AN INTRODUCTION TO DEPENDENCY INJECTION

Kamal Beydoun
Lebanese University – Faculty of Sciences I
Kamal.beydoun@ul.edu.lb

UNDERSTANDING THE BENEFITS OF DEPENDENCY INJECTION

- ASP.NET Core framework has been designed from the ground up to be **modular** and to adhere to good software engineering practices.
- For object-oriented programming, the **SOLID** principles have held up well.
- SOLID is a mnemonic for **"single responsibility principle, open-closed, Liskov substitution, interface segregation, and dependency inversion."**

EXAMPLE WITHOUT DI

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/register/{username}", RegisterUser);    ❶

app.Run();

string RegisterUser(string username)                 ❷
{
    var emailSender = new EmailSender();             ❸
    emailSender.SendEmail(username);                 ❹
    return $"Email sent to {username}!";
}
```

```
public class EmailSender
{
    public void SendEmail(string username)
    {
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

❶ The endpoint is called when a new user is created.

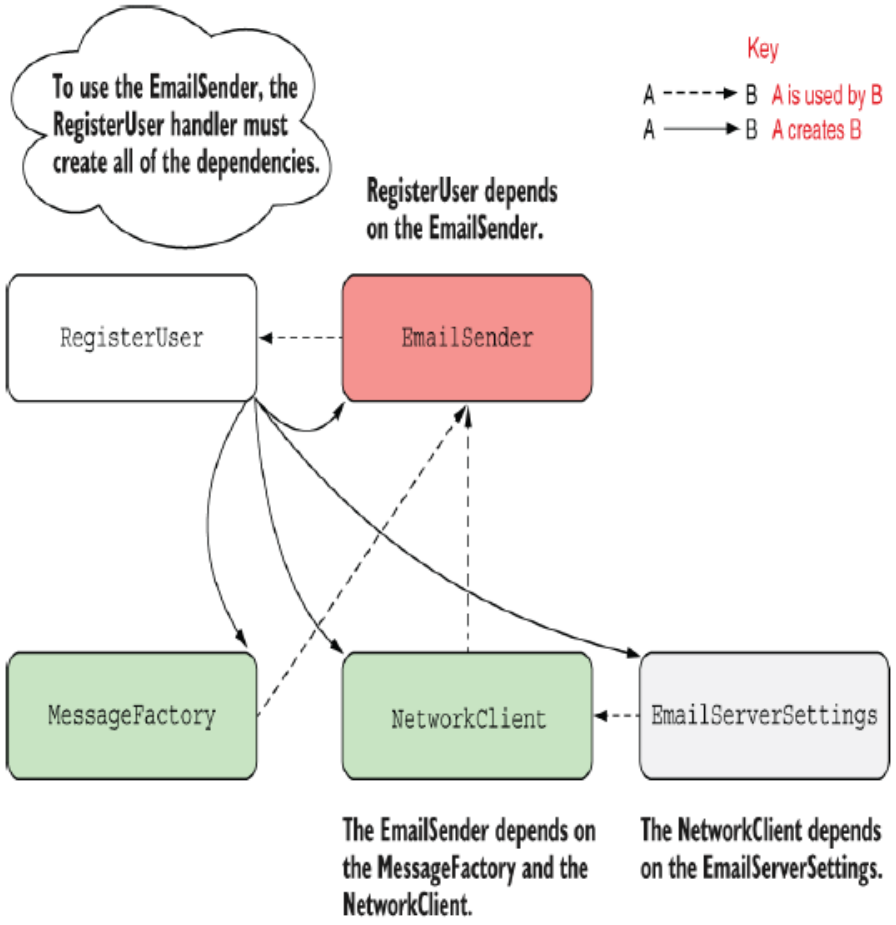
❷ The RegisterUser function is the handler for the endpoint.

❸ Creates a new instance of EmailSender

❹ Uses the new instance to send the email

But what if you later update your implementation of EmailSender so that some of the email-sending logic is implemented by a different class?

