# Flutter Development Assessment

## Questions and Answer Key

### Core Flutter Concepts

#### 1. What is Scaffold in Flutter and what are its two main components?

**Answer:** Scaffold is a fundamental Flutter widget that implements the basic material design visual layout structure. It provides a framework upon which to build an app's UI elements.

Two main components of Scaffold are:

- AppBar: Displays information and actions at the top of the screen
- Body: The primary content area that contains most of the UI elements

---

#### 2. Differentiate between StatelessWidget and StatefulWidget in Flutter.

**Answer:**

- **StatelessWidget**: An immutable widget that cannot change its state during the lifetime of the widget. It is rebuilt entirely when its configuration changes. Suitable for UI components that don't need to maintain state.
- **StatefulWidget**: A widget that has mutable state that can change during the lifetime of the widget. It consists of two classes: the widget itself and a separate State object that contains the mutable state and build logic. Used when the UI can change dynamically.

---

#### 3. What is the initState() function and when is it called?

**Answer:** The `initState()` function is a lifecycle method in a StatefulWidget's State class. It is called exactly once when the State object is inserted into the widget tree. It's the ideal place to:

- Perform one-time initialization
- Set up subscriptions to streams, controllers, or other data sources
- Initialize variables that depend on the widget's context

It is always called before the first build() method call and cannot be asynchronous.

---

#### 4. Explain the difference between Colors and Color() in Flutter.

**Answer:**

- **Colors**: A utility class that provides predefined color constants (e.g., `Colors.blue`, `Colors.red`). These are static properties for commonly used material design colors.

- **Color()**: A constructor that creates a custom color instance by specifying ARGB (Alpha, Red, Green, Blue) values. Example: `Color(0xFF42A5F5)` where 0xFF represents full opacity followed by the hex color code.

---

**5. What is the difference between main axis and cross axis in Column and Row widgets?**

**Answer:**

- **In Column widget**:
  - Main axis: Vertical (top to bottom)
  - Cross axis: Horizontal (left to right)
- **In Row widget**:
  - Main axis: Horizontal (left to right)
  - Cross axis: Vertical (top to bottom)

MainAxisAlignment controls alignment along the main axis, while CrossAxisAlignment controls alignment along the cross axis.

---

**6. Differentiate between final, const, and late in Dart/Flutter.**

**Answer:**

- **final**: A variable that can only be assigned once. The value can be determined at runtime.

```dart
final String name = getName(); // Value determined at runtime
```

- **const**: A compile-time constant. The value must be known at compile time and cannot change.

```dart
const double pi = 3.14159; // Value known at compile time
```

- **late**: Introduced in Dart 2.12, indicates that a non-nullable variable will be initialized later, but before it's used. Useful for:
  - Lazy initialization
  - Variables initialized in methods like initState()

```dart
late String data; // Will be initialized before first use
```

---

**7. What is a nullable variable in Flutter and how do you declare it?**

**Answer:** A nullable variable is a variable that can contain either a value of its type or null. In Dart's null safety (introduced in Dart 2.12), variables are non-nullable by default.

To declare a nullable variable, add a question mark (?) after the type:

```dart
String? nullableName; // Can be String or null
int? age; // Can be int or null
```

This explicit declaration helps prevent null reference exceptions by requiring null checks before using nullable variables.

---

**8. What is a map in Flutter/Dart and how is it commonly used?**

**Answer:** A map in Dart is a collection of key-value pairs where each key must be unique. It's similar to dictionaries or hash maps in other languages.

Declaration and usage:

```dart
Map<String, int> ages = {
  'Alice': 30,
  'Bob': 25,
  'Charlie': 35,
};

// Accessing values
print(ages['Alice']); // 30

// Adding entries
ages['David'] = 40;

// Checking if a key exists
if (ages.containsKey('Bob')) {
  print('Bob exists');
}
```

Common uses in Flutter:

- Storing configuration data
- Managing form input values
- Converting to/from JSON for API communication
- State management

---

**9. What is the difference between NetworkImage and AssetImage?**

**Answer:**

- **NetworkImage**: Loads an image from a URL over the network.

  ```dart
  Image(image: NetworkImage('https://example.com/image.jpg'))
  ```

