>
  <title>Bootstrap - Form</title>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap.min.css" rel="stylesheet">
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/js/bootstrap.bundle.min.js"></script>
 </head>
 <body>
  <form>
   <div class="mb-3">
    <label for=" sampleInputEmail" class="form-label">Username</label>
    <input type="email" class="form-control" id=" sampleInputEmail" aria-describedby="emailHelp">
   </div>
   <div class="mb-3">
    <label for="sampleInputPassword" class="form-label">Password</label>
    <input type="password" class="form-control" id="sampleInputPassword">
   </div>
   <div class="mb-3 form-check">
    <input type="checkbox" class="form-check-input" id="sampleCheck">
    <label class="form-check-label" for="sampleCheck">Remember me</label>
   </div>
   <button type="submit" class="btn btn-primary">Log in</button>
  </form>
```

```
</body>
</html>
```

Output



9.2 Types of Forms

Bootstrap offers three different form layout types:

The same guidelines apply to all three form layouts:

Vertical form (this is default)

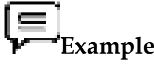
Horizontal form

Inline form

Labels and form controls should be enclosed in <div class="form-group">. All text-containing input>, textarea>, and select> elements should be given the class.form-control.

Vertical Form

In Bootstrap, a vertical form is a form layout where form elements are stacked vertically, one on top of another. This layout is the default behavior of Bootstrap forms, so you don't need to do anything special to create a vertical form.

**Example**

```
<form action="/action_page.php">
  <div class="form-group">
    <label for="email">Email address:</label>
    <input type="email" class="form-control" id="email">
  </div>
  <div class="form-group">
    <label for="pwd">Password:</label>
    <input type="password" class="form-control" id="pwd">
  </div>
  <div class="checkbox">
    <label><input type="checkbox"> Remember me</label>
  </div>
  <button type="submit" class="btn btn-default">Submit</button>
</form>
```

Output

Email address:
Password:
Remember me
Submit

Inline Form

In an inline form, all of the elements are inline, left-aligned, and the labels are alongside. Additional rule for an inline form:

Add class .form-inline to the <form> element

The following example creates an inline form with two input fields, one checkbox, and one submit button:

Example

```
<!DOCTYPE html>
<html>
<head>
  <title>Inline Form Example</title>
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
  <div class="container">
    <h2>Inline Form Example</h2>
    <form class="form-inline">
      <div class="form-group mb-2">
        <label for="name" class="sr-only">Name</label>
        <input type="text" class="form-control" id="name" name="name" placeholder="Name" required>
      </div>

      <div class="form-group mx-sm-3 mb-2">
        <label for="email" class="sr-only">Email</label>
        <input type="email" class="form-control" id="email" name="email" placeholder="Email" required>
      </div>

      <div class="form-group mb-2">
        <label for="message" class="sr-only">Message</label>
        <textarea class="form-control" id="message" name="message" placeholder="Message" rows="4" required></textarea>
      </div>
        <!-- Add more form elements here -->
        <button type="submit" class="btn btn-primary mb-2">Submit</button>
```

```
    </form>
  </div>
</body>
</html>
```

Output

Note: This only applies to forms within viewports that are at least 768px wide!

Horizontal Form

A horizontal form means that the labels are aligned next to the input field (horizontal) on large and medium screens. On small screens (767px and below), it will transform to a vertical form (labels are placed on top of each input).Additional rules for a horizontal form:

- Add class .form-horizontal to the <form> element
- Add class .control-label to all <label> elements

Example

```
<!DOCTYPE html>
<html>
<head>
  <title>Horizontal Form Example</title>
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
  <div class="container">
    <h2>Horizontal Form Example</h2>
    <form>
      <div class="form-group row">
        <label for="name" class="col-sm-2 col-form-label">Name</label>
        <div class="col-sm-10">
          <input type="text" class="form-control" id="name" name="name" required>
        </div>
      </div>

      <div class="form-group row">
```

```
    <label for="email" class="col-sm-2 col-form-label">Email</label>
    <div class="col-sm-10">
      <input type="email" class="form-control" id="email" name="email" required>
    </div>
   </div>

   <div class="form-group row">
    <label for="message" class="col-sm-2 col-form-label">Message</label>
    <div class="col-sm-10">
      <textarea class="form-control" id="message" name="message" rows="4" required></textarea>
    </div>
   </div>
      <!-- Add more form elements here -->
      <div class="form-group row">
    <div class="col-sm-10 offset-sm-2">
      <button type="submit" class="btn btn-primary">Submit</button>
    </div>
   </div>
  </form>
 </div>
</body>
</html>
```

Output

Horizontal Form Example

Note: Use Bootstrap's predefined grid classes to align labels and groups of form controls in a horizontal layout.

9.3 Ng Model

The ngModel is a built-in directive that provides two-way data binding between the form input elements and the component's data model. It is used to link the value of an input field to a property in the component class. When the user modifies the input field, the property in the component is updated automatically, and vice versa. When using ngModel in Bootstrap forms, you can easily

bind form elements to the component's data model, making it easy to work with form data. Here's how you can use ngModel in Bootstrap forms:

Import FormsModule: First, you need to import the FormsModule in your Angular module to use ngModel. Make sure to include it in the imports array of the module where your component is declared.

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
// Other imports


@NgModule({
  imports: [
    // Other modules
    FormsModule
  ],
  declarations: [
    // Your component declarations
  ],
  providers: [],
  bootstrap: []
})
export class AppModule { }
```

Add ngModel to Form Elements: Now, you can use ngModel directive in your Bootstrap form elements to bind them to the component's data model. For example, if you have an input field for the user's name, you can bind it to a property called userName in your component.

```
html
<form>
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control" id="name" name="name" [(ngModel)]="userName" required>
  </div>
   <!-- Other form elements here -->
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Component Class: In your component class, you need to define the userName property, which will be bound to the input field through ngModel.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-your-component',
  templateUrl: './your-component.component.html',
  styleUrls: ['./your-component.component.css']
})
export class YourComponent {
```

```
  userName: string = ''; // Initialize the property with an empty string
  // Other component logic and methods can go here
}
```

With this setup, the userName property in your component will always be in sync with the value entered by the user in the input field. When the user updates the input field, the userName property in the component will be automatically updated, and you can use its value in your component logic or submit it to the server as needed. ngModel is a powerful feature in Angular that simplifies the handling of form elements, and it works seamlessly with Bootstrap forms to enhance the user experience in your web application.

9.4 Adding Validations in Bootstrap Form

Form validation is "the technical process by which a web form verifies the accuracy of the user's input." Either the form will prompt the user to amend their mistake or it will alert them to it. A user will be able to demonstrate validity beforehand. Users may continue with their user interaction because the form is legitimate. The form validation feature checks to see if all required fields have data at first. To iterate through each form field and check for data, only five bootstrapped form validation classes are required. The entered data must then be checked for accuracy of form and value.

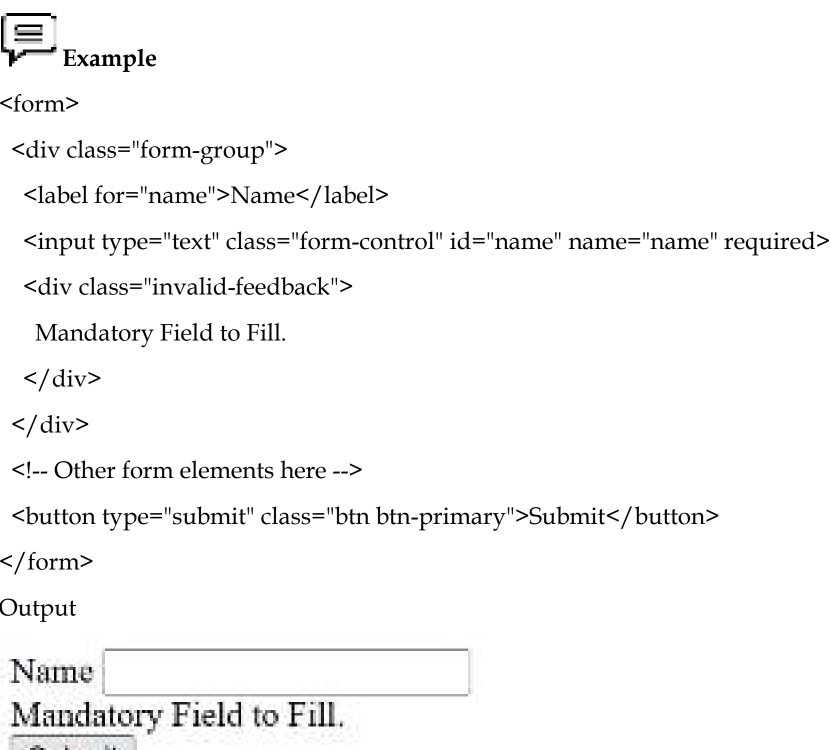
Form validation in Bootstrap is made easy through the use of Bootstrap's built-in CSS classes and Angular's validation features when using Angular forms. By combining these, you can create a visually appealing and user-friendly validation experience for your users. Let's see how to perform form validation in Bootstrap:

Required Fields: To make a field required, simply add the required attribute to the form control element.

Example

```
<form>
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control" id="name" name="name" required>
    <div class="invalid-feedback">
      Mandatory Field to Fill.
    </div>
  </div>
  <!-- Other form elements here -->
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Output



Feedback Styles: Bootstrap provides validation styles for both valid and invalid states. For invalid fields, you can use the is-invalid class along with the invalid-feedback class to display error messages. The is-valid class can be used for valid fields along with the valid-feedback class.

Example

```
<form>
  <div class="form-group">
    <label for="email">Email</label>
    <input type="email" class="form-control" id="email" name="email" required>
    <div class="invalid-feedback">
      Please provide a valid email.
    </div>
    <!-- <div class="valid-feedback">
      Looks good!
    </div> -->
  </div>
  <!-- Other form elements here -->
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Output

Email

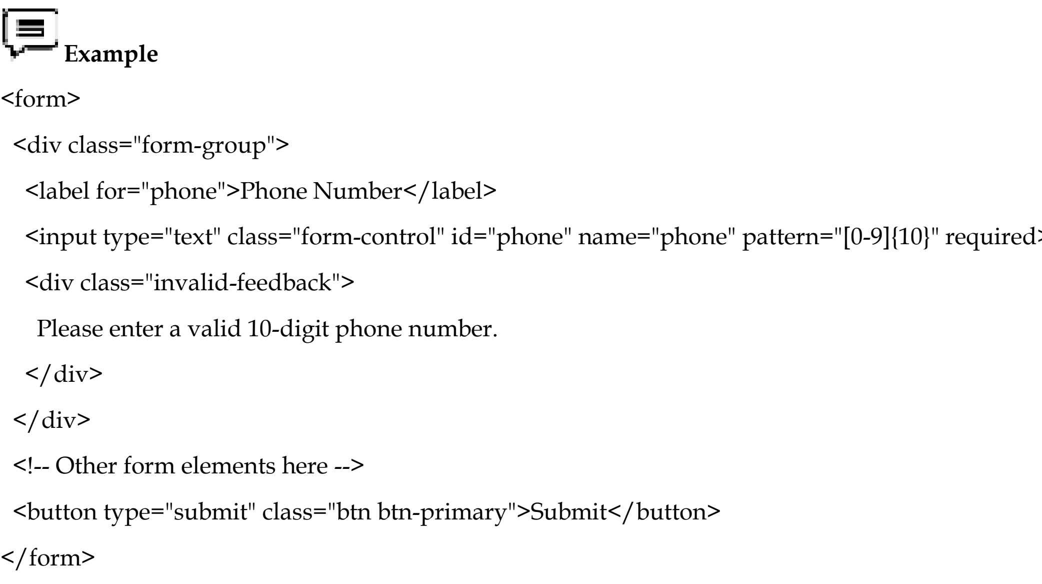
Please provide a valid email.

Submit

9.5 Specific Validation Errors

In Bootstrap, you can display specific validation errors for different types of input validation using the pattern, minlength, and maxlength attributes on form controls, along with Bootstrap's validation styles and error feedback classes. Here's how you can display specific validation errors in a Bootstrap form:

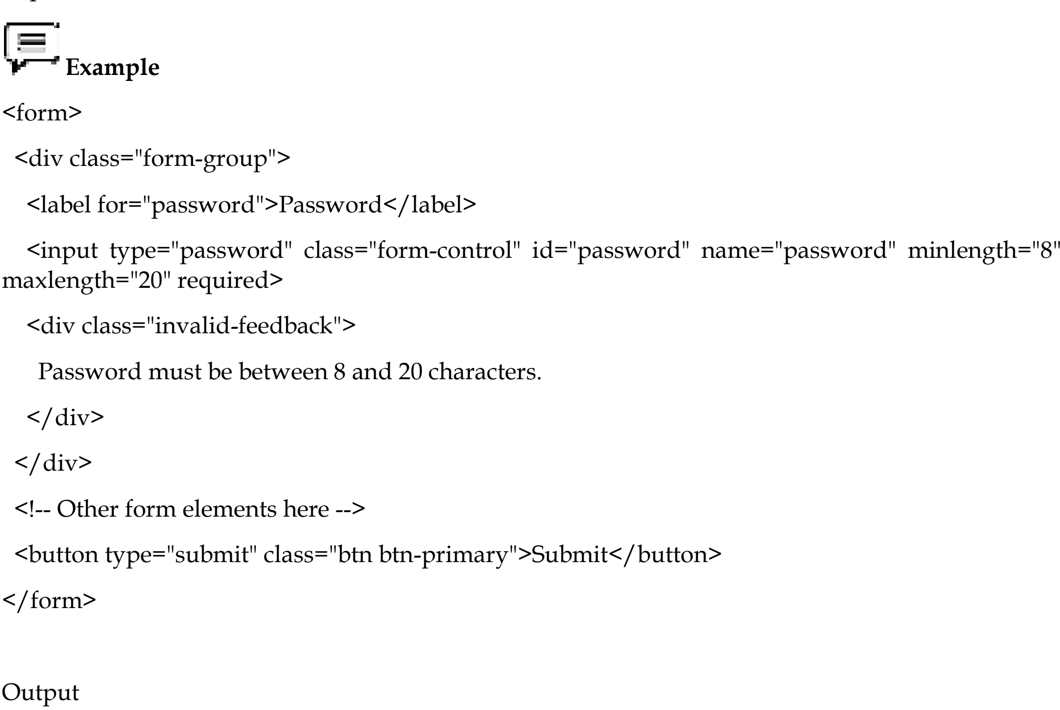
Displaying Error Messages for Pattern Validation: Use the pattern attribute to enforce a specific pattern for an input field, such as a regular expression. Add the invalid-feedback class to display the error message when the input value does not match the specified pattern.



Example

```
<form>
  <div class="form-group">
    <label for="phone">Phone Number</label>
    <input type="text" class="form-control" id="phone" name="phone" pattern="[0-9]{10}" required>
    <div class="invalid-feedback">
      Please enter a valid 10-digit phone number.
    </div>
  </div>
  <!-- Other form elements here -->
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Output



Displaying Error Messages for Minimum and Maximum Length: Use the minlength and maxlength attributes to enforce minimum and maximum lengths for an input field. Add the invalid-feedback class to display the error messages when the input length does not meet the requirements.

Example

```
<form>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control" id="password" name="password" minlength="8" maxlength="20" required>
    <div class="invalid-feedback">
      Password must be between 8 and 20 characters.
    </div>
  </div>
  <!-- Other form elements here -->
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Output

Password

Password must be between 8 and 20 characters.

Submit

Displaying Custom Error Messages with Angular Form Validation: When using Angular forms, you can create custom validation rules using validators. You can display custom error messages for these validators using Angular's form control methods. Add the invalid-feedback class to display the error messages when the input does not meet the custom validation rules.

