Desarrollo de Interfaces  
Unidad 12 - Consumo de APIs y Firebase con Flutter

short line

Autor: Sergi García

Actualizado Septiembre 2025

Licencia

**Reconocimiento - No comercial - CompartirIgual** (BY-NC-SA): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se ha de hacer con una licencia igual a la que regula la obra original.

Nomenclatura

A lo largo de este tema se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

📖 **Importante**

❕ **Atención**

💬 **Interesante**

**Índice**

[**1. Introducción 3**](#_i7n91wajds6v)

[**2. Consumo de APIs REST en Flutter con http 3**](#_eck7cgrl0ec5)

[**3. Uso de Firebase con Flutter 7**](#_ii5pd2qo9cbk)

[**4. Recursos recomendados para aprender Flutter 11**](#_ax44ni7spos4)

Unidad 12. Consumo de APIs y Firebase con Flutter

# 1. Introducción

En el desarrollo de aplicaciones móviles no basta con diseñar pantallas atractivas: la mayoría de las apps necesitan conectarse a servicios externos para ser realmente útiles.  
Por ejemplo, una app de noticias obtiene artículos de un servidor, una app de compras guarda pedidos en la nube y una app de mensajería sincroniza mensajes entre usuarios en tiempo real.

En Flutter, esa comunicación con el “mundo exterior” se puede lograr de dos maneras muy comunes:

1. Consumo de APIs REST, que nos permiten pedir o enviar información a un servidor usando el protocolo HTTP.
2. Uso de Firebase, una plataforma en la nube de Google que ofrece servicios listos para integrar: base de datos en tiempo real, almacenamiento de archivos, autenticación de usuarios, etc.

En esta unidad aprenderemos cómo trabajar con ambas herramientas, comprendiendo su utilidad y cómo encajan en el flujo de una aplicación Flutter.

# 2. Consumo de APIs REST en Flutter con http

Cuando hablamos de una **API REST**, nos referimos a un servicio web que sigue unas reglas comunes y que nos permite:

* Pedir datos (con una petición GET).
* Enviar información (con POST).
* Actualizar registros (con PUT).
* Eliminar datos (con DELETE).

La aplicación no tiene que saber cómo funciona internamente el servidor, solo necesita conocer la **dirección de la API** (la URL) y el **formato de la información** (generalmente JSON).

En Flutter usamos librerías como **http**, que actúa como un “cartero digital”:

* Nosotros preparamos la carta (la petición).
* La enviamos a la dirección correcta (el servidor).
* El cartero nos devuelve la respuesta (los datos).

Para organizar esa información, lo ideal es convertir el JSON en **objetos de Dart** (modelos). De esta forma, en lugar de trabajar con datos “sueltos” y confusos, los alumnos manejarán objetos con nombre, propiedades y tipos bien definidos.

Finalmente, esos datos se muestran en la interfaz con widgets como ListView, de manera que la app responde de forma dinámica al contenido que llega del servidor.

👉 **En resumen:** el consumo de APIs REST nos enseña a conectar nuestra app con servicios externos, transformando datos en objetos útiles que luego podemos mostrar en pantalla

En los siguientes ejemplos vamos a interactuar con servicios externos: obtener datos de un servidor, enviar información de formularios, autenticar usuarios, etc.  
Para esto se usan las APIs REST, que permiten a la app comunicarse con un backend mediante peticiones HTTP (GET, POST, PUT, DELETE, etc.).

Flutter no incluye un cliente HTTP en su SDK, pero existen librerías muy utilizadas como http, que es simple y ligera, ideal para proyectos pequeños y medianos.

**🔷 Hacer una petición GET**

El primer paso es aprender a leer datos desde una API.

| **import** 'dart:convert'; **import** 'package:http/http.dart' **as** http;  Future<**void**> obtenerDatos() **async** {  **final** url = Uri.parse('https://jsonplaceholder.typicode.com/posts');  **final** respuesta = **await** http.**get**(url);   **if** (respuesta.statusCode == 200) {  **final** datos = jsonDecode(respuesta.body);  print(datos);  } **else** {  **throw** Exception('Error al cargar datos');  } } |
| --- |

👉 Aquí usamos la API gratuita JSONPlaceholder, que devuelve posts de ejemplo en formato JSON.