DEPENDENCY DIAGRAM WITHOUT DEPENDENCY INJECTION




```
public class EmailSender
{
    private readonly NetworkClient _client;                          ❶
    private readonly MessageFactory _factory;                        ❶
    public EmailSender(MessageFactory factory, NetworkClient client) ❷
    {                                                                ❷
        _factory = factory;                                          ❷
        _client = client;                                            ❷
    }                                                                ❷
    public void SendEmail(string username)
    {
        var email = _factory.Create(username);                       ❸
        _client.SendEmail(email);                                    ❸
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

❶ Now the EmailSender depends on two other classes.

❷ Instances of the dependencies are provided in the constructor.

❸ The EmailSender coordinates the dependencies to create and send an email.

CODE

```
public class EmailSender
{
    private readonly NetworkClient _client;                          ❶
    private readonly MessageFactory _factory;                        ❶
    public EmailSender(MessageFactory factory, NetworkClient client) ❷
    {                                                                ❷
        _factory = factory;                                          ❷
        _client = client;                                            ❷
    }                                                                ❷
    public void SendEmail(string username)
    {
        var email = _factory.Create(username);                       ❸
        _client.SendEmail(email);                                    ❸
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

❶ Now the EmailSender depends on two other classes.

❷ Instances of the dependencies are provided in the constructor.

❸ The EmailSender coordinates the dependencies to create and send an email.

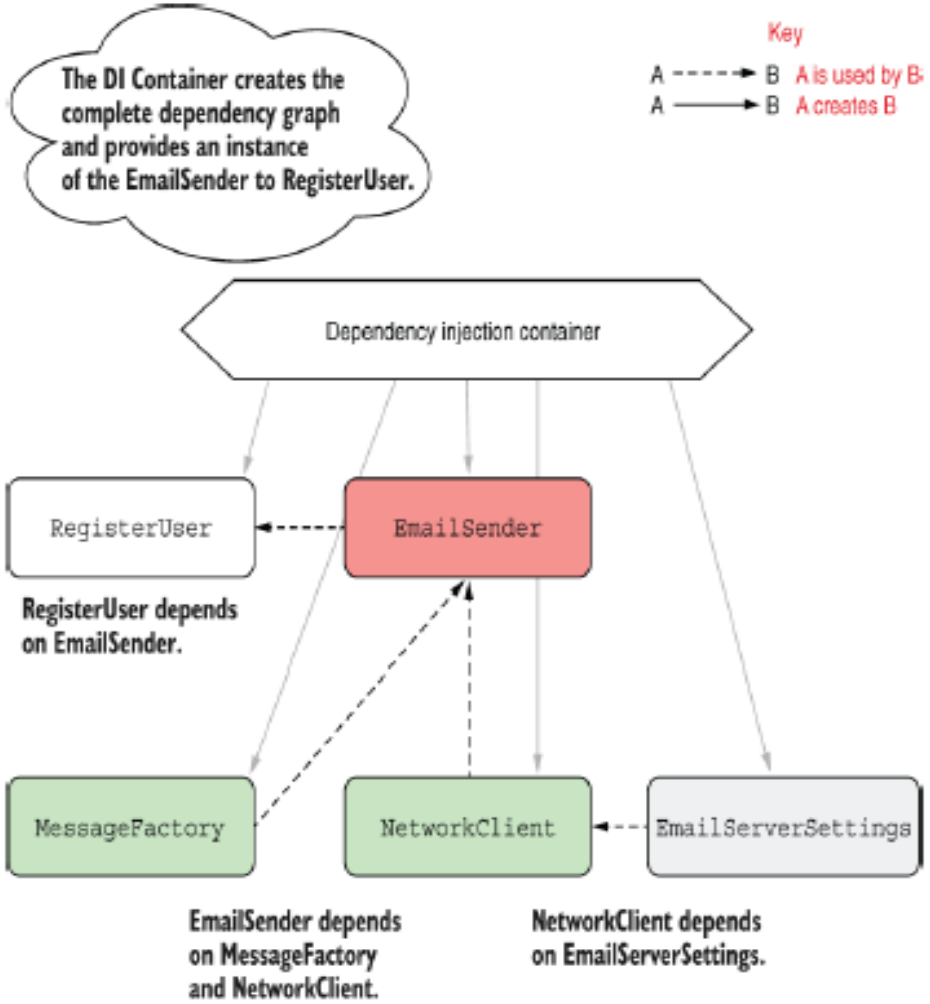
```
public class NetworkClient
{
    private readonly EmailServerSettings _settings;
    public NetworkClient(EmailServerSettings settings)
    {
        _settings = settings;
    }
}
```

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

PROBLEMS IN THE CODE

- **Not obeying the SRP**— Our code is **responsible** for both creating an EmailSender object and using it to send an email.
- **Considerable ceremony**—Ceremony refers to code that you have to write but that isn't adding value directly.
  - In the RegisterUser method, only the **last two lines** are useful, which makes it harder to read and harder to understand the intent of the methods.
- **Tied to the implementation**—If you decide to refactor EmailSender and add another dependency, you'd need to **update every place it's used**. Likewise, if any **dependencies** are refactored, you would need to update this code too.
- **Hard to reuse instance**—In the example code we created new instances of all the objects. But what if creating a new NetworkClient is computationally expensive and we'd like to **reuse instances**? We'd have to add extra code to handle that task, further increasing the amount of boilerplate code.

