Femi Ojo FOLLOW
Software Engineer. Building the future

C# With Dependency Injection

Published Jun 01, 2018

Introduction

Dependency Injection (DI) is an object-oriented programming design pattern that allows us to develop loosely coupled code. DI helps in getting rid of tightly coupled software components. The purpose of DI is to make code maintainable and easy to update.

DI is a technique to create a dependency or dependencies outside the class that uses it. The dependencies are injected from the code that calls the class and any information about their creation are kept away from the inside of the class. This is also why DI is an implementation of the Inversion of control (IoC) principle.

To further explain the dependency injection design pattern, the inversion of control principle has to be broken down.

Inversion of Control Principle

Normally, the flow of the program logic is determined by objects that are bound to one another. With the inversion of control, the flow depends on the defined abstractions to be implemented that is built up during program execution. This principle can be achieved by techniques like dependency injection. In IoC, the code could also be linked statically during compilation to the defined interface functions, but finding the implementation of the function to execute by reading its description from external configuration instead of with a direct reference in the code itself.

How it Works

In dependency injection, a dependent object or service is provided with the object it needs at run time. The provided object will satisfy the dependency during program execution but would not be known at compile time. Rather than directly instantiating dependencies, or using static references, the objects a class needs in order to perform its actions are provided to the class in some abstracted form.

An example of the traditional way:

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private GoogleEmailService _emailService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new GoogleEmailService();
    }

    public void Register(string emailAddress, string password)
    {
        var authResult = _authService.RegisterUser(emailAddress,password);
        _emailService.SendMail(emailAddress, authResult.ConfirmationMessage);
    }
}

public class GoogleOAuthService
{
    public GoogleOAuthResult RegisterUser(string emailAddress, string password)
    {
        //Register a new user
    }
}

public class GoogleEmailService
{
    public SendMail(string emailAddress, string message)
    {
        //Send an email using google
    }
}
```

Looking at the services used in this register action of the account controller, we can observe that changing the service would imply changing a lot of the codebase, especially if the service has been used in multiple parts of the project.

For instance, if the email service is replaced with a new one as shown below, we can see that the tightly coupled email service would need to be changed everywhere is it

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private OutlookEmailService _emailService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new OutlookEmailService();
    }

    public void Register(string emailAddress, string password)
    {
        var authResult = _authService.RegisterUser(emailAddress,password);
        _emailService.SendMail(emailAddress, authResult.ConfirmationMessage);
    }
}

public class OutlookEmailService
{
    public void SendMail(string emailAddress, string message)
    {
        //Send an email using outlook
    }
}
```

From the examples shown, we can observe that there is a generic function that all email services will provide. It sends an email which can help us provide an abstraction of the email service, which is going to be used even when we do not know which exact service (Google or Outlook) is to be used.

```
public interface IEmailService
{
    void SendMail(string emailAddress, string message)
}
```

The corresponding implementation can then inherit from the interface.

```
public class GoogleEmailService: IEmailService
{
    public SendMail(string emailAddress, string message)
    {
        //Send an email using google
    }
}

public class OutlookEmailService: IEmailService
{
    public void SendMail(string emailAddress, string message)
    {
        //Send an email using outlook
    }
}
```

The above abstraction (interface) can be used in our UserLogic such that:

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private IEmailService _emailService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new OutlookEmailService() // or Google;
    }

    public void Register(string emailAddress, string password)
    {
        var authResult = _authService.RegisterUser(emailAddress,password);
        _emailService.SendMail(emailAddress, authResult.ConfirmationMessage);
    }
}
```

The email service is still tightly coupled, so we have to inject it in using one of the multiple types of dependency injection.

1. Constructor Injection

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private IEmailService _emailService;

    public UserLogic(IEmailSevice emailService)
    {
        _authService = new GoogleOAuthService();
        _emailService = emailService;
    }
    ...
}
```

The part of the code calling the `UserLogic` class would have to specify what email service going to be used and pass it in the constructor when instantiating it.

```
...
    GoogleEmailService googleEmailService = new GoogleEmailService();
    UserLogic userLogic = new UserLogic(googleEmailService);
...
```

2. Setter Injection

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private IEmailService _emailService;

    public IEmailService EmailService
    {
        get
        {
            return _emailService;
        }
        set
        {
            _emailService = value;
        }
    }

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
    }
    ...
}
```

The part of the code calling the `UserLogic` class would have to specify what email service going to be used and as one of its property.

```
...
    OutlookEmailService outlookEmailService = new OutlookEmailService();
    UserLogic userLogic = new UserLogic()
        {
            EmailService = outlookEmailService
        };
...
```

3. Method Injection

```
public class UserLogic
{
    private GoogleOAuthService _authService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new OutlookEmailService() // or Google;
    }

    public void Register(string emailAddress, string password, IEmailService emailServ
    {
        var authResult = _authService.RegisterUser(emailAddress,password);
        emailService.SendMail(emailAddress, authResult.ConfirmationMessage);
    }
}
```

The part of the code calling the `UserLogic` Register method would have to specify what email service as one of the parameters.

```
...
    OutlookEmailService outlookEmailService = new OutlookEmailService();
    UserLogic userLogic = new UserLogic();
    userLogic.Register(email, password, outlookEmailService);
...
```

Conclusion

Dependency injection helps achieve improved class coupling, better code reusability, code maintainability, and application unit testing

C# Di Solid principles C# . net

Enjoy this post?

♡ 12 11

Enjoy this post? Give **Femi Ojo** a like if it's helpful.

12 11 SHARE

Femi Ojo

Software Engineer. Building the future

I am a software developer with a commitment to and experience of developing innovative and creative software solutions. I have gained commercial experience and exposure to business software development including planning, testing ...

FOLLOW

11 Replies

Leave a reply

kalyani Dotnet a month ago

nice explanations thanks...happy to see this

Reply

Adam Tate a month ago

Thankyou very much,

In

public class UserLogic

Show more

Sharon A S 7 days ago

Exactly. We can remove that line. He may copied the code from the top. Anyway nice explanation.

Reply

Yikekas Zun a month ago

Thank you very much, good explanation

Reply

Show more replies

Enjoy this post?

12 11