**🔷 Crear un modelo en Dart**

Trabajar directamente con JSON es incómodo y propenso a errores.  
Por eso, convertimos la respuesta en objetos Dart tipados (clases modelo).

**Ejemplo de respuesta JSON:**

| {  "userId": 1,  "id": 1,  "title": "Título",  "body": "Contenido del post" } |
| --- |

**Modelo en Dart:**

| **class** **Post** {  **final** int userId;  **final** int id;  **final** String title;  **final** String body;   Post({  required **this**.userId,  required **this**.id,  required **this**.title,  required **this**.body,  });   **factory** Post.fromJson(Map<String, **dynamic**> json) {  **return** Post(  userId: json['userId'],  id: json['id'],  title: json['title'],  body: json['body'],  );  } } |
| --- |

**🔷 Convertir respuesta en lista de objetos**

Ahora podemos transformar el JSON en una lista de Post:

| Future<List<Post>> fetchPosts() **async** {  **final** url = Uri.parse('https://jsonplaceholder.typicode.com/posts');  **final** response = **await** http.**get**(url);   **if** (response.statusCode == 200) {  **final** List<**dynamic**> lista = jsonDecode(response.body);  **return** lista.map((json) => Post.fromJson(json)).toList();  } **else** {  **throw** Exception('Error al obtener posts');  } } |
| --- |

**🔷 Mostrar datos en un ListView**

Finalmente, integramos la llamada en un FutureBuilder para mostrar los datos en pantalla.

| **class** **PostPage** **extends** **StatefulWidget** {  **@override**  \_PostPageState createState() => \_PostPageState(); }  **class** **\_PostPageState** **extends** **State**<**PostPage**> {  late Future<List<Post>> \_futurePosts;   **@override**  **void** initState() {  **super**.initState();  \_futurePosts = fetchPosts();  }   **@override**  Widget build(BuildContext context) {  **return** Scaffold(  appBar: AppBar(title: Text("Posts")),  body: FutureBuilder<List<Post>>(  future: \_futurePosts,  builder: (context, snapshot) {  **if** (snapshot.connectionState == ConnectionState.waiting) {  **return** Center(child: CircularProgressIndicator());  }   **if** (snapshot.hasError) {  **return** Center(child: Text("Error: ${snapshot.error}"));  }   **final** posts = snapshot.data!;  **return** ListView.builder(  itemCount: posts.length,  itemBuilder: (context, index) {  **final** post = posts[index];  **return** ListTile(  title: Text(post.title),  subtitle: Text(post.body),  );  },  );  },  ),  );  } } |
| --- |

**📌 En Resumen**

* Usamos http para comunicarnos con APIs REST.
* Creamos modelos en Dart para tipar las respuestas y trabajar con objetos en lugar de mapas dinámicos.
* Implementamos FutureBuilder para manejar la carga de datos de forma reactiva.

👉 Esto sienta la base para consumir APIs más complejas (autenticación, envío de datos con POST, subida de archivos, etc.).

# 3. Uso de Firebase con Flutter

Firebase es como un **“backend listo para usar”** que nos evita tener que programar y mantener un servidor propio. Con él podemos:

* Guardar y consultar datos en una **base de datos en la nube** (Firestore).
* Autenticar usuarios con Google, email, redes sociales, etc.
* Almacenar archivos como imágenes o documentos.
* Enviar notificaciones push a los móviles.

La gran ventaja es que todo está ya preparado y optimizado por Google, y Flutter se integra muy bien con estos servicios.

**🔹 El papel de dotenv y los ficheros .env**

Cuando trabajamos con Firebase (o con cualquier servicio externo) necesitamos usar credenciales de acceso como claves de API, IDs de proyecto o dominios de autenticación. Estas claves son muy sensibles:

* No deben compartirse en el código fuente.
* No deberían subirse a repositorios públicos como GitHub.

Para resolverlo, se utiliza la librería dotenv junto con los ficheros .env.

