What's new in ASP.NET Core 5.0

Article • 11/02/2023

This article highlights the most significant changes in ASP.NET Core 5.0 with links to relevant documentation.

ASP.NET Core MVC and Razor improvements

Model binding DateTime as UTC

Model binding now supports binding UTC time strings to `DateTime`. If the request contains a UTC time string, model binding binds it to a UTC `DateTime`. For example, the following time string is bound the UTC `DateTime`:

`https://example.com/mycontroller/myaction?time=2019-06-14T02%3A30%3A04.0576719Z`

Model binding and validation with C# 9 record types

C# 9 record types can be used with model binding in an MVC controller or a Razor Page. Record types are a good way to model data being transmitted over the network.

For example, the following `PersonController` uses the `Person` record type with model binding and form validation:

C#

```
public record Person([Required] string Name, [Range(0, 150)] int
Age);

public class PersonController
{
   public IActionResult Index() => View();

   [HttpPost]
   public IActionResult Index(Person person)
   {
          // ...
   }
}
```

The `Person/Index.cshtml` file:

CSHTML

```
@model Person

Name: <input asp-for="Model.Name" />
<span asp-validation-for="Model.Name" />

Age: <input asp-for="Model.Age" />
<span asp-validation-for="Model.Age" />
```

Improvements to DynamicRouteValueTransformer

ASP.NET Core 3.1 introduced DynamicRouteValueTransformer as a way to use custom endpoint to dynamically select an MVC controller action or a Razor page. ASP.NET Core 5.0 apps can pass state to a `DynamicRouteValueTransformer` and filter the set of endpoints chosen.

Miscellaneous

- The [Compare] attribute can be applied to properties on a Razor Page model.
- Parameters and properties bound from the body are considered required by default.

Web API

OpenAPI Specification on by default

OpenAPI Specification is an industry standard for describing HTTP APIs and integrating them into complex business processes or with third parties. OpenAPI is widely supported by all cloud providers and many API registries. Apps that emit OpenAPI documents from web APIs have a variety of new opportunities in which those APIs can be used. In partnership with the maintainers of the open-source project Swashbuckle.AspNetCore, the ASP.NET Core API template contains a NuGet dependency on Swashbuckle. Swashbuckle is a popular open-source NuGet package that emits OpenAPI documents dynamically. Swashbuckle does this by introspecting over the API controllers and generating the OpenAPI document at run-time, or at build time using the Swashbuckle CLI.

In ASP.NET Core 5.0, the web API templates enable the OpenAPI support by default. To disable OpenAPI:

- From the command line:

  .NET CLI

  ```
  dotnet new webapi --no-openapi true
  ```

- From Visual Studio: Uncheck **Enable OpenAPI support**.

All `.csproj` files created for web API projects contain the Swashbuckle.AspNetCore NuGet package reference.

XML

```
<ItemGroup>
    <PackageReference Include="Swashbuckle.AspNetCore"
Version="5.5.1" />
</ItemGroup>
```

The template generated code contains code in `Startup.ConfigureServices` that activates OpenAPI document generation:

C#

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "WebApp1",
Version = "v1" });
    });
}
```

The `Startup.Configure` method adds the Swashbuckle middleware, which enables the:

- Document generation process.
- Swagger UI page by default in development mode.

The template generated code won't accidentally expose the API's description when publishing to production.

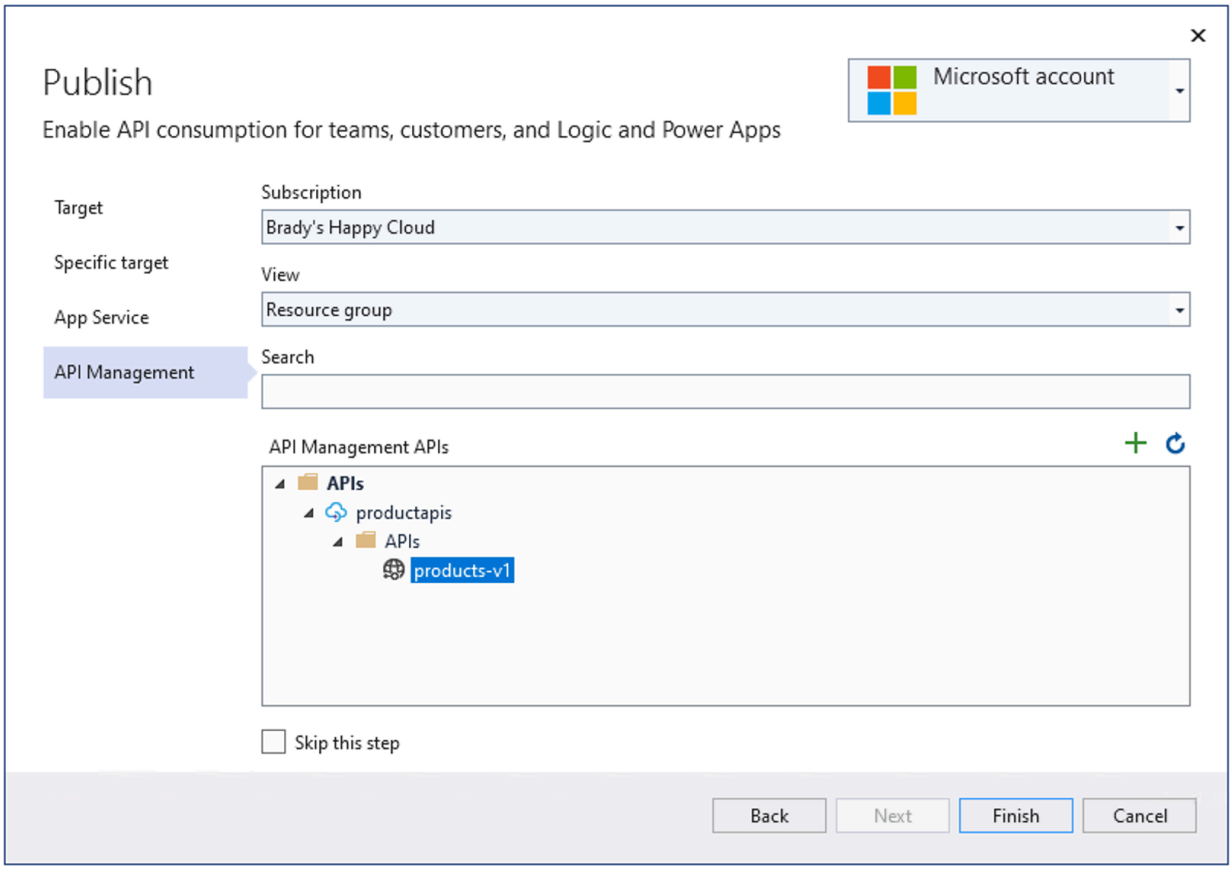
C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{
```

```
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();  // UseSwaggerUI Protected by if (env.IsDe-
velopment())
        app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swag-
ger.json",
                         "WebApp1 v1"));
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

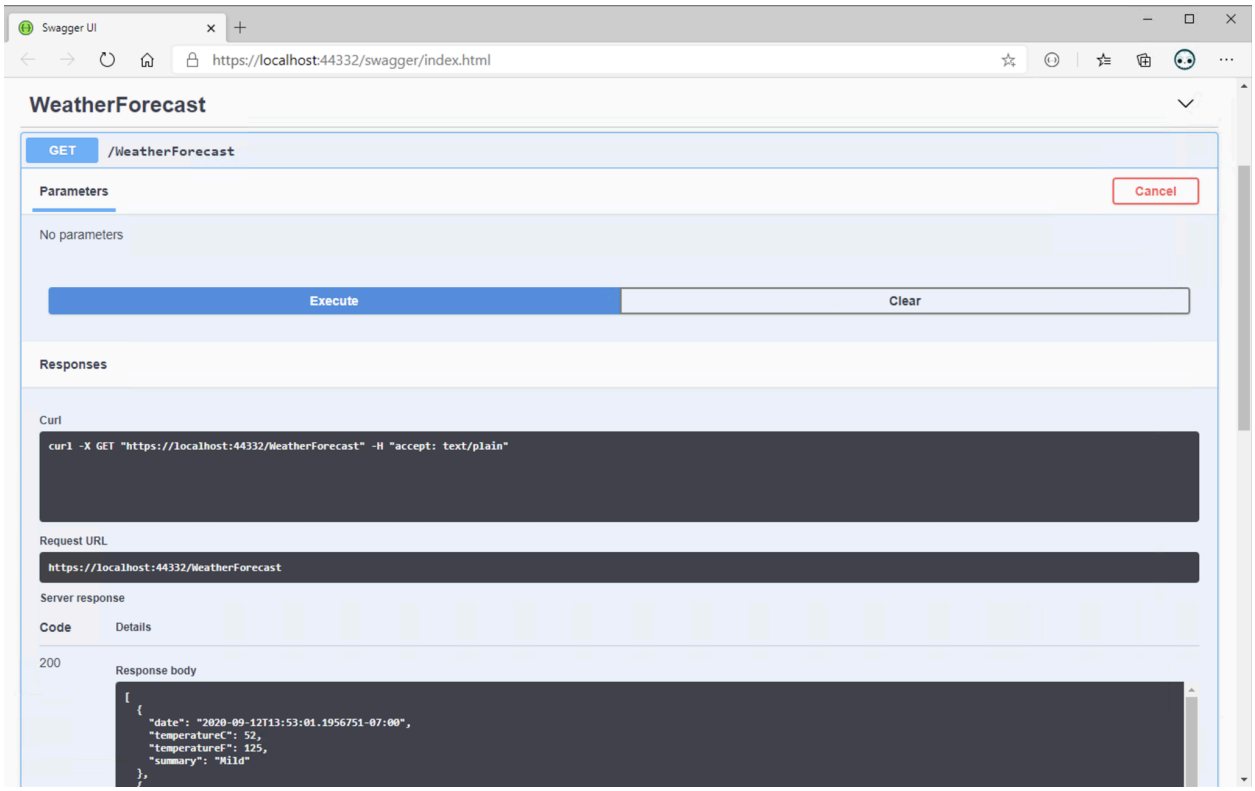
Azure API Management Import

When ASP.NET Core API projects enable OpenAPI, the Visual Studio 2019 version 16.8 and later publishing automatically offer an additional step in the publishing flow. Developers who use Azure API Management have an opportunity to automatically import the APIs into Azure API Management during the publish flow:



Better launch experience for web API projects

With OpenAPI enabled by default, the app launching experience (F5) for web API developers significantly improves. With ASP.NET Core 5.0, the web API template comes pre-configured to load up the Swagger UI page. The Swagger UI page provides both the documentation added for the published API, and enables testing the APIs with a single click.



Blazor

Performance improvements

For .NET 5, we made significant improvements to Blazor WebAssembly runtime performance with a specific focus on complex UI rendering and JSON serialization. In our performance tests, Blazor WebAssembly in .NET 5 is two to three times faster for most scenarios. For more information, see ASP.NET Blog: ASP.NET Core updates in .NET 5 Release Candidate 1.

CSS isolation

Blazor now supports defining CSS styles that are scoped to a given component. Component-specific CSS styles make it easier to reason about the styles in an app and to avoid unintentional side effects of global styles. For more information, see ASP.NET Core Blazor CSS isolation.

New `InputFile` component

The `InputFile` component allows reading one or more files selected by a user for upload. For more information, see ASP.NET Core Blazor file uploads.

New `InputRadio` and `InputRadioGroup` components

