TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA ĐIỆN TỬ VIỄN THÔNG

BÁO CÁO

LẬP TRÌNH ĐA NỀN TẢNG

Thực hành gọi API REST sử dụng cả http package và dio package

Sinh viên thực hiện:
01. Võ Văn Tín Lớp: 21KTMT2 MSSV: 106210254
02. Nguyễn Thị Ánh Nhi Lớp: 21KTMT2 MSSV: 106210247

Người hướng dẫn:
TS. Nguyễn Duy Nhật Viễn

Đà Nẵng, 2025

THUYẾT MINH

BÁO CÁO

LẬP TRÌNH ĐA NỀN TẢNG

Thực hành gọi API REST sử dụng cả http package và dio package

BẢNG PHÂN CÔNG CÔNG VIỆC TRONG NHÓM

STT	HỌ VÀ TÊN	NHIỆM VỤ	KHỐI LƯỢNG
01	Võ Văn Tín	- Tạo ứng dụng lấy danh sách bài viết từ JSONPlaceholder API - So sánh http vs dio về tính năng, performance	50%
02	Nguyễn Thị Ánh Nhi	Authentication và Interceptor với Dio Error Handling và Retry Mechanism	50%

Link code github: tindaiz/rest_api_demo
Link code https://github.com/aanhi27/dio_demo_flutter

Mục lục

1. RESTful API

RESTful API [1] là một tiêu chuẩn dùng trong việc thiết kế API cho các ứng dụng web (thiết kế Web services) để tiện cho việc quản lý các resource. Nó chú trọng vào tài nguyên hệ thống (tệp văn bản, ảnh, âm thanh, video, hoặc dữ liệu động…), bao gồm các trạng thái tài nguyên được định dạng và được truyền tải qua HTTP.

Trong Flutter:

- Client: ứng dụng Flutter (mobile, web, desktop).
- Server: máy chủ chứa cơ sở dữ liệu và API.
- Dữ liệu trao đổi: thường ở dạng JSON (JavaScript Object Notation).

Để làm việc với REST API, bạn cần một thư viện để làm việc với các đường link API. Có 2 thư viện chính mà bạn có thể sử dụng đó là : http và dio.

— http là thư viện chính thức của Dart được phát triển bởi Google, dùng để gửi yêu cầu HTTP (GET, POST, PUT, DELETE) tới server.

— dio là một thư viện HTTP mạnh mẽ và linh hoạt được cộng đồng Flutter sử dụng rất rộng rãi. Nó không chỉ gửi và nhận dữ liệu mà còn cung cấp interceptor, logging, timeout, cancel request, và nhiều tính năng nâng cao khác.

1.1 Http package [2]

- Là gói chính thức do nhóm Flutter phát triển.
- Đơn giản, dễ dùng, phù hợp với các ứng dụng nhỏ hoặc nhu cầu cơ bản.
- Hỗ trợ các phương thức cơ bản: GET, POST, PUT, DELETE.
- Không có sẵn tính năng interceptor hay retry.
- Không hỗ trợ download/upload có theo dõi tiến trình.
- Cần thêm các gói khác để xử lý JSON tự động, logging hoặc caching.

1.2 Dio package [3]

- Là thư viện HTTP mạnh mẽ của cộng đồng, hỗ trợ nhiều tính năng nâng cao.
- Có interceptor giúp theo dõi và xử lý yêu cầu/đáp ứng dễ dàng.
- Hỗ trợ retry, timeout, cancel request.
- Hỗ trợ upload/download file có theo dõi tiến trình.

- Hỗ trợ form data, multipart file upload.
- Dễ tích hợp với token, refresh token, và xử lý lỗi toàn cục.

So sánh http và dio:

Tiêu chí	HTTP Package	Dio Package
Đơn giản	Rất dễ dùng, ít cấu hình	Cần cấu hình nhiều hơn nhưng mạnh mẽ
Interceptor	Không có	Có – có thể ghi log, thêm token, retry...
Hiệu suất	Tốt cho ứng dụng nhỏ	Tối ưu hơn cho ứng dụng lớn
Upload/Download	Không hỗ trợ trực tiếp	Có hỗ trợ tiến trình tải lên/xuống
Quản lý lỗi	Thủ công	Có thể xử lý tự động qua interceptor
Timeout/Cancel	Không hỗ trợ cancel	Có hỗ trợ cancel và timeout linh hoạt
Xử lý JSON	Thủ công bằng `json.decode()`	Tự động nếu kiểu trả về là JSON
Caching/Retry	Không có sẵn	Có thể cấu hình retry, caching
Kích thước gói	Nhẹ hơn	Nặng hơn một chút

Nhận xét:
Về tính năng

— http: hướng tới sự đơn giản — thích hợp cho người mới, gọi API nhanh mà không cần cấu hình.
— dio: hướng tới tính mở rộng và tự động hóa — quản lý header, token, interceptor, và log dễ dàng.

Về performance:

Khía cạnh	Giải thích
Tốc độ phản hồi	Cả hai gần tương đương trong GET/POST đơn giản.
Độ ổn định khi tải dữ liệu lớn	dio hoạt động tốt hơn do có cơ chế streaming và timeout tốt hơn.
Sử dụng tài nguyên CPU / RAM	http nhẹ hơn, nhưng dio tối ưu hơn khi nhiều request đồng thời.

Tối ưu mạng (network optimization)	dio có interceptor giúp quản lý cache, retry, và batching hiệu quả hơn.

Về bảo trì và mở rộng:

- http yêu cầu viết thêm nhiều mã khi muốn mở rộng (thêm token, log, retry…).
- dio có Interceptor pipeline ⇒ dễ thêm các chức năng nâng cao mà không đụng vào code chính.

2. Tạo ứng dụng lấy danh sách bài viết từ JSONPlaceholder API

JSONPlaceholder là một API giả lập miễn phí (Fake Online REST API)**,** thường được dùng để thử nghiệm, học và kiểm thử ứng dụng web hoặc mobile mà không cần phải xây dựng server thật.

