

BÀI TẬP THỰC HÀNH FLUTTER

(BTTH SỐ 03 – XÂY DỰNG API LARAVEL, SỬ DỤNG TRONG FLUTTER
& ĐÓNG GÓI VỚI DOCKER)

PHẦN 1: BÀI TẬP CÓ HƯỚNG DẪN

Bài tập này sẽ hướng dẫn bạn xây dựng một API quản lý sản phẩm (Product) đơn giản và tích hợp nó vào một ứng dụng di động.

Mục tiêu:

1. Xây dựng các endpoint API (CRUD - Create, Read, Update, Delete) bằng Laravel.
2. Kết nối và lưu trữ dữ liệu với MySQL.
3. Kiểm thử và xác minh API hoạt động đúng cách bằng Postman.
4. Xây dựng ứng dụng Flutter để hiển thị và tương tác với dữ liệu từ API.
5. Đóng gói backend Laravel và MySQL vào container bằng Docker để dễ dàng triển khai.

Yêu cầu chuẩn bị:

- Kiến thức cơ bản về PHP, Flutter & Dart.
- **Môi trường local:** PHP, Composer, Laravel Installer, MySQL Server.
- **Công cụ:** Visual Studio Code (hoặc IDE khác), Postman, Docker Desktop.
- **Flutter SDK** đã được cài đặt.

Phần 1: Xây Dựng Backend API, Kiểm Thử & Tích Hợp Flutter

Bước 1: Tạo Project Laravel và Cấu hình Database

Đầu tiên, chúng ta sẽ tạo một project Laravel mới và kết nối nó với cơ sở dữ liệu MySQL.

1. **Tạo project:** Mở terminal và chạy lệnh sau:

```
laravel new product_api  
cd product_api
```

2. **Cấu hình file .env:** Mở file .env ở thư mục gốc của project và cập nhật thông tin kết nối đến database MySQL của bạn.

```
DB_CONNECTION=mysql  
DB_HOST=127.0.0.1  
DB_PORT=3306  
DB_DATABASE=product_db // Tên database bạn muốn tạo  
DB_USERNAME=root // Username của bạn  
DB_PASSWORD= // Password của bạn (để trống nếu không có)
```

3. **Tạo database:** Truy cập vào công cụ quản lý MySQL của bạn (phpMyAdmin, Sequel Pro, etc.) và tạo một database mới có tên là product_db.

Bước 2: Tạo Model và Migration

Chúng ta sẽ tạo một model Product cùng với file migration để định nghĩa cấu trúc bảng trong database.

1. Chạy lệnh artisan:

Bash

```
php artisan make:model Product -m
```

- Lệnh này tạo ra 2 file: app/Models/Product.php (Model) và một file migration trong database/migrations/.
- Option -m là để tự động tạo file migration tương ứng.

2. **Chỉnh sửa file migration:** Mở file migration vừa được tạo (ví dụ: ..._create_products_table.php) và định nghĩa các cột cho bảng products.

PHP

```
// database/migrations/xxxx_xx_xx_xxxxxx_create_products_table.php

public function up()
{
    Schema::create('products', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->text('description')->nullable();
        $table->decimal('price', 8, 2);
        $table->timestamps();
    });
}
```

3. **Chạy migration:** Quay lại terminal và chạy lệnh để tạo bảng trong database.

Bash

```
php artisan migrate
```

4. **Cấu hình Mass Assignment cho Model:** Mở file app/Models/Product.php và thêm thuộc tính \$fillable để cho phép gán giá trị hàng loạt khi tạo mới sản phẩm.

PHP

```
// app/Models/Product.php

use Illuminate\Database\Eloquent\Factories\HasFactory;
```

```
use Illuminate\Database\Eloquent\Model;
```

```
class Product extends Model
```

```
{
```

```
    use HasFactory;
```

```
    protected $fillable = [
```

```
        'name',
```

```
        'description',
```

```
        'price',
```

```
    ];
```

```
}
```

Bước 3: Tạo API Controller

Controller sẽ chứa logic để xử lý các yêu cầu HTTP (request) gửi đến API.

