**ĐẠI HỌC ĐÀ NẴNG**

**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**KHOA ĐIỆN TỬ - VIỄN THÔNG**

**BÁO CÁO**

**VIẾT VÀ THUYẾT TRÌNH KỸ THUẬT**

**ĐỀ TÀI:**

**JSON Serialization và Model Classes**

Sinh viên thực hiện: **Nguyễn Trọng Nhân – 22KTMT1 – 106220229**

**Lê Duy Hoàng – 22KTMT2 - 106220252**

**Đà Nẵng, 10/2025**

BẢNG PHÂN CÔNG CÔNG VIỆC TRONG NHÓM

|  |  |  |  |
| --- | --- | --- | --- |
| STT | HỌ VÀ TÊN | NHIỆM VỤ | KHỐI LƯỢNG |
| 01 | NGUYỄN TRỌNG NHÂN | Tạo model classes với toJson() và fromJson()  Xử lý nested objects và arrays | 50% |
| 02 | LÊ DUY HOÀNG | Sử dụng json\_annotation và build\_runner  Best practices cho complex data structures | 50% |

**Link github: git@github.com:miyamura0309/Nhan\_Hoang.git**

**MỤC LỤC**

[**1. Giới thiệu:** 3](#_Toc212473764)

[**2. Manual serialization:** 4](#_Toc212473765)

[**2.1. Chuyển hóa JSON trực tiếp:** Bằng cách sử dụng dart:convert, chúng ra có thể giải mã và mã hóa JSON bằng cách gọi hàm jsonDecode() và jsonEncode(), với chuỗi JSON là tham số. Cụ thể: 5](#_Toc212473766)

[**2.2. Chuyển hóa JSON bên trong các lớp mô hình:** 6](#_Toc212473767)

[**2.3. Làm việc với JSON phức tạp:** 7](#_Toc212473768)

[**3. Phương pháp tự động sinh mã(Automated serialization using code generation)** 10](#_Toc212473769)

[3.1 Giới thiệu 10](#_Toc212473770)

[3.2 Thực hành 11](#_Toc212473771)

[**4. Các phương pháp tốt nhất khi làm việc với cấu trúc dữ liệu phức tạp** 15](#_Toc212473772)

# **1. Giới thiệu:**

JSON là viết tắt của JavaScript Object Notation, là một định dạng dữ liệu dạng văn bản dùng để lưu trữ và trao đổi dữ liệu giữa các hệ thống mà hầu hết các ngôn ngữ lập trình hiện nay đều có thể đọc được. Nhờ tính đơn giản, dễ đọc và khả năng biểu diễn cấu trúc dữ liệu phức tạp, JSON được ưa chuộng rộng rãi, đặc biệt khi làm việc với các API.

Cấu trúc cơ bản của JSON: gồm 2 kiểu cấu trúc chính:

* Object (Đối tượng): { key:value, key:value, key:value, … }, tập hợp các cặp key-value.
* Array (Mảng): { key:[ value1, value2, … ]}, danh sách thứ tự các giá trị, cách nhau bởi dấu phẩy.

Ví dụ một đoạn JSON đơn giản:



Cụ thể đối tượng biểu diễn thông tin cá nhân của một người, gồm 2 cặp key-value gồm name và gmail cùng với một mảng address. Khi xây dựng các ứng dụng Flutter, đặc biệt là khi làm việc với các API bên ngoài hoặc giao tiếp với máy chủ, việc xử lý dữ liệu JSON , chuyển đổi chúng thành các đối tượng Dart và ngược lại là điều cần thiết. Quá trình này — JSON serialization — là một phần quan trọng trong việc xây dựng các ứng dụng Flutter hiệu quả và có khả năng mở rộng, giúp code an toàn, dễ đọc, dễ bảo trì. Cụ thể:

* Encoding (serialization): chuyển đối tượng hoặc cấu trúc dữ liệu trong Dart thành chuỗi JSON để lưu trữ hoặc gửi đi.
* Decoding (deserialization): chuyển chuỗi JSON nhận được thành đối tượng Dart để xử lý trong ứng dụng.

Bài báo cáo này sẽ làm rõ các cách tiếp cận khác nhau để chuyển đổi JSON, từ các phương pháp thủ công (manual serializaion) đến tạo mã tự động (automation serialization), và giúp tìm ra phương pháp nào phù hợp nhất cho quá trình phát triển ứng dụng Flutter của mình.