JSONPlaceholder trong ứng dụng này thử nghiệm gọi API REST (GET, POST, PUT, PATCH, DELETE)**.**

Ứng dụng cần:

- Gọi API từ internet (cụ thể là JSONPlaceholder).
- Nhận và xử lý dữ liệu JSON.
- Hiển thị danh sách bài viết lên giao diện người dùng (UI).

Quy trình hoạt động (theo mô hình Restful):

1. Ứng dụng Flutter gửi HTTP Request (GET) tới endpoint trên.
2. Server (JSONPlaceholder) phản hồi dữ liệu JSON.
3. Ứng dụng phân tích (parse) JSON → danh sách bài viết (`List<Post>`).
4. Dữ liệu được hiển thị trên UI (ListView, ListTile, Card, v.v.).

Luồng xử lý:

UI (Widget)
↓
API Service
↓
HTTP Request (GET, POST, PUT, PATCH, DELETE)
↓
JSON Data từ Server (Response)
↓
Parse sang Model (Post)

↓

Cập nhật UI (ListView / Dialog / Snackbar / setState)

Chi tiết:

```
lib/
│
├── main.dart
│
├── models/
│   └── post_model.dart
│
├── services/
│   ├── api_http_service.dart
│   └── api_dio_service.dart
│
├── screens/
│   ├── post_list_http.dart
│   ├── post_list_dio.dart
│   └── post_form_screen.dart
│   └── home_screen.dart
│
└── widgets/
    └── post_card.dart
```

1. `models/`

Ánh xạ dữ liệu JSON → đối tượng Dart.

- `class Post { ... }` — định nghĩa trường `id`, `title`, `body`.
- `factory Post.fromJson(Map<String,dynamic> json)` — chuyển JSON sang model.
- `Map<String,dynamic> toJson()` — chuyển model sang JSON khi gửi POST/PUT.
- `copyWith(...)` — hỗ trợ cập nhật immutable object.

2. `services/`

2.1 `api_http_service.dart`

Xử lý API bằng http

- `getPosts()` — dùng `http.get` để GET /posts và `jsonDecode` → `List<Post>`.

- `createPost(Post post)` — dùng `http.post`, header `Content-Type: application/json`, body từ `post.toJson()` → POST /posts.
- `updatePost(Post post)` — `http.put('/posts/{id}')` → PUT.
- `deletePost(int id)` — `http.delete('/posts/{id}')` → DELETE.

2.2 `api_dio_service.dart`

Xử lý API bằng DIO + interceptor

- `Dio(BaseOptions(baseUrl: ...))` — cấu hình global.
- `dio.interceptors.add(InterceptorsWrapper(...))` — Interceptor: log request/response/error.
- Các hàm `getPosts()`, `createPost()`, `updatePost()`, `deletePost()` tương tự nhưng trả `response.data` trực tiếp.

3. `widgets/`
Tái sử dụng UI nhỏ, tách biệt hiển thị 1 item.

- `PostCard` (StatelessWidget) — hiển thị `CircleAvatar` (id), `title`, `body` (ellipsis).
- `onTap` callback để điều hướng sang form (edit).

4. `screens/`
4.1 `home_screen.dart` dùng `BottomNavigationBar` để chuyển giữa 2 tab: HTTP và DIO
4.2 `post_list_http.dart` / `post_list_dio.dart`

Hiển thị danh sách và tìm kiếm

- `initState()` gọi `_load()` → dùng service tương ứng để lấy dữ liệu (`getPosts()`), lưu vào `_posts`.
- `RefreshIndicator` + `onRefresh: _load` → kéo để reload.
- `TextField` search → cập nhật `_filtered` bằng `.where` trên `title`.
- `ListView.builder` hiển thị `PostCard`.
- `FloatingActionButton` mở `PostFormScreen` để thêm (POST).
- Khi pop từ `PostFormScreen`, xử lý `result['action']` để update local list (insert/update/delete).

4.3 `post_form_screen.dart`

Thêm, sửa, xóa bài viết

- Form với `TextFormField` cho `title` và `body`, `GlobalKey<FormState>` để validate.
- Nếu có `postToEdit` → chế độ **edit** (PUT + nút Delete), ngược lại **create** (POST).

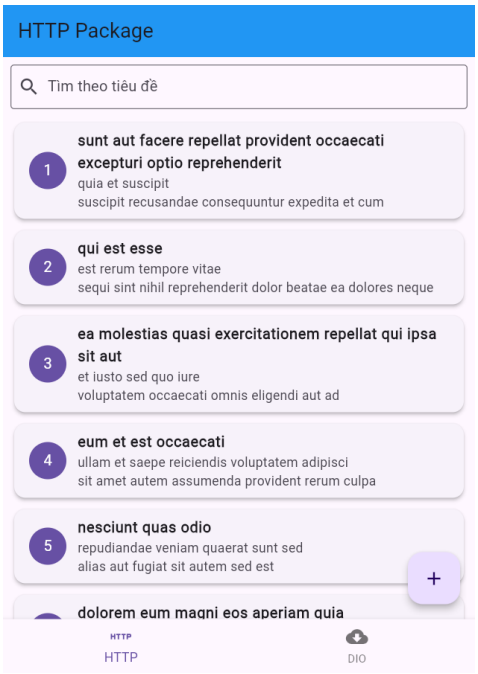
- Gọi `_http.createPost` / `_dio.createPost` hoặc `_http.updatePost` / `_dio.updatePost`.
- Xác nhận xóa (`AlertDialog`) → gọi `deletePost(id)` → `Navigator.pop({'action':'deleted', 'id': id})`.
- Trả về `Map` chứa `action` và `post` để màn trước xử lý.