1. Tạo controller:

Bash

```
php artisan make:controller Api/ProductController --api
```

- Option --api sẽ tạo một controller chỉ chứa các phương thức cần thiết cho API (index, store, show, update, destroy).

2. Viết logic cho các phương thức: Mở file

app/Http/Controllers/Api/ProductController.php và điền logic như sau:

PHP

```
<?php
```

```
// app/Http/Controllers/Api/ProductController.php
```

```
namespace App\Http\Controllers\Api;
```

```
use App\Http\Controllers\Controller;
```

```
use App\Models\Product;
```

```
use Illuminate\Http\Request;
```

```
class ProductController extends Controller
```

```
{
```

```
    // Lấy danh sách tất cả sản phẩm
```

```
public function index()
{
    return Product::all();
}

// Tạo một sản phẩm mới
public function store(Request $request)
{
    $request->validate([
        'name' => 'required|string|max:255',
        'price' => 'required|numeric',
    ]);

    $product = Product::create($request->all());
    return response()->json($product, 201); // 201 Created
}

// Lấy thông tin một sản phẩm cụ thể
public function show(Product $product)
{
    return $product;
}

// Cập nhật thông tin sản phẩm
public function update(Request $request, Product $product)
{
    $request->validate([
        'name' => 'string|max:255',
        'price' => 'numeric',
    ]);

    $product->update($request->all());
}
```

```

        return response()->json($product, 200); // 200 OK
    }

    // Xóa một sản phẩm
    public function destroy(Product $product)
    {
        $product->delete();
        return response()->json(null, 204); // 204 No Content
    }
}

```

- **Giải thích:** Chúng ta sử dụng Eloquent Model (Product) để tương tác với database một cách dễ dàng. Các phương thức trả về dữ liệu dưới dạng JSON, là định dạng chuẩn cho API.

Bước 4: Định nghĩa API Routes

Routes là nơi chúng ta định nghĩa các URL (endpoint) cho API.

1. **Mở file routes/api.php:** Thêm đoạn code sau để đăng ký các routes cho ProductController.

PHP

```

// routes/api.php
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\Api\ProductController;

// Tự động tạo các route cho CRUD: GET, POST, PUT, DELETE,...
Route::apiResource('products', ProductController::class);

```

- Route::apiResource là một cách viết tắt tiện lợi của Laravel để tạo ra toàn bộ các endpoint CRUD cho một resource.

Bước 5: Chạy Server và Kiểm thử với Postman

Bây giờ backend của chúng ta đã sẵn sàng. Hãy khởi động server và dùng Postman để kiểm tra.

1. **Chạy server Laravel:**

Bash

```
php artisan serve
```

Server sẽ chạy tại địa chỉ `http://127.0.0.1:8000`.

2. Mở Postman và thực hiện các request:

- **GET (Lấy danh sách):** `http://127.0.0.1:8000/api/products`
- **POST (Tạo mới):** `http://127.0.0.1:8000/api/products`
 - Trong tab **Body**, chọn **raw** và **JSON**. Nhập dữ liệu:

JSON

```
{
  "name": "Laptop Pro Max",
  "description": "A powerful laptop",
  "price": 1500.99
}
```

- **GET (Xem chi tiết):** `http://127.0.0.1:8000/api/products/1` (thay 1 bằng ID sản phẩm bạn muốn xem).
- **PUT (Cập nhật):** `http://127.0.0.1:8000/api/products/1`
 - Trong tab **Body**, chọn **raw** và **JSON**. Nhập dữ liệu cần cập nhật:

JSON

```
{
  "price": 1450.50
}
```

- **DELETE (Xóa):** `http://127.0.0.1:8000/api/products/1`

Nếu tất cả các request đều trả về kết quả như mong đợi (Status 200 OK, 201 Created, 204 No Content), thì API của bạn đã hoạt động hoàn hảo!

