Built in Ioc Container

The terms Dependency Injection (DI) and inversion of Control (IoC) are generally used interchangeably to describe the same design pattern. Hence some people say IoC Container and some people say DI container but both terms indicate the same thing. So don't be confused by the terminology.

The IoC container that is also known as a DI Container is a framework for implementing automatic dependency injection very effectively. It manages the complete object creation and its lifetime, as well as it also injects the dependencies into the classes.

What is DI Container?

A DI Container is a framework to create dependencies and inject them automatically when required. It automatically creates objects based on the request and injects them when required. DI Container helps us to manage dependencies within the application simply and easily.

The DI container creates an object of the defined class and also injects all the required dependencies as an object, a constructor, a property, or a method that is triggered at runtime and disposes itself at the appropriate time. This process is completed so that we don't have to create and manage objects manually all the time.

Registering Application Service

In ASP.NET Core, the built-in IOC (Inversion of Control) container is used for managing dependencies within your application. Registering an application service with the built-in IOC container involves specifying how the container should create instances of the services your application relies on.

Here's how you can register an application service with the built-in IOC container in ASP.NET Core Web API:

1. **Create Your Service:** First, create the service class that you want to register with the IOC container. This could be a service responsible for handling business logic, data access, or any other functionality your application requires.

```
public interface IMyService
{
   void DoSomething();
}

public class MyService : IMyService
{
   public void DoSomething()
   {
      // Implementation of the service method
```

```
    }
}
```

2. **Register the Service in Startup.cs:** In your application's Startup.cs file, you'll find the ConfigureServices method. This method is where you register your application services with the IOC container.

```
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Register your service with the IOC container
        services.AddScoped<IMyService, MyService>();
    }
}
```

In this example, AddScoped is used to register the service with scoped lifetime. Other options include AddTransient and AddSingleton, each with different lifetimes.

3. **Inject the Service into Controllers or Other Components:** Once the service is registered, you can inject it into your controllers or other components that need to use it.

```
using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("[controller]")]
public class MyController : ControllerBase
{
    private readonly IMyService _myService;

    public MyController(IMyService myService)
    {
        _myService = myService;
    }

    [HttpGet]
    public IActionResult Get()
    {
        _myService.DoSomething();
        return Ok();
    }
}
```

In this example, the MyController constructor injects the IMyService dependency, which allows the controller to use methods defined in the MyService class.

That's it! Your application service is now registered with the built-in IOC container of ASP.NET Core, and you can use constructor injection to access it throughout your application.

Service Lifetime

1. **Transient**: Services registered with transient lifetime are created each time they're requested. This means that a new instance of the service is created for each use. Transient services are suitable for lightweight and stateless components that don't maintain state across multiple requests.

   ```
   services.AddTransient<IMyService, MyService>();
   ```

2. **Scoped**: Scoped services are created once per request within the scope of an HTTP request. This means that the same instance of the service is reused throughout the duration of a single HTTP request, but a new instance is created for each new request. Scoped services are typically used for components that need to maintain state within a single request, such as database contexts.

   ```
   services.AddScoped<IMyService, MyService>();
   ```

3. **Singleton**: Singleton services are created once and shared throughout the lifetime of the application. This means that the same instance of the service is reused across all requests and clients. Singleton services are suitable for stateless or thread-safe components that can be shared safely across the application.

   ```
   services.AddSingleton<IMyService, MyService>();
   ```

Extension method for register services

Extension methods in C# allow you to add new methods to existing types without modifying the original type's code. In the context of ASP.NET Core's dependency injection, extension methods are commonly used to provide cleaner and more organised ways to register application services with the built-in IOC container.

Here's a typical example of how extension methods are used for service registration in ASP.NET Core:

1. **Create an Extension Class:** First, you create a static class that contains extension methods for the IServiceCollection interface. This interface represents the collection of services in the ASP.NET Core application.

   ```
   public static class MyServiceExtensions
   {
      public static IServiceCollection AddMyService(this IServiceCollection services)
      {
   ```

```
        services.AddTransient<IMyService, MyService>();
        return services;
    }
}
```

2. **Define Your Service:** In this example, IMyService is the service interface, and MyService is the concrete implementation of that interface.

```
public interface IMyService
{
    void DoSomething();
}

public class MyService : IMyService
{
    public void DoSomething()
    {
        // Implementation of the service method
    }
}
```

3. **Usage:** Now, you can use the extension method AddMyService() in your Startup.cs file.

```
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMyService();
    }
}
```

This usage pattern helps to keep the Startup.cs file clean and focused on the configuration of services.

