The MSBuild property for the environment is `$(EnvironmentName)`.

When publishing from Visual Studio and using a publish profile, see Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment.

The `ASPNETCORE_ENVIRONMENT` environment variable is automatically added to the *web.config* file when the environment name is specified.

Custom

Custom transformations are run last, after Build configuration, Profile, and Environment transforms.

Include a *{CUSTOM_NAME}.transform* file for each custom configuration requiring a *web.config* transformation.

In the following example, a custom transform environment variable is set in *custom.transform*:

XML

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-
Transform">
  <location>
    <system.webServer>
      <aspNetCore>
        <environmentVariables xdt:Transform="InsertIfMissing">
          <environmentVariable name="Custom_Specific"
                               value="Custom_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the `CustomTransformFileName` property is passed to the dotnet publish command:

.NET CLI

```
dotnet publish --configuration Release
/p:CustomTransformFileName=custom.transform
```

The MSBuild property for the profile name is `$(CustomTransformFileName)`.

Prevent web.config transformation

To prevent transformations of the *web.config* file, set the MSBuild property `$(IsWebConfigTransformDisabled)`:

.NET CLI

```
dotnet publish /p:IsWebConfigTransformDisabled=true
```

Additional resources

- Web.config Transformation Syntax for Web Application Project Deployment
- Web.config Transformation Syntax for Web Project Deployment Using Visual Studio

Use ASP.NET Core with HTTP/2 on IIS

Article • 07/31/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Justin Kotalik

HTTP/2 is supported with ASP.NET Core in the following IIS deployment scenarios:

- Windows Server 2016 or later / Windows 10 or later
- IIS 10 or later
- TLS 1.2 or later connection
- When hosting out-of-process: Public-facing edge server connections use HTTP/2, but the reverse proxy connection to the Kestrel server uses HTTP/1.1.

For an in-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`. For an out-of-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/1.1`.

For more information on the in-process and out-of-process hosting models, see ASP.NET Core Module (ANCM) for IIS.

HTTP/2 is enabled by default for HTTPS/TLS connections. Connections fall back to HTTP/1.1 if an HTTP/2 connection isn't established. For more information on HTTP/2 configuration with IIS deployments, see HTTP/2 on IIS.

Advanced HTTP/2 features to support gRPC

Additional HTTP/2 features in IIS support gRPC, including support for response trailers and sending reset frames.

Requirements to run gRPC on IIS:

- In-process hosting.
- Windows 11 Build 22000 or later, Windows Server 2022 Build 20348 or later.

- TLS 1.2 or later connection.

Trailers

HTTP Trailers are similar to HTTP Headers, except they are sent after the response body is sent. For IIS and HTTP.sys, only HTTP/2 response trailers are supported.

C#

```
if (httpContext.Response.SupportsTrailers())
{
    httpContext.Response.DeclareTrailer("trailername");

    // Write body
    httpContext.Response.WriteAsync("Hello world");

    httpContext.Response.AppendTrailer("trailername", "TrailerValue");
}
```

In the preceding example code:

- `SupportsTrailers` ensures that trailers are supported for the response.
- `DeclareTrailer` adds the given trailer name to the `Trailer` response header. Declaring a response's trailers is optional, but recommended. If `DeclareTrailer` is called, it must be before the response headers are sent.
- `AppendTrailer` appends the trailer.

Reset

Reset allows for the server to reset a HTTP/2 request with a specified error code. A reset request is considered aborted.

C#

```
var resetFeature = httpContext.Features.Get<IHttpResetFeature>();
resetFeature.Reset(errorCode: 2);
```

`Reset` in the preceding code example specifies the `INTERNAL_ERROR` error code. For more information about HTTP/2 error codes, visit the HTTP/2 specification error code section.

Use ASP.NET Core with HTTP/3 on IIS

Article • 07/31/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Chris Ross

HTTP/3 is fully supported with ASP.NET Core in the following IIS scenarios:

- In-process
- Out-of-Process. In Out-of-Process, IIS responds to the client using HTTP/3, but the reverse proxy connection to the Kestrel server uses HTTP/1.1.

For more information on the in-process and out-of-process hosting models, see ASP.NET Core Module (ANCM) for IIS.

The following requirements also need to be met:

- Windows Server 2022 / Windows 11 or later
- An `https` url binding is used.
- The EnableHttp3 registry key is set.

For an in-process deployment when an HTTP/3 connection is established, HttpRequest.Protocol reports `HTTP/3`. For an out-of-process deployment when an HTTP/3 connection is established, HttpRequest.Protocol reports `HTTP/1.1` because that is how IIS proxies the requests to Kestrel.

Alt-Svc

HTTP/3 is discovered as an upgrade from HTTP/1.1 or HTTP/2 via the alt-svc header. That means the first request will normally use HTTP/1.1 or HTTP/2 before switching to HTTP/3. IIS doesn't automatically add the `alt-svc` header, it must be added by the application. The following code is a middleware example that adds the `alt-svc` response header.

C#

```
app.Use((context, next) =>
{
    context.Response.Headers.AltSvc = "h3=\":443\"";
    return next(context);
});
```

Place the preceding code early in the request pipeline.

IIS also supports sending an AltSvc HTTP/2 protocol message rather than a response header to notify the client that HTTP/3 is available. See the EnableAltSvc registry key. Note this requires netsh sslcert bindings that use host names rather than IP addresses.

HTTP.sys web server implementation in ASP.NET Core

Article • 10/21/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Tom Dykstra and Chris Ross

HTTP.sys is a web server for ASP.NET Core that only runs on Windows. HTTP.sys is an alternative to Kestrel server and offers some features that Kestrel doesn't provide.

> **Important**
>
> HTTP.sys isn't compatible with the **ASP.NET Core Module** and can't be used with IIS or IIS Express.

HTTP.sys supports the following features:

- Windows Authentication
- Port sharing
- HTTPS with SNI
- HTTP/2 over TLS (Windows 10 or later)
- Direct file transmission
- Response caching
- WebSockets (Windows 8 or later)

Supported Windows versions:

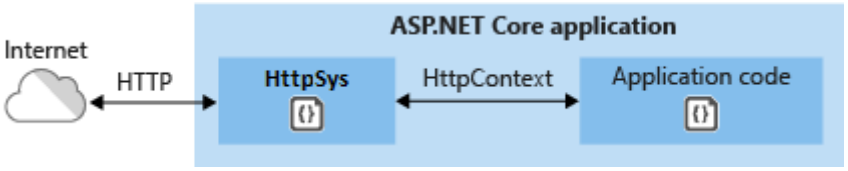
- Windows 7 or later
- Windows Server 2008 R2 or later

View or download sample code (how to download)

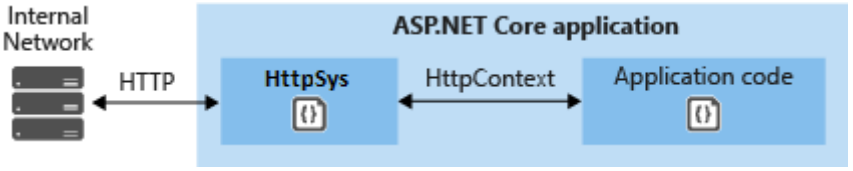
When to use HTTP.sys

HTTP.sys is useful for deployments where:

- There's a need to expose the server directly to the Internet without using IIS.



- An internal deployment requires a feature not available in Kestrel. For more information, see Kestrel vs. HTTP.sys



HTTP.sys is mature technology that protects against many types of attacks and provides the robustness, security, and scalability of a full-featured web server. IIS itself runs as an HTTP listener on top of HTTP.sys.

HTTP/2 support

HTTP/2 is enabled for ASP.NET Core apps when the following base requirements are met:

- Windows Server 2016/Windows 10 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`.

HTTP/2 is enabled by default. If an HTTP/2 connection isn't established, the connection falls back to HTTP/1.1. In a future release of Windows, HTTP/2 configuration flags will be available, including the ability to disable HTTP/2 with HTTP.sys.

HTTP/3 support

HTTP/3 is enabled for ASP.NET Core apps when the following base requirements are met:

- Windows Server 2022/Windows 11 or later
- An `https` url binding is used.
- The EnableHttp3 registry key is set.

The preceding Windows 11 Build versions may require the use of a Windows Insider build.

HTTP/3 is discovered as an upgrade from HTTP/1.1 or HTTP/2 via the `alt-svc` header. That means the first request will normally use HTTP/1.1 or HTTP/2 before switching to HTTP/3. Http.Sys doesn't automatically add the `alt-svc` header, it must be added by the application. The following code is a middleware example that adds the `alt-svc` response header.

```
C#
```

```
app.Use((context, next) =>
{
    context.Response.Headers.AltSvc = "h3=\":443\"";
    return next(context);
});
```

Place the preceding code early in the request pipeline.

Http.Sys also supports sending an AltSvc HTTP/2 protocol message rather than a response header to notify the client that HTTP/3 is available. See the EnableAltSvc registry key. This requires netsh sslcert bindings that use host names rather than IP addresses.

Kernel mode authentication with Kerberos

HTTP.sys delegates to kernel mode authentication with the Kerberos authentication protocol. User mode authentication isn't supported with Kerberos and HTTP.sys. The machine account must be used to decrypt the Kerberos token/ticket that's obtained from Active Directory and forwarded by the client to the server to authenticate the user. Register the Service Principal Name (SPN) for the host, not the user of the app.

Support for kernel-mode response buffering

In some scenarios, high volumes of small writes with high latency can cause significant performance impact to `HTTP.sys`. This impact is due to the lack of a Pipe buffer in the `HTTP.sys` implementation. To improve performance in these scenarios, support for response buffering is included in `HTTP.sys`. Enable buffering by setting HttpSysOptions.EnableKernelResponseBuffering to `true`.

Response buffering should be enabled by an app that does synchronous I/O, or asynchronous I/O with no more than one outstanding write at a time. In these scenarios,

response buffering can significantly improve throughput over high-latency connections.

Apps that use asynchronous I/O and that may have more than one write outstanding at a time should ***not*** use this flag. Enabling this flag can result in higher CPU and memory usage by HTTP.Sys.

How to use HTTP.sys

Configure the ASP.NET Core app to use HTTP.sys

Call the UseHttpSys extension method when building the host, specifying any required HttpSysOptions. The following example sets options to their default values:

C#

```
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.AspNetCore.Hosting.Server.Features;
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Server.HttpSys;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys(options =>
{
    options.AllowSynchronousIO = false;
    options.Authentication.Schemes = AuthenticationSchemes.None;
    options.Authentication.AllowAnonymous = true;
    options.MaxConnections = null;
    options.MaxRequestBodySize = 30_000_000;
    options.UrlPrefixes.Add("http://localhost:5005");
});

builder.Services.AddRazorPages();

var app = builder.Build();
```

Additional HTTP.sys configuration is handled through registry settings.

For more information about HTTP.sys options, see HttpSysOptions.

MaxRequestBodySize

The maximum allowed size of any request body in bytes. When set to `null`, the maximum request body size is unlimited. This limit has no effect on upgraded connections, which are always unlimited.

The recommended method to override the limit in an ASP.NET Core MVC app for a single `IActionResult` is to use the RequestSizeLimitAttribute attribute on an action method:

C#

```
[RequestSizeLimit(100000000)]
public IActionResult MyActionMethod()
```

An exception is thrown if the app attempts to configure the limit on a request after the app has started reading the request. An `IsReadOnly` property can be used to indicate if the `MaxRequestBodySize` property is in a read-only state, meaning it's too late to configure the limit.

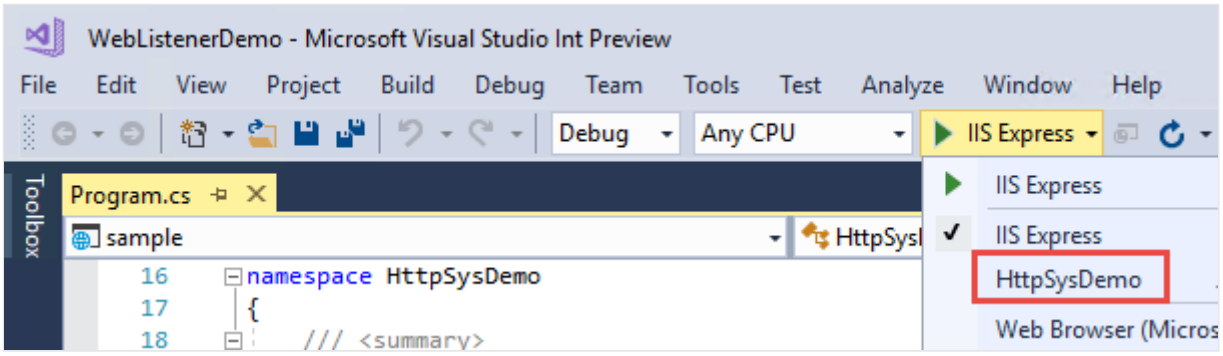
If the app should override MaxRequestBodySize per-request, use the IHttpMaxRequestBodySizeFeature:

C#

```
app.Use((context, next) =>
{
    context.Features.GetRequiredFeature<IHttpMaxRequestBodySizeFeature>()
                                            .MaxRequestBodySize = 10 *
1024;

    var server = context.RequestServices
        .GetRequiredService<IServer>();
    var serverAddressesFeature = server.Features

.GetRequiredFeature<IServerAddressesFeature>();

    var addresses = string.Join(", ", serverAddressesFeature.Addresses);

    var loggerFactory = context.RequestServices
        .GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger("Sample");

    logger.LogInformation("Addresses: {addresses}", addresses);

    return next(context);
});
```

If using Visual Studio, make sure the app isn't configured to run IIS or IIS Express.

In Visual Studio, the default launch profile is for IIS Express. To run the project as a console app, manually change the selected profile, as shown in the following screenshot:



Configure Windows Server

1. Determine the ports to open for the app and use Windows Firewall or the New-NetFirewallRule PowerShell cmdlet to open firewall ports to allow traffic to reach HTTP.sys. In the following commands and app configuration, port 443 is used.

2. When deploying to an Azure VM, open the ports in the Network Security Group. In the following commands and app configuration, port 443 is used.

3. Obtain and install X.509 certificates, if required.

   On Windows, create self-signed certificates using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

   Install either self-signed or CA-signed certificates in the server's **Local Machine** > **Personal** store.

4. If the app is a framework-dependent deployment, install .NET Core, .NET Framework, or both (if the app is a .NET Core app targeting the .NET Framework).

   - **.NET Core**: If the app requires .NET Core, obtain and run the **.NET Core Runtime** installer from .NET Core Downloads. Don't install the full SDK on the server.
   - **.NET Framework**: If the app requires .NET Framework, see the .NET Framework installation guide. Install the required .NET Framework. The installer for the latest .NET Framework is available from the .NET Core Downloads page.

   If the app is a self-contained deployment, the app includes the runtime in its deployment. No framework installation is required on the server.

5. Configure URLs and ports in the app.

   By default, ASP.NET Core binds to `http://localhost:5000`. To configure URL prefixes and ports, options include:

- UseUrls
- `urls` command-line argument
- `ASPNETCORE_URLS` environment variable
- UrlPrefixes

The following code example shows how to use UrlPrefixes with the server's local IP address `10.0.0.4` on port 443:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys(options =>
{
    options.UrlPrefixes.Add("https://10.0.0.4:443");
});

builder.Services.AddRazorPages();

var app = builder.Build();
```

An advantage of `UrlPrefixes` is that an error message is generated immediately for improperly formatted prefixes.

The settings in `UrlPrefixes` override `UseUrls`/`urls`/`ASPNETCORE_URLS` settings. Therefore, an advantage of `UseUrls`, `urls`, and the `ASPNETCORE_URLS` environment variable is that it's easier to switch between Kestrel and HTTP.sys.

HTTP.sys recognizes two types of wild cards in URL prefixes:

- `*` is a *weak binding*, also known as a *fallback binding*. If the URL prefix is `http://*:5000`, and something else is bound to port 5000, this binding won't be used.
- `+` is a *strong binding*. If the URL prefix is `http://+:5000`, this binding will be used before other port 5000 bindings.

For more information, see UrlPrefix Strings.

> ⚠ **Warning**
>
> Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings create app security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names or IP addresses rather than wildcards. Subdomain wildcard binding (for example,

`*.mysub.com`) isn't a security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). For more information, see RFC 9110: Section 7.2: Host and :authority.

Apps and containers are often given only a port to listen on, like port 80, without additional constraints like host or path. HTTP_PORTS and HTTPS_PORTS are config keys that specify the listening ports for the Kestrel and HTTP.sys servers. These keys may be specified as environment variables defined with the `DOTNET_` or `ASPNETCORE_` prefixes, or specified directly through any other config input, such as `appsettings.json`. Each is a semicolon-delimited list of port values, as shown in the following example:

```
ASPNETCORE_HTTP_PORTS=80;8080
ASPNETCORE_HTTPS_PORTS=443;8081
```

The preceding example is shorthand for the following configuration, which specifies the scheme (HTTP or HTTPS) and any host or IP.

```
ASPNETCORE_URLS=http://*:80/;http://*:8080/;https://*:443/;https://*:80
81/
```

The HTTP_PORTS and HTTPS_PORTS configuration keys are lower priority and are overridden by URLS or values provided directly in code. Certificates still need to be configured separately via server-specific mechanics for HTTPS.

These configuration keys are equivalent to top-level wildcard bindings. They're convenient for development and container scenarios, but avoid wildcards when running on a machine that may also host other services.

6. Preregister URL prefixes on the server.

   The built-in tool for configuring HTTP.sys is *netsh.exe*. *netsh.exe* is used to reserve URL prefixes and assign X.509 certificates. The tool requires administrator privileges.

   Use the *netsh.exe* tool to register URLs for the app:

   ```
   Console
   ```

```
netsh http add urlacl url=<URL> user=<USER>
```

- `<URL>`: The fully qualified Uniform Resource Locator (URL). Don't use a wildcard binding. Use a valid hostname or local IP address. *The URL must include a trailing slash.*
- `<USER>`: Specifies the user or user-group name.

In the following example, the local IP address of the server is `10.0.0.4`:

Console

```
netsh http add urlacl url=https://10.0.0.4:443/ user=Users
```

When a URL is registered, the tool responds with `URL reservation successfully added`.

To delete a registered URL, use the `delete urlacl` command:

Console

```
netsh http delete urlacl url=<URL>
```

7. Register X.509 certificates on the server.

   Use the *netsh.exe* tool to register certificates for the app:

   Console

   ```
   netsh http add sslcert ipport=<IP>:<PORT> certhash=<THUMBPRINT> appid="
   {<GUID>}"
   ```

   - `<IP>`: Specifies the local IP address for the binding. Don't use a wildcard binding. Use a valid IP address.
   - `<PORT>`: Specifies the port for the binding.
   - `<THUMBPRINT>`: The X.509 certificate thumbprint.
   - `<GUID>`: A developer-generated GUID to represent the app for informational purposes.

   For reference purposes, store the GUID in the app as a package tag:

   - In Visual Studio:

    - Open the app's project properties by right-clicking on the app in **Solution Explorer** and selecting **Properties**.
    - Select the **Package** tab.
    - Enter the GUID that you created in the **Tags** field.
- When not using Visual Studio:

    - Open the app's project file.

    - Add a `<PackageTags>` property to a new or existing `<PropertyGroup>` with the GUID that you created:

      XML

      ```
      <PropertyGroup>
        <PackageTags>00001111-aaaa-2222-bbbb-
      3333cccc4444</PackageTags>
      </PropertyGroup>
      ```

In the following example:

- The local IP address of the server is `10.0.0.4`.
- An online random GUID generator provides the `appid` value.

Console

```
netsh http add sslcert
    ipport=10.0.0.4:443
    certhash=b66ee04419d4ee37464ab8785ff02449980eae10
    appid="{00001111-aaaa-2222-bbbb-3333cccc4444}"
```

When a certificate is registered, the tool responds with `SSL Certificate successfully added`.

To delete a certificate registration, use the `delete sslcert` command:

Console

```
netsh http delete sslcert ipport=<IP>:<PORT>
```

Reference documentation for *netsh.exe*:

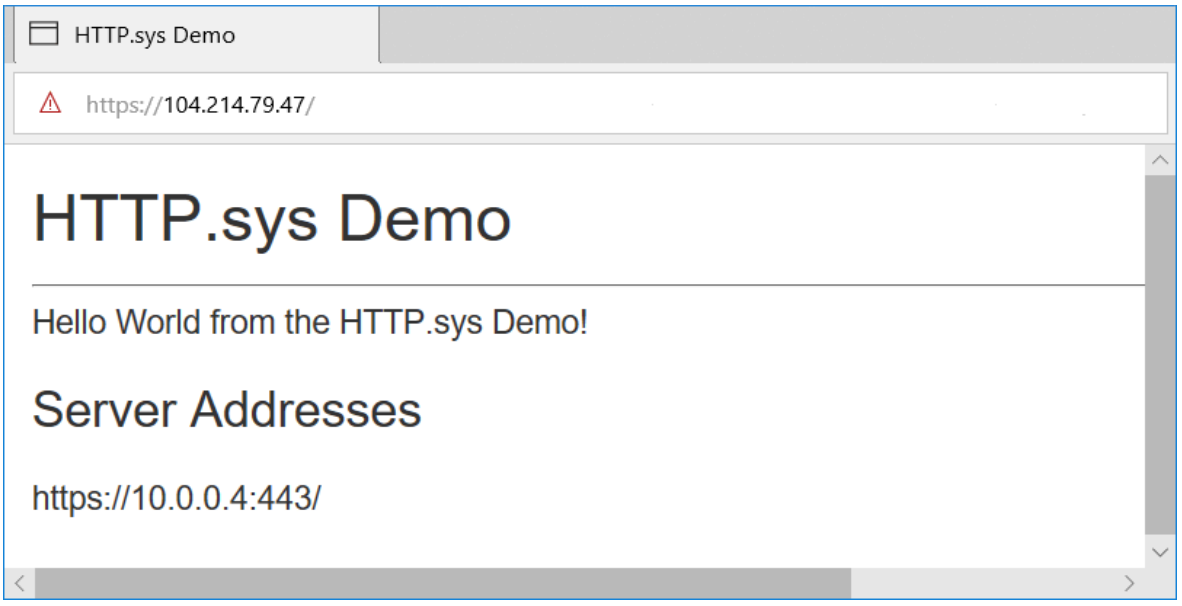
- Netsh Commands for Hypertext Transfer Protocol (HTTP)
- UrlPrefix Strings

8. Run the app.

Administrator privileges aren't required to run the app when binding to localhost using HTTP (not HTTPS) with a port number greater than 1024. For other configurations (for example, using a local IP address or binding to port 443), run the app with administrator privileges.

The app responds at the server's public IP address. In this example, the server is reached from the Internet at its public IP address of `104.214.79.47`.

A development certificate is used in this example. The page loads securely after bypassing the browser's untrusted certificate warning.



Proxy server and load balancer scenarios

For apps hosted by HTTP.sys that interact with requests from the Internet or a corporate network, additional configuration might be required when hosting behind proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Get detailed timing information with IHttpSysRequestTimingFeature

IHttpSysRequestTimingFeature provides detailed timing information for requests:

- Timestamps are obtained using QueryPerformanceCounter.
- The timestamp frequency can be obtained via QueryPerformanceFrequency.
- The index of the timing can be cast to HttpSysRequestTimingType to know what the timing represents.
- The value may be 0 if the timing isn't available for the current request.

- Requires Windows 10 version 2004, Windows Server 2022, or later.

C#

```
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Server.HttpSys;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys();

var app = builder.Build();

app.Use((context, next) =>
{
    var feature =
context.Features.GetRequiredFeature<IHttpSysRequestTimingFeature>();

    var loggerFactory =
context.RequestServices.GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger("Sample");

    var timestamps = feature.Timestamps;

    for (var i = 0; i < timestamps.Length; i++)
    {
        var timestamp = timestamps[i];
        var timingType = (HttpSysRequestTimingType)i;

        logger.LogInformation("Timestamp {timingType}: {timestamp}",
                                            timingType, timestamp);
    }

    return next(context);
});

app.MapGet("/", () => Results.Ok());

app.Run();
```

IHttpSysRequestTimingFeature.TryGetTimestamp retrieves the timestamp for the provided timing type:

C#

```
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Server.HttpSys;
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys();

var app = builder.Build();
```

```
app.Use((context, next) =>
{
    var feature =
context.Features.GetRequiredFeature<IHttpSysRequestTimingFeature>();

    var loggerFactory =
context.RequestServices.GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger("Sample");

    var timingType = HttpSysRequestTimingType.RequestRoutingEnd;

    if (feature.TryGetTimestamp(timingType, out var timestamp))
    {
        logger.LogInformation("Timestamp {timingType}: {timestamp}",
                                            timingType, timestamp);
    }
    else
    {
        logger.LogInformation("Timestamp {timingType}: not available for the
"
                                             + "current request",
timingType);
    }

    return next(context);
});

app.MapGet("/", () => Results.Ok());

app.Run();
```

IHttpSysRequestTimingFeature.TryGetElapsedTime gives the elapsed time between two specified timings:

C#

```
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Server.HttpSys;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys();

var app = builder.Build();

app.Use((context, next) =>
{
    var feature =
context.Features.GetRequiredFeature<IHttpSysRequestTimingFeature>();

    var loggerFactory =
context.RequestServices.GetRequiredService<ILoggerFactory>();
```

```
    var logger = loggerFactory.CreateLogger("Sample");

    var startingTimingType = HttpSysRequestTimingType.RequestRoutingStart;
    var endingTimingType = HttpSysRequestTimingType.RequestRoutingEnd;

    if (feature.TryGetElapsedTime(startingTimingType, endingTimingType, out
var elapsed))
    {
        logger.LogInformation(
            "Elapsed time {startingTimingType} to {endingTimingType}:
{elapsed}",
            startingTimingType,
            endingTimingType,
            elapsed);
    }
    else
    {
        logger.LogInformation(
            "Elapsed time {startingTimingType} to {endingTimingType}:"
            + " not available for the current request.",
            startingTimingType,
            endingTimingType);
    }

    return next(context);
});

app.MapGet("/", () => Results.Ok());

app.Run();
```

Advanced HTTP/2 features to support gRPC

Additional HTTP/2 features in HTTP.sys support gRPC, including support for response trailers and sending reset frames.

Requirements to run gRPC with HTTP.sys:

- Windows 11 Build 22000 or later, Windows Server 2022 Build 20348 or later.
- TLS 1.2 or later connection.

Trailers

HTTP Trailers are similar to HTTP Headers, except they are sent after the response body is sent. For IIS and HTTP.sys, only HTTP/2 response trailers are supported.

