INFORME ACADÉMICO / TÉCNICO

1. Datos Generales

Título del Informe:	Tarea 1.2 Personalización de Interfaces
Autor(a):	Mesias Orlando Mariscal Bryan Roberto Quispe Romero Gabriel Murillo
Carrera:	Ingeniería en Software
Asignatura o Proyecto:	Desarrollo de Aplicaciones Móviles
Tutor o Supervisor:	Doris Karina Chicaiza Angamarca
Institución:	Universidad de las Fuerzas Armadas ESPE – Sede Sangolquí
Fecha de entrega:	6 de noviembre de 2025

Índice

1 Introducción

Este informe se centra en el desarrollo de una aplicación móvil con sistema de autenticación y perfiles, implementando el patrón MVC y Atomic Design. El proyecto integra tres perfiles de estudiantes con credenciales de acceso y datos personales, académicos y profesionales.

El desarrollo utilizó Flutter como framework principal con sistema de temas personalizado en color verde. Se implementó MVC con datos quemados, simplificando la lógica sin dependencias externas.

En conclusión, este trabajo demuestra la aplicación práctica de patrones arquitectónicos en aplicaciones móviles reales. La implementación permite un código organizado, mantenible y escalable, estableciendo bases sólidas para futuros desarrollos. Finalmente, el proyecto sirve como referencia para la construcción de sistemas de autenticación simples con gestión de perfiles en Flutter.

1.1 Objetivos

1.1.1 Objetivo General

Desarrollar una aplicación móvil de autenticación y perfiles mediante el patrón MVC y metodología Atomic Design, utilizando Flutter como framework de desarrollo, para demostrar la implementación de arquitecturas organizadas con datos estáticos.

1.1.2 Objetivos Específicos

1. Implementar el patrón MVC separando modelo, controlador y vista para organizar el código de manera estructurada.

2. Crear componentes reutilizables mediante Atomic Design con átomos y moléculas consistentes para estandarizar la interfaz de usuario.

3. Crear perfiles de usuario con información personal, académica y profesional para mostrar datos estructurados al acceder exitosamente.

2 Marco Teórico

2.1 Patrón Modelo-Vista-Controlador (MVC)

El patrón MVC divide una aplicación en tres componentes interconectados. El Modelo representa los datos y la lógica de negocio, siendo independiente de la interfaz de usuario. La Vista es responsable de la presentación visual de los datos al usuario. El Controlador actúa como intermediario, recibiendo las entradas del usuario, validándolas y coordinando las actualizaciones entre Modelo y Vista.

2.2 Atomic Design

Atomic Design es una metodología para crear sistemas de diseño mediante componentes incrementales. Los Átomos son componentes básicos indivisibles como botones y campos de texto. Las Moléculas son grupos de átomos que funcionan juntos. Los Organismos son componentes complejos formados por moléculas y átomos. Las Plantillas organizan organismos en estructuras de página. Las Páginas son instancias específicas con contenido real.

2.3 Flutter

Flutter es un framework de código abierto desarrollado por Google para crear aplicaciones multiplataforma. Utiliza el lenguaje Dart y se caracteriza por su arquitectura reactiva basada en widgets. La combinación de StatefulWidget y StatelessWidget permite crear interfaces dinámicas. El sistema de temas con ThemeData permite personalizar colores, tipografías y estilos globalmente.

2.4 Gestión de Datos Estáticos

Los datos quemados son valores predefinidos en el código fuente, útiles para prototipos y aplicaciones con información fija. Este enfoque simplifica la arquitectura eliminando bases de datos y servicios externos. La validación de credenciales se realiza mediante comparación directa en memoria, siendo eficiente para conjuntos pequeños de usuarios.

3 Desarrollo

3.1 Arquitectura del Proyecto

La estructura del proyecto sigue una organización basada en MVC con la siguiente distribución de carpetas:

- `lib/modelo/`: Contiene las clases de datos
- `lib/controlador/`: Contiene la lógica de validación y coordinación
- `lib/vista/`: Contiene los componentes de interfaz organizados en átomos, moléculas y páginas
- `lib/temas/`: Contiene la configuración de estilos y colores

