Module 1

What is Angular
Angular setup, Install
Angular Component
Data-Binding
Interpolation

Practice Exercise

Practice 1 (65cdfdab7ba36e064487629c)
1. Create Hello World Component and call the helloWorld Component in the App Component.

*Display massage "Hello World " in control id -> #display-msg

Practice 2 (65cdfdab7ba36e064487629d)
2. HelloworldComponent-> Declare a name variable and store your name in that variable and use header tag to interpolate the variable Greeting of the day with name stored in the variable

*Display "Good morning {user name}" in control id -> #display-greeting
userName = Tom

Assignment Exercise

Assignment 1 (65cdfdab7ba36e0644876299)
1. Create a Rectangle Component that computes the area of the Rectangle.

Rectangle Length: Control ID -> #length -> Default value -> 0

Rectangle Width: Control ID -> #width -> Default value -> 0

Calculate: Control ID -> #btn-calculate

Area of Rectangle(output) -> #rectangle-area -> Default value -> 0
*The output should have two fractional digits.
Example: 0.00, 1124.93,121.11

Assignment 2 (65cdfdab7ba36e064487629a)

2. Create Login Component that validates a user with the below credentials username=admin, password=admin

Username Input: Control ID -> #username -> Default value -> ''
Password Input: Control ID -> #password -> Default value -> ''
Login Button: Control ID -> #login-btn
Login Message: Control ID -> #login-message -> Default value -> ''

*Login messages:
- For successfully login -> "Login successful"
- For failure in login -> "Invalid credentials"

Assignment 3 (65cdfdab7ba36e064487629b)

3. Create a circle component that computes the area of the Circle

Radius Input: Control ID -> #radius
Calculate Button: Control ID -> #calculate-btn
Circle Area Output: Control ID -> #circle-area

*The output should have two fractional digits.
Example: 0.00, 31415.93,121.11

Module 2

What to learn

Data-Binding
Interpolation
Property Binding
Two Way Binding
Event Binding
Built in Pipes

Practice Exercise

Practice 1 (65cdfdab7ba36e06448762a2)
Create one textbox in Helloworld component. Bind its value to the name variable.

Textbox for Name Input: Control ID -> #name-input
<p> tag to Display Value: Control ID -> #displayed-value

*Please set the default variable value to an empty ("") string.

Practice 2 (65cdfdab7ba36e06448762a3)
Create a calculator component that performs addition, multiplication, and subtraction operations.

Number 1 Input: Control ID -> #num1 -> Default value -> 0
Number 2 Input: Control ID -> #num2 -> Default value -> 0
Operation Dropdown: Control ID -> #operation
Options: Addition, Subtraction, Multiplication
Calculate Button: Control ID -> #calculate-btn
Result Output: Control ID -> #result -> Default value -> 0

*The output should have two fractional digits.
Example: 0.00, 5.93, -121.11

Practice 3
Implement a component where a user's name input is reflected in a greeting message in real-time.
Example: Type "John" in the input, and the page displays "Hello, John!"

Practice 4

Create a product quantity selector with a two-way binding feature. The total price should update automatically based on the quantity entered and a fixed product price.

Practice 5

Bind a dropdown menu's selected value to a property in your component and display the selected value below the menu.

Practice 6

Create a form that updates a user's profile information (name, email, and phone number) using two-way data binding.

Practice 7

Develop a two-way bound search box that filters and displays a list of items dynamically as the user types.

Practice 8

Implement a two-way bound counter component with "+" and "-" buttons to increase or decrease the value. Ensure the value is capped between 0 and 100.

Practice 9

Write a component where a text input field is used to update a list of tasks in an array dynamically. Display the tasks in a list format.

Practice 10

Create a budget tracker where the user can enter their monthly income and expenses. Use two-way binding to calculate and display the remaining balance.

Practice 11

Develop a voting system where each user can vote up or down on a topic. Use two-way data binding to reflect the votes dynamically.

Practice 12

Implement a form with a slider input for selecting a discount percentage. Display the discounted price in real-time based on the original price.

Practice 13

Create a shipping cost calculator where a user can select the weight of a package (dropdown) and distance (slider). Use two-way data binding to display the calculated cost dynamically.

Practice 14

Design a temperature converter component where the user can input a value in Celsius or Fahrenheit, and the corresponding value updates automatically.

Practice 15

Implement a quiz component where a user selects answers from radio buttons. Use two-way data binding to update their score after submission.

Practice 16

Create a stock management form with fields for stock name, quantity, and price. Use two-way binding to calculate the total value of the stock dynamically.

Practice 17

Develop a real-time expense tracker where a user enters their expenses in different categories (e.g., Food, Travel). Display the total expenses live as they update the values.

Assignment Exercise

Assignment 1

Create a left bar component which contain some dummy adds.

Assignment 2 (65cdfdab7ba36e064487629f)

Create a signup component that contains fields for Name, Address, and PanNumber, and interpolates these pieces of information into a paragraph.

Name Input: Control ID -> #name-input
Address Input: Control ID -> #address-input
PanNumber Input: Control ID -> #pan-input
Submit Button: Control ID -> #submit-btn
Displayed Information Output: Control ID -> #displayed-info -> Default value -> ""

*The displayed information should be updated after clicking the Submit button, showing the entered values for Name, Address, and PanNumber in the paragraph.

Example: If the user enters "John Doe" for Name, "123 Main St" for Address, and "ABCDE1234F" for PanNumber and clicks Submit, the displayed information should be "Name: John Doe, Address: 123 Main St, PanNumber: ABCDE1234F".

Assignment 3

Call all these component in the App Component.

Assignment 4

Arrange these component properly, Use proper Bootstrap, Two-way Binding

Online Reference

No online Reference

Module 3

What to learn

Built in Directives in Angular
Component->directive with a template
Attribute directive
NgClass
NgStyle
NgModel
Structural directives
NgIf
NgFor
NgSwitch

Practice Exercise

Practice 1
Do the hands on from following link https://angular.io/guide/built-in-directives
Practice 2
Dynamic Welcome Message

Use *ngIf to show a "Welcome, [user's name]!" message only when a name is entered in an input field using two-way binding.

Practice 3
Editable Product List
Use *ngFor to display a list of product names, and allow editing each name using an input field with two-way binding.

Practice 4
Highlight Active User
Use [ngClass] to highlight a user as "active" in a list of users when selected via two-way binding on a dropdown menu.

Practice 5
Character Counter
Display a character count below a text area using two-way binding, and hide it using *ngIf if no text is entered.

Practice 6

Dynamic Style Application
Change the background color of a div based on a dropdown selection using two-way binding and [ngStyle].

Practice 7
Real-time Discount Calculator
Use two-way binding to accept a product price input and display the discounted price dynamically using *ngIf for values greater than zero.

Practice 8
Task Completion Status
Display a list of tasks with checkboxes using *ngFor, and dynamically update a "completed tasks" count using two-way binding.

Practice 9
Search Filter for a Table
Bind an input field to a search term using two-way binding and display filtered rows of a table using *ngFor and *ngIf.

Practice 10
Dynamic Tab Activation
Use *ngSwitch and two-way binding to toggle between content for tabs (e.g., Home, About, Contact).

Practice 11
Password Strength Indicator
Bind a password input field using two-way binding and show a "Weak", "Medium", or "Strong" indicator using *ngIf based on password length.

Practice 12
Dynamic Role Display
Use two-way binding to select a role (Admin, User) from a dropdown, and show a message like "Welcome, Admin!" using *ngIf.

Practice 13
Dynamic Pricing Based on Quantity
Create a quantity input field with two-way binding and display the total price dynamically using [ngStyle] to highlight it when the quantity exceeds 10.

Practice 14
Product Stock Status
Bind an input field to a product's stock quantity using two-way binding, and use [ngClass] to show "In Stock" or "Out of Stock" styles dynamically.

Practice 15
Real-time Word Counter
Use two-way binding with a textarea and display the total word count dynamically

using *ngIf when there is text entered.

Practice 16

Toggle Description Visibility

Bind a checkbox using two-way binding, and toggle the visibility of a product description using *ngIf.

Practice 17

Temperature Converter

Bind an input field for temperature in Celsius using two-way binding and display the equivalent Fahrenheit value dynamically using a formula.

Practice 18

Attendance Tracker

Use *ngFor to display a list of students, and toggle their attendance status (Present/Absent) using a checkbox with two-way binding.

Practice 19

User Profile Updation

Allow a user to edit their name and email using input fields with two-way binding, and display the updated values dynamically using *ngIf.

Practice 20

Dynamic Background Changer

Use a color input with two-way binding and [ngStyle] to update the background color of a div in real time.

Practice 21

Shopping Cart Quantity

Use two-way binding to adjust the quantity of items in a shopping cart and update the total price dynamically using *ngFor and [ngClass].