```
C#
```

```
if (httpContext.Response.SupportsTrailers())
{
    httpContext.Response.DeclareTrailer("trailername");

    // Write body
    httpContext.Response.WriteAsync("Hello world");

    httpContext.Response.AppendTrailer("trailername", "TrailerValue");
}
```

In the preceding example code:

- `SupportsTrailers` ensures that trailers are supported for the response.
- `DeclareTrailer` adds the given trailer name to the `Trailer` response header. Declaring a response's trailers is optional, but recommended. If `DeclareTrailer` is called, it must be before the response headers are sent.
- `AppendTrailer` appends the trailer.

Reset

Reset allows for the server to reset a HTTP/2 request with a specified error code. A reset request is considered aborted.

C#

```
var resetFeature = httpContext.Features.Get<IHttpResetFeature>();
resetFeature.Reset(errorCode: 2);
```

`Reset` in the preceding code example specifies the `INTERNAL_ERROR` error code. For more information about HTTP/2 error codes, visit the HTTP/2 specification error code section.

Tracing

For information about how to get traces from HTTP.sys, see HTTP.sys Manageability Scenarios.

Additional resources

- Enable Windows Authentication with HTTP.sys
- HTTP Server API
- aspnet/HttpSysServer GitHub repository (source code)

- The host
- Troubleshoot and debug ASP.NET Core projects

.NET Hot Reload support for ASP.NET Core

Article • 04/10/2024

.NET Hot Reload applies code changes, including changes to stylesheets, to a running app without restarting the app and without losing app state. Hot Reload is supported for all ASP.NET Core 6.0 and later projects.

Generally, updated code is rerun to take effect with the following conditions:

- Some startup logic is only run once:
  - Middleware, unless the code update is to an inline middleware delegate.
  - Configured services.
  - Route creation and configuration, unless the code update is to a route handler delegate (for example, `OnInitialized`).
- In Blazor apps, the framework triggers a Razor component render automatically.
- In MVC and Razor Pages apps, Hot Reload triggers a browser refresh automatically.
- Removing a Razor component parameter attribute doesn't cause the component to rerender. The app must be restarted.

For more information on supported scenarios, see Supported code changes (C# and Visual Basic).

Blazor WebAssembly

Blazor WebAssembly Hot Reload supports the following code changes:

- New types.
- Nested classes.
- Most changes to method bodies, such as adding, removing, and editing variables, expressions, and statements.
- Changes to the bodies of lambda expressions and local functions.
- Adding static and instance methods to existing types.
- Adding static and instance fields, events, and properties to existing types.
- Adding static lambdas to existing methods.
- Adding lambdas that capture `this` to existing methods that already captured `this` previously.

Note that when an attribute is removed that previously set the value of a component parameter, the component is disposed and re-initialized to set the removed parameter

back to its default value.

The following code changes aren't supported for Blazor WebAssembly apps:

- Adding a new await operator or yield keyword expression.
- Changing the names of method parameters.

.NET CLI

Hot Reload is activated using the dotnet watch command:

.NET CLI

```
dotnet watch
```

To force the app to rebuild and restart, use the keyboard combination `Ctrl`+`R` in the command shell.

When an unsupported code edit is made, called a *rude edit*, `dotnet watch` asks you if you want to restart the app:

- **Yes**: Restarts the app.
- **No**: Doesn't restart the app and leaves the app running without the changes applied.
- **Always**: Restarts the app as needed when rude edits occur.
- **Never**: Doesn't restart the app and avoids future prompts.

To disable support for Hot Reload, pass the `--no-hot-reload` option to the `dotnet watch` command:

.NET CLI

```
dotnet watch --no-hot-reload
```

Disable Hot Reload

The following setting in `Properties/launchSettings.json` disables Hot Reload:

JSON

```
"hotReloadEnabled" : false
```

Additional resources

For more information, see the following resources in the Visual Studio documentation:

- YouTube video .NET 6 Hot Reload in Visual Studio 2022, VS Code, and NOTEPAD?!?
- Introducing the .NET Hot Reload experience for editing code at runtime
- Write and debug running code with Hot Reload in Visual Studio
- Updates for Blazor & Razor editors + Hot Reload for ASP.NET
- Test Execution with Hot Reload

How to use dev tunnels in Visual Studio 2022 with ASP.NET Core apps

Article • 11/06/2024

The *dev tunnels* feature of Visual Studio 2022 enables ad-hoc connections between machines that can't directly connect to each other. A URL is created that enables any device with an internet connection to connect to an ASP.NET Core project while it runs on localhost.

Use cases

Some of the scenarios that dev tunnels enable:

- Test a web app on other devices, like mobile phones and tablets.
- Test an app with external services. For instance, test and debug Power Platform connectors, Azure Communication Services APIs, or Twilio webhooks.
- Make an app temporarily available to others over the internet, for a presentation or to invite others to review your work on a web app or API.
- As an alternative to other port-forwarding solutions.

Prerequisites

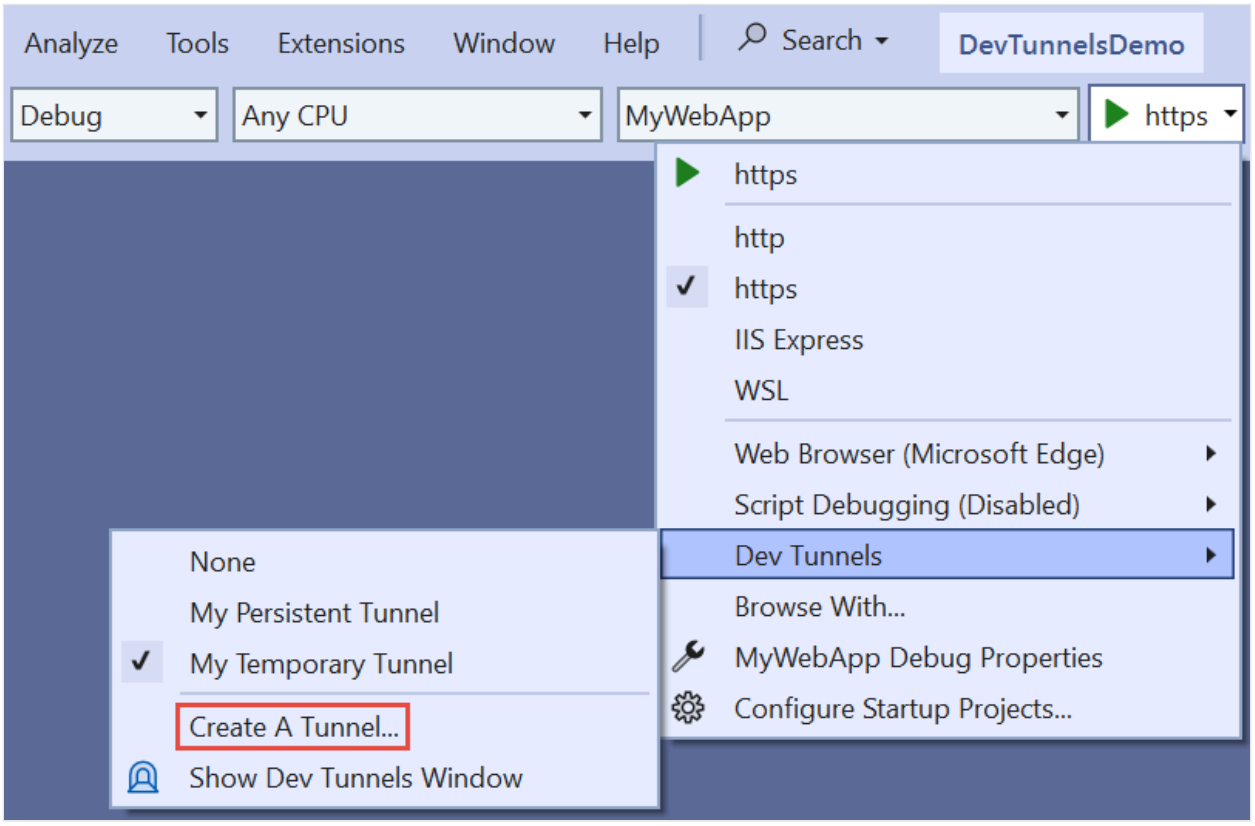
- Visual Studio 2022 version 17.6 or later with the **ASP.NET and web development** workload installed. You need to be signed in to Visual Studio to create and use dev tunnels. The feature isn't available in Visual Studio for Mac.
- One or more ASP.NET Core projects. This article uses a solution with two sample projects to demonstrate the feature.

Create a tunnel

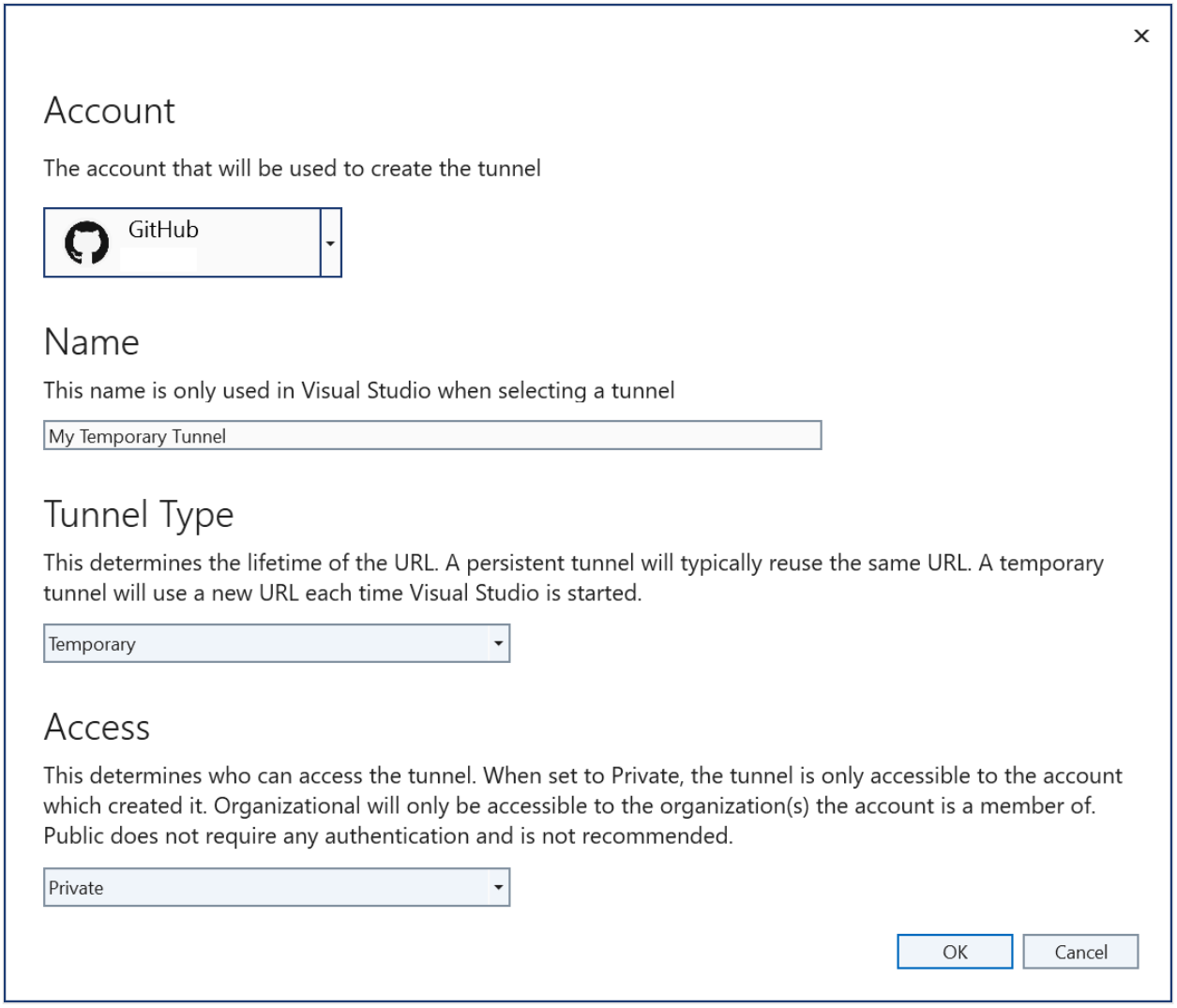
To create a tunnel:

In Visual Studio 2022, open an ASP.NET Core web project, or a solution with at least one web project set as a startup project.

In the debug dropdown, select **Dev Tunnels** > **Create A Tunnel**.



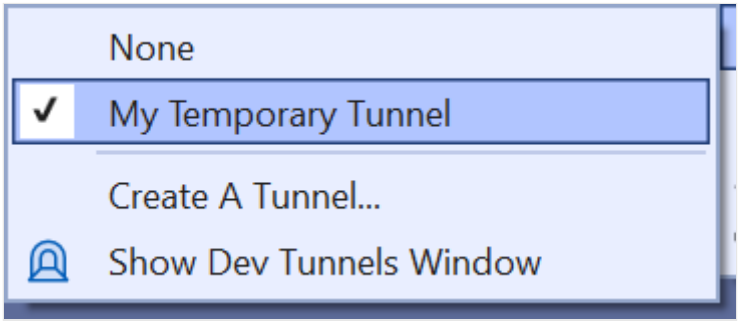
The tunnel creation dialog opens.



- Select the account to use to create the tunnel. Account types that can be used to create tunnels include Azure, Microsoft Account (MSA), and GitHub.
- Enter a name for the tunnel. This name identifies the tunnel in the Visual Studio UI.
- Choose the tunnel type, Persistent or Temporary:
  - A temporary tunnel gets a new URL each time Visual Studio is started.
  - A persistent tunnel gets the same URL each time Visual Studio is started. For more information, see Persistent vs. temporary tunnels later in this article.
- Choose the authentication that is required for access to the tunnel. The following options are available:
  - Private: The tunnel is accessible only to the account that created it.
  - Organization: The tunnel is accessible to accounts in the same organization as the one that created it. If this option is selected for a personal Microsoft account (MSA), the effect is the same as selecting Private. Organization support for Github accounts isn't supported.
  - Public: No authentication required. Choose this option only if it's safe to make the web app or API accessible to anyone on the internet.
- Select **OK**.

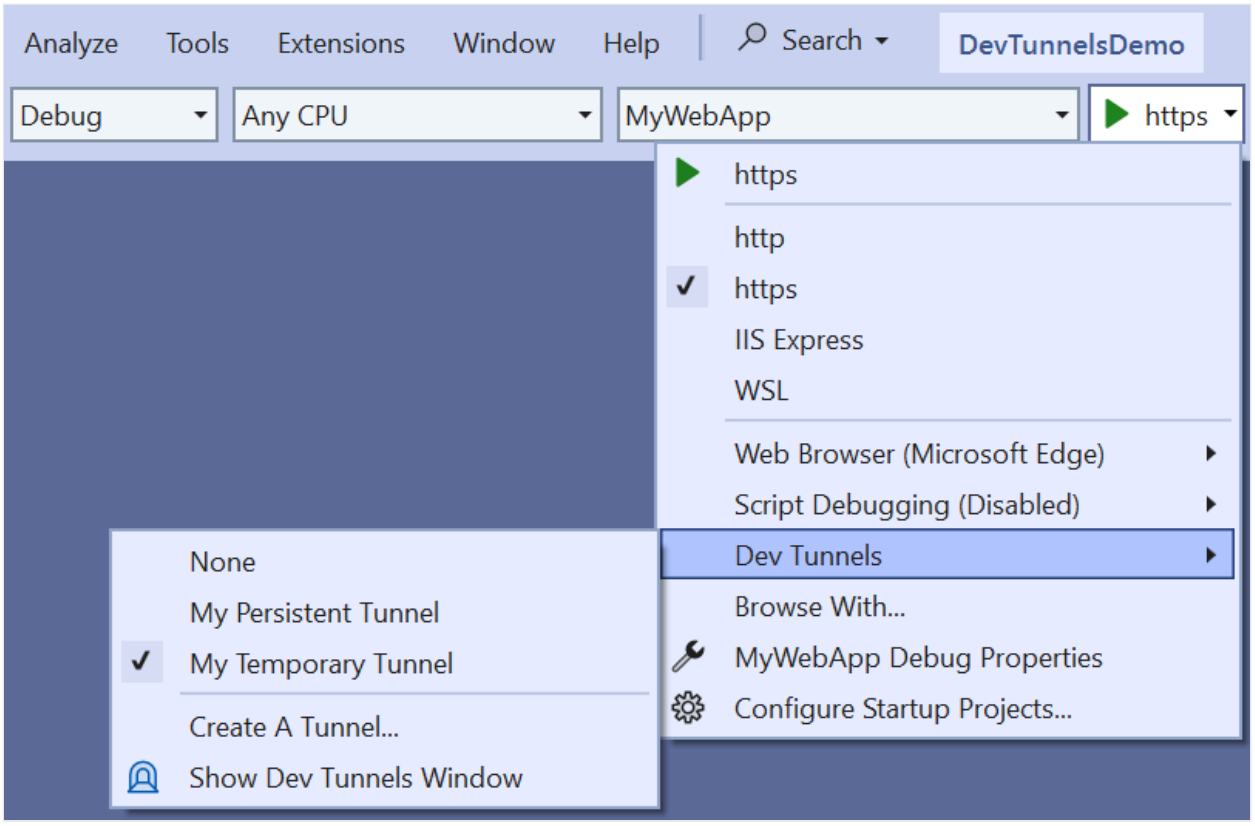
Visual Studio displays confirmation of tunnel creation:

The tunnel appears in the debug dropdown **Dev Tunnels** flyout:



Specify the active tunnel

A project or solution can have multiple tunnels, but only one at a time is active. The **Dev Tunnels** flyout in the debug dropdown can specify the active tunnel. When there is an active tunnel, it's used for all ASP.NET Core projects that are started in Visual Studio. Once a tunnel is selected as active, it remains active until Visual Studio is closed. In the following illustration, **My Temporary Tunnel** is active:



Choose not to use a tunnel by selecting **None** in the flyout. When Visual Studio is restarted, it defaults back to **None**.

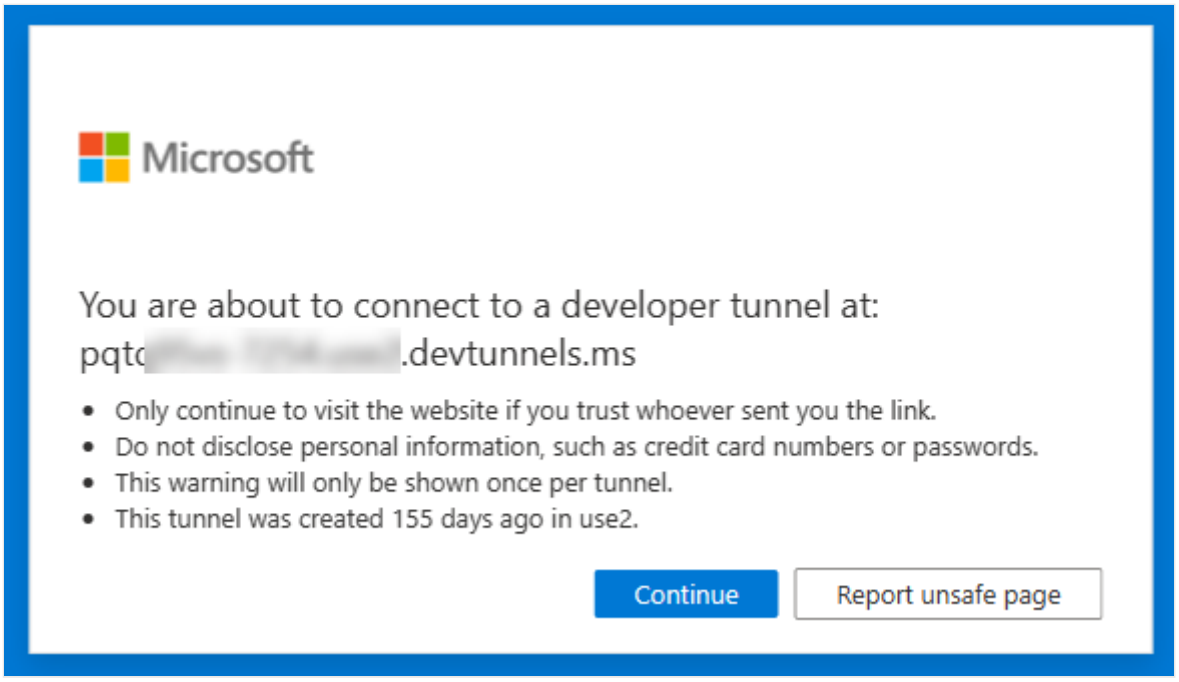
Use a tunnel

When a tunnel is active and Visual Studio runs a web app, the web browser opens to a tunnel URL instead of a localhost URL. The tunnel URL looks like the following example:

```
https://0pbvlk3m-7032.usw2.devtunnels.ms
```

Now any authenticated user can open the same URL on any other internet-connected device. As long as the project continues to run locally, any device with an internet connection can access the web application that is running on a development machine.

For web projects that have browser support, a warning page is shown on the first request sent to the tunnel URL from each device:



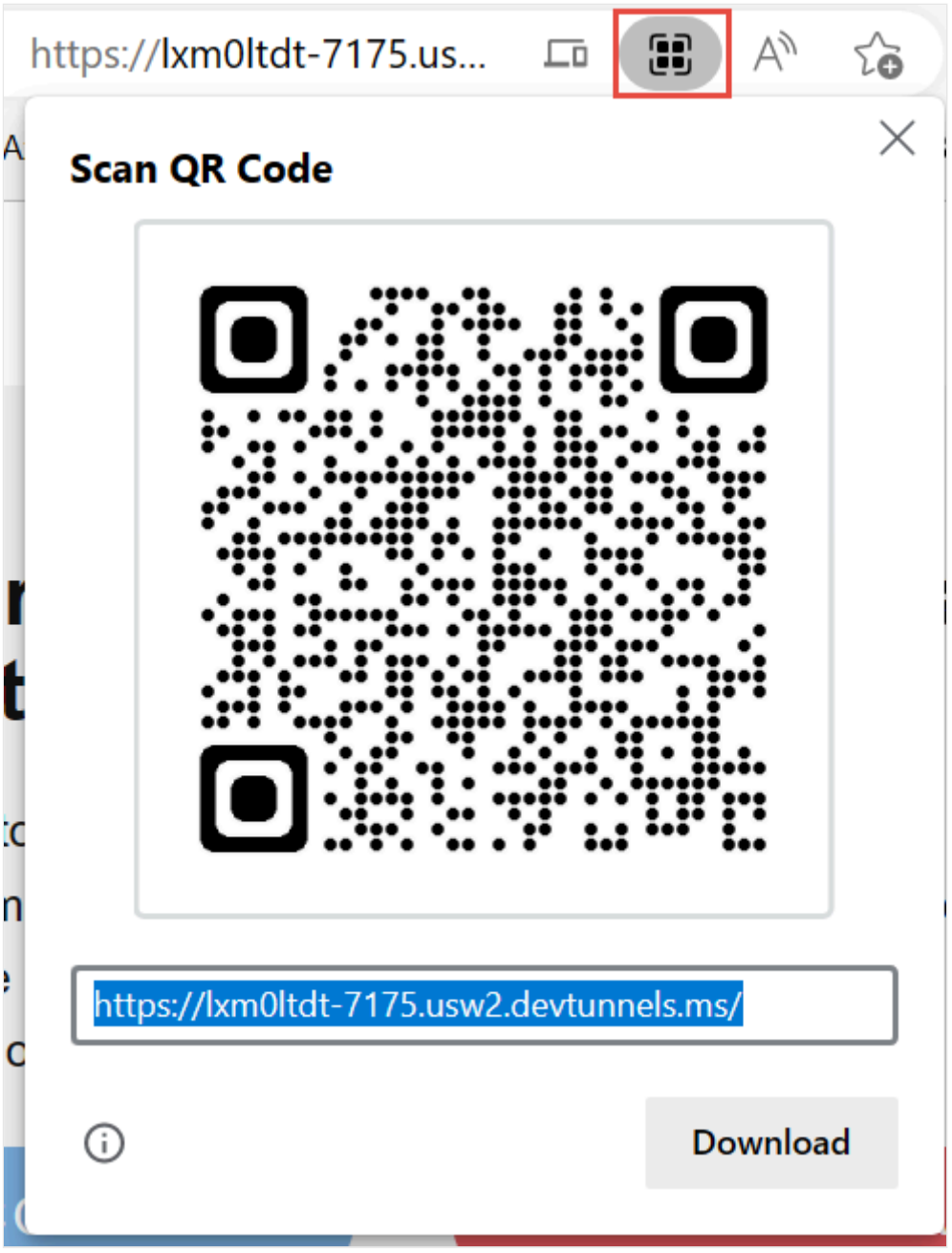
After **Continue** is selected, the request is routed to the local web app. This notification page isn't shown for API requests using dev tunnels.

Use a tunnel to test on a phone or tablet

To test a web app from an external device like a phone or tablet, navigate to the tunnel URL. To make it easier to reproduce the URL on the external device:

- Navigate to the tunnel URL in an Edge browser on the local machine.
- Generate a QR code to the URL in the Edge browser on the local machine:
  - Select the URL bar and the QR code button appears.

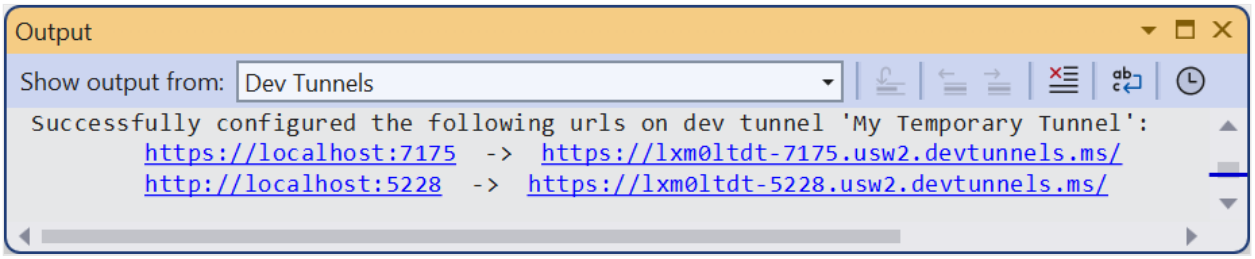
- Select the QR code button to generate and view the QR code.



- Scan this QR code with a phone or tablet to navigate to the URL.

Dev Tunnels output window

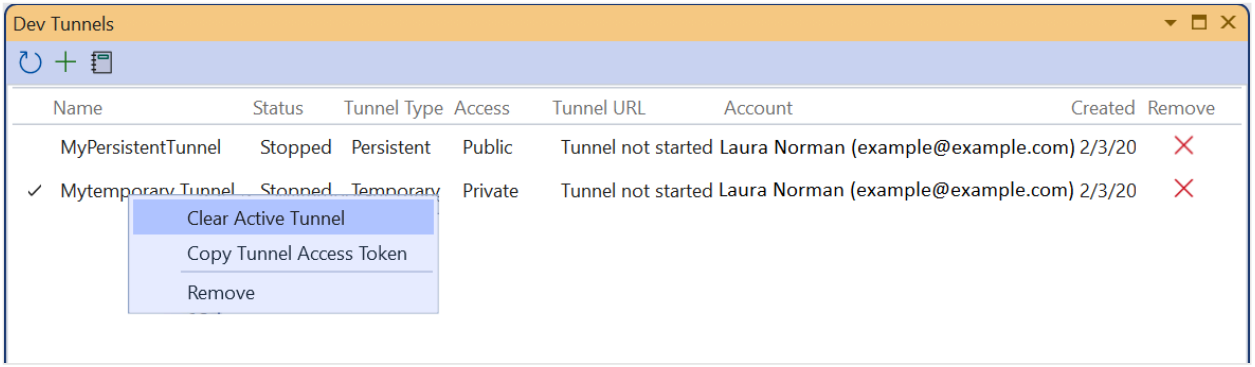
To show the URL of a tunnel of a running project, select **Dev Tunnels** in the **Show output from** dropdown.



This window is especially useful for projects that don't open a browser by default. For example, when working with an Azure Function, this may be the easiest way to discover the public URL that is being used by the dev tunnel.

Dev Tunnels tool window

View and manage dev tunnels in the **Dev Tunnels** tool window:



To open the **Dev Tunnels** window, select the **Show Dev Tunnels Window** menu option in the debug dropdown. Alternatively, select **View** > **Other Windows** > **Dev Tunnels**.

From the **Dev Tunnels** window, create a new tunnel by selecting the green `+` button.

Delete a tunnel by using the red `x` button to the right of the tunnel.

The context menu for a tunnel provides the following options:

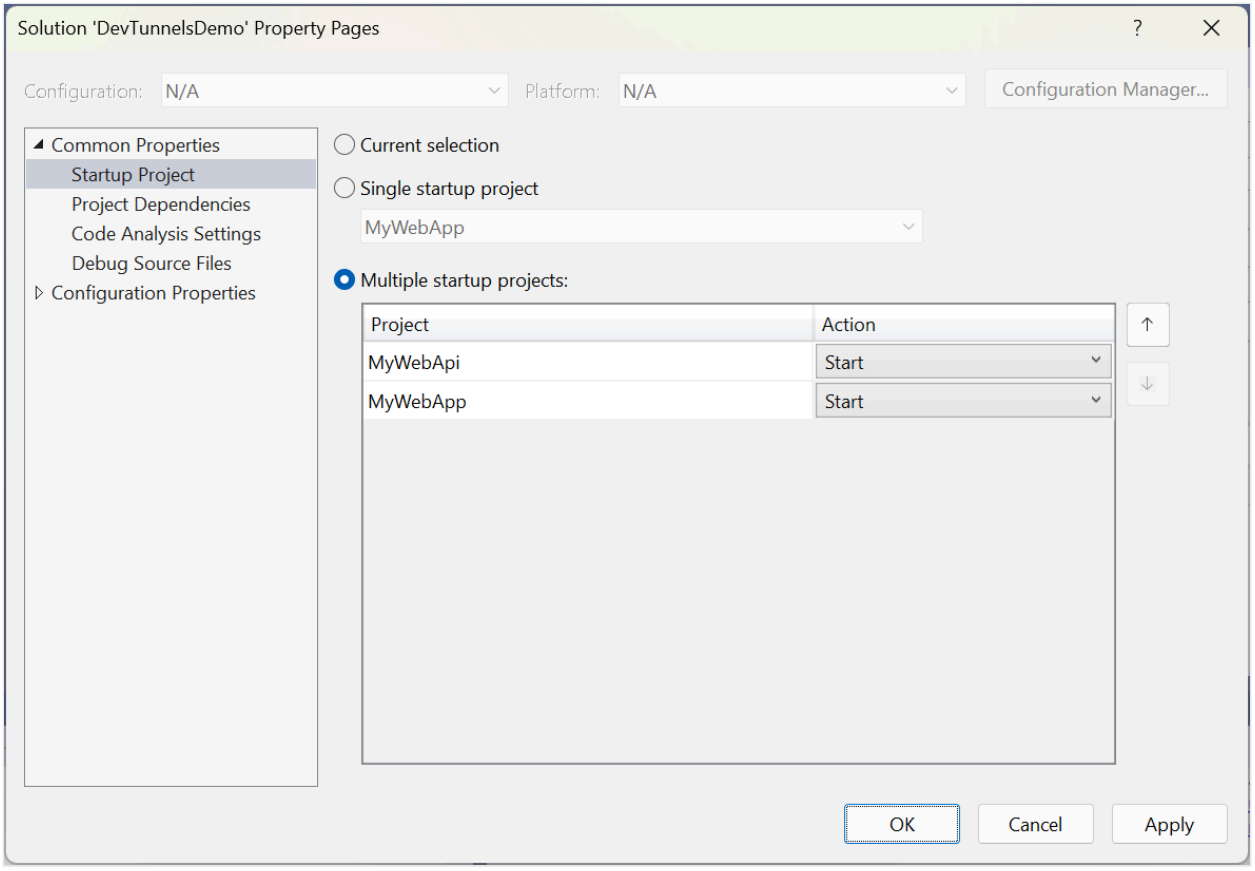
- **Clear Active Tunnel**: Shown when a tunnel is configured as active (indicated by the checkmark on the left hand side), this resets it so the solution is not using a tunnel.
- **Make Active Tunnel**: Shown for tunnels that are not configured as active.
- **Copy Tunnel Access Token**: Provided for scenarios where a tunnel is created with private or organizational access, and the app is a web API. To authenticate for the tunnel, copy and paste the tunnel access token as a header of the form `X-Tunnel-Authorization tunnel <TOKEN>` in the request. If this header is not specified, the request will be blocked because the authentication check failed.
- **Remove**

Tunnel URL environment variables

The dev tunnels feature provides a way to get the tunnel URL of a project programmatically at run time. When an app is launched that uses a tunnel, Visual Studio creates the environment variable `VS_TUNNEL_URL`. The `VS_TUNNEL_URL` value is the URL for the tunnel that is used for the current project. `VS_TUNNEL_URL` can be useful when

integrating the app with an external service, where the tunnel URL needs to be passed to the external service.

If multiple ASP.NET Core projects are configured to start in Visual Studio, the app that is starting up gets an environment variable for any project that started before it. The pattern for this variable name is `VS_TUNNEL_URL_{ProjectName}`, where `{ProjectName}` is the name of the other project. For example, consider this example showing two projects set to start:



Since MyWebApi is above MyWebApp, it's started before the MyWebApp project. When the MyWebApi project is started, it receives its tunnel URL in the `VS_TUNNEL_URL` environment variable. When the MyWebApp project is started, it receives its own tunnel URL in `VS_TUNNEL_URL` and the other project's tunnel URL is provided in the `VS_TUNNEL_URL_MyWebApi` environment variable.

To illustrate, the following highlighted lines of code have been added to the *Program.cs* file in MyWebApp:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
```

```
Console.WriteLine($"Tunnel URL: {Environment.
    GetEnvironmentVariable("VS_TUNNEL_URL")}");
Console.WriteLine($"API project tunnel URL: {Environment.
    GetEnvironmentVariable("VS_TUNNEL_URL_MyWebApi")}");
```

When the web app is started the console output looks like the following example:

Console

```
Tunnel URL: https://lxm0ltdt-7175.usw2.devtunnels.ms/
API project tunnel URL: https://lxm0ltdt-7042.usw2.devtunnels.ms/
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7175
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5228
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\DevTunnelsDemo\MyWebApp
```

For information about how to set up multiple startup projects, see How to: Set multiple startup projects.