* El archivo .env funciona como una caja cerrada donde guardamos nuestras claves.
* Dotenv se encarga de leer esas claves y ponerlas a disposición de la app en tiempo de ejecución.
* De esta forma, el código principal de la aplicación no contiene datos privados, y podemos tener distintos .env para diferentes entornos (desarrollo, pruebas, producción).

**🔹 Conexión con Firestore**

Una vez inicializado Firebase, podemos interactuar con Firestore para hacer operaciones muy similares a las que ya vimos con las APIs REST:

* **Leer datos** de la nube.
* **Añadir registros** nuevos.
* **Actualizar datos existentes**.
* **Eliminar registros**.

La diferencia es que ahora no hacemos peticiones HTTP a una URL, sino que nos conectamos directamente a la base de datos de Firebase, que guarda y sincroniza los datos en tiempo real.

👉 **En resumen:** Firebase nos permite tener un backend completo sin necesidad de programar un servidor, y con dotenv protegemos las claves de acceso que necesita nuestra aplicación.

Con los siguientes ejemplos veremos cómo usar Cloud Firestore para gestionar una lista de tareas.

**🔷 Conectar Flutter con Firebase**

Primero debemos inicializar Firebase en la app en el main.dart:

| Future<**void**> main() **async** {  WidgetsFlutterBinding.ensureInitialized();  **await** Firebase.initializeApp(  options: FirebaseOptions(  apiKey: dotenv.env['FIREBASE\_API\_KEY']!,  authDomain: dotenv.env['FIREBASE\_AUTH\_DOMAIN']!,  projectId: dotenv.env['FIREBASE\_PROJECT\_ID']!,  storageBucket: dotenv.env['FIREBASE\_STORAGE\_BUCKET']!,  messagingSenderId: dotenv.env['FIREBASE\_MESSAGING\_SENDER\_ID']!,  appId: dotenv.env['FIREBASE\_APP\_ID']!,  ),  );   runApp(**const** AplicacionTareas()); } |
| --- |

Con esto, nuestra app ya puede comunicarse con Firebase.

**🔷 Crear un modelo en Dart**

Al igual que con los Post de la API, necesitamos un modelo para las tareas:

| **class** **Tarea** {  String? documentId;  **final** String titulo;  **final** String categoria;  bool estaCompletada;   Tarea({  **this**.documentId,  required **this**.titulo,  required **this**.categoria,  **this**.estaCompletada = **false**,  });   **factory** Tarea.fromJson(Map<String, **dynamic**> json) {  **return** Tarea(  titulo: json['titulo'],  categoria: json['categoria'],  estaCompletada: json['estaCompletada'] ?? **false**,  );  }   Map<String, **dynamic**> toJson() {  **return** {  'titulo': titulo,  'categoria': categoria,  'estaCompletada': estaCompletada,  };  } } |
| --- |

**🔷 CRUD con Firestore (Proveedor)**

Creamos un Provider que gestione la lista de tareas y se comunique con Firestore:

| **class** **TareaProvider** **with** **ChangeNotifier** {  **final** FirebaseFirestore \_firestore = FirebaseFirestore.instance;  List<Tarea> \_tareas = [];   List<Tarea> **get** tareas => \_tareas;   Future<**void**> cargarDatos() **async** {  **final** snapshot = **await** \_firestore.collection('tareas').**get**();  \_tareas = snapshot.docs.map((doc) {  **final** data = doc.data()['tarea'] **as** Map<String, **dynamic**>;  **return** Tarea.fromJson(data)..documentId = doc.id;  }).toList();  notifyListeners();  }   Future<**void**> agregarTarea(Tarea tarea) **async** {  **final** docRef = \_firestore.collection('tareas').doc();  tarea.documentId = docRef.id;  **await** docRef.**set**({'tarea': tarea.toJson()});  \_tareas.add(tarea);  notifyListeners();  }   Future<**void**> eliminarTarea(int indice) **async** {  **final** tarea = \_tareas[indice];  **await** \_firestore.collection('tareas').doc(tarea.documentId).delete();  \_tareas.removeAt(indice);  notifyListeners();  } } |
| --- |

