INFORME ACADÉMICO / TÉCNICO

1. Datos Generales

Título del Informe:	Tarea 2.1 Consumo de APIs + Arquitectura Limpia + Provider
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:	28 de noviembre de 2025

Índice

1 Introducción

El presente informe documenta el desarrollo de una aplicación móvil que consume una API REST externa para mostrar información de personajes, implementando Clean Architecture y el patrón Provider para la gestión del estado. El proyecto demuestra la integración de servicios web externos con una arquitectura de software moderna y escalable.

Para la solución se utilizó Flutter como framework principal, complementado con Provider para la gestión reactiva del estado y el paquete HTTP para las peticiones a la API. La arquitectura se estructuró en capas: data, domain y presentation, permitiendo una clara separación de responsabilidades.

La implementación resultante consume la API de Game of Thrones (ThronesAPI) para obtener y mostrar una lista de personajes con sus respectivas imágenes y detalles. Esta aplicación demuestra las ventajas de combinar el consumo de APIs REST con patrones arquitectónicos modernos en el desarrollo móvil.

1.1 Objetivos

1.1.1 Objetivo General

Desarrollar una aplicación móvil que consuma una API REST externa mediante la implementación de Clean Architecture y Provider, utilizando Flutter y el paquete HTTP como herramientas de desarrollo, para demostrar la integración de servicios web con arquitectura limpia.

1.1.2 Objetivos Específicos

1. Investigar los fundamentos teóricos de consumo de APIs REST, Clean Architecture, Provider y Atomic Design para establecer las bases conceptuales del desarrollo.

2. Implementar la capa de datos con datasources, modelos y repositorios que encapsulen la lógica de comunicación con la API externa.

3. Desarrollar la capa de presentación con ViewModels y vistas que consuman los datos obtenidos de la API mediante Provider.

4. Aplicar la metodología Atomic Design organizando los widgets en átomos, moléculas y organismos para crear una interfaz visual atractiva y reutilizable.

5. Crear un sistema de visualización con ListView, DropdownButton para filtrado por familia y paginación para mostrar los personajes de manera eficiente.

2 Marco Teórico

2.1 APIs REST

REST (Representational State Transfer) es un estilo arquitectónico para sistemas distribuidos que define un conjunto de restricciones para crear servicios web escalables. Las APIs REST utilizan métodos HTTP estándar como GET, POST, PUT y DELETE para realizar operaciones sobre recursos. La comunicación se realiza típicamente en formato JSON, facilitando la interoperabilidad entre diferentes plataformas y lenguajes de programación.

2.2 Clean Architecture

Clean Architecture es un enfoque de diseño de software propuesto por Robert C. Martin que organiza el código en capas concéntricas con dependencias que apuntan hacia el interior. La capa más interna contiene las entidades y reglas de negocio, mientras que las capas externas manejan detalles de implementación como interfaces de usuario y fuentes de datos. Esta separación permite que el núcleo de la aplicación sea independiente de frameworks y herramientas externas.

2.3 Flutter y Provider

Flutter es un framework de código abierto desarrollado por Google que permite crear aplicaciones multiplataforma desde una única base de código. Utiliza el lenguaje Dart y se caracteriza por su sistema de widgets reactivos. Provider es un paquete de gestión de estado recomendado por el equipo de Flutter que implementa el patrón ChangeNotifier, permitiendo que los widgets escuchen y reaccionen a cambios en el estado de la aplicación de manera eficiente y declarativa.

2.4 Paquete HTTP en Flutter

El paquete HTTP de Dart proporciona una API de alto nivel para realizar peticiones HTTP. Permite ejecutar métodos GET, POST, PUT y DELETE de manera asíncrona, manejar respuestas y errores, y trabajar con headers personalizados. Es la herramienta estándar para consumir APIs REST en aplicaciones Flutter.

2.5 Atomic Design

Atomic Design es una metodología de diseño de interfaces propuesta por Brad Frost que organiza los componentes de UI en cinco niveles jerárquicos: átomos, moléculas, organismos, plantillas y páginas. Los átomos son los elementos más básicos e indivisibles (botones, textos, iconos). Las moléculas combinan átomos para formar componentes funcionales (tarjetas, formularios sim-

ples). Los organismos agrupan moléculas para crear secciones completas de interfaz. Esta metodología promueve la reutilización de componentes y la consistencia visual en toda la aplicación.