Persistent versus temporary tunnels

A persistent tunnel is one that uses the same URL after exiting and restarting Visual Studio. Having a URL that doesn't change can be useful when integrating a web app with an external service. For example, implementing a GitHub webhook, or developing an API to integrate with a Power Platform app. In such cases, you might need to specify the callback URL to the external service. With a persistent tunnel, the external service URL only needs to be configured once. Using a temporary tunnel, the tunnel URL must be configured each time Visual Studio restarts.

Persistent doesn't mean the tunnel works when Visual Studio isn't open. A tunnel URL connects to the local machine only if the ASP.NET Core project that the tunnel URL connects to is running in Visual Studio.

A temporary tunnel is fine when the dev tunnel URL needs to work for a short time. For example, sharing in-progress work on a web app with others, or testing an app on an external device. In some cases, it might be best to get a new URL each time Visual Studio starts.

See also

The following resources use an early preview version of the dev tunnels feature, so parts of them are out of date:

- Use dev tunnels in Visual Studio to debug your web APIs
- ACS Call Automation and Azure OpenAI Service
- Use Visual Studio dev tunnels to handle Twilio Webhooks.

Use .http files in Visual Studio 2022

Article • 07/23/2024

The Visual Studio 2022 `.http` file editor provides a convenient way to test ASP.NET Core projects, especially API apps. The editor provides a UI that:

- Creates and updates `.http` files.
- Sends HTTP requests specified in `.http` files.
- Displays the responses.

This article contains documentation for:

- The .http file syntax.
- How to create an .http file.
- How to send a request from an .http file.
- Where to find .http file options that can be configured..
- How to create requests in .http files by using the Visual Studio 2022 **Endpoints Explorer**.

The `.http` file format and editor was inspired by the Visual Studio Code REST Client extension. The Visual Studio 2022 `.http` editor recognizes `.rest` as an alternative file extension for the same file format.

Prerequisites

- Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed.

`.http` file syntax

The following sections explain `.http` file syntax.

Requests

The format for an HTTP request is `HTTPMethod URL HTTPVersion`, all on one line, where:

- `HTTPMethod` is the HTTP method to use, for example:
  - OPTIONS
  - GET
  - HEAD

    - POST
    - PUT
    - PATCH
    - DELETE
    - TRACE
    - CONNECT
- `URL` is the URL to send the request to. The URL can include query string parameters. The URL doesn't have to point to a local web project. It can point to any URL that Visual Studio can access.
- `HTTPVersion` is optional and specifies the HTTP version that should be used, that is, `HTTP/1.1`, `HTTP/2`, or `HTTP/3`.

A file can contain multiple requests by using lines with `###` as delimiters. The following example showing three requests in a file illustrates this syntax:

```HTTP
GET https://localhost:7220/weatherforecast

###

GET https://localhost:7220/weatherforecast?date=2023-05-11&location=98006

###

GET https://localhost:7220/weatherforecast HTTP/3

###
```

Request headers

To add one or more headers, add each header on its own line immediately after the request line. Don't include any blank lines between the request line and the first header or between subsequent header lines. The format is `HeaderName: Value`, as shown in the following examples:

```HTTP
GET https://localhost:7220/weatherforecast
Date: Wed, 27 Apr 2023 07:28:00 GMT

###

GET https://localhost:7220/weatherforecast
Cache-Control: max-age=604800
```

```
Age: 100

###
```

> **Important**
>
> When calling an API that authenticates with headers, do not commit any secrets to a source code repository. See the supported methods for storing secrets later in this article, such as ASP.NET Core user secrets, Azure Key Vault and DPAPI encryption.

Request body

Add the request body after a blank line, as shown in the following example:

HTTP

```
POST https://localhost:7220/weatherforecast
Content-Type: application/json
Accept-Language: en-US,en;q=0.5

{
    "date": "2023-05-10",
    "temperatureC": 30,
    "summary": "Warm"
}

###
```

Comments

Lines that start with either `#` or `//` are comments. These lines are ignored when Visual Studio sends HTTP requests.

Variables

A line that starts with `@` defines a variable by using the syntax `@VariableName=Value`.

Variables can be referenced in requests that are defined later in the file. They're referenced by wrapping their names in double curly braces, `{{` and `}}`. The following example shows two variables defined and used in a request:

HTTP

```
@hostname=localhost
@port=44320
GET https://{{hostname}}:{{port}}/weatherforecast
```

Variables can be defined using values of other variables that were defined earlier in the file. The following example uses one variable in the request instead of the two shown in the preceding example:

HTTP

```
@hostname=localhost
@port=44320
@host={{hostname}}:{{port}}
GET https://{{host}}/api/search/tool
```

Environment files

To give variables different values in different environments, create a file named `http-client.env.json`. Locate the file in the same directory as the `.http` file or in one of its parent directories. Here's an example of an environment file:

JSON

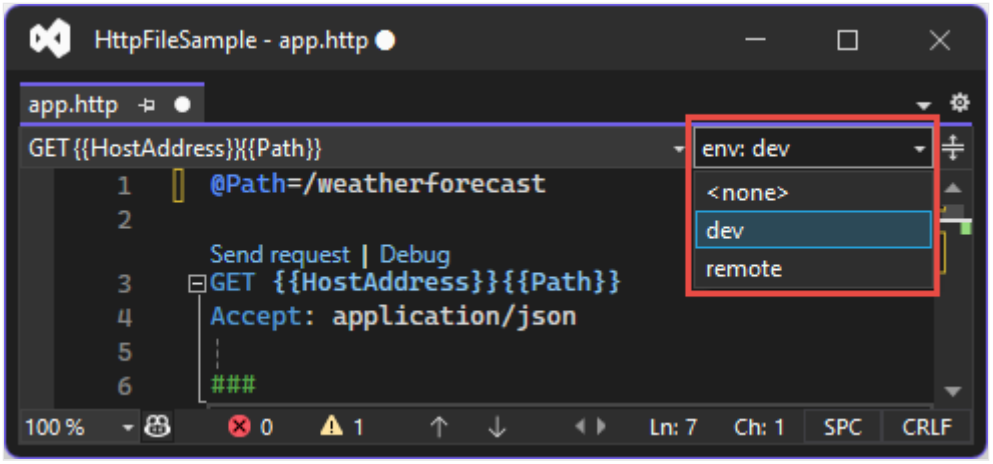
```
{
  "dev": {
    "HostAddress": "https://localhost:44320"
  },
  "remote": {
    "HostAddress": "https://contoso.com"
  }
}
```

The environment file is a JSON file that contains one or more named environments, such as "dev" and "remote" in the preceding example. Each named environment contains one or more variables, such as `HostAddress` in the preceding example. Variables from an environment file are referenced the same way as other variables, as shown in the following example:

HTTP

```
GET {{HostAddress}}/api/search/tool
```

The value that is used for the variable when sending a request is determined by an environment selector dropdown at the upper right corner of the `.http` file editor. The following screenshot shows the selector:



The environment file doesn't have to be in the project folder. Visual Studio looks for an environment file in the folder where the `.http` file exists. If it's not in that folder, Visual Studio looks through the parent directories to find it. When a file named `http-client.env.json` is found, the search ends. The file found nearest to the `.http` file is used.

After creating or editing an `.http` file, you might have to close and reopen the project to see the changes reflected in the environment selector. Press `F6` to select the environment selector.

Visual Studio displays warnings in the following situations:

- The `.http` file references a variable that isn't defined in the `.http` file or in the environment file.
- The environment file contains a variable that isn't referenced in the `.http` file.

A variable defined in an environment file can be the same as one defined in the `.http` file, or it can be different. If a variable is defined in both the `.http` file and the environment file, the value in the `.http` file overrides the value in the environment file.

User-specific environment files

A user-specific value is any value that an individual developer wants to test with but doesn't want to share with the team. Since the `http-client.env.json` file is checked in to source control by default, it wouldn't be appropriate to add user-specific values to this file. Instead, put them in a file named `http-client.env.json.user` located in the same folder as the `http-client.env.json` file. Files that end with `.user` should be

excluded from source control by default when using Visual Studio source control features.

When the `http-client.env.json` file is loaded, Visual Studio looks for a sibling `http-client.env.json.user` file. If a variable is defined in an environment in both the `http-client.env.json` file and the `http-client.env.json.user` file, the value in the `http-client.env.json.user` file wins.

Here's an example scenario that shows how a user-specific environment file works. Suppose the `.http` file has the following content:

HTTP

```
GET {{HostAddress}}/{{Path}}
Accept: application/json
```

And suppose the `http-client.env.json` file contains the following content:

JSON

```
{
  "dev": {
    "HostAddress": "https://localhost:7128",
    "Path": "/weatherforecast"
  },
  "remote": {
    "HostAddress": "https://contoso.com",
    "Path": "/weatherforecast"
  }
}
```

And suppose there's a user-specific environment file that contains the following content:

JSON

```
{
  "dev": {
    "Path": "/swagger/index.html"
  }
}
```

When the user selects the "dev" environment, the request is sent to `https://localhost:7128/swagger/index.html` because the `Path` value in the `http-client.env.json.user` file overrides the value from the `http-client.env.json` file.

With the same environment files, suppose the variables are defined in the `.http` file:

HTTP

```
@HostAddress=https://contoso.com
@Path=/weatherforecast

GET {{HostAddress}}/{{Path}}
Accept: application/json
```

In this scenario, the "dev" environment request is sent to `https://contoso.com/weatherforecast` because variable definitions in `.http` files override environment file definitions.

ASP.NET Core user secrets

To get a value from user secrets, use an environment file that is located in the same folder as the ASP.NET Core project. In the environment file, define a variable that has `provider` and `secretName` properties. Set the `provider` value to `AspnetUserSecrets` and set `secretName` to the name of the desired user secret. For example, the following environment file defines a variable named `ApiKeyDev` that gets its value from the `config:ApiKeyDev` user secret:

JSON

```
{
  "dev": {
    "ApiKeyDev": {
      "provider": "AspnetUserSecrets",
      "secretName": "config:ApiKeyDev"
    }
  }
}
```

To use this variable in the `.http` file, reference it like a standard variable. For example:

HTTP

```
GET {{HostAddress}}{{Path}}
X-API-KEY: {{ApiKeyDev}}
```

When the request is sent, the value of the `ApiKeyDev` secret is in the X-API-KEY header.

As you type in the `http` file, the editor shows a completion list for the variable name but doesn't show its value.

Azure Key Vault

Azure Key Vault is one of several key management solutions in Azure that can be used for secrets management. Of the three secrets stores currently supported for `.http` files, Key Vault is the best choice for sharing secrets across different users. The other two options—ASP.NET User Secrets and DPAPI encryption—aren't easily shared.

To use a value from Azure Key Vault, you must be signed into Visual Studio with an account that has access to the desired Key Vault. Define a variable in an environment file with the metadata to access the secret. The variable is named `AKVSecret` in the following example:

JSON

```
{
  "dev": {
    "AKVSecret": {
      "provider": "AzureKeyVault",
      "secretName": "SecretInKeyVault",
      "resourceId": "/subscriptions/3a914c59-8175a9e0e540/resourceGroups/my-key-vault-rg/providers/Microsoft.KeyVault/vaults/my-key-vault-01182024"
    }
  }
}
```

The variable `AKVSecret` pulls its value from Azure Key Vault. The following properties are defined on `AKVSecret`:

Expand table

Name	Description
provider	For Key Vault, always use `AzureKeyVault`.
secretName	Name of the secret to extract.
resourceId	Azure resource ID for the specific Key Vault to access.

The value for the `resourceId` property can be found in the Azure portal. Go to **Settings** > **Properties** to find it. For `secretName`, use the name of the secret that appears on the **Secrets** page in the Azure portal.

For example, the following `.http` file has a request that uses this secret value.

HTTP

```
GET {{HostAddress}}{{Path}}
X-AKV-SECRET: {{akvSecret}}
```

DPAPI encryption

On Windows, there is a Data Protection API (DPAPI) that can be used to encrypt sensitive data. When DPAPI is used to encrypt data, the encrypted values are always machine-specific, and they're also user-specific in `.http` files. These values can't be shared with other users.

To encrypt a value, use the following console application:

C#

```
using System.Security.Cryptography;
using System.Text;

string stringToEncrypt = "Hello, World!";
byte[] encBytes =
ProtectedData.Protect(Encoding.Unicode.GetBytes(stringToEncrypt),
optionalEntropy: null, scope: DataProtectionScope.CurrentUser);
string base64 = Convert.ToBase64String(encBytes);
Console.WriteLine(base64);
```

The preceding console application references the System.Security.Cryptography.ProtectedData NuGet package. To enable the encrypted value to work in the `.http` file, encrypt with the scope set to DataProtectionScope.CurrentUser. The encrypted value is a base64 encoded string that can be copied and pasted into the environment file.

In the environment file, create a variable that has `provider` and `value` properties. Set `provider` to `Encrypted`, and set `value` to the encrypted value. For example, the following environment file defines a variable named `dpapiValue` that gets its value from a string that was encrypted with DPAPI.

JSON

```
{
  "dev": {
    "dpapiValue": {
      "provider": "Encrypted",
      "value":
"AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAA5qwfg4+Bhk2nsy6ujgg3GAAAAAACAAAAAAAQZgAAAAE
AACAAAAAqNXhXc098k1TtKmaI4cUAbJVALMVP1zOR7mhC1RBJegAAAAOgAAAAIAACAAAABKu4E
9WC/zX5LYZZhOS2pukxMTF9R4yS+XA9HoYF98GzAAAAAzFXatt461ZnVeUWgOV8M/DkqNviWUUje
```

```
xAXOF/JfpJMw/CdsizQyESus2QjsCtZlAAAAAL7ns3u9mEk6wSMIn+KNsW/vdAw51OaI+HPVrt5v
FvXRilTtvGbU/JnxsoIHj0Z7OOxlwOSg1Qdn60zEqmlFJBg=="
    }
  }
}
```

With the preceding environment file, `dpapiValue` can be used in the `.http` file like any other variable. For example:

HTTP

```
GET {{HostAddress}}{{Path}}
X-DPAPI-Secret: {{dpapiSecret}}
```

When this request is sent, X-DPAPI-Secret has the decrypted secret value.

Environment variables

To get the value of an environment variable, use `$processEnv`. The following example puts the value of the USERNAME environment variable in the X-UserName header.

HTTP

```
GET {{HostAddress}}{{Path}}
X-UserName: {{$processEnv USERNAME}}
```

If you try to use `$processEnv` to access an environment variable that doesn't exist, the `.http` file editor displays an error message.

`.env` files

To get the value of a variable that is defined in a .env file, use `$dotenv`. The `.env` file must be in the project folder. The format for `$dotenv` is the same as for `$processEnv`. For example, if the `.env` file has this content:

```
USERNAME=userFromDotenv
```

And the `.http` file has this content:

HTTP

```
GET {{HostAddress}}{{Path}}
X-UserName: {{$dotEnv USERNAME}}
```

The `X-UserName` header will have "userFromDotenv".

When `$dotenv` is entered in the editor, it shows completions for the variables defined in the `.env` file.

> ⓘ Note
>
> `.env` files might not be excluded from source control by default, so be careful to avoid checking in any secret values.

Random integers

To generate a random integer, use `$randomInt`. The syntax is `{{$randomInt [min max]}}` where the `min` and `max` values are optional.

Dates and times

- `$datetime` generates a `datetime` string in UTC. The syntax is `{{$datetime [format] [offset option]}}` where the format and offset options are optional.
- `$localDatetime` generates a `datetime` string in the local time zone. The syntax is `{{$localDatetime [format] [offset option]}}` where the format and offset options are optional.
- `$timeStamp` generates a `timestamp` in UTC. The `timestamp` is the number of seconds since the Unix Epoch in UTC time. The syntax is `{{$timestamp [offset option]}}` where the offset option is optional.

The `[format]` option is one of `rfc1123`, `iso8601`, or a custom format in quotation marks. For example:

HTTP

```
GET https://httpbin.org/headers
X-CUSTOM: {{$datetime "dd-MM-yyyy"}}
X-ISO8601: {{$datetime iso8601}}
X-ISO8601L: {{$localDatetime iso8601}}
X-RFC1123: {{$datetime rfc1123}}
X-RFC1123L: {{$localDatetime rfc1123}}
```

Here are some sample values that the preceding examples generate:

JSON

```
{
  "headers": {
    "X-Custom": "17-01-2024",
    "X-Iso8601": "2024-01-17T22:59:55.5345770+00:00",
    "X-Iso8601L": "2024-01-17T14:59:55.5345770-08:00",
    "X-Rfc1123": "Wed, 17 Jan 2024 22:59:55 GMT",
    "X-Rfc1123L": "Wed, 17 Jan 2024 14:59:55 -08"
  }
}
```

The `[offset option]` syntax is in the form `number` `unit` where `number` is an integer and `unit` is one of the following values:

Expand table

`unit`	**Explanation**
`ms`	Milliseconds
`s`	Seconds
`m`	Minutes
`h`	Hours
`d`	Days
`w`	Weeks
`M`	Months
`y`	Years

For example:

HTTP

```
GET https://httpbin.org/headers
X-Custom-Minus-1-Year: {{$datetime "dd-MM-yyyy" -1 y}}
X-RFC1123-Plus-1-Day: {{$datetime rfc1123 1 d}}
X-Timestamp-Plus-1-Year: {{$timestamp 1 y}}
```

Here are some sample values that the preceding examples generate:

JSON

```
{
  "headers": {
    "X-Custom-Minus-1-Year": "17-01-2023",
    "X-Rfc1123-Plus-1-Day": "Thu, 18 Jan 2024 23:02:48 GMT",
    "X-Timestamp-Plus-1-Year": "1737154968"
  }
}
```

Some of the preceding examples use the free open-source website <httpbin.org>. This is a third-party website not affiliated with Microsoft. In these examples it returns a response body with the headers that were sent in the request. For information about other ways to use this resource for API testing, see the httpbin.org site's home page.

Unsupported syntax

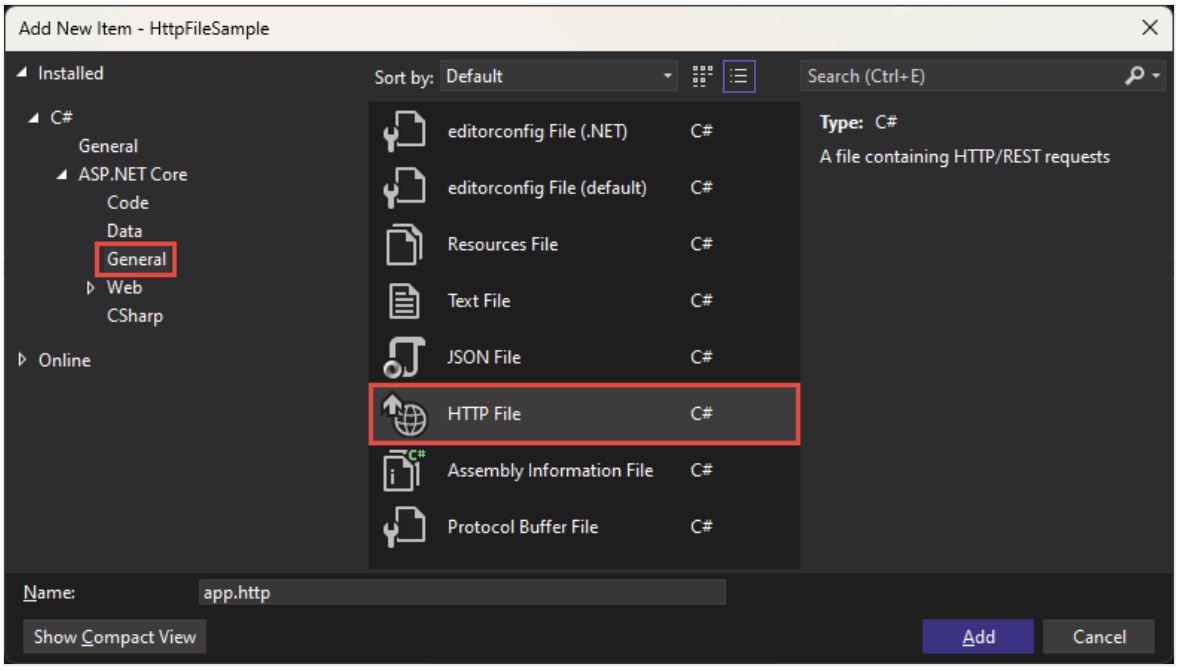
The Visual Studio 2022 `.http` file editor doesn't have all the features that the Visual Studio Code REST Client extension has. The following list includes some of the more significant features available only in the Visual Studio Code extension:

- Request line that spans more than one line
- Named requests
- Specify file path as body of the request
- Mixed format for body when using multipart/form-data
- GraphQL requests
- cURL request
- Copy/paste as cURL
- Request history
- Save response body to file
- Certificate based authentication
- Prompt variables
- Customize response preview
- Per-request settings

Create an `.http` file

- In **Solution Explorer**, right-click an ASP.NET Core project.
- In the context menu, select **Add** > **New Item**.
- In the **Add New Item** dialog, select **ASP.NET Core** > **General**.

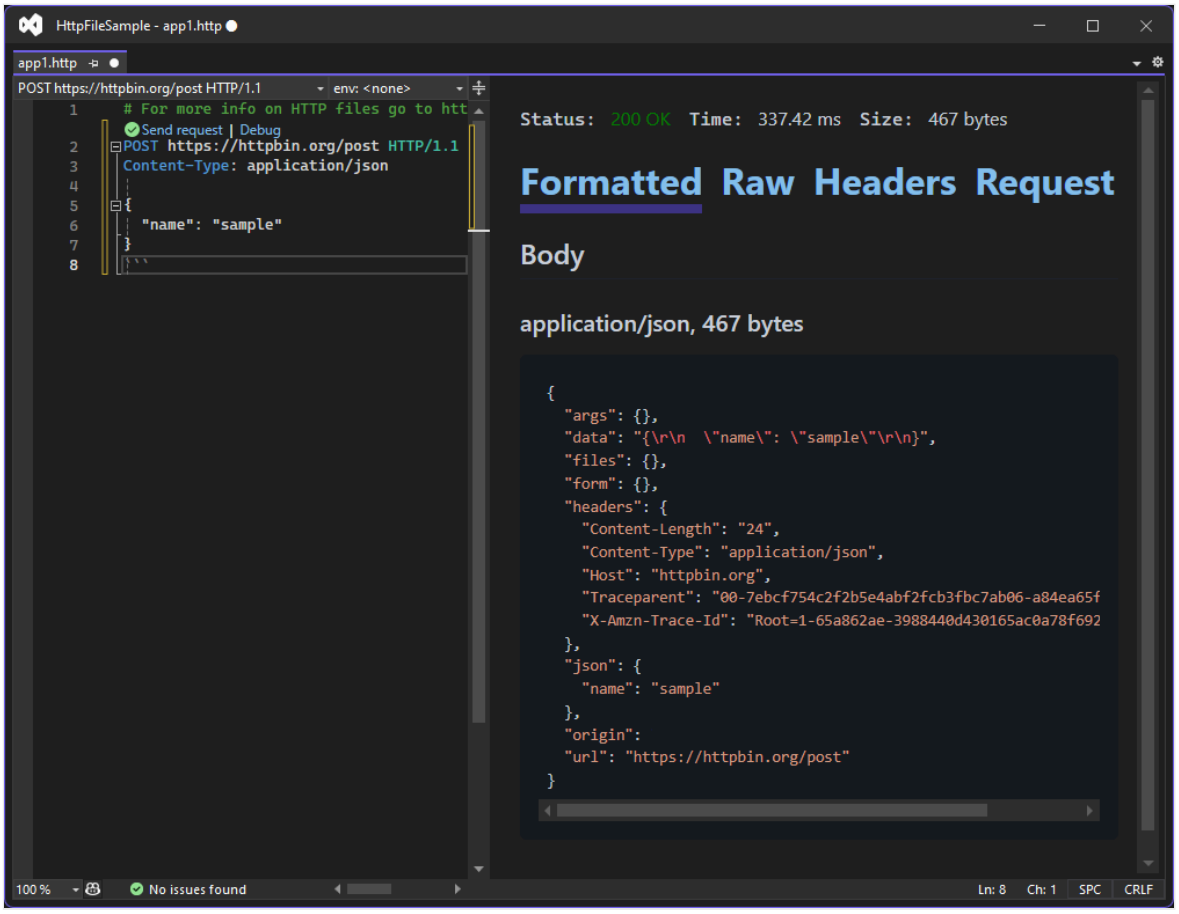
- Select **HTTP File**, and select **Add**.