Blazor has built-in `InputRadio` and `InputRadioGroup` components that simplify data binding to radio button groups with integrated validation. For more information, see ASP.NET Core Blazor input components.

Component virtualization

Improve the perceived performance of component rendering using the Blazor framework's built-in virtualization support. For more information, see ASP.NET Core Razor component virtualization.

`ontoggle` event support

Blazor events now support the `ontoggle` DOM event. For more information, see ASP.NET Core Blazor event handling.

Set UI focus in Blazor apps

Use the `FocusAsync` convenience method on element references to set the UI focus to that element. For more information, see ASP.NET Core Blazor event handling.

Custom validation CSS class attributes

Custom validation CSS class attributes are useful when integrating with CSS frameworks, such as Bootstrap. For more information, see ASP.NET Core Blazor forms validation.

IAsyncDisposable support

Razor components now support the IAsyncDisposable interface for the asynchronous release of allocated resources.

JavaScript isolation and object references

Blazor enables JavaScript isolation in standard JavaScript modules. For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Form components support display name

The following built-in components support display names with the `DisplayName` parameter:

- `InputDate`
- `InputNumber`
- `InputSelect`

For more information, see ASP.NET Core Blazor forms overview.

Catch-all route parameters

Catch-all route parameters, which capture paths across multiple folder boundaries, are supported in components. For more information, see ASP.NET Core Blazor routing and navigation.

Debugging improvements

Debugging Blazor WebAssembly apps is improved in ASP.NET Core 5.0. Additionally, debugging is now supported on Visual Studio for Mac. For more information, see Debug ASP.NET Core Blazor apps.

Microsoft Identity v2.0 and MSAL v2.0

Blazor security now uses Microsoft Identity v2.0 (Microsoft.Identity.Web and Microsoft.Identity.Web.UI) and MSAL v2.0. For more information, see the topics in the Blazor Security and Identity node.

Protected Browser Storage for Blazor Server

Blazor Server apps can now use built-in support for storing app state in the browser that has been protected from tampering using ASP.NET Core data protection. Data can be stored in either local browser storage or session storage. For more information, see ASP.NET Core Blazor state management.