Bước 6: Tích hợp API vào Ứng dụng Flutter

Bây giờ, chúng ta sẽ xây dựng một ứng dụng Flutter đơn giản để hiển thị danh sách sản phẩm từ API.

1. Tạo project Flutter:

Bash

```
flutter create product_app
cd product_app
```

2. Cài đặt package http: Mở file pubspec.yaml và thêm package http.

YAML

```
dependencies:
  http:
```

```
sdk: flutter
```

```
http: ^1.2.1 # Kiểm tra phiên bản mới nhất trên pub.dev
```

Sau đó chạy flutter pub get trong terminal.

3. **Tạo Product Model:** Tạo file lib/product.dart.

Dart

```
// lib/product.dart

class Product {
  final int id;
  final String name;
  final String? description;
  final String price;

  Product({
    required this.id,
    required this.name,
    this.description,
    required this.price,
  });

  factory Product.fromJson(Map<String, dynamic> json) {
    return Product(
      id: json['id'],
      name: json['name'],
      description: json['description'],
      price: json['price'],
    );
  }
}
```

4. **Tạo API Service:** Tạo file lib/api_service.dart để chứa logic gọi API.

Dart

```
// lib/api_service.dart

import 'dart:convert';
```

```

import 'package:http/http.dart' as http;
import 'product.dart';

class ApiService {
  // !! LƯU Ý QUAN TRỌNG VỀ ĐỊA CHỈ IP !!

  // - Nếu dùng máy ảo Android, dùng địa chỉ 10.0.2.2 để trỏ đến
  localhost của máy tính.

  // - Nếu dùng máy thật, đảm bảo điện thoại và máy tính chung một
  mạng WiFi

  // và dùng địa chỉ IP của máy tính (ví dụ: 192.168.1.10).
  static const String baseUrl = 'http://10.0.2.2:8000/api';

  Future<List<Product>> fetchProducts() async {
    final response = await
    http.get(Uri.parse('$baseUrl/products'));

    if (response.statusCode == 200) {
      List<dynamic> body = jsonDecode(response.body);
      List<Product> products = body
        .map(
          (dynamic item) => Product.fromJson(item),
        )
        .toList();
      return products;
    } else {
      throw Exception('Failed to load products');
    }
  }
}

```

5. **Xây dựng giao diện:** Cập nhật file lib/main.dart để hiển thị danh sách sản phẩm.

Dart

```

// lib/main.dart
import 'package:flutter/material.dart';

```



```
import 'api_service.dart';
import 'product.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Product App',
      theme: ThemeData(primarySwatch: Colors.blue),
      home: const ProductListScreen(),
    );
  }
}

class ProductListScreen extends StatefulWidget {
  const ProductListScreen({super.key});
  @override
  State<ProductListScreen> createState() => _ProductListScreenState();
}

class _ProductListScreenState extends State<ProductListScreen> {
  late Future<List<Product>> futureProducts;
  final ApiService apiService = ApiService();

  @override
  void initState() {
    super.initState();
    futureProducts = apiService.fetchProducts();
  }
}
```

```

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Products')),
    body: Center(
      child: FutureBuilder<List<Product>>(
        future: futureProducts,
        builder: (context, snapshot) {
          if (snapshot.connectionState == ConnectionState.waiting) {
            return const CircularProgressIndicator();
          } else if (snapshot.hasError) {
            return Text('Error: ${snapshot.error}');
          } else if (!snapshot.hasData || snapshot.data!.isEmpty) {
            return const Text('No products found');
          }

          final products = snapshot.data!;
          return ListView.builder(
            itemCount: products.length,
            itemBuilder: (context, index) {
              final product = products[index];
              return ListTile(
                title: Text(product.name),
                subtitle: Text(product.description ?? ''),
                trailing: Text('\${product.price}'),
              );
            },
          );
        },
      ),
    ),
  );
}

```

}

- **Giải thích:** Chúng ta dùng FutureBuilder để xử lý việc gọi API bất đồng bộ. Khi dữ liệu trả về, nó sẽ xây dựng một ListView để hiển thị.