3 Desarrollo

3.1 Arquitectura del Proyecto

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

- `lib/data/datasource/`: Contiene las clases que realizan las peticiones HTTP a la API
- `lib/data/models/`: Define los modelos de datos con serialización JSON
- `lib/data/repositories/`: Implementa los repositorios que abstraen el acceso a datos
- `lib/domain/entities/`: Contiene las entidades del dominio
- `lib/domain/usecases/`: Define los casos de uso de la aplicación
- `lib/presentation/viewmodels/`: Gestiona el estado de la aplicación
- `lib/presentation/views/`: Contiene las pantallas de la interfaz
- `lib/presentation/routes/`: Define las rutas de navegación
- `lib/presentation/themes/`: Define el tema visual (app_theme.dart, family_colors.dart)
- `lib/presentation/widgets/atoms/`: Componentes atómicos básicos (Atomic Design)
- `lib/presentation/widgets/molecules/`: Combinaciones de átomos (Atomic Design)
- `lib/presentation/widgets/organisms/`: Secciones completas de UI (Atomic Design)

3.2 Capa de Datos - DataSource

3.2.1 Archivo: lib/data/datasource/base_datasource.dart

Define la interfaz abstracta que deben implementar todos los datasources. Establece el contrato para obtener la lista de personajes desde cualquier fuente de datos.

```dart
import '../models/character_model.dart';

abstract class BaseDataSource {
  /// Devuelve la lista de personajes.
  Future<List<CharacterModel>> fetchCharacters();
```

```
}
```

Listing 1: Interfaz BaseDataSource - lib/data/datasource/base_datasource.dart

La clase abstracta utiliza el tipo `Future` para indicar que la operación es asíncrona, ya que las peticiones HTTP requieren tiempo de espera. Esta abstracción permite cambiar la fuente de datos (API, base de datos local, mock) sin modificar el resto del código.

3.2.2 Archivo: lib/data/datasource/characters_api_datasource.dart

Implementa el consumo de la API REST de ThronesAPI utilizando el paquete HTTP.

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;
import '../models/character_model.dart';
import 'base_datasource.dart';

class CharactersApiDataSource implements BaseDataSource {
  final String base = 'https://thronesapi.com/api/v2/Characters';

  @override
  Future<List<CharacterModel>> fetchCharacters() async {
    final uri = Uri.parse(base);
    final resp = await http.get(uri);
    if (resp.statusCode != 200) {
      throw Exception('Error fetching characters: ${resp.statusCode}');
    }
    final List<dynamic> data = json.decode(resp.body);
    return data.map((e) => CharacterModel.fromJson(
      e as Map<String, dynamic>)).toList();
  }
}
```

Listing 2: Consumo de API - lib/data/datasource/characters_api_datasource.dart

El método `fetchCharacters` realiza una petición GET a la URL base de la API. La validación del código de estado HTTP es fundamental: si `resp.statusCode` es diferente de 200, se lanza una excepción con el código de error, evitando que datos inválidos lleguen a la interfaz. El cuerpo de la respuesta se decodifica de JSON a una lista dinámica, y luego se mapea cada elemento a un objeto `CharacterModel` usando el factory constructor `fromJson`.

Validaciones implementadas en el DataSource:

- Validación de código de estado HTTP (200 para GET exitoso, 201/200 para POST)
- Lanzamiento de excepciones descriptivas con el código de error
- Manejo de diferentes tipos de respuesta según el método HTTP

La clase también implementa métodos adicionales para obtener un personaje por ID y crear nuevos personajes:

```dart
Future<CharacterModel> fetchCharacterById(int id) async {
  final uri = Uri.parse('$base/$id');
  final resp = await http.get(uri);
  if (resp.statusCode != 200) {
    throw Exception('Error fetching character $id: ${resp.statusCode}');
  }
  final Map<String, dynamic> data = json.decode(resp.body);
  return CharacterModel.fromJson(data);
}

Future<CharacterModel> createCharacter(
    Map<String, dynamic> payload) async {
  final uri = Uri.parse(base);
  final resp = await http.post(uri,
    body: json.encode(payload),
    headers: {'Content-Type': 'application/json'});
  if (resp.statusCode != 201 && resp.statusCode != 200) {
    throw Exception('Error creating character: ${resp.statusCode}');
  }
  final Map<String, dynamic> data = json.decode(resp.body);
  return CharacterModel.fromJson(data);
}
```

Listing 3: Métodos adicionales - lib/data/datasource/characters_api_datasource.dart

Estos métodos demuestran el uso de diferentes verbos HTTP: GET para obtener datos y POST para crear nuevos recursos. El método POST incluye headers para especificar el tipo de contenido JSON.

3.3 Capa de Datos - Modelo

3.3.1 Archivo: lib/data/models/character_model.dart

Define el modelo de datos con serialización/deserialización JSON.

```dart
class CharacterModel {
  final int id;
  final String firstName, lastName, fullName;
  final String title, family, image, imageUrl;

  CharacterModel({required this.id, ...});

  factory CharacterModel.fromJson(Map<String, dynamic> json) {
    return CharacterModel(
      id: json['id'] is int ? json['id']
        : int.tryParse('${json['id']}') ?? 0,
      firstName: json['firstName'] ?? '',
      // Validacion con ?? para campos nulos
      ...
    );
  }

  Map<String, dynamic> toJson() => {'id': id, ...};
```

```
}
```

Listing 4: Modelo CharacterModel - lib/data/models/character_model.dart

Validaciones implementadas en el Modelo:

- Verificación de tipo para el campo `id`: comprueba si es entero, caso contrario intenta convertirlo con `int.tryParse`
- Operador `??` para proporcionar cadenas vacías cuando los campos son nulos

3.4 Capa de Datos - Repositorio

3.4.1 Archivo: lib/data/repositories/base_repository.dart

Define la interfaz abstracta que deben implementar todos los repositorios.