Blazor WebAssembly prerendering

Component integration is improved across hosting models, and Blazor WebAssembly apps can now prerender output on the server.

Trimming/linking improvements

Blazor WebAssembly performs Intermediate Language (IL) trimming/linking during a build to trim unnecessary IL from the app's output assemblies. With the release of

ASP.NET Core 5.0, Blazor WebAssembly performs improved trimming with additional configuration options. For more information, see Configure the Trimmer for ASP.NET Core Blazor and Trimming options.

Browser compatibility analyzer

Blazor WebAssembly apps target the full .NET API surface area, but not all .NET APIs are supported on WebAssembly due to browser sandbox constraints. Unsupported APIs throw PlatformNotSupportedException when running on WebAssembly. A platform compatibility analyzer warns the developer when the app uses APIs that aren't supported by the app's target platforms. For more information, see Consume ASP.NET Core Razor components from a Razor class library (RCL).

Lazy load assemblies

Blazor WebAssembly app startup performance can be improved by deferring the loading of some application assemblies until they're required. For more information, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Updated globalization support

Globalization support is available for Blazor WebAssembly based on International Components for Unicode (ICU). For more information, see ASP.NET Core Blazor globalization and localization.

gRPC

Many preformance improvements have been made in gRPC. For more information, see gRPC performance improvements in .NET 5.

For more gRPC information, see Overview for gRPC on .NET.

SignalR

SignalR Hub filters

SignalR Hub filters, called Hub pipelines in ASP.NET SignalR, is a feature that allows code to run before and after Hub methods are called. Running code before and after Hub methods are called is similar to how middleware has the ability to run code before

and after an HTTP request. Common uses include logging, error handling, and argument validation.

For more information, see Use hub filters in ASP.NET Core SignalR.

SignalR parallel hub invocations

ASP.NET Core SignalR is now capable of handling parallel hub invocations. The default behavior can be changed to allow clients to invoke more than one hub method at a time:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(options =>
    {
        options.MaximumParallelInvocationsPerClient = 5;
    });
}
```

