Lección 6: Mejoras Avanzadas y Publicación de la Aplicación

Objetivos de la Lección

- **Refactorizar** el código para mejorar la organización y mantenimiento utilizando patrones de diseño y gestión de estado.
- **Implementar** una solución de gestión de estado, como Provider, para manejar el estado de la aplicación de manera eficiente.
- **Añadir** animaciones y transiciones para mejorar la experiencia de usuario.
- **Realizar pruebas** unitarias y de integración para asegurar la calidad de la aplicación.
- **Preparar** la aplicación para su despliegue y publicación en las tiendas de aplicaciones.

Introducción a la Lección

En las lecciones anteriores, hemos construido una aplicación funcional que simula una red social básica. Sin embargo, a medida que las aplicaciones crecen en complejidad, es esencial refactorizar y optimizar el código para mejorar su mantenibilidad y escalabilidad. En esta lección, nos enfocaremos en implementar una solución de gestión de estado utilizando **Provider**, un paquete popular en Flutter que facilita la gestión y compartición de estado entre widgets.

Además, añadiremos animaciones y transiciones para mejorar la experiencia del usuario y haremos pruebas unitarias y de integración para asegurar que nuestra aplicación funciona correctamente. Finalmente, prepararemos nuestra aplicación para su despliegue, lo que incluye la configuración necesaria para publicar en las tiendas de aplicaciones de iOS y Android.

Desarrollo de Conceptos

Gestión de Estado en Flutter

La gestión de estado es un concepto clave en el desarrollo de aplicaciones. En Flutter, existen varias soluciones para manejar el estado de la aplicación de manera eficiente, como **Provider**, **Bloc**, **Riverpod**, entre otros. Estas soluciones ayudan a mantener el código organizado y a facilitar la comunicación entre diferentes partes de la aplicación.

Provider

Provider es un paquete oficial de Flutter que se basa en el patrón InheritedWidget y ofrece una forma sencilla y efectiva de manejar el estado y la dependencia de objetos a lo largo de la aplicación.

Animaciones y Transiciones

Las animaciones mejoran la experiencia del usuario al hacer que la aplicación se sienta más fluida y atractiva. Flutter proporciona un conjunto de herramientas y widgets para implementar animaciones de manera sencilla.

Pruebas Unitarias y de Integración

Las pruebas son fundamentales para garantizar la calidad y estabilidad de una aplicación. Las pruebas unitarias verifican el correcto funcionamiento de unidades individuales de código, mientras que las pruebas de integración aseguran que diferentes partes de la aplicación funcionen juntas correctamente.

Preparación para el Despliegue

Antes de publicar una aplicación, es necesario realizar ciertos pasos de configuración, como establecer un identificador único, generar iconos de la aplicación y firmar el código.

Secciones Técnicas Específicas

1. Implementación de Provider para Gestión de Estado

Paso 1: Añadir Provider a las Dependencias

En el archivo `pubspec.yaml`, añade `provider` a las dependencias:

```
dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.4
  shared_preferences: ^2.0.7
  provider: ^6.0.0
```

Ejecuta:

```
flutter pub get
```

Paso 2: Crear un Modelo para el Usuario Autenticado

Crea un archivo **lib/providers/auth_provider.dart**:

```
// lib/providers/auth_provider.dart
import 'package:flutter/material.dart';
import '../models/user.dart';
import 'package:shared_preferences/shared_preferences.dart';
import '../services/api_service.dart';

class AuthProvider with ChangeNotifier {
  User? _user;
  ThemeMode _themeMode = ThemeMode.system;
  ApiService _apiService = ApiService();
```

```
  User? get user => _user;
  ThemeMode get themeMode => _themeMode;

  void loadUserFromPreferences() async {
    SharedPreferences prefs = await
SharedPreferences.getInstance();
    int? userId = prefs.getInt('userId');
    if (userId != null) {
      String? userName = prefs.getString('userName');
      String? userEmail = prefs.getString('userEmail');
      String? userUsername = prefs.getString('userUsername');
      _user = User(
        id: userId,
        name: userName ?? '',
        email: userEmail ?? '',
        username: userUsername ?? '',
      );
      notifyListeners();
    }
  }

  Future<bool> login(String email, String password) async {
    List<User> users = await _apiService.getUsers();
    User? user = users.firstWhere(
      (user) => user.email.toLowerCase() == email.toLowerCase()
&& user.username == password,
      orElse: () => null,
    );
    if (user != null) {
      _user = user;
      SharedPreferences prefs = await
SharedPreferences.getInstance();
      await prefs.setInt('userId', user.id);
      await prefs.setString('userName', user.name);
      await prefs.setString('userEmail', user.email);
```

```
      await prefs.setString('userUsername', user.username);
      notifyListeners();
      return true;
    }
    return false;
  }

  void logout() async {
    _user = null;
    SharedPreferences prefs = await
SharedPreferences.getInstance();
    await prefs.remove('userId');
    await prefs.remove('userName');
    await prefs.remove('userEmail');
    await prefs.remove('userUsername');
    notifyListeners();
  }

  void loadThemeFromPreferences() async {
    SharedPreferences prefs = await
SharedPreferences.getInstance();
    int? themeIndex = prefs.getInt('themeMode');
    _themeMode = ThemeMode.values[themeIndex ?? 0];
    notifyListeners();
  }

  void setThemeMode(ThemeMode mode) async {
    _themeMode = mode;
    SharedPreferences prefs = await
SharedPreferences.getInstance();
    await prefs.setInt('themeMode', mode.index);
    notifyListeners();
  }
}
```