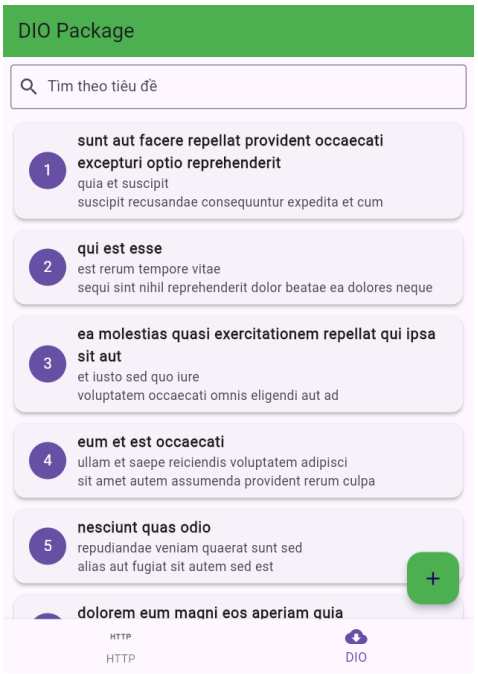
5. `main.dart`
Entry point; cấu hình theme, khởi tạo `HomeScreen`.

JSONPlaceholder mô phỏng CREATE/UPDATE/DELETE: server trả về kết quả hợp lệ nhưng không lưu thay đổi trên backend. Dùng để test client-side logic.

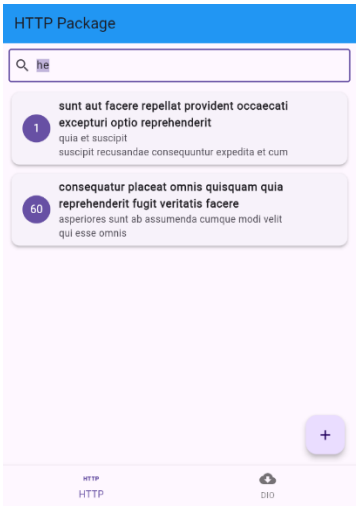
Kết quả:

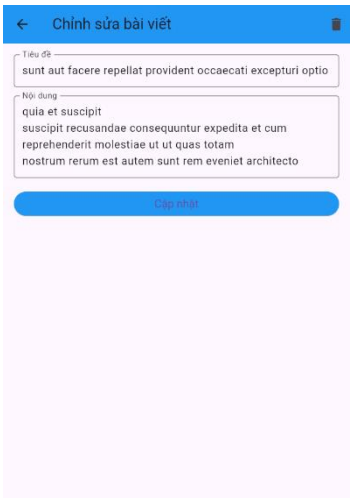
Màn hình hiển thị bài viết và tìm kiếm



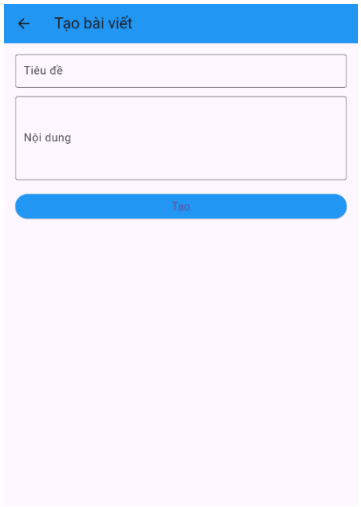


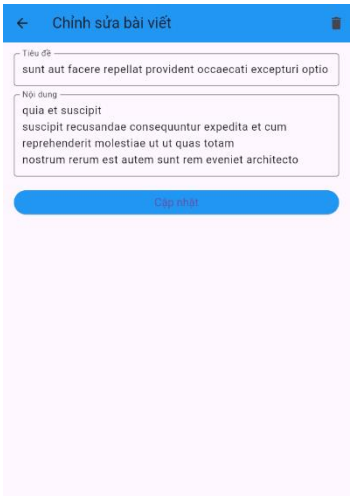
Tìm kiếm và truy cập bài viết:

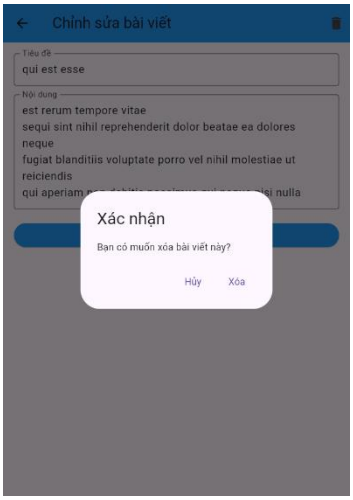




Thêm, sửa, xoá bài viết:







3. Authentication và Interceptor với Dio

3.1 Khái niệm

Trong các ứng dụng di động hiện đại, đặc biệt là những ứng dụng có kết nối với backend, xác thực người dùng (authentication) là bước khởi đầu quan trọng nhằm đảm bảo chỉ những người có quyền hợp lệ mới được truy cập dữ liệu.
Nó diễn ra khi người dùng nhập tên đăng nhập (username) và mật khẩu (password), gửi lên server để kiểm tra.

Nếu hợp lệ, server sẽ tạo ra một mã định danh duy nhất gọi là Access Token — thường là một chuỗi JWT. Token này chứa các thông tin cơ bản về người dùng và có thời hạn hiệu lực. Khi người dùng gửi các yêu cầu sau đó, ứng dụng chỉ cần gửi token này kèm theo để chứng minh danh tính mà không cần nhập lại mật khẩu.

3.2 Cơ chế JWT Authentication

JWT (JSON Web Token) là một chuẩn mã hóa giúp xác thực người dùng giữa client và server. Một JWT có 3 phần, được phân tách bằng dấu . như sau:

header.payload.signature

Header: xác định thuật toán ký (thường là HS256) và loại token.
Payload: chứa dữ liệu của người dùng như id, username, exp (thời điểm hết hạn).
Signature: là chữ ký được tạo từ secret key của server, đảm bảo token không bị giả mạo.

