🏗️ Arquitectura del Proyecto Napphy Services

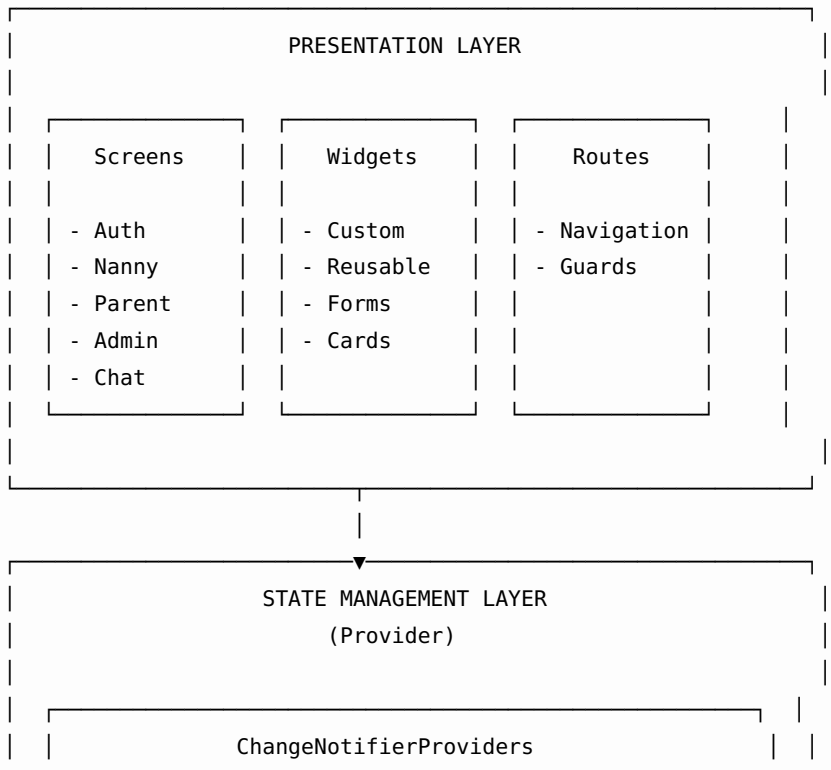
📐 Visión General

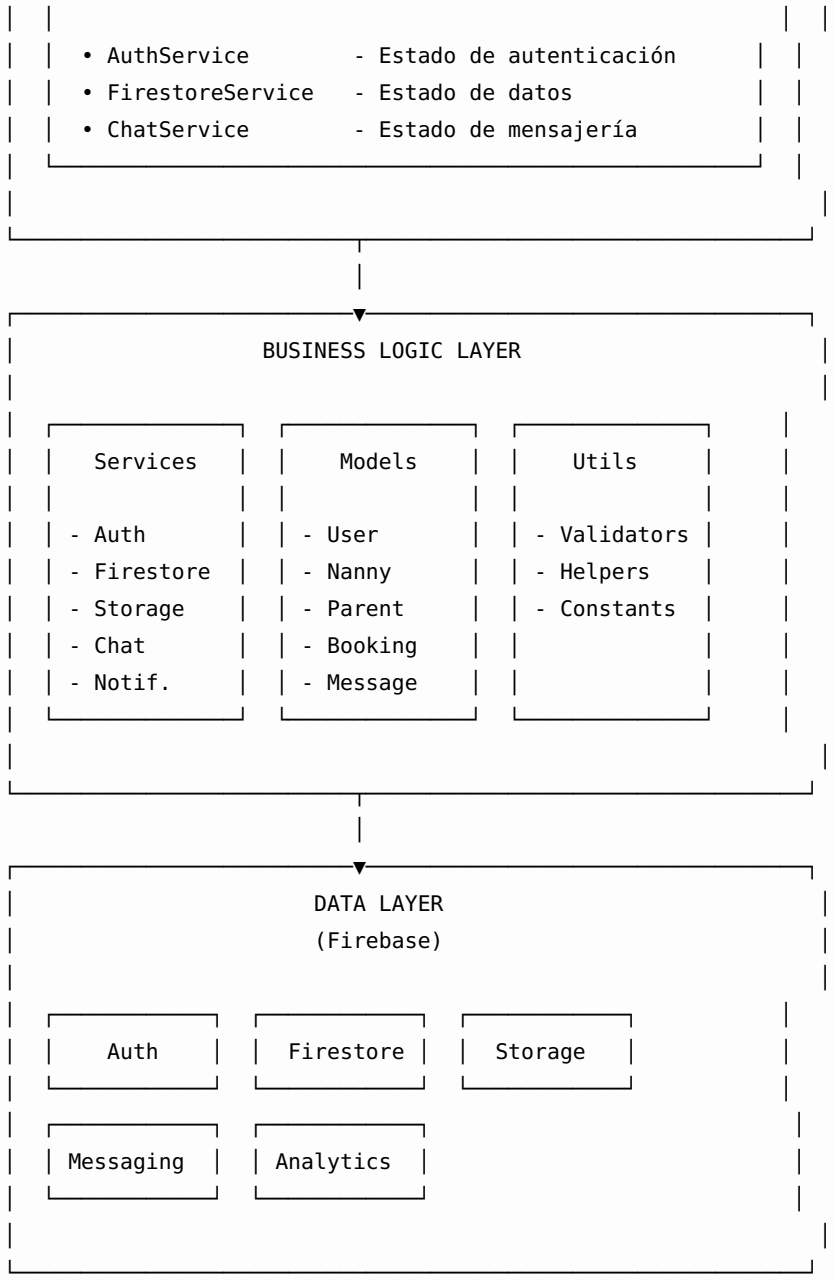
Napphy Services sigue una arquitectura limpia y modular basada en capas, utilizando el patrón **Provider** para la gestión de estado y **Firebase** como backend.