DEPENDENCY DIAGRAM USING DI



IMPLEMENTATION

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

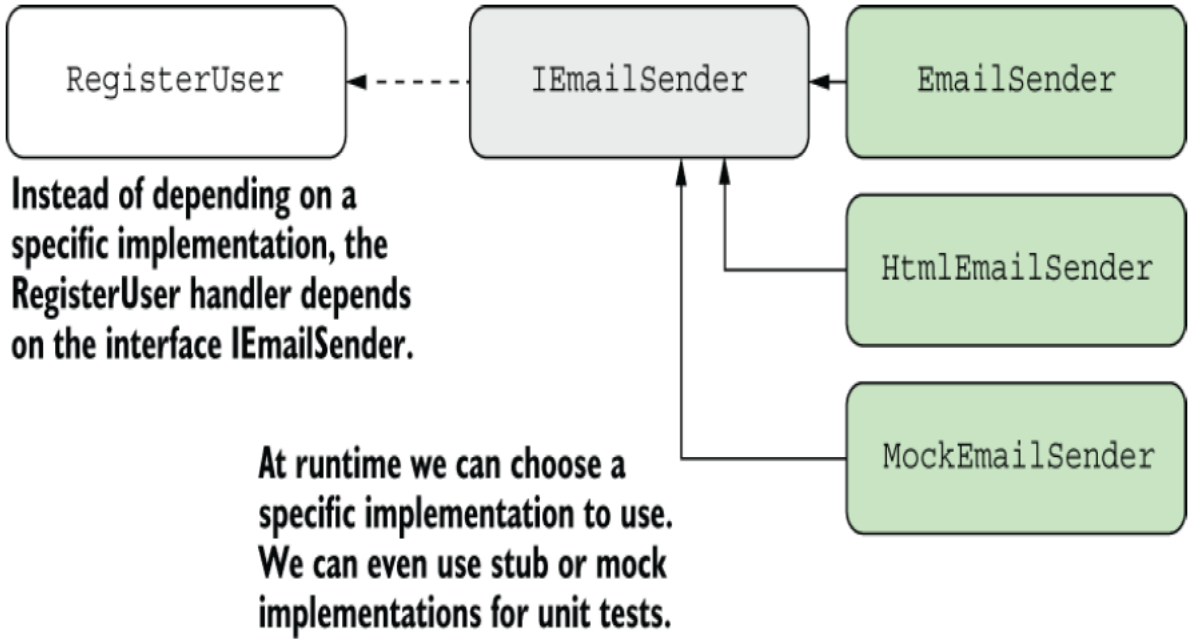
❶ Instead of creating the dependencies implicitly, injects them directly

❷ The handler is easy to read and understand again.

CREATING LOOSELY COUPLED CODE

- **Coupling** is an important concept in object-oriented programming, referring to how a **given class depends** on other classes to **perform** its function.
- **Loosely coupled** code doesn't need to know a lot of details about a particular component to use it.

CREATING LOOSELY COUPLED CODE




```
public interface IEmailSender
{
    public void SendEmail(string username);
}
```

