Flutter Mobile Application Development

Weeks 9-15 Complete Resource Guide

Week 9: State Management & Widget Lifecycle

Topics Covered

- Understanding Widget Lifecycle
- Stateless vs Stateful Widgets
- Widget build process
- Lifecycle methods (initState, dispose, didUpdateWidget)
- setState and rebuilding
- Performance considerations

Key Concepts

- **initState()**: Initialize state, subscribe to streams
- **build()**: Create widget tree
- **didUpdateWidget()**: React to widget parameter changes
- **dispose()**: Cleanup resources, unsubscribe streams
- **setState()**: Trigger rebuilds
- **deactivate()**: Widget removed from tree

Code Examples

```
class MyWidget extends StatefulWidget {
  @override
  State<MyWidget> createState() => _MyWidgetState();
}

class _MyWidgetState extends State<MyWidget> {
  @override
  void initState() {
    super.initState();
```

```
  // Initialize
 }

 @override
 void dispose() {
  super.dispose();
  // Cleanup
 }

 @override
 Widget build(BuildContext context) {
  return Container();
 }
}
```

Learning Outcomes

- Explain widget lifecycle phases
- Implement proper initialization and cleanup
- Use setState effectively
- Prevent memory leaks

Week 10: Advanced State Management (Provider)

Topics Covered

- Provider package overview
- ChangeNotifier pattern
- MultiProvider configuration
- Consumer widget
- Selector for optimization
- Business logic separation (BLoC alternative)

Why Provider?

- Simpler than BLoC for most apps
- Reactive programming
- Automatic rebuilds
- Excellent for medium to large apps
- Well-documented and maintained

Architecture Pattern

data/

├── models/

├── providers/

└── repositories/

ui/

├── screens/

├── widgets/

└── providers/

Implementation Steps

1. Add provider package to pubspec.yaml
2. Create ChangeNotifier classes
3. Wrap app with MultiProvider
4. Use Consumer/Selector in widgets
5. Update state through provider methods

Code Example

```
class CartProvider extends ChangeNotifier {
  List<Item> _items = [];

  List<Item> get items => _items;
```

```
  void addItem(Item item) {
    _items.add(item);
    notifyListeners();
  }
}

// In main.dart
MultiProvider(
  providers: [
    ChangeNotifierProvider(create: (_) => CartProvider()),
  ],
  child: MyApp(),
)

// In widget
Consumer<CartProvider>(
  builder: (context, cart, child) {
    return Text(cart.items.length.toString());
  },
)
```

Best Practices

- One ChangeNotifier per concern
- Use Selector for performance
- Avoid rebuilding entire tree
- Keep business logic separate from UI

Week 11: Local Storage - Shared Preferences

Topics Covered

- Shared Preferences basics
- Storing simple data types
- Key-value pair system
- Retrieving data
- Updating and deleting data
- Best practices and limitations

What is Shared Preferences?

- Lightweight key-value storage
- Best for: user preferences, settings, small data
- Not for: large data, complex objects, sensitive data
- Platform: iOS (NSUserDefaults), Android (SharedPreferences)

Implementation Steps

1. Add shared_preferences package
2. Create instance
3. Store data: await prefs.setString('key', 'value')
4. Retrieve: prefs.getString('key')
5. Delete: await prefs.remove('key')

Code Example

```
import 'package:shared_preferences/shared_preferences.dart';


class UserPreferences {
 static const String _userNameKey = 'userName';
 static const String _themeKey = 'isDarkMode';


 static Future<void> saveUserName(String name) async {
  final prefs = await SharedPreferences.getInstance();
  await prefs.setString(_userNameKey, name);
 }
```

```
  static Future<String?> getUserName() async {
    final prefs = await SharedPreferences.getInstance();
    return prefs.getString(_userNameKey);
  }

  static Future<void> setDarkMode(bool isDark) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setBool(_themeKey, isDark);
  }
}
```

Data Types Supported

- String
- int
- double
- bool
- List<String>

Limitations

- Small data only (< 1MB)
- Not suitable for complex objects
- Single process access only
- No encryption by default

Security Considerations

- Never store passwords directly
- Use flutter_secure_storage for sensitive data
- Encrypt sensitive information
- Clear data on logout

Week 12: SQLite Database & File Handling

Topics Covered

- SQLite basics and setup
- Database operations (CRUD)
- Table creation and migrations
- File reading and writing
- File system navigation
- Path providers

Why SQLite?