Send an HTTP request

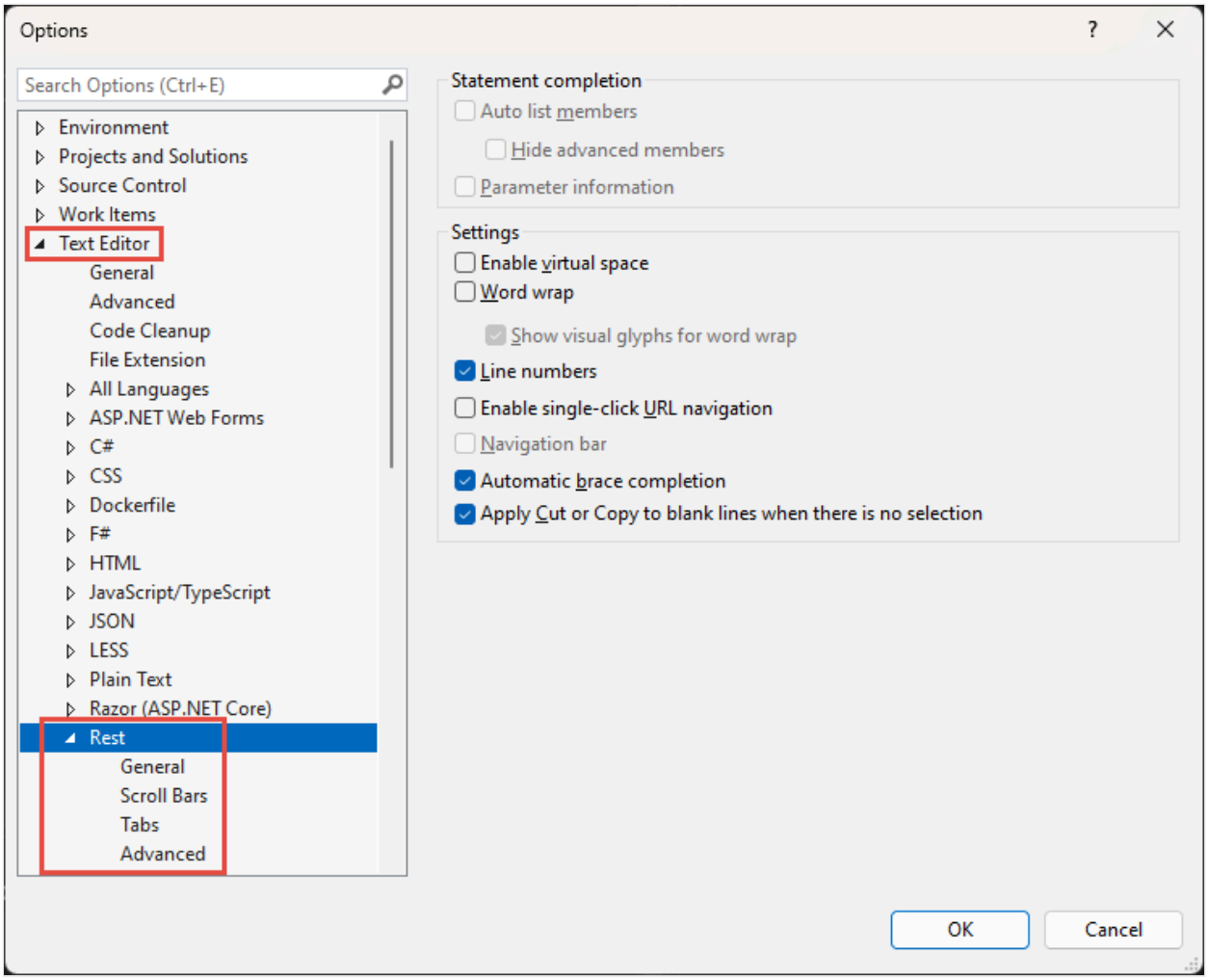
- Add at least one request to an `.http` file and save the file.
- If the request URL points to localhost and the project's port, run the project before trying to send a request to it.
- Select the `Send Request` or `Debug` link that is directly above the request to be sent.

  The request is sent to the specified URL, and the response appears in a separate pane to the right of the editor window.



`.http` file options

Some aspects of `.http` file behavior can be configured. To see what's available, go to **Tools** > **Options** > **Text Editor** > **Rest**. For example, the timeout setting can be configured on the **Advanced** tab. Here's a screenshot of the **Options** dialog:



Use Endpoints Explorer

Endpoints Explorer is a tool window that shows all the endpoints that a web API defines. The tool enables you to send requests to the endpoints by using an `.http` file.

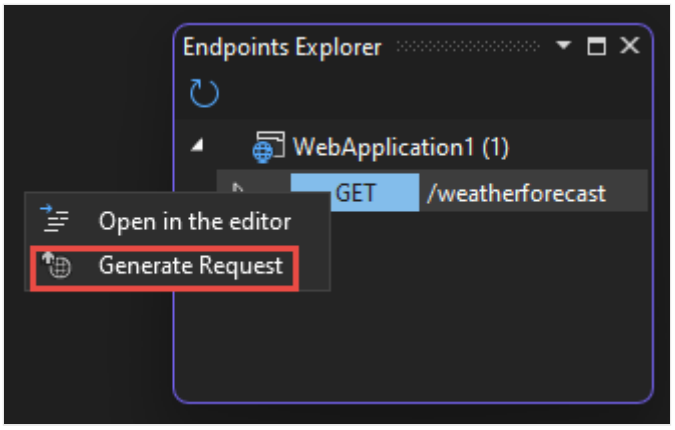
The initial set of endpoints that **Endpoints Explorer** displays are discovered statically. There are some endpoints that can't be discovered statically. For example, endpoints defined in a class library project can't be discovered until runtime. When you run or debug a web API, Visual Studio version 17.11 Preview discovers endpoints dynamically at run time also and adds those to **Endpoints Explorer**.

Open Endpoints Explorer

Select **View** > **Other Windows** > **Endpoints Explorer**.

Add a request to an `.http` file

Right-click a request in **Endpoints Explorer** and select **Generate Request**.



- If an `.http` file with the project name as the file name exists, the request is added to that file.
- Otherwise, an `.http` file is created with the project name as the file name, and the request is added to that file.

The preceding screenshot shows endpoints defined by the minimal API project template. The following example shows the request that is generated for the selected endpoint:

```http
GET {{WebApplication1_HostAddress}}/weatherforecast/
Accept: application/json

###
```

Send the request as described earlier in this article.

See also

- Endpoints Explorer window only recognizes literal strings for routes
- Web API development in Visual Studio 2022
- Visual Studio Code REST Client extension

Test web APIs with the HttpRepl

Article • 07/28/2023

The HTTP Read-Eval-Print Loop (REPL) is:

- A lightweight, cross-platform command-line tool that's supported everywhere .NET Core is supported.
- Used for making HTTP requests to test ASP.NET Core web APIs (and non-ASP.NET Core web APIs) and view their results.
- Capable of testing web APIs hosted in any environment, including localhost and Azure App Service.

The following HTTP verbs are supported:

- DELETE
- GET
- HEAD
- OPTIONS
- PATCH
- POST
- PUT

To follow along, view or download the sample ASP.NET Core web API (how to download).

Prerequisites

- .NET Core 3.1 SDK

Installation

To install the HttpRepl, run the following command:

.NET CLI

```
dotnet tool install -g Microsoft.dotnet-httprepl
```

A .NET Core Global Tool is installed from the Microsoft.dotnet-httprepl NuGet package.

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

On macOS, update the path:

```
Bash

export PATH="$HOME/.dotnet/tools:$PATH"
```

Usage

After successful installation of the tool, run the following command to start the HttpRepl:

```
Console

httprepl
```

To view the available HttpRepl commands, run one of the following commands:

```
Console

httprepl -h
```

```
Console

httprepl --help
```

The following output is displayed:

```
Console

Usage:
  httprepl [<BASE_ADDRESS>] [options]

Arguments:
  <BASE_ADDRESS> - The initial base address for the REPL.

Options:
```

```
  -h|--help - Show help information.

Once the REPL starts, these commands are valid:

Setup Commands:
Use these commands to configure the tool for your API server

connect        Configures the directory structure and base address of the
api server
set header     Sets or clears a header for all requests. e.g. `set header
content-type application/json`

HTTP Commands:
Use these commands to execute requests against your application.

GET            get - Issues a GET request
POST           post - Issues a POST request
PUT            put - Issues a PUT request
DELETE         delete - Issues a DELETE request
PATCH          patch - Issues a PATCH request
HEAD           head - Issues a HEAD request
OPTIONS        options - Issues a OPTIONS request

Navigation Commands:
The REPL allows you to navigate your URL space and focus on specific APIs
that you are working on.

ls             Show all endpoints for the current path
cd             Append the given directory to the currently selected path, or
move up a path when using `cd ..`

Shell Commands:
Use these commands to interact with the REPL shell.

clear          Removes all text from the shell
echo [on/off]  Turns request echoing on or off, show the request that was
made when using request commands
exit           Exit the shell

REPL Customization Commands:
Use these commands to customize the REPL behavior.

pref [get/set] Allows viewing or changing preferences, e.g. 'pref set
editor.command.default 'C:\\Program Files\\Microsoft VS Code\\Code.exe'`
run            Runs the script at the given path. A script is a set of
commands that can be typed with one command per line
ui             Displays the Swagger UI page, if available, in the default
browser

Use `help <COMMAND>` for more detail on an individual command. e.g. `help
get`.
For detailed tool info, see https://aka.ms/http-repl-doc.
```

The HttpRepl offers command completion. Pressing the `Tab` key iterates through the list of commands that complete the characters or API endpoint that you typed. The following sections outline the available CLI commands.

Connect to the web API

Connect to a web API by running the following command:

```
Console

httprepl <ROOT URI>
```

`<ROOT URI>` is the base URI for the web API. For example:

```
Console

httprepl https://localhost:5001
```

Alternatively, run the following command at any time while the HttpRepl is running:

```
Console

connect <ROOT URI>
```

For example:

```
Console

(Disconnected)> connect https://localhost:5001
```

Manually point to the OpenAPI description for the web API

The connect command above will attempt to find the OpenAPI description automatically. If for some reason it's unable to do so, you can specify the URI of the OpenAPI description for the web API by using the `--openapi` option:

```
Console

connect <ROOT URI> --openapi <OPENAPI DESCRIPTION ADDRESS>
```

For example:

```
Console
(Disconnected)> connect https://localhost:5001 --openapi
/swagger/v1/swagger.json
```

Enable verbose output for details on OpenAPI description searching, parsing, and validation

Specifying the `--verbose` option with the `connect` command will produce more details when the tool searches for the OpenAPI description, parses, and validates it.

```
Console
connect <ROOT URI> --verbose
```

For example:

```
Console
(Disconnected)> connect https://localhost:5001 --verbose
Checking https://localhost:5001/swagger.json... 404 NotFound
Checking https://localhost:5001/swagger/v1/swagger.json... 404 NotFound
Checking https://localhost:5001/openapi.json... Found
Parsing... Successful (with warnings)
The field 'info' in 'document' object is REQUIRED [#/info]
The field 'paths' in 'document' object is REQUIRED [#/paths]
```

Navigate the web API

View available endpoints

To list the different endpoints (controllers) at the current path of the web API address, run the `ls` or `dir` command:

```
Console
https://localhost:5001/> ls
```

The following output format is displayed:

Console

```
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/>
```

The preceding output indicates that there are two controllers available: `Fruits` and `People`. Both controllers support parameterless HTTP GET and POST operations.

Navigating into a specific controller reveals more detail. For example, the following command's output shows the `Fruits` controller also supports HTTP GET, PUT, and DELETE operations. Each of these operations expects an `id` parameter in the route:

Console

```
https://localhost:5001/fruits> ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/fruits>
```

Alternatively, run the `ui` command to open the web API's Swagger UI page in a browser. For example:

Console

```
https://localhost:5001/> ui
```

Navigate to an endpoint

To navigate to a different endpoint on the web API, run the `cd` command:

Console

```
https://localhost:5001/> cd people
```

The path following the `cd` command is case insensitive. The following output format is displayed:

Console

```
/people    [get|post]

https://localhost:5001/people>
```

Customize the HttpRepl

The HttpRepl's default colors can be customized. Additionally, a default text editor can be defined. The HttpRepl preferences are persisted across the current session and are honored in future sessions. Once modified, the preferences are stored in the following file:

Windows

%USERPROFILE%\.httpreplprefs

The *.httpreplprefs* file is loaded on startup and not monitored for changes at runtime. Manual modifications to the file take effect only after restarting the tool.

View the settings

To view the available settings, run the `pref get` command. For example:

Console

```
https://localhost:5001/> pref get
```

The preceding command displays the available key-value pairs:

Console

```
colors.json=Green
colors.json.arrayBrace=BoldCyan
colors.json.comma=BoldYellow
colors.json.name=BoldMagenta
colors.json.nameSeparator=BoldWhite
colors.json.objectBrace=Cyan
colors.protocol=BoldGreen
colors.status=BoldYellow
```

Set color preferences

Response colorization is currently supported for JSON only. To customize the default HttpRepl tool coloring, locate the key corresponding to the color to be changed. For instructions on how to find the keys, see the View the settings section. For example, change the `colors.json` key value from `Green` to `White` as follows:

Console

```
https://localhost:5001/people> pref set colors.json White
```

Only the allowed colors may be used. Subsequent HTTP requests display output with the new coloring.

When specific color keys aren't set, more generic keys are considered. To demonstrate this fallback behavior, consider the following example:

- If `colors.json.name` doesn't have a value, `colors.json.string` is used.
- If `colors.json.string` doesn't have a value, `colors.json.literal` is used.
- If `colors.json.literal` doesn't have a value, `colors.json` is used.
- If `colors.json` doesn't have a value, the command shell's default text color (`AllowedColors.None`) is used.

Set indentation size

Response indentation size customization is currently supported for JSON only. The default size is two spaces. For example:

JSON

```
[
  {
    "id": 1,
    "name": "Apple"
  },
  {
    "id": 2,
    "name": "Orange"
  },
  {
    "id": 3,
    "name": "Strawberry"
  }
]
```

To change the default size, set the `formatting.json.indentSize` key. For example, to always use four spaces:

Console

```
pref set formatting.json.indentSize 4
```

Subsequent responses honor the setting of four spaces:

JSON

```
[
    {
        "id": 1,
        "name": "Apple"
    },
    {
        "id": 2,
        "name": "Orange"
    },
    {
        "id": 3,
        "name": "Strawberry"
    }
]
```

Set the default text editor

By default, the HttpRepl has no text editor configured for use. To test web API methods requiring an HTTP request body, a default text editor must be set. The HttpRepl tool launches the configured text editor for the sole purpose of composing the request body. Run the following command to set your preferred text editor as the default:

Console

```
pref set editor.command.default "<EXECUTABLE>"
```

In the preceding command, `<EXECUTABLE>` is the full path to the text editor's executable file. For example, run the following command to set Visual Studio Code as the default text editor:

Windows

Console

```
pref set editor.command.default "C:\Program Files\Microsoft VS
Code\Code.exe"
```

To launch the default text editor with specific CLI arguments, set the `editor.command.default.arguments` key. For example, assume Visual Studio Code is the default text editor and that you always want the HttpRepl to open Visual Studio Code in a new session with extensions disabled. Run the following command:

Console

```
pref set editor.command.default.arguments "--disable-extensions --new-
window"
```

> **Tip**
>
> If your default editor is Visual Studio Code, you'll usually want to pass the `-w` or `--wait` argument to force Visual Studio Code to wait for you to close the file before returning.

Set the OpenAPI Description search paths

By default, the HttpRepl has a set of relative paths that it uses to find the OpenAPI description when executing the `connect` command without the `--openapi` option. These relative paths are combined with the root and base paths specified in the `connect` command. The default relative paths are:

- `swagger.json`
- `swagger/v1/swagger.json`
- `/swagger.json`
- `/swagger/v1/swagger.json`
- `openapi.json`
- `/openapi.json`

To use a different set of search paths in your environment, set the `swagger.searchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

Console

```
pref set swagger.searchPaths
"swagger/v2/swagger.json|swagger/v3/swagger.json"
```

Instead of replacing the default list altogether, the list can also be modified by adding or removing paths.

To add one or more search paths to the default list, set the `swagger.addToSearchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

```
Console

pref set swagger.addToSearchPaths
"openapi/v2/openapi.json|openapi/v3/openapi.json"
```

To remove one or more search paths from the default list, set the `swagger.addToSearchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

```
Console

pref set swagger.removeFromSearchPaths "swagger.json|/swagger.json"
```

Test HTTP GET requests

Synopsis

```
Console

get <PARAMETER> [-F|--no-formatting] [-h|--header] [--response:body] [--
response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

The following options are available for the `get` command:

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:

- `{header}={value}`
- `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP GET request:

1. Run the `get` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/people> get
   ```

   The preceding command displays the following output format:

   Console

   ```
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Fri, 21 Jun 2019 03:38:45 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "name": "Scott Hunter"
     },
     {
       "id": 2,
       "name": "Scott Hanselman"
     },
   ```

```
  {
    "id": 3,
    "name": "Scott Guthrie"
  }
]


https://localhost:5001/people>
```

2. Retrieve a specific record by passing a parameter to the `get` command:

```
Console

https://localhost:5001/people> get 2
```

The preceding command displays the following output format:

```
Console

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 21 Jun 2019 06:17:57 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 2,
    "name": "Scott Hanselman"
  }
]


https://localhost:5001/people>
```

Test HTTP POST requests

Synopsis

```
Console

post <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--
no-formatting] [--response] [--response:body] [--response:headers] [-s|--
streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

Indicates that no HTTP request body is needed.

Example

To issue an HTTP POST request:

1. Run the `post` command on an endpoint that supports it:

   ```
   Console

   https://localhost:5001/people> post -h Content-Type=application/json
   ```

   In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

   ```
   JSON

   {
     "id": 0,
     "name": ""
   }
   ```

   > **Tip**
   >
   > To set the default text editor, see the **Set the default text editor** section.

2. Modify the JSON template to satisfy model validation requirements:

   ```
   JSON

   {
     "id": 0,
     "name": "Scott Addie"
   }
   ```

3. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

   ```
   Console

   HTTP/1.1 201 Created
   Content-Type: application/json; charset=utf-8
   Date: Thu, 27 Jun 2019 21:24:18 GMT
   ```

```
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
  "name": "Scott Addie"
}


https://localhost:5001/people>
```

Test HTTP PUT requests

Synopsis

Console

```
put <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Example

To issue an HTTP PUT request:

1. *Optional*: Run the `get` command to view the data before modifying it:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:07:32 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
   ```

```
  {
    "id": 2,
    "data": "Orange"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]
```

2. Run the `put` command on an endpoint that supports it:

```
Console

https://localhost:5001/fruits> put 2 -h Content-Type=application/json
```

In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

```
JSON

{
  "id": 0,
  "name": ""
}
```

> **Tip**
>
> To set the default text editor, see the **Set the default text editor** section.

3. Modify the JSON template to satisfy model validation requirements:

```
JSON

{
  "id": 2,
  "name": "Cherry"
}
```

4. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

```
Console
```

```
[main 2019-06-28T17:27:01.805Z] update#setState idle
HTTP/1.1 204 No Content
Date: Fri, 28 Jun 2019 17:28:21 GMT
Server: Kestrel
```

5. *Optional*: Issue a `get` command to see the modifications. For example, if you typed "Cherry" in the text editor, a `get` returns the following output:

Console

```
https://localhost:5001/fruits> get
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Sat, 22 Jun 2019 00:08:20 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Cherry"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]


https://localhost:5001/fruits>
```

Test HTTP DELETE requests

Synopsis

Console

```
delete <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP DELETE request:

1. *Optional*: Run the `get` command to view the data before modifying it:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:07:32 GMT
   Server: Kestrel
   Transfer-Encoding: chunked
   ```

```
[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Orange"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]
```

2. Run the `delete` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/fruits> delete 2
   ```

   The preceding command displays the following output format:

   Console

   ```
   HTTP/1.1 204 No Content
   Date: Fri, 28 Jun 2019 17:36:42 GMT
   Server: Kestrel
   ```

3. *Optional*: Issue a `get` command to see the modifications. In this example, a `get` returns the following output:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:16:30 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
     {
       "id": 3,
       "data": "Strawberry"
     }
   ```

```
]


https://localhost:5001/fruits>
```

Test HTTP PATCH requests

Synopsis

Console

```
patch <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't

exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Test HTTP HEAD requests

Synopsis

Console

```
head <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
    - `{header}={value}`
    - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Test HTTP OPTIONS requests

Synopsis

Console

```
options <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Set HTTP request headers

To set an HTTP request header, use one of the following approaches:

- Set inline with the HTTP request. For example:

  Console

  ```
  https://localhost:5001/people> post -h Content-Type=application/json
  ```

  With the preceding approach, each distinct HTTP request header requires its own `-h` option.

- Set before sending the HTTP request. For example:

  Console

  ```
  https://localhost:5001/people> set header Content-Type application/json
  ```

  When setting the header before sending a request, the header remains set for the duration of the command shell session. To clear the header, provide an empty value. For example:

  Console

```
https://localhost:5001/people> set header Content-Type
```

Test secured endpoints

The HttpRepl supports the testing of secured endpoints in the following ways:

- Via the default credentials of the logged in user.
- Through the use of HTTP request headers.

Default credentials

Consider a web API you're testing that's hosted in IIS and secured with Windows authentication. You want the credentials of the user running the tool to flow across to the HTTP endpoints being tested. To pass the default credentials of the logged in user:

1. Set the `httpClient.useDefaultCredentials` preference to `true`:

   Console

   ```
   pref set httpClient.useDefaultCredentials true
   ```

2. Exit and restart the tool before sending another request to the web API.

Default proxy credentials

Consider a scenario in which the web API you're testing is behind a proxy secured with Windows authentication. You want the credentials of the user running the tool to flow to the proxy. To pass the default credentials of the logged in user:

1. Set the `httpClient.proxy.useDefaultCredentials` preference to `true`:

   Console

   ```
   pref set httpClient.proxy.useDefaultCredentials true
   ```

2. Exit and restart the tool before sending another request to the web API.

HTTP request headers

Examples of supported authentication and authorization schemes include:

- basic authentication
- JWT bearer tokens
- digest authentication

For example, you can send a bearer token to an endpoint with the following command:

Console

```
set header Authorization "bearer <TOKEN VALUE>"
```

To access an Azure-hosted endpoint or to use the Azure REST API, you need a bearer token. Use the following steps to obtain a bearer token for your Azure subscription via the Azure CLI. The HttpRepl sets the bearer token in an HTTP request header. A list of Azure App Service Web Apps is retrieved.

1. Sign in to Azure:

   Azure CLI

   ```
   az login
   ```

2. Get your subscription ID with the following command:

   Azure CLI

   ```
   az account show --query id
   ```

3. Copy your subscription ID and run the following command:

   Azure CLI

   ```
   az account set --subscription "<SUBSCRIPTION ID>"
   ```

4. Get your bearer token with the following command:

   Azure CLI

   ```
   az account get-access-token --query accessToken
   ```

5. Connect to the Azure REST API via the HttpRepl:

   Console

```
httprepl https://management.azure.com
```

6. Set the `Authorization` HTTP request header:

```
Console

https://management.azure.com/> set header Authorization "bearer <ACCESS
TOKEN>"
```

7. Navigate to the subscription:

```
Console

https://management.azure.com/> cd subscriptions/<SUBSCRIPTION ID>
```

8. Get a list of your subscription's Azure App Service Web Apps:

```
Console

https://management.azure.com/subscriptions/{SUBSCRIPTION ID}> get
providers/Microsoft.Web/sites?api-version=2016-08-01
```

The following response is displayed:

```
Console

HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Length: 35948
Content-Type: application/json; charset=utf-8
Date: Thu, 19 Sep 2019 23:04:03 GMT
Expires: -1
Pragma: no-cache
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
x-ms-correlation-request-id: <em>xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx</em>
x-ms-original-request-ids: <em>xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx;xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx</em>
x-ms-ratelimit-remaining-subscription-reads: 11999
x-ms-request-id: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
x-ms-routing-request-id: WESTUS:xxxxxxxxxxxxxxxx:xxxxxxxx-xxxx-xxxx-
xxxx-xxxxxxxxxx
{
  "value": [
    <AZURE RESOURCES LIST>
```

```
  ]
}
```

Toggle HTTP request display

By default, display of the HTTP request being sent is suppressed. It's possible to change the corresponding setting for the duration of the command shell session.

Enable request display

View the HTTP request being sent by running the `echo on` command. For example:

Console

```
https://localhost:5001/people> echo on
Request echoing is on
```

Subsequent HTTP requests in the current session display the request headers. For example:

Console

```
https://localhost:5001/people> post

[main 2019-06-28T18:50:11.930Z] update#setState idle
Request to https://localhost:5001...

POST /people HTTP/1.1
Content-Length: 41
Content-Type: application/json
User-Agent: HTTP-REPL

{
  "id": 0,
  "name": "Scott Addie"
}

Response from https://localhost:5001...

HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Fri, 28 Jun 2019 18:50:21 GMT
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
```

```
  "name": "Scott Addie"
}


https://localhost:5001/people>
```

Disable request display

Suppress display of the HTTP request being sent by running the `echo off` command. For example:

Console

```
https://localhost:5001/people> echo off
Request echoing is off
```

Run a script

If you frequently execute the same set of HttpRepl commands, consider storing them in a text file. Commands in the file take the same form as commands executed manually on the command line. The commands can be executed in a batched fashion using the `run` command. For example:

1. Create a text file containing a set of newline-delimited commands. To illustrate, consider a *people-script.txt* file containing the following commands:

   text

   ```
   set base https://localhost:5001
   ls
   cd People
   ls
   get 1
   ```

2. Execute the `run` command, passing in the text file's path. For example:

   Console

   ```
   https://localhost:5001/> run C:\http-repl-scripts\people-script.txt
   ```

   The following output appears:

   Console

```
https://localhost:5001/> set base https://localhost:5001
Using OpenAPI description at
https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/> ls
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/> cd People
/People    [get|post]

https://localhost:5001/People> ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/People> get 1
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 12 Jul 2019 19:20:10 GMT
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 1,
  "name": "Scott Hunter"
}


https://localhost:5001/People>
```

Clear the output

To remove all output written to the command shell by the HttpRepl tool, run the `clear` or `cls` command. To illustrate, imagine the command shell contains the following output:

Console

```
httprepl https://localhost:5001
(Disconnected)> set base "https://localhost:5001"
Using OpenAPI description at https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/> ls
.        []
Fruits   [get|post]
People   [get|post]
```

```
https://localhost:5001/>
```

Run the following command to clear the output:

Console

```
https://localhost:5001/> clear
```

After running the preceding command, the command shell contains only the following output:

Console

```
https://localhost:5001/>
```

Additional resources

- REST API requests
- HttpRepl GitHub repository
- Configure Visual Studio to launch HttpRepl
- Configure Visual Studio Code to launch HttpRepl
- Configure Visual Studio for Mac to launch HttpRepl

HttpRepl telemetry

Article • 06/17/2024

The HttpRepl includes a telemetry feature that collects usage data. It's important that the HttpRepl team understands how the tool is used so it can be improved.

How to opt out

The HttpRepl telemetry feature is enabled by default. To opt out of the telemetry feature, set the `DOTNET_HTTPREPL_TELEMETRY_OPTOUT` environment variable to `1` or `true`.

Disclosure

The HttpRepl displays text similar to the following when you first run the tool. Text may vary slightly depending on the version of the tool you're running. This "first run" experience is how Microsoft notifies you about data collection.

Console

```
Telemetry
---------
The .NET tools collect usage data in order to help us improve your
experience. It is collected by Microsoft and shared with the community. You
can opt-out of telemetry by setting the DOTNET_HTTPREPL_TELEMETRY_OPTOUT
environment variable to '1' or 'true' using your favorite shell.
```

To suppress the "first run" experience text, set the `DOTNET_HTTPREPL_SKIP_FIRST_TIME_EXPERIENCE` environment variable to `1` or `true`.

Data points

The telemetry feature doesn't:

- Collect personal data, such as usernames, email addresses, or URLs.
- Scan your HTTP requests or responses.

The data is sent securely to Microsoft servers and held under restricted access.

Protecting your privacy is important to us. If you suspect the telemetry feature is collecting sensitive data or the data is being insecurely or inappropriately handled, take one of the following actions:

- File an issue in the dotnet/httprepl repository.
- Send an email to dotnet@microsoft.com for investigation.

The telemetry feature collects the following data.