🎯 Principios de Diseño

1. **Separación de Responsabilidades**: Cada capa tiene una responsabilidad clara y definida
2. **Modularidad**: Componentes independientes y reutilizables
3. **Escalabilidad**: Fácil de extender con nuevas funcionalidades
4. **Mantenibilidad**: Código limpio y bien documentado
5. **Testabilidad**: Componentes fáciles de probar unitariamente

📊 Arquitectura en Capas





🔄 Flujo de Datos

1. Flujo de Lectura (Read)

```
Usuario interactúa con UI
         ↓
Screen solicita datos
         ↓
Provider (Service) recibe la solicitud
         ↓
Service consulta Firebase
```

```
         ↓
Firebase retorna datos
         ↓
Service actualiza estado (notifyListeners)
         ↓
UI se reconstruye automáticamente
         ↓
Usuario ve los datos actualizados
```

2. Flujo de Escritura (Write)

```
Usuario ingresa datos en formulario
         ↓
Screen valida datos localmente
         ↓
Screen llama al método del Service
         ↓
Service procesa y valida datos
         ↓
Service envía datos a Firebase
         ↓
Firebase confirma operación
         ↓
Service actualiza estado local
         ↓
UI muestra mensaje de éxito
```

3. Flujo de Autenticación

```
Usuario ingresa credenciales
         ↓
LoginScreen valida formato
         ↓
AuthService.signInWithEmail()
         ↓
Firebase Authentication verifica
         ↓
Si exitoso: AuthService actualiza currentUser
         ↓
AuthService.loadUserData() desde Firestore
         ↓
Navigation redirige según rol de usuario
         ↓
Usuario ve su dashboard correspondiente
```

📦 Componentes Principales

1. Models (Modelos de Datos)

Ubicación: `lib/models/`

Los modelos representan las entidades de datos de la aplicación:

```
// Ejemplo: UserModel
class UserModel {
  final String id;
  final String email;
  final String fullName;
  final UserRole role;

  // Factory constructor para Firebase
  factory UserModel.fromFirestore(DocumentSnapshot doc) { ... }

  // Método para convertir a Map
  Map<String, dynamic> toMap() { ... }

  // CopyWith para inmutabilidad
  UserModel copyWith({...}) { ... }
}
```

Responsabilidades: - Definir estructura de datos - Serialización/Deserialización - Validación de datos (básica)

2. Services (Servicios)

Ubicación: `lib/services/`

Los servicios encapsulan la lógica de negocio y comunicación con Firebase:

AuthService

```
class AuthService extends ChangeNotifier {
  final FirebaseAuth _auth = FirebaseAuth.instance;

  User? get currentUser => _auth.currentUser;
  UserModel? _currentUserModel;

  Future<UserCredential?> signInWithEmail(...) { ... }
  Future<UserCredential?> registerWithEmail(...) { ... }
  Future<void> signOut() { ... }
  Future<void> updateUserProfile(...) { ... }
}
```