3.2 Capa Modelo

3.2.1 Archivo: modelo/usuario_modelo.dart

Este archivo define la estructura de usuarios y almacena las credenciales de acceso. Contiene la clase `UsuarioModelo` con los atributos usuario, contraseña y tipoPerfil. Se implementa una lista estática `usuariosValidos` con tres usuarios: mesias/123456, gabriel/123456 y brayan/123456.

```dart
// Valida si las credenciales son correctas
// Retorna el tipo de perfil si es valido, vacio si no
static String validarCredenciales(String usuario, String contrasena) {
  // Primera validacion: verificar que no esten vacios
  // Si alguno esta vacio, retorna cadena vacia indicando fallo
  if (usuario.trim().isEmpty || contrasena.trim().isEmpty) {
    return '';
  }

  // Buscar usuario en la lista de usuarios validos
  // Se utiliza firstWhere para encontrar coincidencia exacta
  try {
    final usuarioEncontrado = usuariosValidos.firstWhere(
      // Comparar usuario (sin distinguir mayusculas) y contrasena exacta
      (u) => u.usuario.toLowerCase() == usuario.toLowerCase().trim()
          && u.contrasena == contrasena,
    );
    // Si se encuentra, retornar el tipo de perfil (mesias, gabriel, brayan)
    return usuarioEncontrado.tipoPerfil;
  } catch (e) {
    // Si no se encuentra (firstWhere lanza excepcion), retornar vacio
    return '';
  }
```

```
}
```

Listing 1: Validación de Credenciales en usuario_modelo.dart

El método implementa dos niveles de validación: primero verifica que los campos no estén vacíos usando `trim().isEmpty`, y luego busca coincidencias en la lista de usuarios válidos. La comparación del nombre de usuario es insensible a mayúsculas mediante `toLowerCase()`, mientras que la contraseña debe coincidir exactamente. Si no se encuentra el usuario, `firstWhere` lanza una excepción que se captura y retorna cadena vacía.

3.2.2 Archivo: modelo/perfil_modelo.dart

Define la estructura de datos del perfil de usuario con los atributos nombre, apellido, carrera, teléfono, edad, email, sobreMi, githubUrl, habilidades, experiencia y educación. Incluye clases auxiliares `Experiencia` con empresa, cargo, periodo y descripción, y `Educacion` con institución, título y periodo. Proporciona el getter `nombreCompleto` que concatena nombre y apellido.

3.3 Capa Controlador

3.3.1 Archivo: controlador/login_controlador.dart

Gestiona la lógica de autenticación del sistema actuando como intermediario entre la vista y el modelo. Implementa el método `intentarLogin` que realiza validación inicial y delega la autenticación al modelo.

```
// Intenta hacer login con las credenciales proporcionadas
// Retorna el tipo de perfil si es exitoso, vacio si falla
String intentarLogin(String usuario, String contrasena) {
  // Validacion inicial: verificar que no esten vacios antes de consultar modelo
  // Esto evita llamadas innecesarias al modelo con datos invalidos
  if (usuario.trim().isEmpty || contrasena.trim().isEmpty) {
    return '';
  }

  // Delegar la autenticacion al modelo UsuarioModelo
  // El modelo se encarga de buscar y validar las credenciales
  return UsuarioModelo.validarCredenciales(usuario, contrasena);
}
```

Listing 2: Validación en login_controlador.dart

Este controlador simplificado implementa el patrón MVC separando la validación básica (controlador) de la lógica de búsqueda (modelo). La validación de campos vacíos se realiza aquí para reducir la carga del modelo, mientras que la comparación de credenciales permanece en el modelo donde están almacenados los datos.