**🔷 Mostrar datos en un ListView**

En la UI podemos usar Consumer (de provider) para escuchar cambios y mostrar las tareas en pantalla:

| **class** **ListaTareasPage** **extends** **StatelessWidget** {  **@override**  Widget build(BuildContext context) {  **return** Scaffold(  appBar: AppBar(title: Text("Mis Tareas")),  body: Consumer<TareaProvider>(  builder: (context, tareaProvider, child) {  **final** tareas = tareaProvider.tareas;  **if** (tareas.isEmpty) {  **return** Center(child: Text("No hay tareas aún"));  }   **return** ListView.builder(  itemCount: tareas.length,  itemBuilder: (context, index) {  **final** tarea = tareas[index];  **return** ListTile(  title: Text(tarea.titulo),  subtitle: Text("Categoría: ${tarea.categoria}"),  trailing: IconButton(  icon: Icon(Icons.delete),  onPressed: () => tareaProvider.eliminarTarea(index),  ),  );  },  );  },  ),  floatingActionButton: FloatingActionButton(  onPressed: () {  **final** nuevaTarea = Tarea(titulo: "Ejemplo", categoria: "General");  Provider.of<TareaProvider>(context, listen: **false**)  .agregarTarea(nuevaTarea);  },  child: Icon(Icons.add),  ),  );  } } |
| --- |

**📌 En Resumen**

* Inicializamos Firebase con Firebase.initializeApp.
* Creamos un modelo Tarea en Dart para tipar los datos.
* Implementamos un Provider para CRUD con Firestore.
* Mostramos las tareas en un ListView de forma reactiva con Consumer.

👉 Esto sienta la base para integrar otras funcionalidades de Firebase como autenticación de usuarios, subida de archivos o notificaciones push.

# 4. Recursos recomendados para aprender Flutter

**📚 Documentación Oficial**

* Flutter Docs  
  🔗 [https://docs.flutter.dev](https://docs.flutter.dev/)
* Dart Language  
  🔗 <https://dart.dev/language>

**🎓 Cursos Gratuitos**

1. Flutter Crash Course (Google)  
   🔗 <https://docs.flutter.dev/get-started/codelab>
2. Dart en Codecademy  
   🔗 <https://www.codecademy.com/learn/learn-dart>
3. Curso Completo de Flutter (YouTube - Fernando Herrera)  
   🔗 <https://www.youtube.com/watch?v=GXIJJkq_H8g&list=PLV6pYUAZ-ZoE6kzN1t9lfV9aYkRFYhwyj>

**📦 Paquetes y bibliotecas**

* Pub.dev (Repositorio Oficial)  
  🔗 [https://pub.dev](https://pub.dev/)
* Flutter Awesome (Inspiración UI)  
  🔗 [https://flutterawesome.com](https://flutterawesome.com/)

**📌 Libros**

1. "Flutter in Action" (Manning)  
   🔗 <https://www.manning.com/books/flutter-in-action>
2. "Dart Apprentice" (Ray Wenderlich)  
   🔗 <https://www.raywenderlich.com/books/dart-apprentice>

**💬 Comunidad**

* Stack Overflow (Flutter Tag)  
  🔗 <https://stackoverflow.com/questions/tagged/flutter>
* Reddit r/FlutterDev  
  🔗 <https://www.reddit.com/r/FlutterDev>
* Flutter Community en Medium  
  🔗 <https://medium.com/flutter-community>

**🛠️ Herramientas Clave**

1. Flutter DevTools (Debugging)  
   🔗 <https://docs.flutter.dev/tools/devtools>
2. Firebase para Flutter  
   🔗 [https://firebase.flutter.dev](https://firebase.flutter.dev/)
3. Riverpod (Gestión de Estado)  
   🔗 [https://riverpod.dev](https://riverpod.dev/)
4. VS Code + Extensión Flutter  
   🔗 <https://marketplace.visualstudio.com/items?itemName=Dart-Code.flutter>

**🌟 Extra: Proyectos Open-Source**

* Repositorio Oficial de Flutter  
  🔗 <https://github.com/flutter/flutter>
* Ejemplos de Apps en GitHub  
  🔗 [https://github.co/flutter/samples](https://github.com/flutter/samples)