Example

```
<form [formGroup]="myForm">
  <div class="form-group">
    <label for="age">Age</label>
    <input type="number" class="form-control" id="age" formControlName="age">
    <div *ngIf="myForm.controls.age.invalid && myForm.controls.age.errors?.min" class="invalid-feedback">
      Age must be at least 18.
    </div>
    <div *ngIf="myForm.controls.age.invalid && myForm.controls.age.errors?.max" class="invalid-feedback">
      Age must be less than 100.
    </div>
```

```
  </div>
  <!-- Other form elements here -->
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Output

Age
Age must be at least 18.
Age must be less than 100.
Submit

In this example, we have a custom validator in the form group myForm to check the age input. If the age is less than 18 or greater than 100, the corresponding error messages will be displayed using Angular's template syntax.

9.6 Styling Invalid Input Fields

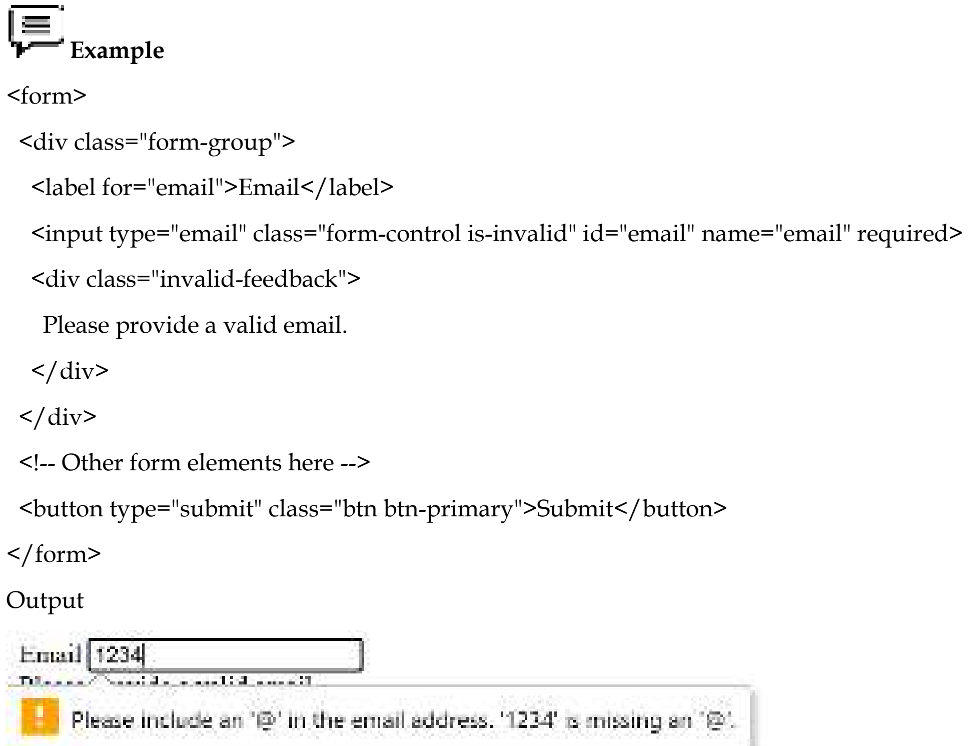
In Bootstrap, you can style invalid input fields to provide visual feedback to users when they submit a form with errors. Bootstrap provides CSS classes to apply specific styles to invalid form controls. To style invalid input fields, follow these steps:

Add the is-invalid class to Invalid Fields: In your HTML form, add the is-invalid class to the form control element when it fails validation. Bootstrap will apply the appropriate styling to indicate that the input is invalid.

Example

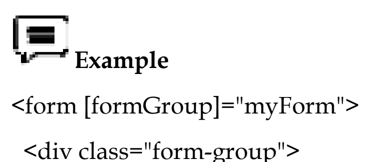
```
<form>
  <div class="form-group">
    <label for="email">Email</label>
    <input type="email" class="form-control is-invalid" id="email" name="email" required>
    <div class="invalid-feedback">
      Please provide a valid email.
    </div>
  </div>
  <!-- Other form elements here -->
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Output



Conditional Styling with Angular Form Validation: If you are using Angular forms and want to conditionally apply the is-invalid class based on form validation, you can use Angular's form control methods and interpolation in your template.

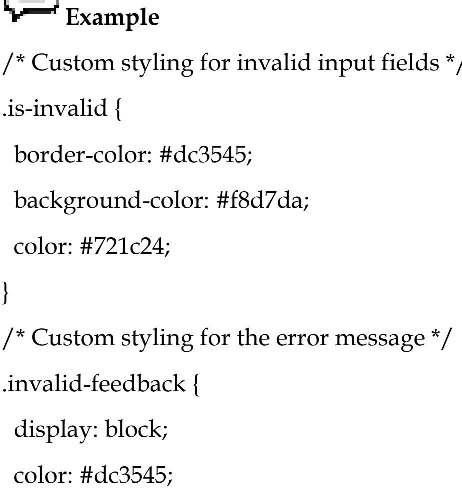
Example



```
<form [formGroup]="myForm">
  <div class="form-group">
```

```
  <label for="password">Password</label>
  <input type="password" class="form-control" id="password" formControlName="password" [class.is-invalid]="myForm.get('password').invalid && myForm.get('password').touched">
  <div *ngIf="myForm.get('password').invalid && myForm.get('password').touched" class="invalid-feedback">
    Password must be at least 8 characters long.
  </div>
 </div>
 <!-- Other form elements here -->
 <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Output



In this example, we use Angular's formControlName to bind the input field to the form control in the component, and we conditionally apply the is-invalid class based on whether the password field is invalid and has been touched by the user.

Customizing Invalid Field Styling: If you want to customize the appearance of invalid input fields, you can override Bootstrap's default styles for the is-invalid class in your CSS.

Example

```
/* Custom styling for invalid input fields */
.is-invalid {
  border-color: #dc3545;
  background-color: #f8d7da;
  color: #721c24;
}
/* Custom styling for the error message */
.invalid-feedback {
  display: block;
  color: #dc3545;
}
```

In this CSS example, we've changed the border color, background color, and text color for invalid input fields. We've also set the error message text color to red.

9.7 Cleaner Templates ng Form

To create cleaner templates when using ngForm in Bootstrap, you can utilize Angular's template-driven forms and take advantage of Angular's form validation features. By structuring your templates effectively and leveraging Angular's directives, you can keep your code clean and concise. Here's how you can achieve a cleaner template when using ngForm in Bootstrap:

Wrap Your Form Elements with ngForm: To use template-driven forms, wrap your form elements with the ngForm directive, which represents the form itself. This helps Angular manage the form and its controls.

```
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
  <!-- Form elements here -->
</form>
```

Use the ngModel Directive: For each form control, use the ngModel directive to bind the input value to a property in your component class. This creates two-way data binding and keeps your component data in sync with the form inputs.

```
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label for="name">Name</label>
    <input type="text" class="form-control" id="name" name="name" [(ngModel)]="formData.name" required>
  </div>
  <div class="form-group">
    <label for="email">Email</label>
    <input type="email" class="form-control" id="email" name="email" [(ngModel)]="formData.email" required>
  </div>
  <!-- Other form elements here -->

  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Handle Form Submission: When the form is submitted, you can handle the form data in your component by creating a method that gets triggered on the form's (ngSubmit) event.

```
<form #myForm="ngForm" (ngSubmit)="onSubmit()">
  <!-- Form elements here -->
</form>
import { Component } from '@angular/core';
@Component({
  selector: 'app-your-component',
  templateUrl: './your-component.component.html',
  styleUrls: ['./your-component.component.css']
})
export class YourComponent {
  formData: any = {}; // Initialize an object to store form data.
  onSubmit() {
    // Perform actions when the form is submitted, e.g., sending data to the server.
    console.log(this.formData);
  }
}
```

9.8 Ng Model Group

ngModelGroup is used to create a top-level form group Instance, and it binds the form to the given form value.

Syntax:

```
<div ngModelGroup="name"></div>
```

NgModule: Module used by NgModelGroup is:

FormsModule

[ngModelGroup]

Example

app.component.ts

```
import { Component, Inject } from '@angular/core';
@Component({
        selector: 'app-root',
        templateUrl: './app.component.html',
        styleUrls: [ './app.component.css' ]
})
export class AppComponent {
        aa:any;

setValue() {
        this.aa = {first: 'Hello', email: 'abc@aa.com'};
}
}
```

app.component.html

```
<form #f="ngForm">
<div ngModelGroup="aa" #nameCtrl="ngModelGroup">
        Name: <input name="first" [ngModel]="aa.first" >
        Email: <input name="email" [ngModel]="aa.email" >
</div>
</form>
<br><br>
<button (click)="setValue()">Set value</button>
```

Output

Summary

- Template-driven forms are part of Angular, and they work seamlessly with Bootstrap to create stylish and responsive forms.

- Template-driven forms are easy to implement as they require less code and minimal setup compared to Reactive forms, making them a quicker choice for simple forms.
- In Template-driven forms, the form structure and validation rules are primarily defined directly in the HTML template.
- Angular's ngModel directive is used for two-way data binding, linking form elements to properties in the component class.
- Bootstrap provides styling for various form validation states like valid, invalid, and touched. Angular automatically adds and removes these classes based on the form's validation status.
- The ngForm directive represents the form itself and is used to manage the form and its controls. It allows you to access the form's validity and control the form submission behavior.
- The ngModelGroup directive is used to group related form controls together under a common name, providing cleaner and more organized templates.
- Bootstrap provides invalid-feedback and valid-feedback classes to display error and success messages for form validation.
- Form submission is handled using Angular's (ngSubmit) event in combination with a method in the component to process the form data.
- You can customize the form layout and styles by applying various Bootstrap classes and CSS styles as per your design requirements.

Keywords

ngForm: The Angular directive used to represent the entire form element. It enables form management and validation.

ngModel: The Angular directive used for two-way data binding, linking form elements to properties in the component class.

formGroup: The Angular directive used to represent a group of form controls. It is mainly used in Reactive forms.

form-control: The Bootstrap class used to style form input elements.

invalid-feedback: The Bootstrap class used to display error messages for invalid form elements.

valid-feedback: The Bootstrap class used to display success messages for valid form elements.

required: The HTML attribute used to make a form element mandatory.

pattern: The HTML attribute used to enforce a specific pattern (regular expression) for a form element.

minlength: The HTML attribute used to specify the minimum length for a form element.

maxlength: The HTML attribute used to specify the maximum length for a form element.

Self Assessment

1. Which Bootstrap class is used to create a horizontal form layout?

A. form-horizontal
B. form-inline
C. form-grid
D. form-horizontal-layout

2. How do you add form validation feedback for invalid input fields in Bootstrap?

A. Use the invalid-field class

B. Add the is-invalid class to the form control
C. Apply the form-error class to the form element
D. Use the error-feedback class

3. Which HTML attribute is used to enforce a specific pattern (regular expression) for a form element in Bootstrap forms?

A. data-pattern
B. regex
C. data-validation
D. pattern

4. In Bootstrap, which class is used to group related form elements together and apply styling like margin and padding?

A. form-inline
B. form-group
C. form-control
D. form-wrapper

5. When using Template-driven forms in Angular, which directive is used to represent the entire form element and enable form management and validation?

A. formGroup
B. ngForm
C. formControl
D. ngModel

6. Which type of Bootstrap form is stacked vertically, with form elements displayed on separate lines?

A. Inline form
B. Horizontal form
C. Vertical form
D. Expanded form

7. Which type of Bootstrap form displays form elements on a single line, without line breaks between them?

A. Stacked form
B. Horizontal form
C. Nested form
D. Inline form

8. In Bootstrap, which type of form is aligned in a row, with labels placed to the left of form controls?

A. Vertical form
B. Stacked form
C. Horizontal form
D. Compact form

9. Which Bootstrap class is used to create a compact and horizontally aligned form?
A. form-inline
B. form-horizontal
C. form-row
D. form-compact

10. Which type of Bootstrap form is suitable for complex form structures or nesting forms within forms?
A. Simple form
B. Template-driven form
C. Reactive form
D. Nested form

11. In Bootstrap, which class is used to style invalid form controls and display error messages?
A. is-invalid
B. form-error
C. error-feedback
D. invalid-form

12. Which HTML attribute is used to make a form element mandatory and enforce form validation in Bootstrap forms?
A. required
B. validation
C. mandatory
D. validate

13. What Bootstrap class is used to style form controls with valid input and display success messages?
A. is-valid
B. form-success
C. valid-feedback
D. success-control

14. In Bootstrap, how can you create custom form validation messages for specific input patterns?
A. Using the data-validation attribute
B. By adding the invalid-feedback class to the form control
C. Utilizing the pattern attribute on the input element
D. Applying the data-pattern attribute to the input element

15. Which Bootstrap class is used to style form elements that fail validation and display the error message?
A. error-input
B. invalid-message
C. error-control
D. invalid-feedback

Answers for Self Assessment

1.	A	2.	B	3.	D	4.	B	5.	B
6.	C	7.	D	8.	C	9.	A	10.	C
11.	A	12.	A	13.	C	14.	B	15.	D

Review Questions

1. Explain the key features and advantages of using Bootstrap forms in web development with the help of example.
2. Describe how form validation can be implemented in Bootstrap forms to enhance the user experience.
3. Explain the three different types of forms available in Bootstrap.
4. When should one prefer using a Vertical form over a Horizontal form in Bootstrap? Explain with the help of an example.
5. Explain the concept of form validation in Bootstrap and how it enhances the user experience.
6. Discuss the different types of validation states and how they are applied in Bootstrap forms.

Further Readings

Spurlock, J. (2013). *Bootstrap: responsive web development*. " O'Reilly Media, Inc.".

Enterprise, J. (2016). *Pemrograman Bootstrap untuk Pemula*. Elex Media Komputindo.

Jakobus, B., & Marah, J. (2016). *Mastering bootstrap 4*. Packt Publishing Ltd.

Efron, B., & Tibshirani, R. J. (1994). *An introduction to the bootstrap*. CRC press.

Davison, A. C., & Hinkley, D. V. (1997). *Bootstrap methods and their application* (No. 1). Cambridge university press.

Web Links

https://getbootstrap.com/docs/5.1/forms/

https://getbootstrap.com/docs/5.1/forms/validation/

https://getbootstrap.com/docs/5.1/examples/

https://angular.io/guide/forms

https://www.toptal.com/angular-js/angular-4-forms-validation

Unit 10: Form Handling in Angular

CONTENTS

Objectives
Introduction
10.1 Control Classes and Directives
10.2 Disabling the Submit Button
10.3 Working with Check Boxes
10.4 Working with Drop-down Lists
10.5 Working with Radio Buttons
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After this unit you will be able to:

- Know the working of Checkboxes and Radio buttons.
- Implement the disabling of submit button.
- Understand the working of drop down list

Introduction

In AngularJS, form handling is an essential part of building dynamic web applications. It allows you to capture user input, validate it, and perform various actions based on the input data. AngularJS provides a powerful and flexible form handling mechanism that makes it easy to work with forms.

To get started with form handling in AngularJS, you need to follow these steps:

Setting up the Form: Start by creating an HTML form using the <form> element. This form will contain input fields, checkboxes, radio buttons, and other form elements. Assign a unique name to the form using the name attribute. This name will be used to reference the form in the AngularJS controller.

Binding Form Controls: Use the ng-model directive to bind form controls (input fields, checkboxes, etc.) to variables in the AngularJS controller. This two-way data binding ensures that changes in the form controls are reflected in the controller, and vice versa.

Validation: AngularJS provides built-in validation directives that you can use to validate user input.

Example, you can use the ng-required directive to make a field required, or the ng-pattern directive to validate input based on a regular expression.

AngularJS automatically adds CSS classes (ng-valid, ng-invalid, ng-dirty, ng-pristine, etc.) to form controls based on their validation state. You can leverage these classes to style the form and provide visual feedback to the user.

Submitting the Form: To handle form submission, you can use the ng-submit directive on the form element. This directive allows you to specify a function to be executed when the form is submitted. In the specified function, you can access the form data stored in the controller's scope and perform actions such as sending data to a server, processing it, or navigating to another page.

Handling Form Errors: AngularJS provides various mechanisms to handle form errors and display error messages. You can use the $error object in the controller's scope to check the validation status of individual form controls and show corresponding error messages.

By following these steps, you can effectively handle forms in AngularJS. Remember to include the AngularJS library in your project, and make sure to set up the AngularJS module and controller to work with your form.