3.3.2 Archivo: controlador/perfil_controlador.dart

Administra la carga de datos de perfiles según el usuario autenticado. El método `cargarPerfil` recibe el tipo de usuario y construye un objeto `PerfilModelo` completo con toda la información.

```
// Carga el perfil segun el tipo de usuario
PerfilModelo cargarPerfil(String tipoPerfil) {
  // Cargar perfil segun el tipo usando toLowerCase para evitar errores
  if (tipoPerfil.toLowerCase() == 'mesias') {
    return PerfilModelo(
      nombre: 'Mesias',
      apellido: 'Orlando Mariscal',
      carrera: 'Software',
      telefono: '0960222440',
      edad: 22,
      email: '',
      sobreMi: 'Estudiante de Ingenieria en Software con experiencia...',
      githubUrl: 'https://github.com/AMVMesias?tab=repositories',
      habilidades: ['Python', 'JavaScript', 'TypeScript', 'Dart',
                    'C#', 'C++', 'HTML/CSS', 'Flutter'],
      experiencia: [
        Experiencia(empresa: 'Agente de Diagnostico de Otitis con IA',
                    cargo: 'Proyecto Academico',
                    periodo: '2024',
                    descripcion: 'Sistema de IA para diagnostico...'),
        Experiencia(empresa: 'Snake con IA', ...),
        // ... mas experiencias
      ],
      educacion: [
        Educacion(institucion: 'Universidad de las Fuerzas Armadas ESPE',
                  titulo: 'Ingenieria en Software',
                  periodo: '2020 - Presente'),
      ],
    );
  } else if (tipoPerfil.toLowerCase() == 'gabriel') {
    // Estructura similar para Gabriel con sus datos especificos
  } else if (tipoPerfil.toLowerCase() == 'brayan') {
    // Estructura similar para Bryan con sus datos especificos
  }
  // Por defecto retornar perfil de mesias como fallback
  return PerfilModelo(...);
}
```

Listing 3: Carga de perfil en perfil_controlador.dart

El controlador organiza los datos quemados de los tres usuarios en estructuras `PerfilModelo`. Cada perfil incluye listas de objetos `Experiencia` y `Educacion` definidos en el modelo. Se utiliza `toLowerCase()` para hacer la comparación insensible a mayúsculas. El caso por defecto retorna el perfil de Mesias como medida de seguridad.

3.4 Capa Vista - Atomic Design

3.4.1 Átomos

Los átomos son componentes básicos reutilizables ubicados en `vista/atomos/`. Se crearon 4 átomos principales: `texto_titulo.dart` para renderizar títulos con estilo consistente, `campo_texto.dart` para campos de entrada con etiqueta y controlador, `boton_primario.dart` para botones principales con color primario del tema, y `boton_secundario.dart` para botones secundarios con estilo outlined.

```dart
// Componente atomico reutilizable para campos de texto
class CampoTexto extends StatelessWidget {
  final String etiqueta;  // Texto que se muestra como label
  final TextEditingController controlador;  // Controlador del TextField

  const CampoTexto({
    Key? key,
    required this.etiqueta,
    required this.controlador,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return TextField(
      controller: controlador,  // Conecta el controlador para obtener/establecer texto
      decoration: InputDecoration(
        labelText: etiqueta,  // Muestra la etiqueta dentro del campo
        border: OutlineInputBorder(),  // Borde rectangular del campo
      ),
    );
  }
}
```

Listing 4: Átomo campo_texto.dart

Este átomo encapsula un `TextField` de Flutter con configuración básica. Recibe dos parámetros obligatorios: la etiqueta que se muestra al usuario y el controlador que maneja el texto. Al ser un `StatelessWidget`, no mantiene estado propio, delegando la gestión del texto al controlador externo. Este patrón permite reutilizar el componente en múltiples pantallas con diferentes etiquetas.

```dart
// Boton principal con estilo del tema
class BotonPrimario extends StatelessWidget {
  final String etiqueta;  // Texto del boton
  final VoidCallback onPressed;  // Funcion al presionar

  const BotonPrimario({
    Key? key,
    required this.etiqueta,
    required this.onPressed,
  }) : super(key: key);
```

```
11
12    @override
13    Widget build(BuildContext context) {
14      return ElevatedButton(
15        onPressed: onPressed,  // Ejecuta la funcion proporcionada
16        child: Text(etiqueta),  // Muestra el texto
17      );
18    }
19  }
```

Listing 5: Átomo boton_primario.dart

El `BotonPrimario` utiliza `ElevatedButton` que automáticamente aplica el estilo definido en `tema_botones.dart`. El parámetro `onPressed` es de tipo `VoidCallback`, permitiendo pasar cualquier función sin parámetros. Este átomo se usa en la página de login para los botones Ïngresarz "Limpiar".

```
1  // Boton secundario con borde
2  class BotonSecundario extends StatelessWidget {
3    final String etiqueta;
4    final VoidCallback onPressed;
5
6    const BotonSecundario({
7      Key? key,
8      required this.etiqueta,
9      required this.onPressed,
10   }) : super(key: key);
11
12   @override
13   Widget build(BuildContext context) {
14     return OutlinedButton(
15       onPressed: onPressed,
16       child: Text(etiqueta),
17     );
18   }
19 }
```