Added Messagepack support in SignalR Java client

A new package, com.microsoft.signalr.messagepack, adds MessagePack support to the SignalR Java client. To use the MessagePack hub protocol, add `.withHubProtocol(new MessagePackHubProtocol())` to the connection builder:

Java

```
HubConnection hubConnection = HubConnectionBuilder.create(
                            "http://localhost:53353/MyHub")
                .withHubProtocol(new MessagePackHubProtocol())
                .build();
```

Kestrel

- Reloadable endpoints via configuration: Kestrel can detect changes to configuration passed to KestrelServerOptions.Configure and unbind from existing endpoints and bind to new endpoints without requiring an application restart when the `reloadOnChange` parameter is `true`. By default when using ConfigureWebHostDefaults or CreateDefaultBuilder, Kestrel binds to the "Kestrel" configuration subsection with `reloadOnChange` enabled. Apps must

pass `reloadOnChange: true` when calling `KestrelServerOptions.Configure` manually to get reloadable endpoints.

- HTTP/2 response headers improvements. For more information, see Performance improvements in the next section.
- Support for additional endpoints types in the sockets transport: Adding to the new API introduced in System.Net.Sockets, the sockets default transport in Kestrel allows binding to both existing file handles and Unix domain sockets. Support for binding to existing file handles enables using the existing `Systemd` integration without requiring the `libuv` transport.
- Custom header decoding in Kestrel: Apps can specify which Encoding to use to interpret incoming headers based on the header name instead of defaulting to UTF-8. Set the `Microsoft.AspNetCore.Server.Kestrel.KestrelServerOptions.RequestHeaderEncodingSelector` property to specify which encoding to use:

```
C#

public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
          webBuilder.ConfigureKestrel(options =>
          {
              options.RequestHeaderEncodingSelector = encoding =>
              {
                    return encoding switch
                    {
                        "Host" => System.Text.Encoding.Latin1,
                        _ => System.Text.Encoding.UTF8,
                    };
              };
          });
          webBuilder.UseStartup<Startup>();
      });
```

Kestrel endpoint-specific options via configuration

Support has been added for configuring Kestrel's endpoint-specific options via configuration. The endpoint-specific configurations includes the:

- HTTP protocols used
- TLS protocols used
- Certificate selected

- Client certificate mode

Configuration allows specifying which certificate is selected based on the specified server name. The server name is part of the Server Name Indication (SNI) extension to the TLS protocol as indicated by the client. Kestrel's configuration also supports a wildcard prefix in the host name.

The following example shows how to specify endpoint-specific using a configuration file:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "EndpointName": {
        "Url": "https://*",
        "Sni": {
          "a.example.org": {
            "Protocols": "Http1AndHttp2",
            "SslProtocols": [ "Tls11", "Tls12"],
            "Certificate": {
              "Path": "testCert.pfx",
              "Password": "testPassword"
            },
            "ClientCertificateMode" : "NoCertificate"
          },
          "*.example.org": {
            "Certificate": {
              "Path": "testCert2.pfx",
              "Password": "testPassword"
            }
          },
          "*": {
            // At least one sub-property needs to exist per
            // SNI section or it cannot be discovered via
            // IConfiguration
            "Protocols": "Http1",
          }
        }
      }
    }
  }
}
```

Server Name Indication (SNI) is a TLS extension to include a virtual domain as a part of SSL negotiation. What this effectively means is that the virtual domain name, or a hostname, can be used to identify the network end point.

Performance improvements

HTTP/2

- Significant reductions in allocations in the HTTP/2 code path.
- Support for HPack dynamic compression of HTTP/2 response headers in Kestrel. For more information, see Header table size and HPACK: the silent killer (feature) of HTTP/2.
- Sending HTTP/2 PING frames: HTTP/2 has a mechanism for sending PING frames to ensure an idle connection is still functional. Ensuring a viable connection is especially useful when working with long-lived streams that are often idle but only intermittently see activity, for example, gRPC streams. Apps can send periodic PING frames in Kestrel by setting limits on KestrelServerOptions:

  C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
          webBuilder.ConfigureKestrel(options =>
          {
              options.Limits.Http2.KeepAlivePingInterval =

TimeSpan.FromSeconds(10);
              options.Limits.Http2.KeepAlivePingTimeout =

TimeSpan.FromSeconds(1);
          });
          webBuilder.UseStartup<Startup>();
      });
```

Containers

Prior to .NET 5.0, building and publishing a *Dockerfile* for an ASP.NET Core app required pulling the entire .NET Core SDK and the ASP.NET Core image. With this release, pulling the SDK images bytes is reduced and the bytes pulled for the ASP.NET Core image is largely eliminated. For more information, see this GitHub issue comment.

Authentication and authorization

Microsoft Entra ID authentication with Microsoft.Identity.Web

The ASP.NET Core project templates now integrate with Microsoft.Identity.Web to handle authentication with Microsoft Entra ID. The Microsoft.Identity.Web package provides:

- A better experience for authentication through Microsoft Entra ID.
- An easier way to access Azure resources on behalf of your users, including Microsoft Graph. See the Microsoft.Identity.Web sample, which starts with a basic login and advances through multi-tenancy, using Azure APIs, using Microsoft Graph, and protecting your own APIs. `Microsoft.Identity.Web` is available alongside .NET 5.