6. **Chạy ứng dụng Flutter:** Mở máy ảo hoặc kết nối thiết bị thật và chạy lệnh flutter run. Bạn sẽ thấy danh sách sản phẩm đã tạo ở các bước trước.

Phần 2: Đóng gói Backend API với Docker 🐳

Bây giờ, chúng ta sẽ "đóng gói" ứng dụng Laravel và database MySQL vào các container để chúng có thể chạy ở bất kỳ đâu có Docker.

Bước 7: Tạo Dockerfile

Dockerfile là một file văn bản chứa các chỉ dẫn để build một image Docker.

1. Trong thư mục gốc của project Laravel (product_api), tạo một file tên là Dockerfile (không có đuôi file).
2. Thêm nội dung sau vào Dockerfile:

Dockerfile

```
# Sử dụng image PHP 8.1 FPM làm image cơ sở
FROM php:8.1-fpm

# Cài đặt các extension cần thiết cho Laravel
RUN apt-get update && apt-get install -y \
    git \
    curl \
    libpng-dev \
    libonig-dev \
    libxml2-dev \
    zip \
    unzip \
    && docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd
```

Cài đặt Composer

```
COPY --from=composer:latest /usr/bin/composer /usr/bin/composer
```

Đặt thư mục làm việc

```
WORKDIR /var/www
```

```
# Copy code của project vào container
COPY . .

# Cài đặt các dependencies của project
RUN composer install

# Cấp quyền cho thư mục storage và bootstrap/cache
RUN chown -R www-data:www-data /var/www/storage /var/www/bootstrap/cache

# Expose port 9000 và chạy php-fpm
EXPOSE 9000

CMD ["php-fpm"]
```

Bước 8: Tạo file Nginx Configuration

Chúng ta cần một web server như Nginx để nhận request từ bên ngoài và chuyển cho PHP-FPM xử lý.

1. Trong thư mục gốc project Laravel, tạo một thư mục mới tên là docker.
2. Bên trong thư mục docker, tạo file nginx.conf:

Nginx

```
// docker/nginx.conf

server {
    listen 80;

    index index.php index.html;

    error_log /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;

    root /var/www/public;

    location ~ /\.php$ {
        try_files $uri =404;

        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass app:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
```

```

    }
    location / {
        try_files $uri $uri/ /index.php?$query_string;
        gzip_static on;
    }
}

```

Bước 9: Tạo file docker-compose.yml

File này dùng để định nghĩa và chạy một ứng dụng multi-container (Laravel app, Nginx, MySQL).

1. Trong thư mục gốc của project Laravel, tạo file docker-compose.yml.
2. Thêm nội dung sau:

YAML

```

# docker-compose.yml
version: '3.8'

services:

  # Laravel App Service
  app:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: laravel_app
    restart: unless-stopped
    working_dir: /var/www/
    volumes:
      - ./:/var/www
    networks:
      - app-network

  # Nginx Service
  nginx:
    image: nginx:alpine
    container_name: nginx
    restart: unless-stopped
    ports:

```

```

    - "8000:80" # Map port 8000 của máy host vào port 80 của container
volumes:
    - ./:/var/www
    - ./docker/nginx.conf:/etc/nginx/conf.d/default.conf
networks:
    - app-network

# MySQL Service
db:
    image: mysql:8.0
    container_name: mysql_db
    restart: unless-stopped
    environment:
        MYSQL_DATABASE: ${DB_DATABASE}
        MYSQL_ROOT_PASSWORD: ${DB_PASSWORD}
        MYSQL_PASSWORD: ${DB_PASSWORD}
        MYSQL_USER: ${DB_USERNAME}
    volumes:
        - db-data:/var/lib/mysql
    ports:
        - "3307:3306" # Map port 3307 của máy host vào port 3306 của container
    networks:
        - app-network

networks:
    app-network:
        driver: bridge

volumes:
    db-data:
        driver: local
    ○ Lưu ý: Cập nhật file .env để DB_HOST trỏ đến service db:

```

Đoạn mã

DB_HOST=db

Bước 10: Build và Chạy Container

Bây giờ, hãy khởi động toàn bộ hệ thống bằng một lệnh duy nhất.