Listing 6: Átomo boton_secundario.dart

El `BotonSecundario` usa `OutlinedButton` en lugar de `ElevatedButton`, mostrando solo el borde sin fondo sólido. Automáticamente toma el estilo definido en `outlinedButtonTema` del archivo de temas. La estructura es idéntica al botón primario, solo cambia el widget base de Flutter utilizado.

```
1  // Texto para titulos con estilo consistente
2  class TextoTitulo extends StatelessWidget {
3    final String texto;
4
5    const TextoTitulo(this.texto, {Key? key}) : super(key: key);
6
7    @override
8    Widget build(BuildContext context) {
9      return Text(
```

```
      texto,
      style: TextStyle(
        fontSize: 24,
        fontWeight: FontWeight.bold,
      ),
      textAlign: TextAlign.center,
    );
  }
}
```

Listing 7: Átomo texto_titulo.dart

El `TextoTitulo` estandariza los títulos con tamaño de fuente 24, peso bold y alineación centrada. Este átomo se utiliza en la página de login para mostrar "Bienvenido". Al tener el estilo definido dentro del componente, todos los títulos mantienen apariencia consistente sin necesidad de especificar el estilo cada vez que se usa.

3.4.2 Páginas

Las páginas se encuentran en `lib/vista/paginas/` y representan pantallas completas de la aplicación.

Archivo: lib/vista/paginas/pagina_login.dart

Implementa la interfaz de autenticación utilizando los átomos creados. Incluye dos campos de texto (usuario y contraseña), botón primario Ïngresar"que valida credenciales mediante el controlador y navega al perfil, y botón secundario "Limpiar"que resetea los campos y mensajes de error.

```
// Maneja el proceso de login
void _handleLogin() {
  // Obtener texto de los controladores de TextField
  final usuario = _usuarioController.text;
  final contrasena = _contrasenaController.text;

  // Llamar al controlador para intentar autenticacion
  final tipoPerfil = _controlador.intentarLogin(usuario, contrasena);

  // Verificar si el login fue exitoso (retorna tipo de perfil no vacio)
  if (tipoPerfil.isNotEmpty) {
    // Login exitoso: navegar a pagina de perfil
    // Se usa push (no pushReplacement) para mantener boton de retroceso
    Navigator.of(context).push(
      MaterialPageRoute(
        builder: (context) => PaginaPerfil(tipoPerfil: tipoPerfil),
      ),
    );
  } else {
    // Login fallido: mostrar mensaje de error generico por seguridad
    setState(() {
      _mensajeError = 'Usuario o contrasena incorrectos';
    });
```

```
  }
}

// Limpia los campos y mensajes de error
void _limpiarCampos() {
  _usuarioController.clear();
  _contrasenaController.clear();
  setState(() {
    _mensajeError = null;
  });
}
```

Listing 8: Manejo de login en pagina_login.dart

El método `_handleLogin` obtiene los valores de los controladores de texto y los envía al `LoginControlador`. Si el resultado no está vacío, significa que la autenticación fue exitosa y se navega a la página de perfil pasando el tipo de usuario. Se utiliza `Navigator.push` en lugar de `pushReplacement` para permitir el botón de retroceso. El mensaje de error es genérico por razones de seguridad, sin revelar si el usuario o la contraseña son incorrectos.

Archivo: lib/vista/paginas/pagina_perfil.dart

Muestra la información completa del usuario autenticado estructurada en secciones. Carga los datos mediante `PerfilControlador` en el `initState` de forma síncrona y presenta la información en formato de texto plano con `SingleChildScrollView` para evitar overflow.

```
class _PaginaPerfilState extends State<PaginaPerfil> {
  late PerfilModelo perfil;  // Variable para almacenar el perfil cargado

  @override
  void initState() {
    super.initState();
    // Cargar perfil inmediatamente al inicializar el estado
    // Se ejecuta una sola vez cuando se crea la pagina
    final controlador = PerfilControlador();
    perfil = controlador.cargarPerfil(widget.tipoPerfil);
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: ColorApp.fondo,  // Aplicar color de fondo del tema
      appBar: AppBar(
        title: const Text('Mi Perfil'),
        // Boton de retroceso automatico por Navigator.push
      ),
      body: _buildBody(),  // Construir cuerpo con datos del perfil
    );
  }

  Widget _buildBody() {
    return SingleChildScrollView(  // Permite scroll para contenido largo
      padding: const EdgeInsets.all(20),
      child: Column(
```

```
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          // Nombre centrado en la parte superior
          Center(child: Text(perfil.nombreCompleto, ...)),
          // Secciones: Sobre Mi, Educacion, Experiencia, Habilidades
          // Cada seccion usa ColorApp.primaio para titulos
        ],
      ),
    );
  }
}
```