Responsabilidades: - Gestionar autenticación - Mantener estado del usuario actual - Sincronizar datos del usuario - Notificar cambios a los listeners

FirestoreService

```
class FirestoreService extends ChangeNotifier {
  final FirebaseFirestore _firestore = FirebaseFirestore.instance;

  // CRUD Operations
  Future<void> createNannyProfile(NannyModel nanny) { ... }
  Future<NannyModel?> getNannyProfile(String userId) { ... }
  Future<void> updateNannyProfile(...) { ... }

  // Streams for real-time updates
  Stream<List<NannyModel>> getNanniesStream(...) { ... }
  Stream<List<BookingModel>> getBookingsForNanny(...) { ... }
}
```

Responsabilidades: - Operaciones CRUD en Firestore - Búsquedas y consultas complejas - Streams para datos en tiempo real - Gestión de transacciones

ChatService

```
class ChatService {
  final FirebaseFirestore _firestore = FirebaseFirestore.instance;

  Future<String> getOrCreateChat(...) { ... }
  Future<void> sendMessage(...) { ... }
  Stream<List<MessageModel>> getMessages(String chatId) { ... }
  Future<void> markMessagesAsRead(...) { ... }
}
```

Responsabilidades: - Gestionar conversaciones - Enviar/recibir mensajes - Mantener historial de chat - Gestionar estado de lectura

3. Screens (Pantallas)

Ubicación: `lib/screens/`

Las pantallas son StatefulWidget o StatelessWidget que componen la UI:

Estructura de una Screen:

```
class ExampleScreen extends StatefulWidget {
  const ExampleScreen({super.key});
```

```
  @override
  State<ExampleScreen> createState() => _ExampleScreenState();
}

class _ExampleScreenState extends State<ExampleScreen> {
  // Estado local
  final _formKey = GlobalKey<FormState>();

  @override
  void initState() {
    super.initState();
    _loadData();
  }

  Future<void> _loadData() async {
    // Cargar datos iniciales
  }

  @override
  Widget build(BuildContext context) {
    // Consumir servicios con Provider
    final service = Provider.of<SomeService>(context);

    return Scaffold(
      appBar: AppBar(title: Text('Example')),
      body: _buildBody(service),
    );
  }

  Widget _buildBody(SomeService service) {
    // Construir UI basada en el estado del servicio
    if (service.isLoading) {
      return Center(child: CircularProgressIndicator());
    }

    return ListView(...);
  }
}
```

Responsabilidades: - Renderizar UI - Manejar interacciones del usuario - Validar formularios - Navegar entre pantallas - Consumir y reaccionar a cambios de estado

4. Widgets (Componentes Reutilizables)

Ubicación: `lib/widgets/`

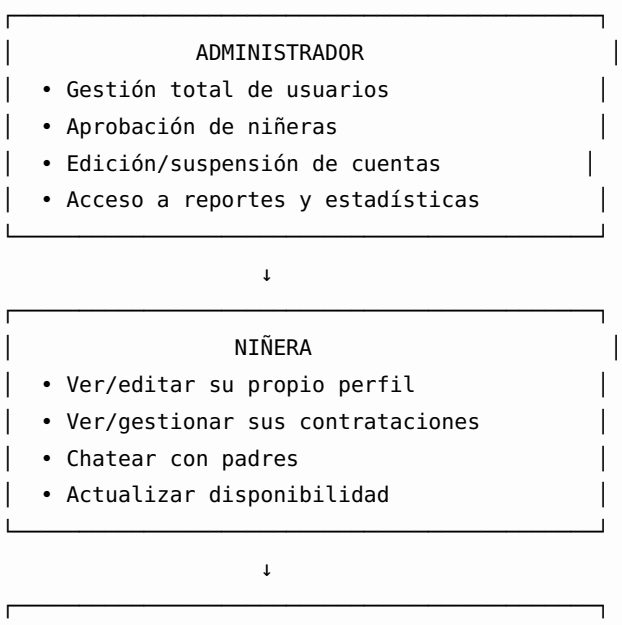
Widgets personalizados y reutilizables:

```
class CustomButton extends StatelessWidget {
  final String text;
  final VoidCallback onPressed;
  final bool isLoading;

  const CustomButton({
    super.key,
    required this.text,
    required this.onPressed,
    this.isLoading = false,
  });

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: isLoading ? null : onPressed,
      child: isLoading
        ? CircularProgressIndicator()
        : Text(text),
    );
  }
}
```