10.1 Control Classes and Directives

In Angular, there are several control classes and directives that are commonly used for form handling. These classes and directives provide powerful features to validate, track the state of form controls, and display error messages. Here are some of the key ones:

ngModel Directive: The ngModel directive is used for two-way data binding between form controls and the controller's scope variables. It allows you to bind form controls to properties in the controller, enabling you to access and manipulate the values entered by the user.

ngForm Directive: The ngForm directive is used to create a form element in Angular. It automatically creates an instance of the NgForm class, which provides additional functionality for form handling. The ngForm directive also exposes properties like valid, invalid, dirty, pristine, etc., which can be used to check the overall validity and state of the form.

ngSubmit Directive: The ngSubmit directive is used to specify a function to be executed when the form is submitted. It is typically used in conjunction with the ngForm directive.

When the form is submitted (either by clicking a submit button or pressing Enter), the specified function is called, allowing you to perform actions such as sending data to a server or performing further processing.

ngModelGroup Directive: The ngModelGroup directive is used to group related form controls together. It allows you to organize form controls into logical groups, which can be useful for validation and data organization. By grouping form controls, you can perform validation and access the group's properties, such as valid or invalid, based on the collective state of the controls within the group.

ngMessages Directive: The ngMessages directive is used to display error messages based on the validation state of form controls. It provides a convenient way to show custom error messages for different validation scenarios. By using the ngMessages directive, you can define error messages for specific validation errors and display them dynamically based on the control's validation state.

ngClass Directive: The ngClass directive allows you to conditionally apply CSS classes to form controls based on certain conditions. It is commonly used to add visual cues to form controls to indicate their validation state. With ngClass, you can dynamically add or remove CSS classes based on the control's validity, dirty/pristine state, or any other custom conditions.

These are just a few of the many directives and control classes available in Angular for form handling. They provide a rich set of features to make form validation, tracking, and error handling more manageable and efficient in your Angular applications.

10.2 Disabling the Submit Button

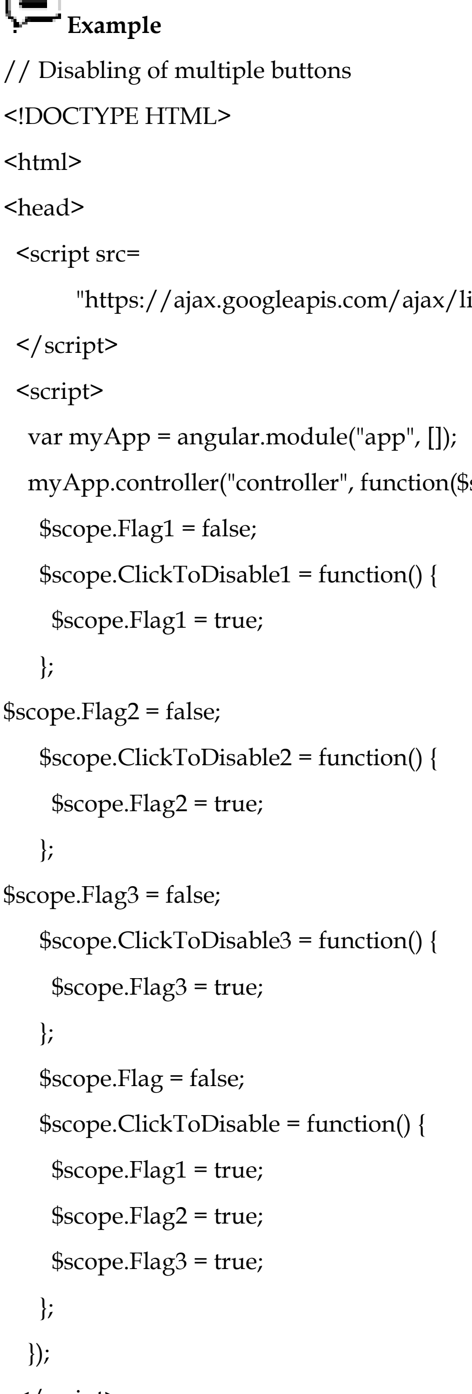
Disabling the submit button is a common practice in form handling to prevent multiple submissions or invalid submissions. In Angular, you can disable the submit button using the following steps:

- Define a boolean variable in your component to track the disabled state of the submit button. For example, let's call it disable Submit Button and initialize it to false.
- In your component's template, add the submit button element and bind the disabled attribute to the disable Submit Button variable using property binding.

Example

```
// Disabling of single submit button
<!DOCTYPE HTML>
<html>
<head>
        <script src=
"https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.min.js">
        </script>
        <script>
                var myApp = angular.module("app", []);
                myApp.controller("controller", function($scope) {
                        $scope.Disable = false;
                        $scope.dis = function() {
                                $scope.Disable = true;
                        };
                });
        </script>
</head>
<body style="text-align:center;">
        <h1 style="color:green;">
                Good Day Everyone
        </h1>
        <h3>
                Click the ok button to Disable it
        </h3>
        <div ng-app="app">
                <div ng-controller="controller">
                        <button ng-click='dis()'
                                        ng-disabled='Disable'>
                                OK
                        </button>
                </div>
        </div>
</body>
</html>
```

Output



Example

```
// Disabling of multiple buttons
<!DOCTYPE HTML>
<html>
<head>
  <script src=
          "https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.min.js">
  </script>
  <script>
    var myApp = angular.module("app", []);
    myApp.controller("controller", function($scope) {
      $scope.Flag1 = false;
      $scope.ClickToDisable1 = function() {
        $scope.Flag1 = true;
      };
$scope.Flag2 = false;
      $scope.ClickToDisable2 = function() {
        $scope.Flag2 = true;
      };
$scope.Flag3 = false;
      $scope.ClickToDisable3 = function() {
        $scope.Flag3 = true;
      };
      $scope.Flag = false;
      $scope.ClickToDisable = function() {
        $scope.Flag1 = true;
        $scope.Flag2 = true;
        $scope.Flag3 = true;
      };
    });
  </script>
</head>
<body style="text-align:center;">
<h1 style="color:red;">
```

```
  Welcome Everyone
</h1>
<h3>
  <b>Implementing submit button Disable</b>
</h3>
<div ng-app="app">
  <div ng-controller="controller">
    <button ng-click='ClickToDisable1()'
        ng-disabled='Flag1'>
      Click to disable
    </button>
    <button ng-click='ClickToDisable2()'
        ng-disabled='Flag2'>
      Click to Disable
    </button>
    <button ng-click='ClickToDisable3()'
        ng-disabled='Flag3'>
Click to Disable
    </button>
    <br>
    <br>
    <button ng-click='ClickToDisable()'
        ng-disabled='Flag'>
      Click to Disable All
    </button>
  </div>
</div>
</body>
</html>
```

Output

Welcome Everyone

Implementing submit button Disable

Click to disable | Click to Disable | Click to Disable

Click to Disable All

10.3 Working with Check Boxes

Step 1: Import FormsModule

If you want to create form in angular app then you need to import FormsModule from @angular/forms library. so let's add following code to app.module.ts file.

src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
 import { AppComponent } from './app.component';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
@NgModule({
 declarations: [
  AppComponent
 ],
 imports: [
  BrowserModule,
  FormsModule,
  ReactiveFormsModule
 ],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 2: Form with ngModel

In this step, we will write code of html form with ngModel. so add following code to app.component.html file.

src/app/app.component.html

```
<div class="container">
  <h1>Angular CheckBox Example - ItSolutionStuff.com</h1>
      <form [formGroup]="form" (ngSubmit)="submit()">
        <div class="form-group">
      <label for="website">Website:</label>
      <div *ngFor="let web of websiteList">
        <label>
          <input type="checkbox" [value]="web.id" (change)="onCheckboxChange($event)" />
          {{web.name}}
        </label>
      </div>
    </div>
        <button class="btn btn-primary" type="submit" [disabled]="!form.valid">Submit</button>
  </form>
</div>
```

Step 3: updated Ts File

In ts file. we will write submit() and get all input fields values. we need to import this libraries FormBuilder, FormGroup, FormControl, Validators, FormArray. so let's add following code to app.component.ts file.

src/app/app.component.ts

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, FormControl, Validators, FormArray} from '@angular/forms';
 @Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
 form: FormGroup;
 websiteList: any = [
  { id: 1, name: 'ItSolutionStuff.com' },
  { id: 2, name: 'HDTuto.com' },
  { id: 3, name: 'NiceSnippets.com' }
 ];
  constructor(private formBuilder: FormBuilder) {
  this.form = this.formBuilder.group({
    website: this.formBuilder.array([], [Validators.required])
  })
 }
    onCheckboxChange(e) {
  const website: FormArray = this.form.get('website') as FormArray;
    if (e.target.checked) {
    website.push(new FormControl(e.target.value));
  } else {
     const index = website.controls.findIndex(x => x.value === e.target.value);
     website.removeAt(index);
  }
 }
  submit(){
  console.log(this.form.value);
 }
  }
```

Now you can run your application using following command:

```
ng serve
```

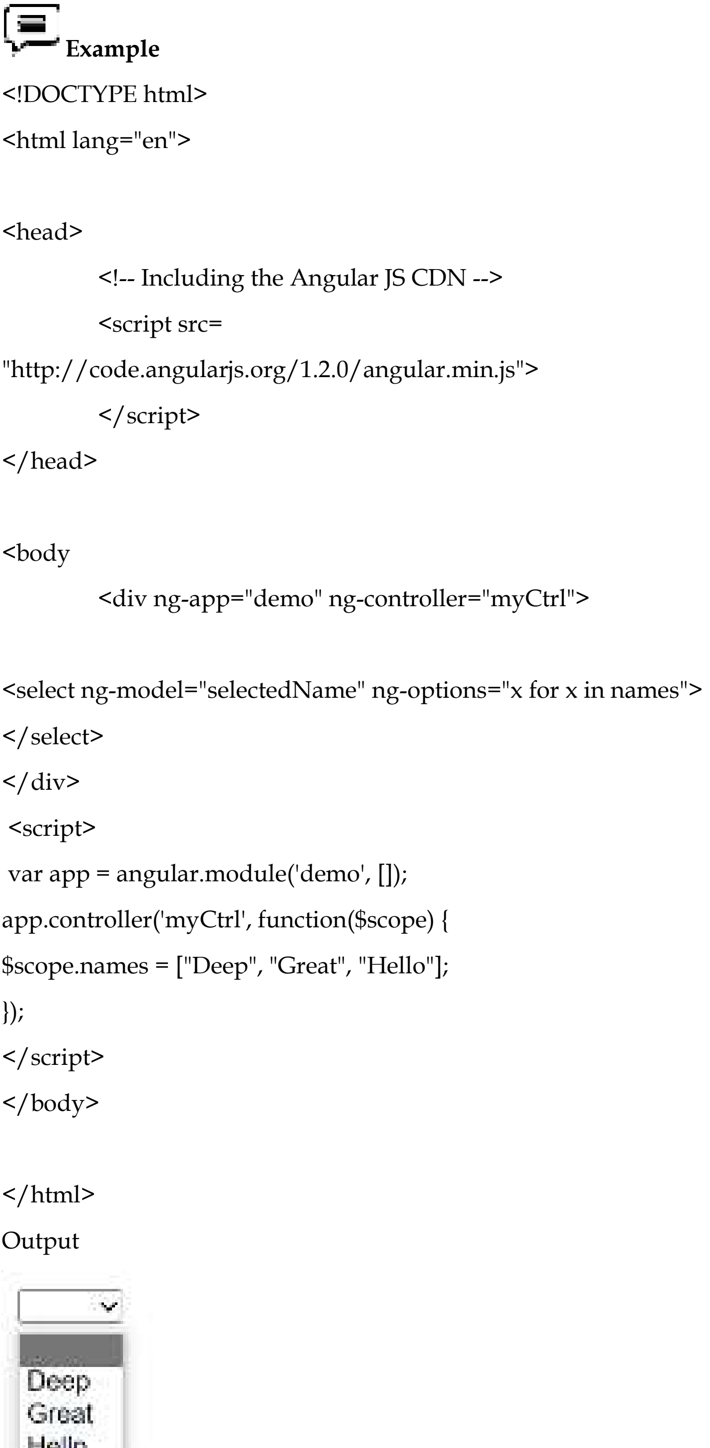
10.4 Working with Drop-down Lists

Working with dropdown lists in Angular involves using the <select> element along with the (change) event and data binding. In order to create a reactive form that will contain a dropdown box, we will use the loop. In this section, we will provide two examples to select dropdown. In the first example, when we click on the submit button, we are able to get the value of selected dropdown box. In the second example, when the change event has occurred, we are able to get the value of selected dropdown box.

Example

```
<!DOCTYPE html>
<html lang="en">

<head>
        <!-- Including the Angular JS CDN -->
        <script src=
"http://code.angularjs.org/1.2.0/angular.min.js">
        </script>
</head>

<body
        <div ng-app="demo" ng-controller="myCtrl">

<select ng-model="selectedName" ng-options="x for x in names">
</select>
</div>
 <script>
 var app = angular.module('demo', []);
app.controller('myCtrl', function($scope) {
$scope.names = ["Deep", "Great", "Hello"];
});
</script>
</body>

</html>
```

Output



10.5 Working with Radio Buttons

The <mat-radiobutton> is used for <input type="radio"> for enhance the material design-based styling. It provides the same functionality as the <input type="radio"> with Angular Material Design styling and animations.

Radio-Button Label

The radio-button label is provided as the content of the <mat-radiobutton> element. If you do not want the label to appear next to the radio button, you use the area-label to specify the appropriate label.

Radio Groups

The radio-button should be placed inside the <mat-radio-group> unless the DOM structure makes that impossible (e.g., the radio button inside the table cell) impossible. Different radio buttons inside a radio group will inherit the group name.

Use with @angular/forms

<mat-radio-group> is compatible with @angular/forms and support the Reactive Forms Module.

<Mat-radio-button> uses internal <input type = "radio"> to provide an accessible experience. This internal radio button receives focus and is automatically labelled the text content of <mat-radio-button> element.

Radio button groups must be given a meaningful label through an area-label or area-labelled by.

Example

app.module.ts

```
import { AppComponent } from 'app.component';
import {FormsModule, ReactiveFormsModule} from '@angular/forms';
import {BrowserAnimationsModule} from '@angular/platform-browser/animations';
import {MatRadioModule} from '@angular/material'
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    MatRadioModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.css.

```
.tp-radio-group {
  display: inline-flex;
  flex-direction: column;
}
.tp-radio-button {
  margin: 5px;
}
.tp-selected-value {
  margin: 15px 0;
}
```

app.component.ts

```
import { Validators } from "@angular/forms";
import { FormControl } from "@angular/forms";
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'materialApp';
  favoriteSeason: string;
  seasons: string[] = ['Winter', 'Spring', 'Summer', 'Autumn'];
}
```

Output

Winter
Spring
Summer
Autumn

Selected Season: Winter

Summary

- Angular provides the ngForm directive to create a form in the template. It acts as a container for form controls and handles form-related functionality.
- The ngModel directive enables two-way data binding, allowing form controls to be bound to component variables.

- Changes in the form control update the variable, and changes in the variable reflect in the form control.
- Angular provides various form controls such as input fields, checkboxes, radio buttons, and dropdown lists. These controls are bound to component variables using ngModel or formControl directives.
- FormGroup represents a group of form controls, while FormControl represents an individual form control. FormGroup enables organizing and validating related fields, while FormControl tracks the value and validation state of a single control.
- Angular offers built-in validators, including required, minLength, maxLength, pattern, and more. Validators are used to enforce validation rules and ensure the entered data meets specific criteria.
- Angular provides classes and directives to display validation feedback to users. These include ngModel, ngModelGroup, ngForm, and ngControl directives, along with CSS classes for styling.
- Angular allows handling form submission using the (ngSubmit) event. It triggers a method in the component when the form is submitted, enabling custom logic such as data processing, API calls, or navigation.

Keywords

ngForm: A directive used to create an Angular form and enable form-related functionality.

ngModel: A directive used for two-way data binding, allowing data to be bound between form controls and component variables.

ngSubmit: An event binding used to handle the submission of a form, typically triggered by a submit button or pressing the Enter key.

[checked]: An attribute used to bind the checked state of a checkbox to a property or expression in the component.

(change): An event binding used to capture the checkbox state change event and trigger a method or perform an action in response.

[disabled]: An attribute used to disable a checkbox, preventing user interaction with it.