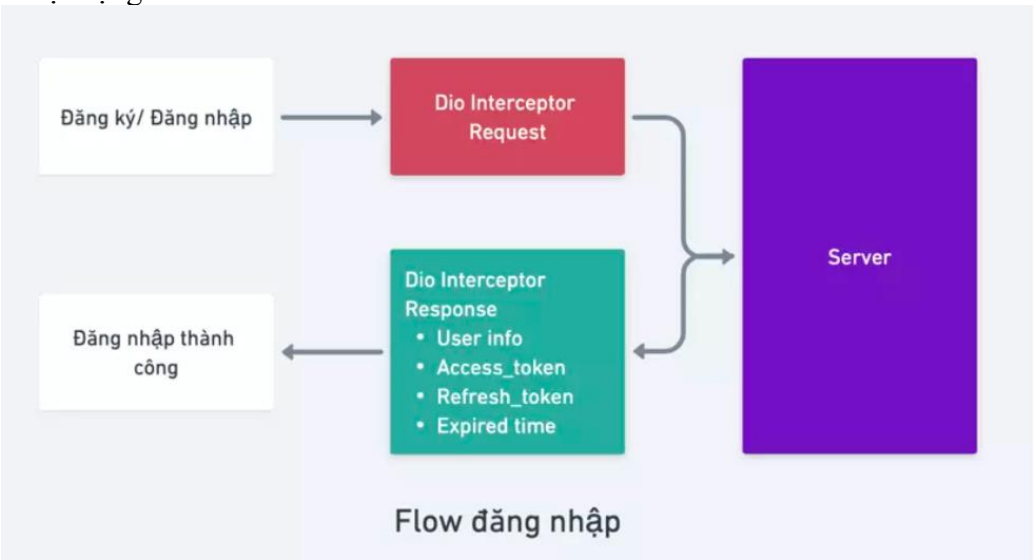
Khi người dùng đăng nhập thành công:

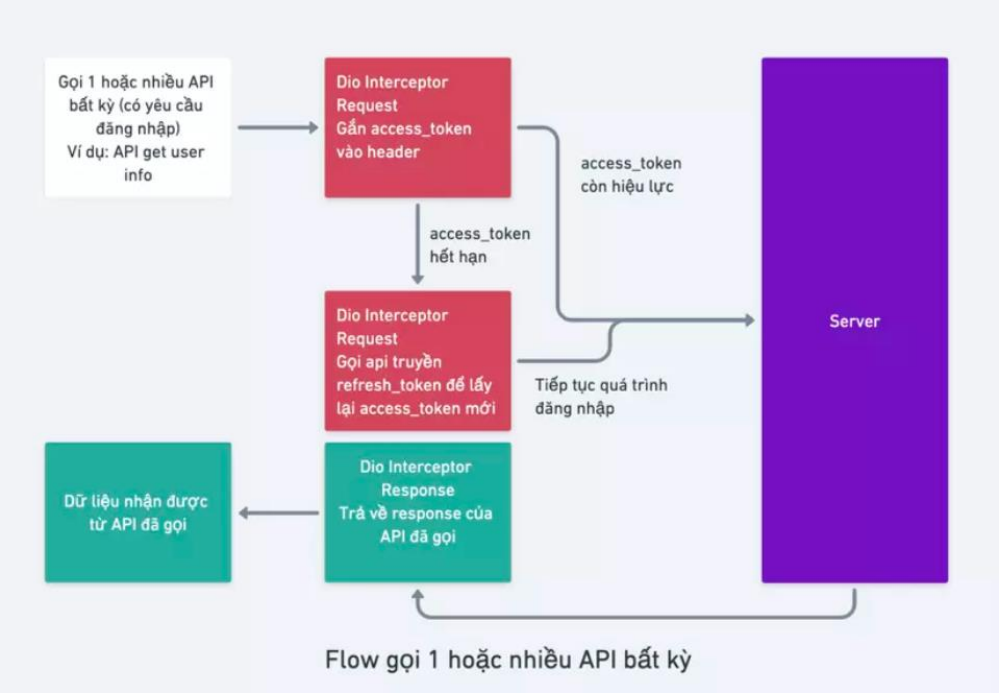
- access_token: Định danh user nào đăng nhập. Thông thường access_token sẽ tồn tại với thời gian khoảng ngắn
- refresh_token: Token dùng để lấy lại access_token mới khi access_token cũ hết hạn. Thời hạn tồn tại của refresh_token sẽ dài hơn access_token

- Client lưu cả hai token này để sử dụng.

Vậy, để duy trình đăng nhập thì cần phải có cơ chế refresh token hay Lấy lại access_token mới để duy trì đăng nhập (tiếp tục call những API có yêu cầu access_token).

3.3 Cách hoạt động





Toàn bộ quy trình xác thực diễn ra như sau:

Người dùng nhập thông tin đăng nhập.
Flutter gửi request POST /login đến server Node.js.

Server kiểm tra tài khoản:
Nếu đúng → gửi về accessToken và refreshToken.
Nếu sai → trả lỗi 401 Unauthorized.
Flutter lưu 2 token này trong bộ nhớ an toàn (Storage).

Mỗi lần gọi API, Flutter sẽ tự động:
Lấy token từ bộ nhớ.
Gắn token vào header Authorization: Bearer <accessToken>.
Khi accessToken hết hạn, nếu server trả về 401, ứng dụng sẽ:
Gửi POST /refresh kèm refreshToken.
Nhận accessToken mới và lưu lại.
Gửi lại request ban đầu mà người dùng không hề nhận thấy lỗi.

Interceptor có thể hiểu như một bước tường lưới chặn các request, response của ứng dụng để cho phép kiểm tra, thêm vào header hoặc thay đổi các param của request, response. Nó cho phép chúng ta kiểm tra các token ứng dụng, Content-Type hoặc tự thêm các header vào request.

Các thành phần chính của Dio Interceptor:

onRequest(RequestOptions options): dùng để handle request trước khi gửi cho server.
onResponse(Response response): dùng để handle reponse trước khi gửi cho client.
onError(DioError error): handle error trước khi gửi cho client.