  - Requires internet permission
  - Images are cached after first load
  - Can specify headers for authentication

- **AssetImage**: Loads an image that is bundled with the application.

  ```dart
  Image(image: AssetImage('assets/images/logo.png'))
  ```

  - Must be declared in pubspec.yaml under assets
  - Always available offline
  - Faster loading as it's bundled with the app

---

**10. What is the importance of the pubspec.yaml file in Flutter?**

**Answer:** The pubspec.yaml file is a crucial configuration file in Flutter projects that:

- Defines project metadata (name, description, version)
- Manages dependencies and their versions
- Declares assets (images, fonts, etc.) to be included in the app
- Configures Flutter-specific settings
- Controls environment settings (Dart/Flutter SDK versions)
- Defines dev dependencies for development tools

Any changes to pubspec.yaml typically require running `flutter pub get` to apply them.

---

## UI Components and Layout

**11. What is the difference between Spacer(flex:) and SizedBox(height:, width:)?**

**Answer:**

- **Spacer(flex:)**: Creates an expandable empty space that fills available space in a Flex container (Row or Column).
  - Takes proportional space based on flex factor

- Only useful inside Row or Column widgets
- Expands to fill available space

```dart
Row(
  children: [
    Text('Left'),
    Spacer(flex: 2), // Takes 2/3 of available space
    Spacer(flex: 1), // Takes 1/3 of available space
    Text('Right'),
  ],
)
```

- **SizedBox(height:, width:)**: Creates a fixed-size empty box.
  - Dimensions are exact and fixed
  - Can be used anywhere, not just in Flex containers
  - Does not expand or contract

```dart
Column(
  children: [
    Text('Top'),
    SizedBox(height: 20), // Exactly 20 logical pixels tall
    Text('Bottom'),
  ],
)
```

---

**12. What is the difference between Divider and VerticalDivider?**

**Answer:**

- **Divider**: Creates a horizontal separator line.
  - Typically used in Column layouts
  - Has height, thickness, and indent properties
  - Extends across the parent's width

```dart
Column(
  children: [
    ListTile(title: Text('Item 1')),
    Divider(thickness: 1.5, color: Colors.grey),
    ListTile(title: Text('Item 2')),
  ],
)
```

- **VerticalDivider**: Creates a vertical separator line.
  - Typically used in Row layouts
  - Has width, thickness, and indent properties
  - Extends across the parent's height

```dart
Row(
  children: [
    Icon(Icons.home),
    VerticalDivider(width: 20, thickness: 1, color: Colors.grey),
    Icon(Icons.settings),
  ],
)
```

---

### 13. When should we use setState(() { ... }) in Flutter?

**Answer:** `setState()` should be used in StatefulWidgets when you need to:

- Update the UI based on changed state variables
- Notify Flutter that the widget's internal state has changed
- Trigger a rebuild of the widget subtree

Example use cases:

- Toggling visibility of UI elements
- Updating counters or progress indicators
- Changing UI based on user interaction
- Refreshing the UI after data loading completes

```dart
class CounterWidget extends StatefulWidget {
  @override
  _CounterWidgetState createState() => _CounterWidgetState();
}

class _CounterWidgetState extends State<CounterWidget> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++; // UI will update to show new counter value
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Count: $_counter'),
        ElevatedButton(
          onPressed: _incrementCounter,
          child: Text('Increment'),
        ),
      ],
    );
  }
}
```

---

### 14. What is the difference between Imperative and Declarative navigation in Flutter?

**Answer:**

- **Imperative Navigation**: Directly manipulates the navigation stack with explicit commands.
  - Uses Navigator.push(), Navigator.pop(), etc.
  - More verbose and procedural
  - More flexible for complex navigation scenarios

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

- **Declarative Navigation**: Describes the current navigation state rather than the transitions.
  - Uses Router, GoRouter, or other navigation packages
  - More concise for simple navigation flows
  - Better for deep linking and web integration

```dart
// Using GoRouter as an example
GoRouter(
  routes: [
    GoRoute(
      path: '/',
      builder: (context, state) => HomeScreen(),
    ),
    GoRoute(
      path: '/details/:id',
      builder: (context, state) => DetailsScreen(id: state.params['id']!),
    ),
  ],
)
```

---

**15. In stack-based navigation, which command do we use to navigate to a new page?**

**Answer:** `Navigator.push()` is used to navigate to a new page in stack-based navigation.

Example:

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

This adds the new page to the top of the navigation stack, allowing users to return to the previous page using the back button.