Checkbox group: A collection of checkboxes that are associated using the same name attribute, allowing them to function as a group.

Self Assessment

1. In Angular, which directive is used to create a dropdown list?

A. ng-dropdown
B. ng-select
C. ng-options
D. ng-list

2. How can you bind data to a dropdown list in Angular?

A. Using the [value] attribute
B. Using the [ngModel] directive
C. Using the [selected] property
D. Using the [dropdown] attribute

3. Which event is triggered when the selected value in a dropdown list changes?

A. (onSelect)
B. (onChange)
C. (onValueChange)
D. (onSelectionChange)

4. How can you disable a dropdown list in Angular?

A. Using the [disabled] attribute
B. Using the [enabled] attribute
C. Using the (disabled) event
D. Using the (enabled) event

5. How can you set a default selected value in a dropdown list in Angular?

A. By setting the selected property of the option element
B. By using the [default] attribute
C. By using the (default) event
D. By setting the [(ngModel)] directive to the desired value

6. How can you populate a dropdown list with dynamic data in Angular?

A. Using the [data] attribute
B. Using the (populate) event
C. Using the *ngFor directive with option elements
D. Using the (dynamic) event

7. In Angular, which directive is used to create a checkbox?

A. ng-checkbox
B. ng-input
C. ng-checkbox-input
D. ng-model

8. How can you bind the state of a checkbox to a variable in Angular?

A. Using the [checked] attribute
B. Using the [(ngModel)] directive
C. Using the [state] property
D. Using the (onChange) event

9. Which event is triggered when the state of a checkbox changes in Angular?

A. (onCheck)
B. (onChange)
C. (onStateChange)
D. (onCheckboxChange)

10. How can you disable a checkbox in Angular?

A. Using the [disabled] attribute
B. Using the [enabled] attribute
C. Using the (disabled) event
D. Using the (enabled) event

11. How can you handle the state of a checkbox in Angular?

A. By using the (checked) event
B. By subscribing to the (valueChange) event of the [(ngModel)] directive
C. By using the (change) event
D. By assigning the state to a variable using [(ngModel)]

12. In Angular, which directive is used to create a radio button?

A. ng-radio
B. ng-input
C. ng-radio-input
D. ng-model

13. How can you bind the selected state of a radio button to a variable in Angular?

A. Using the [selected] attribute
B. Using the [(ngModel)] directive
C. Using the [state] property
D. Using the (onChange) event

14. Which event is triggered when a radio button is selected in Angular?

A. (onCheck)
B. (onChange)
C. (onStateChange)
D. (onRadioButtonChange)

15. How can you disable a radio button in Angular?

A. Using the [disabled] attribute
B. Using the [enabled] attribute
C. Using the (disabled) event
D. Using the (enabled) event

Answers for Self Assessment

1.	B	2.	B	3.	B	4.	A	5.	D
6.	C	7.	D	8.	B	9.	B	10.	A

11.	B	12.	D	13.	B	14.	B	15.	D

Review Questions

1. How to disable submit button in angular? Give example.
2. Explain how checkboxes work in Angular and how you can handle their state and events.
3. "Angular provides features and directives for handling checkboxes" Justify the statement with example.
4. Explain how dropdown lists work in Angular and how you can populate them.
5. How do you create a radio button in Angular and bind the selected state of a radio button to a variable in Angular?

Further Readings

Murray, N., Coury, F., Lerner, A., & Taborda, C. (2018). *Ng-book: The complete guide to Angular*. CreateSpace Independent Publishing Platform.

Wilken, J. (2018). *Angular in Action*. Simon and Schuster.

Hajian, M. (2019). *Progressive web apps with angular: create responsive, fast and reliable PWAs using angular*. Apress.

Seshadri, S. (2018). *Angular: Up and running: Learning angular, step by step*. " O'Reilly Media, Inc.".

Web Links

https://codegen.studio/1157/how-to-correctly-disable-submit-button-in-reactive-forms-in-angular/

http://www.angulartutorial.net/2017/10/disable-submit-button-until-all.html

https://material.angular.io/components/checkbox/overview

https://mdbootstrap.com/docs/angular/forms/checkbox/

Unit 11: Consuming HTTP Services

CONTENTS

Objectives

Introduction

11.1 Template Driven Form

11.2 Model Driven Form

11.3 Form Validation

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit, the student would be able to:

- learn basic concepts of consuming HTTP services.
- understand template drive form and model driven form.
- understand validation of form.

Introduction

We can post to it, retrieve external data with the aid of the HTTP Service. To use the http service, we must import the http module. To comprehend how to use the http service, let's look at an example.

We need to import the module into app.module.ts as shown below in order to begin using the http service.

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

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
```

```
    BrowserAnimationsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

If you look at the highlighted code, we have added the HttpModule to the imports array as well as importing it from @angular/http.

Use the http service now in the app.component.ts file.

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  constructor(private http: Http) { }
  ngOnInit() {
    this.http.get("http://jsonplaceholder.typicode.com/users").
    map((response) ⇒ response.json()).
    subscribe((data) ⇒ console.log(data))
  }
}
```

Let's examine the code that was highlighted earlier. To use the service, we must import http, which is accomplished as follows:

```
import { Http } from '@angular/http';
```

The private property http of type Http and the constructor are both generated in the class AppComponent. We must utilise the get API provided by http in the manner described below to retrieve the data.

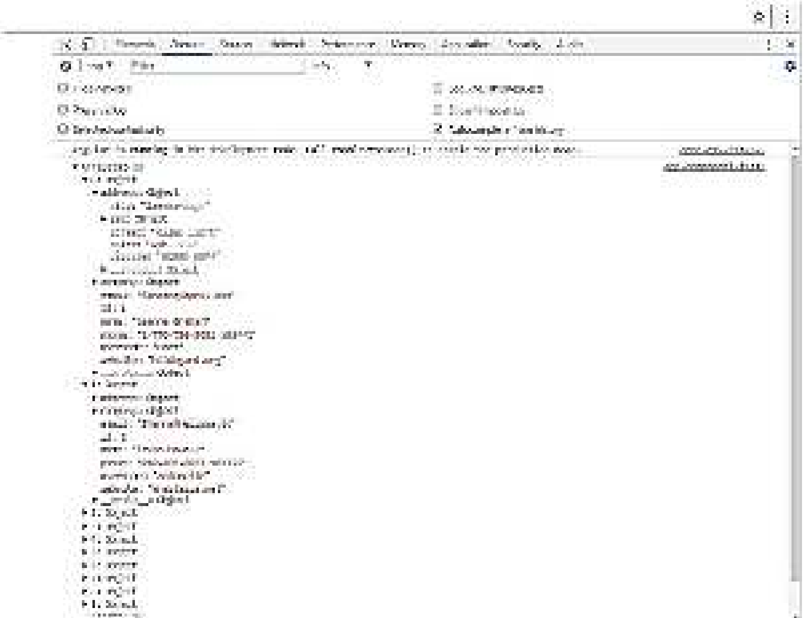
```
this.http.get();
```

As seen in the code, it requires the URL to be obtained as a parameter.

To fetch the json data, we will use the test address https://jsonplaceholder.typicode.com/users. On the retrieved url, two operations—data map and subscribe—are carried out. The data is converted to json format with the aid of the Map technique. We must import the map as shown below in order to use it.

```
import 'rxjs/add/operator/map';
```

Following completion of the map, the subscriber will record the output in the console for display in the browser.



The json objects are shown in the terminal, as you can see. The objects may also be seen in the browser.

Update the codes in app.component.html and app.component.ts as needed to display the objects in the browser:

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

import { Http } from '@angular/http';

import 'rxjs/add/operator/map';


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private http: Http) { }
  httpdata;
  ngOnInit() {
    this.http.get("http://jsonplaceholder.typicode.com/users").
    map(
      (response) ⇒ response.json()
    ).
    subscribe(
      (data) ⇒ {this.displaydata(data);}
    )
  }
  displaydata(data) {this.httpdata = data;}
}
```

We will execute the display data method in app.component.ts using the subscribe method and send the data we just fetched as an argument.

The information will be kept in the variable httpdata and used in the display data method. Using this httpdata variable, which is done in the app.component.html file, the data is displayed in the browser.

```
<ul *ngFor = "let data of httpdata">
<li>Name : {{data.name}} Address: {{data.address.city}}</li>
</ul>
```

The json object is as follows –

```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",

  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },

  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
    "bs": "harness real-time e-markets"
  }
}
```

The object has attributes like id, name, username, email, address, which internally contains street, city, etc., as well as information about phone, website, and company. The name and the city information from the app.component.html file will be shown in the browser using the for loop.

The display in the browser looks like this:

- Name : Leanne Graham Address: Gwenborough
- Name : Ervin Howell Address: Wisokyburgh
- Name : Clementine Bauch Address: McKenziehaven
- Name : Patricia Lebsack Address: South Elvis
- Name : Chelsey Dietrich Address: Roscoeview
- Name : Mrs. Dennis Schulist Address: South Christy
- Name : Kurtis Weissnat Address: Howemouth
- Name : Nicholas Runolfsdottir V Address: Aliyaview
- Name : Glenna Reichert Address: Bartholomebury
- Name : Clementina DuBuque Address: Lebsackbury

Let's now include the search parameter, which will perform a data-based filter. Based on the search parameter that was supplied, we must retrieve the data.

The files app.component.html and app.component.ts have undergone the changes listed below:

app.component.ts

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
  searchparam = 2;
  jsondata;
  name;
  constructor(private http: Http) { }
  ngOnInit() {
    this.http.get("http://jsonplaceholder.typicode.com/users?id="+this.searchparam).
    map(
      (response) ⇒ response.json()
    ).
    subscribe((data) ⇒ this.converttoarray(data))
  }
  converttoarray(data) {
    console.log(data);
    this.name = data[0].name;
```

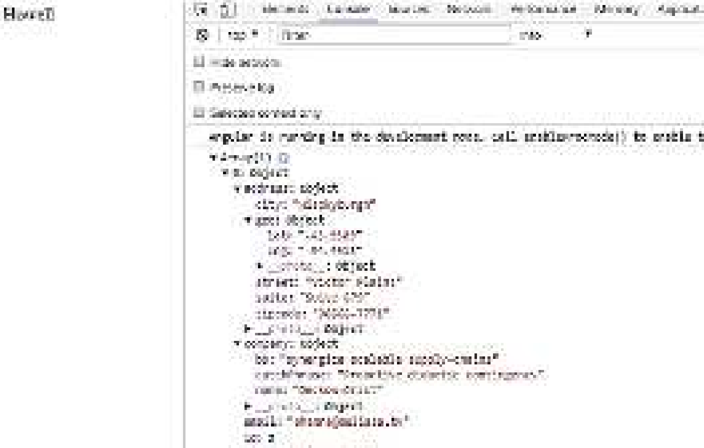
```
  }
}
```

We will add the search param id = this to the get api.searchparam. The value of the searchparam is 2. We require information about id=2 from the json file.

app.component.html

{{name}}

This is how the browser is displayed –



The data that the browser receives from the http has been consoled. The browser console shows the same information. The browser shows the name from the json file with id=2.

11.1 Template Driven Form

In a form driven by a template, the bulk of the work is completed in the template; in a form driven by a model, the majority of the work is completed in the component class.

Now let's think about working on the form powered by templates. We'll make a straightforward login form, add the email address and password, and include a submit button. Initially, we must import @angular/core into FormsModule, which is done in app.module.ts as follows:

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

import { HttpModule } from '@angular/http';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

import { MyserviceService } from './myservice.service';
import { NewCmpComponent } from './new-cmp/new-cmp.component';
import { ChangeTextDirective } from './change-text.directive';
import { SqrtPipe } from './app.sqrt';
```

```
@NgModule({
  declarations: [
    SqrtPipe,
    AppComponent,
    NewCmpComponent,
    ChangeTextDirective
  ],
  imports: [
    BrowserModule,
    HttpModule,
    FormsModule,
    RouterModule.forRoot([
      {path: 'new-cmp',component: NewCmpComponent}
    ])
  ],
  providers: [MyserviceService],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

As can be seen in the highlighted code, we imported the FormsModule into app.module.ts and added it to the imports array.

Let's now modify the app.component.html file to create our form.

```
<form #userlogin = "ngForm" (ngSubmit) = "onClickSubmit(userlogin.value)" >
  <input type = "text" name = "emailid" placeholder = "emailid" ngModel>
  <br/>
  <input type = "password" name = "passwd" placeholder = "passwd" ngModel>
  <br/>
  <input type = "submit" value = "submit">
</form
```

We have made a straightforward form with submit, email, and password input tags. It has a type, name, and placeholder given to it.

The ngModel directive and name attribute must be included in order to build the model form controls in template-driven forms. As a result, add ngModel to the tag as indicated above anywhere we want Angular to retrieve our data through forms. Now, we need to add the ngModel across it if we need to read the emailid and passwd.

You'll see that we've also included the ngForm in the #userlogin. The form template we made has to have the ngForm directive applied to it. Additionally, we added the function onClickSubmit and gave it the parameter userlogin.value.

Now let's build the method in the app.component.ts file and retrieve the form's input values.

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

import { MyserviceService } from './myservice.service';
```

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

export class AppComponent {
   title = 'Angular 4 Project!';
   todaydate;
   componentproperty;
   constructor(private myservice: MyserviceService) { }
   ngOnInit() {
      this.todaydate = this.myservice.showTodayDate();
   }
   onClickSubmit(data) {
      alert("Entered Email id : " + data.emailid);
   }
}
```

The function onClickSubmit has been defined in the aforementioned app.component.ts file. The control will go to the aforementioned function after you press the form's submit button.

This is the representation of the browser.

The form appears as it is depicted below. Let's fill it out with the information. The email address is already entered in the submit function.

11.2 Model Driven Form

We must use the Reactive Forms Module from @angular/forms in the imports array and import it into the model-driven form.

A change has been made, and it appears in app.module.ts.

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

import { HttpModule } from '@angular/http';
import { ReactiveFormsModule } from '@angular/forms';
import { AppComponent } from './app.component';

import { MyserviceService } from './myservice.service';
import { NewCmpComponent } from './new-cmp/new-cmp.component';
import { ChangeTextDirective } from './change-text.directive';
import { SqrtPipe } from './app.sqrt';

@NgModule({
  declarations: [
    SqrtPipe,
    AppComponent,
    NewCmpComponent,
    ChangeTextDirective
  ],
  imports: [
    BrowserModule,
    HttpModule,
    ReactiveFormsModule,
    RouterModule.forRoot([
      {
        path: 'new-cmp',
        component: NewCmpComponent
      }
    ])
  ],
  providers: [MyserviceService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

We must import a few modules into app.component.ts for the model-driven form. Import FormGroup and FormControl, for instance, from "@angular/forms."

```
import { Component } from '@angular/core';
import { MyserviceService } from './myservice.service';
import { FormGroup, FormControl } from '@angular/forms';
```

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Angular 4 Project!';
  todaydate;
  componentproperty;
  emailid;
  formdata;
  constructor(private myservice: MyserviceService) { }
  ngOnInit() {
    this.todaydate = this.myservice.showTodayDate();
    this.formdata = new FormGroup({
      emailid: new FormControl("angular@gmail.com"),
      passwd: new FormControl("abcd1234")
    });
  }
  onClickSubmit(data) {this.emailid = data.emailid;}
}
```

FormGroup is initialised along with the variable formdata as shown above at the beginning of the class. Emailid and Password are initialised with default values so that they may be seen in the form. If you want, you can leave it empty.

The values will appear in the form's UI like this.

Email entered is :

We used formdata to initialise the form values; in the UI form component.html, we also need to utilise formdata.

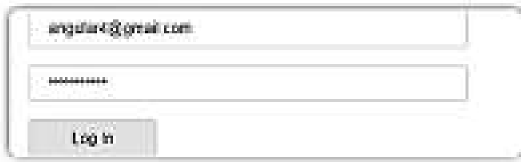
```
<div>
  <form [formGroup]="formdata" (ngSubmit) = "onClickSubmit(formdata.value)" >
    <input type="text" class="fortextbox" name="emailid" placeholder="emailid"
      formControlName="emailid">
    <br/>

    <input type="password" class="fortextbox" name="passwd"
      placeholder="passwd" formControlName="passwd">
```

```
    <br/>


    <input type="submit" class="forsubmit" value="Log In">
  </form>
</div>
<p>
  Email entered is : {{emailid}}
</p>
```

For the form in the.html file, we have utilised the formGroup square bracket; for instance, [formGroup]="formdata". When submitting, the formdata.value parameter is sent to the method onClickSubmit.