3.4 Demo

- cấu trúc server (Node js)

```
const express = require('express');
const jwt = require('jsonwebtoken');
const bodyParser = require('body-parser');
const cors = require('cors');

const app = express();
app.use(cors());
app.use(bodyParser.json());

const ACCESS_TOKEN_SECRET = 'sieu_bi_mat_access';
const REFRESH_TOKEN_SECRET = 'sieu_bi_mat_refresh';
const ACCESS_TOKEN_EXPIRY = '30s';

let validRefreshTokens = [];
let posts = [
  { id: 1, title: 'Bài viết 1' },
  { id: 2, title: 'Bài viết 2' }
];

function generateAccessToken(user) {
  return jwt.sign(user, ACCESS_TOKEN_SECRET, { expiresIn: ACCESS_TOKEN_EXPIRY
});
}

// LOGIN
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  if (username !== 'flutter' || password !== '123') {
    return res.status(400).json({ message: 'Sai username hoặc password' });
  }
  const user = { name: 'Anh Nhi' };
  const accessToken = generateAccessToken(user);
  const refreshToken = jwt.sign(user, REFRESH_TOKEN_SECRET);
  validRefreshTokens.push(refreshToken);
  res.json({ accessToken, refreshToken });
});

// MIDDLEWARE XÁC THỰC
function verifyToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];
  if (!token) {
```

```
    console.log('[AUTH] Không có token trong request');
    return res.sendStatus(401);
  }

  jwt.verify(token, ACCESS_TOKEN_SECRET, (err, user) => {
    if (err) {
      if (err.name === 'TokenExpiredError') {
        console.log('[AUTH] Token hết hạn!');
        return res.status(401).json({ message: 'Token hết hạn' });
      }
      console.log('[AUTH] Token không hợp lệ!');
      return res.status(403).json({ message: 'Token không hợp lệ' });
    }
    req.user = user;
    next();
  });
}

// REFRESH TOKEN
app.post('/refresh', (req, res) => {
  const { refreshToken } = req.body;
  if (!refreshToken) {
    console.log('[REFRESH] Không có refresh token!');
    return res.sendStatus(401);
  }
  if (!validRefreshTokens.includes(refreshToken)) {
    console.log('[REFRESH] Refresh token không hợp lệ!');
    return res.status(403).json({ message: 'Refresh token không hợp lệ' });
  }
  jwt.verify(refreshToken, REFRESH_TOKEN_SECRET, (err, user) => {
    if (err) {
      console.log('[REFRESH] Refresh token bị sai!');
      return res.status(403).json({ message: 'Refresh token bị sai' });
    }
    const newAccessToken = generateAccessToken({ name: user.name });
    console.log('[REFRESH] Cấp accessToken mới:', newAccessToken);
    res.json({ accessToken: newAccessToken });
  });
});
```

- Xây dựng giao diện đăng nhập trong Flutter

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text("Login")),
    body: Padding(
      padding: const EdgeInsets.all(16),
      child: Column(
        children: [
          TextField(
            controller: emailController,
            decoration: InputDecoration(labelText: "Email"),
          ), // TextField
          TextField(
            controller: passController,
            decoration: InputDecoration(labelText: "Password"),
            obscureText: true,
          ), // TextField
          const SizedBox(height: 20),
          if (loading) CircularProgressIndicator(),
          ElevatedButton(onPressed: login, child: Text("Login")),
        ],
```

Trong ứng dụng Flutter, phần đăng nhập được thực hiện trong file login_page.dart.
Người dùng nhập tài khoản và mật khẩu, ứng dụng gọi API /login qua thư viện **Dio**.

```
oid login() async {
   setState(() => loading = true);

   try {
     // Chỉ dùng path '/login', baseUrl sẽ được Dio xử lý
     final response = await widget.dio.post('/login', data: {
       'username': emailController.text,
       'password': passController.text,
     });
```

Nếu đăng nhập thành công, server sẽ trả về 2 token:

```
     // Lưu token vào storage
     await widget.storage.save('accessToken', response.data['accessToken']);
     await widget.storage.save('refreshToken', response.data['refreshToken']);
```

Sau đó, điều hướng sang HomePage
Nếu có lỗi (ví dụ sai mật khẩu hoặc server không phản hồi), ứng dụng sẽ hiển thị thông báo qua SnackBar

- Tự động thêm Token bằng Interceptor

Thay vì phải thêm token thủ công cho mỗi request, Dio cung cấp Interceptor — đây là lớp trung gian cho phép can thiệp vào trước khi gửi request, sau khi nhận response, hoặc khi có lỗi.

```dart
class AuthInterceptor extends Interceptor {
  final Dio dio;
  final Storage storage;
  AuthInterceptor({required this.dio, required this.storage});