---

**16. What is the name of the side menu widget in Flutter?**

**Answer:** The side menu widget in Flutter is called `Drawer`.

It's typically used with Scaffold:

dart

```dart
Scaffold(
  appBar: AppBar(title: Text('My App')),
  drawer: Drawer(
    child: ListView(
      children: [
        DrawerHeader(
          child: Text('Menu'),
          decoration: BoxDecoration(color: Colors.blue),
        ),
        ListTile(
          title: Text('Home'),
          onTap: () {
            // Navigate to home page
            Navigator.pop(context); // Close drawer
          },
        ),
        ListTile(
          title: Text('Settings'),
          onTap: () {
            // Navigate to settings page
            Navigator.pop(context); // Close drawer
          },
        ),
      ],
    ),
  ),
  body: Center(child: Text('Main Content')),
)
```

---

**17. In named route navigation, do we use * to represent the initial page? True or false?**

**Answer:** False.

In named route navigation, the initial page is designated using the `initialRoute` parameter in the MaterialApp widget, not with an asterisk (*).

Example:

```dart
MaterialApp(
  initialRoute: '/',  // This is the initial page
  routes: {
    '/': (context) => HomeScreen(),
    '/details': (context) => DetailsScreen(),
    '/settings': (context) => SettingsScreen(),
  },
)
```

---

### 18. What is GestureDetector in Flutter?

**Answer:** GestureDetector is a widget that detects various user gestures on its child widget. It allows you to make any widget respond to user interactions, even if that widget doesn't have built-in interaction capabilities.

Key features:

- Detects taps, double taps, long presses
- Handles dragging, panning, and scaling gestures
- Can detect the start, update, and end of gestures
- Doesn't add any visual effects by itself

Example:

```dart
GestureDetector(
  onTap: () {
    print('Widget tapped!');
  },
  onDoubleTap: () {
    print('Widget double-tapped!');
  },
  onLongPress: () {
    print('Widget long-pressed!');
  },
  child: Container(
    width: 100,
    height: 100,
    color: Colors.blue,
    child: Center(child: Text('Tap me')),
  ),
)
```

**19. What is the difference between Form, TextFormField, and Validator in Flutter?**

**Answer:**

- **Form**: A container widget that groups and validates multiple form fields together.
  - Provides methods like validate(), save(), and reset()
  - Requires a GlobalKey to access these methods
  - Manages the state of form fields as a group
- **TextFormField**: An enhanced TextField that integrates with the Form widget.
  - Provides built-in validation and state management
  - Includes properties for decoration, controllers, and callbacks
  - Maintains its own state (value, focus, error messages)
- **Validator**: Not a widget but a function used with form fields to validate input.
  - Returns an error message string if validation fails, null if valid
  - Used in the validator property of FormField widgets
  - Can implement complex validation logic

Example:

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

Form(
  key: _formKey,
  child: Column(
    children: [
      TextFormField(
        decoration: InputDecoration(labelText: 'Email'),
        validator: (value) {
          if (value == null || value.isEmpty) {
            return 'Please enter your email';
          }
          if (!value.contains('@')) {
            return 'Please enter a valid email';
          }
          return null; // No error
        },
      ),
      ElevatedButton(
        onPressed: () {
          if (_formKey.currentState!.validate()) {
            _formKey.currentState!.save();
            // Process data
          }
        },
        child: Text('Submit'),
      ),
    ],
  ),
)
```

---

### 20. What is the usage of GlobalKey<FormState>?

**Answer:** A `GlobalKey<FormState>` provides a way to access and control a Form widget from anywhere in your code. Its primary uses are:

- **Form validation**: Access the form's validate() method to check if all fields are valid
- **Saving form data**: Call save() to trigger onSaved callbacks on all form fields
- **Resetting the form**: Use reset() to clear all form fields and errors
- **Accessing form state**: Get the current state of the form and its fields

Example:

dart