```dart
import '../../domain/entities/producto_entity.dart';

abstract class BaseRepository {
  Future<List<ProductoEntity>> getProductos();
}
```

Listing 5: Interfaz BaseRepository - lib/data/repositories/base_repository.dart

Esta clase abstracta establece el contrato para los repositorios, permitiendo diferentes implementaciones (API, local, mock) sin afectar las capas superiores.

3.4.2 Archivo: lib/data/repositories/character_repository_impl.dart

Actúa como intermediario entre el datasource y las capas superiores.

```dart
import '../datasource/characters_api_datasource.dart';
import '../models/character_model.dart';

class CharacterRepositoryImpl {
  final CharactersApiDataSource _ds;

  CharacterRepositoryImpl(this._ds);

  Future<List<CharacterModel>> getAllCharacters() =>
    _ds.fetchCharacters();

  Future<CharacterModel> getCharacterById(int id) =>
    _ds.fetchCharacterById(id);

  Future<CharacterModel> createCharacter(
    Map<String, dynamic> payload) =>
    _ds.createCharacter(payload);
}
```

Listing 6: Repositorio - lib/data/repositories/character_repository_impl.dart

El repositorio recibe el datasource mediante inyección de dependencias en el constructor. Esto facilita las pruebas unitarias al permitir inyectar mocks del datasource. Los métodos del repositorio delegan las operaciones al datasource correspondiente.

3.5 Capa de Dominio - Entidades

3.5.1 Archivo: lib/domain/entities/producto_entity.dart

Define la entidad del dominio con las propiedades esenciales del negocio.

```dart
class ProductoEntity {
  final String nombre;
  final double precio;
  final int stock;
  final String categoria;

  ProductoEntity({
    required this.nombre,
    required this.precio,
    required this.stock,
    required this.categoria,
  });
}
```

Listing 7: Entidad ProductoEntity - lib/domain/entities/producto_entity.dart

La entidad representa un objeto del dominio puro, sin dependencias de frameworks externos. Define las propiedades requeridas mediante el constructor con parámetros nombrados obligatorios.

3.6 Capa de Dominio - Casos de Uso

3.6.1 Archivo: lib/domain/usecases/getproductos_usecase.dart

Implementa la lógica de negocio para obtener productos.

```dart
import '../entities/producto_entity.dart';
import '../../data/repositories/base_repository.dart';

class GetProductosUseCase {
  final BaseRepository repository;

  GetProductosUseCase(this.repository);

  Future<List<ProductoEntity>> call() async {
    return await repository.getProductos();
  }
}
```

Listing 8: Caso de Uso GetProductosUseCase - lib/domain/usecases/getproductos_usecase.dart

El caso de uso encapsula una acción específica del negocio. Recibe el repositorio mediante inyección de dependencias y expone un método `call()` que puede invocarse como función. Esto sigue el principio de responsabilidad única.

3.7 Capa de Presentación - ViewModel

3.7.1 Archivo: lib/presentation/viewmodels/base_viewmodel.dart

Define la clase base abstracta para todos los ViewModels.

```dart
import 'package:flutter/material.dart';

abstract class BaseViewModel extends ChangeNotifier {
  bool loading = false;

  void setLoading(bool value) {
    loading = value;
    notifyListeners();
  }
}
```

Listing 9: BaseViewModel - lib/presentation/viewmodels/base_viewmodel.dart

La clase abstracta proporciona funcionalidad común para todos los ViewModels, incluyendo el manejo del estado de carga y la notificación a los listeners cuando cambia el estado.

3.7.2 Archivo: lib/presentation/viewmodels/character_viewmodel.dart

Gestiona el estado de la aplicación implementando `ChangeNotifier`.

```dart
import 'package:flutter/foundation.dart';
import '../../data/repositories/character_repository_impl.dart';
import '../../data/models/character_model.dart';

class CharacterViewModel extends ChangeNotifier {
  final CharacterRepositoryImpl repository;

  bool loading = false;
  List<CharacterModel> characters = [];

  CharacterViewModel(this.repository);

  Future<void> loadCharacters() async {
    loading = true;
    notifyListeners();
    try {
      characters = await repository.getAllCharacters();
    } catch (e) {
      characters = [];
    }
    loading = false;
    notifyListeners();
```

```
  }
}
```

Listing 10: ViewModel - lib/presentation/viewmodels/character_viewmodel.dart

El ViewModel mantiene dos estados: `loading` indica si hay una petición en curso, y `characters` almacena la lista de personajes obtenidos. El método `loadCharacters` implementa un patrón de manejo de errores robusto:

Validaciones y manejo de errores en el ViewModel:

- Bloque `try-catch` que captura cualquier excepción durante la petición
- Asignación de lista vacía en caso de error, evitando estados nulos
- Notificación a listeners antes y después de la operación

3.8 Implementación de Filtrado y Paginación

El ViewModel implementa filtrado por familia y paginación con las siguientes funciones clave:

```
// Extrae familias unicas para el DropdownButton
void _extractFamilies() {
  final familySet = <String>{};
  for (var c in _allCharacters) {
    if (c.family.isNotEmpty) familySet.add(c.family);
  }
  _families = familySet.toList()..sort();
}

// Filtra personajes por familia seleccionada
void filterByFamily(String? family) {
  _selectedFamily = family;
  _currentPage = 0;
  _filteredCharacters = (family == null || family.isEmpty)
    ? _allCharacters
    : _allCharacters.where((c) => c.family == family).toList();
  notifyListeners();
}

// Paginacion: 10 elementos por pagina
List<CharacterModel> get _paginatedCharacters {
  final start = _currentPage * itemsPerPage;
  final end = start + itemsPerPage;
  if (start >= _filteredCharacters.length) return [];
  return _filteredCharacters.sublist(start,
    end > _filteredCharacters.length ? _filteredCharacters.length : end);
}

// Navegacion entre paginas con validacion
void nextPage() { if (hasNextPage) { _currentPage++; notifyListeners(); } }
```

```
31 void previousPage() { if (hasPreviousPage) { _currentPage--; notifyListeners(); } }
```

Listing 11: Filtrado y Paginación - lib/presentation/viewmodels/character_viewmodel.dart

Validaciones de filtrado y paginación:

- Uso de `Set` para eliminar familias duplicadas
- Validación de `family.isNotEmpty` antes de agregar
- Reinicio de página a 0 al cambiar filtro
- Validación de índices para evitar desbordamiento en `sublist`
- Comprobación de `hasNextPage/hasPreviousPage` antes de navegar

3.9 Capa de Presentación - Vista

3.9.1 Archivo: lib/presentation/views/home_page.dart

La vista principal consume el estado del ViewModel mediante Provider y utiliza Atomic Design.

```dart
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Consumer<CharacterViewModel>(
      builder: (context, viewModel, child) {
        return Scaffold(
          appBar: _buildAppBar(context, viewModel),
          body: Column(children: [
            // Organismo: Lista de personajes
            Expanded(child: CharacterListOrganism(
              isLoading: viewModel.loading,
              characters: viewModel.characters,
              onCharacterTap: (c) => _showCharacterDetail(context, c),
            )),
            // Molecula: Controles de paginacion
            PaginationControls(
              currentPage: viewModel.currentPage,
              totalPages: viewModel.totalPages,
              hasPrevious: viewModel.hasPreviousPage,
              hasNext: viewModel.hasNextPage,
              onPrevious: viewModel.previousPage,
              onNext: viewModel.nextPage,
            ),
          ]),
        );
      },
    );
  }
```

```
}
```

Listing 12: HomePage con Atomic Design - lib/presentation/views/home_page.dart

Validaciones implementadas en la Vista:

- `CharacterListOrganism` valida `isLoading` para mostrar LoadingSpinner
- Valida lista vacía para mostrar `EmptyState`
- `PaginationControls` se oculta si `totalPages <= 1`
- Botones deshabilitados en límites usando `hasPrevious/hasNext`

3.10 Implementación de Atomic Design

Para mejorar la organización y reutilización de componentes, se implementó Atomic Design.

3.10.1 Archivo: lib/presentation/widgets/atoms/custom_avatar.dart

Átomo para mostrar imágenes de perfil con borde y caché.

```dart
class CustomAvatar extends StatelessWidget {
  final String imageUrl;
  final double size;
  final Color? borderColor;

  @override
  Widget build(BuildContext context) {
    return Container(
      width: size, height: size,
      decoration: BoxDecoration(
        shape: BoxShape.circle,
        border: Border.all(color: borderColor ?? AppTheme.goldColor),
      ),
      child: ClipOval(
        child: CachedNetworkImage(imageUrl: imageUrl, fit: BoxFit.cover),
      ),
    );
  }
}
```

Listing 13: CustomAvatar - lib/presentation/widgets/atoms/custom_avatar.dart

3.10.2 Archivo: lib/presentation/widgets/atoms/app_text.dart

Átomo para textos con estilos predefinidos.

```dart
class AppText extends StatelessWidget {
  final String text;
  final TextType type;
  final Color? color;

  @override
  Widget build(BuildContext context) {
    return Text(text, style: _getStyle().copyWith(color: color));
  }

  TextStyle _getStyle() {
    switch (type) {
      case TextType.headingLarge: return AppTheme.headingLarge;
      case TextType.headingMedium: return AppTheme.headingMedium;
      case TextType.body: return AppTheme.bodyMedium;
      case TextType.caption: return AppTheme.caption;
      // ...
    }
  }
}

enum TextType { headingLarge, headingMedium, bodyLarge, body, caption, badge }
```

Listing 14: AppText - lib/presentation/widgets/atoms/app_text.dart

3.10.3 Archivo: lib/presentation/widgets/atoms/custom_button.dart

Átomo para botones con gradiente y estados.

