MODULE-4 (THEORY ASSIGNMENT)

>NAVIGATION & ROUTING<

1. **Explain how the Navigator widget works in Flutter.**

➔ The Navigator widget in Flutter manages the navigation stack using a stack-based approach (LIFO - Last In, First Out). It allows moving between screens (routes) using the following methods:

➔ Push a new screen:

```
Navigator.push(context, MaterialPageRoute(builder: (context) =>
SecondScreen()));
```

- Adds a new screen on top of the stack.

➔ Pop the current screen

```
Navigator.pop(context);
```

- Removes the current screen and returns to the previous one.

➔ Replace the current screen:

```
Navigator.pushReplacement(context, MaterialPageRoute(builder: (context) =>
NewScreen()));
```

- Replaces the current screen without keeping it in the stack.

➔ Using Named Routes

```
Navigator.pushNamed(context, '/details');
```

- Uses predefined routes for navigation.

2. <u>Describe the concept of named routes and their advantages over direct route navigation.</u>

➔ Named routes in Flutter allow navigation between screens using predefined route names instead of directly creating route objects.

1. Define Routes in MaterialApp:

```
void main() {
  runApp(MaterialApp(
    initialRoute: '/',
    routes: {
      '/': (context) => HomeScreen(),
      '/details': (context) => DetailScreen(),
    },
  ));
}
```

2. Navigate Using Route Name:

```
Navigator.pushNamed(context, '/details');
```

➔ Advantages of Named Routes over Direct Navigation

1. Code Readability & Maintainability:
    - Named routes keep navigation organized, making the code cleaner.
2. Easier Navigation Management:
    - Changing the navigation logic doesn't require modifying every Navigator.push() call.
3. Supports Dynamic Route Handling:
    - You can extract and use route arguments easily.

```
Navigator.pushNamed(context, '/details', arguments: "Hello");
```

4. Ideal for Large Applications:
    - When handling multiple screens, named routes make routing structured and scalable.

3. Explain how data can be passed between screens using route arguments.

➔ In Flutter, data can be passed between screens using route arguments when using named routes.

➔ **Passing Data to a Screen:**

When navigating to a screen, data can be passed using the arguments parameter in Navigator.pushNamed():

```
Navigator.pushNamed(
  context,
  '/details',
  arguments: 'Hello from Home!',
);
```

➔ **Receiving Data in the Destination Screen:**

To access the passed data inside the DetailScreen, use ModalRoute.of(context)?.settings.arguments:

```
class DetailScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final String message = ModalRoute.of(context)!.settings.arguments as
String;

    return Scaffold(
      appBar: AppBar(title: Text("Details")),
      body: Center(child: Text(message)),
    );
  }
}
```

➔ **Defining Named Routes in MaterialApp:**

To ensure the navigation works properly, define routes in MaterialApp:

```
void main() {
  runApp(MaterialApp(
    initialRoute: '/',
    routes: {
      '/': (context) => HomeScreen(),
      '/details': (context) => DetailScreen(),
    },
  ));
}
```

➔ **Benefits of Using Route Arguments:**

- Decouples navigation logic from UI widgets.
- Allows dynamic data transfer between screens.
- Improves maintainability in large applications.

MODULE-6 (THEORY ASSIGNMENT)

>WORKING WITH FORMS AND USER INPUT<

1. **Explain the structure and purpose of forms in Flutter.**

➔ In Flutter, forms are used to collect and validate user input. They are built using the Form widget, which works with TextFormField and a GlobalKey<FormState> to manage validation and state.

➔ Structure of a Form in Flutter

1. Define a GlobalKey<FormState> (to manage form state).
2. Use a Form widget (wraps input fields).
3. Add TextFormField widgets (for user input).
4. Validate input using validator properties (checks correctness).
5. Submit the form using FormState.validate() (ensures valid input).

➔ Example: Simple Login Form

```
import 'package:flutter/material.dart';

class LoginForm extends StatefulWidget {
  @override
  _LoginFormState createState() => _LoginFormState();
}

class _LoginFormState extends State<LoginForm> {
  final _formKey = GlobalKey<FormState>();
  String email = '';

  void _submitForm() {
    if (_formKey.currentState!.validate()) {
      // Form is valid, proceed with action
      print("Email: $email");
    }
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Login")),
      body: Padding(
        padding: EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: Column(
            children: [
              TextFormField(
                decoration: InputDecoration(labelText: "Email"),
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return "Please enter an email";
                  }
                  return null;
                },
                onSaved: (value) => email = value!,
              ),
              SizedBox(height: 20),
              ElevatedButton(
                onPressed: () {
                  if (_formKey.currentState!.validate()) {
                    _formKey.currentState!.save();
                    _submitForm();
                  }
                },
                child: Text("Submit"),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

➔ Purpose of Forms in Flutter:

- Collect User Input – Forms help gather data like login credentials, contact details, etc.
- Validate Input – Prevents invalid data (e.g., empty fields, incorrect email format).
- Manage Form State – Allows saving, resetting, or submitting data efficiently.

➔ Forms are essential for handling structured user input, making them crucial for login screens, registration pages, and other input-driven appS.

2. <u>Describe how controllers and listeners are used to manage form input.</u>

➔ In Flutter, controllers and listeners are used to manage form input dynamically, enabling real-time tracking and updates of user input.

- <u>What is a TextEditingController?</u>

  ➢ A TextEditingController allows you to read, modify, and clear text fields programmatically.

- <u>What is a Listener?</u>

  ➢ A listener is a function that executes whenever the text field’s value changes. It helps in responding to user input dynamically.

➔ <u>Example: Using Controllers and Listeners in a Form</u>

