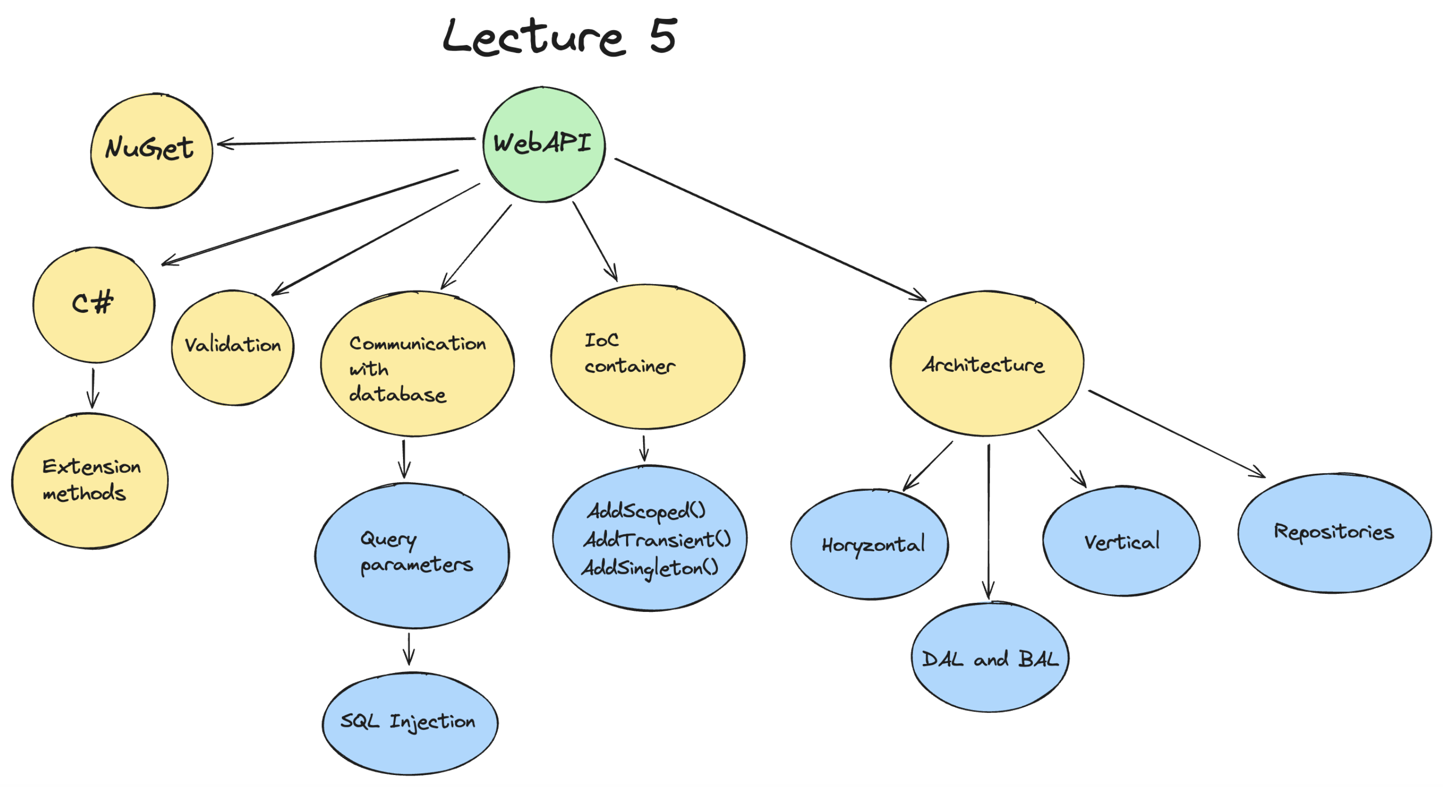


Extension method

In C#, an extension method is a special kind of static method defined in a static class that allows you to "extend" an existing type with new methods, without modifying the original type's source code or using inheritance. This feature can be particularly useful when you want to add methods to types that you do not have access to, such as types in .NET Framework or third-party libraries.