- Local relational database
- Perfect for offline-first apps
- Structured data
- Query capabilities
- No network required

Setup Steps

dependencies:

sqflite: ^2.0.0

path_provider: ^2.0.0

Database Implementation

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


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


 factory DatabaseHelper() {
  return _instance;
 }
```

```
DatabaseHelper._internal();

static Database? _database;

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

Future<Database> _initDatabase() async {
 final documentsDirectory = await getApplicationDocumentsDirectory();
 final path = '${documentsDirectory.path}/app.db';

 return await openDatabase(
  path,
  version: 1,
  onCreate: _onCreate,
 );
}

Future<void> _onCreate(Database db, int version) async {
 await db.execute('''
  CREATE TABLE users(
   id INTEGER PRIMARY KEY,
   name TEXT,
   email TEXT
```

```
    )
  ''');
 }

 Future<int> insertUser(User user) async {
  final db = await database;
  return await db.insert('users', user.toMap());
 }

 Future<List<User>> getUsers() async {
  final db = await database;
  final result = await db.query('users');
  return result.map((json) => User.fromMap(json)).toList();
 }

 Future<int> updateUser(User user) async {
  final db = await database;
  return await db.update('users', user.toMap(),
     where: 'id = ?', whereArgs: [user.id]);
 }

 Future<int> deleteUser(int id) async {
  final db = await database;
  return await db.delete('users', where: 'id = ?', whereArgs: [id]);
 }
}
```

CRUD Operations

- **Create**: insert()

- **Read**: query(), rawQuery()
- **Update**: update()
- **Delete**: delete()

File Handling

```
import 'dart:io';
import 'package:path_provider/path_provider.dart';

Future<String> get _localPath async {
 final directory = await getApplicationDocumentsDirectory();
 return directory.path;
}

Future<File> get _localFile async {
 final path = await _localPath;
 return File('$path/counter.txt');
}

Future<File> writeFile(String contents) async {
 final file = await _localFile;
 return file.writeAsString(contents);
}

Future<String> readFile() async {
 try {
  final file = await _localFile;
  return await file.readAsString();
 } catch (e) {
  return '';
```

```
 }
}
```

Directory Types

- getApplicationDocumentsDirectory()
- getTemporaryDirectory()
- getApplicationSupportDirectory()
- getExternalStorageDirectory()

Week 13: Networking - REST API & HTTP

Topics Covered

- HTTP protocol basics
- GET, POST, PUT, DELETE requests
- Headers and authentication
- Request/Response handling
- Error handling
- Timeout and retry logic

HTTP Package Setup

```
dependencies:
 http: ^1.1.0
 dio: ^5.0.0  # Alternative with more features
```

Basic GET Request

```
import 'package:http/http.dart' as http;

Future<List<Post>> fetchPosts() async {
 try {
  final response = await http.get(
   Uri.parse('https://jsonplaceholder.typicode.com/posts'),
   headers: {
```

```
      'Content-Type': 'application/json',
    },
  ).timeout(Duration(seconds: 10));

  if (response.statusCode == 200) {
    final List<dynamic> jsonData = json.decode(response.body);
    return jsonData.map((p) => Post.fromJson(p)).toList();
  } else {
    throw Exception('Failed to load posts');
  }
 } catch (e) {
  throw Exception('Error: $e');
 }
}
```

POST Request with Authentication

```
Future<User> createUser(User user) async {
 final response = await http.post(
  Uri.parse('https://api.example.com/users'),
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer $token',
  },
  body: json.encode(user.toJson()),
 );

 if (response.statusCode == 201) {
  return User.fromJson(json.decode(response.body));
 } else {
```

```
    throw Exception('Failed to create user');
  }
}
```

Error Handling

```
try {
  final response = await http.get(Uri.parse(url))
    .timeout(Duration(seconds: 10));

  if (response.statusCode == 200) {
    // Success
  } else if (response.statusCode == 401) {
    // Unauthorized
  } else if (response.statusCode == 404) {
    // Not found
  } else {
    // Other errors
  }
} on TimeoutException {
  print('Request timeout');
} on SocketException {
  print('Network error');
} catch (e) {
  print('Error: $e');
}
```

Using Dio (Recommended for Complex APIs)

```
import 'package:dio/dio.dart';

class ApiClient {
```

```
 late Dio _dio;