```
import 'package:flutter/material.dart';

class InputForm extends StatefulWidget {
  @override
  _InputFormState createState() => _InputFormState();
}

class _InputFormState extends State<InputForm> {
  final TextEditingController _nameController = TextEditingController();

  @override
  void initState() {
    super.initState();
    _nameController.addListener(_printLatestValue);
  }

  void _printLatestValue() {
    print("Current text: ${_nameController.text}");
  }

  @override
  void dispose() {
    _nameController.dispose(); // Free up resources
    super.dispose();
  }

  @override
```

```
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(title: Text("Form Example")),
     body: Padding(
       padding: EdgeInsets.all(16.0),
       child: Column(
         children: [
           TextField(
             controller: _nameController,
             decoration: InputDecoration(labelText: "Enter your name"),
           ),
           SizedBox(height: 20),
           ElevatedButton(
             onPressed: () {
               print("Submitted Name: ${_nameController.text}");
             },
             child: Text("Submit"),
           ),
         ],
       ),
     ),
   );
 }
}
```

➔ How It Works

- Controller (TextEditingController):
  - Tracks text input.
  - Retrieves text using _nameController.text.
  - Can programmatically set values (_nameController.text = "Hello").

- Listener (addListener()):
  - Calls _printLatestValue() whenever the user types.
  - Useful for validations, live search, and dynamic UI updates.

- dispose() Method:
  - Releases resources when the widget is removed from the tree.

➔ Advantages of Using Controllers & Listeners

- Real-time input tracking (e.g., live search, auto-suggestions).
- Modify text dynamically (e.g., formatting phone numbers).
- Better form validation and control over user input.

Q-3) **List some common form validation techniques and provide examples.**

➔ Form validation ensures that user input is correct before submission. In Flutter, validation is typically handled using the Form and TextFormField widgets along with a validator function.

1. *Required Field Validation*

Ensures that the user does not leave a field empty.

```
TextFormField(
  decoration: InputDecoration(labelText: "Username"),
  validator: (value) {
    if (value == null || value.isEmpty) {
      return "Username is required";
    }
    return null;
  },
)
```

2. *Email Validation*

Checks if the input is in a valid email format using a regular expression.

```
TextFormField(
  decoration: InputDecoration(labelText: "Email"),
  validator: (value) {
    if (value == null || value.isEmpty) {
      return "Please enter an email";
    }
    if (!RegExp(r'^[^@]+@[^@]+\.[^@]+').hasMatch(value)) {
      return "Enter a valid email";
    }
    return null;
  }, )
```

3. *Password Validation (Length & Strength Check)*

Ensures that the password meets minimum security requirements.

```
TextFormField(
  decoration: InputDecoration(labelText: "Password"),
  obscureText: true,
  validator: (value) {
    if (value == null || value.length < 6) {
      return "Password must be at least 6 characters";
    }
    if (!RegExp(r'^(?=.*[A-Z])(?=.*\d)').hasMatch(value)) {
      return "Must contain at least 1 uppercase letter & 1 number";
    }
    return null;
  },
)
```

4. *Confirm Password Validation*

Ensures that the confirmed password matches the original password.

```
final TextEditingController passwordController = TextEditingController();

TextFormField(
  controller: passwordController,
  decoration: InputDecoration(labelText: "Password"),
  obscureText: true,
),

TextFormField(
  decoration: InputDecoration(labelText: "Confirm Password"),
  obscureText: true,
  validator: (value) {
    if (value != passwordController.text) {
      return "Passwords do not match";
    }
    return null;
  },
)
```

5. *Phone Number Validation*

Ensures that the input contains only digits and follows a proper format.

```
TextFormField(
  decoration: InputDecoration(labelText: "Phone Number"),
  keyboardType: TextInputType.phone,
  validator: (value) {
    if (value == null || !RegExp(r'^\d{10}$').hasMatch(value)) {
      return "Enter a valid 10-digit phone number";
    }
    return null;
  },
)
```

6. *Numeric Input Validation*

Ensures the input contains only numbers.

```
TextFormField(
  decoration: InputDecoration(labelText: "Age"),
  keyboardType: TextInputType.number,
  validator: (value) {
    if (value == null || int.tryParse(value) == null) {
      return "Enter a valid number";
    }
    return null;
  },
)
```

7. *Custom Validation for Username (No Special Characters)*

Ensures that usernames contain only letters and numbers.

```
TextFormField(
  decoration: InputDecoration(labelText: "Username"),
  validator: (value) {
    if (value == null || !RegExp(r'^[a-zA-Z0-9]+$').hasMatch(value)) {
      return "Only letters and numbers allowed";
    }
    return null;
  },
)
```

➔ *Implementing Validation in a Form*

```
final _formKey = GlobalKey<FormState>();

Form(
  key: _formKey,
  child: Column(
    children: [
      TextFormField(
        decoration: InputDecoration(labelText: "Email"),
        validator: (value) {
          if (value == null || value.isEmpty) {
            return "Email is required";
          }
          return null;
        },
      ),
      SizedBox(height: 20),
      ElevatedButton(
        onPressed: () {
          if (_formKey.currentState!.validate()) {
            // Form is valid, proceed with submission
            print("Form submitted successfully!");
          }
        },
        child: Text("Submit"),
      ),
    ],
  ),
)
```