To create an extension method, you follow these steps:

1. Define a static class that will contain the extension method.
2. Create a static method within that class. The first parameter of the method specifies the type that the method extends. This parameter is preceded by the `this` modifier, indicating an extension method.

```
using System;

namespace ExtensionMethodsDemo
{
    public static class StringExtensions
    {
        // Extension method for the String class
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            string exampleSentence = "Hello, world! Welcome to
C# extension methods.";
            int count = exampleSentence.WordCount();
            Console.WriteLine($"Word count: {count}");
        }
    }
}
```

In this example, the `WordCount` method is an extension method for the `System.String` class, allowing any string instance to call `WordCount` as if it were an instance method of `String`.

Nuget

NuGet is a package manager for the Microsoft development platform, including .NET. It is the largest database of third-party components for .NET developers, allowing them to share and consume reusable code. NuGet packages contain compiled code (DLLs), along with other content needed in the projects that consume these packages.

Declarative validation in WebAPI

Declarative validation refers to the practice of using attributes or other declarative means to specify validation rules directly on model properties. This is in contrast to imperative validation, where you write explicit code to check conditions. Declarative validation makes the code cleaner, easier to read, and maintain by embedding validation logic as metadata on the models.

```
public class Product
{
    public int Id { get; set; }

    [Required]
    [StringLength(100, MinimumLength = 3)]
    public string Name { get; set; }

    [Range(0.01, 10000.00)]
    public decimal Price { get; set; }

    [Required]
    public string Category { get; set; }
}
```

In this example, the `Product` class uses Data Annotations to declare validation rules directly on its properties. For instance, the `Name` property must be a string between 3 and 100 characters and is required. The `Price` property must be a number within a specified range.

1. **`[Required]`**: Indicates that a property must have a value; it cannot be null.
2. **`[StringLength(int maximumLength, int minimumLength = 0)]`**: Specifies the maximum and optional minimum length of character strings.
3. **`[Range(int minimum, int maximum)]`** or **`[Range(double minimum, double maximum)]`**: Specifies the minimum and maximum value for a property. This attribute can apply to numeric data types.
4. **`[Compare(string otherPropertyName)]`**: Provides a way to compare two properties of a model. For example, comparing a `Password` field with a `ConfirmPassword` field.
5. **`[RegularExpression(string pattern)]`**: Specifies that a string property must match a regular expression pattern.
6. **`[EmailAddress]`**: Validates that the property has an email format.
7. **`[PhoneNumber]`**: Validates that the property has a phone number format.
8. **`[Url]`**: Validates that the property has a URL format.
9. **`[CreditCard]`**: Validates that the property has a credit card format.

10. `[MaxLength(int length)]` and `[MinLength(int length)]`: Specifies the maximum and minimum length for a property, respectively.
11. `[DataType(DataType enumeration)]`: Specifies the data type of a property. For example, `DataType.EmailAddress`, `DataType.Date`, etc. This doesn't provide validation by itself but can be used to semantically define the type of data a property holds.
12. `[Display(Name = "Display Name")]`: Used for labeling a property with a friendly name, especially useful in UI scenarios. Not a validation attribute but often used alongside them.
13. `[EnumDataType(typeof(EnumType))]`: Validates that the value of the property is one of the valid values in the specified enumeration.

Built-in IoC container

In ASP.NET Core and by extension in WebAPI, dependency injection is a first-class citizen, fully integrated into the framework.

The built-in IoC (Inversion of Control) container supports the construction and injection of dependencies at runtime, facilitating loose coupling and improving the testability and maintainability of applications. The IoC container manages the lifecycle of the objects it creates through dependency injection. The AddScoped, AddSingleton, and AddTransient methods are used to register services with the container, each dictating a different lifecycle for the service instances.