Explicación

- **AuthProvider**: Esta clase maneja el estado del usuario autenticado y las preferencias de tema.
- **ChangeNotifier**: Permite notificar a los widgets que escuchan los cambios en el estado.
- **loadUserFromPreferences**: Carga la información del usuario almacenada en `SharedPreferences`.
- **login**: Autentica al usuario y almacena su información.
- **logout**: Cierra la sesión del usuario y elimina su información almacenada.
- **loadThemeFromPreferences** y **setThemeMode**: Manejan la preferencia de tema.

Paso 3: Envolver la Aplicación con Provider

En **lib/main.dart**, envuelve tu aplicación con `ChangeNotifierProvider`.

```
// lib/main.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'providers/auth_provider.dart';
import 'screens/home_screen.dart';
import 'screens/login_screen.dart';

void main() {
  runApp(
    ChangeNotifierProvider(
      create: (_) => AuthProvider(),
      child: MyApp(),
    ),
  );
```

```
}

class MyApp extends StatelessWidget {
  // Este widget es la raíz de la aplicación.
  @override
  Widget build(BuildContext context) {
    final authProvider = Provider.of<AuthProvider>(context);
    authProvider.loadUserFromPreferences();
    authProvider.loadThemeFromPreferences();

    return MaterialApp(
      title: 'Red Social Básica',
      theme: ThemeData(
        brightness: Brightness.light,
        primarySwatch: Colors.blue,
        accentColor: Colors.blueAccent,
      ),
      darkTheme: ThemeData(
        brightness: Brightness.dark,
        primarySwatch: Colors.blue,
        accentColor: Colors.blueAccent,
      ),
      themeMode: authProvider.themeMode,
      home: authProvider.user == null ? LoginScreen() :
HomeScreen(),
    );
  }
```

```
}
```

Paso 4: Modificar las Pantallas para Usar Provider

LoginScreen

```
// lib/screens/login_screen.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '../providers/auth_provider.dart';
import 'home_screen.dart';

class LoginScreen extends StatefulWidget {
  @override
  _LoginScreenState createState() => _LoginScreenState();
}

// ... Código existente

void _login() async {
  if (_formKey.currentState!.validate()) {
    _formKey.currentState!.save();
    setState(() {
      _isLoading = true;
    });
    final authProvider = Provider.of<AuthProvider>(context,
listen: false);
    bool success = await authProvider.login(_email, _password);
```

```
    if (success) {
      Navigator.pushReplacement(
        context,
        MaterialPageRoute(builder: (context) => HomeScreen()),
      );
    } else {
      _showError('Credenciales incorrectas');
    }
    setState(() {
      _isLoading = false;
    });
  }
}
```

