Introduction

ASP.NET Core supports many methods of configuration. In ASP.NET Core application, the configuration is stored in name-value pairs and it can be read at runtime from various parts of the application. The name-value pairs may be grouped into a multi-level hierarchy. The application configuration data may come from

* File, such as JSON, XML, INI
* Environment variables
* Command Line arguments
* An in-memory collection
* Custom providers

Configuration System in ASP.NET Core is restructured from the older version of ASP.NET. The older version uses "System.Configuration" namespace and is able to read XML configuration files such as web.config. The new configuration model can be accessed to the key/value-based settings and it can retrieve various sources, such as JSON, XML, and INI.

Example of reading configuration using JSON provider

Generally, the configuration remains in simple key/value structure or might in hierarchical structure when using external files. Consider the following appsetting.json file for this example.

**appSetting.json**

1. {
2. "status" : "This Status read from appSettings.json file",
3. "ConnectionStrings": {
4. "DefaultConnection": "Server=.\\sqlexpress;Database=Test;Trusted\_Connection=True;MultipleActiveResultSets=true"
5. }
6. }

To read the JSON configuration file, we need to add "Microsoft.Extensions.Configuration" and "Microsoft.Extensions.Configuration.Json" libraries as a dependency. Following two lines need to be added in package.json file in dependency section.

1. "Microsoft.Extensions.Configuration":"1.0.0",
2. "Microsoft.Extensions.Configuration.Json":"1.0.0"

The "AddJsonFile" method of "JsonConfigurationExtensions" class is used to JSON file to the builder. We can get a simple configuration value by using key name. If the configuration value is in a hierarchical structure, it can be retrieved using a ":" separated key, starting from root of the hierarchy. In this example, if we want to get value for "DefaultConnection", then the key becomes "ConnectionStrings:DefaultConnection".

**Starpup.cs**

1. **using** Microsoft.AspNetCore.Builder;
2. **using** Microsoft.AspNetCore.Http;
3. **using** Microsoft.Extensions.Configuration;
4. **using** Microsoft.Extensions.DependencyInjection;
6. **namespace** WebApplication
7. {
8. **public** **class** Startup {
9. **public** IConfiguration Configuration { **get**; **set**; }
10. **public** Startup() {
11. var builder = **new** ConfigurationBuilder()
12. .AddJsonFile("appSettings.json");
13. Configuration = builder.Build();
14. }
15. **public** **void** ConfigureServices(IServiceCollection services)
16. {
17. services.AddMvc();
18. }
19. **public** **void** Configure(IApplicationBuilder app){
20. app.UseMvc();
21. app.Run(context => {
22. var status = Configuration["status"];
23. var connectionString = Configuration["ConnectionStrings:DefaultConnection"];
24. context.Response.WriteAsync("Default Connection: " + connectionString);
25. context.Response.WriteAsync("<br/>");
26. **return** context.Response.WriteAsync("Status: "+status);
27. });
28. }
29. }
30. }

**Output**



Retrieve Configuration Data at the controller

In the above-mentioned example, I am retrieving the configuration value in the startup class itself. The new version of ASP.NET has built-in support for dependency injection. Using DI, we can inject the value of the configuration to the Controller.

Using AddSingleton method of ServiceCollectionServiceExtensions class, we can add a singleton service of the specified type with an instance of service.

**Startup.cs**

1. **public** **void** ConfigureServices(IServiceCollection services)
2. {
3. services.AddMvc();
4. services.AddSingleton<IConfiguration>(Configuration);
5. }

**HomeController.cs**

1. **using** Microsoft.AspNetCore.Mvc;
2. **using** Microsoft.Extensions.Configuration;
4. **public** **class** HomeController : Controller
5. {
6. IConfiguration \_configuration;
7. **public** HomeController(IConfiguration configuration)
8. {
9. \_configuration = configuration;
10. }
11. [Route("home/index")]
12. **public** IActionResult Index()
13. {
14. ViewBag.connectionstring = \_configuration["ConnectionStrings:DefaultConnection"];
15. **return** View();
16. }
17. }

**Output**



Get Configuration object using options pattern

The Options pattern enables us to use custom option classes to represent a group of related settings. The Option class must have a public read-write property that is defined for each setting and the class must not take any parameters.