```dart
class CustomButton extends StatelessWidget {
  final String? text;
  final IconData? icon;
  final VoidCallback? onPressed;
  final bool isEnabled;
  final ButtonType type;

  @override
  Widget build(BuildContext context) {
    final isDisabled = !isEnabled || onPressed == null;
    return AnimatedContainer(
      decoration: BoxDecoration(
        gradient: isDisabled ? null : _getGradient(),
        color: isDisabled ? AppTheme.dividerColor : null,
      ),
      child: InkWell(
        onTap: isDisabled ? null : onPressed,
        child: Row(children: [
          if (icon != null) Icon(icon, color: isDisabled
            ? AppTheme.textSecondary : Colors.white),
          if (text != null) Text(text!, style: TextStyle(
            color: isDisabled ? AppTheme.textSecondary : Colors.white)),
        ]),
      ),
    );
```

```
  }
}

enum ButtonType { primary, secondary, outline }
```

Listing 15: CustomButton - lib/presentation/widgets/atoms/custom_button.dart

3.10.4 Archivo: lib/presentation/widgets/atoms/badge.dart

Átomo para mostrar etiquetas con gradiente.

```
class CustomBadge extends StatelessWidget {
  final String text;
  final Color? backgroundColor;
  final IconData? icon;

  @override
  Widget build(BuildContext context) {
    final bgColor = backgroundColor ?? AppTheme.accentColor;
    return Container(
      decoration: BoxDecoration(
        gradient: LinearGradient(colors: [bgColor, bgColor.withValues(alpha: 0.7)]),
        borderRadius: BorderRadius.circular(20),
      ),
      child: Row(children: [
        if (icon != null) Icon(icon, size: 12, color: Colors.white),
        Text(text, style: AppTheme.badge),
      ]),
    );
  }
}
```

Listing 16: CustomBadge - lib/presentation/widgets/atoms/badge.dart

3.10.5 Archivo: lib/presentation/widgets/atoms/loading_spinner.dart

Átomo para indicador de carga con animación.

```
class LoadingSpinner extends StatelessWidget {
  final double size;
  final String? message;

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(children: [
        CircularProgressIndicator(
          valueColor: AlwaysStoppedAnimation<Color>(AppTheme.goldColor),
        ),
        if (message != null) Text(message!, style: AppTheme.bodyMedium),
      ]),
    );
  }
```

```
}
```

Listing 17: LoadingSpinner - lib/presentation/widgets/atoms/loading_spinner.dart

3.10.6 Archivo: lib/presentation/widgets/atoms/empty_state.dart

Átomo para estados vacíos o sin resultados.

```dart
class EmptyState extends StatelessWidget {
  final IconData icon;
  final String title;
  final String? subtitle;

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Column(children: [
        Icon(icon, size: 60, color: AppTheme.textSecondary),
        Text(title, style: AppTheme.headingMedium),
        if (subtitle != null) Text(subtitle!, style: AppTheme.bodyMedium),
      ]),
    );
  }
}
```

Listing 18: EmptyState - lib/presentation/widgets/atoms/empty_state.dart

3.10.7 Archivo: lib/presentation/widgets/molecules/character_card.dart

Molécula que combina CustomAvatar con información del personaje.

```dart
class CharacterCard extends StatelessWidget {
  final CharacterModel character;
  final VoidCallback? onTap;

  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: BoxDecoration(gradient: AppTheme.cardGradient),
      child: Row(children: [
        CustomAvatar(
          imageUrl: character.imageUrl,
          borderColor: FamilyColors.getColorByFamily(character.family),
        ),
        Column(children: [
          Text(character.fullName),
          // Badge con color de la casa
          Container(
            decoration: BoxDecoration(
              gradient: LinearGradient(colors: [
                FamilyColors.getColorByFamily(character.family),
                FamilyColors.getColorByFamily(character.family)
                    .withValues(alpha: 0.7),
```

```
              ]),
            ),
            child: Text(character.family),
          ),
        ]),
      ]),
    );
  }
}
```

Listing 19: CharacterCard - lib/presentation/widgets/molecules/character_card.dart

3.10.8 Archivo: lib/presentation/widgets/molecules/family_filter.dart

Molécula para filtrar personajes por familia usando DropdownButton.

```
class FamilyFilter extends StatelessWidget {
  final String? selectedFamily;
  final List<String> families;
  final ValueChanged<String?> onFamilyChanged;

  @override
  Widget build(BuildContext context) {
    return Container(
      decoration: BoxDecoration(
        gradient: LinearGradient(colors: [
          AppTheme.surfaceColor.withOpacity(0.6),
          AppTheme.surfaceColor.withOpacity(0.4),
        ]),
        border: Border.all(color: AppTheme.goldColor.withOpacity(0.3)),
      ),
      child: DropdownButton<String?>(
        value: selectedFamily,
        hint: Text('Todas las casas'),
        items: [
          DropdownMenuItem(value: null, child: Text('Todas las casas')),
          ...families.map((family) => DropdownMenuItem(
            value: family,
            child: Row(children: [
              Icon(Icons.shield, color: _getFamilyColor(family)),
              Text(family),
            ]),
          )),
        ],
        onChanged: onFamilyChanged,
      ),
    );
  }

  Color _getFamilyColor(String family) {
    if (family.contains('Stark')) return Color(0xFF607D8B);
    if (family.contains('Lannister')) return Color(0xFFD4AF37);
    if (family.contains('Targaryen')) return Color(0xFFE94560);
    // ... mas casas
```

```
    return AppTheme.accentColor;
  }
}
```

Listing 20: FamilyFilter - lib/presentation/widgets/molecules/family_filter.dart

3.10.9 Archivo: lib/presentation/widgets/molecules/pagination_controls.dart

Molécula para navegación entre páginas.