Assignment Exercise

Assignment 1 (65cdfdab7ba36e06448762a4)

Create a student Model interface with field(ID,Name,Age,Average,grade,Active)
StudentList Component will create an array of student type.Display an array with NGFor in table. Grade wise color should be given using ngSwitch and ngClass or ngStyle. Only active should displayed using ngIf.

Student Table: control ID -> #student-table

* Use Below json

```
students=[
  { ID: 1, Name: 'John', Age: 20, Average: 85, Grade: 'A', Active: true },
  { ID: 2, Name: 'Alice', Age: 22, Average: 75, Grade: 'B', Active: false }
];
```

Assignment 2

Use Case: Employee Leave Management System

Domain: **HR Recruitment**

Scenario: **An** Employee Leave Management System **where employees can apply for leave. The system will allow employees to select the** type of leave, dates of leave, **and provide a reason for the leave. Based on the selected leave type, the system will calculate whether it is a** paid or unpaid **leave and display the appropriate message in real time. It will also show the** available leave balance **for each leave type.**

UI Control IDs:

Employee Name Input

- **Control ID**: employeeName
- **Type**: text
- **Label**: Employee Name
- **Required**: Yes

Leave Type Dropdown

- **Control ID**: leaveType
- **Type**: select
- **Options**: Paid Leave, Sick Leave, Casual Leave
- **Label**: Leave Type
- **Required**: Yes

Leave Start Date

- **Control ID**: startDate
- **Type**: date
- **Label**: Start Date
- **Required**: Yes

Leave End Date

- **Control ID**: endDate
- **Type**: date
- **Label**: End Date

Required: Yes

Leave Reason Input

Control ID: leaveReason
Type: textarea
Label: Reason for Leave
Required: Yes

Leave Balance Display

Control ID: leaveBalance
Type: text
Label: Available Leave Balance
Read-Only: Yes

Leave Approval Status

Control ID: leaveApproval
Type: text
Label: Leave Approval Status
Read-Only: Yes

Submit Leave Application Button

Control ID: submitLeave
Type: button
Label: Submit Leave Application
Action: Submit leave application

Functional Flow:

Employee enters their name:

Employee types their **name** into the employeeName input field.
This will automatically store the employee's name in the system.

Selects Leave Type:

Employee selects the type of leave from the **Leave Type Dropdown** (leaveType).
The available options are **Paid Leave**, **Sick Leave**, and **Casual Leave**.
Each leave type will have different **leave balance** and **approval criteria**.

Leave Dates:

The employee selects **start date** and **end date** of the leave using the date pickers (startDate, endDate).
The system calculates the **duration** of leave in days based on the difference between the selected start and end dates.

Enter Leave Reason:

Employee provides the **reason for leave** in the leaveReason textarea. This will be used to evaluate whether the leave is approved and for record-keeping purposes.

Check Leave Balance:

Once the employee selects the leave type, the system will display the **leave balance** for that type in the **Leave Balance Display** (leaveBalance).

For example:

- **Paid Leave**: 12 days remaining
- **Sick Leave**: 8 days remaining
- **Casual Leave**: 5 days remaining

The system will validate if the **requested leave duration** is within the employee's available leave balance. If the leave balance is insufficient, an alert is shown.

Approval Status:

Based on the leave type and duration, the system will calculate whether the leave is **approved** or **rejected**:

- **Paid Leave**: Can be applied for up to 12 days in a year.
- **Sick Leave**: May require a doctor's note for approval.
- **Casual Leave**: Requires manager approval.

The **Leave Approval Status** (leaveApproval) is updated in real time based on the employee's inputs and the available leave balance.

Submit Leave Application:

When the employee is ready to submit the leave application, they click the **Submit Leave Application** button (submitLeave).

The system validates the form, ensuring all fields are filled correctly, and processes the leave request based on availability and approval rules.

If the leave is **approved**, a confirmation message is shown; if not, the employee is prompted to correct the details or try again later.

Leave Application Confirmation:

Once the leave application is successfully submitted, the **Leave Approval Status** (leaveApproval) updates with a message like "Leave approved" or "Leave pending approval" based on the internal logic.

Business Logic:

Leave Type Logic:

Paid Leave: The employee can take paid leave from the available balance. If the requested days exceed the balance, the system will not allow submission and display a message stating “Insufficient Paid Leave Balance.”

Sick Leave: Sick leave may require documentation (doctor’s note). If the employee selects Sick Leave, the system will show a field for uploading documents.

Casual Leave: Casual leave requires manager approval. The system will display a message “Pending Manager Approval” once the leave is requested.

Leave Duration Calculation:

The system calculates the number of days requested based on the difference between startDate and endDate. If the start date is after the end date, an error message will appear.

Leave Balance:

The system retrieves and displays the **remaining leave balance** based on the employee’s leave type. It deducts the requested leave days after approval.

Approval Flow:

For **Sick Leave** and **Casual Leave**, additional steps are added to request manager approval or upload supporting documentation.

For **Paid Leave**, it automatically checks the leave balance and allows submission if sufficient days are available.

Expected User Flow:

The **employee** fills in the necessary details in the leave form, including their **name**, **leave type**, **start and end dates**, and **leave reason**.

The system calculates the **leave duration** and displays the **leave balance** for the selected leave type.

The system checks if the employee has sufficient leave balance and approves or rejects the application based on the type of leave and duration.

The employee submits the leave application.

The employee receives an immediate notification on the **leave approval status**.

Time Estimate:

UI Design and Layout: 1-1.5 hours for setting up the form, labels, and controls.

Business Logic Implementation: 1 hour for implementing the leave validation, approval flow, and balance calculations.

- **Testing and Debugging:** 30-45 minutes for ensuring the correct flow and handling errors gracefully.

This use case covers a **real-world scenario** in **HR Recruitment** where an employee applies for leave, and the system ensures that business logic is followed for **leave approval**, **balance calculation**, and **user interaction**. It leverages Angular's **two-way binding**, **built-in directives**, and **real-time validation** for a smooth user experience.

Online Reference

No online Reference

Module 4

What to learn

Form
Reactive Form
Form Group
Form Control

Practice Exercise

Practice 1 (65cdfdab7ba36e06448762a8)
Create A Reactive Form and try to update partially some field using the patch value

User Form: control id-> #userForm
First Name: control id-> #firstName
Email: control id-> #email
Submit button: control id-> #submit
Patch value button: control id-> #patch-value
User List: control id-> #user-data

*Patch value should be = "updatedemail@gmail.com"

Practice 2 (6744120f9a32ad5122546106)

Basic Input Field Validation

Use Case: Implement a reactive form with an email field.

Functional Flow: The email field should accept only valid email addresses.
UI: Label: *Email*, Input ID: email-input, Error Message: *Invalid email address*, Error Control ID: email-error.

Practice 3 (6744120f9a32ad5122546107)
Basic Input Field Validation

Use Case: Add a required full name field with a minimum of 3 characters.

Functional Flow: Display an error when the input is empty or less than 3 characters.
UI: Label: *Full Name*, Input ID: fullname-input, Error Message: *Full name must be at least 3 characters*, Error Control ID: fullname-error.

Practice 4 (6744120f9a32ad5122546108)
Use Case: Include a numeric-only phone number field.
Functional Flow: Allow only numbers with 10 digits.
UI: Label: *Phone Number*, Input ID: phone-input, Error Message: *Phone number must be 10 digits*, Error Control ID: phone-error.

Practice 5 (6744120f9a32ad5122546109)
Use Case: Add a password field with a minimum of 8 characters, including a number and a special character.
Functional Flow: Validate format and show specific messages for missing criteria.
UI: Label: *Password*, Input ID: password-input, Error Message: *Password must contain at least 8 characters, a number, and a special character*, Error Control ID: password-error.

Practice 6 (6744120f9a32ad512254610a)

Use Case: Create a confirm password field that matches the password field.

Functional Flow: Validate password match.

UI: Label: *Confirm Password*, Input ID: confirm-password-input, Error Message: *Passwords do not match*, Error Control ID: confirm-password-error.

Practice 7 (67441336b7decf5124191945)

Use Case: Add a date picker for date of birth with validation for age (minimum 18 years).

Functional Flow: Validate that the date entered corresponds to an age of 18 or older.

UI: Label: *Date of Birth*, Input ID: dob-input, Error Message: *You must be at least 18 years old*, Error Control ID: dob-error.

Practice 8 (67441336b7decf5124191946)

Use Case: Include a postal code field with a regex pattern.

Functional Flow: Accept formats like 12345 or 12345-6789.

UI: Label: *Postal Code*, Input ID: postal-input, Error Message: *Invalid postal code format*, Error Control ID: postal-error.

Practice 9 (67441336b7decf5124191947)

Use Case: Add a conditional validation: If a checkbox is checked, a related input field becomes required.

Functional Flow: Example: If "Subscribe to newsletter" is checked, email must be filled.

UI: Checkbox ID: subscribe-checkbox, Related Input ID: email-input, Error Message: *Email is required for subscription*, Error Control ID: subscribe-error.