Listing 9: Carga de datos en pagina_perfil.dart

La página utiliza `late` para declarar la variable perfil, que se inicializa en `initState` antes de construir la interfaz. La carga es síncrona (sin Future ni async) porque los datos están quemados en el controlador. El `SingleChildScrollView` permite desplazamiento cuando el contenido excede la altura de la pantalla, crucial cuando aparece el teclado o en dispositivos pequeños. Los títulos de sección utilizan `ColorApp.primaio` para mantener consistencia con el tema.

3.5 Sistema de Temas

El sistema de temas se encuentra en `lib/temas/` y define la apariencia visual global de la aplicación.

3.5.1 Archivo: lib/temas/esquema_color.dart

Define la paleta de colores centralizada de la aplicación garantizando consistencia visual en todos los componentes. Los colores se definen como constantes estáticas para acceso global sin necesidad de instanciar la clase.

```
class ColorApp {
  // Color principal de la aplicacion (verde profundo)
  // Usado en AppBar, titulos de seccion y elementos principales
  static const Color primaio = Color(0xFF2E7D32);

  // Color secundario (verde claro)
  // Usado en botones secundarios y elementos de apoyo
  static const Color secundario = Color(0xFF66BB6A);

  // Color de acento (cian)
  // Usado para resaltar elementos especificos
  static const Color acento = Color(0xFF26C6DA);

  // Color de fondo general
  static const Color fondo = Color(0xFFF5F5F5);
}
```

Listing 10: Definición de colores en esquema_color.dart

Los colores utilizan el formato hexadecimal de Flutter `0xFF` seguido del código de color. El modificador `static const` permite acceder a los colores como `ColorApp.primaio` desde cualquier parte de la aplicación sin crear instancias. Esta centralización facilita cambios globales de tema modificando un solo archivo.

3.5.2 Archivo: lib/temas/tema_general.dart

Configura el tema global de la aplicación mediante `ThemeData`, integrando todos los temas específicos de componentes individuales. Combina la configuración de AppBar, botones, formularios, fondo y tipografía en un objeto único que se aplica en el `MaterialApp` del archivo principal.

```
class TemaGeneral {
  static ThemeData get claro {
    return ThemeData(
      // Aplicar tema personalizado de AppBar
      appBarTheme: TemaAppBar.appBarTema,

      // Aplicar tema de botones elevados y outlined
      elevatedButtonTheme: TemaBotones.elevatedButtonTema,
      outlinedButtonTheme: TemaBotones.outlinedButtonTema,

      // Aplicar tema de campos de formulario
      inputDecorationTheme: TemaFormulario.inputTema,

      // Aplicar color de fondo general
      scaffoldBackgroundColor: TemaFondo.colorFondo,

      // Aplicar configuracion de tipografia
      textTheme: Tipografia.textTheme,
    );
  }
}
```

Listing 11: Configuración del tema general en tema_general.dart

El tema general actúa como punto central de configuración, importando y combinando los temas específicos de cada componente. Esta estructura modular permite modificar estilos individuales sin afectar otros componentes, facilitando el mantenimiento y la consistencia visual.

3.5.3 Archivo: lib/temas/tema_appbar.dart

Configura el estilo del AppBar aplicando el color primario y ajustando la tipografía del título.

```
class TemaAppBar {
  static AppBarTheme get appBarTema {
    return AppBarTheme(
      backgroundColor: ColorApp.primaio,  // Color verde profundo
```

```
      foregroundColor: Colors.white,  // Texto blanco
      elevation: 0,  // Sin sombra
      centerTitle: true,  // Titulo centrado
      titleTextStyle: TextStyle(
        fontSize: 20,
        fontWeight: FontWeight.bold,
        color: Colors.white,
      ),
    );
  }
}
```

Listing 12: Configuración de AppBar en tema_appbar.dart

El `AppBarTheme` define propiedades globales para todos los AppBar de la aplicación. La propiedad `elevation: 0` elimina la sombra predeterminada, dando apariencia plana. El `centerTitle: true` centra el título en lugar de alinearlo a la izquierda.