  @override
  void onRequest(RequestOptions options, RequestInterceptorHandler handler) async {
    final token = await storage.get('accessToken');
    if (token != null) options.headers['Authorization'] = 'Bearer $token';
    handler.next(options);
  }
```

Giải thích:

- Trước khi gửi bất kỳ request nào, interceptor sẽ lấy accessToken từ storage.
- Nếu tồn tại token, nó sẽ được thêm vào phần headers của request.
- Nhờ đó, mọi API đều được xác thực tự động mà không cần viết lại nhiều lần.

Tự động làm mới (Refresh Token) khi token hết hạn

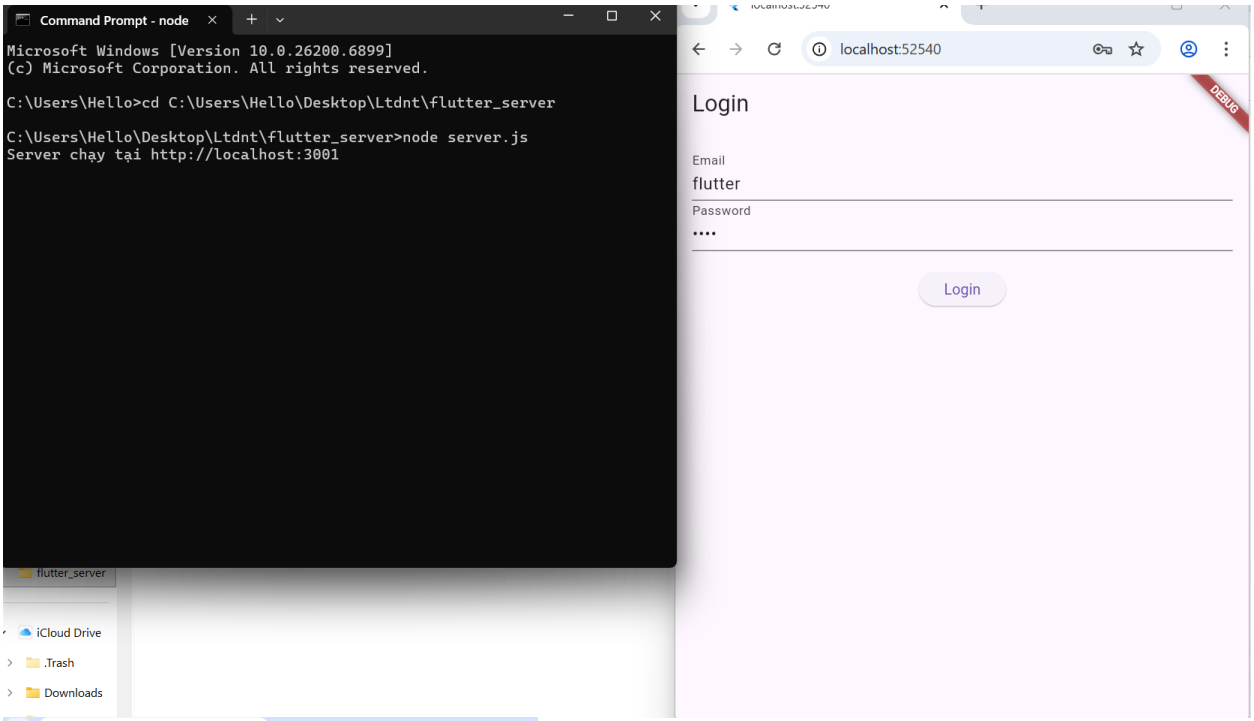
Khi accessToken hết hạn, server sẽ trả về mã lỗi 401 Unauthorized.
Interceptor sẽ phát hiện lỗi này trong phương thức onError

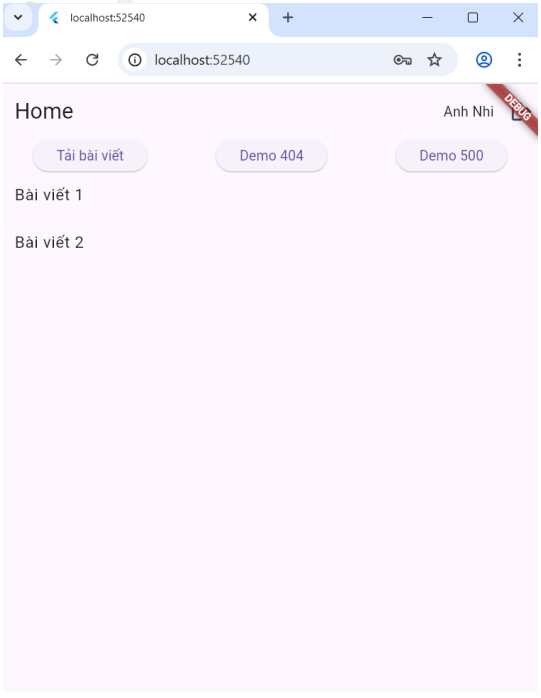
```dart
class AuthInterceptor extends Interceptor {
  @override
  void onError(DioException err, ErrorInterceptorHandler handler) async {
    if (err.response?.statusCode == 401) {
      final refreshToken = await storage.get('refreshToken');
      if (refreshToken == null) return handler.reject(err);

      try {
        final response = await dio.post('/refresh', data: {'refreshToken': refreshToken});
        final newAccessToken = response.data['accessToken'];
        await storage.save('accessToken', newAccessToken);