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

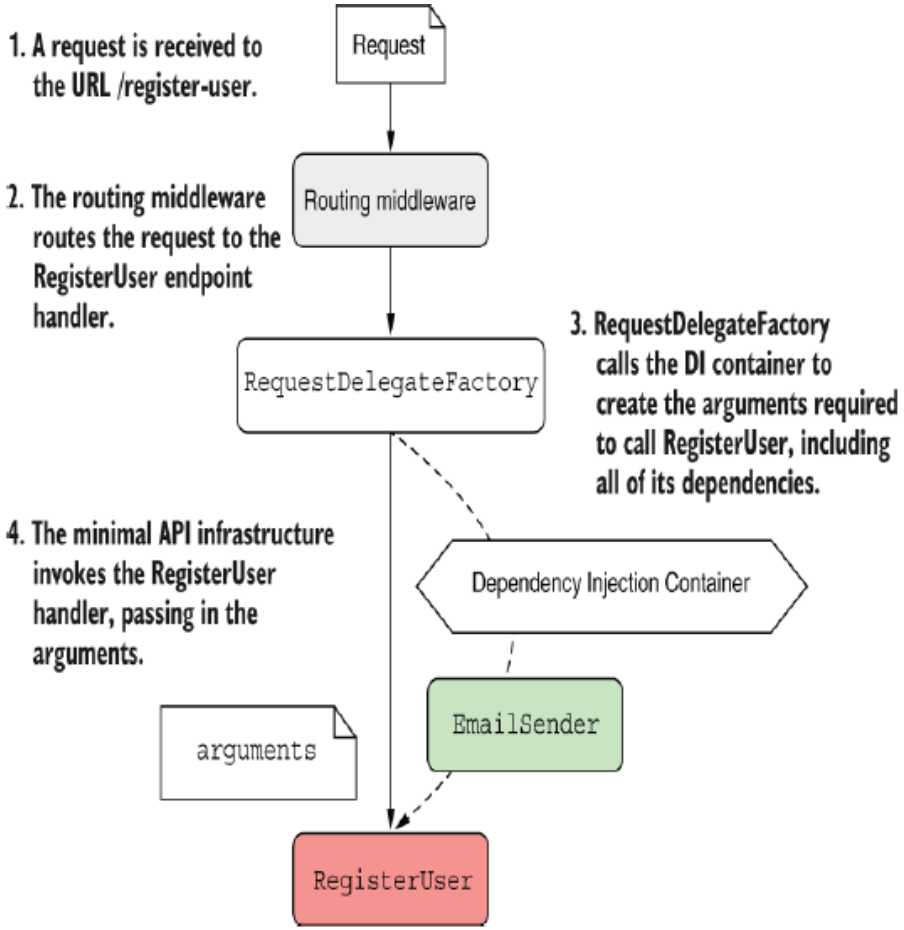
❶ Now you depend on IEmailSender instead of the specific EmailSender implementation.

❷ You don't care what the implementation is as long as it implements IEmailSender.

