6. Dependency Injection

DI in ASP.NET Core

- The ASP.NET Core framework has been designed from the ground up to be modular and to adhere to good software engineering practices, such as the SOLID principles.
- On that basis, ASP.NET Core has DI baked into the heart of the framework.

52

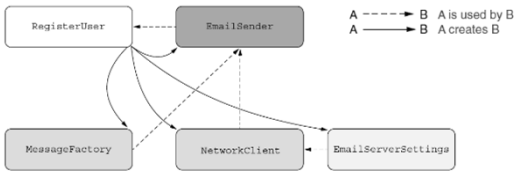
An Example without DI (一)

- Suppose that a user has registered on your web app, and you want to send them an email.
- EmailSender would need to:
  - Create an email message
  - Configure the settings of the email server
  - Send the email to the email server
- Doing all that in one class would go against the single-responsibility principle.

53

An Example without DI (二)

- Each class has several dependencies, so the "root" caller—the RegisterUser handler—needs to know how to create every class it depends on, which is called dependency graph.



54

An Example without DI (三)

- This code is turning into something gnarly.

```
string RegisterUser(string username)
{
    var emailSender = new EmailSender(
        new MessageFactory(),
        new NetworkClient(
            new EmailServerSettings
            (
                Host: "smtp.server.com",
                Port: 25
            ))
    );
    emailSender.SendEmail(username);
    return $"Email sent to {username}!";
}
```

To create EmailSender, you must create all its dependencies.

You need a new MessageFactory.

The NetworkClient also has dependencies.

You're already two layers deep, but there could feasibly be more.

Finally, you can send the email.

55

An Example without DI (四)

- This code has several problems:
  - Not obeying the SRP
  - Considerable ceremony
  - Tied to the implementation
  - Hard to reuse instance
- RegisterUser has an **implicit** dependency on the EmailSender class, as it creates the object manually itself. The only way to know that RegisterUser uses EmailSender is to look at its source code.

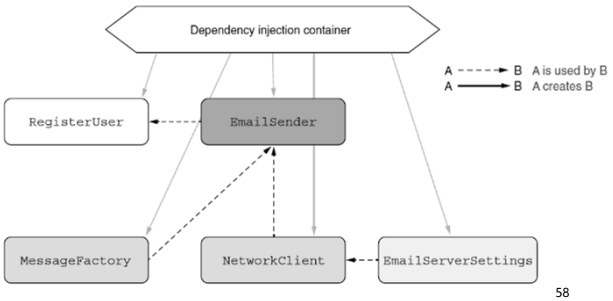
56

Understanding DI (一)

- Generally speaking, any dependencies in your code should be **explicit**, not implicit.
- DI aims to solve the problem of building a **dependency graph** by inverting the chain of dependencies.
- The **DI container** is responsible for creating instances of services.
- The DI container knows how to construct an instance of a service by creating all its dependencies and passing them to the constructor.

57

Understanding DI (二)



58

Understanding DI (三)

- The code shows how the RegisterUser handler would look if DI is used to create EmailSender instead of creating it manually.

```
string RegisterUser(string username, EmailSender emailSender)
{
    emailSender.SendEmail(username);
    return $"Email sent to {username}!";
}
```

The handler is easy to read and understand again.

Instead of creating the dependencies implicitly, injects them directly

59

Using DI in ASP.NET Core (一)

- ASP.NET Core uses DI to configure both its internal components, such as the Kestrel web server, and extra features, such as MVC framework.
- To use these components at runtime, the DI container needs to know about all the classes it will need.

60

Using DI in ASP.NET Core (二)

- Register these services with the Services property on the WebApplicationBuilder instance in Program.cs.

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<Dependency1>();
builder.Services.AddSingleton<Dependency2>();
builder.Services.AddSingleton<Dependency3>();
builder.Services.AddDemoFeature();
```

61

Using DI in ASP.NET Core (三)

- In ASP.NET Core, there are three possible service lifetimes to choose from:

Lifetime	Description	Code sample
Transient	The container creates a new object every time you ask for one.	`services.AddTransient<ISomeService, SomeService>();`
Scoped	The container creates an object per HTTP request and passes that object around to all other objects that want to use it.	`services.AddScoped<ISomeService, SomeService>();`
Singleton	The container creates a single instance of that dependency and always passes that unique object around.	`services.AddSingleton<ISomeService, SomeService>();`

62

Project: Register the services

```
//注册MVC服务
builder.services.AddControllersWithViews();
//仓库接口依赖注入，仅实例化一个对象
builder.services.AddSingleton<IEmployeeRepository,
EmployeeRepository>();

app.MapDefaultControllerRoute();
```

Program.cs

63

7. URL Routing

Convention Routing（一）

- Controllers rely on convention routing instead of the Route attribute.
- The convention in this term refers to the use of the controller class name and the action method name used to configure the routing system.

65

Convention Routing（二）

- MapDefaultControllerRoute registers the MVC Framework as a source of endpoints using a **default convention** for mapping requests to classes and methods.