```dart
class MyFormWidget extends StatefulWidget {
  @override
  _MyFormWidgetState createState() => _MyFormWidgetState();
}

class _MyFormWidgetState extends State<MyFormWidget> {
  // Create a global key that uniquely identifies the Form widget
  final _formKey = GlobalKey<FormState>();
  String _email = '';

  void _submitForm() {
    // Validate all form fields
    if (_formKey.currentState!.validate()) {
      // If all fields are valid, save their values
      _formKey.currentState!.save();

      // Now process the data
      print('Submitted email: $_email');
    }
  }

  @override
  Widget build(BuildContext context) {
    return Form(
      key: _formKey,
      child: Column(
        children: [
          TextFormField(
            decoration: InputDecoration(labelText: 'Email'),
            validator: (value) => value!.contains('@') ? null : 'Invalid email',
            onSaved: (value) => _email = value!,
          ),
          ElevatedButton(
            onPressed: _submitForm,
            child: Text('Submit'),
          ),
          ElevatedButton(
            onPressed: () => _formKey.currentState!.reset(),
            child: Text('Reset'),
          ),
        ],
      ),
    );
  }
}
```

**21. What are Future, async, and await in Flutter/Dart?**

**Answer:**

- **Future**: An object representing a computation that doesn't complete immediately. It represents a potential value or error that will be available at some time in the future.

  ```dart
  Future<String> fetchData() {
    return Future.delayed(Duration(seconds: 2), () => 'Data loaded');
  }
  ```

- **async**: A keyword that marks a function as asynchronous, allowing the use of await inside it. An async function always returns a Future.

  ```dart
  Future<void> loadData() async {
    // Async function body
  }
  ```

- **await**: A keyword that pauses the execution of an async function until the Future completes, then returns the result value. It can only be used inside async functions.

  ```dart
  Future<void> displayData() async {
    String data = await fetchData(); // Waits for fetchData() to complete
    print(data); // Runs after fetchData() completes
  }
  ```

Together, these features enable asynchronous programming in Dart, allowing operations like network requests, file I/O, and database access to run without blocking the UI thread.

---

**22. State two ways to handle exceptions with future functions.**

**Answer:**

1. **Using try-catch with async/await**:

```dart
Future<void> loadData() async {
  try {
    final result = await fetchDataFromServer();
    // Process successful result
  } catch (e) {
    print('Error loading data: $e');
    // Handle the error (show message, retry, etc.)
  } finally {
    // Code that runs regardless of success or failure
    hideLoadingIndicator();
  }
}
```

2. **Using .then(), .catchError(), and .whenComplete() methods**:

```dart
void loadData() {
  fetchDataFromServer()
    .then((result) {
      // Process successful result
    })
    .catchError((error) {
      print('Error loading data: $error');
      // Handle the error
    })
    .whenComplete(() {
      // Code that runs regardless of success or failure
      hideLoadingIndicator();
    });
}
```

Both approaches allow you to:

- Handle errors gracefully
- Present user-friendly error messages
- Implement retry mechanisms
- Ensure cleanup code runs regardless of success or failure

---

**23. What is the difference between Container and Expanded widgets?**

**Answer:**

- **Container**:

- A convenience widget that combines common painting, positioning, and sizing
- Has fixed dimensions unless constrained by its parent
- Can have decoration (color, border, shadow, etc.)
- Does not automatically expand to fill available space

```dart
Container(
  width: 100,
  height: 100,
  padding: EdgeInsets.all(8.0),
  margin: EdgeInsets.symmetric(vertical: 10.0),
  decoration: BoxDecoration(
    color: Colors.blue,
    borderRadius: BorderRadius.circular(8.0),
  ),
  child: Text('Hello'),
)
```

- **Expanded**:
  - A widget that expands a child of a Row, Column, or Flex to fill available space
  - Must be a descendant of a Flex widget (like Row or Column)
  - Takes a flex factor to determine how to divide available space
  - Cannot be used outside of Flex widgets

```dart
Row(
  children: [
    Text('Fixed width'),
    Expanded(
      flex: 2,
      child: Container(color: Colors.red, height: 100),
    ),
    Expanded(
      flex: 1,
      child: Container(color: Colors.blue, height: 100),
    ),
  ],
)
```

The main difference is that Container has a fixed size by default, while Expanded forces its child to fill available space along the main axis of its parent Flex widget.

---

**24. If we have a non-clickable widget (without press, tap, or click events), what wrapper widget would you use to make it clickable?**

**Answer:** There are several options to make a non-clickable widget clickable:

1. **GestureDetector**: The most versatile option that can detect various gestures.