 ApiClient() {
  _dio = Dio(BaseOptions(
   baseUrl: 'https://api.example.com',
   connectTimeout: Duration(seconds: 10),
   receiveTimeout: Duration(seconds: 10),
  ));
 }

 Future<List<Post>> getPosts() async {
  try {
   final response = await _dio.get('/posts');
   return (response.data as List)
    .map((p) => Post.fromJson(p))
    .toList();
  } catch (e) {
   rethrow;
  }
 }
}
```

Best Practices

- Always use error handling
- Set appropriate timeouts
- Use proper HTTP methods
- Add authentication headers
- Validate responses
- Implement retry logic

- Use SSL pinning for security
- Mock APIs for testing

Week 14: JSON Parsing & API Integration

Topics Covered

- JSON structure and syntax
- Parsing JSON responses
- Model classes with fromJson/toJson
- JSON serialization libraries
- Handling complex nested JSON
- Type safety

JSON Basics

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com",
  "posts": [
    {
      "id": 1,
      "title": "First Post"
    }
  ]
}
```

Manual JSON Parsing

```
class User {
  final int id;
  final String name;
  final String email;
```

```
  User({
    required this.id,
    required this.name,
    required this.email,
  });

  factory User.fromJson(Map<String, dynamic> json) {
    return User(
      id: json['id'],
      name: json['name'],
      email: json['email'],
    );
  }

  Map<String, dynamic> toJson() {
    return {
      'id': id,
      'name': name,
      'email': email,
    };
  }
}
```

Using json_serializable (Recommended)

```
dependencies:
  json_annotation: ^4.0.0
dev_dependencies:
  json_serializable: ^6.0.0
```

```
  build_runner: ^2.0.0
import 'package:json_annotation/json_annotation.dart';

part 'user.g.dart';

@JsonSerializable()
class User {
  final int id;
  final String name;
  final String email;

  User({
    required this.id,
    required this.name,
    required this.email,
  });

  factory User.fromJson(Map<String, dynamic> json) =>
    _$UserFromJson(json);

  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

Handling Nested JSON

```
@JsonSerializable()
class Post {
  final int id;
  final String title;
  final User author;
```

```
  final List<Comment> comments;

  Post({
    required this.id,
    required this.title,
    required this.author,
    required this.comments,
  });

  factory Post.fromJson(Map<String, dynamic> json) =>
      _$PostFromJson(json);

  Map<String, dynamic> toJson() => _$PostToJson(this);
}
```

Complete API Integration Example

```
class PostRepository {
  final ApiClient _apiClient;

  PostRepository(this._apiClient);

  Future<List<Post>> getAllPosts() async {
    try {
      final response = await _apiClient.get('/posts');
      return (response.data as List)
          .map((json) => Post.fromJson(json))
          .toList();
    } catch (e) {
      throw Exception('Failed to fetch posts: $e');
```

```
  }
 }

 Future<Post> getPost(int id) async {
  try {
   final response = await _apiClient.get('/posts/$id');
   return Post.fromJson(response.data);
  } catch (e) {
   throw Exception('Failed to fetch post: $e');
  }
 }
}
```

Running json_serializable

```
flutter pub run build_runner build
# Or watch for changes
flutter pub run build_runner watch
```

Week 15: Theming, Responsiveness & Deployment

Topics Covered

- Material Design principles
- Creating custom themes
- Dark mode implementation
- Responsive design strategies
- Layout adaptation
- Building and deploying apps

Material Design Theming

```
class MyApp extends StatelessWidget {
 @override
```

```
 Widget build(BuildContext context) {
  return MaterialApp(
   theme: ThemeData(
    useMaterial3: true,
    colorScheme: ColorScheme.fromSeed(
     seedColor: Colors.blue,
    ),
    typography: Typography.material2021(
     platform: defaultTargetPlatform,
    ),
   ),
   darkTheme: ThemeData(
    useMaterial3: true,
    brightness: Brightness.dark,
    colorScheme: ColorScheme.fromSeed(
     seedColor: Colors.blue,
     brightness: Brightness.dark,
    ),
   ),
   themeMode: ThemeMode.system,
   home: HomeScreen(),
  );
 }
}
```

Custom Theme Provider

```
class ThemeProvider extends ChangeNotifier {
 ThemeMode _themeMode = ThemeMode.light;
```