3.5.4 Archivo: lib/temas/tema_botones.dart

Define los estilos para botones primarios (ElevatedButton) y secundarios (OutlinedButton) manteniendo coherencia visual.

```
class TemaBotones {
  // Estilo para BotonPrimario (ElevatedButton)
  static ElevatedButtonThemeData get elevatedButtonTema {
    return ElevatedButtonThemeData(
      style: ElevatedButton.styleFrom(
        backgroundColor: ColorApp.primaio,  // Fondo verde
        foregroundColor: Colors.white,  // Texto blanco
        padding: EdgeInsets.symmetric(vertical: 15),
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8),
        ),
      ),
    );
  }

  // Estilo para BotonSecundario (OutlinedButton)
  static OutlinedButtonThemeData get outlinedButtonTema {
    return OutlinedButtonThemeData(
      style: OutlinedButton.styleFrom(
        foregroundColor: ColorApp.primaio,  // Texto verde
        side: BorderSide(color: ColorApp.primaio, width: 2),  // Borde verde
        padding: EdgeInsets.symmetric(vertical: 15),
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8),
        ),
      ),
    );
  }
```

```
}
```

Listing 13: Configuración de botones en tema_botones.dart

Los botones primarios tienen fondo sólido del color primario mientras los secundarios solo tienen borde. Ambos comparten el mismo padding y border radius para mantener consistencia. El `RoundedRectangleBorder` con `borderRadius` crea esquinas redondeadas de 8 píxeles.

3.5.5 Archivo: lib/temas/tema_formulario.dart

Configura el estilo de los campos de texto (TextField) definiendo bordes, colores y espaciado.

```dart
class TemaFormulario {
  static InputDecorationTheme get inputTema {
    return InputDecorationTheme(
      border: OutlineInputBorder(
        borderRadius: BorderRadius.circular(8),
        borderSide: BorderSide(color: Colors.grey),
      ),
      focusedBorder: OutlineInputBorder(
        borderRadius: BorderRadius.circular(8),
        borderSide: BorderSide(color: ColorApp.primaio, width: 2),
      ),
      contentPadding: EdgeInsets.symmetric(horizontal: 15, vertical: 15),
      labelStyle: TextStyle(color: Colors.grey[700]),
    );
  }
}
```

Listing 14: Configuración de formularios en tema_formulario.dart

El `InputDecorationTheme` define dos estados de borde: `border` para estado normal (gris) y `focusedBorder` para cuando el campo tiene foco (verde primario). El `contentPadding` ajusta el espaciado interno del texto dentro del campo. Esta configuración se aplica automáticamente a todos los TextField de la aplicación.

3.6 Navegación entre Pantallas

La navegación se implementa en `lib/vista/paginas/pagina_login.dart` utilizando `Navigator.push` en lugar de `pushReplacement` para mantener el botón de retroceso automático en el AppBar.

```dart
// Navegar a pagina de perfil pasando el tipo de usuario
Navigator.of(context).push(
  MaterialPageRoute(
    builder: (context) => PaginaPerfil(tipoPerfil: tipoPerfil),
  ),
```

```
);
```

Listing 15: Navegación en pagina_login.dart

El método `push` apila la nueva pantalla sobre la actual, manteniendo el historial de navegación. Esto permite al usuario retroceder al login usando el botón de retroceso nativo. El parámetro `tipoPerfil` se pasa al constructor de `PaginaPerfil` para identificar qué datos cargar.

3.7 Validación de Datos en Múltiples Niveles

El sistema implementa validación en tres niveles distribuidos en diferentes archivos del proyecto.

3.7.1 Primera Validación: Campos Vacíos en Controlador

El archivo `lib/controlador/login_controlador.dart` realiza una validación inicial de campos vacíos antes de consultar el modelo, evitando procesamiento innecesario.

```dart
String intentarLogin(String usuario, String contrasena) {
  // Primera validacion: campos no vacios
  if (usuario.trim().isEmpty || contrasena.trim().isEmpty) {
    return '';  // Retorna vacio indicando fallo
  }
  // Continuar con validacion de credenciales...
}
```

Listing 16: Validación de campos vacíos en login_controlador.dart

El método `trim()` elimina espacios en blanco al inicio y final. Si algún campo está vacío después del trim, se retorna cadena vacía sin consultar el modelo. Esta validación rápida mejora el rendimiento.