```dart
GestureDetector(
  onTap: () {
    // Handle tap
  },
  child: MyNonClickableWidget(),
)
```

2. **InkWell**: Similar to GestureDetector but adds Material Design ink splash effect.

```dart
InkWell(
  onTap: () {
    // Handle tap
  },
  splashColor: Colors.blue.withOpacity(0.5),
  child: MyNonClickableWidget(),
)
```

3. **TextButton/ElevatedButton/OutlinedButton**: For text-based widgets when you want button styling.

```dart
TextButton(
  onPressed: () {
    // Handle press
  },
  child: Text('Click me'),
)
```

4. **MouseRegion**: For desktop applications to detect hover and other mouse events.

```dart
MouseRegion(
  onEnter: (_) => print('Mouse entered'),
  child: GestureDetector(
    onTap: () => print('Clicked'),
    child: MyNonClickableWidget(),
  ),
)
```

GestureDetector is generally the best choice due to its flexibility and no visual changes to the wrapped widget.

---

**25. What are reusable components in Flutter and when should you use them?**

**Answer:** Reusable components in Flutter are custom widgets that encapsulate UI elements and functionality that can be used multiple times throughout an application.

Characteristics:

- Self-contained with their own styling and behavior
- Accept parameters to customize appearance and functionality
- Follow the DRY (Don't Repeat Yourself) principle
- Improve maintainability and consistency

When to use reusable components:

- When the same UI pattern appears in multiple places
- When complex widget trees need to be simplified
- To maintain consistent styling across the app
- When functionality needs to be packaged together with UI
- To improve code readability and reduce duplication

Example of a reusable component:

dart

```dart
class CustomButton extends StatelessWidget {
  final String text;
  final VoidCallback onPressed;
  final Color color;
  final IconData? icon;

  const CustomButton({
    Key? key,
    required this.text,
    required this.onPressed,
    this.color = Colors.blue,
    this.icon,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      style: ElevatedButton.styleFrom(
        backgroundColor: color,
        padding: EdgeInsets.symmetric(horizontal: 16, vertical: 10),
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8),
        ),
      ),
      child: Row(
        mainAxisSize: MainAxisSize.min,
        children: [
          if (icon != null) ...[
            Icon(icon),
            SizedBox(width: 8),
          ],
          Text(text),
        ],
      ),
    );
  }
}

// Usage:
CustomButton(
  text: 'Submit',
  onPressed: () => submitForm(),
  color: Colors.green,
```

```
  icon: Icons.send,
)
```

---

**26. How can you save application data locally in Flutter? What technologies would you use?**

**Answer:** Several options exist for saving application data locally in Flutter:

1. **Shared Preferences**: For small key-value pairs like settings, flags, and simple user preferences.

```dart
// Save data
final prefs = await SharedPreferences.getInstance();
await prefs.setString('username', 'John');

// Retrieve data
final username = prefs.getString('username') ?? 'Guest';
```

2. **SQLite (via sqflite package)**: For structured data that requires queries, relationships, and transactions.

```dart
// Create and open database
final database = await openDatabase(
  join(await getDatabasesPath(), 'app_database.db'),
  onCreate: (db, version) {
    return db.execute(
      'CREATE TABLE users(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',
    );
  },
  version: 1,
);

// Insert data
await database.insert('users', {'name': 'Alice', 'age': 25});

// Query data
final users = await database.query('users');
```

3. **File storage**: For binary data like images, documents, or large text files.

```dart
// Get application directory
final directory = await getApplicationDocumentsDirectory();
final file = File('${directory.path}/data.txt');

// Write data
await file.writeAsString('Hello, Flutter!');

// Read data
final contents = await file.readAsString();
```

4. **Hive**: A lightweight, pure-Dart key-value database for simple data.