Practice 10 (67441336b7decf5124191948)

Use Case: Create a field for usernames that cannot contain "admin".

Functional Flow: Implement custom validation logic to reject "admin" as a username.

UI: Label: *Username*, Input ID: username-input, Error Message: *Username cannot contain "admin"*, Error Control ID: username-error.

Practice 11 (67441336b7decf5124191949)

Use Case: Validate a field for a specific range (e.g., age between 1 and 100).

Functional Flow: Allow only values within the range.

UI: Label: *Age*, Input ID: age-input, Error Message: *Age must be between 1 and 100*, Error Control ID: age-error.

Practice 12 (67441336b7decf512419194a)

Use Case: Create an address form group with required fields (Street, City, ZIP).

Functional Flow: Ensure all fields are validated as a group.

UI: Inputs: street-input, city-input, zip-input, Error Message: *All address fields are required*, Error Control ID: address-error.

Practice 13 (674417e04df0ff511ced9e3a)

Use Case: Implement validation to ensure at least one field in a group (e.g., phone or email) is filled.

Functional Flow: Display an error if both fields are empty.

UI: Group ID: contact-group, Error Message: *At least one contact method is required*, Error Control ID: contact-error.

Practice 14 (674417e04df0ff511ced9e3b)

Use Case: Add a dropdown with dependent validation for another field.

Functional Flow: If "Other" is selected in the dropdown, an input field becomes required.

UI: Dropdown ID: category-select, Dependent Input ID: other-input, Error Message: *Please specify "Other" value*, Error Control ID: other-error.

Practice 15 (674417e04df0ff511ced9e3c)

Use Case: Implement a checkbox for terms and conditions with required validation.

Functional Flow: Show error if unchecked.
UI: Checkbox ID: terms-checkbox, Error Message: *You must agree to the terms and conditions*, Error Control ID: terms-error.

Practice 16 (674417e04df0ff511ced9e3d)

Use Case: Validate a field for only uppercase input.
Functional Flow: Allow only uppercase characters.
UI: Label: *Code*, Input ID: code-input, Error Message: *Only uppercase letters are allowed*, Error Control ID: code-error.

Practice 17 (674417e04df0ff511ced9e3e)

Use Case: Add a file upload field that accepts only specific file types (e.g., .pdf or .docx).

Functional Flow: Show an error for unsupported types.
UI: Input ID: file-upload-input, Error Message: *Only PDF and DOCX files are allowed*, Error Control ID: file-error.

Practice 18 (674417e04df0ff511ced9e3f)

Use Case: Implement live validation feedback (e.g., as the user types).

Functional Flow: Display errors instantly for invalid inputs.
UI: Input ID: live-feedback-input, Error Message: *Invalid value*, Error Control ID: live-error.

Practice 19 (674417e04df0ff511ced9e40)

Use Case: Validate a multi-field relationship (e.g., end date must be after start date).

Functional Flow: Compare two date fields and validate accordingly.
UI: Start Date ID: start-date-input, End Date ID: end-date-input, Error Message: *End date must be after start date*, Error Control ID: date-error.

Practice 20

Dynamic Contact Information (Phone Numbers)

Business Logic:

The user can add multiple phone numbers.
At least one phone number must be entered, and it must be a valid 10-digit number.
Each phone number should be validated as the user types.

Functional Flow:

The user can click the "Add Phone Number" button to add a new phone number field.
The "Remove" button should remove the corresponding phone number.
Validation should trigger for each phone number when the user tries to submit the form.

UI and Control IDs:

Label: Contact Phone Numbers
Field ID: phone-array
Input ID: phone-input-{i} (where {i} is the dynamic index)
Error Message: "Phone number must be 10 digits"
Error Control ID: phone-error-{i}
Add Button ID: add-phone-button
Remove Button ID: remove-phone-button-{i}
Submit Button ID: submit-button

Validation:

Rule: At least one phone number is required.
Pattern: Only 10-digit numeric values are allowed.

Practice 21

Question 2: Dynamic Address List (Multiple Addresses)

Business Logic:

The user can add multiple addresses (Home, Office, etc.).
Each address should have a street, city, and zip code field.
Zip code must follow the pattern of 5 digits or 5-4 digits (e.g., 12345 or 12345-6789).

Functional Flow:

The user can click the "Add Address" button to add a new address group.
The "Remove" button should remove the corresponding address group.
The form should validate that all address fields (street, city, zip code) are filled before submitting.

UI and Control IDs:

Label: Addresses
Field ID: address-array
Street Input ID: street-input-{i}
City Input ID: city-input-{i}
Zip Input ID: zip-input-{i}
Error Message: "Zip code must be in the format 12345 or 12345-6789"
Error Control ID: zip-error-{i}
Add Button ID: add-address-button
Remove Button ID: remove-address-button-{i}
Submit Button ID: submit-button

Validation:

Rule: Zip code must be valid.
Pattern: Zip code must match ^\d{5}(-\d{4})?$.

Practice 22

Dynamic Skill Set (Multiple Skills)

Business Logic:

The user can add multiple skills.
Each skill should have a name (required) and a skill level (1-5).
Skill level must be a numeric value between 1 and 5.

Functional Flow:

The user can click the "Add Skill" button to add a new skill.
The "Remove" button should remove the corresponding skill.
The form should ensure that all skill levels are valid when submitting.

UI and Control IDs:

Label: Skills
Field ID: skills-array
Skill Name Input ID: skill-name-input-{i}
Skill Level Input ID: skill-level-input-{i}
Error Message: "Skill level must be between 1 and 5"
Error Control ID: skill-level-error-{i}
Add Button ID: add-skill-button
Remove Button ID: remove-skill-button-{i}
Submit Button ID: submit-button

Validation:

Rule: Skill level must be between 1 and 5.
Range: Skill level should be a number between 1 and 5.

Practice 23

Dynamic Payment Methods (Multiple Cards)

Business Logic:

The user can add multiple credit card details.
Each card should have a card number (16 digits), expiry date, and CVV.
The card number should be validated for the correct length and format (16 digits).
The expiry date should not be in the past.

Functional Flow:

The user can click the "Add Card" button to add a new card.
The "Remove" button should remove the corresponding card.
The form should validate each card's information (card number, expiry date, CVV) when the user tries to submit.

UI and Control IDs:

Label: Credit Cards
Field ID: cards-array
Card Number Input ID: card-number-input-{i}
Expiry Date Input ID: expiry-date-input-{i}
CVV Input ID: cvv-input-{i}
Error Message: "Invalid card number (16 digits required)" or "Expiry date must be in the future"
Error Control ID: card-number-error-{i}, expiry-date-error-{i}, cvv-error-{i}
Add Button ID: add-card-button
Remove Button ID: remove-card-button-{i}
Submit Button ID: submit-button

Validation:

Card Number: Must be exactly 16 digits.
Expiry Date: Must be a valid date in the future.
CVV: Must be 3 digits.

Practice 24

Dynamic Emergency Contacts (Multiple Contacts)

Business Logic:

The user can add multiple emergency contacts.
Each contact must have a name, phone number, and relationship.
The phone number should be validated as a 10-digit number.
The relationship field should have a set of predefined options (e.g., "Mother", "Father", "Friend").

Functional Flow:

The user can click the "Add Contact" button to add a new contact.
The "Remove" button should remove the corresponding contact.
The form should validate that all fields (name, phone, relationship) are filled before submission.

UI and Control IDs:

Label: Emergency Contacts
Field ID: emergency-contacts-array
Name Input ID: contact-name-input-{i}
Phone Input ID: contact-phone-input-{i}
Relationship Input ID: contact-relationship-input-{i}
Error Message: "Phone number must be 10 digits"
Error Control ID: contact-phone-error-{i}
Add Button ID: add-contact-button
Remove Button ID: remove-contact-button-{i}

Submit Button ID: submit-button

Validation:

Phone Number: Must be 10 digits.
Relationship: Must be a valid selection from the predefined list.

Assignment Exercise

Assignment 1 (65cdfdab7ba36e06448762a6)
1. Create a Reactive form with validations which are provided Below

Student Form: control id -> #StudentForm,
Username: control id -> #email,
Password: control id -> #password,
Date of Birth: control id -> #dob,
Phone: control id -> #phone,
Address: control id -> #address

student List: control id -> #studentTable

submit: button with type submit

validations :

1. Required validations -> {control name} is required.
   Example: Address is required.

2. Other validations ->
   -Username must be at least 3 characters.,
   -Invalid email format.,
   -Password must be at least 8 characters.,
   -Date of Birth is required.,
   -Phone must be a 10-digit number.

After submitting details Forms info should be added in StudentList Array and displayed in the Table Form

Assignment 2 (65cdfdab7ba36e06448762a7)

CRUD Operations for Managing Employee Records

Objective: Create a form for adding and editing employee records with full CRUD functionality.