The formControlName input tag is employed. A value that we utilised in the app.component.ts file is provided for it.

The function onClickSubmit, which is specified in the app.component.ts file, will receive control when the submit button is pressed.



Email entered is : angular4@gmail.com

On clicking Login, the value will be displayed as shown in the above screenshot.

11.3 Form Validation

Now let's talk about model-driven form validation. You have the option of using the built-in form validation or a custom validation method. Both methods will be applied in the form. The identical example that we developed in one of our earlier sections will serve as our guide moving forward. As seen below, in order to use Angular 4, we must import Validators from @angular/forms.

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

Validators like mandatory field, minlength, maxlength, and pattern are already included with Angular. These can be accessible by using the module for validators.

To inform Angular if a given field is essential, you can simply add validators or an array of validators.

Let us now try the same on one of the input textboxes, i.e., email id. For the email id, we have added the following validation parameters −

- Required
- Pattern matching

This is how a code undergoes validation in app.component.ts.

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


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

```
})
export class AppComponent {
  title = 'Angular 4 Project!';
  todaydate;
  componentproperty;
  emailid;
  formdata;
  ngOnInit() {
    this.formdata = new FormGroup({
      emailid: new FormControl("", Validators.compose([
        Validators.required,
        Validators.pattern("[^ @]*@[^ @]*")
      ])),
      passwd: new FormControl("")
    });
  }
  onClickSubmit(data) {this.emailid = data.emailid;}
}
```

The list of things you wish to validate on the input field can be added to Validators.compose. To ensure that we only accept authentic emails, we now included the needed and pattern matching parameters.

If any of the form inputs are invalid, the submit button in the app.component.html is disabled. The procedure is as follows.

```
<div>
  <form [formGroup] = "formdata" (ngSubmit) = "onClickSubmit(formdata.value)" >
    <input type = "text" class = "fortextbox" name = "emailid" placeholder = "emailid"
      formControlName = "emailid">
    <br/>
    <input type = "password" class = "fortextbox" name = "passwd"
      placeholder = "passwd" formControlName = "passwd">
    <br/>
    <input type = "submit" [disabled] = "!formdata.valid" class = "forsubmit"
      value = "Log In">
  </form>
</div>


<p>
  Email entered is : {{emailid}}
</p>
```

We added disabled in a square bracket with the specified value, "!formdata.valid," for the submit button. As a result, the button will remain blocked and the user won't be able to submit the form if the formdata.valid is invalid.

Let's check how this functions in the browser.

Email entered is :

The login button is disabled in the example above because the email address given is incorrect. Now let's try entering a legitimate email address to compare the results.

Email entered is : sites@gmail.com

The email address you entered is now valid. As a result, we can see that the login button is active and that the user can submit it. With this, the entered email address is shown at the bottom.

Let's now use the same form to test custom validation. We can create our own custom function and include the necessary information in it to do custom validation. Now, we'll see an illustration of the same.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'Angular 4 Project!';
  todaydate;
  componentproperty;
  emailid;
  formdata;
  ngOnInit() {
    this.formdata = new FormGroup({
      emailid: new FormControl("", Validators.compose([
        Validators.required,
        Validators.pattern("[^ @]*@[^ @]*")
      ])),
      passwd: new FormControl("", this.passwordvalidation)
    });
```

```
}
passwordvalidation(formcontrol) {
  if (formcontrol.value.length <'; 5) {
    return {"passwd" : true};
  }
}
onClickSubmit(data) {this.emailid = data.emailid;}
}
```

In the example above, we constructed a function for password validation, and the formcontrol passwd: new FormControl("", this.passwordvalidation) uses it in the previous section.

We will determine whether the length of the characters entered is appropriate using the function we have built. Return "passwd": true; will be used if the characters are less than five, as indicated above. More than five characters will be deemed legitimate, and the login will then be available.

Now, let's examine how this appears in the browser.

The login is disabled because we only put three characters in the password. We require more than five characters to enable login. Now let's check by entering a character length that is acceptable.

The login is permitted because the password and email address are both legitimate. As we log in, the email is shown at the bottom.

Summary

Angular Framework: A popular open-source JavaScript framework developed by Google for building web applications. It provides a structured architecture, powerful tools, and a set of libraries to simplify frontend development.

Components: Modular building blocks in Angular that encapsulate a part of the user interface, along with its logic and styling. They are the core elements of Angular applications.

Templates: HTML-based views in Angular that define how the user interface should be rendered. They can include data binding, directives, and other Angular-specific features.

Directives: Angular constructs that allow you to extend HTML with custom behaviors and functionalities. Examples include ngIf, ngFor, and ngStyle.

Data Binding: A feature that allows you to synchronize data between the component and the view. Angular supports one-way and two-way data binding.

Services: Reusable components in Angular that provide specific functionality, such as data fetching, authentication, or logging. They promote separation of concerns and are often used to share data and functionality across components.

Dependency Injection: A design pattern used in Angular to provide and manage the dependencies of components and services. It helps in creating more modular and testable code.

Modules: Containers that group related components, directives, services, and other parts of an Angular application. They help organize and manage the application's structure.

Routing: The capability to navigate between different views or components in an Angular application. It's essential for building single-page applications (SPAs).

Forms: Angular provides a powerful form handling mechanism with features like form controls, validation, and form groups. It simplifies capturing and validating user input.

Observables: A way to handle asynchronous data streams in Angular. Observables are used extensively for handling events, asynchronous operations, and data synchronization.

Pipes: Angular pipes allow you to transform data in the view. They can be used for formatting dates, numbers, and other values.

Lifecycle Hooks: Methods that Angular components implement to respond to different lifecycle events, such as initialization, changes, and destruction.

Testing: Angular provides tools and libraries for writing unit tests, integration tests, and end-to-end tests to ensure the quality and reliability of your application.

Angular CLI: The Angular Command Line Interface is a powerful tool that simplifies tasks such as creating, building, testing, and deploying Angular applications.

Angular Material: A UI component library developed by Google that follows the Material Design guidelines. It provides pre-styled components for building responsive and attractive user interfaces.

Ivy: Angular's next-generation compilation and rendering pipeline that aims to improve performance, bundle size, and overall developer experience.

TypeScript: The programming language used in Angular development. It's a superset of JavaScript that adds static typing and other features to enhance code quality and maintainability.

Lazy Loading: A technique in Angular where modules are loaded on-demand, improving the initial loading time of an application by only loading what is needed when it's needed.

Angular Universal: A framework for rendering Angular applications on the server side, which can improve performance, SEO, and user experience.

Keywords

HTTP services: These are communication channels established over the Hypertext Transfer Protocol (HTTP) that allow applications to interact with each other over the internet. They facilitate the exchange of data between a client (requesting entity) and a server (providing entity).

HTTP methods: Also known as HTTP verbs, these define the type of action a client wants to perform on a resource. Common methods include GET (retrieve data), POST (send data), PUT (update data), and DELETE (remove data).

HTTP requests: These are messages sent by a client to a server, typically to request a specific action or retrieve data. Requests include information like the HTTP method, headers, and optional data payload.

HTTP responses: After receiving a request, a server sends back an HTTP response. This includes a status code indicating the outcome of the request (success, error, redirection, etc.), headers containing metadata, and potentially a response body with requested data.

Authentication: The process of verifying the identity of a user or system trying to access a resource. It ensures that only authorized parties can access specific functionalities or data.

Authorization: Once a user or system is authenticated, authorization determines the level of access they have to certain resources. It specifies what actions a user is allowed to perform based on their role or permissions.

Query parameters: These are key-value pairs included in a URL's query string. They provide additional information to the server about the request, helping to filter, sort, or customize the response data.

Headers: These are metadata elements sent in both requests and responses. They convey information about the request/response, such as content type, authentication tokens, or caching directives.

RESTful APIs: Representational State Transfer (REST) APIs adhere to a set of architectural principles for designing networked applications. They use standard HTTP methods and responses to create a consistent and scalable way to interact with resources.

Error handling: This involves strategies and techniques to manage and communicate errors ***that*** occur during the interaction between client and server. Proper error handling enhances user experience and helps diagnose issues.

Status codes: These three-digit numbers are included in HTTP responses to indicate the outcome of a request. They provide information about whether the request was successful, encountered an error, or requires further action.

Best practices: These are established guidelines and recommendations that help developers write efficient, maintainable, and secure code when working with HTTP services, methods, and APIs.

Libraries: Software libraries are pre-written code components that provide reusable functions and routines to simplify development. In the context of HTTP, libraries can offer tools to handle requests, responses, and other related tasks.

Frameworks: Frameworks provide a more comprehensive structure for building applications. In the realm of HTTP, frameworks can offer predefined architectures and components to streamline the process of consuming and providing web services.

Self Assessment

1. What is the primary protocol used for communication between web clients and servers?
A. HTML
B. CSS
C. HTTP
D. JSON

2. Which HTTP method is typically used to retrieve data from a server?
A. GET
B. POST
C. PUT
D. DELETE

3. In Angular, what is responsible for sending HTTP requests to a server?

A. Components
B. Services
C. Directives
D. Modules

4. Which Angular module is used for making HTTP requests and handling responses?

A. HttpClientModule
B. HttpModule

C. HttpServiceModule

D. RestModule

5. Which keyword in Angular is used to handle asynchronous operations and manage data streams? A. async

B. promise

C. subscribe

D. await

6. Which Angular directive is used to conditionally display elements based on a certain condition?

A. ngIf

B. ngFor

C. ngSwitch

D. ngWhile

7. What is the purpose of query parameters in HTTP requests?

A. To specify the HTTP method

B. To include authentication credentials

C. To customize the request URL

D. To define the response format

8. Which HTTP status code indicates that a resource was successfully created on the server?

A. 200 OK

B. 201 Created

C. 204 No Content

D. 400 Bad Request

9. In Angular, which module is responsible for managing application routes and navigation?

A. RouteModule

B. NavigationModule

C. RouterModule

D. AngularRouteModule

10. Which Angular CLI command is used to generate a new service?

A. ng create service

B. ng make service

C. ng generate service

D. ng build service

11. In Angular, which module is responsible for making HTTP requests and handling responses?

A. @angular/http
B. @angular/core
C. @angular/httpclient
D. @angular/net

12. Which HTTP method is typically used to update existing data on the server in Angular?

A. POST
B. DELETE
C. PUT
D. PATCH

13. What is the role of the "HttpClient" service in Angular when consuming HTTP services?

A. To manage component lifecycle
B. To define routing configuration
C. To make HTTP requests and handle responses
D. To create and manage services

14. Which operator is commonly used to handle errors when subscribing to an Observable for an HTTP request?

A. map()
B. catchError()
C. switchMap()
D. filter()

15. In Angular, where are HTTP headers typically configured when making an HTTP request?

A. In the component template
B. In the service provider
C. In the App Module
D. In the Http Client Module configuration

Answers for Self Assessment

1.	C	2.	A	3.	B	4.	A	5.	C
6.	A	7.	C	8.	B	9.	C	10.	C
11.	A	12.	C	13.	C	14.	B	15.	B

Review Questions

1. How do you delete a post using JSON Place holder's API? Explain the required HTTP method and the process.

2. How would you create a new post using JSON Place holder's API? Describe with the help of necessary steps.
3. What HTTP methods (verbs) can you use to interact with JSON Place holder's API, and what are their corresponding CRUD.
4. If you want to retrieve a specific post by its ID, what would be the format of the API endpoint? Provide an example.
5. How can you access JSON Place holder's API? What is the base URL for making requests to JSON Place holder?

Further Readings

Official Angular Documentation:

Angular HTTP Guide: https://angular.io/guide/http

Angular HttpClient API: https://angular.io/api/common/http/HttpClient

Online Tutorials and Courses:

Udemy: "Angular - The Complete Guide" by Maximilian Schwarzmüller

Pluralsight: "Angular HTTP Communication" by Deborah Kurata

Coursera: "Single Page Web Applications with Angular" by Yaakov Chaikin

Blogs and Articles:

"HTTP Client in Angular" by Todd Palmer: https://toddpalmer.com/http-client-in-angular/

"Angular HTTP Client – Quickstart Guide" by Thoughtram: https://blog.thoughtram.io/angular/2016/11/28/testing-services-with-http-in-angular-2.html

Web Links

The Net Ninja: https://www.youtube.com/thenetninja (Check for Angular HTTP-related tutorials)

Academind: https://www.youtube.com/academind (Look for Angular HTTP tutorials)

GitHub Repositories and Examples:

Angular HttpClient Example: https://github.com/angular/angular/tree/main/packages/common/http/test

Angular CRUD Example: https://github.com/cornflourblue/angular-8-crud-example

Angular Community Forums:

Stack Overflow (Angular tag): https://stackoverflow.com/questions/tagged/angular

Angular Reddit Community: https://www.reddit.com/r/Angular2/

Angular CLI Documentation:

Angular CLI: https://angular.io/cli

Generate HTTP Client using Angular CLI: https://angular.io/cli/generate#service

Unit 12: Routing and Navigation

CONTENTS

Objectives

Introduction

12.1 Routing in a Nutshell

12.2 Configuring Routes

12.3 Router Outlet

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit the students will learn:

- To Understand the basic concept of routing and Navigation
- The methods to configure Routes

Introduction

Routing and navigation are essential aspects of building modern web applications, including those developed using AngularJS. AngularJS is a JavaScript-based framework that allows you to create dynamic and interactive single-page applications (SPAs). Routing and navigation in AngularJS enable you to build multi-page experiences within a single-page application by dynamically loading different views or components based on the URL.

Why Routing and Navigation?

In a typical web application, clicking on links or typing URLs in the browser's address bar reloads the entire page. In contrast, SPAs like AngularJS apps aim to provide a smoother and more responsive user experience by loading only the necessary content without full page reloads.

Routing enables you to map different URLs to specific views or components, allowing users to navigate through your application seamlessly.

12.1 Routing in a Nutshell

Routing in AngularJS, in a nutshell, involves configuring your application to navigate between different views or pages within a single-page application (SPA) based on the URL. Here's a concise overview of routing in AngularJS:

ngRoute Module: AngularJS provides the ngRoute module to implement routing. You need to include this module in your application to enable routing features.

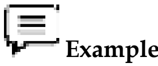
Configuring Routes: Define routes by configuring the $routeProvider service in your AngularJS application's module configuration. You specify the URL, controller, and template/view associated with each route.

```
angular.module('myApp', ['ngRoute'])
.config(function($routeProvider) {
  $routeProvider
    .when('/home', {
      templateUrl: 'partials/home.html',
      controller: 'HomeController'
    })
    .when('/about', {
      templateUrl: 'partials/about.html',
      controller: 'AboutController'
    })
    .otherwise({
      redirectTo: '/home'
    });
});
```

Creating Views: Views in AngularJS are typically defined using HTML templates. These templates can contain placeholders for dynamic content.

```
<!-- partials/home.html -->
<div ng-controller="HomeController">
  <h1>Welcome to the Home Page</h1>
  <!-- Other content goes here -->
</div>
```

Navigating Between Routes: Use the ng-href or ng-click directives to create links and navigation elements in your views.



Example

```
<!-- Example navigation links -->
<a ng-href="#/home">Home</a>
<a ng-href="#/about">About</a>
```

Controller and Scope: Each route can have an associated controller. Controllers manage the logic and data for a specific view. The scope of the controller is typically limited to the view it controls.

Route Parameters: You can define route parameters to make your routes more dynamic. For example, a route like /user/:id can capture different user IDs, allowing you to display user-specific content.

Dependency Injection: AngularJS uses dependency injection to provide components like controllers, services, and factories to your application. This promotes modularity and testability.

Handling Route Changes: You can listen for route changes and perform actions when a route changes. This can be useful for tasks like loading data for a specific route.

Nested Routes: AngularJS supports nested routes, allowing you to create complex application structures with parent and child views.

12.2 Configuring Routes

Configuring routes in AngularJS involves setting up route definitions to map specific URLs to views and controllers within your application. Here are the steps to configure routes in an AngularJS application:

Include the ngRoute Module: First, make sure you include the ngRoute module in your application. You can either download it and include it as a script tag or use a package manager like npm or Bower to install it.

Example

```
<script src="angular.js"></script>

<script src="angular-route.js"></script>
```

Define the Main Application Module: Create your main application module and include 'ngRoute' as a dependency.