Expand table

.NET SDK versions	Data
>=5.0	Timestamp of invocation.
>=5.0	Three-octet IP address used to determine the geographical location.
>=5.0	Operating system and version.
>=5.0	Runtime ID (RID) the tool is running on.
>=5.0	Whether the tool is running in a container.
>=5.0	Hashed Media Access Control (MAC) address: a cryptographically (SHA256) hashed and unique ID for a machine.
>=5.0	Kernel version.
>=5.0	HttpRepl version.
>=5.0	Whether the tool was started with `help`, `run`, or `connect` arguments. Actual argument values aren't collected.
>=5.0	Command invoked (for example, `get`) and whether it succeeded.
>=5.0	For the `connect` command, whether the `root`, `base`, or `openapi` arguments were supplied. Actual argument values aren't collected.
>=5.0	For the `pref` command, whether a `get` or `set` was issued and which preference was accessed. If not a well-known preference, the name is hashed. The value isn't collected.
>=5.0	For the `set header` command, the header name being set. If not a well-known header, the name is hashed. The value isn't collected.
>=5.0	For the `connect` command, whether a special case for `dotnet new webapi` was used and, whether it was bypassed via preference.
>=5.0	For all HTTP commands (for example, GET, POST, PUT), whether each of the options was specified. The values of the options aren't collected.

Additional resources

- .NET Core SDK telemetry
- .NET CLI telemetry data

Unit and integration tests in Minimal API apps

Article • 07/26/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Fiyaz Bin Hasan, and Rick Anderson

Introduction to integration tests

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

- Database
- File system
- Network appliances
- Request-response pipeline

Unit tests use fabricated components, known as *fakes* or *mock objects*, in place of infrastructure components.

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Therefore, limit the use of integration tests to the most important infrastructure scenarios. If a behavior can be tested using either a unit test or an integration test,

choose the unit test.

In discussions of integration tests, the tested project is frequently called the ***System Under Test***, or "**SUT**" for short. "SUT" is used throughout this article to refer to the ASP.NET Core app being tested.

Don't write integration tests for every permutation of data and file access with databases and file systems. Regardless of how many places across an app interact with databases and file systems, a focused set of read, write, update, and delete integration tests are usually capable of adequately testing database and file system components. Use unit tests for routine tests of method logic that interact with these components. In unit tests, the use of infrastructure fakes or mocks result in faster test execution.

ASP.NET Core integration tests

Integration tests in ASP.NET Core require the following:

- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual *Arrange*, *Act*, and *Assert* test steps:

1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The *Arrange* test step is executed: The test app prepares a request.
4. The *Act* test step is executed: The client submits the request and receives the response.
5. The *Assert* test step is executed: The *actual* response is validated as a *pass* or *fail* based on an *expected* response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For example, a different database or different app settings might be used for the tests.

Infrastructure components, such as the test web host and in-memory test server (TestServer), are provided or managed by the Microsoft.AspNetCore.Mvc.Testing package. Use of this package streamlines test creation and execution.

The `Microsoft.AspNetCore.Mvc.Testing` package handles the following tasks:

- Copies the dependencies file (`.deps`) from the SUT into the test project's `bin` directory.
- Sets the content root to the SUT's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory class to streamline bootstrapping the SUT with `TestServer`.

The unit tests documentation describes how to set up a test project and test runner, along with detailed instructions on how to run tests and recommendations for how to name tests and test classes.

Separate unit tests from integration tests into different projects. Separating the tests:

- Helps ensure that infrastructure testing components aren't accidentally included in the unit tests.
- Allows control over which set of tests are run.

The sample code on GitHub provides an example of unit and integration tests on a Minimal API app.

IResult implementation types

Public IResult implementation types in the Microsoft.AspNetCore.Http.HttpResults namespace can be used to unit test minimal route handlers when using named methods instead of lambdas.

The following code uses the NotFound<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsNotFoundIfNotExists()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var notFoundResult = (NotFound) result.Result;
```

```
    Assert.NotNull(notFoundResult);
}
```

The following code uses the Ok<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsTodoFromDatabase()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    context.Todos.Add(new Todo
    {
        Id = 1,
        Title = "Test title",
        Description = "Test description",
        IsDone = false
    });

    await context.SaveChangesAsync();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var okResult = (Ok<Todo>)result.Result;

    Assert.NotNull(okResult.Value);
    Assert.Equal(1, okResult.Value.Id);
}
```

Additional Resources

- Basic authentication tests is not a .NET repository but was written by a member of the .NET team. It provides examples of basic authentication testing.
- View or download sample code
- Authentication and authorization in minimal APIs
- Use port tunneling Visual Studio to debug web APIs
- Test controller logic in ASP.NET Core
- Razor Pages unit tests in ASP.NET Core

Test Razor components in ASP.NET Core Blazor

Article • 03/08/2024

> **ⓘ Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 9 version of this article**.

By Egil Hansen

Testing Razor components is an important aspect of releasing stable and maintainable Blazor apps.

To test a Razor component, the *component under test* (CUT) is:

- Rendered with relevant input for the test.
- Depending on the type of test performed, possibly subject to interaction or modification. For example, event handlers can be triggered, such as an `onclick` event for a button.
- Inspected for expected values. A test passes when one or more inspected values matches the expected values for the test.

Test approaches

Two common approaches for testing Razor components are end-to-end (E2E) testing and unit testing:

- **Unit testing**: Unit tests are written with a unit testing library that provides:
  - Component rendering.
  - Inspection of component output and state.
  - Triggering of event handlers and life cycle methods.
  - Assertions that component behavior is correct.

  bUnit is an example of a library that enables Razor component unit testing.

- **E2E testing**: A test runner runs a Blazor app containing the CUT and automates a browser instance. The testing tool inspects and interacts with the CUT through the browser. Playwright for .NET is an example of an E2E testing framework that can be used with Blazor apps.

In unit testing, only the Razor component (Razor/C#) is involved. External dependencies, such as services and JS interop, must be mocked. In E2E testing, the Razor component and all of its auxiliary infrastructure are part of the test, including CSS, JS, and the DOM and browser APIs.

Test scope describes how extensive the tests are. Test scope typically has an influence on the speed of the tests. Unit tests run on a subset of the app's subsystems and usually execute in milliseconds. E2E tests, which test a broad group of the app's subsystems, can take several seconds to complete.

Unit testing also provides access to the instance of the CUT, allowing for inspection and verification of the component's internal state. This normally isn't possible in E2E testing.

With regard to the component's environment, E2E tests must make sure that the expected environmental state has been reached before verification starts. Otherwise, the result is unpredictable. In unit testing, the rendering of the CUT and the life cycle of the test are more integrated, which improves test stability.

E2E testing involves launching multiple processes, network and disk I/O, and other subsystem activity that often lead to poor test reliability. Unit tests are typically insulated from these sorts of issues.

The following table summarizes the difference between the two testing approaches.

Expand table

Capability	Unit testing	E2E testing
Test scope	Razor component (Razor/C#) only	Razor component (Razor/C#) with CSS/JS
Test execution time	Milliseconds	Seconds
Access to the component instance	Yes	No
Sensitive to the environment	No	Yes
Reliability	More reliable	Less reliable

Choose the most appropriate test approach

Consider the scenario when choosing the type of testing to perform. Some considerations are described in the following table.

Expand table

Scenario	Suggested approach	Remarks
Component without JS interop logic	Unit testing	When there's no dependency on JS interop in a Razor component, the component can be tested without access to JS or the DOM API. In this scenario, there are no disadvantages to choosing unit testing.
Component with simple JS interop logic	Unit testing	It's common for components to query the DOM or trigger animations through JS interop. Unit testing is usually preferred in this scenario, since it's straightforward to mock the JS interaction through the IJSRuntime interface.
Component that depends on complex JS code	Unit testing and separate JS testing	If a component uses JS interop to call a large or complex JS library but the interaction between the Razor component and JS library is simple, then the best approach is likely to treat the component and JS library or code as two separate parts and test each individually. Test the Razor component with a unit testing library, and test the JS with a JS testing library.
Component with logic that depends on JS manipulation of the browser DOM	E2E testing	When a component's functionality is dependent on JS and its manipulation of the DOM, verify both the JS and Blazor code together in an E2E test. This is the approach that the Blazor framework developers have taken with Blazor's browser rendering logic, which has tightly-coupled C# and JS code. The C# and JS code must work together to correctly render Razor components in a browser.
Component that depends on 3rd party class library with hard-to-mock dependencies	E2E testing	When a component's functionality is dependent on a 3rd party class library that has hard-to-mock dependencies, such as JS interop, E2E testing might be the only option to test the component.

Test components with bUnit

There's no official Microsoft testing framework for Blazor, but the community-driven project bUnit provides a convenient way to unit test Razor components.

> **Note**
>
> bUnit is a third-party testing library and isn't supported or maintained by Microsoft.

bUnit works with general-purpose testing frameworks, such as MSTest, NUnit, and xUnit. These testing frameworks make bUnit tests look and feel like regular unit tests. bUnit tests integrated with a general-purpose testing framework are ordinarily executed with:

- Visual Studio's Test Explorer.
- dotnet test CLI command in a command shell.
- An automated DevOps testing pipeline.

> **Note**
>
> Test concepts and test implementations across different test frameworks are similar but not identical. Refer to the test framework's documentation for guidance.

The following demonstrates the structure of a bUnit test on the `Counter` component in an app based on a Blazor project template. The `Counter` component displays and increments a counter based on the user selecting a button in the page:

razor

```razor
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

The following bUnit test verifies that the CUT's counter is incremented correctly when the button is selected:

razor

```
@code {
    [Fact]
    public void CounterShouldIncrementWhenClicked()
    {
        // Arrange
        using var ctx = new TestContext();
        var cut = ctx.Render(@<Counter />);
        var paraElm = cut.Find("p");

        // Act
        cut.Find("button").Click();

        // Assert
        var paraElmText = paraElm.TextContent;
        paraElm.MarkupMatches("Current count: 1");
    }
}
```

Tests can also be written in a C# class file:

C#

```
public class CounterTests
{
    [Fact]
    public void CounterShouldIncrementWhenClicked()
    {
        // Arrange
        using var ctx = new TestContext();
        var cut = ctx.RenderComponent<Counter>();
        var paraElm = cut.Find("p");

        // Act
        cut.Find("button").Click();

        // Assert
        var paraElmText = paraElm.TextContent;
        paraElmText.MarkupMatches("Current count: 1");
    }
}
```

The following actions take place at each step of the test:

- *Arrange*: The `Counter` component is rendered using bUnit's `TestContext`. The CUT's paragraph element (`<p>`) is found and assigned to `paraElm`. In Razor syntax,

a component can be passed as a RenderFragment to bUnit.

- *Act*: The button's element (`<button>`) is located and selected by calling `Click`, which should increment the counter and update the content of the paragraph tag (`<p>`). The paragraph element text content is obtained by calling `TextContent`.
- *Assert*: `MarkupMatches` is called on the text content to verify that it matches the expected string, which is `Current count: 1`.

> **Note**
>
> The `MarkupMatches` assert method differs from a regular string comparison assertion (for example, `Assert.Equal("Current count: 1", paraElmText);`). `MarkupMatches` performs a semantic comparison of the input and expected HTML markup. A semantic comparison is aware of HTML semantics, meaning things like insignificant whitespace is ignored. This results in more stable tests. For more information, see Customizing the Semantic HTML Comparison.

Additional resources

- Getting Started with bUnit: bUnit instructions include guidance on creating a test project, referencing testing framework packages, and building and running tests.
- How to create maintainable and testable Blazor components - Egil Hansen - NDC Oslo 2022

Razor Pages unit tests in ASP.NET Core

Article • 09/17/2023

ASP.NET Core supports unit tests of Razor Pages apps. Tests of the data access layer (DAL) and page models help ensure:

- Parts of a Razor Pages app work independently and together as a unit during app construction.
- Classes and methods have limited scopes of responsibility.
- Additional documentation exists on how the app should behave.
- Regressions, which are errors brought about by updates to the code, are found during automated building and deployment.

This topic assumes that you have a basic understanding of Razor Pages apps and unit tests. If you're unfamiliar with Razor Pages apps or test concepts, see the following topics:

- Introduction to Razor Pages in ASP.NET Core
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Unit testing C# in .NET Core using dotnet test and xUnit

View or download sample code (how to download)

The sample project is composed of two apps:

Expand table

App	Project folder	Description
Message app	*src/RazorPagesTestSample*	Allows a user to add a message, delete one message, delete all messages, and analyze messages (find the average number of words per message).
Test app	*tests/RazorPagesTestSample.Tests*	Used to unit test the DAL and Index page model of the message app.

The tests can be run using the built-in test features of an IDE, such as Visual Studio. If using Visual Studio Code or the command line, execute the following command at a command prompt in the *tests/RazorPagesTestSample.Tests* folder:

.NET CLI

```
dotnet test
```

Message app organization

The message app is a Razor Pages message system with the following characteristics:

- The Index page of the app (`Pages/Index.cshtml` and `Pages/Index.cshtml.cs`) provides a UI and page model methods to control the addition, deletion, and analysis of messages (find the average number of words per message).
- A message is described by the `Message` class (`Data/Message.cs`) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using Entity Framework's in-memory database†.
- The app contains a DAL in its database context class, `AppDbContext` (`Data/AppDbContext.cs`). The DAL methods are marked `virtual`, which allows mocking the methods for use in the tests.
- If the database is empty on app startup, the message store is initialized with three messages. These *seeded messages* are also used in tests.

†The EF topic, Test with InMemory, explains how to use an in-memory database for tests with MSTest. This topic uses the xUnit test framework. Test concepts and test implementations across different test frameworks are similar but not identical.

Although the sample app doesn't use the repository pattern and isn't an effective example of the Unit of Work (UoW) pattern, Razor Pages supports these patterns of development. For more information, see Designing the infrastructure persistence layer and Test controller logic in ASP.NET Core (the sample implements the repository pattern).

Test app organization

The test app is a console app inside the *tests/RazorPagesTestSample.Tests* folder.

Expand table

Test app folder	Description
UnitTests	• `DataAccessLayerTest.cs` contains the unit tests for the DAL. • `IndexPageTests.cs` contains the unit tests for the Index page model.
Utilities	Contains the `TestDbContextOptions` method used to create new database context options for each DAL unit test so that the database is reset to its baseline condition for each test.

The test framework is xUnit. The object mocking framework is Moq.

Unit tests of the data access layer (DAL)

The message app has a DAL with four methods contained in the `AppDbContext` class (`src/RazorPagesTestSample/Data/AppDbContext.cs`). Each method has one or two unit tests in the test app.

Expand table

DAL method	Function
`GetMessagesAsync`	Obtains a `List<Message>` from the database sorted by the `Text` property.
`AddMessageAsync`	Adds a `Message` to the database.
`DeleteAllMessagesAsync`	Deletes all `Message` entries from the database.
`DeleteMessageAsync`	Deletes a single `Message` from the database by `Id`.

Unit tests of the DAL require DbContextOptions when creating a new `AppDbContext` for each test. One approach to creating the `DbContextOptions` for each test is to use a DbContextOptionsBuilder:

C#

```
var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
    .UseInMemoryDatabase("InMemoryDb");

using (var db = new AppDbContext(optionsBuilder.Options))
{
    // Use the db here in the unit test.
}
```

The problem with this approach is that each test receives the database in whatever state the previous test left it. This can be problematic when trying to write atomic unit tests that don't interfere with each other. To force the `AppDbContext` to use a new database context for each test, supply a `DbContextOptions` instance that's based on a new service provider. The test app shows how to do this using its `Utilities` class method `TestDbContextOptions` (`tests/RazorPagesTestSample.Tests/Utilities/Utilities.cs`):

C#

```
public static DbContextOptions<AppDbContext> TestDbContextOptions()
{
    // Create a new service provider to create a new in-memory database.
    var serviceProvider = new ServiceCollection()
        .AddEntityFrameworkInMemoryDatabase()
        .BuildServiceProvider();

    // Create a new options instance using an in-memory database and
    // IServiceProvider that the context should resolve all of its
    // services from.
    var builder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb")
        .UseInternalServiceProvider(serviceProvider);

    return builder.Options;
}
```

Using the `DbContextOptions` in the DAL unit tests allows each test to run atomically with a fresh database instance:

C#

```
using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
{
    // Use the db here in the unit test.
}
```

Each test method in the `DataAccessLayerTest` class (`UnitTests/DataAccessLayerTest.cs`) follows a similar Arrange-Act-Assert pattern:

1. Arrange: The database is configured for the test and/or the expected outcome is defined.
2. Act: The test is executed.
3. Assert: Assertions are made to determine if the test result is a success.

For example, the `DeleteMessageAsync` method is responsible for removing a single message identified by its `Id` (`src/RazorPagesTestSample/Data/AppDbContext.cs`):

C#

```
public async virtual Task DeleteMessageAsync(int id)
{
    var message = await Messages.FindAsync(id);

    if (message != null)
    {
        Messages.Remove(message);
        await SaveChangesAsync();
```

```
    }
}
```

There are two tests for this method. One test checks that the method deletes a message when the message is present in the database. The other method tests that the database doesn't change if the message `Id` for deletion doesn't exist. The `DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound` method is shown below:

C#

```
[Fact]
public async Task DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var seedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(seedMessages);
        await db.SaveChangesAsync();
        var recId = 1;
        var expectedMessages =
            seedMessages.Where(message => message.Id != recId).ToList();

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

First, the method performs the Arrange step, where preparation for the Act step takes place. The seeding messages are obtained and held in `seedMessages`. The seeding messages are saved into the database. The message with an `Id` of `1` is set for deletion. When the `DeleteMessageAsync` method is executed, the expected messages should have all of the messages except for the one with an `Id` of `1`. The `expectedMessages` variable represents this expected outcome.

C#

```
// Arrange
var seedMessages = AppDbContext.GetSeedingMessages();
await db.AddRangeAsync(seedMessages);
await db.SaveChangesAsync();
var recId = 1;
```

```
var expectedMessages =
    seedMessages.Where(message => message.Id != recId).ToList();
```

The method acts: The `DeleteMessageAsync` method is executed passing in the `recId` of `1`:

C#

```
// Act
await db.DeleteMessageAsync(recId);
```

Finally, the method obtains the `Messages` from the context and compares it to the `expectedMessages` asserting that the two are equal:

C#

```
// Assert
var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

In order to compare that the two `List<Message>` are the same:

- The messages are ordered by `Id`.
- Message pairs are compared on the `Text` property.

A similar test method, `DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound` checks the result of attempting to delete a message that doesn't exist. In this case, the expected messages in the database should be equal to the actual messages after the `DeleteMessageAsync` method is executed. There should be no change to the database's content:

C#

```
[Fact]
public async Task
DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var expectedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(expectedMessages);
        await db.SaveChangesAsync();
        var recId = 4;
```

```
        // Act
        try
        {
            await db.DeleteMessageAsync(recId);
        }
        catch
        {
            // recId doesn't exist
        }

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

Unit tests of the page model methods

Another set of unit tests is responsible for tests of page model methods. In the message app, the Index page models are found in the `IndexModel` class in `src/RazorPagesTestSample/Pages/Index.cshtml.cs`.

Expand table

Page model method	Function
`OnGetAsync`	Obtains the messages from the DAL for the UI using the `GetMessagesAsync` method.
`OnPostAddMessageAsync`	If the ModelState is valid, calls `AddMessageAsync` to add a message to the database.
`OnPostDeleteAllMessagesAsync`	Calls `DeleteAllMessagesAsync` to delete all of the messages in the database.
`OnPostDeleteMessageAsync`	Executes `DeleteMessageAsync` to delete a message with the `Id` specified.
`OnPostAnalyzeMessagesAsync`	If one or more messages are in the database, calculates the average number of words per message.

The page model methods are tested using seven tests in the `IndexPageTests` class (`tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs`). The tests use the familiar Arrange-Assert-Act pattern. These tests focus on:

- Determining if the methods follow the correct behavior when the ModelState is invalid.
- Confirming the methods produce the correct IActionResult.
- Checking that property value assignments are made correctly.

This group of tests often mock the methods of the DAL to produce expected data for the Act step where a page model method is executed. For example, the `GetMessagesAsync` method of the `AppDbContext` is mocked to produce output. When a page model method executes this method, the mock returns the result. The data doesn't come from the database. This creates predictable, reliable test conditions for using the DAL in the page model tests.

The `OnGetAsync_PopulatesThePageModel_WithAListOfMessages` test shows how the `GetMessagesAsync` method is mocked for the page model:

```
C#

var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
var expectedMessages = AppDbContext.GetSeedingMessages();
mockAppDbContext.Setup(
    db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
var pageModel = new IndexModel(mockAppDbContext.Object);
```

When the `OnGetAsync` method is executed in the Act step, it calls the page model's `GetMessagesAsync` method.

Unit test Act step (`tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs`):

```
C#

// Act
await pageModel.OnGetAsync();
```

`IndexPage` page model's `OnGetAsync` method (`src/RazorPagesTestSample/Pages/Index.cshtml.cs`):

```
C#

public async Task OnGetAsync()
{
    Messages = await _db.GetMessagesAsync();
}
```

The `GetMessagesAsync` method in the DAL doesn't return the result for this method call. The mocked version of the method returns the result.

In the `Assert` step, the actual messages (`actualMessages`) are assigned from the `Messages` property of the page model. A type check is also performed when the messages are assigned. The expected and actual messages are compared by their `Text` properties. The test asserts that the two `List<Message>` instances contain the same messages.

C#

```
// Assert
var actualMessages = Assert.IsAssignableFrom<List<Message>>
(pageModel.Messages);
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

Other tests in this group create page model objects that include the DefaultHttpContext, the ModelStateDictionary, an ActionContext to establish the `PageContext`, a `ViewDataDictionary`, and a `PageContext`. These are useful in conducting tests. For example, the message app establishes a `ModelState` error with AddModelError to check that a valid PageResult is returned when `OnPostAddMessageAsync` is executed:

C#

```
[Fact]
public async Task
OnPostAddMessageAsync_ReturnsAPageResult_WhenModelStateIsInvalid()
{
    // Arrange
    var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb");
    var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
    var expectedMessages = AppDbContext.GetSeedingMessages();
    mockAppDbContext.Setup(db =>
db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
    var httpContext = new DefaultHttpContext();
    var modelState = new ModelStateDictionary();
    var actionContext = new ActionContext(httpContext, new RouteData(), new
PageActionDescriptor(), modelState);
    var modelMetadataProvider = new EmptyModelMetadataProvider();
    var viewData = new ViewDataDictionary(modelMetadataProvider,
modelState);
    var tempData = new TempDataDictionary(httpContext,
Mock.Of<ITempDataProvider>());
    var pageContext = new PageContext(actionContext)
    {
```

```
            ViewData = viewData
        };
        var pageModel = new IndexModel(mockAppDbContext.Object)
        {
            PageContext = pageContext,
            TempData = tempData,
            Url = new UrlHelper(actionContext)
        };
        pageModel.ModelState.AddModelError("Message.Text", "The Text field is
required.");

        // Act
        var result = await pageModel.OnPostAddMessageAsync();

        // Assert
        Assert.IsType<PageResult>(result);
    }
```

Additional resources

- Unit testing C# in .NET Core using dotnet test and xUnit
- Test controller logic in ASP.NET Core
- Unit Test Your Code (Visual Studio)
- Integration tests in ASP.NET Core
- xUnit.net
- Getting started with xUnit.net: Using .NET Core with the .NET SDK command line
- Moq
- Moq Quickstart

Unit test controller logic in ASP.NET Core

Article • 06/17/2024

By Steve Smith

Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

Unit testing controllers

Set up unit tests of controller actions to focus on the controller's behavior. A controller unit test avoids scenarios such as filters, routing, and model binding. Tests that cover the interactions among components that collectively respond to a request are handled by *integration tests*. For more information on integration tests, see Integration tests in ASP.NET Core.

If you're writing custom filters and routes, unit test them in isolation, not as part of tests on a particular controller action.

To demonstrate controller unit tests, review the following controller in the sample app.

View or download sample code (how to download)

The Home controller displays a list of brainstorming sessions and allows the creation of new brainstorming sessions with a POST request:

C#

```
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new
StormSessionViewModel()
```

```
            {
                Id = session.Id,
                DateCreated = session.DateCreated,
                Name = session.Name,
                IdeaCount = session.Ideas.Count
            });

            return View(model);
        }

        public class NewSessionModel
        {
            [Required]
            public string SessionName { get; set; }
        }

        [HttpPost]
        public async Task<IActionResult> Index(NewSessionModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            else
            {
                await _sessionRepository.AddAsync(new BrainstormSession()
                {
                    DateCreated = DateTimeOffset.Now,
                    Name = model.SessionName
                });
            }

            return RedirectToAction(actionName: nameof(Index));
        }
    }
```

The preceding controller:

- Follows the Explicit Dependencies Principle.
- Expects dependency injection (DI) to provide an instance of `IBrainstormSessionRepository`.
- Can be tested with a mocked `IBrainstormSessionRepository` service using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Introduction to integration tests.

The `HTTP GET Index` method has no looping or branching and only calls one method. The unit test for this action:

- Mocks the `IBrainstormSessionRepository` service using the `GetTestSessions` method. `GetTestSessions` creates two mock brainstorm sessions with dates and session names.
- Executes the `Index` method.
- Makes assertions on the result returned by the method:
  - A ViewResult is returned.
  - The ViewDataDictionary.Model is a `StormSessionViewModel`.
  - There are two brainstorming sessions stored in the `ViewDataDictionary.Model`.

C#

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

C#

```
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

The Home controller's `HTTP POST Index` method tests verifies that:

- When ModelState.IsValid is `false`, the action method returns a *400 Bad Request* ViewResult with the appropriate data.
- When `ModelState.IsValid` is `true`:
  - The `Add` method on the repository is called.
  - A RedirectToActionResult is returned with the correct arguments.

An invalid model state is tested by adding errors using AddModelError as shown in the first test below:

C#

```
[Fact]
public async Task
IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task
IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);
```

```
    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>
(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}
```

When ModelState isn't valid, the same `ViewResult` is returned as for a GET request. The test doesn't attempt to pass in an invalid model. Passing an invalid model isn't a valid approach, since model binding isn't running (although an integration test does use model binding). In this case, model binding isn't tested. These unit tests are only testing the code in the action method.

The second test verifies that when the `ModelState` is valid:

- A new `BrainstormSession` is added (via the repository).
- The method returns a `RedirectToActionResult` with the expected properties.

Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows mock validation in the test. This is performed with the call to `mockRepo.Verify`, which fails the test if the expected method wasn't called.

> **Note**
>
> The Moq library used in this sample makes it possible to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

SessionController in the sample app displays information related to a particular brainstorming session. The controller includes logic to deal with invalid `id` values (there are two `return` scenarios in the following example to cover these scenarios). The final `return` statement returns a new `StormSessionViewModel` to the view (`Controllers/SessionController.cs`):

C#

```
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }
```

```
    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

The unit tests include one test for each `return` scenario in the Session controller `Index` action:

C#

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task
IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
```

```
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var contentResult = Assert.IsType<ContentResult>(result);
    Assert.Equal("Session not found.", contentResult.Content);
}

[Fact]
public async Task IndexReturnsViewResultWithStormSessionViewModel()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSessions().FirstOrDefault(
            s => s.Id == testSessionId));
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsType<StormSessionViewModel>(
        viewResult.ViewData.Model);
    Assert.Equal("Test One", model.Name);
    Assert.Equal(2, model.DateCreated.Day);
    Assert.Equal(testSessionId, model.Id);
}
```

Moving to the Ideas controller, the app exposes functionality as a web API on the `api/ideas` route:

- A list of ideas (`IdeaDTO`) associated with a brainstorming session is returned by the `ForSession` method.
- The `Create` method adds new ideas to a session.

C#

```
[HttpGet("forsession/{sessionId}")]
public async Task<IActionResult> ForSession(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);
    if (session == null)
    {
        return NotFound(sessionId);
```

```
        }

        var result = session.Ideas.Select(idea => new IdeaDTO()
        {
            Id = idea.Id,
            Name = idea.Name,
            Description = idea.Description,
            DateCreated = idea.DateCreated
        }).ToList();

        return Ok(result);
    }

    [HttpPost("create")]
    public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        var session = await _sessionRepository.GetByIdAsync(model.SessionId);
        if (session == null)
        {
            return NotFound(model.SessionId);
        }

        var idea = new Idea()
        {
            DateCreated = DateTimeOffset.Now,
            Description = model.Description,
            Name = model.Name
        };
        session.AddIdea(idea);

        await _sessionRepository.UpdateAsync(session);

        return Ok(session);
    }
```

Avoid returning business domain entities directly via API calls. Domain entities:

- Often include more data than the client requires.
- Unnecessarily couple the app's internal domain model with the publicly exposed API.

Mapping between domain entities and the types returned to the client can be performed:

- Manually with a LINQ `Select`, as the sample app uses. For more information, see LINQ (Language Integrated Query).