Responsabilidades: - Encapsular componentes reutilizables - Mantener consistencia visual - Reducir duplicación de código

🔐 Seguridad y Permisos

Niveles de Acceso



```
|                 PADRE                          |
|  • Buscar niñeras                              |
|  • Ver perfiles de niñeras aprobadas           |
|  • Crear contrataciones                        |
|  • Chatear con niñeras                         |
|  • Calificar servicios                         |
└────────────────────────────────────────────────┘
```

Guards de Navegación

```
// Verificación de autenticación
if (authService.currentUser == null) {
  Navigator.pushReplacementNamed(context, Routes.login);
  return;
}

// Verificación de rol
switch (authService.currentUserModel!.role) {
  case UserRole.nanny:
    Navigator.pushReplacementNamed(context, Routes.nannyHome);
    break;
  case UserRole.parent:
    Navigator.pushReplacementNamed(context, Routes.parentHome);
    break;
  case UserRole.admin:
    Navigator.pushReplacementNamed(context, Routes.adminDashboard);
    break;
}
```

🎛️ Gestión de Estado

Provider Pattern

Configuración Global:

```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (_) => AuthService()),
        ChangeNotifierProvider(create: (_) => FirestoreService()),
      ],
      child: NapphyServicesApp(),
    ),
  );
}
```

Consumo en Widgets:

```
// Método 1: Consumer (solo reconstruye el Consumer)
Consumer<AuthService>(
  builder: (context, authService, child) {
    return Text(authService.currentUser?.email ?? 'No user');
  },
)

// Método 2: Provider.of (reconstruye todo el widget)
final authService = Provider.of<AuthService>(context);

// Método 3: Provider.of sin listen (no reconstruye)
final authService = Provider.of<AuthService>(context, listen:
false);
```

🔄 Patrones de Diseño Utilizados

1. Repository Pattern

Los Services actúan como repositorios, abstrayendo el acceso a datos.

2. Factory Pattern

Los modelos usan factory constructors para crear instancias desde Firebase.

3. Observer Pattern

Provider implementa el patrón Observer para la gestión de estado.

4. Singleton Pattern

Algunos servicios (como NotificationService) usan el patrón Singleton.

5. Strategy Pattern

Diferentes estrategias de autenticación (Email, Google, etc.)

🧪 Estrategia de Testing

Tests Unitarios

- Modelos: Serialización/Deserialización
- Services: Lógica de negocio
- Validators: Validación de datos

Tests de Integración

- Flujos completos de autenticación
- Operaciones CRUD en Firestore
- Navegación entre pantallas

Tests de Widget

- Renderizado de componentes
- Interacciones de usuario
- Estados de carga/error

🚀 Optimizaciones

1. Caché de Datos

- Persistencia de sesión con SharedPreferences
- Caché de imágenes con cached_network_image

2. Lazy Loading

- Paginación en listas largas
- Carga diferida de imágenes

3. Streams Optimizados

- Uso de StreamBuilder para datos en tiempo real
- Cancelación de suscripciones en dispose()

4. Build Optimization

- Keys para widgets que cambian de posición
- const constructors donde sea posible
- Separación de widgets para reconstrucciones parciales

🖨 Mejores Prácticas

1. **Naming Conventions**
   - Clases: PascalCase
   - Variables: camelCase
   - Constantes: SCREAMING_SNAKE_CASE
   - Archivos: snake_case
2. **Estructura de Archivos**
   - Un archivo por clase (generalmente)
   - Agrupación por funcionalidad, no por tipo
3. **Comentarios**

- Documentar clases y métodos públicos
- Explicar “por qué”, no “qué”

4. **Manejo de Errores**
   - Try-catch en operaciones asíncronas
   - Mensajes de error amigables al usuario
   - Logging para debugging
5. **Performance**
   - Evitar operaciones costosas en build()
   - Usar const constructors
   - Minimizar rebuilds innecesarios

Esta arquitectura permite que Napphy Services sea **escalable**, **mantenible** y **fácil de extender** con nuevas funcionalidades.