HomeScreen

```
// lib/screens/home_screen.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '../providers/auth_provider.dart';
// ... Código existente

class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final authProvider = Provider.of<AuthProvider>(context);
    // ... Código existente

    return Scaffold(
```

```
      appBar: AppBar(
        title: Text('Bienvenido, ${authProvider.user?.name ??
''}'),
        actions: [
          IconButton(
            icon: Icon(Icons.people),
            onPressed: () {
              // Navegar a UsersScreen
            },
          ),
          PopupMenuButton<String>(
            onSelected: (value) {
              if (value == 'theme') {
                _showThemeSelection(context);
              } else if (value == 'logout') {
                authProvider.logout();
                Navigator.pushReplacement(
                  context,
                  MaterialPageRoute(builder: (context) =>
LoginScreen()),
                );
              }
            },
            itemBuilder: (context) => [
              PopupMenuItem(
                value: 'theme',
                child: Text('Cambiar Tema'),
```

```
              ),
              PopupMenuItem(
                value: 'logout',
                child: Text('Cerrar Sesión'),
              ),
            ],
          ),
        ],
      ),
      // ... Código existente
    );
  }

  void _showThemeSelection(BuildContext context) {
    final authProvider = Provider.of<AuthProvider>(context,
listen: false);
    showDialog(
      context: context,
      builder: (context) => AlertDialog(
        title: Text('Seleccionar Tema'),
        content: Column(
          mainAxisSize: MainAxisSize.min,
          children: [
            RadioListTile<ThemeMode>(
              title: Text('Predeterminado'),
              value: ThemeMode.system,
              groupValue: authProvider.themeMode,
```

```
            onChanged: (value) {
              authProvider.setThemeMode(value!);
              Navigator.of(context).pop();
            },
          ),
          RadioListTile<ThemeMode>(
            title: Text('Modo Claro'),
            value: ThemeMode.light,
            groupValue: authProvider.themeMode,
            onChanged: (value) {
              authProvider.setThemeMode(value!);
              Navigator.of(context).pop();
            },
          ),
          RadioListTile<ThemeMode>(
            title: Text('Modo Oscuro'),
            value: ThemeMode.dark,
            groupValue: authProvider.themeMode,
            onChanged: (value) {
              authProvider.setThemeMode(value!);
              Navigator.of(context).pop();
            },
          ),
        ],
      ),
    ),
  );
```

```
  }
}
```

Explicación

- **Uso de Provider**: Ahora obtenemos el `authProvider` desde el contexto y accedemos al usuario y tema directamente.
- **Notificación de Cambios**: Al llamar a `notifyListeners()`, los widgets que escuchan a `AuthProvider` se actualizan automáticamente.

2. Añadir Animaciones y Transiciones

Paso 1: Implementar Animaciones en las Transiciones de Pantallas

Utiliza el paquete **animations** de Flutter:

```
dependencies:
  animations: ^2.0.0
```

Ejecuta:

```
flutter pub get
```

Paso 2: Utilizar `FadeThroughTransition` para Transiciones Suaves

En lugar de `MaterialPageRoute`, utilizamos `PageRouteBuilder` con transiciones animadas.

Ejemplo en **HomeScreen**:

```
Navigator.push(
  context,
  PageRouteBuilder(
```

```
    pageBuilder: (context, animation, secondaryAnimation) =>
UsersScreen(),
    transitionsBuilder: (context, animation, secondaryAnimation,
child) {
      return FadeThroughTransition(
        animation: animation,
        secondaryAnimation: secondaryAnimation,
        child: child,
      );
    },
  ),
);
```

Paso 3: Añadir Animaciones a Widgets

Podemos usar widgets como `AnimatedContainer`, `AnimatedOpacity`, `Hero`, etc.

Ejemplo con `Hero`:

En **HomeScreen**, envolvemos el avatar del usuario:

```
CircleAvatar(
  child: Hero(
    tag: 'avatar-${post.userId}',
    child: Text(post.userId.toString()),
  ),
),
```

En **UserProfileScreen**:

```
Hero(
  tag: 'avatar-${user.id}',
  child: CircleAvatar(
    radius: 50,
    child: Text(user.name.substring(0, 1)),
  ),
),
```

3. Realizar Pruebas Unitarias y de Integración

Paso 1: Escribir Pruebas Unitarias para los Modelos

Crea una carpeta **test/** y añade archivos de prueba.

Ejemplo para `user_test.dart`:

```
// test/user_test.dart
import 'package:flutter_test/flutter_test.dart';
import '../lib/models/user.dart';

void main() {
  test('User model fromJson', () {
    final json = {
      'id': 1,
      'name': 'John Doe',
      'username': 'johndoe',
      'email': 'john@example.com',
```

```
    };
    final user = User.fromJson(json);
    expect(user.id, 1);
    expect(user.name, 'John Doe');
    expect(user.username, 'johndoe');
    expect(user.email, 'john@example.com');
  });
}
```

Paso 2: Escribir Pruebas para Servicios

Ejemplo para `api_service_test.dart`:

```
// test/api_service_test.dart
import 'package:flutter_test/flutter_test.dart';
import '../lib/services/api_service.dart';

void main() {
  final apiService = ApiService();

  test('Fetch users', () async {
    final users = await apiService.getUsers();
    expect(users.length, greaterThan(0));
  });

  // Más pruebas para otros métodos
}
```

Paso 3: Ejecutar las Pruebas

En la terminal, ejecuta:

```
flutter test
```

4. Preparación para el Despliegue

Paso 1: Configurar el Identificador de la Aplicación

En **android/app/build.gradle**, cambia `applicationId`:

```
defaultConfig {
    applicationId "com.tuempresa.redsocial"
    // ... resto del código
}
```

En **ios/Runner.xcodeproj**, cambia el **Bundle Identifier** en Xcode.