```
class PaginationControls extends StatelessWidget {
  final int currentPage;
  final int totalPages;
  final bool hasPrevious;
  final bool hasNext;
  final VoidCallback onPrevious;
  final VoidCallback onNext;

  @override
  Widget build(BuildContext context) {
    if (totalPages <= 1) return SizedBox.shrink();

    return Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        CustomButton(
          icon: Icons.chevron_left,
          text: 'Anterior',
          onPressed: hasPrevious ? onPrevious : null,
          isEnabled: hasPrevious,
          type: ButtonType.outline,
        ),
        Container(
          child: Text('${currentPage + 1} / $totalPages'),
        ),
        CustomButton(
          text: 'Siguiente',
          icon: Icons.chevron_right,
          onPressed: hasNext ? onNext : null,
          isEnabled: hasNext,
          type: ButtonType.primary,
        ),
      ],
    );
  }
}
```

Listing 21: PaginationControls - lib/presentation/widgets/molecules/pagination_controls.dart

3.10.10 Archivo: lib/presentation/widgets/organisms/character_list_organism.dart

Organismo que combina moléculas para formar la lista completa.

```dart
class CharacterListOrganism extends StatelessWidget {
  final bool isLoading;
  final List<CharacterModel> characters;

  @override
  Widget build(BuildContext context) {
    if (isLoading) return LoadingSpinner(message: 'Cargando...');
    if (characters.isEmpty) return EmptyState(title: 'Sin resultados');

    return ListView.builder(
      itemCount: characters.length,
      itemBuilder: (context, index) => CharacterCard(
        character: characters[index],
      ),
    );
  }
}
```

Listing 22: CharacterListOrganism - lib/presentation/widgets/organisms/character_list_organism.dart

3.10.11 Archivo: lib/presentation/widgets/index.dart

Barrel file para exportar todos los widgets de Atomic Design.

```dart
// Barrel file para exportar todos los widgets de Atomic Design
// ATOMS
export 'atoms/app_text.dart';
export 'atoms/badge.dart';
export 'atoms/custom_avatar.dart';
export 'atoms/custom_button.dart';
export 'atoms/empty_state.dart';
export 'atoms/loading_spinner.dart';

// MOLECULES
export 'molecules/character_card.dart';
export 'molecules/family_filter.dart';
export 'molecules/pagination_controls.dart';

// ORGANISMS
export 'organisms/character_list_organism.dart';
```

Listing 23: Barrel File - lib/presentation/widgets/index.dart

Este archivo permite importar todos los widgets desde un único punto, simplificando los imports en otras partes del código.

3.11 Temas Visuales

3.11.1 Archivo: lib/presentation/themes/app_theme.dart

Define los colores, gradientes y estilos de texto de la aplicación.

```dart
class AppTheme {
  static const Color primaryColor = Color(0xFF1A1A2E);   // Azul oscuro
  static const Color accentColor = Color(0xFFE94560);    // Rojo dragon
  static const Color goldColor = Color(0xFFD4AF37);      // Dorado Lannister

  static const LinearGradient cardGradient = LinearGradient(
    colors: [Color(0xFF1E2A47), Color(0xFF16213E)],
  );
}
```

Listing 24: AppTheme - lib/presentation/themes/app_theme.dart

3.11.2 Archivo: lib/presentation/themes/family_colors.dart

Define colores específicos para cada casa de Westeros.

```dart
class FamilyColors {
  static const Color stark = Color(0xFF607D8B);       // Gris acero
  static const Color lannister = Color(0xFFD4AF37);   // Dorado
  static const Color targaryen = Color(0xFFE94560);   // Rojo dragon
  static const Color baratheon = Color(0xFFFFD700);   // Amarillo dorado
  // ... mas casas

  static Color getColorByFamily(String family) {
    final familyLower = family.toLowerCase();
    if (familyLower.contains('stark')) return stark;
    if (familyLower.contains('lannister')) return lannister;
    // ... validacion para cada casa
    return AppTheme.accentColor;
  }
}
```

Listing 25: FamilyColors - lib/presentation/themes/family_colors.dart

3.12 Sistema de Rutas

3.12.1 Archivo: lib/presentation/routes/app_routes.dart

Define las rutas de navegación de manera centralizada.

```dart
class AppRoutes {
  static Map<String, WidgetBuilder> routes = {
    "/": (_) => const HomePage(),
  };
}
```

Listing 26: AppRoutes - lib/presentation/routes/app_routes.dart

3.13 Punto de Entrada de la Aplicación

3.13.1 Archivo: lib/main.dart

Configura la inyección de dependencias y Provider como gestor de estado.