AddSingleton

When you register a service with `AddSingleton`, the IoC container will create and share a single instance of the service across all requests and throughout the application's lifetime. This is useful for services that are stateless or whose state is thread-safe and can be shared without conflicts.

```
services.AddSingleton<IMySingletonService, MySingletonService>
();
```

AddScoped

`AddScoped` registration means that a new instance of the service will be created for each client request (in web applications, this typically equates to an HTTP request). This is useful when you want to maintain a service instance state within the scope of a request but not beyond that.

```
services.AddScoped<IMyScopedService, MyScopedService>();
```

AddTransient

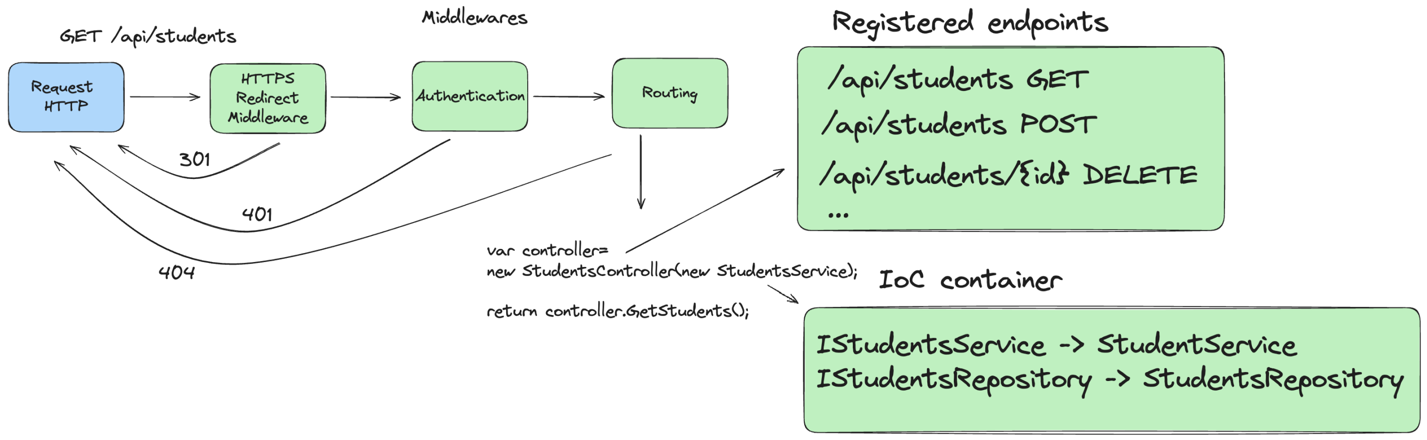
Services registered as transient with `AddTransient` will have a new instance created each time they are requested from the service container. This is appropriate for lightweight, stateless services.

```
services.AddTransient<IMyTransientService, MyTransientService>
();
```

Choosing the Right Service Lifetime

- Use **Singleton** when the service implementation doesn't maintain any state or maintains state that is fully thread-safe and can be shared across multiple requests.
- Use **Scoped** for services that maintain state within a request's scope. This is often the default choice for web applications.
- Use **Transient** for services that are lightweight, stateless, or need to be unique every time they are injected.

Using IoC in WebAPI



```
var controller=
new StudentsController(new StudentsService);

return controller.GetStudents();
```

Vertical slice vs Horizontal slice

The architectural approach of a software application significantly influences its development, maintenance, and scalability. Two distinct architectural styles in the context of ASP.NET Core and web APIs are **Vertical Slice Architecture** and **Traditional Horizontal Layered Architecture**. Each has its advantages and conceptual differences, especially when considering the development of web applications with ASP.NET Core's Minimal APIs compared to more conventional structured approaches.

Vertical slice architecture

Vertical Slice Architecture focuses on dividing the application into vertical slices, where each slice represents a specific feature or use case of the application.

This approach contrasts with the traditional horizontal layering by not segregating the application into layers based on technical concerns (like UI, business logic, data access) but instead organizing it around functional concerns.

Each slice contains all the necessary components—from the UI down to the data access—needed to implement a specific functionality or feature. This approach aims to make features more self-contained, thereby reducing dependencies across the application and simplifying scaling and maintenance.