Paso 2: Añadir Iconos de la Aplicación

Utiliza el paquete **flutter_launcher_icons**:

```
dev_dependencies:
  flutter_launcher_icons: ^0.9.0

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

Ejecuta:

```
flutter pub run flutter_launcher_icons:main
```

Paso 3: Firmar la Aplicación (Android)

Sigue las instrucciones oficiales de Flutter para firmar la aplicación:

https://flutter.dev/docs/deployment/android

Paso 4: Preparar la Aplicación para Publicar

- **Android**: Genera el APK o App Bundle.
- **iOS**: Configura los certificados y perfiles de aprovisionamiento en Xcode.

5. Mejora de la Experiencia de Usuario

Paso 1: Optimizar el Rendimiento

- **Lazy Loading**: En las listas, utiliza `ListView.builder` y `CachedNetworkImage` para imágenes.
- **Optimización de Imágenes**: Usa imágenes de tamaño adecuado.

Paso 2: Manejar Errores y Estados

- **Estados Vacíos**: Muestra mensajes cuando no hay datos.
- **Errores de Conexión**: Informa al usuario si hay problemas de red.

Paso 3: Accesibilidad

- **Etiquetas Semánticas**: Utiliza `Semantics` para mejorar la accesibilidad.
- **Compatibilidad con VoiceOver y TalkBack**: Asegúrate de que los lectores de pantalla funcionen correctamente.

Ejemplos en Código

Uso de Provider para Gestión de Estado

```
class AuthProvider with ChangeNotifier {
  // ... código existente

  void setThemeMode(ThemeMode mode) async {
    _themeMode = mode;
    SharedPreferences prefs = await
SharedPreferences.getInstance();
    await prefs.setInt('themeMode', mode.index);
    notifyListeners();
  }
}
```

Explicación:

- **ChangeNotifier**: Permite que los widgets escuchen cambios en el estado.
- **notifyListeners()**: Notifica a los widgets para que se actualicen.

Relación con Otros Temas

Esta lección se relaciona con:

- **Patrones de Diseño**: Uso de Provider para mejorar la arquitectura de la aplicación.
- **Optimización y Mantenimiento**: Refactorización del código para mejorar su calidad.
- **Experiencia de Usuario**: Añadir animaciones y optimizar el rendimiento.
- **Calidad de Software**: Implementación de pruebas para asegurar el correcto funcionamiento.

- **Despliegue de Aplicaciones**: Preparación y publicación de la aplicación en las tiendas.

Resumen de la Lección

En esta lección, hemos llevado nuestra aplicación al siguiente nivel al refactorizar el código utilizando Provider para la gestión de estado, mejorando la organización y mantenibilidad. Añadimos animaciones y transiciones para una experiencia de usuario más atractiva. Implementamos pruebas unitarias y de integración para asegurar la calidad del código. Finalmente, preparamos la aplicación para su despliegue, realizando las configuraciones necesarias para publicar en las tiendas de aplicaciones.

Actividad de la Lección

Objetivo de la Actividad:

- Demostrar tu capacidad para refactorizar y mejorar el código de una aplicación Flutter.
- Asegurar que comprendes cómo implementar una gestión de estado eficiente con Provider.
- Mostrar que puedes preparar una aplicación para su despliegue y publicación.
- Consolidar tus conocimientos y habilidades en el desarrollo de aplicaciones móviles con Flutter.

Tarea Práctica:

1. **Implementa Provider** en tu proyecto para manejar el estado de la aplicación.
2. **Refactoriza el código** para utilizar Provider en lugar de pasar funciones y datos manualmente entre widgets.
3. **Añade animaciones y transiciones** en la navegación y en los widgets interactivos.
4. **Escribe pruebas unitarias** para los modelos y servicios de tu aplicación.

5. **Prepara la aplicación para el despliegue**, configurando los identificadores y añadiendo iconos.
6. **Prueba la aplicación** en un dispositivo físico y asegúrate de que todas las funcionalidades funcionan correctamente.
7. **Documenta tu proceso**: Crea un documento en PDF que incluya capturas de pantalla y explica las mejoras realizadas y los retos encontrados.
8. **Entrega**: Sube el PDF y el código fuente actualizado del proyecto en una carpeta comprimida.