- Automatically with a library, such as AutoMapper.

Next, the sample app demonstrates unit tests for the `Create` and `ForSession` API methods of the Ideas controller.

The sample app contains two `ForSession` tests. The first test determines if `ForSession` returns a NotFoundObjectResult (HTTP Not Found) for an invalid session:

C#

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

The second `ForSession` test determines if `ForSession` returns a list of session ideas (`<List<IdeaDTO>>`) for a valid session. The checks also examine the first idea to confirm its `Name` property is correct:

C#

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
```

```
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

To test the behavior of the `Create` method when the `ModelState` is invalid, the sample app adds a model error to the controller as part of the test. Don't try to test model validation or model binding in unit tests—just test the action method's behavior when confronted with an invalid `ModelState`:

C#

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

The second test of `Create` depends on the repository returning `null`, so the mock repository is configured to return `null`. There's no need to create a test database (in memory or otherwise) and construct a query that returns this result. The test can be accomplished in a single statement, as the sample code illustrates:

C#

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
```

```
    Assert.IsType<NotFoundObjectResult>(result);
}
```

The third `Create` test, `Create_ReturnsNewlyCreatedIdeaForSession`, verifies that the repository's `UpdateAsync` method is called. The mock is called with `Verifiable`, and the mocked repository's `Verify` method is called to confirm the verifiable method is executed. It's not the unit test's responsibility to ensure that the `UpdateAsync` method saved the data—that can be performed with an integration test.

C#

```
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription,
returnSession.Ideas.LastOrDefault().Description);
}
```

Test ActionResult<T>

ActionResult<T> (ActionResult<TValue>) can return a type deriving from `ActionResult` or return a specific type.

The sample app includes a method that returns a `List<IdeaDTO>` for a given session `id`. If the session `id` doesn't exist, the controller returns NotFound:

C#

```
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int
sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

Two tests of the `ForSessionActionResult` controller are included in the `ApiIdeasControllerTests`.

The first test confirms that the controller returns an `ActionResult` but not a nonexistent list of ideas for a nonexistent session `id`:

- The `ActionResult` type is `ActionResult<List<IdeaDTO>>`.
- The Result is a NotFoundObjectResult.

C#

```
[Fact]
public async Task
ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
```

```
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await
controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the second test confirms that the method returns:

- An `ActionResult` with a `List<IdeaDTO>` type.
- The ActionResult<T>.Value is a `List<IdeaDTO>` type.
- The first item in the list is a valid idea matching the idea stored in the mock session (obtained by calling `GetTestSession`).

C#

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

The sample app also includes a method to create a new `Idea` for a given session. The controller returns:

- BadRequest for an invalid model.
- NotFound if the session doesn't exist.
- CreatedAtAction when the session is updated with the new idea.

C#

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>>
CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id
}, session);
}
```

Three tests of `CreateActionResult` are included in the `ApiIdeasControllerTests`.

The first text confirms that a BadRequest is returned for an invalid model.

C#

```
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
```

```
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

The second test checks that a NotFound is returned if the session doesn't exist.

C#

```
[Fact]
public async Task
CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the final test confirms that:

- The method returns an `ActionResult` with a `BrainstormSession` type.
- The ActionResult<T>.Result is a CreatedAtActionResult. `CreatedAtActionResult` is analogous to a *201 Created* response with a `Location` header.
- The ActionResult<T>.Value is a `BrainstormSession` type.
- The mock call to update the session, `UpdateAsync(testSession)`, was invoked. The `Verifiable` method call is checked by executing `mockRepo.Verify()` in the assertions.
- Two `Idea` objects are returned for the session.
- The last item (the `Idea` added by the mock call to `UpdateAsync`) matches the `newIdea` added to the session in the test.

C#

```
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>
(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>
(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription,
returnValue.Ideas.LastOrDefault().Description);
}
```

Additional resources

- Integration tests in ASP.NET Core
- Create and run unit tests with Visual Studio
- MyTested.AspNetCore.Mvc - Fluent Testing Library for ASP.NET Core MVC: Strongly-typed unit testing library, providing a fluent interface for testing MVC and web API apps. (*Not maintained or supported by Microsoft.*)

- JustMockLite : A mocking framework for .NET developers. (*Not maintained or supported by Microsoft.*)

Integration tests in ASP.NET Core

Article • 06/17/2024

By Jos van der Til, Martin Costello, and Javier Calvarro Nelson.

Integration tests ensure that an app's components function correctly at a level that includes the app's supporting infrastructure, such as the database, file system, and network. ASP.NET Core supports integration tests using a unit test framework with a test web host and an in-memory test server.

This article assumes a basic understanding of unit tests. If unfamiliar with test concepts, see the Unit Testing in .NET Core and .NET Standard article and its linked content.

View or download sample code (how to download)

The sample app is a Razor Pages app and assumes a basic understanding of Razor Pages. If you're unfamiliar with Razor Pages, see the following articles:

- Introduction to Razor Pages
- Get started with Razor Pages
- Razor Pages unit tests

For testing SPAs, we recommend a tool such as Playwright for .NET, which can automate a browser.

Introduction to integration tests

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

- Database
- File system
- Network appliances
- Request-response pipeline

Unit tests use fabricated components, known as *fakes* or *mock objects*, in place of infrastructure components.

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Therefore, limit the use of integration tests to the most important infrastructure scenarios. If a behavior can be tested using either a unit test or an integration test, choose the unit test.

In discussions of integration tests, the tested project is frequently called the ***System Under Test***, or "**SUT**" for short. "SUT" is used throughout this article to refer to the ASP.NET Core app being tested.

Don't write integration tests for every permutation of data and file access with databases and file systems. Regardless of how many places across an app interact with databases and file systems, a focused set of read, write, update, and delete integration tests are usually capable of adequately testing database and file system components. Use unit tests for routine tests of method logic that interact with these components. In unit tests, the use of infrastructure fakes or mocks result in faster test execution.

ASP.NET Core integration tests

Integration tests in ASP.NET Core require the following:

- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual *Arrange*, *Act*, and *Assert* test steps:

1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The *Arrange* test step is executed: The test app prepares a request.
4. The *Act* test step is executed: The client submits the request and receives the response.
5. The *Assert* test step is executed: The *actual* response is validated as a *pass* or *fail* based on an *expected* response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For example, a different database or different app settings might be used for the tests.

Infrastructure components, such as the test web host and in-memory test server (TestServer), are provided or managed by the Microsoft.AspNetCore.Mvc.Testing package. Use of this package streamlines test creation and execution.

The `Microsoft.AspNetCore.Mvc.Testing` package handles the following tasks:

- Copies the dependencies file (`.deps`) from the SUT into the test project's `bin` directory.
- Sets the content root to the SUT's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory class to streamline bootstrapping the SUT with `TestServer`.

The unit tests documentation describes how to set up a test project and test runner, along with detailed instructions on how to run tests and recommendations for how to name tests and test classes.

Separate unit tests from integration tests into different projects. Separating the tests:

- Helps ensure that infrastructure testing components aren't accidentally included in the unit tests.
- Allows control over which set of tests are run.

There's virtually no difference between the configuration for tests of Razor Pages apps and MVC apps. The only difference is in how the tests are named. In a Razor Pages app, tests of page endpoints are usually named after the page model class (for example, `IndexPageTests` to test component integration for the Index page). In an MVC app, tests are usually organized by controller classes and named after the controllers they test (for example, `HomeControllerTests` to test component integration for the Home controller).

Test app prerequisites

The test project must:

- Reference the Microsoft.AspNetCore.Mvc.Testing package.
- Specify the Web SDK in the project file (`<Project Sdk="Microsoft.NET.Sdk.Web">`).

These prerequisites can be seen in the sample app. Inspect the `tests/RazorPagesProject.Tests/RazorPagesProject.Tests.csproj` file. The sample app

uses the xUnit test framework and the AngleSharp parser library, so the sample app also references:

- AngleSharp
- xunit
- xunit.runner.visualstudio

In apps that use xunit.runner.visualstudio version 2.4.2 or later, the test project must reference the Microsoft.NET.Test.Sdk package.

Entity Framework Core is also used in the tests. See the project file in GitHub.

SUT environment

If the SUT's environment isn't set, the environment defaults to Development.

Basic tests with the default WebApplicationFactory

Expose the implicitly defined `Program` class to the test project by doing one of the following:

- Expose internal types from the web app to the test project. This can be done in the SUT project's file (`.csproj`):

  XML

  ```
  <ItemGroup>
       <InternalsVisibleTo Include="MyTestProject" />
  </ItemGroup>
  ```

- Make the Program class public using a partial class declaration:

  diff

  ```
  var builder = WebApplication.CreateBuilder(args);
  // ... Configure services, routes, etc.
  app.Run();
  + public partial class Program { }
  ```

  The sample app uses the `Program` partial class approach.

WebApplicationFactory<TEntryPoint> is used to create a TestServer for the integration tests. `TEntryPoint` is the entry point class of the SUT, usually `Program.cs`.

Test classes implement a *class fixture* interface (IClassFixture) to indicate the class contains tests and provide shared object instances across the tests in the class.

The following test class, `BasicTests`, uses the `WebApplicationFactory` to bootstrap the SUT and provide an HttpClient to a test method, `Get_EndpointsReturnSuccessAndCorrectContentType`. The method verifies the response status code is successful (200-299) and the `Content-Type` header is `text/html; charset=utf-8` for several app pages.

CreateClient() creates an instance of `HttpClient` that automatically follows redirects and handles cookies.

C#

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<Program>>
{
    private readonly WebApplicationFactory<Program> _factory;

    public BasicTests(WebApplicationFactory<Program> factory)
    {
        _factory = factory;
    }

    [Theory]
    [InlineData("/")]
    [InlineData("/Index")]
    [InlineData("/About")]
    [InlineData("/Privacy")]
    [InlineData("/Contact")]
    public async Task Get_EndpointsReturnSuccessAndCorrectContentType(string
url)
    {
        // Arrange
        var client = _factory.CreateClient();

        // Act
        var response = await client.GetAsync(url);

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

By default, non-essential cookies aren't preserved across requests when the General Data Protection Regulation consent policy is enabled. To preserve non-essential cookies, such as those used by the TempData provider, mark them as essential in your tests. For instructions on marking a cookie as essential, see Essential cookies.

AngleSharp vs `Application Parts` for antiforgery checks

This article uses the AngleSharp parser to handle the antiforgery checks by loading pages and parsing the HTML. For testing the endpoints of controller and Razor Pages views at a lower-level, without caring about how they render in the browser, consider using `Application Parts`. The Application Parts approach injects a controller or Razor Page into the app that can be used to make JSON requests to get the required values. For more information, see the blog Integration Testing ASP.NET Core Resources Protected with Antiforgery Using Application Parts and associated GitHub repo by Martin Costello.

Customize WebApplicationFactory

Web host configuration can be created independently of the test classes by inheriting from WebApplicationFactory<TEntryPoint> to create one or more custom factories:

1. Inherit from `WebApplicationFactory` and override ConfigureWebHost. The IWebHostBuilder allows the configuration of the service collection with IWebHostBuilder.ConfigureServices

```
C#

public class CustomWebApplicationFactory<TProgram>
    : WebApplicationFactory<TProgram> where TProgram : class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            var dbContextDescriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbContextOptions<ApplicationDbContext>));

            services.Remove(dbContextDescriptor);

            var dbConnectionDescriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbConnection));
```

```
            services.Remove(dbConnectionDescriptor);

            // Create open SqliteConnection so EF won't automatically
close it.
            services.AddSingleton<DbConnection>(container =>
            {
                var connection = new
SqliteConnection("DataSource=:memory:");
                connection.Open();

                return connection;
            });

            services.AddDbContext<ApplicationDbContext>((container,
options) =>
            {
                var connection =
container.GetRequiredService<DbConnection>();
                options.UseSqlite(connection);
            });
        });

        builder.UseEnvironment("Development");
    }
}
```

Database seeding in the sample app is performed by the `InitializeDbForTests` method. The method is described in the Integration tests sample: Test app organization section.

The SUT's database context is registered in `Program.cs`. The test app's `builder.ConfigureServices` callback is executed *after* the app's `Program.cs` code is executed. To use a different database for the tests than the app's database, the app's database context must be replaced in `builder.ConfigureServices`.

The sample app finds the service descriptor for the database context and uses the descriptor to remove the service registration. The factory then adds a new `ApplicationDbContext` that uses an in-memory database for the tests..

To connect to a different database, change the `DbConnection`. To use a SQL Server test database:

- Reference the Microsoft.EntityFrameworkCore.SqlServer NuGet package in the project file.
- Call `UseInMemoryDatabase`.

2. Use the custom `CustomWebApplicationFactory` in test classes. The following example uses the factory in the `IndexPageTests` class:

```
C#

public class IndexPageTests :
    IClassFixture<CustomWebApplicationFactory<Program>>
{
    private readonly HttpClient _client;
    private readonly CustomWebApplicationFactory<Program>
        _factory;

    public IndexPageTests(
        CustomWebApplicationFactory<Program> factory)
    {
        _factory = factory;
        _client = factory.CreateClient(new
WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });
    }
```

The sample app's client is configured to prevent the `HttpClient` from following redirects. As explained later in the Mock authentication section, this permits tests to check the result of the app's first response. The first response is a redirect in many of these tests with a `Location` header.

3. A typical test uses the `HttpClient` and helper methods to process the request and the response:

```
C#

[Fact]
public async Task Post_DeleteAllMessagesHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var defaultPage = await _client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await _client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),

(IHtmlButtonElement)content.QuerySelector("button[id='deleteAllBtn']"))
;

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
```

```
        Assert.Equal("/", response.Headers.Location.OriginalString);
    }
```

Any POST request to the SUT must satisfy the antiforgery check that's automatically made by the app's data protection antiforgery system. In order to arrange for a test's POST request, the test app must:

1. Make a request for the page.
2. Parse the antiforgery cookie and request validation token from the response.
3. Make the POST request with the antiforgery cookie and request validation token in place.

The `SendAsync` helper extension methods (`Helpers/HttpClientExtensions.cs`) and the `GetDocumentAsync` helper method (`Helpers/HtmlHelpers.cs`) in the sample app use the AngleSharp parser to handle the antiforgery check with the following methods:

- `GetDocumentAsync`: Receives the HttpResponseMessage and returns an `IHtmlDocument`. `GetDocumentAsync` uses a factory that prepares a *virtual response* based on the original `HttpResponseMessage`. For more information, see the AngleSharp documentation.
- `SendAsync` extension methods for the `HttpClient` compose an HttpRequestMessage and call SendAsync(HttpRequestMessage) to submit requests to the SUT. Overloads for `SendAsync` accept the HTML form (`IHtmlFormElement`) and the following:
  - Submit button of the form (`IHtmlElement`)
  - Form values collection (`IEnumerable<KeyValuePair<string, string>>`)
  - Submit button (`IHtmlElement`) and form values (`IEnumerable<KeyValuePair<string, string>>`)

AngleSharp is a **third-party parsing library used for demonstration purposes** in this article and the sample app. AngleSharp isn't supported or required for integration testing of ASP.NET Core apps. Other parsers can be used, such as the Html Agility Pack (HAP). Another approach is to write code to handle the antiforgery system's request verification token and antiforgery cookie directly. See AngleSharp vs Application Parts for antiforgery checks in this article for more information.

The EF-Core in-memory database provider can be used for limited and basic testing, however the ***SQLite provider* is the recommended choice for in-memory testing**.

See Extend Startup with startup filters which shows how to configure middleware using IStartupFilter, which is useful when a test requires a custom service or middleware.

Customize the client with WithWebHostBuilder

When additional configuration is required within a test method, WithWebHostBuilder creates a new `WebApplicationFactory` with an IWebHostBuilder that is further customized by configuration.

The sample code calls `WithWebHostBuilder` to replace configured services with test stubs. For more information and example usage, see Inject mock services in this article.

The `Post_DeleteMessageHandler_ReturnsRedirectToRoot` test method of the sample app demonstrates the use of `WithWebHostBuilder`. This test performs a record delete in the database by triggering a form submission in the SUT.

Because another test in the `IndexPageTests` class performs an operation that deletes all of the records in the database and may run before the `Post_DeleteMessageHandler_ReturnsRedirectToRoot` method, the database is reseeded in this test method to ensure that a record is present for the SUT to delete. Selecting the first delete button of the `messages` form in the SUT is simulated in the request to the SUT:

C#

```
[Fact]
public async Task Post_DeleteMessageHandler_ReturnsRedirectToRoot()
{
    // Arrange
    using (var scope = _factory.Services.CreateScope())
    {
        var scopedServices = scope.ServiceProvider;
        var db = scopedServices.GetRequiredService<ApplicationDbContext>();

        Utilities.ReinitializeDbForTests(db);
    }

    var defaultPage = await _client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await _client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("form[id='messages']")
            .QuerySelector("div[class='panel-body']")
            .QuerySelector("button"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
```

```
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

Client options

See the WebApplicationFactoryClientOptions page for defaults and available options when creating `HttpClient` instances.

Create the `WebApplicationFactoryClientOptions` class and pass it to the CreateClient() method:

C#

```
public class IndexPageTests :
    IClassFixture<CustomWebApplicationFactory<Program>>
{
    private readonly HttpClient _client;
    private readonly CustomWebApplicationFactory<Program>
        _factory;

    public IndexPageTests(
        CustomWebApplicationFactory<Program> factory)
    {
        _factory = factory;
        _client = factory.CreateClient(new
WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });
    }
```

NOTE: To avoid HTTPS redirection warnings in logs when using HTTPS Redirection Middleware, set `BaseAddress = new Uri("https://localhost")`

Inject mock services

Services can be overridden in a test with a call to ConfigureTestServices on the host builder. To scope the overridden services to the test itself, the WithWebHostBuilder method is used to retrieve a host builder. This can be seen in the following tests:

- Get_QuoteService_ProvidesQuoteInPage
- Get_GithubProfilePageCanGetAGithubUser
- Get_SecurePageIsReturnedForAnAuthenticatedUser

The sample SUT includes a scoped service that returns a quote. The quote is embedded in a hidden field on the Index page when the Index page is requested.

`Services/IQuoteService.cs`:

C#

```
public interface IQuoteService
{
    Task<string> GenerateQuote();
}
```

`Services/QuoteService.cs`:

C#

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Dr. Who: Planet of Evil
// https://www.bbc.co.uk/programmes/p00pyrx6
public class QuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Come on, Sarah. We've an appointment in London, " +
            "and we're already 30,000 years late.");
    }
}
```

`Program.cs`:

C#

```
services.AddScoped<IQuoteService, QuoteService>();
```

`Pages/Index.cshtml.cs`:

C#

```
public class IndexModel : PageModel
{
    private readonly ApplicationDbContext _db;
    private readonly IQuoteService _quoteService;

    public IndexModel(ApplicationDbContext db, IQuoteService quoteService)
    {
        _db = db;
        _quoteService = quoteService;
    }
```

```
    [BindProperty]
    public Message Message { get; set; }

    public IList<Message> Messages { get; private set; }

    [TempData]
    public string MessageAnalysisResult { get; set; }

    public string Quote { get; private set; }

    public async Task OnGetAsync()
    {
        Messages = await _db.GetMessagesAsync();

        Quote = await _quoteService.GenerateQuote();
    }
```

`Pages/Index.cs`:

CSHTML

```
<input id="quote" type="hidden" value="@Model.Quote">
```

The following markup is generated when the SUT app is run:

HTML

```
<input id="quote" type="hidden" value="Come on, Sarah. We&#x27;ve an
appointment in
    London, and we&#x27;re already 30,000 years late.">
```

To test the service and quote injection in an integration test, a mock service is injected into the SUT by the test. The mock service replaces the app's `QuoteService` with a service provided by the test app, called `TestQuoteService`:

`IntegrationTests.IndexPageTests.cs`:

C#

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Pyramids of Mars
// https://www.bbc.co.uk/programmes/p00pys55
public class TestQuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult(
            "Something's interfering with time, Mr. Scarman, " +
            "and time is my business.");
```

```
    }
}
```

`ConfigureTestServices` is called, and the scoped service is registered:

C#

```
[Fact]
public async Task Get_QuoteService_ProvidesQuoteInPage()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddScoped<IQuoteService, TestQuoteService>();
            });
        })
        .CreateClient();

    //Act
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);
    var quoteElement = content.QuerySelector("#quote");

    // Assert
    Assert.Equal("Something's interfering with time, Mr. Scarman, " +
        "and time is my business.", quoteElement.Attributes["value"].Value);
}
```

The markup produced during the test's execution reflects the quote text supplied by `TestQuoteService`, thus the assertion passes:

HTML

```
<input id="quote" type="hidden" value="Something&#x27;s interfering with
time,
    Mr. Scarman, and time is my business.">
```

Mock authentication

Tests in the `AuthTests` class check that a secure endpoint:

- Redirects an unauthenticated user to the app's sign in page.
- Returns content for an authenticated user.

In the SUT, the `/SecurePage` page uses an AuthorizePage convention to apply an AuthorizeFilter to the page. For more information, see Razor Pages authorization conventions.

C#

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/SecurePage");
});
```

In the `Get_SecurePageRedirectsAnUnauthenticatedUser` test, a WebApplicationFactoryClientOptions is set to disallow redirects by setting AllowAutoRedirect to `false`:

C#

```
[Fact]
public async Task Get_SecurePageRedirectsAnUnauthenticatedUser()
{
    // Arrange
    var client = _factory.CreateClient(
        new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });

    // Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.StartsWith("http://localhost/Identity/Account/Login",
        response.Headers.Location.OriginalString);
}
```

By disallowing the client to follow the redirect, the following checks can be made:

- The status code returned by the SUT can be checked against the expected HttpStatusCode.Redirect result, not the final status code after the redirect to the sign in page, which would be HttpStatusCode.OK.
- The `Location` header value in the response headers is checked to confirm that it starts with `http://localhost/Identity/Account/Login`, not the final sign in page response, where the `Location` header wouldn't be present.

The test app can mock an AuthenticationHandler<TOptions> in ConfigureTestServices in order to test aspects of authentication and authorization. A minimal scenario returns an

AuthenticateResult.Success:

```
C#

public class TestAuthHandler :
AuthenticationHandler<AuthenticationSchemeOptions>
{
    public TestAuthHandler(IOptionsMonitor<AuthenticationSchemeOptions>
options,
        ILoggerFactory logger, UrlEncoder encoder)
        : base(options, logger, encoder)
    {
    }

    protected override Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        var claims = new[] { new Claim(ClaimTypes.Name, "Test user") };
        var identity = new ClaimsIdentity(claims, "Test");
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, "TestScheme");

        var result = AuthenticateResult.Success(ticket);

        return Task.FromResult(result);
    }
}
```

The `TestAuthHandler` is called to authenticate a user when the authentication scheme is set to `TestScheme` where `AddAuthentication` is registered for `ConfigureTestServices`. It's important for the `TestScheme` scheme to match the scheme your app expects. Otherwise, authentication won't work.

```
C#

[Fact]
public async Task Get_SecurePageIsReturnedForAnAuthenticatedUser()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddAuthentication(defaultScheme: "TestScheme")
                    .AddScheme<AuthenticationSchemeOptions, TestAuthHandler>
(
                        "TestScheme", options => { });
            });
        })
        .CreateClient(new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false,
```

```
        });

    client.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue(scheme: "TestScheme");

    //Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

For more information on `WebApplicationFactoryClientOptions`, see the Client options section.

Basic tests for authentication middleware

See this GitHub repository for basic tests of authentication middleware. It contains a test server that's specific to the test scenario.

Set the environment

Set the environment in the custom application factory:

C#

```
public class CustomWebApplicationFactory<TProgram>
    : WebApplicationFactory<TProgram> where TProgram : class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            var dbContextDescriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbContextOptions<ApplicationDbContext>));

            services.Remove(dbContextDescriptor);

            var dbConnectionDescriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbConnection));

            services.Remove(dbConnectionDescriptor);

            // Create open SqliteConnection so EF won't automatically close
it.
            services.AddSingleton<DbConnection>(container =>
            {
```

```
                var connection = new
SqliteConnection("DataSource=:memory:");
                connection.Open();

                return connection;
            });

            services.AddDbContext<ApplicationDbContext>((container, options)
=>
            {
                var connection = container.GetRequiredService<DbConnection>
();
                options.UseSqlite(connection);
            });
        });

        builder.UseEnvironment("Development");
    }
}
```

How the test infrastructure infers the app content root path

The `WebApplicationFactory` constructor infers the app content root path by searching for a WebApplicationFactoryContentRootAttribute on the assembly containing the integration tests with a key equal to the `TEntryPoint` assembly `System.Reflection.Assembly.FullName`. In case an attribute with the correct key isn't found, `WebApplicationFactory` falls back to searching for a solution file (*.sln*) and appends the `TEntryPoint` assembly name to the solution directory. The app root directory (the content root path) is used to discover views and content files.

Disable shadow copying

Shadow copying causes the tests to execute in a different directory than the output directory. If your tests rely on loading files relative to `Assembly.Location` and you encounter issues, you might have to disable shadow copying.

To disable shadow copying when using xUnit, create a `xunit.runner.json` file in your test project directory, with the correct configuration setting :

JSON

```
{
  "shadowCopy": false
```

```
}
```

Disposal of objects

After the tests of the `IClassFixture` implementation are executed, TestServer and HttpClient are disposed when xUnit disposes of the WebApplicationFactory. If objects instantiated by the developer require disposal, dispose of them in the `IClassFixture` implementation. For more information, see Implementing a Dispose method.

Integration tests sample

The sample app is composed of two apps:

Expand table

App	Project directory	Description
Message app (the SUT)	`src/RazorPagesProject`	Allows a user to add, delete one, delete all, and analyze messages.
Test app	`tests/RazorPagesProject.Tests`	Used to integration test the SUT.

The tests can be run using the built-in test features of an IDE, such as Visual Studio. If using Visual Studio Code or the command line, execute the following command at a command prompt in the `tests/RazorPagesProject.Tests` directory:

Console

```
dotnet test
```

Message app (SUT) organization

The SUT is a Razor Pages message system with the following characteristics:

- The Index page of the app (`Pages/Index.cshtml` and `Pages/Index.cshtml.cs`) provides a UI and page model methods to control the addition, deletion, and analysis of messages (average words per message).
- A message is described by the `Message` class (`Data/Message.cs`) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using Entity Framework's in-memory database†.

- The app contains a data access layer (DAL) in its database context class, `AppDbContext` (`Data/AppDbContext.cs`).
- If the database is empty on app startup, the message store is initialized with three messages.
- The app includes a `/SecurePage` that can only be accessed by an authenticated user.

†The EF article, Test with InMemory, explains how to use an in-memory database for tests with MSTest. This topic uses the xUnit test framework. Test concepts and test implementations across different test frameworks are similar but not identical.

Although the app doesn't use the repository pattern and isn't an effective example of the Unit of Work (UoW) pattern, Razor Pages supports these patterns of development. For more information, see Designing the infrastructure persistence layer and Test controller logic (the sample implements the repository pattern).

Test app organization

The test app is a console app inside the `tests/RazorPagesProject.Tests` directory.

Expand table

Test app directory	Description
`AuthTests`	Contains test methods for: • Accessing a secure page by an unauthenticated user. • Accessing a secure page by an authenticated user with a mock AuthenticationHandler<TOptions>. • Obtaining a GitHub user profile and checking the profile's user login.
`BasicTests`	Contains a test method for routing and content type.
`IntegrationTests`	Contains the integration tests for the Index page using custom `WebApplicationFactory` class.
`Helpers/Utilities`	• `Utilities.cs` contains the `InitializeDbForTests` method used to seed the database with test data. • `HtmlHelpers.cs` provides a method to return an AngleSharp `IHtmlDocument` for use by the test methods. • `HttpClientExtensions.cs` provide overloads for `SendAsync` to submit requests to the SUT.

The test framework is xUnit. Integration tests are conducted using the Microsoft.AspNetCore.TestHost, which includes the TestServer. Because the Microsoft.AspNetCore.Mvc.Testing package is used to configure the test host and test server, the `TestHost` and `TestServer` packages don't require direct package references in the test app's project file or developer configuration in the test app.

Integration tests usually require a small dataset in the database prior to the test execution. For example, a delete test calls for a database record deletion, so the database must have at least one record for the delete request to succeed.

The sample app seeds the database with three messages in `Utilities.cs` that tests can use when they execute:

C#

```
public static void InitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.AddRange(GetSeedingMessages());
    db.SaveChanges();
}

public static void ReinitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.RemoveRange(db.Messages);
    InitializeDbForTests(db);
}

public static List<Message> GetSeedingMessages()
{
    return new List<Message>()
    {
        new Message(){ Text = "TEST RECORD: You're standing on my scarf." },
        new Message(){ Text = "TEST RECORD: Would you like a jelly baby?" },
        new Message(){ Text = "TEST RECORD: To the rational mind, " +
            "nothing is inexplicable; only unexplained." }
    };
}
```

The SUT's database context is registered in `Program.cs`. The test app's `builder.ConfigureServices` callback is executed *after* the app's `Program.cs` code is executed. To use a different database for the tests, the app's database context must be replaced in `builder.ConfigureServices`. For more information, see the Customize WebApplicationFactory section.

Additional resources

- Unit tests
- Razor Pages unit tests in ASP.NET Core
- ASP.NET Core Middleware
- Test controller logic in ASP.NET Core
- Basic tests for authentication middleware

ASP.NET Core load/stress testing

Article • 03/07/2024

Load testing and stress testing are important to ensure a web app is performant and scalable. Load and stress testing have different goals even though they often share similar tests.

Load tests: Test whether the app can handle a specified load of users for a certain scenario while still satisfying the response goal. The app is run under normal conditions.

Stress tests: Test app stability when running under extreme conditions, often for a long period of time. The tests place high user load, either spikes or gradually increasing load on the app, or they limit the app's computing resources.

Stress tests determine if an app under stress can recover from failure and gracefully return to expected behavior. Under stress, the app is run at abnormally high stress.

Azure Load Testing is a fully managed load-testing service that enables you to generate high-scale load. The service simulates traffic for apps, regardless of where they're hosted. Azure Load Testing Preview enables you to use existing Apache JMeter scripts to generate high-scale load.

Visual Studio 2019 load testing has been deprecated. The corresponding Azure DevOps cloud-based load testing service has been closed.

Third-party tools

The following list contains third-party web performance tools with various feature sets:

- Apache JMeter
- ApacheBench (ab)
- Gatling
- jmeter-dotnet-dsl
- k6
- Locust
- West Wind WebSurge
- Netling
- Vegeta
- NBomber

Load and stress test with release builds

Load and stress tests should be done in release and production mode and not in debug and development mode. Release configurations are fully optimized with minimal logging. Debug configuration is not optimized. Development mode enables more information logging that can impact performance.

Test ASP.NET Core middleware

Article • 01/11/2024

By Chris Ross

Middleware can be tested in isolation with TestServer. It allows you to:

- Instantiate an app pipeline containing only the components that you need to test.
- Send custom requests to verify middleware behavior.

Advantages:

- Requests are sent in-memory rather than being serialized over the network.
- This avoids additional concerns, such as port management and HTTPS certificates.
- Exceptions in the middleware can flow directly back to the calling test.
- It's possible to customize server data structures, such as HttpContext, directly in the test.

Set up the TestServer

In the test project, create a test:

- Build and start a host that uses TestServer.
- Add any required services that the middleware uses.
- Add a package reference to the project for the Microsoft.AspNetCore.TestHost NuGet package.
- Configure the processing pipeline to use the middleware for the test.