# **2. Manual serialization:**

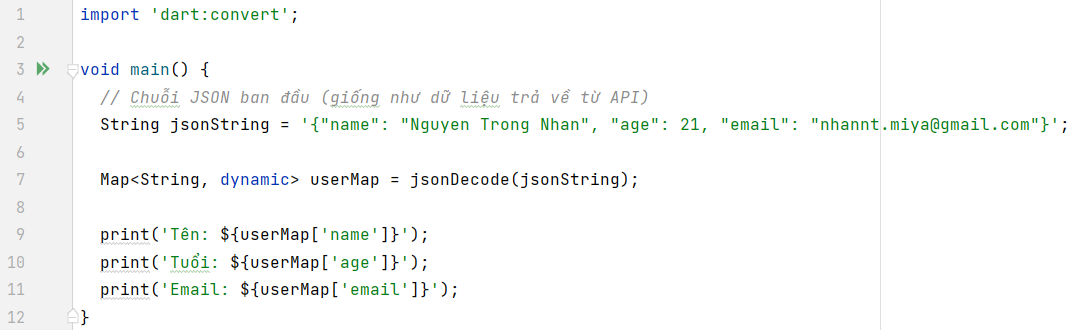
Manual serialization (Chuyển đổi thủ công) được hỗ trợ bởi Flutter rất đơn giản với thư viện dart:convert, bao gồm trình mã hóa và giải mã trực quan. Với dữ liệu mẫu JSON ban đầu, chúng ta có thể chuyển hóa thủ công theo 2 cách: Chuyển hóa JSON trực tiếp và Chuyển hóa JSON bên trong các lớp mô hình.

## **2.1. Chuyển hóa JSON trực tiếp:** Bằng cách sử dụng dart:convert, chúng ra có thể giải mã và mã hóa JSON bằng cách gọi hàm jsonDecode() và jsonEncode(), với chuỗi JSON là tham số. Cụ thể:

* jsonDecode(): chuyển chuỗi JSON (String) → Map<String, dynamic> hoặc List<dynamic>
* jsonEncode(): chuyển Map/List/Object → chuỗi JSON (String)

Ví dụ minh họa:

Giải mã JSON (Decode):



Kết quả chạy chương trinh minh họa:



Mã hóa JSON (Encode)



Kết quả chạy chương trình minh họa:



## **2.2. Chuyển hóa JSON bên trong các lớp mô hình:**

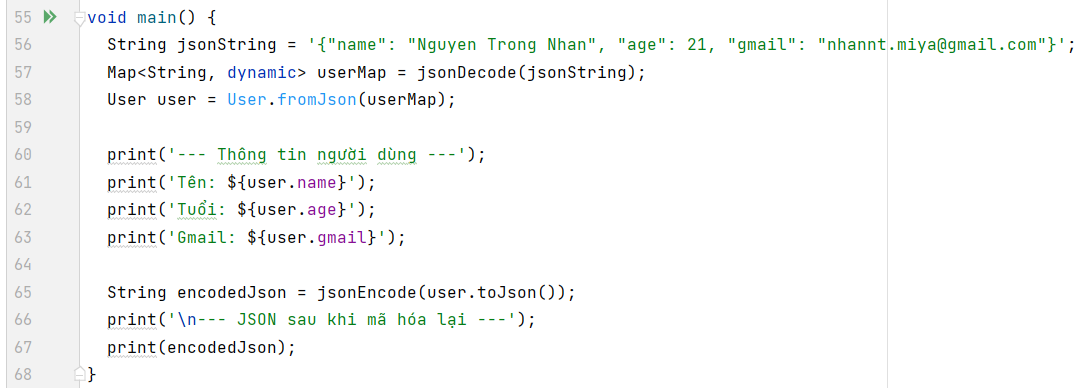
Tuy nhiên, jsonDecode() trả về kiểu dynamic, nghĩa là không biết kiểu dữ liệu của các giá trị cho đến khi chạy. Cách tiếp cận này khiến mất hầu hết các tính năng của ngôn ngữ kiểu tĩnh như kiểm tra kiểu hay các lỗi biên dịch, do đó, mã sẽ trở nên dễ lỗi hơn. Ví dụ, bất cứ khi nào truy cập các trường name hoặc email, có thể dễ dàng gõ sai — lỗi mà trình biên dịch không phát hiện được vì JSON chỉ nằm trong cấu trúc map. Do đó, chúng ta sử dụng phương pháp chuyển hóa JSON bên trong các lớp mô hình. Trong lớp này có:

* **Hàm tạo** fromJson() được dùng để **tạo một đối tượng mới từ một cấu trúc dạng Map** (thường là dữ liệu đã được giải mã từ JSON).
* **Phương thức** toJson() được dùng để **chuyển đổi một đối tượng User thành một Map**, chuẩn bị cho việc **mã hóa sang JSON.**

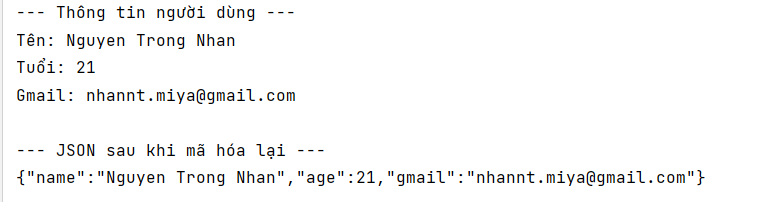
Để mã hóa người dùng, chỉ cần truyền đối tượng User vào hàm jsonEncode().

Ví dụ minh họa:





Kết quả chạy chương trình minh họa:



## **2.3. Làm việc với JSON phức tạp:**

Trong lĩnh vực phát triển ứng dụng di động, việc tương tác với các dịch vụ web và API là một nhiệm vụ phổ biến. Các dịch vụ này thường trả về dữ liệu ở định dạng JSON, vốn nhẹ và dễ phân tích cú pháp. Tuy nhiên, khi xử lý các cấu trúc JSON phức tạp, chẳng hạn như các đối tượng và mảng được lồng sâu, việc phân tích có thể trở nên khó khăn. Phần này sẽ hướng dẫn bạn cách xử lý dữ liệu JSON phức tạp trong Flutter bằng Dart, tập trung vào các chiến lược giúp duy trì tính dễ đọc và hiệu quả của chương trình.

Một đối tượng JSON thông thường có thể trông đơn giản, nhưng trong các ứng dụng thực tế, bạn thường phải xử lý các cấu trúc phức tạp hơn. Những cấu trúc này có thể bao gồm **các đối tượng lồng nhau (nested objects), mảng các đối tượng (arrays of objects),** hoặc sự kết hợp của cả hai.

Ví dụ về cấu trúc JSON phức tạp: Xét ví dụ về cấu trúc JSON sau, biểu diễn hồ sơ người dùng với thông tin địa chỉ và liên hệ được lồng bên trong:



JSON này bao gồm đối tượng lồng nhau (address) và một mảng các đối tượng (contacts). Việc phân tích những cấu trúc như vậy đòi hỏi một cách tiếp cận có hệ thống. Để phân tích dữ liệu JSON phức tạp, chúng ta có các cách thức xử lý sau:

* Tạo các lớp Dart có cấu trúc lồng nhau: Đối với mỗi cấu trúc lồng nhau trong JSON, hãy tạo một lớp Dart tương ứng. Cách tiếp cận này không chỉ giúp tổ chức mã hợp lý mà còn giúp việc quản lý và mở rộng trở nên dễ dàng hơn.
* Sử dụng factory constructor để giải mã phức tạp: factory constructor là một loại **hàm khởi tạo đặc biệt** trong Dart, được dùng khi bạn **không muốn luôn tạo ra một đối tượng mới** mỗi lần khởi tạo, mà có thể **trả về một đối tượng đã tồn tại, một kiểu con (subclass),** hoặc **xử lý logic trước khi tạo đối tượng,** đặc biệt hữu ích khi chuyển đổi dữ liệu JSON thành đối tượng Dart. Bằng cách sử dụng factory constructor, bạn có thể đóng gói logic phân tích JSON bên trong lớp, giúp mã gọn gàng và dễ bảo trì hơn.
* Ví dụ minh họa: Với dữ liệu JSON phức tạp mẫu như trên, với mỗi cấu trúc lồng nhau đã tạo một lớp Dart tương ứng và trong lớp User, factory constructor User.fromJson được dùng để phân tích dữ liệu JSON. Nó xử lý quá trình chuyển đổi của các đối tượng JSON lồng nhau và mảng thành các đối tượng Dart:
  + Trường address được phân tích bằng Address.fromJson.
  + Mảng contacts được phân tích bằng cách ánh xạ từng đối tượng JSON trong mảng sang đối tượng Contact thông qua Contact.fromJson().