You are tasked with building a simple Employee Management system that allows you to manage employee records effectively. The system should provide the following functionalities:

Add Employee: Use a Reactive Form to add new employees, including their name, email, and department.
Edit Employee: When an employee record is clicked, the form should be populated with the selected employee's details using patchValue.
Delete Employee: Ability to remove an employee from the list.
Display Employees: Show the list of employees in a tabular format.

Your Task

Create the Reactive Form with the following fields:

Name: Control ID - employeeName
Email: Control ID - employeeEmail
Department: Control ID - employeeDepartment

Implement the following:

Use patchValue to populate the form when editing an employee.
Implement logic to add, edit, and delete employee records.
Display the list of employees in a table format with actions:

Edit Button Control ID - editEmployeeButton-<id> (replace <id> with the actual employee ID)
Delete Button Control ID - deleteEmployeeButton-<id> (replace <id> with the actual employee ID)
Dynamic ID for each Employee Record: Each record should have a unique identifier in the format employee-<id>.

UI Control IDs

Field	Control ID	Error Control ID	Validation Rule	Err
Employee Name	employeeName	err-employee-name	Required	"Er
Employee Email	employeeEmail	err-employee-email	Required, Email format	"V
Employee Department	employeeDepartment	err-employee-department	Required	"D
Add Employee Button	addEmployeeButton		N/A	N/
Success Message	success-add-employee		N/A	"Er
Update Success Message	success-update-employee		N/A	"Er
Delete Success Message	success-delete-employee		N/A	"Er
Employee Record ID	employee-<id>		N/A	N/

Functional Flow

Adding Employee:

Input values for Employee Name, Email, and Department in the respective fields.
Click the "Add" button identified by addEmployeeButton.
Validate all fields before submission. If validation passes, add the employee record and display the success message in success-add-employee. If validation fails, display error messages using their respective Error Control IDs.

Editing Employee:

Click on an employee entry to edit.
Use patchValue to populate the fields with the selected employee's details.
Make any necessary edits and click the "Update" button identified by updateEmployeeButton-<id> (where <id> represents the actual employee ID).
Validate inputs and update the employee record, displaying the success message in success-update-employee.

Deleting Employee:

Click the "Delete" button identified by deleteEmployeeButton-<id> (where <id> represents the actual employee ID) next to an employee record.
After deleting, display the success message in success-delete-employee.

Display Employee List:

Render the list of employees in a table format where each employee row has a unique identifier such as employee-<id> (replace <id> with the actual employee ID).

Validation Elements

Validate that each field is filled out before submission.
Display error messages in the specified Error Control IDs (e.g., err-employee-name, err-employee-email, err-employee-department) depending on the specific field that fails validation.

Business Logic

Employee IDs should be auto-incremented starting from 1.
Email should follow proper email formatting for successful validation.
Ensure that deletion requires confirmation to prevent accidental removals.

Selectors for Test Cases

The following control IDs are necessary for automated test case generation:

Control ID	Purpose
employeeName	Input for employee name.
employeeEmail	Input for employee email.
employeeDepartment	Input for employee department.
addEmployeeButton	Button to add a new employee.
success-add-employee	Used for success message after adding employee.
success-update-employee	Used for success message after updating employee.
success-delete-employee	Used for success message after deleting employee.
err-employee-name	Shows error for employee name input.
err-employee-email	Shows error for employee email input.
err-employee-department	Shows error for department selection.
employee-<id>	Unique ID for each employee record in the list.
editEmployeeButton-<id>	Dynamic edit button ID for each employee.
updateEmployeeButton-<id>	Button that updates the information of employee.
deleteEmployeeButton-<id>	Dynamic delete button ID for each employee.

By following these detailed requirements, you will ensure the successful implementation and testing of the Employee Management System using Angular.

Assignment 3

CRUD Operations for Managing Products

Objective: Implement a product management system where users can add, edit, and delete products.

Question:
You need to implement a **Product Management** system where users can:

- Add a new product with name, price, description.
- Edit a product's details by selecting it from the list, which will populate the form using patchValue.
- Delete a product from the list.
- Display products in a table.
- Your task is to:
- Build a form for adding products with the following fields:
  - Product Name (productName)
  - Price (productPrice)
  - Description (productDescription)
- Use patchValue to pre-fill the form when editing a product.

- Implement add, edit, and delete actions for products.
- Display the list of products in a table with columns: Product Name, Price, Description, and Actions (Edit, Delete).

UI Control IDs:

- Form Inputs:
  - Product Name: productName
  - Price: productPrice
  - Description: productDescription
- Table Columns:
  - Product Name: productNameColumn
  - Price: productPriceColumn
  - Description: productDescriptionColumn
  - Edit Button: editProductButton
  - Delete Button: deleteProductButton

Assignment 4

CRUD Operations for Managing Customer Feedback

Objective: Build a feedback form that allows users to submit, edit, and delete customer feedback.

Question:
Create a **Customer Feedback Management** system where users can:

- Submit a new feedback entry with customer name, feedback message, and rating (1-5).
- Edit a feedback entry by selecting it from the list and using patchValue to load its details into the form.
- Delete a feedback entry.
- Display feedback entries in a table.
- Your task is to:
- Implement a form with the following fields:
  - Customer Name (customerName)
  - Feedback Message (feedbackMessage)
  - Rating (feedbackRating)
- Use patchValue to load the selected feedback's details when editing.
- Implement actions for submitting, editing, and deleting feedback.
- Display the list of feedback in a table with columns: Customer Name, Message, Rating, and Actions (Edit, Delete).

UI Control IDs:

- Form Inputs:
  - Customer Name: customerName
  - Feedback Message: feedbackMessage
  - Rating: feedbackRating
- Table Columns:
  - Customer Name: feedbackCustomerNameColumn
  - Feedback Message: feedbackMessageColumn
  - Rating: feedbackRatingColumn
  - Edit Button: editFeedbackButton
  - Delete Button: deleteFeedbackButton

Assignment 5

Student Record Management with Reactive Form

Objective: Build a student record management system where users can add, edit, and delete student records.

Question:
You need to implement a **Student Management** system with the following features:

- Add new student records with name, email, course, and grade.
- Edit an existing student record by selecting it from the list. When selected, the student's data should be patched into the form using patchValue.
- Delete a student record.
- Display the students in a table format.
- Your task is to:
- Create a Reactive Form with the following fields:
    - Name (studentName)
    - Email (studentEmail)
    - Course (studentCourse)
    - Grade (studentGrade)
- Use patchValue to populate the form when editing.
- Implement add, edit, and delete functionality for student records.
- Display the student records in a table with columns: Name, Email, Course, Grade, and Actions (Edit, Delete).

UI Control IDs:

- Form Inputs:
    - Name: studentName
    - Email: studentEmail
    - Course: studentCourse
    - Grade: studentGrade
- Table Columns:
    - Name: studentNameColumn
    - Email: studentEmailColumn
    - Course: studentCourseColumn
    - Grade: studentGradeColumn
    - Edit Button: editStudentButton
    - Delete Button: deleteStudentButton

Assignment 6

CRUD Operations for Task Management System

Objective: Implement a simple task management system where users can add, edit, and delete tasks.

Question:
Create a **Task Management** system where users can:

- Add tasks with a title, description, and due date.
- Edit a task by selecting it from the list, which will populate the form using patchValue.
- Delete a task from the list.
- Display tasks in a table format.
- Your task is to:
- Build a form to create tasks with the following fields:
    - Title (taskTitle)
    - Description (taskDescription)
    - Due Date (taskDueDate)
- Use patchValue to auto-fill the form with the selected task's details when editing.
- Implement actions for adding, editing, and deleting tasks.
- Display tasks in a table with columns: Title, Description, Due Date, and Actions (Edit, Delete).

UI Control IDs:

- Form Inputs:
    - Title: taskTitle
    - Description: taskDescription
    - Due Date: taskDueDate
- Table Columns:
    - Title: taskTitleColumn
    - Description: taskDescriptionColumn
    - Due Date: taskDueDateColumn
    - Edit Button: editTaskButton
    - Delete Button: deleteTaskButton

Online Reference

No online Reference

Module 5

What to learn

Form Array
FormBuilder
Validation in Reactive Form

Practice Exercise

Practice 1 (65cdfdab7ba36e06448762aa)
Do the hands on from the following url for Reactive Form and Validating Form

https://angular.io/guide/reactive-forms

Create A Reactive Form and try to update partialy some field using patch value

Assignment Exercise

Assignment 1 (65cdfdab7ba36e06448762a9)
Create a Student Admission Form with the Following Field

Student Information: control id -> #studentForm
firstName: control id -> #firstName
middleName: control id -> #middleName
lastName: control id -> #lastName
DOB: control id -> #dob
place of birth: control id -> #placeOfBirth
first Language: control id -> #firstLanguage
city name: control id -> #city
state name: control id -> #state
country name: control id -> #country
pincode: control id -> #pin

Father Information:
father fullName: control id -> #fatherFullName
father email: control id -> #fatherEmail
father education qualification: control id -> #fatherEducation
father profession: control id -> #fatherProfession

father designation: control id -> #fatherDesignation
father phone number: control id -> #fatherPhone

Mother Information:
mother fullName: control id -> #motherFullName
mother email: control id -> #motherEmail
mother education qualification: control id -> #motherEducation
mother profession: control id -> #motherProfession
mother designation: control id -> #motherDesignation
mother phone number: control id -> #motherPhone

Emergency Contact: (formArrayName)
relation: control id -> #"'emergencyContact' + {i}"
*Dropdown with value (Sibling, Uncle, Aunt, Grandparent)

contact no: control id -> #"'emergencyNumber' + {i}"

Buttons:
Add More: control id -> #add-emergencyContacts (Add more Emergency Contact)
Submit: control id -> #submit

validations:

1. Required -> {control name} is required (for all fields)
2. Email -> {control name} email format is invalid (for email type)
3. Phone number -> {control name} number should 10 characters (for phone number pattern)

Online Reference

No online Reference

Module 6

What to learn

Passing Data from Parent to Child Component
Passing Data from Child to Parent Component using EventEmiter
Custom Pipes
Custom Directives

Practice Exercise

Practice 1
Do the hands on from the following link to understand the concept @input and @output decorator https://angular.io/start
Practice 2
Create a custom pipe named capitalize that capitalizes the first letter of a given string. Use it in a template to transform "hello world" to "Hello world".