```dart
void main() {
  final charactersDs = CharactersApiDataSource();
  final charactersRepo = CharacterRepositoryImpl(charactersDs);
  final characterVm = CharacterViewModel(charactersRepo)..loadCharacters();
  runApp(MyApp(characterVm: characterVm));
}

class MyApp extends StatelessWidget {
  final CharacterViewModel characterVm;

  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [ChangeNotifierProvider.value(value: characterVm)],
      child: MaterialApp(
        title: "Game of Thrones - API",
        theme: AppTheme.theme,
        routes: AppRoutes.routes,
      ),
    );
  }
}
```

Listing 27: main.dart - lib/main.dart

3.14 Dependencias del Proyecto

3.14.1 Archivo: pubspec.yaml

```yaml
dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^1.0.8
  provider: ^6.0.0           # Gestion de estado
  http: ^1.1.0               # Peticiones HTTP a APIs
  cached_network_image: ^3.3.0  # Cache de imagenes
```

Listing 28: Dependencias - pubspec.yaml

Las dependencias clave incluyen: `provider` para la gestión reactiva del estado, `http` para realizar peticiones HTTP a la API REST, y `cached_network_image` para descargar y almacenar en caché las imágenes de los personajes.

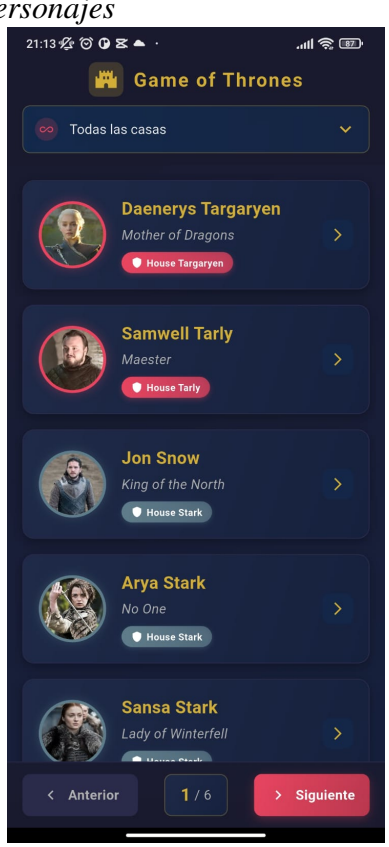
4 Resultados

Los resultados obtenidos durante el desarrollo del proyecto demuestran el cumplimiento satisfactorio de los objetivos planteados. La aplicación consume exitosamente la API de ThronesAPI y muestra la lista de personajes con sus respectivos detalles, incluyendo funcionalidades de filtrado por familia y paginación.

La pantalla principal presenta una lista paginada de personajes obtenidos de la API. Cada elemento muestra la imagen del personaje (con caché para optimizar rendimiento), el nombre completo, el título nobiliario y la familia a la que pertenece. En la parte superior se encuentra el DropdownButton para filtrar por familia y en la parte inferior los controles de paginación.

Figura 1

Pantalla Principal con Lista de Personajes

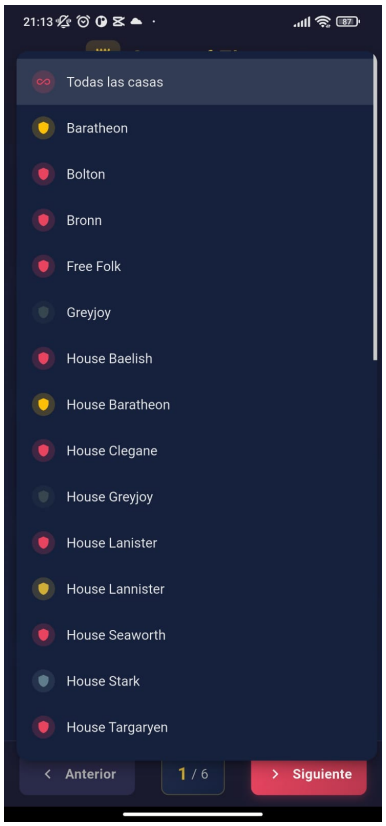


Nota. Captura del emulador Android mostrando la lista de personajes de ThronesAPI con tarjetas estilizadas. Fuente: Autor.

El sistema de filtrado mediante DropdownButton permite al usuario seleccionar una familia específica para visualizar únicamente los personajes pertenecientes a esa casa. Al desplegar el selector, se muestran todas las casas extraídas dinámicamente de los datos de la API.

Figura 2

Selector de Casas Desplegado

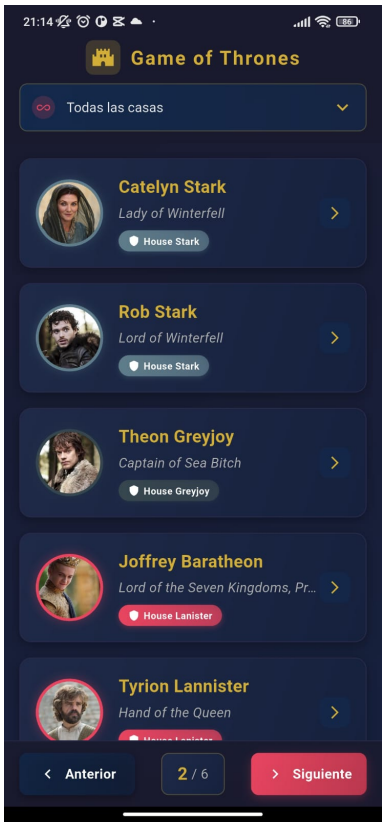


Nota. Captura del emulador Android con el DropdownButton desplegado mostrando las casas de Westeros. Fuente: Autor.

La funcionalidad de paginación divide la lista de personajes en páginas de 10 elementos cada una, permitiendo una navegación fluida mediante los botones de anterior y siguiente. El indicador muestra la página actual y el total de páginas disponibles.

Figura 3

Navegación a la Segunda Página



Nota. Captura del emulador Android exhibiendo la navegación por páginas con controles y nuevo contenido. Fuente: Autor.

La arquitectura implementada demuestra una clara separación de responsabilidades: el datasource maneja exclusivamente las peticiones HTTP, el repositorio abstrae el acceso a datos, el ViewModel gestiona el estado de la aplicación incluyendo el filtrado y la paginación, y la vista se encarga únicamente de renderizar la interfaz. Esta separación facilita el mantenimiento y las pruebas del código.

El sistema de gestión de estado con Provider permite que la interfaz se actualice automáticamente cuando los datos están disponibles o cuando el usuario interactúa con el filtro o la paginación, mostrando un indicador de carga mientras se realiza la petición a la API y la lista filtrada y paginada una vez completada.