```
var myApp = angular.module('myApp', ['ngRoute']);
```

Configure Routes: Inside the application module's configuration block, use the $routeProvider service to define routes. Each route is configured using the .when() method. You specify the URL, template or templateUrl, and controller for each route.

Example

```
myApp.config(function($routeProvider) {
 $routeProvider
  .when('/home', {
   templateUrl: 'partials/home.html',
   controller: 'HomeController'
  })
  .when('/about', {
   templateUrl: 'partials/about.html',
   controller: 'AboutController'
  })
  .when('/contact', {
   templateUrl: 'partials/contact.html',
   controller: 'ContactController'
  })
  .otherwise({
   redirectTo: '/home'
  });
});
```

In this example:

/home, /about, and /contact are the URLs that will trigger different views.

templateUrl or template specifies the HTML template associated with each view.

controller specifies the controller to be used for each view.

The otherwise method specifies the default route to redirect to if an unknown URL is provided.

Create Views: Create HTML templates for each view and save them in a directory (e.g., partials). These templates will be loaded when the corresponding route is activated.

```
<!-- partials/home.html -->
<div ng-controller="HomeController">
  <h1>Welcome to the Home Page</h1>
  <!-- Other content goes here -->
</div>
```

Create Controllers: Define controllers for each view. These controllers will contain the logic and data associated with their respective views.

```
myApp.controller('HomeController', function($scope) {
  // Controller logic for the Home view
});
myApp.controller('AboutController', function($scope) {
  // Controller logic for the About view
});
myApp.controller('ContactController', function($scope) {
  // Controller logic for the Contact view
});
```

Set up Navigation: Create navigation links or buttons in your views to navigate between different routes. Use the ng-href or ng-click directives to handle navigation.

```
<a ng-href="#/home">Home</a>
<a ng-href="#/about">About</a>
<a ng-href="#/contact">Contact</a>
```

Handle Route Changes (Optional): If you need to perform specific actions when a route changes (e.g., loading data), you can use the $route service or listen for route change events provided by AngularJS.

That's a basic overview of how to configure routes in an AngularJS application. With this setup, your application will be able to navigate between different views or pages based on the URL, providing a seamless user experience within a single-page application.

12.3 Router Outlet

In Angular, routerOutlet is a directive used in conjunction with the Angular Router module to define where the routed components should be displayed within the application's layout. It's part of the Angular Router module and is used for creating dynamic views or content based on the current route.

Working of routerOutlet

Router Configuration: First, you define your application's routes in the Angular Router configuration. This is typically done in the app-routing.module.ts file using the RouterModule.forRoot() method.

```
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';
const routes: Routes = [
  { path: '', component: HomeComponent },
```

```
  { path: 'about', component: AboutComponent },
  // Other route configurations...
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

HTML Template: In your application's HTML template, you use the <router-outlet></router-outlet> element to specify where the content of the currently active route should be displayed.

```
<div>
  <!-- Other fixed content of your layout -->
  <router-outlet></router-outlet>
</div>
```

The <router-outlet></router-outlet> tag acts as a placeholder. When you navigate to a route defined in your router configuration, Angular will dynamically load and display the associated component's template inside this placeholder.

Navigating between Routes: To navigate between different routes, you typically use the routerLink directive on anchor tags (<a>) or programmatic navigation using the Router service.

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

Or in your component code:

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

// ...

constructor(private router: Router) { }

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

When a user clicks a link with a routerLink attribute or you programmatically navigate to a route using the Router, Angular will load the associated component and render it inside the <router-outlet></router-outlet> element. This approach enables you to create single-page applications with dynamic views based on the current route, providing a smooth and interactive user experience.

Summary

- Routing and navigation are essential concepts in Angular for building single-page applications (SPAs).
- Angular's RouterModule is a module that provides routing and navigation capabilities.
- The <router-outlet></router-outlet> directive is used as a placeholder in your application's layout to determine where the content of the currently active route should be displayed.

- routerLink directive is used to generates the appropriate URL based on the route configuration.
- Route guards are used to protect routes and control access based on certain conditions.
- Each route object specifies a path (URL segment) and the associated component to be displayed when that path is navigated to.
- Route parameters allow you to pass data to a component when the route is activated.

Keywords

RouterModule: Angular module for setting up routing.

Route Configuration: Defining routes using path and component mappings.

Router Outlet: Placeholder for displaying routed components.

Route Paths: URL segments mapped to components.

RouterLink: Directive for generating navigation links.

Programmatic Navigation: Navigating to routes using the Router service.

Route Parameters: Dynamic parts of URL segments.

Child Routes: Nested routes within other routes.

Route Guards: Mechanisms for protecting routes (e.g., CanActivate, CanDeactivate).

Lazy Loading: Loading modules and routes on-demand for improved performance.

Route Resolvers: Pre-fetching data before route activation.

Wildcards and Redirects: Handling unmatched routes or configuring redirects.

Navigation Events: Events and hooks for observing and responding to navigation.

Router State: Accessing information about the current route and parameters.

Route Data: Additional metadata associated with a route.

Self Assessment

1. What is the primary role of RouterModule in Angular routing?

A. To define the application's layout structure
B. To set up and configure routes
C. To create navigation links
D. To handle HTTP requests

2. Which directive is used to specify the location where routed components should be displayed?

A. <app-router>
B. <router-outlet>
C. <route-view>
D. <component-target>

3. What does a route parameter in Angular allow you to do?

A. Protect a route from unauthorized access
B. Define child routes
C. Pass data to a component based on the URL
D. Specify a default route

4. In Angular, which guard can be used to restrict access to a specific route?
A. CanActivate
B. HttpClientGuard
C. RouteResolver
D. RouteProtect

5. What does lazy loading in Angular routing refer to?
A. Delaying the initial rendering of the application
B. Loading modules and routes only when they are needed
C. Loading all routes when the application starts
D. Reducing the application's performance

6. What is the primary purpose of the route configuration in Angular?
A. To define the application's layout structure
B. To specify the application's root component
C. To configure the mapping between routes and components
D. To set up global styles and themes

7. Which Angular module is essential for configuring routes in an Angular application?
A. HttpClientModule
B. RouterModule
C. FormsModule
D. BrowserAnimationsModule

8. How do you define a route path for a component in Angular?
A. Using the @RoutePath decorator
B. In the component's constructor
C. In the template HTML file
D. In the route configuration

9. What is a route parameter in Angular?
A. A variable that holds the current route's path
B. A dynamic part of a URL that can be extracted and used in a component
C. An optional query parameter for navigation
D. A parameter passed to the routerLink directive

10. Which directive is used to define a navigation link in an Angular template?
A. <router-outlet>
B. <a>
C. <router-link>
D. <navigate>

11. What is the primary purpose of <router-outlet> in Angular?
A. To define the application's layout structure

B. To configure route parameters
C. To display the content of the currently active route
D. To create navigation links
12. In an Angular application, where is <router-outlet> typically placed?
A. In the component's constructor
B. In the root module's configuration
C. In the application's main HTML template
D. In the routing module

13. How is the content displayed within <router-outlet> determined?
A. It displays all components at once.
B. It displays the component specified in the route configuration.
C. It displays the component that was initialized first.
D. It displays the component with the highest priority.

14. What directive is used to generate navigation links to different routes in Angular templates?
A. <a>
B. <router-link>
C. <navigate>
D. <route-outlet>
15. How can you configure multiple <router-outlet>s in an Angular application?
A. By using named outlets in the route configuration
B. By nesting <router-outlet> elements
C. By adding multiple router modules
D. By creating a separate outlet component

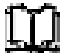
Answers for Self Assessment

1.	B	2.	B	3.	C	4.	A	5.	B
6.	C	7.	B	8.	D	9.	B	10.	B
11.	C	12.	C	13.	B	14.	A	15.	A

Review Questions

1. Describe in detail the purpose and significance of the Router Module in Angular's routing system.
2. Discuss the concept of the <router-outlet> directive in Angular. Explain how it works to display components based on the current route with example.
3. Explain the importance of route parameters in Angular routing and how they allow passing data within a route.
4. Discuss the concept of route guards and provide examples of scenarios where Can Activate and Can Deactivate guards might be used.
5. Explain the concept of location strategies in Angular routing, including Hash Location Strategy and Path Location Strategy.

6. Discuss the usage of the router Link directive for generating links and the programmatic navigation approach using the Router service.



Further Readings

Savkin, V. (2017). Angular router. Packt Publishing Ltd.

Goel, N. (2019). Step-by-Step Angular Routing: Learn To Create client-side and Single Page Apps with Routing and Navigation. BPB Publications.

Moiseev, A., & Fain, Y. (2018). Angular Development with TypeScript. Simon and Schuster.

Noring, C., & Deeleman, P. (2017). Learning Angular-: A no-nonsense guide to building real-world apps with Angular 5. Packt Publishing.

Nayrolles, M., Gunasundaram, R., & Rao, S. (2017). Expert Angular. Packt Publishing Ltd.

Web Links

https://angular.io/guide/routing-overview

https://angular.io/tutorial/tour-of-heroes/toh-pt5

https://www.smashingmagazine.com/2018/11/a-complete-guide-to-routing-in-angular/

https://www.geeksforgeeks.org/routing-in-angular-9-10/

https://www.tutorialspoint.com/angular8/angular8_routing_and_navigation.htm

Unit 13: Authentication and Authorization

CONTENTS

Objectives

Introduction

13.1 Application Overview

13.2 Architecture

13.3 JSON Web Tokens

13.4 Implementing Login

13.5 Implementing Logout

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After this unit you will be able to:

- Learn the concept of authorization and authentication.
- Understand the architecture of authorization and authentication.
- Implement the login and logout forms.

Introduction

Authentication is the process of verifying the identity of a user, typically using credentials such as a username and password. In AngularJS, authentication involves confirming that a user is who they claim to be before granting access to protected parts of a web application. AngularJS provides the tools and mechanisms to implement user authentication, including the creation of login forms, capturing user input, and sending authentication requests to the server. Upon successful authentication, the server typically responds with an authentication token, such as a JSON Web Token (JWT). This token is then stored securely on the client-side (e.g., in an HTTP-only cookie or local storage) and sent with each subsequent HTTP request to identify the user.

Authentication in AngularJS can be further enhanced with features like token expiration, token refresh, and integration with third-party authentication providers, such as OAuth or social logins.

Authorization, on the other hand, is the process of determining what actions or resources a user is allowed to access after they have been authenticated. It controls permissions, roles, and access rights to specific features or sections of a web application. AngularJS provides several mechanisms for implementing authorization, including route guards and custom directives. Route guards, such as canActivate, enable developers to protect specific routes from unauthorized access. Custom directives can conditionally display or hide elements in the user interface based on the user's authorization level.

Authorization can be role-based or permission-based. Role-based authorization assigns roles (e.g., admin, user) to users, and access to certain features is determined by the user's role. Permission-

based authorization grants specific permissions to users, allowing fine-grained control over who can perform specific actions or access particular resources.

13.1 Application Overview

Authentication and authorization are essential components of web applications, including those built with AngularJS, which is a JavaScript framework for building web applications. In AngularJS, you can implement authentication and authorization using various techniques and libraries. Here's an overview of how you can approach authentication and authorization in an AngularJS application:

Authentication: Authentication is the process of verifying the identity of a user. In AngularJS, you can implement authentication using techniques such as:

Token-based authentication: This is a popular method where a user logs in with their credentials (username/password) and receives a token (usually a JSON Web Token or JWT) upon successful authentication. This token is then included in subsequent requests to prove the user's identity.

Session-based authentication: In this approach, a user session is created on the server upon successful login, and a session identifier is stored on the client side (usually in a cookie). The server validates requests based on this session identifier.

OAuth and OAuth2: If your application integrates with external identity providers like Google or Facebook, you can use OAuth or OAuth2 for authentication.

Firebase Authentication: Firebase provides an authentication service that can be easily integrated into AngularJS applications.

Authorization: Authorization determines what actions a user is allowed to perform within the application after they have been authenticated. AngularJS typically handles authorization on the client-side, but it's important to enforce it on the server-side as well.

Role-based authorization: Assign roles (e.g., admin, user) to users, and restrict access to certain routes or features based on these roles. You can implement this using AngularJS routing and custom directives.

Permission-based authorization: Instead of relying solely on roles, you can assign specific permissions to users and check these permissions before allowing them to perform certain actions.

AngularJS Services and Modules: You can create AngularJS services and modules to encapsulate authentication and authorization logic.

Example, you might create an AuthService to handle login/logout and store user tokens, and an AuthGuard to protect routes based on user roles or permissions.

HTTP Interceptors: You can use AngularJS HTTP interceptors to automatically attach authentication tokens to outgoing HTTP requests. This ensures that authenticated users can access protected resources.

User Interface: Create a user-friendly login page and user profile page to manage authentication. Depending on the authentication method, you may use AngularJS forms for login and display user information in the UI.

Error Handling: Implement error handling for authentication and authorization failures, such as expired tokens or insufficient permissions.

Backend Integration: Ensure that your server-side code enforces authorization rules. Even if the client-side code restricts access to certain UI elements, the server should validate requests to prevent unauthorized actions.

Testing and Security: Thoroughly test your authentication and authorization flows to identify and fix vulnerabilities. Pay attention to security best practices, such as password hashing, token expiration, and secure communication (HTTPS).

13.2 Architecture

In an AngularJS application, implementing a secure authentication and authorization architecture involves a combination of client-side and server-side components. Below, I'll outline a high-level architecture for authentication and authorization in an AngularJS application:

1. Server-Side Authentication:

- User Database: Maintain a database or user store where user credentials and other relevant information are securely stored. Hash and salt passwords for security.
- Authentication API: Create an API endpoint that handles user authentication (e.g., login and logout). This API should validate user credentials and generate tokens upon successful login. Tokens are typically JSON Web Tokens (JWTs) that contain user information and are signed with a secret key.
- Token Management: Implement token expiration and refresh mechanisms. Tokens should have a limited lifespan to enhance security. Refresh tokens can be used to obtain new access tokens without requiring the user to log in again.

2. AngularJS Client-Side Implementation:

- Authentication Service: Create an AngularJS service (e.g., AuthService) responsible for managing user authentication. This service should include methods for login, logout, and checking the user's authentication status.
- Token Storage: Store the authentication token (JWT) securely on the client side, typically in a secure HTTP-only cookie or local storage. Ensure it's sent with each HTTP request as an authorization header.
- Authentication Guards: Implement route guards (e.g., AuthGuard) to protect specific routes from unauthorized access. These guards can check the user's authentication status and role/permissions before allowing or denying access to routes.
- Login Component: Create a login component for users to enter their credentials and initiate the authentication process. This component communicates with the AuthService to send login requests to the server.
- Logout Component/Functionality: Implement a logout component or functionality that communicates with the AuthService to log the user out and clear the stored token.
- User Profile/Settings: Create a user profile or settings component where authenticated users can manage their account details and preferences.

3. Authorization Rules: Define role-based or permission-based authorization rules on the server-side. These rules determine what actions and resources a user can access. Roles and permissions can be associated with users in the user database.

On the client side, use route guards or custom directives to enforce these rules.

Example, restrict access to admin-related routes or features based on the user's role.

4. Secure Communication: Ensure that your AngularJS application communicates with the server over HTTPS to encrypt data in transit and prevent man-in-the-middle attacks.

5. Error Handling: Implement error handling for authentication and authorization failures. Provide meaningful error messages to users and log security-related issues for administrators.