```dart
// Initialize Hive
await Hive.initFlutter();

// Open a box
final box = await Hive.openBox('settings');

// Save data
box.put('isDarkMode', true);

// Retrieve data
final isDarkMode = box.get('isDarkMode', defaultValue: false);
```

5. **Secure Storage**: For sensitive data like passwords or API tokens.

```dart
final storage = FlutterSecureStorage();

// Save data
await storage.write(key: 'api_token', value: 'my_token');

// Retrieve data
final token = await storage.read(key: 'api_token');
```

Choose based on:

- Data complexity (simple values vs structured data)
- Security requirements
- Performance needs
- Size of data
- Query requirements

**27. What is the sqflite package and how is it used for managing app states and form data?**

**Answer:** The sqflite package is a Flutter plugin that provides SQLite database functionality. It allows Flutter applications to store and retrieve structured data using SQL queries.

**Key features:**

- Full SQLite database implementation
- Supports CRUD operations (Create, Read, Update, Delete)
- Handles transactions and batched operations
- Provides migration capabilities
- Thread-safe implementation

**For managing app states:**

- Sqflite provides persistence for app state across app restarts
- Can store complex state objects by serializing them to JSON
- Allows querying and filtering state data
- Supports atomic updates to prevent data corruption

**For form data:**

- Can store form submissions even when offline
- Enables data validation against existing database records
- Provides history of previous form submissions
- Allows complex relationships between form data entities

**Basic implementation example:**

dart

```dart
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

class DatabaseHelper {
  static final DatabaseHelper _instance = DatabaseHelper._internal();
  static Database? _database;

  factory DatabaseHelper() => _instance;

  DatabaseHelper._internal();

  Future<Database> get database async {
    if (_database != null) return _database!;
    _database = await _initDatabase();
    return _database!;
  }

  Future<Database> _initDatabase() async {
    String path = join(await getDatabasesPath(), 'forms_database.db');
    return await openDatabase(
      path,
      version: 1,
      onCreate: _createDb,
    );
  }

  Future<void> _createDb(Database db, int version) async {
    await db.execute('''
      CREATE TABLE forms(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT NOT NULL,
        description TEXT,
        isCompleted INTEGER NOT NULL,
        createdAt TEXT NOT NULL
      )
    ''');
  }

  // Insert form data
  Future<int> insertForm(Map<String, dynamic> form) async {
    Database db = await database;
    return await db.insert('forms', form);
  }

  // Get all forms
  Future<List<Map<String, dynamic>>> getForms() async {
```

```
    Database db = await database;
    return await db.query('forms');
  }

  // Update form
  Future<int> updateForm(Map<String, dynamic> form) async {
    Database db = await database;
    return await db.update(
      'forms',
      form,
      where: 'id = ?',
      whereArgs: [form['id']],
    );
  }

  // Delete form
  Future<int> deleteForm(int id) async {
    Database db = await database;
    return await db.delete(
      'forms',
      where: 'id = ?',
      whereArgs: [id],
    );
  }
}
```

**Usage in a Flutter application:**

dart

```dart
class FormScreen extends StatefulWidget {
  @override
  _FormScreenState createState() => _FormScreenState();
}

class _FormScreenState extends State<FormScreen> {
  final _formKey = GlobalKey<FormState>();
  final titleController = TextEditingController();
  final descController = TextEditingController();
  final dbHelper = DatabaseHelper();

  void _saveForm() async {
    if (_formKey.currentState!.validate()) {
      Map<String, dynamic> formData = {
        'title': titleController.text,
        'description': descController.text,
        'isCompleted': 0,
        'createdAt': DateTime.now().toIso8601String()
      };

      await dbHelper.insertForm(formData);
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Form saved successfully!')),
      );

      titleController.clear();
      descController.clear();
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Form Example')),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: Column(
            children: [
              TextFormField(
                controller: titleController,
                decoration: InputDecoration(labelText: 'Title'),
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Please enter a title';
```

```
                }
                return null;
              },
            ),
            TextFormField(
              controller: descController,
              decoration: InputDecoration(labelText: 'Description'),
              maxLines: 3,
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: _saveForm,
              child: Text('Save'),
            ),
          ],
        ),
      ),
    ),
  );
 }
}
```