3.7.2 Segunda Validación: Comparación de Credenciales en Modelo

El archivo `lib/modelo/usuario_modelo.dart` busca coincidencias exactas en la lista de usuarios válidos implementando comparación case-insensitive para el usuario.

```dart
static String validarCredenciales(String usuario, String contrasena) {
  if (usuario.trim().isEmpty || contrasena.trim().isEmpty) {
    return '';
  }

  try {
    // Buscar usuario con firstWhere
    final usuarioEncontrado = usuariosValidos.firstWhere(
      (u) => u.usuario.toLowerCase() == usuario.toLowerCase().trim()
          && u.contrasena == contrasena,  // Contrasena exacta
    );
    return usuarioEncontrado.tipoPerfil;
```

```
  } catch (e) {
    return '';  // Usuario no encontrado
  }
}
```

Listing 17: Búsqueda de usuario en usuario_modelo.dart

El método `firstWhere` busca el primer elemento que cumple la condición. El nombre de usuario se compara ignorando mayúsculas con `toLowerCase()`, pero la contraseña debe coincidir exactamente. Si no se encuentra coincidencia, `firstWhere` lanza excepción que se captura retornando cadena vacía.

3.7.3 Validación Visual: Mensajes de Error Genéricos

El archivo `lib/vista/paginas/pagina_login.dart` muestra mensajes de error genéricos por razones de seguridad, sin revelar si el usuario o la contraseña son incorrectos.

```
if (tipoPerfil.isNotEmpty) {
  // Login exitoso
  Navigator.of(context).push(...);
} else {
  // Login fallido: mensaje generico
  setState(() {
    _mensajeError = 'Usuario o contrasena incorrectos';
  });
}
```

Listing 18: Manejo de errores en pagina_login.dart

El mensaje genérico Üsuario o contraseña incorrectos"no especifica cuál campo es incorrecto, evitando que atacantes determinen usuarios válidos mediante intentos repetidos. El método `setState` actualiza la interfaz mostrando el error en rojo.

4 Resultados

4.1 Implementación Completa

Se implementó exitosamente una aplicación móvil con sistema de autenticación y perfiles siguiendo el patrón MVC. La estructura del proyecto comprende 2 archivos de modelo, 2 archivos de controlador, 7 archivos de átomos, 2 archivos de páginas y 8 archivos de configuración de tema, totalizando 21 archivos Dart organizados.

4.2 Componentes Desarrollados

Se crearon 4 componentes atómicos reutilizables que mantienen consistencia visual en toda la aplicación. El sistema de temas personalizado aplica la paleta de colores verde en todos los componentes. La arquitectura MVC permite modificar la lógica sin afectar la interfaz y viceversa.

4.3 Funcionalidad del Sistema

El sistema de autenticación valida correctamente las tres cuentas de usuario registradas. La navegación entre pantallas funciona fluidamente con transiciones nativas de Flutter. Los perfiles muestran información completa organizada en secciones claras. El botón de limpiar en login resetea todos los campos y mensajes de error.

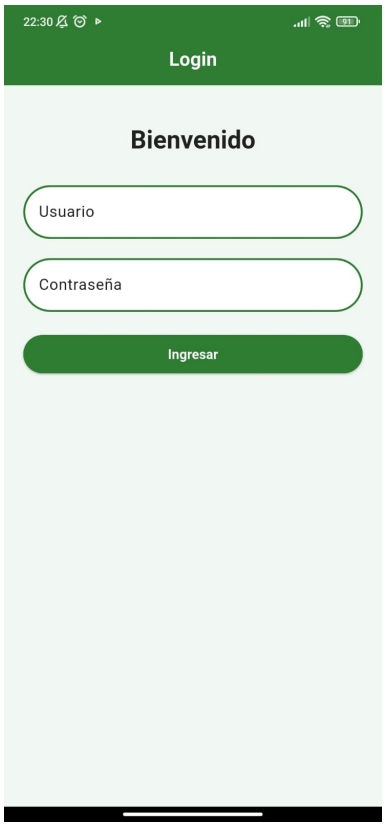
4.4 Evidencias de Funcionamiento

4.4.1 Pantalla de Login

La Figura 1 presenta la interfaz de autenticación con campos para usuario y contraseña, implementando los componentes atómicos desarrollados.

Figura 1

Interfaz de Autenticación



Nota. Captura de pantalla de la aplicación mostrando la página de login.