        err.requestOptions.headers['Authorization'] = 'Bearer $newAccessToken';
        final cloneReq = await dio.fetch(err.requestOptions);
        return handler.resolve(cloneReq);
      } catch (e) {
        await storage.clearAll();
        return handler.reject(err);
      }
    }
    handler.next(err);
  }
```

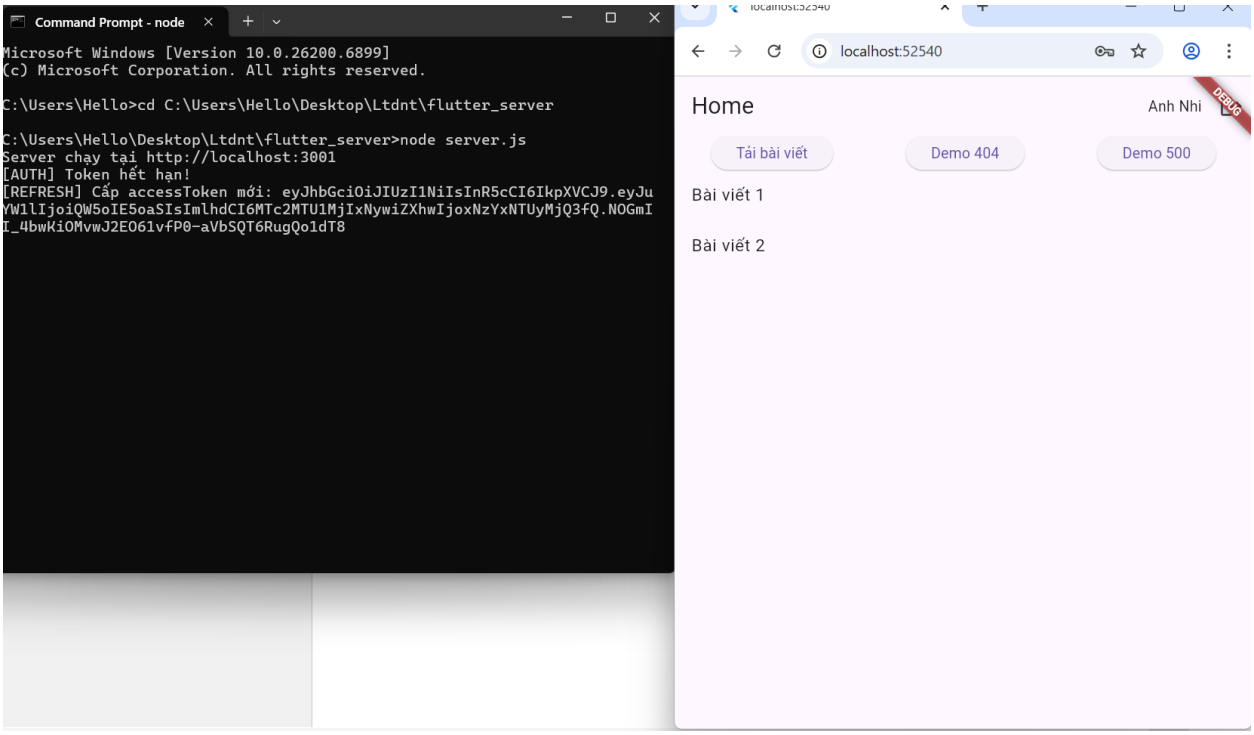
Khi nhận lỗi 401, Interceptor kiểm tra có refreshToken hay không.
Nếu có, gửi request /refresh lên server.
Server trả về token mới → lưu lại vào storage.
Interceptor tự động thực hiện lại request ban đầu với token mới.

Người dùng không cần đăng nhập lại — trải nghiệm mượt mà.

Nếu quá trình refresh thất bại (do refreshToken hết hạn hoặc bị thu hồi), Interceptor sẽ xóa toàn bộ dữ liệu lưu trữ (storage.clearAll()) để buộc người dùng đăng nhập lại.







4. Error Handling và Retry Mechanism
4.1 Tổng quan về Error Handling trong Dio

Khi ứng dụng giao tiếp với server qua HTTP, lỗi là điều không thể tránh khỏi — có thể đến từ mạng, token hết hạn, API không tồn tại, hoặc server bị lỗi nội bộ.

Thư viện Dio hỗ trợ cơ chế mạnh mẽ để phát hiện, phân loại và xử lý lỗi thông qua:

try-catch thông thường với DioException

Interceptor (onError) để xử lý lỗi tập trung, ví dụ như tự động refresh token

Hiển thị lỗi lên UI bằng SnackBar hoặc dialog để người dùng biết

Các loại lỗi trong Dio

connectionTimeout Hết thời gian chờ khi kết nối đến server
sendTimeout Gửi request quá lâu
receiveTimeout Chờ phản hồi quá lâu
badResponse Server trả về lỗi HTTP (4xx, 5xx)
cancel Request bị huỷ
unknown Lỗi không xác định (mất mạng, sai domain, v.v.)

4.2 Cách xử lý lỗi bằng try-catch

```dart
Future<List<Map<String, dynamic>>> fetchPosts() async {
  try {
    final response = await dio.get('/posts');
    return List<Map<String, dynamic>>.from(response.data);
  } on DioException catch (e) {
    if (e.response != null) {
      print('⚠️ Lỗi fetchPosts: ${e.response!.statusCode} - ${e.response!.data}');
      if (e.response!.statusCode == 401) throw Exception('Chưa đăng nhập hoặc token hết hạn');
      if (e.response!.statusCode == 404) throw Exception('Không tìm thấy bài viết');
    }
    throw Exception('Lỗi máy chủ');
  }
}
```

on DioException catch (e): Bắt lỗi từ Dio (thay vì lỗi hệ thống)
e.response?.statusCode: Mã lỗi HTTP (401, 404, 500, …)
throw Exception(): Ném lỗi lên trên để UI xử lý

Ở UI (HomePage), bạn bắt lại lỗi và hiển thị:

ScaffoldMessenger.of(context).showSnackBar(
  SnackBar(content: Text('⚠ $e')),

4.3 Cơ chế Retry (Thử lại Request)

- Khi nào cần Retry?
Có 2 trường hợp phổ biến:

- Lỗi mạng tạm thời (timeout, mất Wi-Fi ngắn)
- Token hết hạn → refresh thành công → cần gửi lại request cũ
-

- Retry bằng cách thủ công (trong Interceptor)

Trong code AuthInterceptor, cơ chế retry tự động khi refresh token:
err.requestOptions.headers['Authorization'] = 'Bearer $newAccessToken';
final cloneReq = await dio.fetch(err.requestOptions);
return handler.resolve(cloneReq);
- Dòng dio.fetch() chính là retry lại request ban đầu sau khi có token mới.

Retry nâng cao (ví dụ lỗi mạng)
Bạn có thể tự viết hàm **retry có giới hạn số lần thử**.

Future<Response<dynamic>> _retryRequest(
  Future<Response<dynamic>> Function() request, {
  int retries = 3,
  Duration delay = const Duration(seconds: 2),
}) async {

```
  DioException? lastError;

  for (int attempt = 0; attempt < retries; attempt++) {
    try {
      print('🔁 Thử gọi API (lần ${attempt + 1}/$retries)');
      final resp = await request();
      print('✅ Thành công ở lần ${attempt + 1}');
      return resp;
    } on DioException catch (e) {
      lastError = e;

      // Nhận diện lỗi có thể retry
      final message = e.message?.toLowerCase() ?? '';
      final errorStr = e.error?.toString().toLowerCase() ?? '';

      final retryable = e.type == DioExceptionType.connectionTimeout ||
          e.type == DioExceptionType.sendTimeout ||
          e.type == DioExceptionType.receiveTimeout ||
          e.type == DioExceptionType.badCertificate ||
          e.type == DioExceptionType.unknown ||
          message.contains('xmlhttprequest onerror') || // Flutter Web
          message.contains('network error') ||
          errorStr.contains('socketexception') ||
          errorStr.contains('connection refused');

      if (retryable && attempt < retries - 1) {
        print('⚠️ Lỗi mạng hoặc XMLHttpRequest, chờ ${delay.inSeconds}s rồi thử lại...');
        await Future.delayed(delay);
        continue;
      }