The “Microsoft.Extensions.Options.ConfigurationExtensions” dependency contains the extension method for IServiceCollection.Configure. We need to add this dependency in project.json file.

1. "Microsoft.Extensions.Options.ConfigurationExtensions": "1.0.0"

Here, I have defined my model and I am using same appSetting.json that was used in previous example. In this example, I have bound the connection string setting of appSetting.json with my model class.

1. **public** **class** ConnectionString
2. {
3. **public** ConnectionString()
4. {
5. }
6. **public** **string** DefaultConnection { **get**; **set**; }
7. **public** **string** MainDBConnectionString { **get**; **set**; }
8. }

In the following code, IConfigureOptions<TOptions> service is added to the Service Container. It binds ConnectionStrings class to the section "ConnectionStrings" of the appsettings.json file.

**Startup.cs**

1. **public** **void** ConfigureServices(IServiceCollection services)
2. {
4. services.AddOptions();
6. //Configure Option using Extensions method
7. services.Configure<ConnectionString>(Configuration.GetSection("ConnectionStrings"));
9. services.AddSingleton<IConfiguration>(Configuration);
10. services.AddMvc();
11. }

Using IOptions<TOptions> accessor service, we can inject Options into our application. To setup the IOptions<TOptions> service, we need to call "AddOptions" extension method during startup in our ConfigureServices method of Startup class.

**HomeController.cs**

1. **using** Microsoft.AspNetCore.Mvc;
2. **using** Microsoft.Extensions.Options;
4. **public** **class** HomeController : Controller
5. {
6. ConnectionString \_connectionstring;
7. **public** HomeController(IOptions<ConnectionString> connectionstringAccessor)
8. {
9. \_connectionstring = connectionstringAccessor.Value;
10. }
11. [Route("home/index")]
12. **public** IActionResult Index()
13. {
14. ViewBag.connectionstring = \_connectionstring.DefaultConnection;
15. **return** View();
16. }
17. }

**Output**



Working with In-memory provider

The new configuration framework of ASP.NET Core does also supports in-memory configuration. In this type of configuration, the values are directly stored into the code and the later part of the application uses this configuration. Following is a sample that shows how to use the in-memory provider and bind to a class.

1. **public** Startup()
2. {
3. var builder = **new** ConfigurationBuilder();
5. var dic = **new** Dictionary<**string**, **string**>
6. {
7. {"Profile:FirstName", "Jignesh"},
8. {"Profile:LastName", "Trivedi"},
9. {"Profile:Designation", "PL"}
10. };
11. builder.AddInMemoryCollection(dic);
12. Configuration = builder.Build();
13. }

Using Configuration["Profile:FirstName"], we can get the value of firstname of profile configuration. We can also bind this value with custom model. Her,e I have created the following “Profile” class to bind this profile value.

1. **public** **class** Profile
2. {
3. **public** **string** FirstName { **get**; **set**;}
4. **public** **string** LastName { **get**; **set**;}
5. **public** **string** Designation { **get**; **set**;}
6. }

The following sample shows how to bind to a profile and use the options pattern with ASP.NET Core MVC application.

**Startup.cs**

1. **public** **void** ConfigureServices(IServiceCollection services)
2. {
3. services.AddOptions();
4. services.Configure<Profile>(Configuration.GetSection("Profile"));
5. services.AddMvc();
6. }

**HomeController.cs**

1. **using** Microsoft.AspNetCore.Mvc;
2. **using** Microsoft.Extensions.Configuration;
3. **using** Microsoft.Extensions.Options;
5. **public** **class** HomeController : Controller
6. {
7. Profile \_profile;
8. **public** HomeController(IOptions<Profile> Profile)
9. {
10. \_profile = Profile.Value;
11. }
12. [Route("home/index")]
13. **public** IActionResult Index()
14. {
15. ViewBag.FirstName = \_profile.FirstName;
16. **return** View();
17. }
18. }