4. **Optional Parameters:** Extension methods can also accept additional parameters to customise service registration.

```
public static class MyServiceExtensions
{
      public static IServiceCollection AddMyService(this IServiceCollection services,
ServiceLifetime lifetime = ServiceLifetime.Transient)
    {
        switch (lifetime)
        {
            case ServiceLifetime.Transient:
                services.AddTransient<IMyService, MyService>();
                break;
            case ServiceLifetime.Scoped:
                services.AddScoped<IMyService, MyService>();
                break;
            case ServiceLifetime.Singleton:
                services.AddSingleton<IMyService, MyService>();
                break;
            default:
                throw new ArgumentOutOfRangeException(nameof(lifetime), lifetime, null);
        }
        return services;
    }
}
```

This modification allows you to specify the service lifetime when registering the service.

Constructor Injection

Constructor injection is a pattern used in dependency injection (DI) where dependencies required by a class are provided through the class's constructor. In this pattern, the dependencies are typically interfaces or abstract classes, and the concrete implementations are injected into the class at runtime.

Here's how constructor injection works:

1. **Define Dependencies:** Identify the dependencies that a class needs to perform its tasks. These dependencies are typically declared as interfaces or abstract classes to promote loose coupling.

```
public interface ILogger
{
    void Log(string message);
}

public class DatabaseService
{
    private readonly ILogger _logger;

    public DatabaseService(ILogger logger)
    {
        _logger = logger;
    }

    public void PerformDatabaseOperation()
    {
        // Use _logger to log messages
        _logger.Log("Performing database operation...");
    }
}
```

2. **Constructor Injection:** In the class's constructor, define parameters for each dependency that the class needs. In this example, the DatabaseService class depends on an ILogger instance.

3. **Dependency Injection Container:** Use a dependency injection container provided by the framework (like ASP.NET Core's built-in container) or a third-party container to wire up dependencies and manage the object lifecycle.

4. **Configuration:** Configure the dependency injection container to map each dependency interface to its concrete implementation.

```
services.AddSingleton<ILogger, ConsoleLogger>();
```

In this example, whenever an instance of ILogger is requested, the dependency injection container will provide an instance of ConsoleLogger.

5. **Consuming the Dependency:** Inside the class, use the injected dependency as needed to perform operations.

```
public void PerformDatabaseOperation()
{
    // Use _logger to log messages
    _logger.Log("Performing database operation...");
}
```

With constructor injection, the dependencies are provided to the class externally, making the class easier to test, maintain, and extend. It also promotes loose coupling between components, as the class does not need to be aware of how its dependencies are created or managed. Additionally, constructor injection allows for easier mocking of dependencies during unit testing, as mock implementations can be injected instead of the actual implementations.

UseDeveloperExceptionPage

In ASP.NET Core, UseDeveloperExceptionPage is a middleware that displays detailed error information in the browser during development. It's typically included in the middleware pipeline during development to help developers diagnose and debug issues more easily.

Here's how it works:

1. **Add UseDeveloperExceptionPage Middleware:** In the Configure method of the Startup class, add the UseDeveloperExceptionPage middleware to the middleware pipeline. This middleware should be added early in the pipeline to catch exceptions and display detailed error information.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // Other error handling middleware for production
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }

    // Other middleware configurations...
}
```

Here, UseDeveloperExceptionPage is added conditionally based on the environment. It's enabled only when the application is running in the development environment (env.IsDevelopment()).

2. **Detailed Error Information:** When an unhandled exception occurs during a request, the UseDeveloperExceptionPage middleware intercepts the exception and displays detailed error information in the browser, including the exception message, stack trace, and request information. This information is extremely helpful for diagnosing issues during development.

3. **Security Considerations:** It's important to remember that the information displayed by UseDeveloperExceptionPage can expose sensitive information about your application, such as stack traces and environment variables. Therefore, it should only be enabled in development environments and disabled in production environments to avoid leaking sensitive information to users.

4. **Alternatives in Production:** In production environments, instead of UseDeveloperExceptionPage, you would typically use more generic error handling middleware, such as UseExceptionHandler, which logs exceptions and displays a generic error page to users while still logging detailed error information for developers.

In summary, UseDeveloperExceptionPage is a middleware in ASP.NET Core that provides detailed error information in the browser during development, helping developers diagnose and debug issues more easily. However, it should be used with caution and disabled in production environments to prevent exposing sensitive information.

UseExceptionHandler

UseExceptionHandler is a middleware in ASP.NET Core that is used to catch exceptions that occur during request processing and provide a centralised location for handling those exceptions. In the context of a Web API, UseExceptionHandler can be used to handle exceptions thrown by controllers or middleware and return appropriate error responses to clients.

Here's how you can use UseExceptionHandler for a Web API:

1. **Add UseExceptionHandler Middleware:** In the Configure method of the Startup class, add the UseExceptionHandler middleware to the middleware pipeline. This middleware should be added after all other middleware to catch any unhandled exceptions that occur during request processing.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
        app.UseHsts();
    }

    // Other middleware configurations...
}
```

In this example, if an unhandled exception occurs during request processing, it will be caught by the UseExceptionHandler middleware and redirected to the /error endpoint.