1. Build và chạy container ở chế độ nền:

Bash

```
docker-compose up -d --build
```

2. Chạy migration bên trong container:

Bash

```
docker-compose exec app php artisan migrate
```


Bước 11: Kiểm tra lại

- Dùng Postman:** API của bạn bây giờ sẽ có thể truy cập tại `http://localhost:8000/api/products`. Hãy thử lại các request GET, POST,... để chắc chắn mọi thứ vẫn hoạt động.
- Dùng Flutter:** Cập nhật lại `baseUrl` trong `ApiService` của Flutter thành `http://localhost:8000/api` (nếu chạy Flutter trên desktop) hoặc `http://<IP-máy-tính>:8000/api` (nếu chạy trên thiết bị thật) và chạy lại ứng dụng.

Chúc mừng! Bạn đã hoàn thành một chu trình phát triển full-stack hoàn chỉnh, từ việc xây dựng backend, kiểm thử, tích hợp vào ứng dụng di động cho đến việc đóng gói để triển khai.

PHẦN 2: BÀI TẬP TỰ NGHIÊN CỨU

Bài 1: Nâng cao Validation và Xử lý Lỗi cho API (Dễ)

 **Mục tiêu:** Hiểu sâu hơn về hệ thống Validation của Laravel và cách trả về lỗi một cách tường minh cho client.

 **Yêu cầu:**

1. Phần Backend (Laravel):

- Trong `ProductController` tại hàm `store` và `update`, hãy thêm các quy tắc (rules) validation chi tiết hơn:
 - `name`: phải là duy nhất (unique) trong bảng `products`.
 - `description`: có độ dài tối đa là 500 ký tự.
 - `price`: phải là số dương, không được nhỏ hơn 0.
- Khi validation thất bại, Laravel mặc định trả về mã lỗi 422 Unprocessable Entity cùng danh sách các lỗi. Hãy thử dùng Postman gửi dữ liệu không hợp lệ để xem kết quả trả về.


2. Phần Frontend (Flutter):

- (Tùy chọn nâng cao) Khi thực hiện request tạo sản phẩm mới từ Flutter và nhận về lỗi 422, hãy đọc nội dung lỗi trả về từ API và hiển thị thông báo thân thiện cho người dùng (ví dụ: "Tên sản phẩm đã tồn tại").

Gợi ý:

- Tìm hiểu thêm về các [Validation Rules có sẵn của Laravel](#).
- Trong Flutter, khi http.post trả về lỗi, bạn có thể truy cập `response.statusCode` và `response.body` để lấy thông tin chi tiết.

Bài 2: Thêm Chức năng CRUD từ Giao diện Flutter (Trung bình)

 **Mục tiêu:** Hoàn thiện luồng tương tác của ứng dụng Flutter, luyện tập kỹ năng xử lý Form, điều hướng (navigation) và quản lý trạng thái đơn giản.

Yêu cầu:

1. Thêm mới sản phẩm:

- Trên màn hình `ProductListScreen`, thêm một `FloatingActionButton` (nút +).
- Khi nhấn vào nút này, điều hướng người dùng sang một màn hình mới tên là `AddProductScreen`.
- Màn hình `AddProductScreen` chứa một Form với các `TextFormField` để nhập tên, mô tả, và giá sản phẩm.
- Trong `ApiService`, viết hàm `createProduct(Product product)`.
- Khi người dùng nhấn nút "Lưu" trên form, gọi hàm `createProduct`. Nếu thành công, quay trở lại màn hình danh sách (`Navigator.pop`) và **tải lại danh sách sản phẩm**.


