Câu 1:

Giải thích về Dependency Injection: là một kỹ thuật trong lập trình giúp quản lý và cung cấp các phụ thuộc (dependencies) của các lớp trong ứng dụng một cách tự động, giúp cho việc quản lý mã nguồn trở nên dễ dàng hơn, tăng tính mở rộng và giảm sự phụ thuộc lẫn nhau giữa các lớp.

Trong .NET Core, DI được hỗ trợ trực tiếp và có thể dễ dàng cấu hình thông qua IServiceCollection và IServiceProvider. ví dụ:

public interface IMessageService

{

string GetMessage();

}

public class MessageService : IMessageService

{

public string GetMessage()

{

return "Hello from MessageService!";

}

}

// Startup.cs

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

// Đăng ký dịch vụ MessageService

services.AddScoped<IMessageService, MessageService>();

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

// Các cấu hình khác

}

}

// Controller sử dụng DI

public class HomeController : Controller

{

private readonly IMessageService \_messageService;

// Constructor Injection

public HomeController(IMessageService messageService)

{

\_messageService = messageService;

}

public IActionResult Index()

{

var message = \_messageService.GetMessage();

return View(model: message);

}

}

Trong .NET Framework, bạn cần cài đặt các thư viện bên ngoài như Unity, Autofac hoặc Ninject để sử dụng DI. Ví dụ:

Install-Package Unity

public interface IMessageService

{

string GetMessage();

}

public class MessageService : IMessageService

{

public string GetMessage()

{

return "Hello from MessageService!";

}

}

// Global.asax.cs

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

var container = new UnityContainer();

container.RegisterType<IMessageService, MessageService>();

DependencyResolver.SetResolver(new UnityDependencyResolver(container));

AreaRegistration.RegisterAllAreas();

RouteConfig.RegisterRoutes(RouteTable.Routes);

}

}

// Controller

public class HomeController : Controller

{

private readonly IMessageService \_messageService;

// Constructor Injection

public HomeController(IMessageService messageService)

{

\_messageService = messageService;

}

public ActionResult Index()

{

var message = \_messageService.GetMessage();

return View((object)message);

}

}

Nếu không hỗ trợ trực tiếp, thực hiện việc tiêm phụ thuộc (injection) một cách thủ công bằng cách truyền các phụ thuộc cần thiết qua constructor hoặc setter. Ví dụ:

// Interface của dịch vụ

public interface IMessageService

{

string GetMessage();

}

// Triển khai dịch vụ

public class MessageService : IMessageService

{

public string GetMessage()

{

return "Hello from MessageService!";

}

}

// Controller sử dụng dịch vụ

public class HomeController

{

private readonly IMessageService \_messageService;

// Constructor injection (tự quản lý phụ thuộc)

public HomeController(IMessageService messageService)

{

\_messageService = messageService;

}

public void ShowMessage()

{

Console.WriteLine(\_messageService.GetMessage());

}

}

// Khởi tạo thủ công trong Main hoặc tại điểm gọi

class Program

{

static void Main(string[] args)

{

// Tự tạo instance của MessageService và truyền nó vào controller

IMessageService messageService = new MessageService();

HomeController controller = new HomeController(messageService);

controller.ShowMessage();

}

}

Câu 2: Bài làm em để ở file Bai2.js

Câu 3: Làm thế nào để tối ưu tốc độ truy vấn khi lưu trữ dữ liệu sự kiện hàng ngày trong SQL với 10 triệu bản ghi mỗi ngày?

Để tối ưu tốc độ truy vấn khi lưu trữ dữ liệu sự kiện hàng ngày với khối lượng lớn (10 triệu bản ghi mỗi ngày) trong SQL, ta có thể áp dụng một số kỹ thuật sau:

1. Phân vùng dữ liệu (Partitioning)

Phân vùng bảng giúp chia bảng lớn thành nhiều phần nhỏ dựa trên một tiêu chí nào đó (ví dụ: theo ngày). Điều này sẽ giúp truy vấn chỉ quét một phần dữ liệu cần thiết thay vì toàn bộ bảng.

Trong trường hợp lưu trữ dữ liệu sự kiện hàng ngày, ta có thể sử dụng partition by range dựa trên cột ngày (event\_date). Điều này giúp SQL chỉ cần truy vấn dữ liệu của các phân vùng liên quan đến ngày được yêu cầu thay vì quét toàn bộ bảng.

CREATE TABLE events (

id BIGINT,

event\_time TIMESTAMP,

event\_data TEXT

) PARTITION BY RANGE (event\_time);

Sau đó, tạo các phân vùng theo từng ngày:

CREATE TABLE events\_20231001 PARTITION OF events

FOR VALUES FROM ('2023-10-01') TO ('2023-10-02');

2. Chỉ mục (Indexing)

Tạo chỉ mục trên các cột thường xuyên được dùng để lọc và truy vấn dữ liệu, chẳng hạn như event\_date hoặc user\_id. Điều này giúp cải thiện tốc độ truy vấn dữ liệu theo các cột này.

Tuy nhiên, không nên tạo quá nhiều chỉ mục vì nó có thể làm giảm tốc độ ghi dữ liệu.

CREATE INDEX idx\_event\_time ON events (event\_time);

3. Tối ưu hóa schema

Loại bỏ cột không cần thiết và chuẩn hóa dữ liệu khi có thể. Bằng cách sử dụng các kiểu dữ liệu phù hợp (như INTEGER thay vì BIGINT nếu giá trị không quá lớn), bạn có thể tiết kiệm dung lượng lưu trữ và tăng tốc độ truy vấn.

Sử dụng định dạng dữ liệu phù hợp cho từng loại cột, ví dụ: dùng TEXT thay vì VARCHAR nếu độ dài của dữ liệu trong cột không cố định.

4. Tối ưu hóa truy vấn (Query Optimization)

Hạn chế \*\*SELECT \*\*\*, chỉ truy xuất những cột cần thiết.

5. Caching

Nếu có một số truy vấn lặp lại, bạn có thể sử dụng cache để giảm tải việc truy vấn liên tục từ cơ sở dữ liệu. Ví dụ: sử dụng Redis để lưu trữ các kết quả truy vấn thường xuyên.

6. Sử dụng Sharding

Nếu lượng dữ liệu quá lớn, bạn có thể xem xét sharding dữ liệu ra nhiều cơ sở dữ liệu vật lý khác nhau dựa trên các tiêu chí như ngày hoặc user\_id. Mỗi shard chứa một phần dữ liệu, giúp giảm tải cho mỗi máy chủ.

7. Lưu trữ dữ liệu archive

Dữ liệu cũ ít được truy vấn có thể được chuyển sang các bảng lưu trữ (archive table) hoặc thậm chí sang các hệ thống lưu trữ khác (như lưu trữ trong hệ thống kho dữ liệu, ví dụ Google BigQuery hoặc Snowflake).