UNITED STATES

MICROSOFT ARCHITECT
By Joydip Kanjilal, Columnist, InfoWorld
JUN 15, 2020 3:00 AM PDT

How to use Data Transfer Objects in ASP.NET Core 3.1

Learn the benefits of Data Transfer Objects, why they should be immutable, and how to take advantage of them in your ASP.NET Core applications

A Data Transfer Object (commonly known as a DTO) is usually an instance of a POCO (plain old CLR object) class used as a container to encapsulate data and pass it from one layer of the application to another. You would typically find DTOs being used in the service layer to return data back to the presentation layer. The biggest advantage of using DTOs is decoupling clients from your internal data structures.

This article discusses why we should use Data Transfer Objects and how we can work with them in ASP.NET Core 3.1. To work with the code examples provided in this article, you should have Visual Studio 2019 installed in your system. If you don't already have a copy, you can download Visual Studio 2019 here.

[Also on InfoWorld: The best free data science courses during quarantine]

Create an ASP.NET Core 3.1 API project

First off, let's create an ASP.NET Core project in Visual Studio. Assuming Visual Studio 2019 is installed in your system, follow the steps outlined below to create a new ASP.NET Core API project in Visual Studio.

1. Launch the Visual Studio IDE.
2. Click on "Create new project."

3. In the “Create new project” window, select “ASP.NET Core Web Application” from the list of the templates displayed.
4. Click Next.
5. In the “Configure your new project” window, specify the name and location for the new project.
6. Click Create.
7. In the “Create New ASP.NET Core Web Application” window shown next, select .NET Core as the runtime and ASP.NET Core 3.1 (or later) from the drop-down list at the top.
8. Select “API” as the project template to create a new ASP.NET Core API application.
9. Ensure that the check boxes “Enable Docker Support” and “Configure for HTTPS” are unchecked as we won’t be using those features here.
10. Ensure that Authentication is set as “No Authentication” as we won’t be using authentication either.
11. Click Create.

This will create a new ASP.NET Core API project in Visual Studio. We’ll use this project to work with Data Transfer Objects in the subsequent sections of this article.

Why use Data Transfer Objects (DTOs)?

When designing and developing an application, if you're using models to pass data between the layers and sending data back to the presentation layer, then you're exposing the internal data structures of your application. That's a major design flaw in your application.

By decoupling your layers DTOs make life easier when you're implementing APIs, MVC applications, and also messaging patterns such as Message Broker. A DTO is a great choice when you would like to pass a lightweight object across the wire — especially when you're passing your object via a medium that is bandwidth-constrained.

Use DTOs for abstraction

You can take advantage of DTOs to abstract the domain objects of your application from the user interface or the presentation layer. In doing so, the presentation layer of your application is decoupled from the service layer. So if you would like to change the presentation layer, you can do that easily while the application will continue to work with the existing domain layer. Similarly, you can change the domain layer of your application without having to change the presentation layer of the application.

Use DTOs for data hiding

Another reason you would want to use DTOs is data hiding. That is, by using DTOs you can return only the data requested. As an example, assume you have a method named GetAllEmployees() that returns all the data pertaining to all employees. Let's illustrate this by writing some code.

In the project we created earlier, create a new file called Employee.cs. Write the following code inside this file to define a model class named Employee.

```
public class Employee
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string DepartmentName { get; set; }
        public decimal Basic { get; set; }
        public decimal DA { get; set; }
        public decimal HRA { get; set; }
        public decimal NetSalary { get; set; }
    }
```

Note the Employee class contains properties including Id, FirstName, LastName, Department, Basic, DA, HRA, and NetSalary. However, the presentation layer might only need the Id, FirstName, LastName, and Department Name of the employees from the GetAllEmployees() method. If this method returns a List<Employee> then anyone would be able to see the salary details of an employee. You don't want that.

To avoid this problem, you might design a DTO class named EmployeeDTO that would contain only the properties that are requested (such as Id, FirstName, LastName, and Department Name).

Create a DTO class in C#

To achieve this, create a file named EmployeeDTO.cs and write the following code in there.

```
public class EmployeeDTO
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string DepartmentName { get; set; }
    }
```

Now that the model and data transfer object classes are available, you might want to create a converter class that contains two methods: one to convert an instance of the Employee model class to an instance of EmployeeDTO and (vice versa) one to convert an instance of EmployeeDTO to an instance of the Employee model class. You might also take advantage of AutoMapper, a popular object-to-object mapping library to map these two dissimilar types. You can read more about AutoMapper here.

You should create a List<EmployeeDTO> in the service layer of your application and return the collection back to the presentation layer.

Immutability of DTOs

A DTO is meant to transport data from one layer of an application to another layer. The consumer of a DTO might be built in .NET/C#/Java or even JavaScript/TypeScript. A DTO is often serialized so that it can be independent of the technology used in the receiver. In most cases, the receiver of the data does not need to modify that data after receipt — ideally it shouldn't!

This is a classic example of the importance of immutability. And it's exactly why a DTO should be immutable!

There are several ways in which you can implement immutable DTOs in C#. You could use a ReadOnlyCollection or the thread-safe immutable collection types present in the System.Collections.Immutable namespace. You can take advantage of record types in C# 9 to implement immutable DTOs as well.

Domain-driven design expects the domain objects to be externally immutable. This is a good reason to make your DTOs immutable, isn't it?

DTO serialization challenges

You should be able to serialize/deserialize a DTO seamlessly so that it can be passed down the wire. In practice, however, you might have to solve some serialization problems when working with DTOs. You might have several entities or model classes in a real-world application and each of them may hold references to each other.