2. **Create Error Handling Endpoint:** Create an endpoint to handle errors and return appropriate error responses to clients. This could be a controller action or a Razor page that generates error responses in the desired format (e.g., JSON).

```
[ApiController]
public class ErrorController : ControllerBase
{
    [Route("/error")]
    public IActionResult Error()
    {
        // Generate error response
        var problemDetails = new ProblemDetails
```

```
            {
                Status = (int)HttpStatusCode.InternalServerError,
                Title = "An error occurred",
                Detail = "Internal server error"
            };
            return StatusCode(problemDetails.Status.Value, problemDetails);
        }
    }
```

In this example, the Error action returns a ProblemDetails object representing the error response with an appropriate status code and message.

3. **Customise Error Handling:** You can customise the error handling logic in the Error action to suit your application's requirements. This could include logging the exception, returning different error responses based on the exception type, or translating exceptions into custom error codes.

Using UseExceptionHandler in a Web API allows you to centralise error handling logic and provide consistent error responses to clients in case of exceptions. It helps improve the reliability and usability of your API by ensuring that clients receive informative error messages when errors occur.

Logging API

In software development, logging is the process of recording events, messages, and other information that occurs during the execution of an application. Logging is essential for diagnosing issues, monitoring application behaviour, and understanding how an application is performing in production environments. Logging APIs provide a standardised way for developers to implement logging functionality in their applications.

Here's an overview of a typical logging API:

1. **Log Levels**: Logging APIs typically support different log levels to categorise the severity of log messages. Common log levels include:
   - Trace: Detailed information, typically used for debugging purposes.
   - Debug: Information useful for debugging the application.
   - Info: General information about the application's operation.
   - Warning: Indications of potential issues that do not necessarily impact application operation.
   - Error: Indications of errors that need attention but do not necessarily halt application execution.
   - Critical: Critical errors that require immediate attention and may result in application termination.

2. **Logger Interface:** Logging APIs often define a logger interface or abstract class that serves as the entry point for logging messages. This interface typically includes methods for logging messages at different log levels.

```
public interface ILogger
{
   void Log(LogLevel level, string message);
}
```

3. **Logger Implementation:** Developers can implement the logger interface to provide specific logging functionality for their applications. Logger implementations may write log messages to various destinations, such as the console, files, databases, or external logging services.

4. **Configuration:** Logging APIs often provide configuration options to customise the behaviour of the logger, such as specifying the log level threshold, defining log message formats, and configuring logging destinations.

5. **Logging Frameworks:** In addition to simple logger interfaces, logging frameworks or libraries provide more advanced features such as log filtering, log aggregation, log rotation, and integration with external services.

   Examples of popular logging frameworks in the .NET ecosystem include:

   - Microsoft.Extensions.Logging: A logging abstraction provided by ASP.NET Core and used by many other .NET libraries.
   - Serilog: A versatile logging library with support for structured logging and various output sinks.
   - NLog: A flexible logging library with extensive configuration options and support for many logging destinations.

6. **Usage:** Developers use the logging API to log messages throughout their application code. They specify the desired log level and provide a message to be logged.

```
ILogger logger = new MyLogger(); // Instantiate logger
logger.Log(LogLevel.Info, "Application started."); // Log a message
```

Logging APIs play a crucial role in modern software development by enabling developers to gain insights into application behaviour and diagnose issues effectively. By logging relevant information at appropriate log levels, developers can monitor application health, track down bugs, and troubleshoot problems in production environments.

Logging Providers

Logging providers are components within logging frameworks or libraries that handle the actual writing or output of log messages to various destinations. These providers allow developers to configure where log messages are sent and how they are formatted. Logging providers play a crucial role in the logging ecosystem, as they determine how log data is collected, stored, and analysed.

Here's an explanation of logging providers and their typical functionalities:

1. **Console Provider:** This provider writes log messages to the console output (stdout). It's often used during development for quick debugging and troubleshooting.

2. F**ile Provider**: The file provider writes log messages to text files on the local file system. It's commonly used in production environments for long-term storage of log data. File providers typically support features like log rotation, which helps manage the size and number of log files.

3. **Event Log Provider (Windows):** On Windows systems, the event log provider writes log messages to the Windows Event Log. This allows applications to integrate with the native Windows logging infrastructure.

4. **Database Provider:** The database provider writes log messages to a database table. This allows for centralised storage and analysis of log data using database querying and reporting tools. Database providers may support various database systems such as SQL Server, MySQL, PostgreSQL, etc.

5. **Network Provider:** The network provider sends log messages over the network to a remote logging server or service. This allows for centralised logging in distributed systems or cloud environments. Network providers typically use protocols like TCP, UDP, or HTTP for communication.

Logging providers are typically configured through the logging framework or library's configuration system, allowing developers to specify which providers to use, their settings, and any customizations required for logging behaviour. By choosing appropriate logging providers and configuring them effectively, developers can ensure that log messages are collected, stored, and analysed in a manner that meets their application's requirements.