Practice 3
Write a custom pipe called reverse to reverse a given string. For example, "Angular" should become "ralugnA".

Practice 4
Build a custom pipe named truncate to shorten a string to a specified length and append "..." if truncated. Pass the length as an argument.

Practice 5
Create a pipe named filter that filters an array of strings based on a search keyword. Demonstrate its use in an *ngFor loop.

Practice 6
Write a custom pipe called square that takes a number and returns its square. Use it in a template.

Practice 7
Develop a custom pipe named currencyConvert to convert an amount from one currency to another. Pass the conversion rate as an argument.

Practice 8
Create an orderBy pipe to sort an array of objects by a given property (e.g., sort by name or age).

Practice 9
Write a custom pipe evenFilter that filters an array of numbers and returns only even numbers. Use it in a component.

Practice 10
Implement a pipe named toDateString that converts a timestamp to a human-readable date format. Use it with a timestamp like 1633024800.

Practice 11
Build a pipe called mask that masks a credit card number, showing only the last 4 digits. For example, 1234567812345678 should become ************5678.

Practice 12
Create a Basic Directive

Create a directive named highlight that changes the background color of an element to yellow when it is applied.

Practice 13
Add Event Listener

Write a directive that listens for a click event on an element and logs "Element clicked!" to the console.

Practice 14
Change Text Color

Build a directive that changes the text color of an element to red.

Practice 15
Accept Input Property

Create a directive appBorder that accepts an input property for border color and applies it to the element.

Practice 16
Toggle CSS Class

Implement a directive that toggles a CSS class active on an element when clicked.

Practice 17
Show/Hide on Hover

Write a directive that shows an element on hover and hides it otherwise.

Practice 18
Conditional Style

Develop a directive that applies a green background if a boolean input property isValid is true, and red if false.

Practice 19
Log Mouse Coordinates

Write a directive that logs the x and y coordinates of the mouse whenever it moves over an element.

Practice 20
Dynamic Font Size

Create a directive that adjusts the font size of an element based on an input property fontSize.

Practice 21

Disable Button

Build a directive that disables a button if the input property isDisabled is true.

Practice 22

Change Content Dynamically

Create a directive that changes the text content of an element to "Hello World" on double-click.

Practice 23

Listen to Focus Event

Write a directive that listens for the focus event on an input field and changes its border to blue.

Practice 24

Style Multiple Elements

Implement a directive that applies a bold font weight to all child elements of a container.

Practice 25

Apply Gradient Background

Build a directive that applies a gradient background with two input colors to an element.

Practice 26

Restrict Input

Create a directive that restricts input to numeric characters only.

Practice 27

Highlight Active Element

Develop a directive that highlights the currently focused input field with a green border.

Practice 28

Scroll Listener

Write a directive that logs a message to the console when a user scrolls within a div.

Practice 29

Rotate Element

Create a directive that rotates an element by 45 degrees when clicked.

Practice 30

Animate Width

Build a directive that gradually increases the width of an element when hovered over.

Practice 31

Detect Outside Click

Develop a directive that detects clicks outside an element and logs a message to the console.

Assignment 1

Reference to the day10 Task. In the app component call Student Component and StudentList Component. Use Event Emitter to pass data from student component to the app component. From app component send data to studentList component using @Input decorator.

Assignment 2

Parent-to-Child Communication (@Input)

Use case: A parent component manages a list of products. The child component displays product details.

Question: Implement a parent component that passes a list of products to a child component using @Input. The child displays the details of the selected product.

Assignment 3

Two-Way Binding

Use case: A user updates their profile information using a form component.

Question: Create a form component with two-way binding to allow users to update their profile details (e.g., name, email). Reflect the changes in the parent component when the form is submitted.

Assignment 4

Service-Based Data Sharing

Use case: Two sibling components need to share a selected product's details.

Question: Use a shared service to enable data communication between two sibling components. Simulate the selection of a product in one component and display its details in the other.

Assignment 5

Dynamic Component Rendering

Use case: Display different types of notifications (success, error, info) in a notification container component.

Question: Implement a notification service and container that dynamically renders notification components based on type.

Assignment 6

Dynamic Component Interaction with @ViewChild

Use case: A parent component dynamically adds and interacts with multiple child components.

Question: Create a parent component that dynamically adds child components to a list. Use @ViewChild to call a method (e.g., highlight()) on a selected child component.

Assignment 7

Parent Controlling Child Visibility

Use case: A parent component shows or hides a child component based on user actions.

Question: Create a child component that displays a message. Use @Input in the child to control its visibility from the parent.

Assignment 8

Child Passing Events to Parent

Use case: A child component contains a button to mark a task as completed.
Question: Implement a task child component that uses @Output to notify the parent component when a task is marked as completed.

Assignment 9

Synchronizing Input and Output

Use case: A parent component controls the state of a child toggle component.
Question: Create a toggle component that uses @Input for the initial state and @Output to notify the parent of state changes when toggled.

Online Reference

https://angular.io/api/core/EventEmitter

https://dzone.com/articles/understanding-output-and-eventemitter-in-angular

https://dzone.com/articles/understanding-output-and-eventemitter-in-angular

https://angular.io/api/core/ViewChild

Module 7

What to learn

Dependency Injection
Creating Service
Hierarchical DI in Angular
Injection service into another service
Registering Value Data Service
Using Observable to pass values

Practice Exercise

Practice 1
Do the hands on from the following url https://angular.io/guide/dependency-injection https://angular.io/guide/dependency-injection-providers
Practice 2

User Authentication System

Use Case: Build a user authentication system where a **LoginService** handles login, and a **UserService** manages the user data after login.

Business Logic:
Implement a login form using Reactive Forms. Upon successful login, the **LoginService** will fetch user credentials and pass them to the **UserService** to fetch and store the user's profile.
If login fails, show an error message.
UI:
Login form with fields: **Username**, **Password**
Upon successful login, display the user name and email from the **UserService**.
Control IDs:

Login Form:
Username: loginUsername
Password: loginPassword
Submit Button: loginSubmit
User Profile:
User Name: userNameDisplay
User Email: userEmailDisplay
Practice 3

Product Catalog with Search Functionality

Use Case: You need to display a list of products. Create a **ProductService** that fetches the product list and a **SearchService** to filter products based on the search term.

- **Business Logic**:
  - Implement a form to search for products by name.
  - Use **SearchService** to filter products based on user input.
  - The **ProductService** will fetch products when the component is loaded, and **SearchService** will filter them dynamically.
- **UI**:
  - Product list displayed in a table.
  - Search bar at the top to filter products by name.

Control IDs:

- Search Form:
  - Search Input: searchProduct
  - Search Button: searchButton
- Product Table:
  - Product Name: productNameColumn
  - Product Price: productPriceColumn
  - Edit Button: editProductButton
  - Delete Button: deleteProductButton

Practice 4

Customer Feedback Form

Use Case: Build a customer feedback form where the **FeedbackService** saves and fetches customer feedback.

- **Business Logic**:
  - Upon form submission, save the feedback using the **FeedbackService**.
  - The service should save feedback data and also fetch all submitted feedback to display in a table.
- **UI**:
  - A form for customer name, feedback, and rating (1-5).
  - Display all feedback in a table with the ability to edit or delete.

Control IDs:

- Feedback Form:
  - Customer Name: customerName
  - Feedback Message: feedbackMessage
  - Rating: feedbackRating

        - Submit Button: submitFeedbackButton
    - Feedback Table:
        - Customer Name: feedbackCustomerNameColumn
        - Feedback Message: feedbackMessageColumn
        - Rating: feedbackRatingColumn
        - Edit Button: editFeedbackButton
        - Delete Button: deleteFeedbackButton

Practice 5

Order Management System

Use Case: Manage orders for an e-commerce site. **OrderService** manages the order details and **PaymentService** handles payment processing.

