MICROSOFT ARCHITECT
By Joydip Kanjilal, Columnist, InfoWorld
MAY 11, 2020 3:00 AM PDT

How to use the options pattern in ASP.NET Core

Take advantage of the options pattern to load configuration data directly into your classes at runtime for simple and flexible configuration data management

When working in ASP.NET Core you will often specify your application's settings, store them in some file, and then retrieve these settings when the application needs them. Typically, you would register your dependencies in the ConfigureServices method of the Startup class. You can specify your application's settings in the appsettings.json or some other .json file and then take advantage of dependency injection through IOptions<T> to read these settings in your application.

The options patterns provide an elegant way to add strongly typed settings to your ASP.NET Core application. The options pattern, which is an extension on top of the IServiceCollection interface, takes advantage of classes to represent a group of related settings. This article talks about the options pattern, why it is useful, and how it can be used for working with configuration data in ASP.NET Core.

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

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.

Create an ASP.NET Core 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 templates displayed.
4. Click Next.
5. In the "Configure your new project" window shown next, specify the name and location for the new project.
6. Click Create.
7. In the "Create New ASP.NET Core Web Application" window, select .NET Core as the runtime and ASP.NET Core 3.0 (or later) from the drop-down list at the top. I'll be using ASP.NET Core 3.1 here.
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. Select the Controllers solution folder in the Solution Explorer window and click "Add -> Controller..." to create a new controller named DefaultController. We'll use this project in the subsequent sections of this article.

Implement the options pattern in ASP.NET Core

To use the options pattern in ASP.NET Core, you need the Microsoft.Extensions.Options.ConfigurationExtensions package. Incidentally, the ASP.NET Core applications implicitly references the

Microsoft.Extensions.Options.ConfigurationExtensions package by default.

When using the options pattern, you would typically want to use classes to represent a group of related settings. In isolating the configuration settings into separate classes, your application adheres to the following principles:

- Separation of concerns: The settings used in different modules of the application are decoupled from one another.
- Interface segregation principle: The classes that represent these settings depend only on the configuration settings that they would use.

Now write the following settings in the appsettings.json file.

```
"DatabaseSettings": {
    "Server": "localhost",
    "Provider": "SQL Server",
    "Database": "DemoDb",
    "Port": 23,
    "UserName": "sa",
    "Password": "Joydip123"
  }
```

Note that your configuration class should have public get and set properties. We'll take advantage of the following class to read these settings shortly.

```
public class DatabaseSettings
   {
       public string Server { get; set; }
       public string Provider { get; set; }
       public string Database { get; set; }
       public int Port { get; set; }
       public string UserName { get; set; }
       public string Password { get; set; }
   }
```

You can now use the Configure extension method of IServiceCollection to bind your settings class to your configuration as shown in the code snippet given below.

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddControllers();
   services.Configure<DatabaseSettings>
   (options => Configuration.GetSection("DatabaseSettings").Bind(options));
}
```

Read configuration data in the controller in ASP.NET Core

We'll now take advantage of the DefaultController we created earlier to demonstrate how we can read configuration data in the controller. The IOptions<T> interface exposes a Value property that can be used to retrieve the instance of the settings class.

The following code snippet shows how you can use the DatabaseSettings class in your controller named DefaultController. Note how dependency injection (constructor injection in this example) has been used here.

```
public class DefaultController : ControllerBase
{
   private DatabaseSettings _settings;
   public DefaultController(IOptions<DatabaseSettings> settings)
   {
      _settings = settings.Value;
   }
   //Action methods
}
```

Enforce rules for configurations in ASP.NET Core

You can also enforce certain rules as shown in the code snippet below. Note how an instance of the helper class for SQL Server or MySQL is being added as a singleton here.