5 Conclusiones

Como resultado, la investigación de los fundamentos teóricos de consumo de APIs REST, Clean Architecture, Provider y Atomic Design permitió establecer una base conceptual sólida que guió las decisiones de diseño durante todo el desarrollo, resultando en una estructura de código organizada que separa claramente las responsabilidades de cada capa.

Es posible destacar que la implementación de la capa de datos con datasources, modelos y repositorios demostró ser efectiva para encapsular la lógica de comunicación con la API externa, facilitando su modificación y prueba de manera independiente. El uso del paquete HTTP permitió realizar peticiones asíncronas de manera eficiente.

En resumen, el desarrollo de la capa de presentación con ViewModels y vistas que consumen datos mediante Provider produjo una interfaz reactiva que se actualiza automáticamente cuando cambia el estado. El patrón ChangeNotifier facilita la notificación de cambios a los widgets suscritos.

Es importante señalar que la aplicación de Atomic Design organizando widgets en átomos (CustomAvatar, CustomBadge, LoadingSpinner), moléculas (CharacterCard, FamilyFilter, PaginationControls) y organismos (CharacterListOrganism) resultó en componentes altamente reutilizables. Además, la clase `FamilyColors` permite asignar colores únicos a cada casa de Westeros, mejorando la identificación visual de los personajes.

Para concluir, la creación del sistema de visualización con ListView, DropdownButton para filtrado por familia y paginación resultó en una interfaz completa que permite a los usuarios explorar los personajes de manera eficiente, filtrando por casa/familia y navegando entre páginas de 10 elementos cada una.

6 Recomendaciones

Es conveniente profundizar en el estudio de patrones adicionales como Repository Pattern con interfaces y Dependency Injection con paquetes como GetIt para complementar la comprensión de Clean Architecture en proyectos de mayor escala.

En próximas iteraciones, sería beneficioso implementar manejo de errores más robusto con estados específicos (loading, success, error) y mensajes informativos para el usuario cuando la API no esté disponible.

Resulta oportuno expandir la funcionalidad agregando búsqueda por nombre mediante un SearchDelegate, almacenamiento local con SQLite o Hive para funcionamiento offline, y sincronización de datos en segundo plano.

Sería valioso implementar tests unitarios para el datasource, repositorio y ViewModel, así como tests de widgets para validar el comportamiento de la interfaz de usuario incluyendo el filtrado

y la paginación.

7 Referencias Bibliográficas

Dart Team. (2024). HTTP package. Pub.dev. https://pub.dev/packages/http

Flutter. (2024). Provider package. Pub.dev. https://pub.dev/packages/provider

Frost, B. (2016). Atomic Design. Brad Frost. https://atomicdesign.bradfrost.com/

Google. (2024). Flutter documentation. Flutter.dev. https://docs.flutter.dev/

Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.

ThronesAPI. (2024). Game of Thrones API. https://thronesapi.com/

8 Anexos

El código fuente completo del proyecto se encuentra disponible en el siguiente repositorio de GitHub:

https://github.com/AMVMesias/Desarrollo-Movil/tree/main/2P/Tarea%202.1%20Consumo%20de%20APIS%2BArquitectura%20Limpia%20%2B%20Provider

El repositorio incluye toda la estructura del proyecto con los archivos de configuración, código fuente organizado según Clean Architecture y documentación adicional.