      print('❌ Thử lại thất bại: ${e.message}');
      rethrow;
    } catch (e) {
      print('❌ Lỗi không phải DioException: $e');
      rethrow;
    }
  }

  throw lastError ?? Exception('Unknown error during retry');
}
```

Áp dụng retry cho API thật

Giờ, thay vì gọi dio.get('/posts') trực tiếp, ta bao quanh bằng_retryRequest:

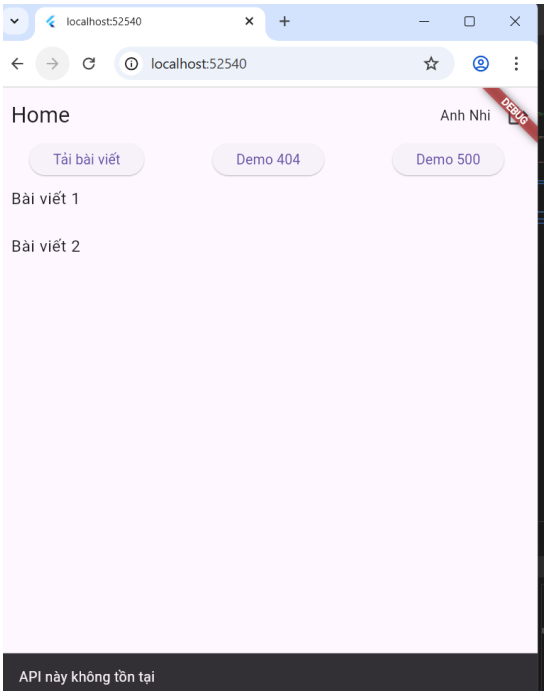
```
Future<List<Map<String, dynamic>>> fetchPosts() async {
```

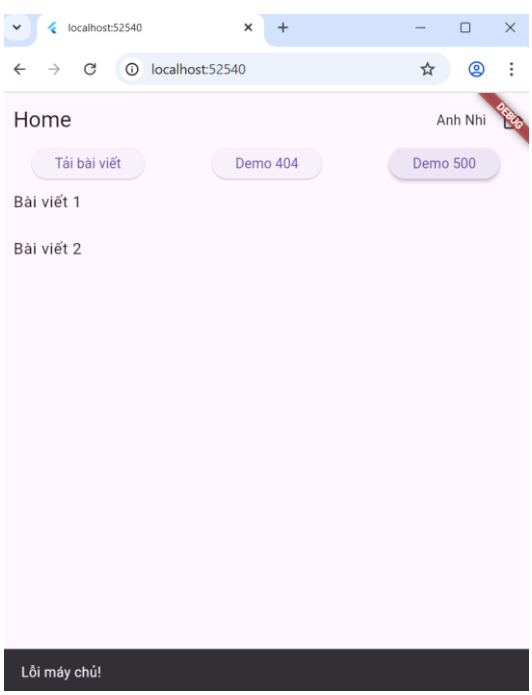
```
  try {
    final response = await _retryRequest(() => dio.get('/posts'));
    return List<Map<String, dynamic>>.from(response.data);
  } on DioException catch (e) {
    if (e.response != null) {
      if (e.response!.statusCode == 401) throw Exception('Chưa đăng nhập hoặc token hết hạn');
      if (e.response!.statusCode == 404) throw Exception('Không tìm thấy bài viết');
    }
    throw Exception('Lỗi máy chủ hoặc mạng');
  }
}
```

4.4 Demo

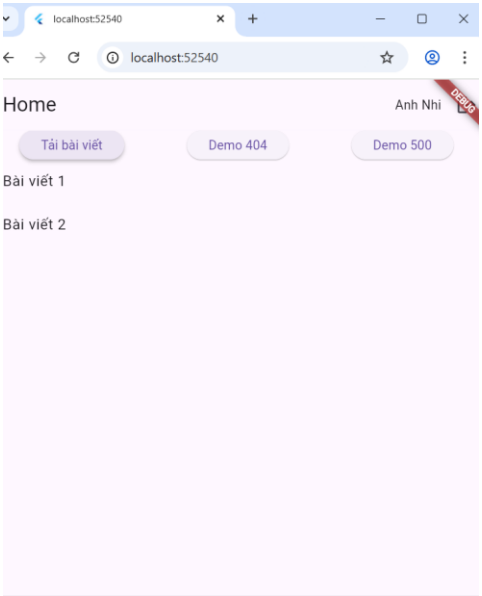
Trong HomePage, khi bấm **Demo 404** hoặc **Demo 500**, ta mô phỏng lỗi từ server:

```dart
void demo404() async {
  try {
    await repo.callNotFound();
  } on DioException catch (e) {
    showDioError(context, e);
  } catch (e) {
    ScaffoldMessenger.of(context)
        .showSnackBar(SnackBar(content: Text('Lỗi không xác định')));
  }
}

void demo500() async {
  try {
    await repo.callServerError();
  } on DioException catch (e) {
    showDioError(context, e);
  } catch (e) {
    ScaffoldMessenger.of(context)
        .showSnackBar(SnackBar(content: Text('Lỗi không xác định')));
  }
```





retry




```
🔁 Thử gọi API (lần 1/3)
! Lỗi mạng hoặc XMLHttpRequest, chờ 2s rồi thử lại...
🔁 Thử gọi API (lần 2/3)
! Lỗi mạng hoặc XMLHttpRequest, chờ 2s rồi thử lại...
🔁 Thử gọi API (lần 3/3)
❌ Thử lại thất bại: The connection errored: The XMLHttpRequest onError callback was called.
e
network layer. This indicates an error which most likely cannot be solved by the library.
```

TÀI LIỆU THAM KHẢO

[1]. TÌM HIỂU VỀ API REST TRONG FLUTTER
[2]. http | Dart package
[3]. dio | Dart package
[4] Refresh JWT Token Interceptor in Flutter,
https://www.dev-influence.com/article/refresh-jwt-token-interceptor-in-flutter
[5] Error Handling with Dio in Flutter Apps
https://tillitsdone.com/blogs/error-handling-with-dio-flutter/