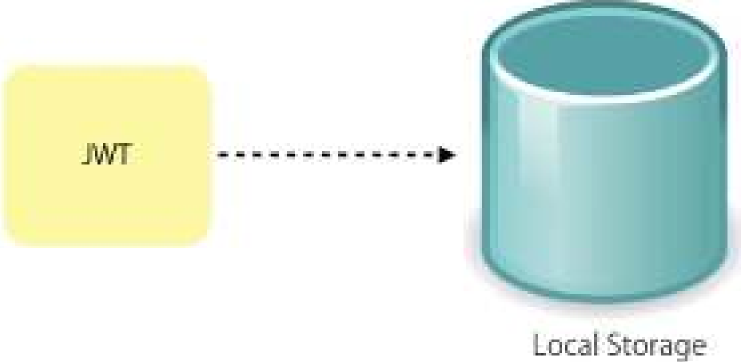
6. Testing and Security Auditing: Conduct thorough testing, including security testing (e.g., penetration testing), to identify vulnerabilities.

Stay up to date with security best practices and regularly update your application and dependencies to address known vulnerabilities.

7. Third-Party Authentication: If your application supports third-party authentication (e.g., OAuth, social logins), implement the necessary components to integrate with external identity providers.

13.3 JSON Web Tokens

JSON Web Tokens (JWTs) are commonly used in AngularJS (and other web applications) for authentication and authorization. JWTs are a compact and self-contained means of representing information between two parties, typically the client and the server. They can be used to securely transmit information, including user authentication data, between different components of an application.



Storage of JSON Web Token inside the browsers' local storage.

Here's how you can implement JWT-based authentication and authorization in an AngularJS application:

1. Server-Side Implementation:

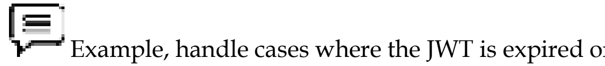
- User Authentication: When a user successfully logs in, the server generates a JWT containing user information and signs it using a secret key. The JWT is then returned to the client.
- Token Expiration: JWTs typically have a limited lifespan to enhance security. Set an expiration time in the JWT payload to control how long the token is valid.
- Token Refresh (Optional): Implement a token refresh mechanism if needed. When a JWT expires, the client can request a new one using a refresh token (if available) without requiring the user to log in again.

2. AngularJS Client-Side Implementation:

- Authentication Service: Create an AngularJS service (e.g., AuthService) responsible for managing JWT-based authentication.
- Token Storage: Store the JWT securely on the client side, usually in a secure HTTP-only cookie or local storage. Ensure it's sent with each HTTP request as an authorization header.
- Login: When a user logs in, send their credentials to the server using an HTTP request. Upon successful authentication, store the JWT in the client's storage.
- Logout: Implement a logout function that clears the stored JWT when the user logs out.
- Authorization Guards: Use AngularJS route guards (e.g., resolve or custom guards) to protect specific routes. These guards should check if the user is authenticated and, optionally, if they have the necessary roles or permissions.

3. Secure Communication: Ensure that your AngularJS application communicates with the server over HTTPS to encrypt data in transit and prevent eavesdropping.

4. Error Handling: Implement error handling for authentication and authorization failures.



Example, handle cases where the JWT is expired or invalid.

5. Server-Side Authorization: On the server side, implement authorization logic based on the user's roles or permissions contained in the JWT payload. Check these roles or permissions before allowing access to specific resources or actions.

6. Token Validation: When receiving a request on the server, validate the JWT's signature using the secret key to ensure its authenticity. Verify the expiration time and other claims in the JWT payload.

7. Token Refresh (Optional): If you implement token expiration and refresh, create a mechanism for the client to request a new JWT using a refresh token when the current JWT expires.

8. Testing and Security: Conduct thorough testing, including security testing, to identify vulnerabilities. Stay up to date with security best practices and regularly update your application and dependencies to address known vulnerabilities.

By implementing JWT-based authentication and authorization in your AngularJS application, you can secure your application's endpoints and ensure that users have the necessary permissions to access specific resources while keeping user data and session state on the client side.

Note: JWTs should be used carefully and should not contain sensitive information, as they are not encrypted and can be decoded on the client side.

13.4 Implementing Login

Implementing a basic login functionality in an AngularJS application involves creating a login form, handling user input, and making an authentication request to a server. Here's a step-by-step example of how you can implement a simple login page in AngularJS:

1. Set Up Your AngularJS App: First, make sure you have AngularJS included in your project. You can include it via a CDN or use a package manager like npm or bower to install it.

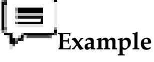
Example

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
  <title>Login Page</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.8.2/angular.min.js"></script>
</head>
<body ng-controller="LoginController">
  <h1>Login Page</h1>
  <!-- Create a login form -->
  <form ng-submit="login()">
    <label for="username">Username:</label>
    <input type="text" id="username" ng-model="credentials.username" required><br>

    <label for="password">Password:</label>
    <input type="password" id="password" ng-model="credentials.password" required><br>
    <button type="submit">Login</button>
  </form>
  <div ng-show="error" style="color: red;">{{ error }}</div>
```

</body>

</html>

2. Create an AngularJS Controller: Create an AngularJS controller to manage the login functionality and handle user input.

**Example**

```
// Define your AngularJS app
var app = angular.module('myApp', []);
// Create a controller for the login page
app.controller('LoginController', function ($scope, $http) {
  // Initialize credentials object to hold username and password
  $scope.credentials = {
    username: '',
    password: ''
  };
  // Initialize error message
  $scope.error = '';

  // Function to handle the login form submission
  $scope.login = function () {
    // Send a POST request to your server for authentication
    $http.post('/api/login', $scope.credentials)
      .then(function (response) {
        // Authentication successful, perform necessary actions (e.g., redirect)
        alert('Login successful');
      })
      .catch(function (error) {
        // Authentication failed, display an error message
        $scope.error = 'Authentication failed. Please check your credentials.';
      });
  };
});
```

3. Server-Side Authentication: On the server side, you should have an endpoint (e.g., /api/login) that handles the authentication logic. When the client sends a POST request with the provided credentials, the server should validate the credentials against the database or any authentication system and respond accordingly.

Remember to implement server-side authentication logic securely, including password hashing and protection against brute-force attacks.

4. Error Handling: In the example, the $scope.error variable is used to display error messages to the user. You can customize the error handling according to your application's requirements.

5. Styling and Enhancement: You can enhance the login page by adding CSS for styling, implementing features like password reset, or integrating with JWT-based authentication, as discussed in previous responses.

Note: Resolve is used to pre-fetch the data before navigating to the route.

13.5 Implementing Logout

Implementing a logout functionality in an AngularJS application is relatively straightforward. Logout involves clearing the user's session or authentication token and, optionally, redirecting them to a different page. Below is a step-by-step example of how you can implement a basic logout feature in AngularJS:

1. Create a Logout Button: Add a "Logout" button or link in your application's user interface. This button will trigger the logout process when clicked.

Example

```
<!-- Inside your AngularJS view or template -->

<button ng-click="logout()">Logout</button>
```

2. AngularJS Controller: Create an AngularJS controller to manage the logout functionality. This controller should clear the user's session or authentication token and perform any necessary actions, such as redirecting to a login page.

Example

```
// Define your AngularJS app

var app = angular.module('myApp', []);

// Create a controller for the logout functionality

app.controller('LogoutController', function ($scope, $http, $window) {

  // Function to handle the logout action

  $scope.logout = function () {

    // Optionally, send a request to the server to invalidate the session or token

    // This step depends on your server-side authentication implementation


    // Clear the user's session or token on the client side (e.g., remove a token from local storage)

    // Replace 'myToken' with the actual name of your token/key

    $window.localStorage.removeItem('myToken');

    // Redirect the user to the login page or any other desired destination

    $window.location.href = '/login'; // Replace with your login page URL

  };