2. Xóa sản phẩm:

- Trong `ApiService`, viết hàm `deleteProduct(int id)`.
- Trên mỗi `ListTile` của danh sách sản phẩm, thêm một nút xóa (ví dụ: `IconButton` ở trailing).
- Khi nhấn nút xóa, hiển thị một hộp thoại xác nhận (`AlertDialog`).
- Nếu người dùng xác nhận, gọi hàm `deleteProduct` và cập nhật lại giao diện.

Gợi ý:

- Sử dụng `Navigator.push()` để chuyển màn hình.
 - Để tải lại dữ liệu sau khi thêm/xóa, bạn có thể gọi lại hàm `apiService.fetchProducts()` trong `setState(() {})`.
 - Tìm hiểu về Form widget và `GlobalKey<FormState>` để validate và lấy dữ liệu.
-

Bài 3: Xây dựng Quan hệ Dữ liệu (Product & Category) (Trung bình - Khó)

 **Mục tiêu:** Luyện tập cách thiết kế và triển khai quan hệ một-nhiều (one-to-many) trong database, một kịch bản rất phổ biến trong thực tế.

 **Yêu cầu:**

1. Phần Backend (Laravel):

- Tạo Model Category và file migration tương ứng (php artisan make:model Category -m). Bảng categories chỉ cần cột name.
- Cập nhật lại file migration của bảng products để thêm cột category_id (foreign key). Chạy lại migration (lưu ý: việc này có thể xóa dữ liệu cũ, hãy dùng migrate:fresh).
- Trong Model Product, định nghĩa quan hệ belongsTo với Category.
- Trong Model Category, định nghĩa quan hệ hasMany với Product.
- Tạo CategoryController với các chức năng CRUD cơ bản cho Category.
- Cập nhật ProductController để khi lấy danh sách/chi tiết sản phẩm, hãy trả về cả thông tin category của nó (sử dụng **eager loading** Product::with('category')->get()).
- Cập nhật hàm store sản phẩm để có thể nhận category_id khi tạo.


2. Phần Frontend (Flutter):

- Tạo model category.dart.
- Cập nhật model product.dart để chứa một đối tượng Category.
- Cập nhật giao diện ProductListScreen để hiển thị tên category của từng sản phẩm.

 **Gợi ý:**

- Đọc về [Eloquent Relationships](#) trong Laravel.
- Từ khóa quan trọng: **Eager Loading** để tránh vấn đề N+1 query.
- Trong Flutter, bạn sẽ cần cập nhật hàm Product.fromJson để xử lý object category được lồng trong JSON.

Bài 4: Tích hợp Cache với Redis (Nâng cao)

 **Mục tiêu:** Tìm hiểu cách tích hợp một service mới (Redis) vào hệ thống Docker và áp dụng kỹ thuật caching để tối ưu hiệu năng API.

 **Yêu cầu:**

1. Phần Docker & Laravel:

- Mở file docker-compose.yml, thêm một service mới cho **Redis** (sử dụng image redis:alpine).
- Đảm bảo service app (Laravel) và redis cùng nằm trong một network.
- Cập nhật file .env của Laravel để cấu hình kết nối đến Redis (ví dụ: CACHE_DRIVER=redis, REDIS_HOST=redis).
- Trong ProductController, tại hàm index, sử dụng Cache facade của Laravel để lưu trữ danh sách sản phẩm.
- Ví dụ: `Cache::remember('products', 60, function () { return Product::all(); });` (lưu cache trong 60 giây).
- Khi có một sản phẩm mới được tạo, cập nhật hoặc xóa, hãy xóa cache cũ đi (`Cache::forget('products')`).

2. Kiểm tra:

- Dùng Postman gọi API lấy danh sách sản phẩm. Lần đầu tiên sẽ mất một khoảng thời gian. Những lần gọi tiếp theo (trong vòng 60 giây) sẽ cực kỳ nhanh vì dữ liệu được lấy từ cache.

Gợi ý:

- Tìm hiểu cách thêm service vào docker-compose.yml.
- Đọc tài liệu về [Laravel Cache](#) để hiểu cách sử dụng remember và forget.