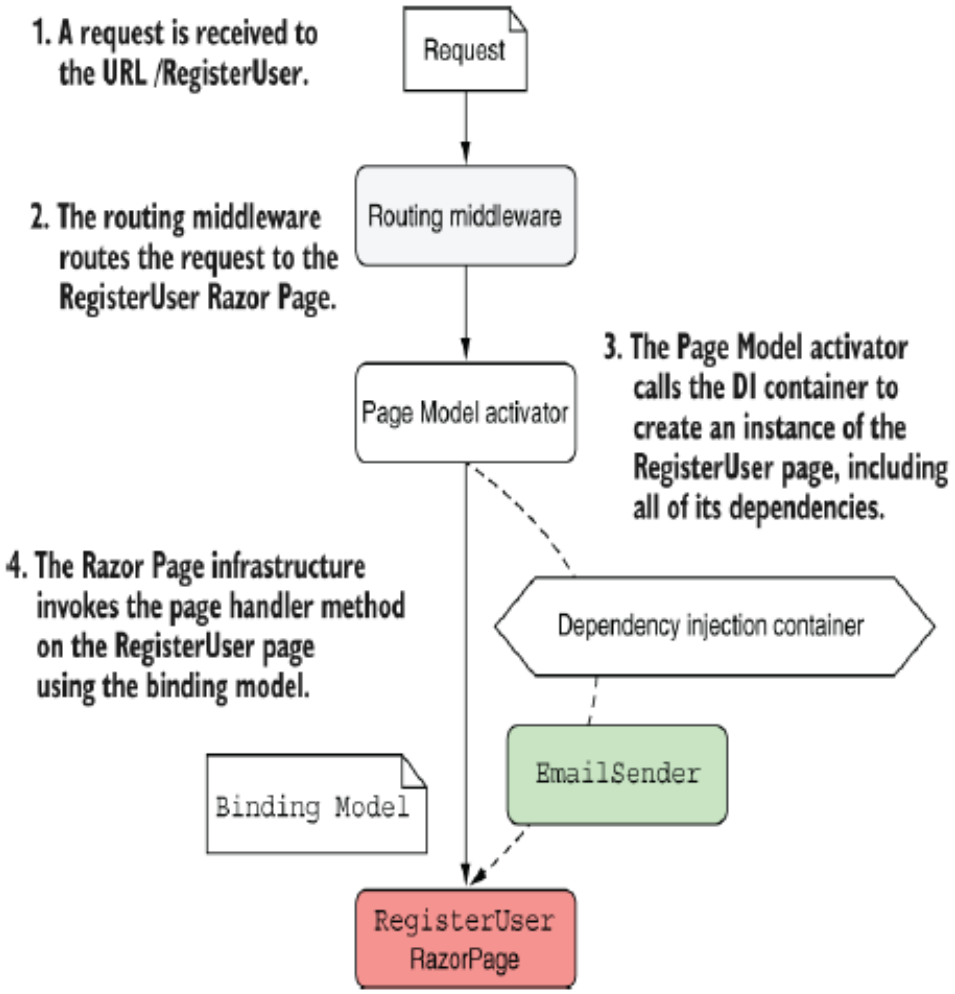
CREATING LOOSELY COUPLED CODE

- How does the application know to use EmailSender in production instead of DummyEmailSender?
- The process of telling your DI container "When you need IEmailSender, use EmailSender" is called **registration**.
- You **register services with a DI container** so that it knows which implementation to use for each requested service.

USING DEPENDENCY INJECTION IN ASP.NET CORE



USING DEPENDENCY INJECTION IN ASP.NET CORE



ADDING ASP.NET CORE FRAMEWORK SERVICES TO THE CONTAINER

- ASP.NET Core uses DI to configure both its internal components, such as the Kestrel web server, and extra features, such as Razor Pages.
- You **register** these services with the **Services property** on the **WebApplicationBuilder** instance in Program.cs.
  - The Services property of WebApplicationBuilder is of type **IServiceCollection**.

ADDING ASP.NET CORE FRAMEWORK SERVICES TO THE CONTAINER

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddRazorPages();    ❶

WebApplication app = builder.Build();
app.MapRazorPages();                 ❷
app.Run();
```

EXCEPTION

Command Prompt (light)

```
Unhandled exception. System.InvalidOperationException: Unable to find the required services
. Please add all the required services by calling 'IServiceCollection.AddRazorPages' inside
 the call to 'ConfigureServices(...)' in the application startup code.
   at Microsoft.AspNetCore.Builder.RazorPagesEndpointRouteBuilderExtensions.EnsureRazorPage
sServices(IEndpointRouteBuilder endpoints)
   at Microsoft.AspNetCore.Builder.RazorPagesEndpointRouteBuilderExtensions.MapRazorPages(I
EndpointRouteBuilder endpoints)
   at Program.<Main>$(String[] args) in C:\repos\asp-dot-net-core-in-action-3e\Chapter 08\S
endingAnEmailWithoutDI\SendingAnEmailWithoutDI\Program.cs:line 6
```

CONFIGURING SERVICES

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options => options.Conventions.)

var app = builder.Build();

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

app.Run();
```

- Add
- AddAreaFolderApplicationModelConvention
- AddAreaFolderRouteModelConvention
- AddAreaPageApplicationModelConvention
- AddAreaPageRoute
- AddAreaPageRouteModelConvention
- AddFolderApplicationModelConvention
- AddFolderRouteModelConvention
- AddPageApplicationModelConvention

USING SERVICES FROM THE DI CONTAINER

- In a minimal API application, you have two main ways to access services from the DI container:
  - Inject services into an endpoint handler.
  - Access the DI container directly in Program.cs.

USING SERVICES FROM THE DI CONTAINER

```
app.MapGet("/links", (LinkGenerator links) =>     ❶
{
    string link = links.GetPathByName("products");
    return $"View the product at {link}";
});
```

❶ The DI container creates a LinkGenerator instance and passes it as the argument to the handler.

USING SERVICES FROM THE DI CONTAINER

- Sometimes you need to access a service **outside the context of a request**.
- The **IServiceProvider** acts as a service **locator**, so you can request services from it directly by using :
  - **GetService<T>()**—Returns the requested service T if it is available in the DI container; otherwise, returns null
  - **GetRequiredService<T>()**—Returns the requested service T if it is available in the DI container; otherwise, throws an InvalidOperationException

USING SERVICES FROM THE DI CONTAINER

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

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

```
LinkGenerator links =                                          ❶
    app.Services.GetRequiredService<LinkGenerator>();          ❶

app.Run();                                                     ❷
```

❶ Retrieves a service from the DI container using the GetRequiredService<T>() extension method

❷ You must access services before app.Run(), as this call blocks until your app exits.