```
services.Configure<DatabaseSettings>(options =>
 {
     if (options.Provider.ToLower().Trim().Equals("sqlserver"))
     {
        services.AddSingleton(new SqlDbHelper());
     }
     else if(options.Provider.ToLower().Trim().Equals("mysql"))
     {
         services.AddSingleton(new MySqlDbHelper());
     }
 });
```

Support for strongly typed configuration is a great feature in ASP.NET Core that enables you to apply the separation of concerns and interface segregation principles. In a future post here on the options pattern, I'll talk about configuration validation and reloadable configuration with a special focus on the IOptionsMonitor<TOptions> interface. Until then, you can read more about the options pattern in Microsoft's online documentation here.

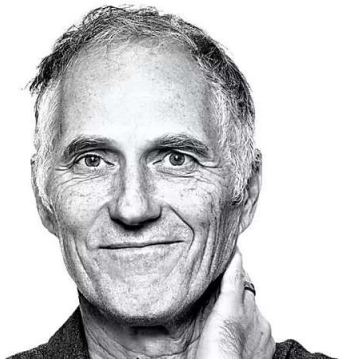
How to do more in ASP.NET and ASP.NET Core:

- How to use in-memory caching in ASP.NET Core
- How to handle errors in ASP.NET Web API
- How to pass multiple parameters to Web API controller methods
- How to log request and response metadata in ASP.NET Web API
- How to work with HttpModules in ASP.NET
- Advanced versioning in ASP.NET Core Web API
- How to use dependency injection in ASP.NET Core
- How to work with sessions in ASP.NET
- How to work with HTTPHandlers in ASP.NET
- How to use IHostedService in ASP.NET Core
- How to consume a WCF SOAP service in ASP.NET Core
- How to improve the performance of ASP.NET Core applications

- How to consume an ASP.NET Core Web API using RestSharp
- How to work with logging in ASP.NET Core
- How to use MediatR in ASP.NET Core
- How to work with session state in ASP.NET Core
- How to use Nancy in ASP.NET Core
- Understand parameter binding in ASP.NET Web API
- How to upload files in ASP.NET Core MVC
- How to implement global exception handling in ASP.NET Core Web API
- How to implement health checks in ASP.NET Core
- Best practices in caching in ASP.NET
- How to use Apache Kafka messaging in .NET
- How to enable CORS on your Web API
- When to use WebClient vs. HttpClient vs. HttpWebRequest
- How to work with Redis Cache in .NET
- When to use Task.WaitAll vs. Task.WhenAll in .NET

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



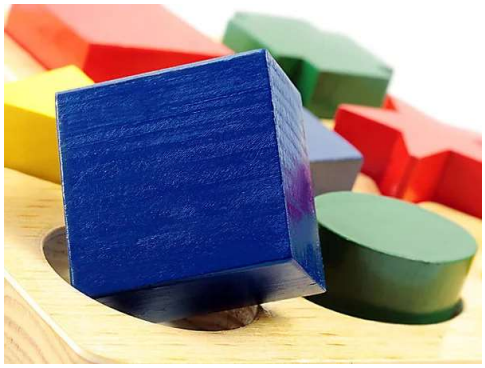
Tim O'Reilly: the golden age of the programmer is over

The 24 highest paying developer roles in 2020

Edge computing archetypes are emerging, and they are not pretty

8 databases supporting in-database machine learning

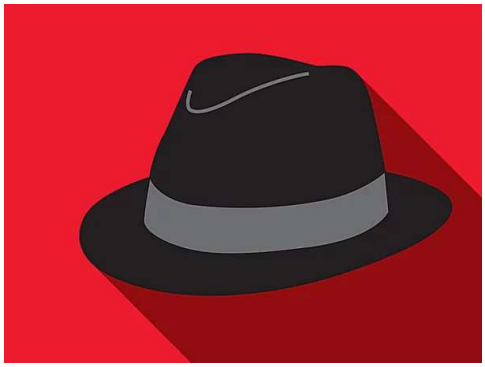
3 enterprise AI success stories



When Kubernetes is not the solution

JetBrains IntelliJ IDE debuts ‘run targets’

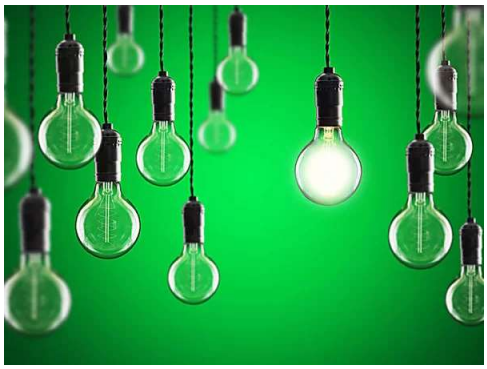
Are industry clouds an opportunity or a



Red Hat OpenShift ramps up security and

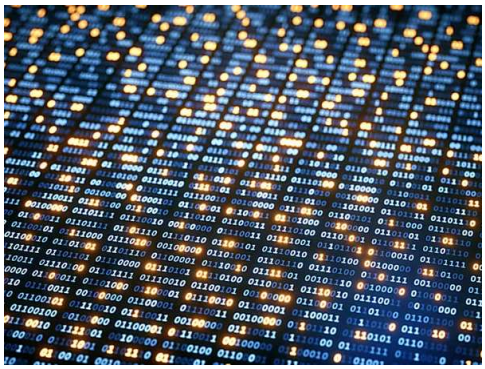
When to use Task.WaitAll vs. Task.WhenAll in .NET

How IT priorities are shifting during the COVID-19 crisis

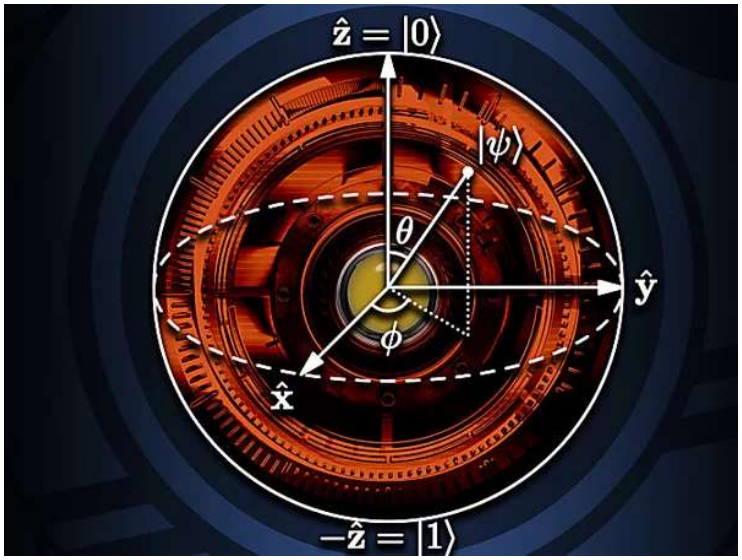


10 tips for tuning React UI performance

The decline of Heroku



What is a computational storage drive? Much-



Amazon Braket: Get started with quantum computing

Microsoft Visual Studio 2022 preview is coming soon