4.4.2 Validación de Credenciales

La Figura 2 muestra el mensaje de error cuando se ingresan credenciales incorrectas, evidenciando el funcionamiento del sistema de validación.

Figura 2

Mensaje de Error en Autenticación

Nota. Captura de pantalla de la aplicación mostrando el mensaje de error de autenticación.

4.4.3 Perfil de Usuario Mesias

La Figura 3 presenta el perfil completo del usuario Mesias con información personal, académica y profesional organizada en secciones.

Figura 3

Perfil Completo - Mesias Orlando Mariscal

Nota. Captura de pantalla de la aplicación mostrando el perfil del usuario logueado.

4.4.4 Perfil de Usuario Gabriel

La Figura 4 muestra el perfil del usuario Gabriel con su información académica y experiencia en desarrollo web y aplicaciones móviles.

Figura 4

Perfil Completo - Gabriel Murillo

Nota. Captura de pantalla de la aplicación mostrando el perfil del usuario logueado.

4.4.5 Perfil de Usuario Bryan

La Figura 5 presenta el perfil del usuario Bryan con experiencia en desarrollo web y machine learning.

Figura 5

Perfil Completo - Bryan Roberto Quispe Romero

Nota. Captura de pantalla de la aplicación mostrando el perfil del usuario logueado.

4.5 Análisis de Cumplimiento

Se cumplieron todos los objetivos específicos planteados. La arquitectura MVC se implementó correctamente separando responsabilidades. Los componentes Atomic Design garantizan reutilización y consistencia. El sistema de autenticación valida correctamente las credenciales. Los perfiles muestran información completa con el tema personalizado aplicado.

5 Conclusiones

En conclusión, la implementación del patrón Modelo-Vista-Controlador demostró ser efectiva para organizar el código separando modelo, controlador y vista, estableciendo límites claros entre la lógica de negocio, la validación y la presentación visual. Esta separación facilita el mantenimiento futuro del código y permite modificaciones independientes en cada capa sin afectar las demás, cumpliendo con el primer objetivo específico planteado.

Finalmente, la metodología Atomic Design permitió crear componentes reutilizables mediante átomos y moléculas consistentes en toda la aplicación. Los átomos como `CampoTexto`,

`BotonPrimario` y `BotonSecundario` estandarizaron la interfaz, pudiendo ser empleados en nuevas pantallas y reduciendo el tiempo de desarrollo. Esta modularidad representa una ventaja significativa para proyectos que requieren escalabilidad, cumpliendo con el segundo objetivo específico.

En síntesis, se crearon exitosamente perfiles de usuario con información personal, académica y profesional estructurada que se muestra al acceder exitosamente mediante el sistema de autenticación. Los tres perfiles implementados (Mesias, Gabriel y Bryan) contienen datos completos organizados en secciones claras, cumpliendo con el tercer objetivo específico y proporcionando bases sólidas para futuras mejoras.

6 Recomendaciones

Se recomienda que se implemente un sistema de gestión de estado más robusto como Provider o Riverpod para manejar datos de usuario globalmente, permitiendo mejorar la separación del patrón MVC y facilitando el acceso a información del perfil sin necesidad de pasar parámetros entre pantallas, escalando la arquitectura implementada.

Se recomienda que se expandan los componentes atómicos creando una biblioteca documentada que especifique el uso de cada átomo y molécula, facilitando la estandarización de nuevas interfaces y acelerando el desarrollo de pantallas adicionales manteniendo la consistencia visual establecida por Atomic Design.

Se recomienda que se desarrollen perfiles adicionales integrando bases de datos o APIs para cargar información dinámica, permitiendo que el sistema de perfiles escale más allá de los tres usuarios actuales y aprovechando la estructura de datos ya establecida en los modelos.

7 Referencias

Frost, B. (2016). *Atomic design*. Brad Frost Web. `https://atomicdesign.bradfrost.com/`

Google LLC. (2023). *Flutter documentation*. `https://docs.flutter.dev/`

Reenskaug, T. (1979). *The original MVC reports*. Xerox PARC.

8 Anexos

8.1 Anexo A: Repositorio del Código Fuente

El código completo del proyecto, incluyendo todos los archivos de la carpeta lib con modelos, controladores, vistas y temas, se encuentra disponible en el siguiente repositorio de GitHub:

https://github.com/AMVMesias/Desarrollo-Movil/tree/main/1P/Lab2/lib