- **Business Logic**:
    - On order submission, use the **OrderService** to save the order, and **PaymentService** to handle payment status.
    - Display a list of orders with their status (pending, processed, completed).
- **UI**:
    - Form to place an order (order details, payment status).
    - Display all orders in a table with their status.

Control IDs:

- Order Form:
    - Order Item: orderItem
    - Quantity: orderQuantity
    - Payment Status: paymentStatus
    - Submit Button: placeOrderButton
- Orders Table:
    - Order ID: orderIdColumn
    - Order Item: orderItemColumn
    - Quantity: orderQuantityColumn
    - Payment Status: orderPaymentStatusColumn
    - Edit Button: editOrderButton
    - Delete Button: deleteOrderButton

Practice 6

Employee Time Tracking

Use Case: Track employee working hours. Use **TimeTrackingService** to log time-in and time-out for employees.

- **Business Logic**:
    - Create a form that records the **time-in** and **time-out** for employees.

        - **TimeTrackingService** will store the logs and calculate total hours worked.
        - Display all logs in a table with employee name and total hours worked.
    - **UI**:
        - A time-in/time-out form.
        - A table that displays employee hours worked.

Control IDs:

- Time Tracking Form:
    - Employee Name: employeeName
    - Time In: timeIn
    - Time Out: timeOut
    - Submit Button: submitTimeLogButton
- Time Logs Table:
    - Employee Name: employeeNameColumn
    - Time In: timeInColumn
    - Time Out: timeOutColumn
    - Total Hours Worked: totalHoursWorkedColumn

Practice 7

Observables: User Dashboard with Real-time Data

Use Case: You are building a user dashboard that fetches data (e.g., user activities, notifications) in real-time using Observables.

- **Business Logic**:
    - Create a **UserService** that returns user activity data and notifications using an **Observable**.
    - The component will subscribe to the Observable and update the dashboard when new data arrives.
    - Implement a **refresh button** to manually trigger the refresh of data.
- **UI**:
    - A **User Dashboard** with sections for user activity and notifications.
    - **Refresh Button** to fetch new data.

Control IDs:

- Dashboard Sections:
    - Activity List: activityListSection
    - Notifications List: notificationsListSection
    - Refresh Button: refreshButton
- Data Display:
    - Activity Item: activityItem

Notification Item: notificationItem

Practice 8

Observables: Live Search for Products

Use Case: Implement a live search for products that filters results as the user types in the search box, using an **Observable** for real-time data.

Business Logic:

Create a **ProductService** that returns an Observable of product data.

The component subscribes to the search Observable, triggering the **ProductService** each time the user types a new search term.

Display the filtered results dynamically.

UI:

Search Box to input the search term.

Product List that updates in real-time as the search term is entered.

Control IDs:

Search Box:

Search Input: searchInput

Search Button: searchButton

Product List:

Product Name: productNameColumn

Product Price: productPriceColumn

Add to Cart Button: addToCartButton

Practice 9

Observables: Fetching User Details with RxJS Operators

Use Case: Fetch user details from a server using **Observables** and **RxJS operators** like **map** and **catchError** to process data and handle errors.

Business Logic:

Create a **UserService** that returns user data from an API endpoint.

Use RxJS operators like **map** to transform data and **catchError** to handle errors gracefully.

Display the user data or an error message in the component.

UI:

Display the user's name, email, and other details if the fetch is successful.

Show an error message if the request fails.

Control IDs:

User Data:

User Name: userName
User Email: userEmail
User Address: userAddress

Error Message:

Error Message: errorMessage

Practice 10

Observables: Stock Price Tracker

Use Case: Implement a stock price tracker that updates the price in real-time using **Observables**.

- **Business Logic**:
    - Create a **StockService** that emits stock price updates at regular intervals (using **setInterval** or a similar approach) as an **Observable**.
    - Display the current price of a stock, and allow the user to subscribe to the Observable to receive price updates.
- **UI**:
    - **Stock Price Display** with current price.
    - **Subscribe Button** to start receiving updates.
    - **Unsubscribe Button** to stop receiving updates.

Control IDs:

- Stock Price Display:
    - Stock Price: stockPriceDisplay
- Subscription Buttons:
    - Subscribe Button: subscribeButton
    - Unsubscribe Button: unsubscribeButton

Practice 11

Observables: Real-Time Chat Application

Use Case: Implement a real-time chat application where messages are sent and received using **Observables**.

- **Business Logic**:
    - Create a **ChatService** that emits new messages via an **Observable** whenever a new message is sent.
    - Use **RxJS operators** to filter, map, or combine messages.
    - The component subscribes to the Observable and updates the chat window with new messages.
- **UI**:
    - **Chat Input Box** to type messages.
    - **Message List** that updates in real-time.

Send Button to send messages.

Control IDs:

Chat Window:

Message Input: messageInput
Send Button: sendMessageButton
Message List: messageList
Individual Message: messageItem

Practice 12

Hierarchical DI: Managing User Preferences Across Multiple Components

Use Case: You need to manage user preferences (theme, language) across multiple components using Angular's **Hierarchical DI**. A shared service should be injected into different parts of the application but should maintain separate states based on the component hierarchy.

Business Logic:

Create a service that manages user preferences like theme and language.
In the root component, the service should hold the default preferences.
For a child component, inject the same service but allow it to override the preferences for that component only.

UI:

Main App Component that displays the current theme and language.
Child Component where users can change their theme and language independently.

Control IDs:

Main App:

Current Theme: currentTheme
Current Language: currentLanguage

Child Component:

Theme Dropdown: themeDropdown
Language Dropdown: languageDropdown

Assignment Exercise

Assignment 1

Create StudentService which will contains operation for crud operation using student type Array.

Assignment 2

Create a Log service which will be injected by StudentService on every crud operation will console list message in console. Inject student service Student Component and StudentList Component

Online Reference

No online Reference

Module 8

What to learn

Routing
Generate an app with routing enabled
Importing your new components into AppRoutingModule
Defining a basic route
Define your routes in your Routes array.
Add your routes to your application.
Template with routerLink and router-outlet
Getting route information
Router
ActivatedRoute
ParamMap
Setting up redirects

Practice Exercise

Practice 1
Do the hands for tours of heroes Application from the following link. (Cover section milestone1) https://angular.io/guide/router-tutorial-toh
Practice 2
Use Case:
When a user logs in, navigate them to:

/welcome if it's their first login.
/dashboard for subsequent logins.

Key Steps:

Check a flag in local storage or an API (isFirstLogin).
Redirect to the appropriate route based on the flag.

Practice 3

Order Management with Filtered List

Use Case:
Create a route /orders with query parameters ?status=pending or ?status=completed to filter orders.

Display orders dynamically based on the status query parameter.

Key Steps:

Use ActivatedRoute to read query parameters.
Call a service to fetch filtered orders.

Practice 4

E-commerce Product Pagination

Use Case:
Build a product list page with pagination using routes like /products?page=1.

Fetch and display products for the specified page.

Key Steps:

Read the page query parameter and dynamically load products for the given page.
Add navigation buttons to switch pages.

Practice 5

Dynamic Tab Navigation in a Profile Page

Use Case:
Implement a user profile with tabs like "Details", "Orders", and "Settings".

Use child routes such as /profile/details, /profile/orders, and /profile/settings.
Highlight the active tab dynamically.

Key Steps:

Create child routes for tabs.
Use routerLinkActive to indicate the active tab.

Practice 6

Admin Panel with Restricted Access by Role

Use Case:
In an admin panel, restrict access to specific sections:

/admin/users for Admin users.
/admin/reports for Manager users.
Redirect unauthorized users to an "Access Denied" page.

Key Steps:

Use a route guard to check user roles dynamically.
Configure routes based on role.

Practice 7

Checkout Flow with Step Validation

Use Case:
Implement a multi-step checkout flow:

/checkout/cart
/checkout/shipping
/checkout/payment
Ensure users can’t proceed to /checkout/payment without completing /checkout/shipping.

Key Steps:

Use route guards to enforce step validation.
Pass data between steps using a shared service.

Practice 8

Event Calendar with Date Navigation

Use Case:
Create a calendar with routes like /events?date=2024-12-01 to display events for a specific date.

Add "Next Day" and "Previous Day" buttons to navigate dates.

Key Steps:

Use query parameters for the date.
Fetch events for the given date and update the calendar dynamically.

Practice 9

Favorite Items with State Persistence

Use Case:
Build a route /favorites to display a user's favorite items.

Allow toggling of items between "Favorite" and "Not Favorite".
Persist the state locally or via an API.

Key Steps:

Fetch favorite items on route load.
Update the list dynamically based on user actions.

Practice 10

Content Preview with Secure Access

Use Case:
Create a route /preview/:id that displays restricted content previews for specific users.