Allow anonymous access to an endpoint

The `AllowAnonymous` extension method allows anonymous access to an endpoint:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        })
        .AllowAnonymous();
    });
}
```

Custom handling of authorization failures

Custom handling of authorization failures is now easier with the new IAuthorizationMiddlewareResultHandler interface that is invoked by the authorization Middleware. The default implementation remains the same, but a custom handler can be registered in [Dependency injection, which allows custom HTTP responses based on

why authorization failed. See this sample that demonstrates usage of the `IAuthorizationMiddlewareResultHandler`.

Authorization when using endpoint routing

Authorization when using endpoint routing now receives the `HttpContext` rather than the endpoint instance. This allows the authorization middleware to access the `RouteData` and other properties of the `HttpContext` that were not accessible though the Endpoint class. The endpoint can be fetched from the context using context.GetEndpoint.

Role-based access control with Kerberos authentication and LDAP on Linux

See Kerberos authentication and role-based access control (RBAC)

API improvements

JSON extension methods for HttpRequest and HttpResponse

JSON data can be read and written to from an `HttpRequest` and `HttpResponse` using the new ReadFromJsonAsync and `WriteAsJsonAsync` extension methods. These extension methods use the System.Text.Json serializer to handle the JSON data. The new `HasJsonContentType` extension method can also check if a request has a JSON content type.

The JSON extension methods can be combined with endpoint routing to create JSON APIs in a style of programming we call ***route to code***. It is a new option for developers who want to create basic JSON APIs in a lightweight way. For example, a web app that has only a handful of endpoints might choose to use route to code rather than the full functionality of ASP.NET Core MVC:

C#

```
endpoints.MapGet("/weather/{city:alpha}", async context =>
{
    var city = (string)context.Request.RouteValues["city"];
    var weather = GetFromDatabase(city);
```

```
    await context.Response.WriteAsJsonAsync(weather);
});
```

System.Diagnostics.Activity

The default format for System.Diagnostics.Activity now defaults to the W3C format. This makes distributed tracing support in ASP.NET Core interoperable with more frameworks by default.

FromBodyAttribute

FromBodyAttribute now supports configuring an option that allows these parameters or properties to be considered optional:

C#

```
public IActionResult Post([FromBody(EmptyBodyBehavior =
EmptyBodyBehavior.Allow)]
                          MyModel model)
{
    ...
}
```

Miscellaneous improvements

We've started applying nullable annotations to ASP.NET Core assemblies. We plan to annotate most of the common public API surface of the .NET 5 framework.

Control Startup class activation

An additional UseStartup overload has been added that lets an app provide a factory method for controlling `Startup` class activation. Controlling `Startup` class activation is useful to pass additional parameters to `Startup` that are initialized along with the host:

C#

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var logger = CreateLogger();
        var host = Host.CreateDefaultBuilder()
            .ConfigureWebHost(builder =>
```

```
            {
                builder.UseStartup(context => new Startup(logger));
            })
            .Build();

        await host.RunAsync();
    }
}
```

Auto refresh with dotnet watch

In .NET 5, running dotnet watch on an ASP.NET Core project both launches the default browser and auto refreshes the browser as changes are made to the code. This means you can:

- Open an ASP.NET Core project in a text editor.
- Run `dotnet watch`.
- Focus on the code changes while the tooling handles rebuilding, restarting, and reloading the app.

Console Logger Formatter

Improvements have been made to the console log provider in the `Microsoft.Extensions.Logging` library. Developers can now implement a custom `ConsoleFormatter` to exercise complete control over formatting and colorization of the console output. The formatter APIs allow for rich formatting by implementing a subset of the VT-100 escape sequences. VT-100 is supported by most modern terminals. The console logger can parse out escape sequences on unsupported terminals allowing developers to author a single formatter for all terminals.

JSON Console Logger

In addition to support for custom formatters, we've also added a built-in JSON formatter that emits structured JSON logs to the console. The following code shows how to switch from the default logger to JSON:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
           Host.CreateDefaultBuilder(args)
  .ConfigureLogging(logging =>
  {
     logging.AddJsonConsole(options =>
     {
```

```
            options.JsonWriterOptions = new JsonWriterOptions()
            { Indented = true };
        });
    })
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
```

Log messages emitted to the console are JSON formatted:

JSON

```
{
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Now listening on: https://localhost:5001",
  "State": {
    "Message": "Now listening on: https://localhost:5001",
    "address": "https://localhost:5001",
    "{OriginalFormat}": "Now listening on: {address}"
  }
}
```