# **3. Phương pháp tự động sinh mã(Automated serialization using code generation)**

## 3.1 Giới thiệu

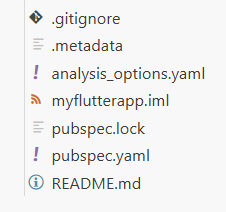
JSON Serialization được sử dụng để lưu dữ liệu vào file hoặc database, cũng như để gửi dữ liệu giữa client và server qua API. Định dạng JSON phổ biến, nhẹ và dễ đọc giúp cho việc ghi và đọc dữ liệu trở nên thuận tiện. Tuy nhiên, nếu thực hiện việc chuyển đổi dữ liệu bằng tay, rất dễ phát sinh lỗi, đặc biệt khi các class có nhiều trường hoặc nested objects. Do đó, trong các dự án lớn, người ta thường sử dụng JSON Serialization tự động bằng cách kết hợp các công cụ như **json\_annotation** để đánh dấu class và field, **json\_serializable** để sinh các hàm fromJson/toJson, và **build\_runner** để chạy code generation. Việc này giúp giảm lỗi, tiết kiệm thời gian và dễ bảo trì, mở rộng dự án hơn so với cách thủ công.

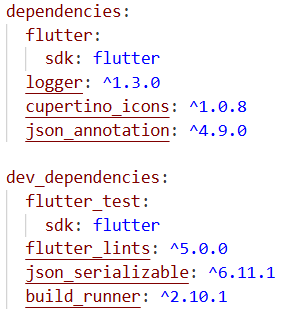
|  |  |  |
| --- | --- | --- |
| Tiêu chí | Thủ công | Tự động |
| Cách thực hiện | Viết tay fromJson() / toJson() | Dùng json\_annotation, json\_serializable, build\_runner để sinh code |
| Ưu điểm | Dễ hiểu với project nhỏ Không cần cài thêm package | Giảm lỗi khi convert dữ liệu Tiết kiệm thời gian với class lớn hoặc nhiều nested objects Dễ bảo trì và mở rộng |
| Nhược điểm | Dễ sai sót khi viết tay Khó bảo trì khi class phức tạp Mất thời gian nếu class nhiều field | Cần cài đặt và học thêm package Phụ thuộc vào quá trình build/code generation |

Định nghĩa các công cụ cơ bản sử dụng trong json serialization:

|  |  |  |
| --- | --- | --- |
| json\_annotation | json\_serializable | build\_runner |
| Cung cấp các **annotation** để đánh dấu class hoặc field cần serialize  Chỉ đánh dấu, **không sinh code trực tiếp**  VD: @JsonSerializable(), @JsonKey() | **Sinh code** fromJson() **và** toJson() **tự động** dựa trên annotation  Khi chạy code generation, tạo file .g.dart chứa các hàm serialize/deserialze  VD:\_$StudentFromJson(json), \_$StudentToJson(this) | Công cụ **chạy code generation**  Phân tích code, tìm annotation, sinh file .g.dart và quản lý xung đột  VD: dart run build\_runner build / watch / --delete-conflicting-outputs |

## 3.2 Thực hành

Bước 1: Thêm dependency (thêm các package vào pubspec.yaml):

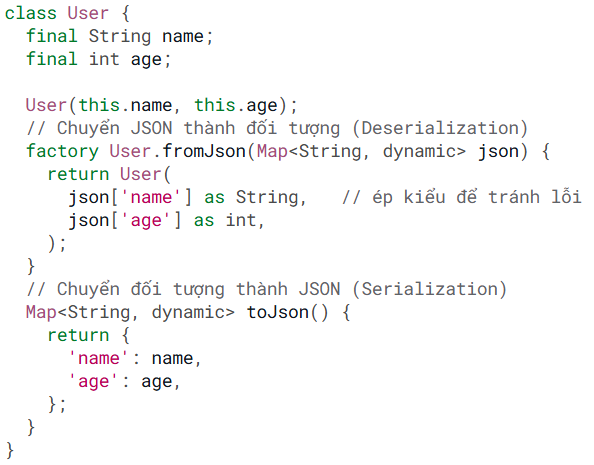


Lưu file để VSCode tự động tải thêm các gói được thêm vào

Hoặc sử dụng lệnh trên terminal:

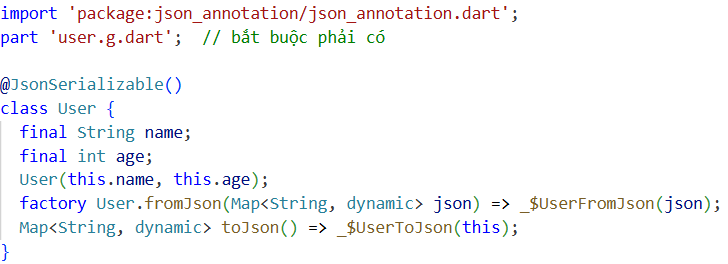
“flutter pub add json\_annotation dev:build\_runner dev:json\_serializable”

Bước 2: Đánh dấu class với annotation

****

Bên trên là một class bình thường khi làm thủ công và đây là sau khi chuyển đổi để sinh ra code generation làm tự động:

user.dart



import 'package:json\_annotation/json\_annotation.dart';

Import package json\_annotation, cung cấp các **annotation** như @JsonSerializable() để đánh dấu class cần sinh code.

part 'user.g.dart';

Khai báo file **phần mở rộng** (user.g.dart) mà công cụ build\_runner sẽ **tự động sinh ra**.  
File này chứa 2 hàm:

* \_$UserFromJson() — chuyển JSON → Object
* \_$UserToJson() — chuyển Object → JSON

@JsonSerializable()

**Annotation** cho biết class User sẽ được xử lý bởi json\_serializable để **tự động sinh code** serialize/deserialze.

class User {

final String name;

final int age;

User(this.name, this.age);

Định nghĩa class User có hai thuộc tính: name và age. Cả hai là **final** (chỉ đọc) và được gán qua constructor.

factory User.fromJson(Map<String, dynamic> json) => \_$UserFromJson(json);

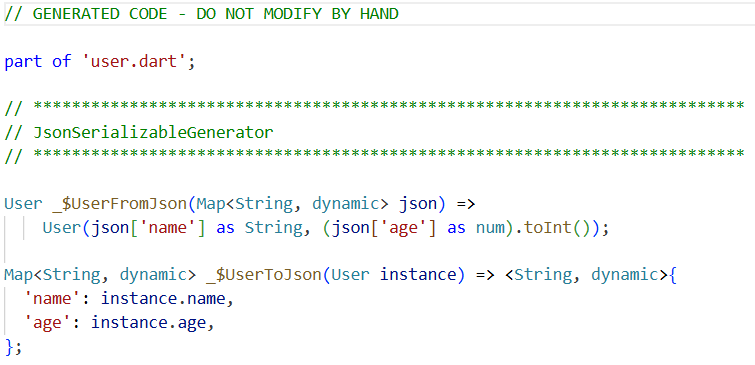
Hàm **factory constructor** để **chuyển JSON thành đối tượng** User. Phần \_$UserFromJson là hàm **được sinh ra tự động** trong file user.g.dart

Map<String, dynamic> toJson() => \_$UserToJson(this);

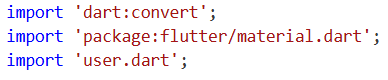
Hàm **chuyển đối tượng** User **thành JSON**. \_$UserToJson(this) cũng là hàm **tự động sinh ra** trong user.g.dart

Bước 3: Chạy lệnh trên terminal sẽ sinh ra user.g.dart:

“dart run build\_runner build --delete-conflicting-outputs”



Bước 4: Gọi lại trong file main.dart để sử dụng.



* dart:convert: cung cấp jsonDecode / jsonEncode để chuyển đổi giữa String JSON và Map<String, dynamic>.
* package:flutter/material.dart: thư viện UI chính của Flutter (Material widgets).
* user.dart: file chứa class User (được định nghĩa với fromJson/toJson). Phải đảm bảo user.g.dart đã được sinh (nếu bạn dùng json\_serializable).



* jsonData là chuỗi JSON mẫu.
* jsonDecode(jsonData) chuyển chuỗi JSON => Map<String, dynamic>
* User.fromJson(...) gọi factory constructor để tạo User từ Map.
* runApp(MyApp(user: user)) khởi chạy ứng dụng và truyền đối tượng user vào widget gốc MyApp.



Tạo một UI đơn giản.