```
app.MapDefaultControllerRoute();
```

66

Convention Routing（三）

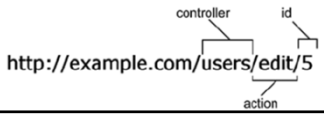
- MapControllerRoute method

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=EmployeeManager}/{action=List}/{id?}");
```

Route name

URL with params

Param defaults



67

Convention Routing（四）

Exercise

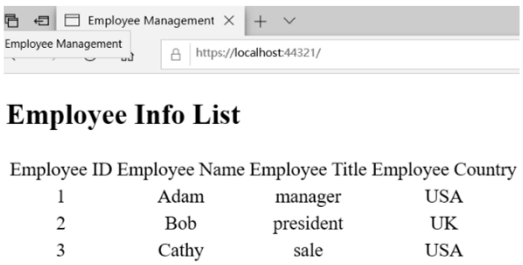
- What's the problem with the code?

```
endpoints.MapControllerRoute(
     name: "default",
     pattern: "{controller}/{action}");

endpoints.MapControllerRoute(
     name: " admin ",
     pattern: "Admin/{action}");
```

68

Project: Run the Application



69

8. Logging Service

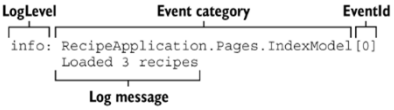
Understanding Logging (一)

- Logging is the process of recording events or activities in an app.
  - Debug errors
  - Trace operations
  - Analyze usage
- Logging is a cross-cutting concern, meaning it applies to every piece of your application.

71

Understanding Logging (二)

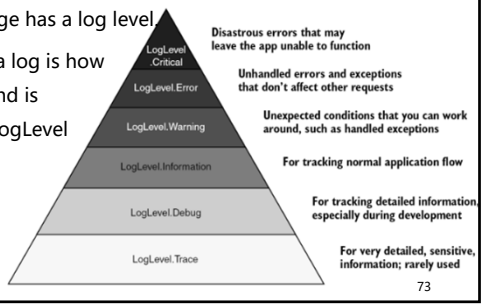
- Logging often involves writing a record to a console, a file, the Windows Event Log, or some other system.
- A log is made up of log entries.
- Each log entry is considered as an event that happened during the program's execution.



72

Understanding Logging (三)

- Every log message has a log level.
- The log level of a log is how important it is and is defined by the LogLevel enum.



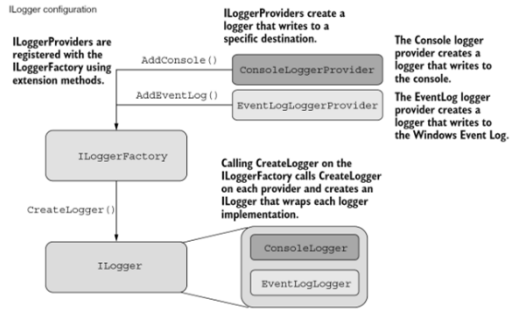
73

The ASP.NET Core Logging Framework (一)

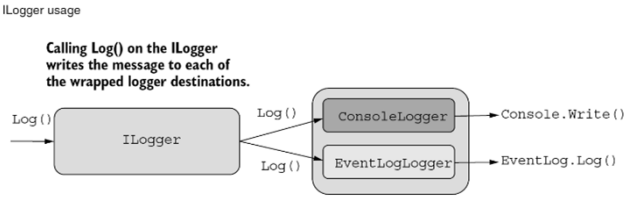
- The ASP.NET Core logging framework consists of several abstractions:
  - Ilogger - It has a Log() method, which is used to write a log message.
  - ILoggerProvider - Create a custom instance of an ILogger, depending on the provider.
  - ILoggerFactory - Glue between the ILoggerProvider instances and the Ilogger.

74

The ASP.NET Core Logging Framework (二)



The ASP.NET Core Logging Framework (三)



Adding a Console Log Provider in Program.cs

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Logging.AddConsole()

WebApplication app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Adds a new provider using the Logging property on WebApplicationBuilder

Adding Log Messages to Your Application

```
public class IndexModel : PageModel
{
    private readonly RecipeService _service;
    private readonly ILogger<IndexModel> _log;

    public ICollection<RecipeSummaryViewModel> Recipes { get; set; }

    public IndexModel(
        RecipeService service,
        ILogger<IndexModel> log)
    {
        _service = service;
        _log = log;
    }

    public void OnGet()
    {
        Recipes = _service.GetRecipes();
        _log.LogInformation(
            "Loaded {RecipeCount} recipes", Recipes.Count);
    }
}
```

Injects the generic ILogger<T> using DI, which implements ILogger

Writes an Information-level log. The RecipeCount variable is substituted in the message.

Changing Log Verbosity with Filtering (一)

- ASP.NET Core includes the ability to filter out log messages before they're written, based on a combination of three things:
  - The log level of the message
  - The category of the logger (who created the log)
  - The logger provider (where the log will be written)

Changing Log Verbosity with Filtering (二)

- The Logging:LogLevel section of the appsettings.json file is used to set the minimum level for logging messages.
- Log messages that are below the minimum level are discarded.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Warning",
      "Microsoft": "Warning"
    },
    "File": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Debug",
        "Microsoft": "Warning"
      }
    }
  }
}
```

Rules to apply if there are no specific rules for a provider

Rules to apply to the File provider

Rules to apply to the Console provider