Ensure only users with valid access tokens can view the preview.

Key Steps:

Implement a route guard that validates tokens.
Redirect unauthorized users to /login.

Practice 11

Dynamic Redirect Based on User Action

Use Case:
For an application with a "Contact Us" form, after submission:

- Redirect to /thank-you if the submission is successful.
- Redirect to /form-error if it fails.

Key Steps:

- Use the Router to programmatically navigate based on the form response.

Assignment Exercise

Assignment 1
Tutorial site task site needs to be implemented. Application should have topnav bar which contains html/CSS/javascript Respective daywise assignment should be loaded in the router outlet.

Assignment 2

Use Case: Employee Management System with Filter Logic

Scenario:

You are tasked with creating an **Employee Management System** where users can view, add, edit, and delete employee data, as well as manage the projects they are assigned to. The system includes **nested routes** and **filter functionality** to manage employees based on their **status** and **position**.

Routes:

- **Employee List**
  - **Route**: /employees
    - **Description**: Displays a list of all employees.
    - **Actions**:
      - Add, Edit, View, and Delete Employee.
      - **Filter Functionality**:
        - Filters by **Employee Status** (e.g., Active, Inactive).
        - Filters by **Position** (e.g., Developer, Manager).
      - **Business Logic**:

            - Display employees based on selected filters.
            - Automatically sort employees by **Joining Date**.
- **Employee Details with Nested Tabs**
    - **Route**: /employees/:id
        - **Description**: Displays employee details with nested tabs for Personal Info and Assigned Projects.
        - **Tabs**:
            - **Personal Info**: /employees/:id/personal
                - Show employee's personal information such as name, email, phone number, and position.
            - **Assigned Projects**: /employees/:id/projects
                - Display all projects assigned to the employee, with options to **add** or **remove** projects.
        - **Business Logic**:
            - Disable the **Assigned Projects** tab if the employee has no assigned projects yet.
- **Add New Employee**
    - **Route**: /employees/new
        - **Description**: Opens a form to add a new employee.
        - **Business Logic**:
            - **Field Validation**:
                - Ensure that the **email** is unique across the system before submission.
                - Ensure that **position** is selected from a predefined list (e.g., Manager, Developer).
                - Ensure **phone number** is unique.
            - Redirect to the employee’s details page upon successful creation (/employees/:id).
- **Edit Employee**
    - **Route**: /employees/:id/edit
        - **Description**: Opens a form to edit employee details.
        - **Business Logic**:
            - Validate the **email** and **phone number** fields before submitting.
            - Disable the **email** and **position** fields for employees with "Manager" status.

        - Display a **confirmation** prompt before allowing an edit.
        - **Redirect** to the employee details page after successful edit (/employees/:id).
- **Assign Project to Employee**
    - **Route**: /employees/:id/projects/assign
        - **Description**: Opens a form to assign a project to the employee.
        - **Business Logic**:
            - Ensure the employee does not already have the project assigned.
            - Prevent assigning the same project multiple times.
            - **Filter projects** by status (e.g., Active, Completed) before displaying them in the form.

Additional Features & Business Logic:

Filter Logic on Employee List:

- **Filter by Status**: Allow users to filter employees based on their status (Active, Inactive).
- **Filter by Position**: Allow users to filter employees by their position (e.g., Developer, Manager).
- **Sort Employees**: Automatically sort the employee list by their **Joining Date**, with the most recent joiners at the top.
- **Pagination**: Implement pagination to limit the number of employees displayed per page, especially useful when the number of employees is large.
- **Search**: Allow users to search for employees by **name** or **email**.

Confirmation Dialog for Deleting Employees:

- When a user attempts to delete an employee, show a **confirmation dialog**:
    - "Are you sure you want to delete this employee?"
    - Only allow deletion if the employee is **Active**.
    - If the employee is assigned to projects, ask whether to remove the employee from those projects before deletion.

Redirect Logic:

- After adding or editing an employee, redirect the user to the **Employee Details** page:
    - For new employees: /employees/:id.
    - For edited employees: /employees/:id.

Handling Errors:

If a submission fails (for adding or editing an employee), display a meaningful error message explaining why the operation failed (e.g., "Email already exists" or "Invalid phone number").

Expected User Flow:

Employee List Page (/employees):

The user lands on the employee list page, where they can filter by status or position, search, and paginate through employees.
The user can click on an employee to view or edit details.

Employee Details Page (/employees/:id):

When the user clicks on a specific employee, they are directed to the details page.
If the employee has no assigned projects, the "Assigned Projects" tab is disabled.
The user can navigate to the "Personal Info" or "Assigned Projects" tabs, where they can modify or view the employee's data.

Add/Edit Employee Page (/employees/new or /employees/:id/edit):

The user fills out a form to either add a new employee or edit an existing one.
The form includes validation for uniqueness of email and phone number, and appropriate field disabling (e.g., for managers).

Assign Project Page (/employees/:id/projects/assign):

The user can assign a project to an employee. Only active, non-duplicate projects are shown.
After assigning, the employee's assigned projects are updated in the database, and the user is redirected to the employee details page.

Business Logic Summary:

Filtering: Allows users to filter employees by status and position.
Field Validation: Ensures data integrity, such as unique email, phone, and position.
Tab Disabling: The "Assigned Projects" tab is disabled if no projects are assigned.
Conditional Deletion: Employees can only be deleted if they are **Active**, and the system will check for project assignments before allowing deletion.
Redirecting: After adding or editing an employee, the user is redirected to the details page.

Online Reference

No online Reference

Module 9

What to learn

Nested Routes
Using relative paths

Practice Exercise

Practice 1
Do the hands on from the following link https://www.tektutorialshub.com/angular/angular-child-routes-nested-routes/

Assignment Exercise

Assignment 1
With reference to above assignment, for the HTML/CSS and javascript create seprate sidebar component which will contain router-outlet and display the specific content day wise. Use bootstrap

Online Reference

https://angular.io/guide/router

Course Project-Forms

Using Pipes to Transform Output

Making Http Requests

Module 10

What to learn

Routing module
Integrate routing with your app

Practice Exercise

Practice 1
Do the hands on from following url https://angular.io/guide/router-tutorial-toh for milestone2 and 3
Practice 2
Implement an **Auth Guard** in Angular to protect the **Admin Dashboard** (/admin) and **User Dashboard** (/user) routes. The **Admin Dashboard** should be accessible only to users with the **Admin** role, while the **User Dashboard** should be accessible to any authenticated user. Redirect unauthenticated users to the **login page** (/login) when they try to access these protected routes.

Assignment Exercise

Assignment 1
With Reference to the Day16 Assignment Create a Separate Module for HTML/CSS and JavaScript and do the routing accordingly.
Assignment 2

Use Case: Angular Application with Auth Guard and Lazy Loading

Scenario:

You are tasked with building an **Angular application** with multiple modules. Some modules are publicly accessible, while others require authentication to access. You will need to implement **lazy loading** for efficiency and **auth guards** to protect specific routes.

Features:

Public Routes:

- **Home Page** (/home): Public route accessible by any user.

- **About Page** (/about): Public route accessible by any user.

Protected Routes:

- **Dashboard Page** (/dashboard): Only accessible to authenticated users.
- **Settings Page** (/dashboard/settings): A child route under **Dashboard**, also protected.

Modules:

- **Public Module** (PublicModule): Lazy-loaded module containing **Home** and **About** pages.
- **Dashboard Module** (DashboardModule): Lazy-loaded module containing **Dashboard** and **Settings** pages.
- **Auth Module**: Handles login and authentication logic.

Auth Guard:

- Protect the **Dashboard** and **Settings** routes with an **Auth Guard** that checks if the user is authenticated.
- If a user tries to access these routes without being authenticated, redirect them to the **login page**.

Login and Logout:

- A **login page** that allows users to enter their credentials and authenticate.
- A **logout function** that clears the authentication token and redirects the user to the **home page**.

Route Configuration:

- **App Routing** (app-routing.module.ts):
  - Set up routes for the **Public Module** (/home, /about) and **Dashboard Module** (/dashboard).
  - Lazy load the **PublicModule** for home and about pages.
  - Lazy load the **DashboardModule** for the dashboard and settings pages.
  - Protect **Dashboard** and **Settings** routes with the **Auth Guard.**

Function Flow:

Routing Flow:

- The **app-routing.module.ts** file configures the routes:
  - The root / redirects to the public pages (Home and About).
  - The /dashboard route is lazily loaded using loadChildren. It's protected by the **Auth Guard.**
  - The /dashboard/settings route is lazily loaded as a child of /dashboard, and it's also protected by the **Auth Guard**.

Auth Guard Flow:

- The **AuthGuard** intercepts the navigation to protected routes (/dashboard, /dashboard/settings):
    - If the user is authenticated (based on a token stored in local storage), the route is activated.
    - If the user is not authenticated, the **AuthGuard** redirects the user to the **login page** (/login).