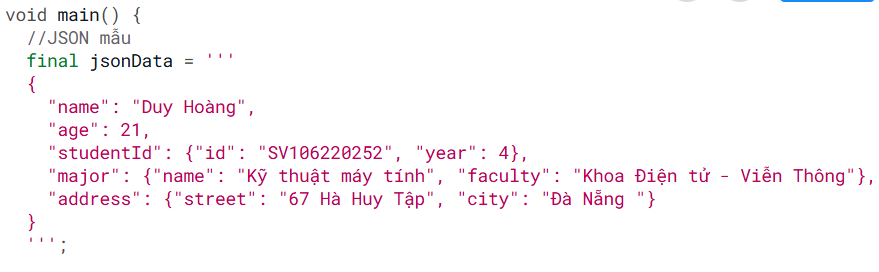
# **4. Các phương pháp tốt nhất khi làm việc với cấu trúc dữ liệu phức tạp**

Khi làm việc với **dữ liệu JSON phức tạp**, điều quan trọng là **giữ code có tổ chức và dễ bảo trì.**Dưới đây là một số **phương pháp tốt nhất:**

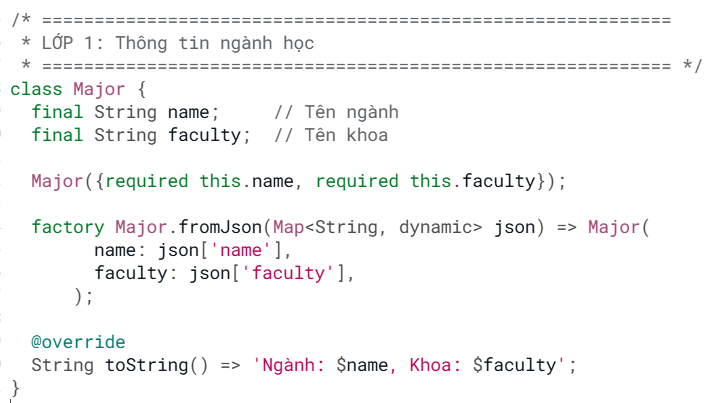
1. **Tách code thành các module:** Chia code thành các **thành phần nhỏ, có thể tái sử dụng.** Mỗi lớp nên **xử lý một phần cụ thể của cấu trúc JSON.**
2. **Sử dụng tên mô tả rõ ràng:** Đặt tên cho các lớp và biến **rõ ràng,** phản ánh **mục đích và dữ liệu mà chúng đại diện.**
3. **Ghi chú trong code**: Dùng **comment** để giải thích **logic phức tạp** và ghi lại **mục đích của từng lớp và phương thức.**
4. **Kiểm tra logic phân tích JSON:** Viết **unit test** để đảm bảo logic phân tích hoạt động chính xác, đặc biệt khi làm việc với **các trường tùy chọn hoặc cấu trúc dữ liệu biến đổi.**

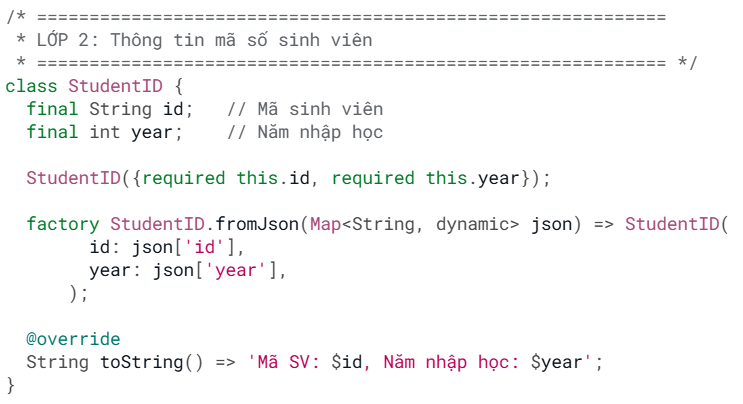
VD:

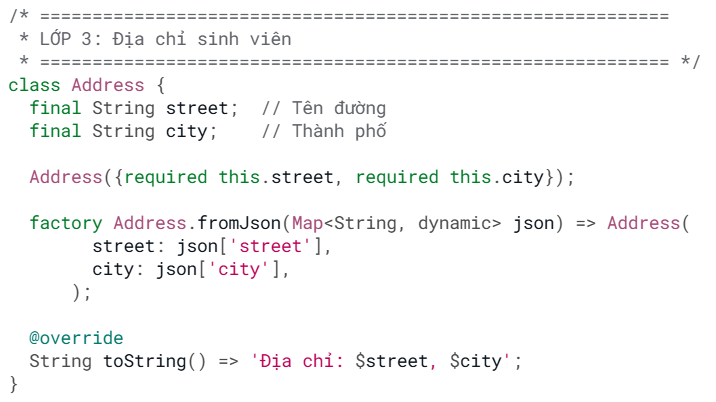
Json mẫu

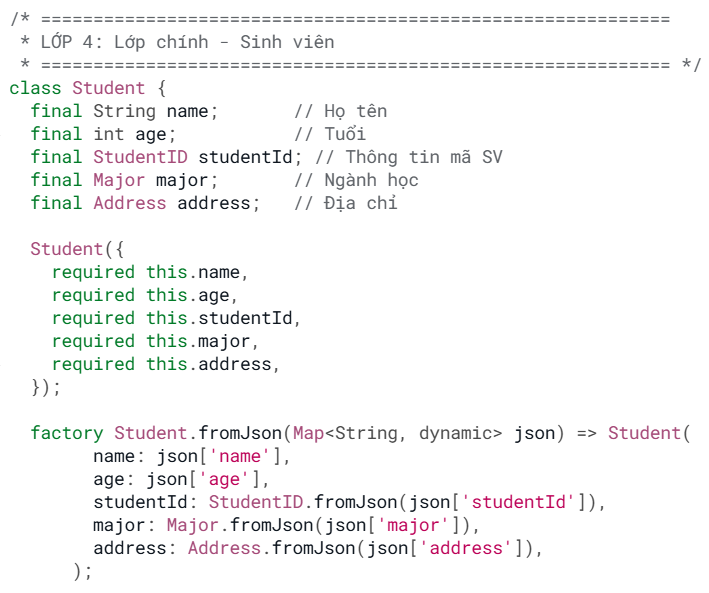


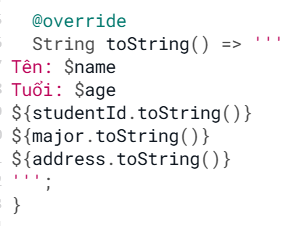
Chia các lớp riêng biệt và chú thích



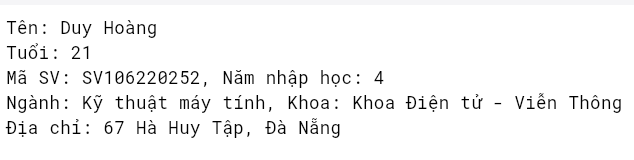




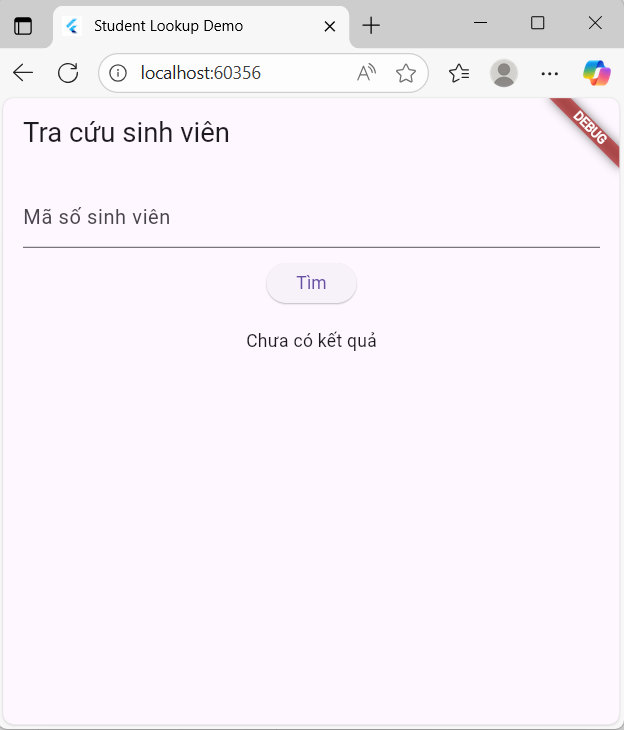


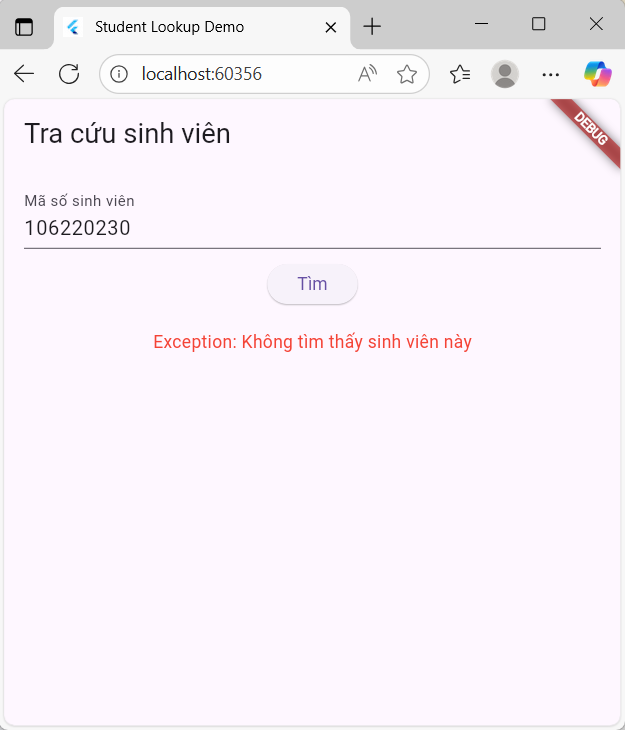


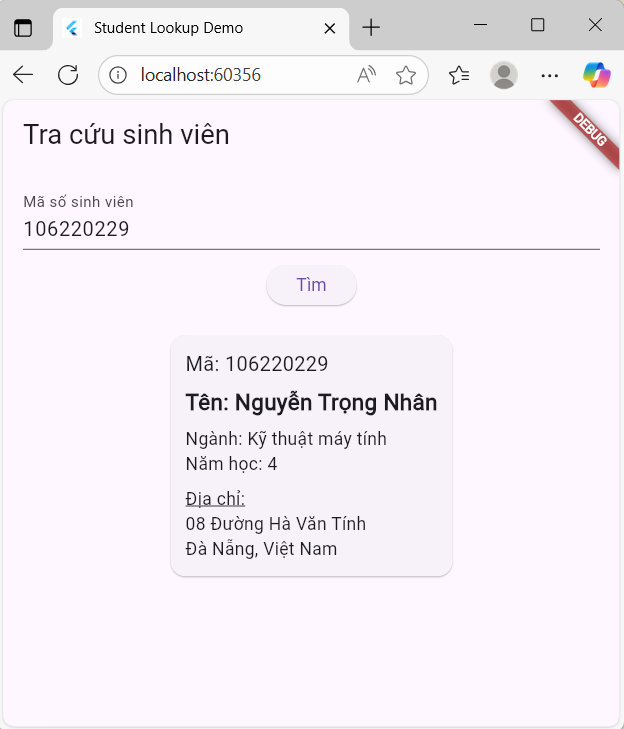
Kết quả:

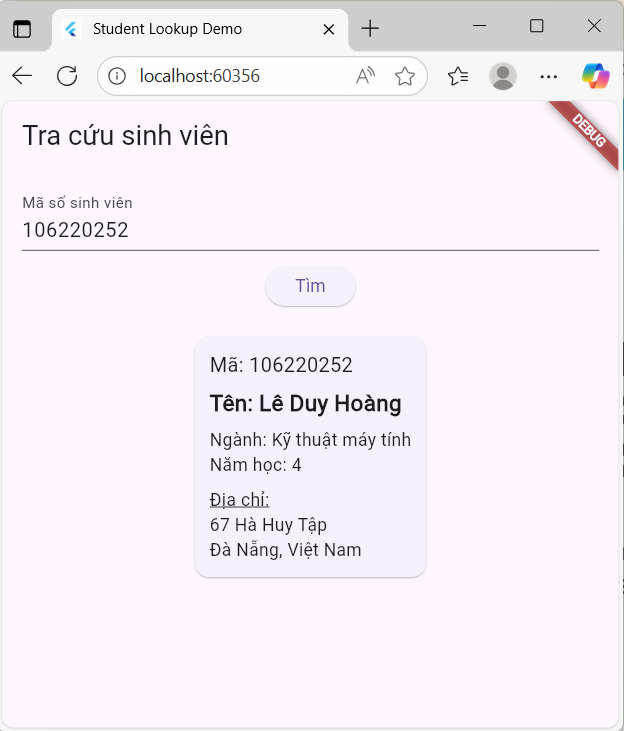


VD: Ứng dụng tra cứu thông tin sinh viên thông qua mã số sinh viên









**TÀI LIỆU THAM KHẢO**

[1] Nguyễn Duy Nhật Viễn, Slide bài giảng môn Lập trình đa nền tảng, Trường Đại học Bách Khoa Đà Nẵng, 2025.

[2] Flutter Team, Flutter Json Serialization: Flutter Documentation, 2025.

[3] Flutter Mastery Library, Mastering Complex JSON Parsing in Flutter, 2025