Module 6: Working with Forms and User Input

Theory Assignments

Name: Chandvaiya Abhay

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

Structure of a Form in Flutter

A typical Flutter form consists of these main parts:

A. Form Widget

The Form widget acts as a **container** for grouping and managing multiple form fields.

It:

- Holds form fields (like text inputs)
- Manages validation
- Tracks form state
- Allows resetting and saving data

It usually requires a **GlobalKey<FormState>** to control and access the form's state.

B. GlobalKey<FormState>

This key allows you to:

- Validate the form (validate())
- Save the form (save())
- Reset the form (reset())

Example:

final _formKey = GlobalKey<FormState>();

C. Form Field Widgets

Common input widgets used inside a Form:

- TextFormField (most common)
- DropdownButtonFormField
- CheckboxListTile
- RadioListTile

These are special because they integrate directly with the Form for validation and state management.

D. Validation Logic

Each TextFormField can include a validator function:

```
validator: (value) {
  if (value == null || value.isEmpty) {
    return 'Please enter your email';
  }
  return null;
}
```

When you call:

```
_formKey.currentState!.validate();
```

All validators inside the form run automatically.

E. Submit Button

A button triggers validation:

```
ElevatedButton(
  onPressed: () {
    if (_formKey.currentState!.validate()) {
```

```
      // Process data
    }
  },
  child: Text('Submit'),
)
```

Purpose of Forms in Flutter

1. Group Related Input Fields

Forms logically group multiple input fields together.

Example:

- Email
- Password
- Confirm Password

2. Validate User Input

Forms allow:

- Required field checks
- Email format validation
- Password strength checks
- Custom validation logic

3. Manage Form State

The FormState allows:

- Saving field values
- Resetting fields
- Tracking validation status

4. Improve User Experience

Forms:

- Show inline error messages
- Prevent invalid submissions
- Guide users toward correct input

2. Describe how controllers and listeners are used to manage form input.

TextEditingController

The most commonly used controller for forms is:

- TextEditingController

It is attached to a TextField or TextFormField to control and access the text being entered.

Purpose of a Controller

A TextEditingController allows you to:

- Read the current input value
- Set or update text programmatically
- Clear the input field
- Listen to changes
- Control cursor position and selection

Why Controllers Are Important

Without a controller:

- You can only access values during form validation (onSaved).

With a controller:

- You can access values anytime.
- You can modify text dynamically.
- You can react instantly to changes.

Best Practices

Always dispose controllers in dispose()
Use StatefulWidget when using controllers
Avoid unnecessary listeners
Keep validation separate from business logic

In Simple Terms

- **Controller** = Remote control for your input field
- **Listener** = Event detector that reacts when user types

3. List some common form validation techniques and provide examples.

1 Required Field Validation (Empty Check)

Purpose:

Ensure the user does not leave a field blank.

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

2 Email Format Validation (Regex)

Purpose:

Ensure the email follows correct structure.

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

    final emailRegex =
        RegExp(r'^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$');

    if (!emailRegex.hasMatch(value)) {
```

```
      return 'Enter a valid email address';
    }

    return null;
  },
)
```

3 Minimum / Maximum Length Validation

Purpose:

Ensure input length meets requirements.

```
TextFormField(
  decoration: InputDecoration(labelText: 'Password'),
  obscureText: true,
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'Password is required';
    }
    if (value.length < 8) {
      return 'Password must be at least 8 characters';
    }
    return null;
  },
)
```

4 Numeric Validation

Purpose:

Ensure only numbers are entered.

```
TextFormField(
  decoration: InputDecoration(labelText: 'Age'),
  keyboardType: TextInputType.number,
```

```
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'Age is required';
    }
    if (int.tryParse(value) == null) {
      return 'Enter a valid number';
    }
    return null;
  },
)
```

5 Range Validation

Purpose:

Ensure a number falls within a specific range.

```
validator: (value) {
  if (value == null || value.isEmpty) {
    return 'Enter age';
  }

  final age = int.tryParse(value);
  if (age == null) {
    return 'Invalid number';
  }

  if (age < 18 || age > 60) {
    return 'Age must be between 18 and 60';
  }

  return null;
}
```

6 Confirm Password Validation (Cross-Field Validation)

Purpose:

Ensure two fields match.

```
final _passwordController = TextEditingController();

TextFormField(
  controller: _passwordController,
  decoration: InputDecoration(labelText: 'Password'),
)

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

7 Pattern Validation (Custom Regex)

Purpose:

Validate specific formats like phone numbers.

```
validator: (value) {
  final phoneRegex = RegExp(r'^\d{10}$');

  if (!phoneRegex.hasMatch(value!)) {
    return 'Enter a valid 10-digit phone number';
  }
  return null;
}
```

Used for:

- Phone numbers
- Postal codes
- IDs

8 Real-Time Validation (Auto-Validation)

Instead of validating only on submit:

```
Form(
  autovalidateMode: AutovalidateMode.onUserInteraction,
  child: TextFormField(
    validator: (value) {
      if (value == null || value.isEmpty) {
        return 'Field required';
      }
      return null;
    },
  ),
)
```