Horizontal slice architecture

The Horizontal Layered Architecture, also known as the n-tier architecture, divides an application into separate layers, each responsible for a specific aspect of the application. Common layers include the Presentation Layer (UI), Business Logic Layer (BLL), and Data Access Layer (DAL). This separation of concerns facilitates a clean organization of code and responsibilities. However, it often means that changes to a single feature can require modifications across multiple layers.

Vertical vs Horizontal architecture

- **Use Vertical Slice Architecture** when you aim for a highly modular application where features are developed, deployed, and scaled independently. It's particularly well-suited for agile development processes and microservices architectures.
- **Use Horizontal Layered Architecture** in scenarios where the application's domain complexity and business rules are expected to be stable and reusability across different parts of the application is a priority. This approach might be more familiar and easier to manage for teams accustomed to traditional enterprise application development.

Repositories

The Repository Pattern is a design pattern widely used in software development, particularly within the context of domain-driven design and data access layer (DAL) management.

Its primary goal is to decouple the business logic of an application from the data access logic, promoting a cleaner separation of concerns. By abstracting data access behind a well-defined interface, the Repository Pattern makes the application's business logic agnostic to the specifics of the data source (which could be a database, a web service, or even a flat file).

A typical implementation involves defining a repository interface with methods for performing CRUD (Create, Read, Update, Delete) operations and other data retrieval actions. Concrete classes then implement this interface, encapsulating the logic required to interact with the data source.

```
public interface IProductRepository
{
    Product GetById(int id);
    IEnumerable<Product> GetAll();
    void Add(Product product);
    void Update(Product product);
    void Delete(int id);
}
```

SqlConnection and SqlCommand

To communicate with Microsoft SQL Server in a .NET application, you typically use the SqlConnection and SqlCommand classes provided by the System.Data.SqlClient namespace (or Microsoft.Data.SqlClient for newer projects).

Establishing a Connection

First, you need to create a connection to the database using an `SqlConnection` object. This requires a connection string that specifies the location of the database, credentials, and other options.

```
string connectionString =
"Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
using (SqlConnection connection = new
SqlConnection(connectionString))
{
    connection.Open();
    // Use the connection
}
```

Executing a Command with Parameters

You can execute SQL queries, stored procedures, or any SQL command using the `SqlCommand` object. Parameters can be added to the command to protect against SQL injection and handle data dynamically.

```
string query = "SELECT * FROM Products WHERE CategoryID =
@categoryID";
using (SqlCommand command = new SqlCommand(query, connection))
{
    // Adding a parameter to the command
    command.Parameters.AddWithValue("@categoryID", 1);

    // Execute the command
    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read())
        {
            // Read your results here
        }
    }
}
```

Passing Parameters

Passing parameters to your SQL command helps prevent SQL injection and makes your application more secure and robust. Parameters are specified in the SQL command text with a prefixed `@`, and their values are assigned by adding `SqlParameter` objects to the `SqlCommand.Parameters` collection.

```
command.Parameters.AddWithValue("@categoryID", yourVariable);
```

Reading Results

After executing a query that returns results (e.g., a `SELECT` statement), you can read the results using an `SqlDataReader`.

```
using (SqlDataReader reader = command.ExecuteReader())
{
    while (reader.Read())
    {
        // Access your field by name or index, e.g.,
reader["ColumnName"] or reader.GetInt32(0)
        string productName = reader["ProductName"].ToString();
        // Process each row accordingly
    }
}
```

Putting It All Together

```
string connectionString =
"Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
string query = "SELECT * FROM Products WHERE CategoryID =
@categoryID";
```

```
using (SqlConnection connection = new
SqlConnection(connectionString))
{
    connection.Open();
    using (SqlCommand command = new SqlCommand(query,
connection))
    {
        command.Parameters.AddWithValue("@categoryID", 1); //
Assuming you're looking for CategoryID 1
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                string productName =
reader["ProductName"].ToString();
                // Output or process your data here
            }
        }
    }
}
```

This example demonstrates how to open a connection to a SQL Server database, execute a parameterized query to select data, and read through the results with an `SqlDataReader`. Remember to install the necessary NuGet package (`System.Data.SqlClient` or `Microsoft.Data.SqlClient`) to use these classes in your project.