Login Flow:

- On accessing the **login page**, the user is prompted to enter credentials (username, password).
- When the user submits the login form:
    - The **AuthService** checks if the credentials are valid.
    - If valid, the **AuthService** stores a token in **localStorage** to mark the user as authenticated.
    - The user is redirected to the **dashboard** page (/dashboard).
- If invalid, an error message is displayed, and the user stays on the login page.

Logout Flow:

- When the user clicks the **logout** button:
    - The **AuthService** clears the authentication token from **localStorage**.
    - The user is redirected to the **home page** (/home).

Protected Route Flow:

- If an authenticated user tries to access the **dashboard** or **settings** route directly:
    - The **AuthGuard** checks for the authentication token in **localStorage**.
    - If the token is valid, the route is activated, and the user can view the page.
    - If the token is not valid or absent, the user is redirected to the **login page**.

Implementation Steps:

Set Up App Routing (app-routing.module.ts):

- Create routes for **public** and **protected** pages.
- Use **lazy loading** for **PublicModule** and **DashboardModule**.
- Implement the **AuthGuard** on the **Dashboard** and **Settings** routes.

Create Auth Guard (auth.guard.ts):

- Implement the **AuthGuard** to protect routes.
- Redirect unauthorized users to the **login page**.

Create Auth Service (auth.service.ts):

- Implement methods for **login**, **logout**, and **checking authentication** status using **localStorage**.

Create Login Component (login.component.ts):

- Implement the login form, authenticate the user, and store the authentication token.

Create Public Module (public.module.ts):

- Implement the **Home** and **About** pages as lazily-loaded components.

Create Dashboard Module (dashboard.module.ts):

- Implement the **Dashboard** and **Settings** pages as lazily-loaded components.

Function Flow Diagram:

- **User navigates to /home or /about** → The route is publicly accessible and loads the respective component.
- **User navigates to /dashboard or /dashboard/settings** → Auth Guard checks if the user is authenticated:
  - If **Authenticated** → Route is activated and the component is displayed.
  - If **Not Authenticated** → Redirect to /login page.
- **User logs in** → AuthService stores token in **localStorage** → Redirect to /dashboard.
- **User clicks logout** → AuthService removes token from **localStorage** → Redirect to /home.

Deliverables:

- **Routing Configuration** for lazy loading and auth guards.
- **Auth Guard** that prevents unauthorized access.
- **Login/Logout logic** to authenticate users and manage sessions.
- **Modular, Lazy-Loaded Application** for performance optimization.

Online Reference

No online Reference

Introduction

the basics

course project-basics

Module 11

What to learn

Understanding forRoot and forChild Concept

Practice Exercise

Practice 1

Understanding forRoot and forChild in Angular Routing

Imagine you are developing a multi-module Angular application for an online bookstore. You have two main modules: **BooksModule** and **UsersModule**. Your task is to set up routing for these modules using the **forRoot** and **forChild** methods. Create a routing configuration where:

- The **AppRoutingModule** uses **forRoot** to define the main routes for the application.
- The **BooksModule** uses **forChild** to define routes specific to book-related components.
- The **UsersModule** also uses **forChild** for user-related components.

Ensure that the main application can navigate to the books and users sections seamlessly.

Practice 2

Creating a Shared Module with forRoot

In your online bookstore application, you want to create a **SharedModule** that contains common components, directives, and pipes that can be used across different modules. Your task is to implement this module using the **forRoot** method. The **SharedModule** should:

- Export a common component, such as **HeaderComponent**, which will be used in both the **BooksModule** and **UsersModule**.
- Provide a service, **AuthService**, that manages user authentication and should be a singleton across the application.

Make sure to demonstrate how to import this **SharedModule** in the main application module and in the feature modules.

Practice 3

Lazy Loading with forChild

As part of optimizing your online bookstore application, you decide to implement lazy loading for the **BooksModule**. Your task is to configure the routing such that:

- The **BooksModule** is loaded only when the user navigates to the '/books' route.
- Use the **forChild** method in the **BooksRoutingModule** to define the routes for book-related components.
- Ensure that the main application module does not import the **BooksModule** directly, but instead uses the router to load it lazily.

Provide the necessary code snippets to demonstrate this lazy loading configuration.

Practice 4

Guarding Routes with forChild

In your online bookstore application, you want to protect certain routes in the **UsersModule** to ensure that only authenticated users can access them. Your task is to implement route guards using the **forChild** method.
The **UsersRoutingModule** should:

- Define routes for user profile and order history, which should be protected by an **AuthGuard**.
- Use the **forChild** method to set up the routes, ensuring that the guard is applied correctly.

Provide the implementation details for the **AuthGuard** and how it integrates with the routing configuration.

Practice 5

Testing forRoot and forChild Implementations

After implementing the routing for your online bookstore application, you want to ensure that everything works as expected. Your task is to write unit tests for the routing configurations. Specifically, you should:

- Test that the main application routes are correctly defined using **forRoot**.
- Test that the child routes in the **BooksModule** and **UsersModule** are correctly defined using **forChild**.
- Verify that lazy loading works as intended and that guards are applied correctly.

Provide examples of the test cases you would write to validate these routing configurations.

Assignment 1

Understanding forRoot and forChild Concepts in Angular Routing

In this assignment, you will create a multi-module Angular application that demonstrates the use of the `forRoot` and `forChild` methods in routing. The application will consist of a main module and a feature module, each with its own routing configuration.

Functional Requirements

Your application should include the following:

- A main module that uses `forRoot` to define the primary routes.
- A feature module that uses `forChild` to define additional routes specific to that module.
- Navigation between the main module and the feature module through a navigation bar.

UI Control IDs

Define the following UI elements with unique IDs:

- Navigation bar: `nav-bar`
- Link to Feature Module: `link-feature-module`
- Feature Module Header: `feature-header`

Functional Flow

The user should be able to:

- Load the application and see the navigation bar.
- Click on the link to the Feature Module, which should navigate to the feature module's route.
- View the header of the Feature Module displayed on the UI.

Validation Elements

Include the following validation elements:

- Success message upon successful navigation: `success-message` with the text 'Navigation Successful!'
- Error message if navigation fails: `error-message` with the text 'Navigation Failed!'

Business Logic

Implement the following business logic:

- Ensure that the feature module can only be accessed if the user is authenticated. If not authenticated, display the error message.
- Use a service to manage the authentication state.

Initial Data

Bootstrap the application with the following initial data:

- User authentication state (true/false).

Selectors for Test Cases

Ensure that all control IDs are specified for automated test case generation:

- For navigation: `nav-bar`, `link-feature-module`
- For messages: `success-message`, `error-message`

By completing this assignment, you will gain a deeper understanding of how to implement routing in Angular applications using `forRoot` and `forChild` methods.

Online Reference

No online Reference

Module 12

What to learn

Communicating with backend services using HTTP
Setup for server communication

Practice Exercise

Practice 1
Do the hands on from the following url https://angular.io/tutorial/toh-pt6

Assignment Exercise

Assignment 1
With reference to day10 assignment Do the crud operation with Student Admission Form.
Assignment 2
CreateComponent,Edit Component and Delete Component. Create a Web API using .netCore and MS SQL Database, Entity Framework core.
Assignment 3
Create an Angular service to perform **CRUD operations** (Create, Read, Update, Delete) on the **GoRest API** (https://gorest.co.in/). Implement the following:

Read: Fetch and display a list of **users** from the API (GET request).

Implement pagination and only show **10 users per page**.
Display user details such as **name**, **email**, and **status**.

Create: Implement a form to create a new **user** and add it to the list (POST request).

Validate the form to ensure the **name** and **email** are provided, and **email** is in a valid format.
Add a **default status** of "active" unless specified otherwise.

Update: Allow editing of an existing **user's** information (name, email, status) and update it via the API (PUT request).

Before updating, check if the **email** already exists for another user. If it does, show a warning that the email is already in use.

Delete: Provide a way to delete a **user** from the list (DELETE request).

- Implement a confirmation dialog before deleting a user to prevent accidental deletions.
- After successful deletion, refresh the user list.

Business Logic:

Active User Filtering:

- Add a filter to display only **active** users by default.
- Allow toggling between **active** and **inactive** users with a button, updating the list accordingly.

Error Handling:

- Ensure proper error handling for each API request (e.g., if a user tries to create a user with an existing email, show a specific error message).
- Display a success message after successful operations (e.g., user creation, update, or deletion).

Online Reference

No online Reference

Using Pipes to Transform Output

Making Http Requests

Module 13

What to learn

Sending GET/PUT/POST and delete request and subscribe the observable

Practice Exercise

No practice exercise

Assignment Exercise

Assignment 1
With reference to day10 assignment Do the crud operation with Student Admission Form. CreateComponent,Edit Component and Delete Component. Create a Web API using .netCore and MS SQL Database, Entity Framework core.

Online Reference

No online Reference

Handling Forms in Angular Apps

Course Project-Forms

Using Pipes to Transform Output

Making Http Requests