```
 ThemeMode get themeMode => _themeMode;

 void toggleTheme() {
  _themeMode = _themeMode == ThemeMode.light
    ? ThemeMode.dark
    : ThemeMode.light;
  notifyListeners();
 }

 void setTheme(ThemeMode mode) {
  _themeMode = mode;
  notifyListeners();
 }
}
```

Responsive Design

```
class ResponsiveLayout extends StatelessWidget {
 final Widget mobileBody;
 final Widget tabletBody;
 final Widget desktopBody;

 ResponsiveLayout({
  required this.mobileBody,
  required this.tabletBody,
  required this.desktopBody,
 });

 @override
 Widget build(BuildContext context) {
```

```
  return LayoutBuilder(
   builder: (context, constraints) {
    if (constraints.maxWidth < 600) {
     return mobileBody;
    } else if (constraints.maxWidth < 1200) {
     return tabletBody;
    } else {
     return desktopBody;
    }
   },
  );
 }
}
```

Media Query for Responsive Design

```
class AdaptiveScreen extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  final screenWidth = MediaQuery.of(context).size.width;
  final isMobile = screenWidth < 600;

  return Scaffold(
   body: SingleChildScrollView(
    child: Column(
     children: [
      if (isMobile)
       MobileLayout()
      else
       DesktopLayout(),
```

```
        ],
      ),
    ),
  );
 }
}
```

Building for Release

iOS

```
flutter build ios --release

# Build and upload to App Store

flutter build ios --release -t lib/main.dart
```

Android

```
# Generate keystore

keytool -genkey -v -keystore ~/key.jks -keyalg RSA -keysize 2048 -validity 10000 -alias key


# Sign APK

flutter build apk --release


# Build App Bundle for Play Store

flutter build appbundle --release
```

Deployment Checklist

- [ ] Test on multiple devices
- [ ] Check performance and battery usage
- [ ] Remove debug code and print statements
- [ ] Update version number
- [ ] Create release notes
- [ ] Test in release mode: flutter run --release

- [ ] Optimize assets and images
- [ ] Implement proper error handling
- [ ] Add app icon and splash screen
- [ ] Configure app signing
- [ ] Review privacy policy

App Icon and Splash Screen

```
dev_dependencies:
  flutter_launcher_icons: ^0.13.0
  flutter_native_splash: ^2.0.0

flutter_icons:
  android: "launcher_icon"
  ios: true
  image_path: "assets/icon/icon.png"

flutter_native_splash:
  color: "#FFFFFF"
  image: assets/splash.png
```

Performance Optimization

- Use const constructors
- Avoid rebuilds with Selector
- Lazy load images: Image.network(..., loadingBuilder: ...)
- Use RepaintBoundary for complex widgets
- Profile with DevTools
- Monitor memory usage

Final Project Requirements

1. Complete working application
2. At least 5+ screens

3. Network API integration
4. Local data storage (SQLite or SharedPreferences)
5. State management implementation
6. Error handling
7. Responsive design
8. Proper app icon and splash screen
9. Documentation
10. Code quality and organization

Summary: Key Topics by Week

Week	Focus Area	Key Skills
9	Widget Lifecycle	Memory management, proper cleanup
10	Advanced State Management	Provider pattern, scalability
11	Local Storage	SharedPreferences, data persistence
12	Database & Files	SQLite operations, file I/O
13	Networking	HTTP requests, API communication
14	JSON Parsing	Data serialization, API integration
15	Theming & Deployment	UI polish, production readiness

Resources for Further Learning

Official Documentation

- Flutter.dev - Complete documentation
- Dart.dev - Language reference
- pub.dev - Package registry

Popular Packages

- provider - State management
- dio - HTTP client

- sqflite - SQLite database
- shared_preferences - Key-value storage
- json_serializable - JSON serialization
- flutter_secure_storage - Secure storage

Best Practices

- Follow Material Design guidelines
- Write unit and widget tests
- Use meaningful variable names
- Document your code
- Implement proper error handling
- Profile and optimize performance

Assessment & Evaluation

Continuous Assessment (40%)

- Weekly assignments (10%)
- Quizzes (10%)
- Lab participation (10%)
- Code quality (10%)

Mid-Term Exam (20%)

- Week 9-11 content
- Practical coding tasks

Final Project (20%)

- Complete working app
- Code documentation
- Presentation

Final Exam (20%)

- Week 12-15 content
- Comprehensive practical exam

Course Completion: Students will have built a complete, production-ready Flutter application demonstrating mastery of all course concepts.