Let's say you have built an attendance management system for the employees in your organization. Typically, you might have a class called Employee in your application that references the User class (i.e., an Employee is a user of the application) which in turn references the Role class. The Role class might reference the Permission class which in turn might reference the PermissionType and PermissionGroup classes. Now, when you serialize an instance of the Employee class, you will end up serializing these objects as well. It's easy to see that, in some complicated cases, you might end up serializing several types.

This is where lazy loading or asynchronous loading comes to the rescue. This is a feature that can help you load entities only when asked for. For more information on how to perform lazy loading, you can take a look at my article on lazy initialization in C#.

Data Transfer Objects typically don't contain any business logic — they only contain data. Immutability is a desired feature when working with DTOs. There are several ways in which you can implement immutable DTOs. I'll discuss more on immutability in C# in a later post here.

How to do more in ASP.NET Core:

- How to handle 404 errors in ASP.NET Core MVC
- How to use dependency injection in action filters in ASP.NET Core 3.1
- How to use the options pattern in ASP.NET Core
- How to use endpoint routing in ASP.NET Core 3.0 MVC
- How to export data to Excel in ASP.NET Core 3.0
- How to use LoggerMessage in ASP.NET Core 3.0
- How to send emails in ASP.NET Core
- How to log data to SQL Server in ASP.NET Core
- How to schedule jobs using Quartz.NET in ASP.NET Core
- How to return data from ASP.NET Core Web API
- How to format response data in ASP.NET Core
- How to consume an ASP.NET Core Web API using RestSharp

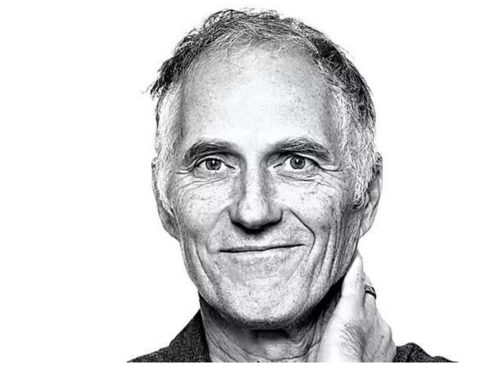
- How to perform async operations using Dapper
- How to use feature flags in ASP.NET Core
- How to use the FromServices attribute in ASP.NET Core
- How to work with cookies in ASP.NET Core
- How to work with static files in ASP.NET Core
- How to use URL Rewriting Middleware in ASP.NET Core
- How to implement rate limiting in ASP.NET Core
- How to use Azure Application Insights in ASP.NET Core
- Using advanced NLog features in ASP.NET Core
- How to handle errors in ASP.NET Web API
- How to implement global exception handling in ASP.NET Core MVC
- How to handle null values in ASP.NET Core MVC
- Advanced versioning in ASP.NET Core Web API
- How to work with worker services in ASP.NET Core
- How to use the Data Protection API in ASP.NET Core
- How to use conditional middleware in ASP.NET Core
- How to work with session state in ASP.NET Core
- How to write efficient controllers in ASP.NET Core

Joydip Kanjilal is a Microsoft MVP in ASP.Net, as well as a speaker and author of several books and articles. He has more than 20 years of experience in IT including more than 16 years in Microsoft .Net and related technologies.

Follow

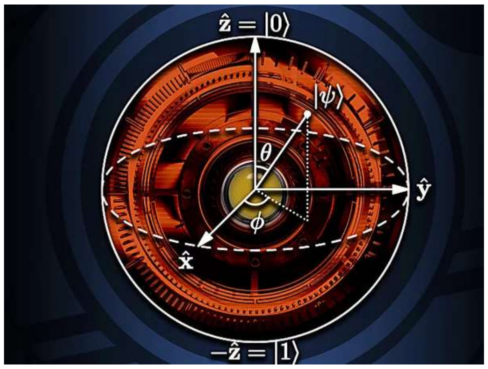
YOU MAY ALSO LIKE

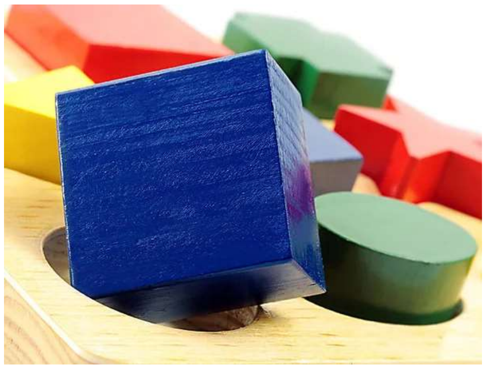
Recommended by


Tim O'Reilly: the golden age of the programmer is over

Edge computing archetypes are emerging, and they are not pretty

Deno Company forms to back Node.js rival

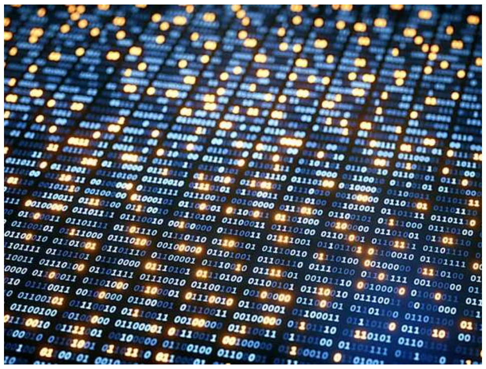

Amazon Braket: Get started with quantum computing


When Kubernetes is not the solution

Tapping into the smartest software developers

The decline of Heroku

Red Hat OpenShift ramps up security and


What is a computational storage drive? Much-

The 24 highest paying developer roles in 2020

The best open source software of 2020