});
```

3. Securely Clear User Session: Make sure to securely clear the user's session or authentication token on the client-side. In this example, we use localStorage to store the token; you should remove it as appropriate to your implementation. If you're using cookies, clear the cookie.

4. Server-Side Considerations (Optional): Depending on your server-side authentication implementation, you may need to send a request to the server to invalidate the session or token. This step is essential to ensure that the user cannot use the invalidated token after logout. The exact server-side implementation will depend on your backend technology stack.

5. Logout Button Usage: Place the "Logout" button in your application's user interface where users can access it easily, typically in a navigation bar or user profile section.

6. Testing: Thoroughly test the logout functionality to ensure that it clears the user's session or token, prevents access to protected pages, and redirects the user to the login page or other desired destinations.

Summary

- Authorization determines what actions or resources a user is allowed to access after authentication.
- AngularJS provides route guards and custom directives to control access to specific parts of your application based on user roles or permissions.
- Define authorization rules on both the client and server sides, ensuring that server-side authorization is the final layer of defense.
- Implement role-based or permission-based authorization logic on the server to restrict access to specific endpoints or resources.
- AngularJS allows you to create login forms and handle user authentication with server-side APIs.
- You can use techniques like token-based authentication (JWT), session-based authentication, OAuth, or Firebase Authentication.
- Securely store user authentication tokens on the client-side and send them with each HTTP request to identify the user.
- Implement token expiration and refresh mechanisms for security.
- Authorization determines what actions or resources a user is allowed to access after authentication.
- AngularJS provides route guards and custom directives to control access to specific parts of your application based on user roles or permissions.
- Define authorization rules on both the client and server sides, ensuring that server-side authorization is the final layer of defense.
- Implement role-based or permission-based authorization logic on the server to restrict access to specific endpoints or resources.
- Secure communication between the client and server using HTTPS to prevent eavesdropping.

Keywords

Login: The process by which a user provides their credentials (usually a username and password) to gain access to an application.

Logout: The process of terminating a user's session and revoking their access to protected resources.

Authentication Service: An AngularJS service responsible for managing user login, logout, and authentication-related functionality.

Token-Based Authentication: A method where a user receives a token (e.g., JWT) upon successful login, which is used to authenticate subsequent requests.

Session-Based Authentication: A method where a user's session is created on the server upon login, and a session identifier is used to track authenticated users.

Authorization: The process of determining what actions or resources a user is allowed to access based on their identity and permissions.

Role-Based Authorization: Assigning roles (e.g., admin, user) to users and restricting access to certain features or routes based on these roles.

Permission-Based Authorization: Granting specific permissions to users and controlling access based on these permissions.

Error Handling: Implementing mechanisms to handle authentication and authorization errors gracefully, providing meaningful feedback to users.

Logout Functionality: A feature that allows users to log out, clear their session or token, and revoke their access to protected resources.

Self Assessment

1. What is authentication in the context of web applications?
A. Determining user roles and permissions.
B. Verifying the identity of a user.
C. Controlling access to specific resources.
D. Encrypting data during transmission.

2. Which of the following is NOT a commonly used authentication method in web applications?
A. Token-based authentication
B. Session-based authentication
C. OAuth
D. URL-based authentication

3. What is the purpose of a JSON Web Token (JWT) in authentication?
A. To store user passwords securely.
B. To encrypt all communication between the client and server.
C. To verify the identity of the user and store user-related information.
D. To generate random tokens for temporary access.

4. Which of the following is a common technique for authorization in web applications?
A. Token-based authentication
B. URL rewriting
C. Session management
D. Role-based access control

5. What is the purpose of a route guard in AngularJS when it comes to authorization?
A. To authenticate the user with a username and password.
B. To handle errors during authentication.
C. To restrict access to certain routes based on user roles or permissions.
D. To manage user sessions.

6. What is the primary purpose of a JSON Web Token (JWT) in AngularJS authentication?
A. To store user passwords securely.
B. To encrypt all communication between the client and server.
C. To verify the identity of the user and store user-related information.
D. To create temporary access tokens for user sessions.

7. Which of the following statements about JWTs is true?
A. JWTs are always encrypted to protect the data they contain.
B. JWTs can only be used for authentication but not authorization.
C. JWTs are self-contained and can store information about the user.
D. JWTs are validated on the client-side to verify user identity.

8. How are JWTs typically transmitted between the client and the server in AngularJS applications?
A. In a secure HTTP-only cookie.
B. As part of the URL parameters.
C. As a plain text header in HTTP requests.
D. In the request body as form data.

9. What is the role of the secret key when verifying JWTs in AngularJS authentication?
A. It's used to encrypt the JWT payload.
B. It's used to generate new JWTs for each request.
C. It's used to sign and verify the authenticity of the JWT.
D. It's used to decode the JWT on the client-side.

10. Which of the following claims is commonly found in a JWT's payload for authorization purposes?
A. "expiration_time" (exp) to indicate when the JWT expires.
B. "username" (sub) to identify the user.
C. "token_type" (typ) to specify the type of authentication token.
D. "access_level" (acl) to store user permissions.

11. What is the primary purpose of a login functionality in an AngularJS application?
A. To display user information on the homepage.
B. To authenticate users and grant access to protected resources.
C. To generate random tokens for secure communication.
D. To log application errors for debugging.

12. In an AngularJS application, where is user input typically collected when implementing a login feature?
A. In a server-side script.
B. In an AngularJS controller from a login form.
C. In a URL parameter.
D. In a global configuration file.

13. What does the term "logout" refer to in web development?
A. Terminating a user's browser session.
B. Forcing a user to re-enter their credentials.
C. Clearing the browser's history.
D. Blocking access to specific routes.

14. When implementing logout functionality in AngularJS, what is typically cleared or reset?
A. The user's username and password.
B. The user's authentication token or session on the client-side.
C. The server's session data.
D. The application's configuration settings.

15. What is the primary benefit of using a secure HTTP-only cookie to store user authentication tokens in an AngularJS application?

A. Cookies are more convenient for the user.

B. Cookies are easier to implement than other storage options.

C. Cookies provide better performance compared to other storage methods.

D. Cookies cannot be accessed by JavaScript, enhancing security

.

Answers for Self Assessment

1.	B	2.	D	3.	C	4.	D	5.	C
6.	C	7.	C	8.	C	9.	C	10.	B
11.	B	12.	B	13.	A	14.	B	15.	D

Review Questions

1. Explain the concept of authentication in AngularJS. How does AngularJS facilitate user authentication in web applications?
2. Discuss the significance of authorization in web applications with the help of appropriate example.
3. How can AngularJS be used for implementing authorization? Give example.
4. Describe the components of an authentication service in an AngularJS application.
5. Discuss the security considerations when implementing login and logout functionality in an AngularJS application.

Further Readings

Soni, R. K. (2017). Full stack angularJS for java developers: Build a full-featured web application from scratch using angularJS with spring RESTful. Apress.

Waikar, M. (2015). Data-oriented Development with AngularJS. Packt Publishing Ltd.

Williamson, K. (2015). Learning AngularJS: A Guide to AngularJS Development. " O'Reilly Media, Inc.".

Moreno, Z. (2015). AngularJS Deployment Essentials. Packt Publishing Ltd.

Panda, S. (2014). AngularJS: Novice to Ninja: Elegant, Powerful, Testable, Extendable. SitePoint Pty Ltd

Web Links

https://www.sitepoint.com/easy-angularjs-authentication-with-auth0/

https://grigorkh.medium.com/easy-angularjs-authentication-with-auth0-7214f60850dd#:~:text=Authentication%20for%20Angular%20Apps,when%20we%20make%20HTTP%20requests.

https://www.oreilly.com/library/view/angularjs-novice-to/9781457174360/ch16.html

https://www.codeproject.com/Tips/811782/AngularJS-Security-Authorization-on-Angular-Routes

https://www.innominds.com/blog/client-side-authentication-using-angularjs

Unit 14: Deployment

CONTENTS

Objectives
Introduction
14.1 Preparing for Deployment
14.2 Deploying to GitHub Pages
14.3 Deploying to Firebase
Summary
Keywords
Self Assessment
Answers for Self Assessment
Review Questions
Further Readings

Objectives

After this unit you will be able to :

- Learn the steps of preparing deployment in angular applications.
- Understand the deployment of angular applications GitHub Pages and Firebase

Introduction

AngularJS, also known as Angular 1.x, is an older version of the Angular framework. While it is not as widely used today as the more recent versions of Angular (such as Angular 2+), it's still essential to understand how to deploy AngularJS applications if you are working with legacy code or maintaining existing projects. Here's an introduction to deploying AngularJS applications:

Prepare Your AngularJS Application: Before deploying your AngularJS application, make sure it's fully developed and tested. Ensure that you have a working build of your application ready for production.

Optimize Your Code: Minimize and compress your JavaScript and CSS files to reduce load times. You can use tools like UglifyJS for JavaScript minification and CSS minifiers for CSS files.

Configure Environment Variables: Ensure that your application is correctly configured for production by setting environment variables. You might need to change API endpoints or other configurations for the production environment.

Remove Debugging Code: Remove any console.log statements or debugging code that you might have used during development. These should not be present in your production code.

Set Up a Production Server: You'll need a web server to host your AngularJS application. Common choices include Apache, Nginx, or a serverless platform like AWS Lambda.

Deploy to Your Server: Depending on your server setup, you'll need to copy your production-ready AngularJS files (HTML, CSS, and JavaScript) to your server's web directory. This can be done using FTP, SCP, or other file transfer methods.

Set Up Routing (if applicable): If your AngularJS application uses client-side routing (e.g., using ngRoute), make sure that your server is configured to serve the application's entry point (usually

the index.html file) for all routes. This typically involves configuring your server to handle URL rewrites or redirects.

Configure Domain and DNS (if applicable): If you have a custom domain name for your AngularJS application, configure your domain's DNS settings to point to your server's IP address. You may also need to configure server settings to handle incoming requests for that domain.

Secure Your Application: Implement security measures, such as HTTPS, to protect data transmission between the client and server. Use security headers like Content Security Policy (CSP) to enhance security.

Backup and Rollback Plan: Have a backup and rollback plan in place in case something goes wrong during deployment. This should include regular backups of your application and server configurations.

Testing: Thoroughly test your deployed AngularJS application on the production server to ensure that everything works as expected.

Monitoring and Maintenance: After deployment, continuously monitor your application in the production environment. Apply updates and patches as needed and perform regular maintenance to keep your application running smoothly.

Documentation: Ensure that you have clear and up-to-date documentation for deployment procedures and configurations, especially if multiple team members are involved in the deployment process.

Deploying an AngularJS application involves similar principles to deploying other web applications. While AngularJS has been largely replaced by newer versions of Angular (Angular 2+), understanding the deployment process for legacy AngularJS applications can still be valuable if you work on projects that use this older framework.

14.1 Preparing for Deployment

Preparing an Angular application for deployment involves several steps to ensure that your app is optimized, secure, and ready to be hosted on a web server. Here's a checklist to help you prepare your Angular app for deployment:

Optimize Your Code: Use production builds: Generate a production build of your app using the Angular CLI with the --prod flag. This minifies and optimizes your code for better performance.

ng build --prod

Enable AOT (Ahead-of-Time) compilation to reduce the size of your bundles and improve load times.

Remove Development Code: Ensure that you've removed or disabled any development-only code, such as console.log statements or debugging code.

Optimize Assets: Compress and optimize images, fonts, and other static assets to reduce load times.

Consider using lazy loading for modules to defer loading of non-essential code until it's needed.

Set Base Href: Update the <base href="/"> tag in your index.html to match the base URL where your app will be hosted.

Configure Routing for Hash Strategy (Optional): If you encounter issues with server configuration, you can enable the hash strategy for routing. Update your routing configuration in app-routing.module.ts:

Example

imports: [RouterModule.forRoot(routes, { useHash: true })]

Secure Your App: Implement security measures, such as using HTTPS, to protect data transmission between the client and server. Implement appropriate security headers like Content Security Policy (CSP), X-Content-Type-Options, etc.

Environment Configuration: Ensure that your application's environment configurations are set correctly for production, including API endpoints and other environment-specific variables.

Testing: Perform thorough testing on your production build to ensure that everything works as expected.

Update Dependencies: Keep your Angular and other dependencies up-to-date. Run ng update to update Angular packages.

Error Handling and Logging: Implement error handling and logging mechanisms to monitor and troubleshoot issues in the production environment.

Server Configuration: If you're deploying to a specific server or hosting platform (e.g., Apache, Nginx, AWS, or Firebase), follow the deployment guidelines and configure your server or hosting environment accordingly.

Build for Multiple Environments: Create separate builds for different environments (e.g., development, staging, production) with appropriate configurations.

CD/CI Pipeline (Optional): Consider setting up a Continuous Integration/Continuous Deployment (CI/CD) pipeline to automate the deployment process.

Testing on Staging: Before deploying to the production server, test your application on a staging server to catch any last-minute issues.

Backup and Rollback Plan: Have a backup and rollback plan in place in case something goes wrong during deployment.

Monitoring and Analytics: Integrate monitoring tools and analytics to track the performance and usage of your application in the production environment.

Documentation: Ensure that you have clear and up-to-date documentation for deployment procedures and configurations.

DNS and Domain Configuration: If you're using a custom domain, configure DNS settings to point to your hosting server or platform.

Final Testing and Verification: Perform final testing and verification on the production environment to ensure that everything is working as expected.

Backup Data (if applicable): If your application involves data storage, make sure you have proper backup mechanisms in place.

Deployment: Deploy your application to the production server or hosting platform following the deployment procedure for your chosen environment.

Monitoring and Maintenance: Continuously monitor your deployed application, apply updates and patches, and perform regular maintenance as needed.

By following these steps, you can effectively prepare your Angular application for deployment and ensure a smooth and secure transition to a production environment.

Note: If you copy the files into a server sub-folder, append the build flag, --base-href and set the <base href="./"> or use the appropriate path from the root directory.

14.2 Deploying to GitHub Pages

Deploying an Angular application to GitHub Pages is a straightforward process. GitHub Pages is a free hosting service provided by GitHub that allows you to host static websites, including Angular applications. Here's how you can deploy your Angular app to GitHub Pages:

slm Before deploying your Angular app to GitHub Pages, ensure that you have built and optimized your application for production. Use the Angular CLI to create a production build:

ng build --prod

This command will generate a dist/ directory containing your production-ready application.

Create a GitHub Repository: If you haven't already, create a GitHub repository to host your Angular app. Make sure your Angular project is version-controlled using Git, and push it to this repository. To simplify the deployment process, you can use the angular-cli-ghpages package, which automates the deployment to GitHub Pages. Install it globally or locally in your project:

npm install -g angular-cli-ghpages

Deploy Your App: Now, you can deploy your Angular app to GitHub Pages using the angular-cli-ghpages package. Run the following command in your project's root directory:

npx ngh --dir=dist/<your-app-name>

Replace <your-app-name> with the actual name of your Angular app. This command will build your app (if not already built) and deploy it to GitHub Pages.

Set the Repository's GitHub Pages Branch: By default, GitHub Pages deploys from the gh-pages branch of your repository. If you haven't configured it yet, go to your GitHub repository's settings and choose the gh-pages branch as the source for GitHub Pages.

Access Your Deployed App: Your Angular app should now be deployed to GitHub Pages. You can access it at the following URL:

Example

https://<username>.github.io/<repository-name>/

Replace <username> with your GitHub username and <repository-name> with the name of your GitHub repository.

Custom Domain (Optional): If you want to use a custom domain for your GitHub Pages site, you can configure it in your repository's settings on GitHub. Follow GitHub's documentation on setting up a custom domain for more details.

Testing and Troubleshooting: Test your deployed app thoroughly to ensure that it works as expected in the GitHub Pages environment. If you encounter any issues, check the browser's developer console for error messages.

Update and Redeploy: Whenever you make changes to your Angular app and want to update the deployed version on GitHub Pages, simply rebuild and redeploy using the npx ngh command again.

By following these steps, you can easily deploy your Angular application to GitHub Pages and make it accessible to a wide audience on the web.

14.3 Deploying to Firebase

Deploying an Angular application to Firebase is a convenient way to host and serve your app. Firebase provides hosting services that are well-suited for static websites, including Angular applications. Here's a step-by-step guide on how to deploy your Angular app to Firebase:

Prepare Your Angular App: Before deploying to Firebase, ensure that your Angular app is optimized for production. Use the Angular CLI to create a production build:

ng build --prod

This command will generate a dist/ directory containing your production-ready Angular app.

Install the Firebase CLI: If you haven't already, you need to install the Firebase CLI globally on your machine. You can do this using npm:

npm install -g firebase-tools

Log In to Firebase: Use the Firebase CLI to log in to your Firebase account. Run the following command and follow the prompts:

firebase login

This will open a browser window for authentication.

Initialize Firebase in Your Project: Navigate to your Angular project's root directory and run the following command to initialize Firebase:

firebase init

During the initialization process, you will be prompted to configure Firebase features. Make sure to select Firebase Hosting.

Set Up Your Firebase Project: If you haven't created a Firebase project yet, you can do so at the Firebase Console (https://console.firebase.google.com/). Then, associate your local project with the Firebase project you just created.

Configure Hosting: During the Firebase initialization process, you will be asked to configure Firebase Hosting settings. Set the public directory to dist/<your-app-name>, where <your-app-name> is the name of your Angular app.

Example, if your app is named "my-angular-app," the public directory should be dist/my-angular-app.

Deploy Your App: Once Firebase Hosting is configured, you can deploy your Angular app to Firebase by running:

firebase deploy

Firebase will upload your app files to the hosting environment, and you'll receive a hosting URL where your app is accessible.

Access Your Deployed App: After a successful deployment, Firebase will provide a hosting URL where your Angular app is now hosted. You can access your deployed app using this URL.

https://<your-firebase-project-id>.web.app

The URL will be based on your Firebase project ID.

Custom Domain (Optional): If you want to use a custom domain for your Firebase-hosted Angular app, you can configure it through the Firebase Console under Hosting settings. Follow the instructions to set up your custom domain.

Testing and Troubleshooting: Test your deployed app thoroughly to ensure it functions correctly in the Firebase Hosting environment. If you encounter any issues, check the browser's developer console for error messages.

Update and Redeploy: Whenever you make changes to your Angular app and want to update the deployed version on Firebase Hosting, run the firebase deploy command again.

By following these steps, you can easily deploy your Angular application to Firebase Hosting, taking advantage of Firebase's hosting services for a scalable and reliable web hosting solution.

Summary

- Use the Angular CLI to generate a production build: ng build --prod.
- Deploying your application is the process of compiling, or building, your code and hosting the JavaScript, CSS, and HTML on a web server.
- To use the ng deploy command, use ng add to add a package that implements deployment capabilities to your favorite platform.
- Adding the package automatically updates your workspace configuration, adding a deployment CLI builder.
- Firebase will upload your app files to the hosting environment, and you'll receive a hosting URL where your app is accessible.

- Deploying an Angular application to Firebase is a convenient way to host and serve your app.
- Firebase provides hosting services that are well-suited for static websites, including Angular applications.
- Continuous deployment is going one step further by also automating the deployment.

Keywords

Build: The process of converting your Angular application's source code into optimized and bundled production-ready files.

Production Build: A build configuration in Angular that optimizes code and assets for deployment in a production environment.

Environment Configuration: Settings and variables specific to the deployment environment, such as API endpoints, database connections, and feature flags.

Minification: The process of removing unnecessary characters and spaces from code to reduce its size and improve load times.

UglifyJS: A popular JavaScript minification tool used to compress and obfuscate JavaScript code.

Routing Configuration: Setting up server-side routing rules or redirects to ensure that client-side routes work correctly when users access specific URLs.

Web Server: The software responsible for serving your Angular application to users over the internet, such as Apache, Nginx, or IIS.

Server Configuration: The setup and configuration of the web server to handle incoming requests, including security settings and routing rules.

Hosting: The platform or service where your Angular application's files are stored and served to users, e.g., Firebase Hosting, AWS S3, GitHub Pages.

Domain and DNS: The domain name system (DNS) settings that link a custom domain to a server's IP address, allowing users to access your app via a human-readable URL.

HTTPS: The secure version of HTTP (Hypertext Transfer Protocol) that encrypts data transmitted between the client and server, ensuring data security.

Security Headers: Headers sent by the server to enhance security, including Content Security Policy (CSP) and X-Content-Type-Options.

Backup and Rollback: Strategies and plans for data and configuration backups, as well as procedures for reverting to a previous version in case of issues.

Self Assessment

1. What is the purpose of a production build in Angular?

A. To create a backup of your application
B. To optimize code and assets for deployment
C. To enable debugging features
D. To generate documentation

2. Which Angular CLI command is used to create a production build?

A. ng compile
B. ng optimize
C. ng build --prod
D. ng deploy

3. What does HTTPS provide for a deployed Angular application?
A. Faster load times
B. Improved user experience
C. Encryption of data transmitted between the client and server
D. Enhanced debugging capabilities

4. What is the purpose of minifying JavaScript and CSS files during deployment?
A. To make code more readable
B. To increase file size
C. To improve performance and reduce load times
D. To add security features

5. Which of the following is NOT a key consideration during the deployment of an Angular application?
A. Domain and DNS configuration
B. Server setup and security
C. Debugging code preservation
D. Continuous monitoring and maintenance

6. What is GitHub Pages?
A. A code repository hosting service
B. A cloud-based database service
C. A free hosting service for static websites
D. A continuous integration tool

7. How can you deploy an Angular app to GitHub Pages?
A. By using the "ng deploy" command
B. By manually uploading files to the GitHub Pages server
C. By running "npm deploy" from the project root
D. By using a third-party deployment tool

8. What is the default branch for GitHub Pages deployments?
A. master
B. staging
C. gh-pages
D. production

9. What command is used to install the Angular CLI deployment package for GitHub Pages?
A. ng add angular-cli-ghpages
B. npm install -g angular-cli-ghpages
C. ng deploy --github-pages
D. npm deploy angular-gh-pages

10. After deploying an Angular app to GitHub Pages, what is the URL where your app can be accessed?
A. https://github.com/username/repo

B. https://github.io/username/repo
C. https://username.github.io/repo
D. https://pages.github.com/username/repo

11. What is Firebase in the context of web development?
A. A front-end JavaScript framework
B. A back-end server technology
C. A hosting platform for web applications
D. A version control system

12. Which command is used to initialize a Firebase project in your Angular application directory?
A. ng new firebase-project
B. ng add @angular/fire
C. firebase init
D. ng init firebase

13. What is the purpose of the ng build --prod command before deploying to Firebase?
A. To transpile TypeScript code to JavaScript
B. To optimize and prepare the Angular app for production
C. To deploy the app to Firebase
D. To install Firebase CLI globally

14. Which Firebase service is commonly used for hosting Angular applications?
A. Firebase Firestore
B. Firebase Authentication
C. Firebase Storage
D. Firebase Hosting

15. What should you configure in your Firebase project before deploying an Angular app?
A. Firebase Realtime Database rules
B. Firebase Authentication settings
C. Firebase Cloud Functions
D. Firebase Hosting settings

Answers for Self Assessment

1.	B	2.	C	3.	C	4.	C	5.	C
6.	C	7.	B	8.	A	9.	B	10.	C
11.	C	12.	C	13.	B	14.	D	15.	D

Review Questions

1. Discuss why it's essential to use a production build when deploying an Angular application to a live environment.

2. Define what environment configuration means in Angular and explain why it's crucial to configure environment-specific variables for the production environment.
3. Write down the instructions using the Angular CLI, setting up GitHub Pages, and configuring the deployment.
4. Explain why thorough testing is crucial before and after deployment. Describe the benefits of continuous monitoring and how it aids in maintaining a deployed Angular application.
5. Explain in detail the Firebase CLI commands and configurations required for deployment.
6. Explain the Steps Involved in Deploying an Angular Application to Firebase Hosting.

Further Readings

Moreno, Z. (2015). *AngularJS Deployment Essentials*. Packt Publishing Ltd.

Akopkokhyants, S., & Radford, S. (2016). *Learning Web Development with Bootstrap and Angular*. Packt Publishing.

Bampakos, A., & Deeleman, P. (2023). *Learning Angular: A no-nonsense guide to building web applications with Angular 15*. Packt Publishing Ltd.

Wilken, J. (2018). *Angular in action*. Simon and Schuster.

Williamson, K. (2015). *Learning AngularJS: A Guide to AngularJS Development*. " O'Reilly Media, Inc.".

Web Links

https://angular.io/guide/deployment

https://www.cs.virginia.edu/~up3f/cs4640/supplement/angular-deployment.html

https://blog.back4app.com/how-to-deploy-an-angular-application/

https://betterprogramming.pub/how-to-build-and-deploy-an-angular-application-directly-from-github-a0aa5f28e6aa

https://www.ngdevelop.tech/angular/tutorial/deployment/

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in