```
C#

[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
```

```
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    ...
}
```

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Send requests with HttpClient

Send a request using HttpClient:

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();
```

```
    var response = await host.GetTestClient().GetAsync("/");

    ...
}
```

Assert the result. First, make an assertion the opposite of the result that you expect. An initial run with a false positive assertion confirms that the test fails when the middleware is performing correctly. Run the test and confirm that the test fails.

In the following example, the middleware should return a 404 status code (*Not Found*) when the root endpoint is requested. Make the first test run with `Assert.NotEqual( ... );`, which should fail:

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.NotEqual(HttpStatusCode.NotFound, response.StatusCode);
}
```

Change the assertion to test the middleware under normal operating conditions. The final test uses `Assert.Equal( ... );`. Run the test again to confirm that it passes.

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
```

```
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}
```

Send requests with HttpContext

A test app can also send a request using SendAsync(Action<HttpContext>, CancellationToken). In the following example, several checks are made when `https://example.com/A/Path/?and=query` is processed by the middleware:

C#

```
[Fact]
public async Task TestMiddleware_ExpectedResponse()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var server = host.GetTestServer();
    server.BaseAddress = new Uri("https://example.com/A/Path/");

    var context = await server.SendAsync(c =>
```

```
    {
        c.Request.Method = HttpMethods.Post;
        c.Request.Path = "/and/file.txt";
        c.Request.QueryString = new QueryString("?and=query");
    });

    Assert.True(context.RequestAborted.CanBeCanceled);
    Assert.Equal(HttpProtocol.Http11, context.Request.Protocol);
    Assert.Equal("POST", context.Request.Method);
    Assert.Equal("https", context.Request.Scheme);
    Assert.Equal("example.com", context.Request.Host.Value);
    Assert.Equal("/A/Path", context.Request.PathBase.Value);
    Assert.Equal("/and/file.txt", context.Request.Path.Value);
    Assert.Equal("?and=query", context.Request.QueryString.Value);
    Assert.NotNull(context.Request.Body);
    Assert.NotNull(context.Request.Headers);
    Assert.NotNull(context.Response.Headers);
    Assert.NotNull(context.Response.Body);
    Assert.Equal(404, context.Response.StatusCode);
    Assert.Null(context.Features.Get<IHttpResponseFeature>().ReasonPhrase);
}
```

SendAsync permits direct configuration of an HttpContext object rather than using the HttpClient abstractions. Use SendAsync to manipulate structures only available on the server, such as HttpContext.Items or HttpContext.Features.

As with the earlier example that tested for a *404 - Not Found* response, check the opposite for each `Assert` statement in the preceding test. The check confirms that the test fails correctly when the middleware is operating normally. After you've confirmed that the false positive test works, set the final `Assert` statements for the expected conditions and values of the test. Run it again to confirm that the test passes.

Add request routes

Additional routes can be added by configuration using the test `HttpClient`:

C#

```
    [Fact]
    public async Task TestWithEndpoint_ExpectedResponse ()
    {
        using var host = await new HostBuilder()
            .ConfigureWebHost(webBuilder =>
            {
                webBuilder
                    .UseTestServer()
                    .ConfigureServices(services =>
                    {
                        services.AddRouting();
```

```
                    })
                    .Configure(app =>
                    {
                        app.UseRouting();
                        app.UseMiddleware<MyMiddleware>();
                        app.UseEndpoints(endpoints =>
                        {
                            endpoints.MapGet("/hello", () =>
                                TypedResults.Text("Hello Tests"));
                        });
                    });
            })
            .StartAsync();

        var client = host.GetTestClient();

        var response = await client.GetAsync("/hello");

        Assert.True(response.IsSuccessStatusCode);
        var responseBody = await response.Content.ReadAsStringAsync();
        Assert.Equal("Hello Tests", responseBody);
```

Additional routes can also be added using the approach `server.SendAsync`.

TestServer limitations

TestServer:

- Was created to replicate server behaviors to test middleware.
- Does ***not*** try to replicate all HttpClient behaviors.
- Attempts to give the client access to as much control over the server as possible, and with as much visibility into what's happening on the server as possible. For example it may throw exceptions not normally thrown by `HttpClient` in order to directly communicate server state.
- Doesn't set some transport specific headers by default as those aren't usually relevant to middleware. For more information, see the next section.
- Ignores the `Stream` position passed through StreamContent. HttpClient sends the entire stream from the start position, even when positioning is set. For more information, see this GitHub issue.

Content-Length and Transfer-Encoding headers

TestServer does ***not*** set transport related request or response headers such as Content-Length or Transfer-Encoding. Applications should avoid depending on these headers because their usage varies by client, scenario, and protocol. If `Content-Length`

and `Transfer-Encoding` are necessary to test a specific scenario, they can be specified in the test when composing the HttpRequestMessage or HttpContext. For more information, see the following GitHub issues:

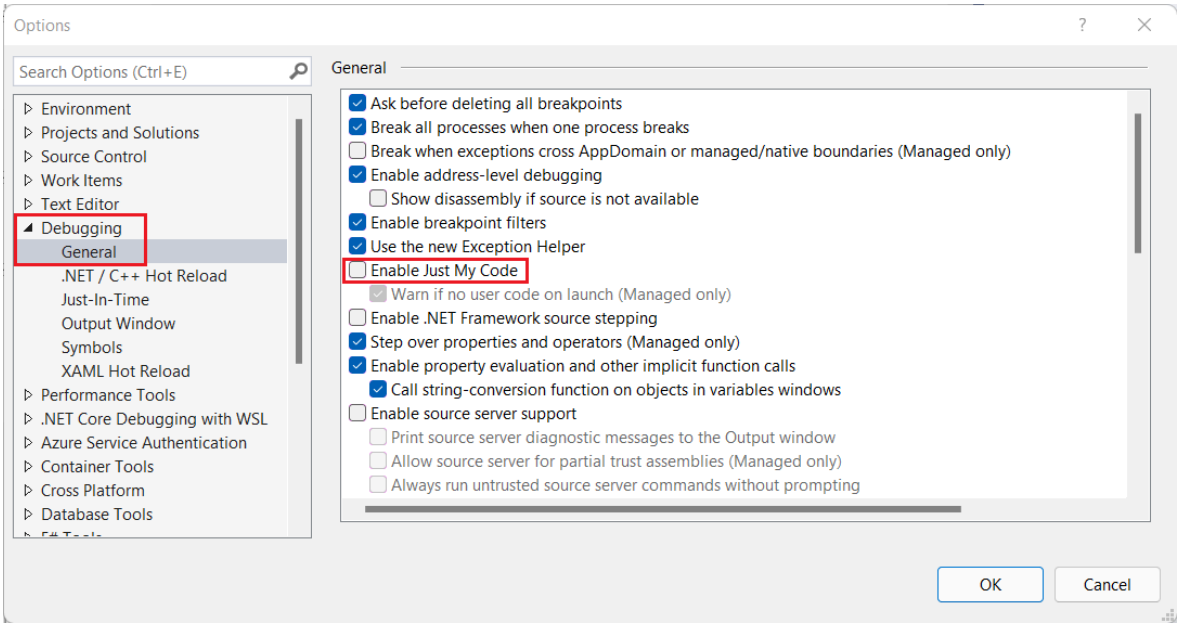
- dotnet/aspnetcore#21677
- dotnet/aspnetcore#18463
- dotnet/aspnetcore#13273

Debug .NET and ASP.NET Core source code with Visual Studio

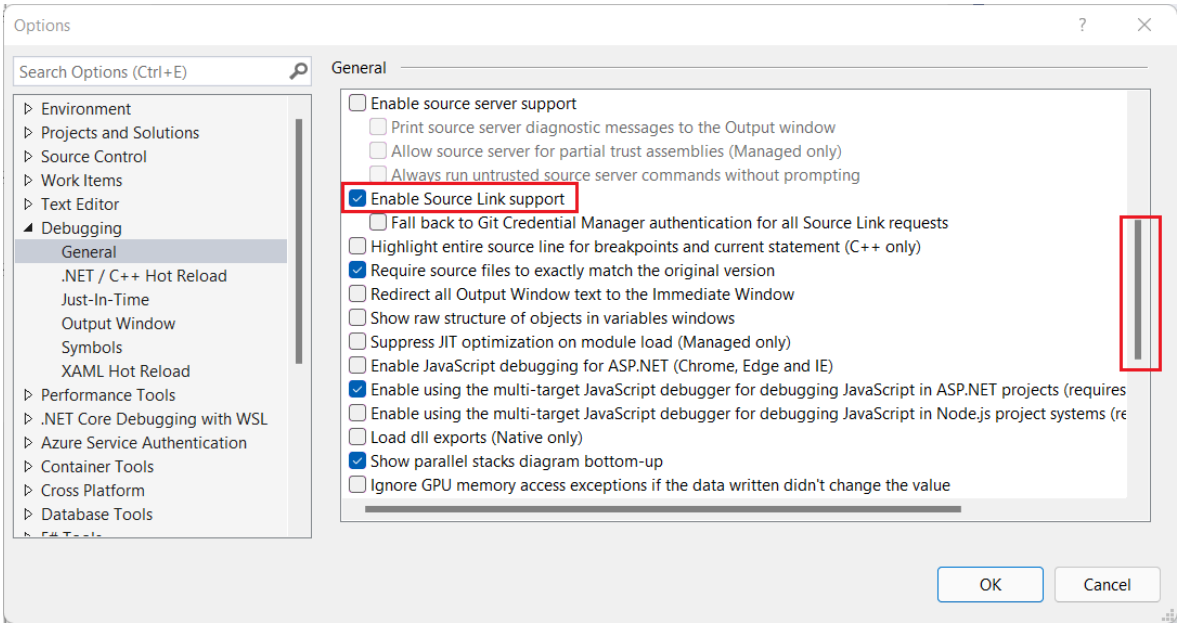
Article • 09/18/2024

To debug .NET and ASP.NET Core source code in Visual Studio:

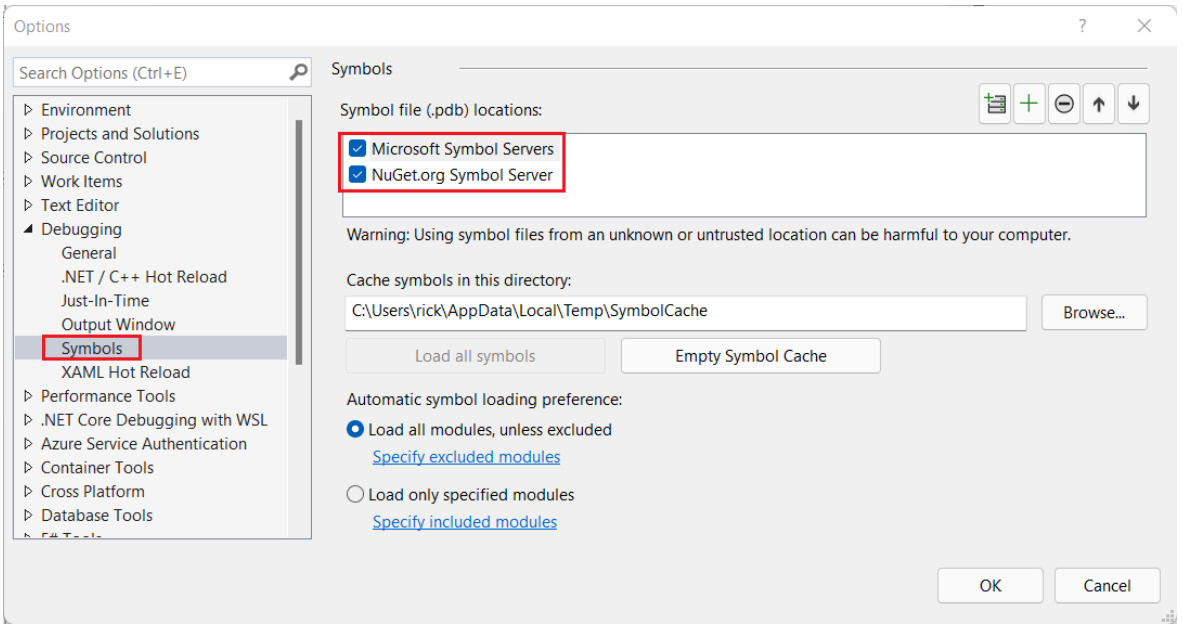
- In **Tools -> Options -> Debugging -> General**, un-check **Enable Just My Code**.



- Verify **Enable Source Link support** is checked.

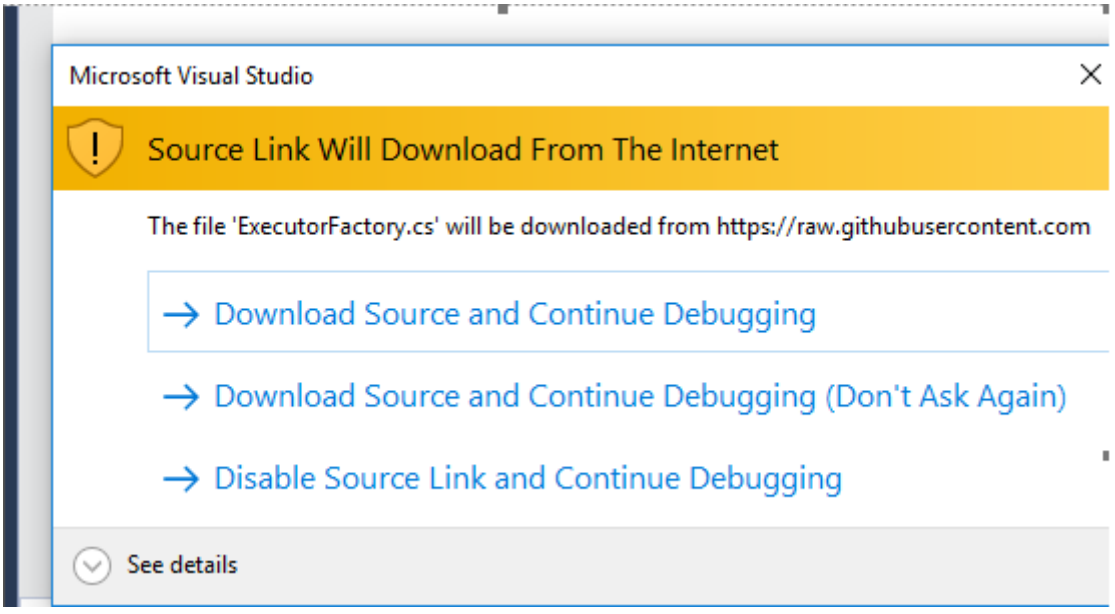


- In **Tool -> Options -> Debugging -> Symbols**, enable **Microsoft Symbol Servers**.



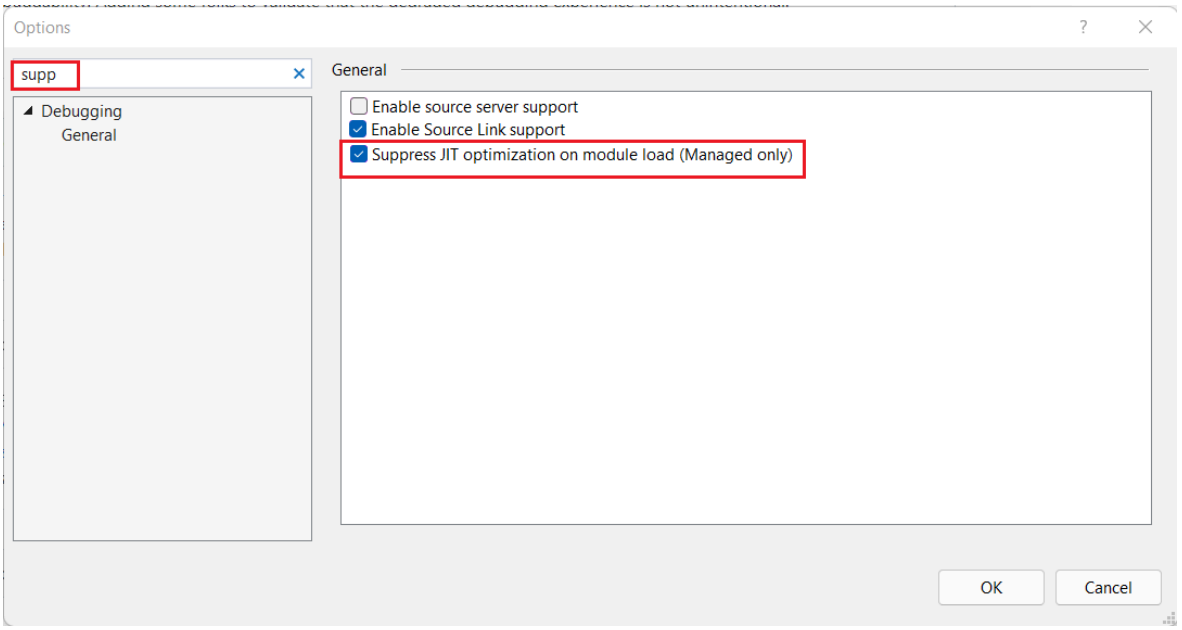
When you step into any .NET or ASP.NET Core code, Visual Studio displays the source code. For example:

- Set a break point in `OnGet` in `Pages/Privacy.cshtml.cs` and select the **Privacy** link.
- Select one of the **Download Source and Continue Debugging** options.



The preceding instructions work for basic stepping into functions, but the optimized .NET code often removes local variables and functions. To disable optimizations and allow better source debugging:

- In **Tools -> Options -> Debugging -> General**, enable **Suppress JIT optimization on module load (Managed only)**:



- Add the environment variable and value `COMPlus_ReadyToRun=0` to the `Properties/launchSettings.json` file:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:10892",
      "sslPort": 44315
    }
  },
  "profiles": {
    "WebApplication18": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7045;http://localhost:5045",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "COMPlus_ReadyToRun": "0"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "COMPlus_ReadyToRun": "0"
      }
    }
  }
}
```

If you have debugged an app before with the previous version of .NET, delete the `%TEMP%/SymbolCache` directory as it can have old PDBs that are out of date.

Debugging .NET Core on Unix over SSH

- Debugging .NET Core on Unix over SSH
- Debugging ASP Core on Linux with Visual Studio 2017

Additional resources

- JIT Optimization and Debugging
- Limitations of the 'Suppress JIT optimization' option To set `COMPlus_ReadyToRun` to `0`
- .NET Hot Reload support for ASP.NET Core
- Test Execution with Hot Reload
- Debug ASP.NET Core Blazor apps

Remote Debug ASP.NET Core on IIS using an Azure VM from Visual Studio

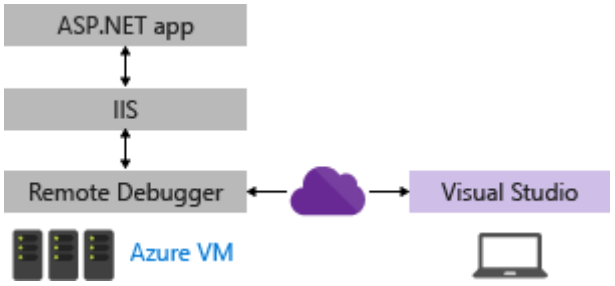
Article • 04/23/2024

This guide explains how to set up and configure a Visual Studio ASP.NET Core app, deploy it to IIS using an Azure VM, and attach the remote debugger from Visual Studio.

For IIS scenarios, Linux isn't supported.

To debug IIS on an Azure VM, follow the steps in this article. Using this method, you can use a customized configuration of IIS, but the setup and deployment steps are more complicated. If you don't need to customize IIS for your scenario, you might choose simpler methods to host and debug the app in Azure App Service instead.

For an Azure VM, you must deploy your app from Visual Studio to Azure and you also need to manually install the IIS role and the remote debugger, as shown in the following illustration.



> ⚠ **Warning**
>
> Be sure to delete the Azure resources that you create when you have completed the steps in this tutorial. That way you can avoid incurring unnecessary charges.

These procedures have been tested on these server configurations:

- Windows Server 2022 and IIS 10
- Windows Server 2019 and IIS 10
- Windows Server 2016 and IIS 10

Prerequisites

Visual Studio 2019 or later versions is required to follow the steps shown in this article.

Network requirements

Debugging between two computers connected through a proxy isn't supported. Debugging over a high latency or low bandwidth connection, such as dialup Internet, or over the Internet across countries/regions isn't recommended and might fail or be unacceptably slow. For a complete list of requirements, see Requirements.

App already running in IIS on the Azure VM?

This article includes steps on setting up a basic configuration of IIS on Windows server and deploying the app from Visual Studio. These steps are included to make sure that the server has the required components installed, that the app can run correctly, and that you're ready to remote debug.

- If your app is running in IIS and you just want to download the remote debugger and start debugging, go to Download and Install the remote tools on Windows Server.
- If you want help ensuring your app is set up, deployed, and running correctly in IIS so that you can debug, follow all the steps in this article.
    - Before you begin, follow all the steps described in Create a Windows Virtual Machine, which includes steps to install the IIS web server.
    - Make sure you open port 80 in the Azure Network security group. When you verify that port 80 is open, also open the correct port for the remote debugger (4026, 4024, or 4022). That way, you don't have to open it later. If you're using Web Deploy, also open port 8172.

Create the ASP.NET Core application on the Visual Studio computer

1. Create a new ASP.NET Core web application.

   In Visual Studio, choose **File** > **Start window** to open the Start window, and then choose **Create a new project**. In the search box, type **web app**, then choose **C#** as the language, then choose **ASP.NET Core Web Application (Model-View-Controller)**, and then choose **Next**. On the next screen, name the project **MyASPApp**, and then choose **Next**.

   Choose either the recommended target framework or .NET 8, and then choose **Create**. The version must match the version installed on the server.

2. Open the *HomeController.cs* file in the Controllers folder and set a breakpoint in the `return View;` statement in the `Privacy` method.

   In older templates, open the *Privacy.cshtml.cs* file and set a breakpoint in the `OnGet` method.

Update browser security settings on Windows Server

If you're using an older version of Windows Server, you might need to add some domains as trusted sites to enable you to download some of the web server components. Add the trusted sites by going to **Internet Options** > **Security** > **Trusted Sites** > **Sites**. Add the following domains.

- microsoft.com
- go.microsoft.com
- download.microsoft.com
- iis.net

When you download the software, you might get requests to grant permission to load various web site scripts and resources. Some of these resources aren't required, but to simplify the process, select **Add** when prompted.

Install ASP.NET Core on Windows Server

1. Install the .NET Core Hosting Bundle on the hosting system. The bundle installs the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module. For more in-depth instructions, see Publishing to IIS.

   For the current .NET Core hosting bundle, install the ASP.NET Core Hosting Bundle.

   > **Note**
   >
   > If you previously installed IIS, the ASP.NET Core IIS Module gets installed with ASP.NET Core. Otherwise, install the ASP.NET Core IIS Module manually.

   For .NET Core 2, install the .NET Core Windows Server Hosting.

   > **Note**

If the system doesn't have an Internet connection, obtain and install the *Microsoft Visual C++ 2015 Redistributable* before installing the .NET Core Windows Server Hosting bundle.

2. Restart the system (or execute **net stop was /y** followed by **net start w3svc** from a command prompt to pick up a change to the system PATH).

Choose a deployment option

If you need help deploying the app to IIS, consider these options:

- Deploy by creating a publish settings file in IIS and importing the settings in Visual Studio. In some scenarios, this approach is a fast way to deploy your app. When you create the publish settings file, permissions are automatically set up in IIS.
- Deploy by publishing to a local folder and copying the output by a preferred method to a prepared app folder on IIS.

(Optional) Deploy using a publish settings file

You can use this option create a publish settings file and import it into Visual Studio.

> **Note**
>
> This deployment method uses Web Deploy, which must be installed on the server. If you want to configure Web Deploy manually instead of importing the settings, you can install Web Deploy 3.6 instead of Web Deploy 3.6 for Hosting Servers. However, if you configure Web Deploy manually, you will need to make sure that an app folder on the server is configured with the correct values and permissions (see Configure ASP.NET Web site).

Configure the ASP.NET Core web site

1. In IIS Manager, in the left pane under **Connections**, select **Application Pools**. Open **DefaultAppPool** and set the **.NET CLR version** to **No Managed Code**. This is required for ASP.NET Core. The Default Web Site uses the DefaultAppPool.

2. Stop and restart the DefaultAppPool.

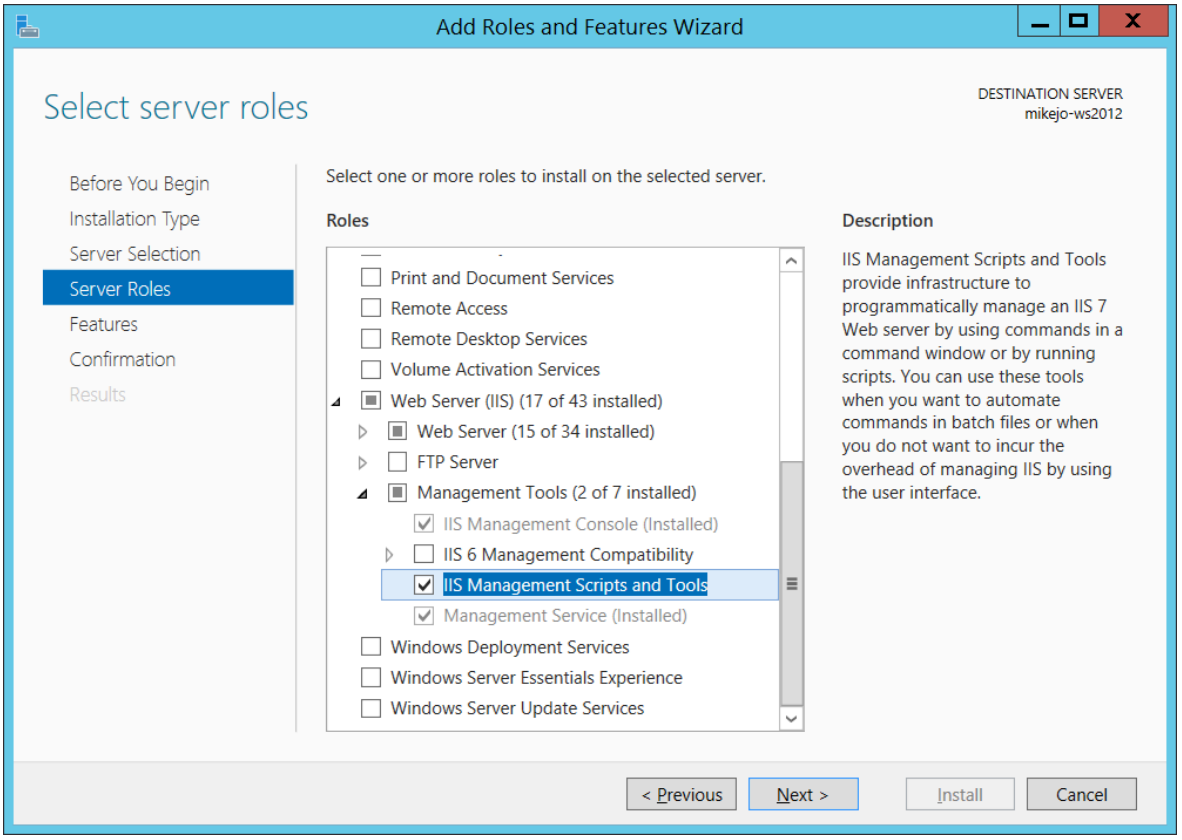
Install and configure Web Deploy on Windows Server

Web Deploy provides additional configuration features that enable the creation of the publish settings file from the UI.

> **Note**
>
> The Web Platform Installer reached End-of-Life on 7/1/22. For more information, see Web Platform Installer - End of support and sunsetting the product/application feed. You can directly install Web Deploy 4.0 to create the publish settings file.

1. If you did not already install **IIS Management Scripts and Tools**, install it now.

   Go to **Select server roles** > **Web Server (IIS)** > **Management Tools**, and then select the **IIS Management Scripts and Tools** role, click **Next**, and then install the role.



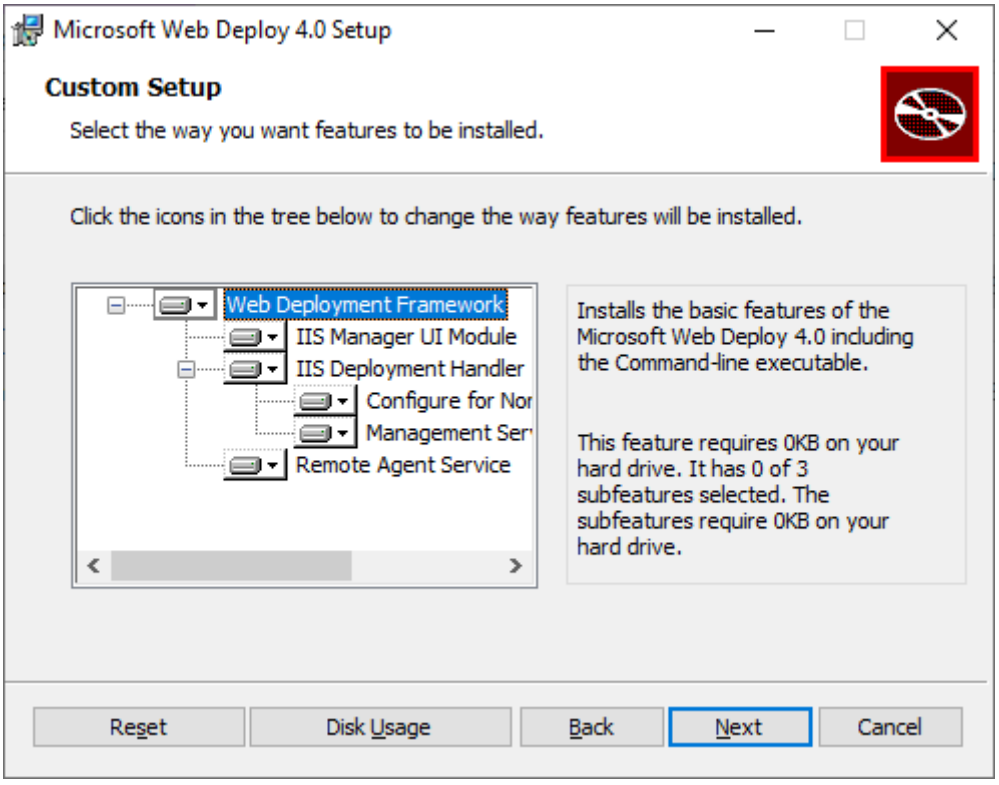
   The scripts and tools are required to enable the generation of the publish settings file.

   Make sure you also install the **Management Service** and **IIS Management Console** (they may be already installed).

2. On Windows Server, download Web Deploy 4.0.

3. Run the Web Deploy installation program, and make sure you select **Complete** installation instead of a typical installation.

With a complete installation, you get the components you need to generate a publish settings file. (If you choose **Custom** instead, you can see the list of components, as shown in the following illustration.)



4. (Optional) Verify that Web Deploy is running correctly by opening **Control Panel** > **System and Security** > **Administrative Tools** > **Services**, and then make sure that:

   - **Web Deployment Agent Service** is running (the service name is different in older versions).
   - **Web Management Service** is running.

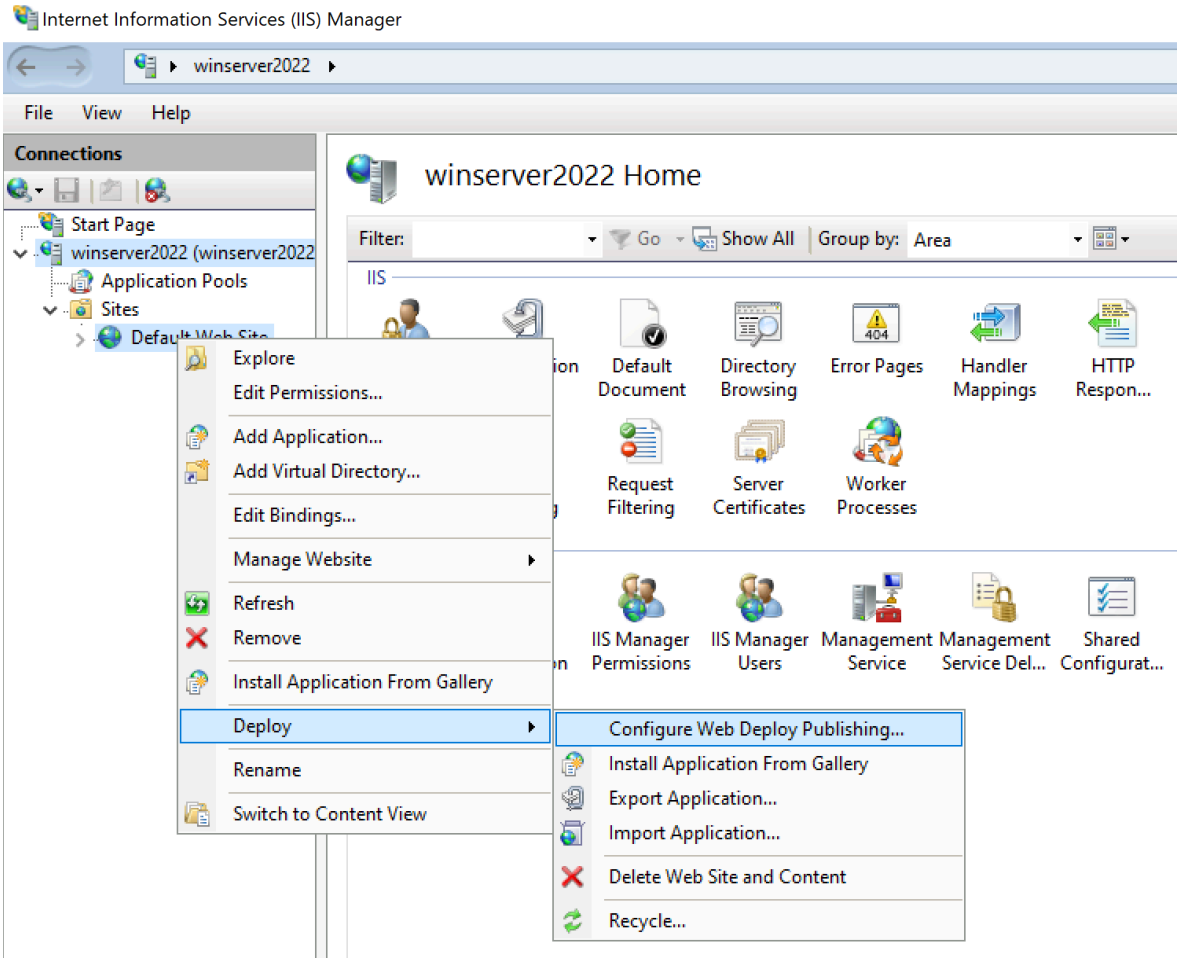
   If one of the agent services is not running, restart the **Web Deployment Agent Service**.

   If the Web Deployment Agent Service is not present at all, go to **Control Panel** > **Programs** > **Uninstall a program**, find **Microsoft Web Deploy <version>**. Choose to **Change** the installation and make sure that you choose **Will be installed to the local hard drive** for the Web Deploy components. Complete the change installation steps.

Create the publish settings file in IIS on Windows Server

1. Close and reopen the IIS Management Console to show updated configuration options in the UI.

2. In IIS, right-click the **Default Web Site**, choose **Deploy** > **Configure Web Deploy Publishing**.



If you don't see the **Deploy** menu, see the preceding section to verify that Web Deploy is running.

3. In the **Configure Web Deploy Publishing** dialog box, examine the settings.

4. Click **Setup**.

   In the **Results** panel, the output shows that access rights are granted to the specified user, and that a file with a *.publishsettings* file extension has been generated in the location shown in the dialog box.

```xml
<?xml version="1.0" encoding="utf-8"?>
<publishData>
  <publishProfile
    publishUrl="https://myhostname:8172/msdeploy.axd"
    msdeploySite="Default Web Site"
    destinationAppUrl="http://myhostname:80/"
    profileName="Default Settings"
    publishMethod="MSDeploy"
```

```
    userName="myhostname\myusername" />
</publishData>
```

Depending on your Windows Server and IIS configuration, you see different values in the XML file. Here are a few details about the values that you see:

- The *msdeploy.axd* file referenced in the `publishUrl` attribute is a dynamically generated HTTP handler file for Web Deploy. (For testing purposes, `http://myhostname:8172` generally works as well.)
- The `publishUrl` port is set to port 8172, which is the default for Web Deploy.
- The `destinationAppUrl` port is set to port 80, which is the default for IIS.
- If, in later steps, you are unable to connect to the remote host from Visual Studio using the host name, test the server's IP address in place of the host name.

> ⓘ **Note**
>
> If you are publishing to IIS running on an Azure VM, you must open an inbound port for Web Deploy and IIS in the Network Security group. For detailed information, see Open ports to a virtual machine.

5. Copy this file to the computer where you are running Visual Studio.

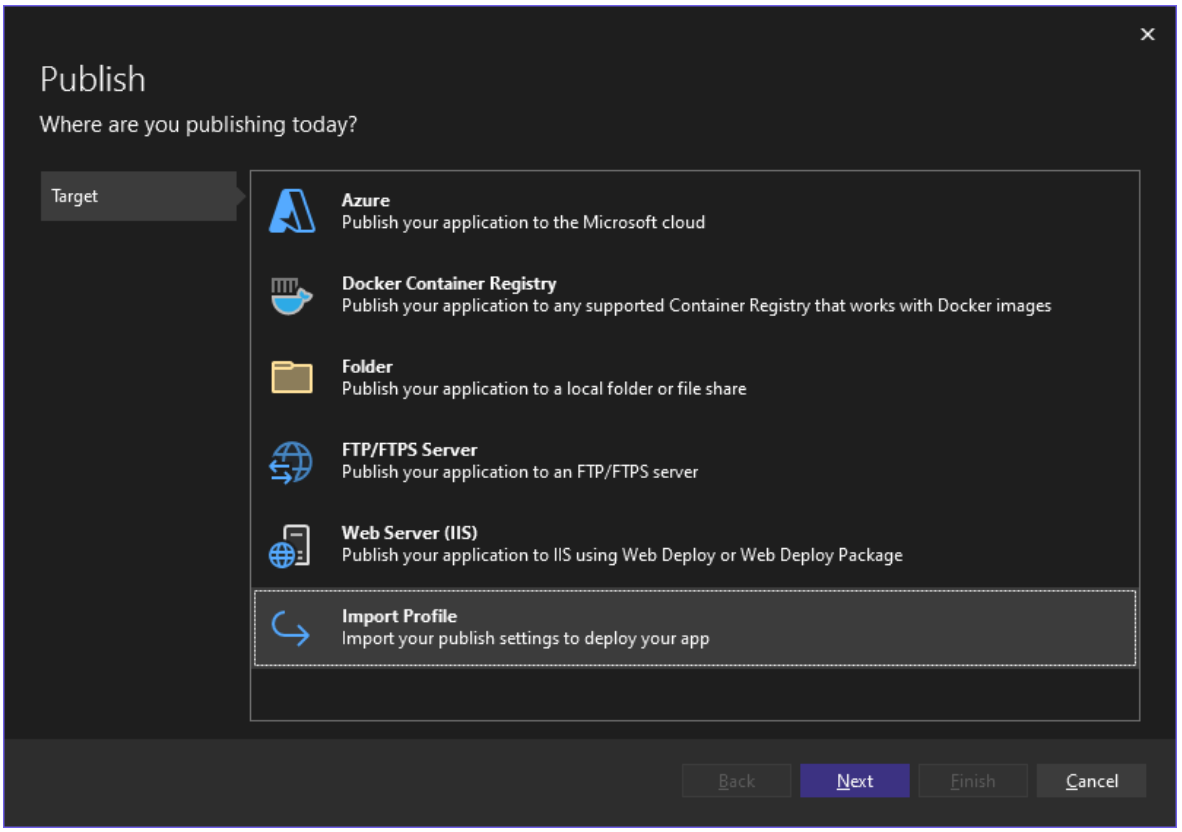
Import the publish settings in Visual Studio and deploy

1. On the computer where you have the ASP.NET project open in Visual Studio, right-click the project in Solution Explorer, and choose **Publish**.

   If you have previously configured any publishing profiles, the **Publish** pane appears. Click **New** or **Create new profile**.

2. Select the option to import a profile.

   In the **Publish** dialog box, click **Import Profile**.



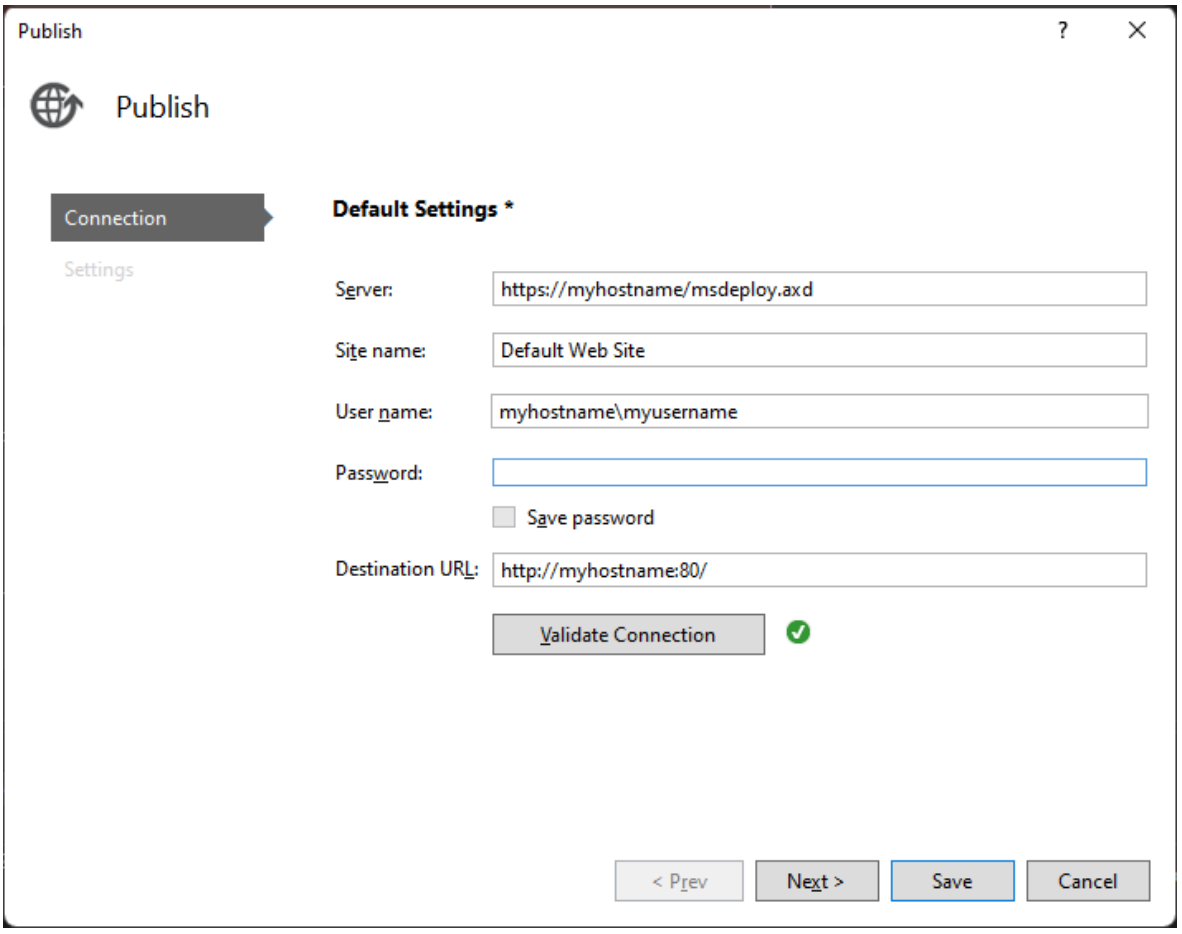
3. Navigate to the location of the publish settings file that you created in the previous section.

4. In the **Import Publish Settings File** dialog, navigate to and select the profile that you created in the previous section, and click **Open**.

   Click **Finish** to save the publishing profile, and then click **Publish**.

   Visual Studio begins the deployment process, and the Output window shows progress and results.

   If you get an any deployment errors, click **More Actions** > **Edit** to edit settings. Modify settings and click **Validate** to test new settings. If the host name is not found, try the IP address instead of the host name in both the **Server** and **Destination URL** fields.



> **Note**
>
> If you restart an Azure VM, the IP address might change.

After the app deploys successfully, it should start automatically.

- If the app doesn't start after deployment, start the app in IIS to verify that it runs correctly.
- For ASP.NET Core, make sure the Application pool field for the **DefaultAppPool** is set to **No Managed Code**.

When you're ready, switch to a debug configuration.

> **Important**
>
> If you choose to debug a Release configuration, you disable debugging in the *web.config* file when you publish.

1. Select **More Options** > **Edit** to edit the profile, and then select **Settings**.
2. Select **Save** and then republish the app.
3. Select a **Debug** configuration, and then select **Remove additional files at destination** under the **File Publish** options.

> ⚠ **Warning**
>
> Using username and password credentials (basic authentication) is not the most secure method of authentication. Whenever possible, use alternative methods. For example, consider publishing to a package from Visual Studio, and then use *WebDeploy.exe* from a command line to deploy the package. With that method, you can use IIS Manager to configure authorized Windows users who can publish to the web server, and run *WebDeploy.exe* under that Windows user account. See Installing and Configuring Web Deploy on IIS 8.0 or Later. If you do use password credentials, be sure to use a strong password, and secure the password from being leaked or shared.

(Optional) Deploy by publishing to a local folder

You can use this option to deploy your app if you want to copy the app to IIS using PowerShell, RoboCopy, or you want to manually copy the files.

Configure the ASP.NET Core Web site on the Windows Server computer

If you're importing publish settings, you can skip this section.

1. Open the **Internet Information Services (IIS) Manager** and go to **Sites**.
2. Right-click the **Default Web Site** node and select **Add Application**.
3. Set the **Alias** field to **MyASPApp** and the Application pool field to **No Managed Code**. Set the **Physical path** to **C:\Publish** (where you later deploy the ASP.NET Core project).
4. With the site selected in the IIS Manager, choose **Edit Permissions**, and make sure that IUSR, IIS_IUSRS, or the user configured for the Application Pool is an authorized user with Read & Execute rights.

   If you don't see one of these users with access, go through steps to add IUSR as a user with Read & Execute rights.

> ⓘ **Important**

For security information related to the built-in accounts, see [Understanding Built-In User and Group Accounts in IIS 7](#).

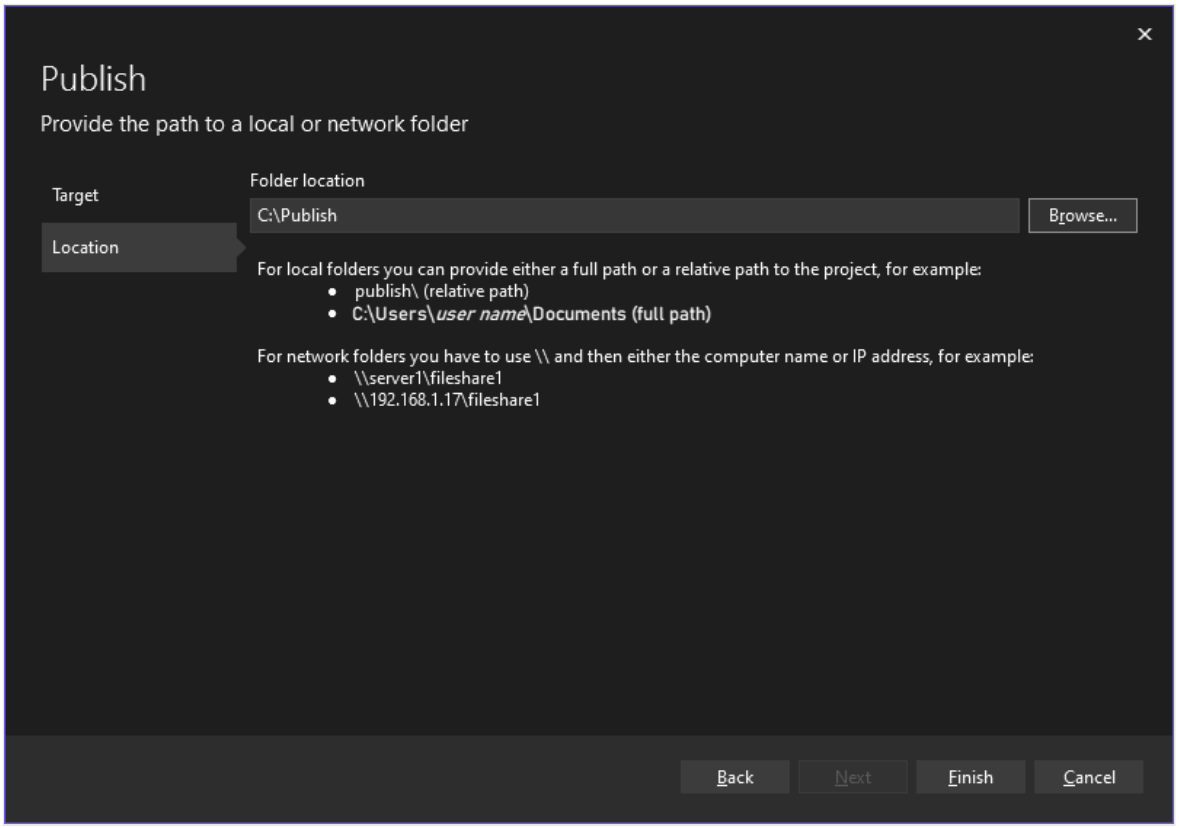
(Optional) Publish and Deploy the app by publishing to a local folder from Visual Studio

If you're not using Web Deploy, you must publish and deploy the app using the file system or other tools. You can start by creating a package using the file system, and then either deploy the package manually or use other tools like PowerShell, Robocopy, or XCopy. In this section, we assume you're manually copying the package if you aren't using Web Deploy.

1. In the **Solution Explorer**, right-click the project node and select **Publish** (for Web Forms, **Publish Web App**).

   If you have previously configured any publishing profiles, the **Publish** pane appears. Click **New profile**.

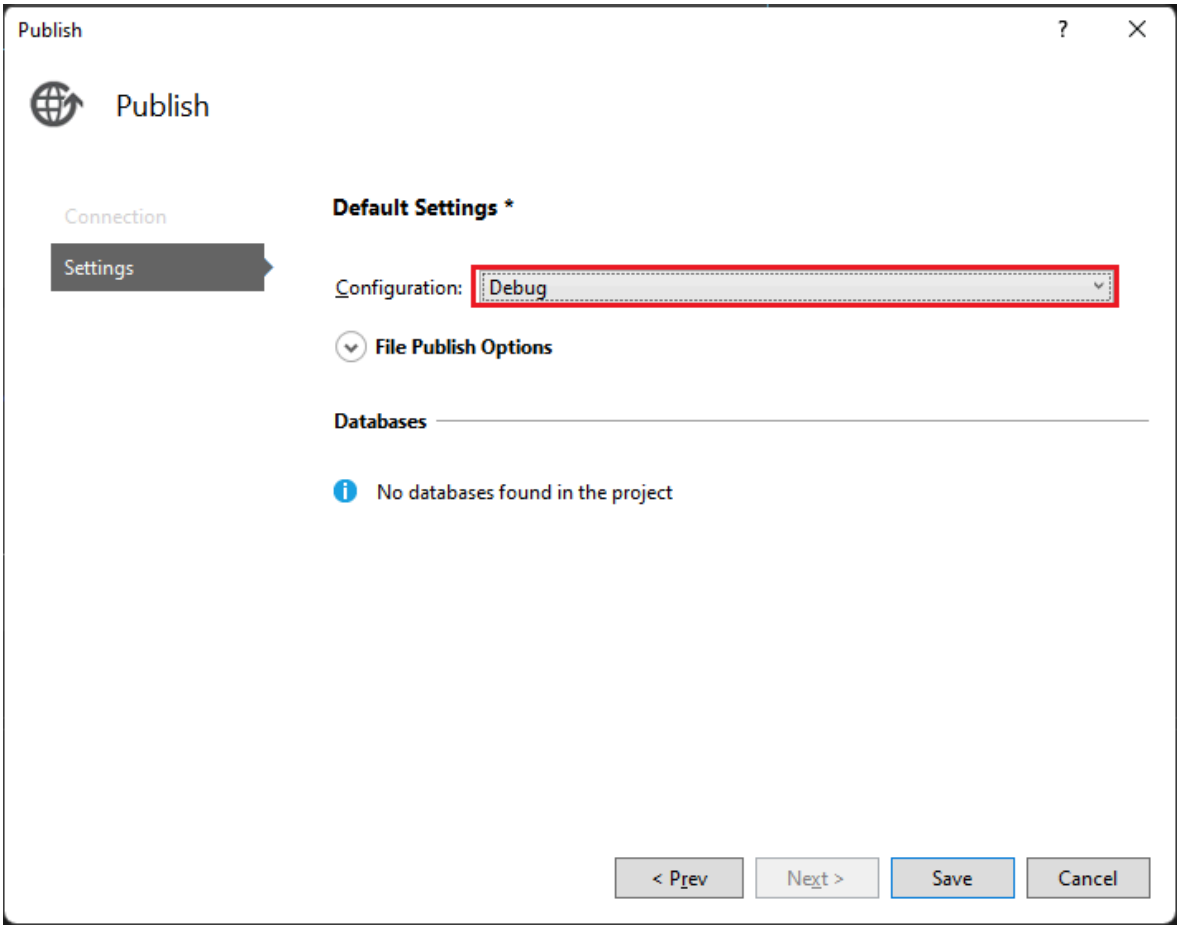
2. In the **Publish** dialog box, select **Folder**, click **Browse**, and create a new folder, **C:\Publish**.

   Click **Finish** to save the publish profile.

3. Switch to a debug configuration.

Choose **Edit** to edit the profile, and then choose **Settings**. Choose a **Debug** configuration, and then choose **Remove additional files at destination** under the **File Publish** options.



> ⓘ **Note**
>
> If you use a Release build, you disable debugging in the *web.config* file when you publish.

4. Click **Publish**.

   The application publishes a **Debug** configuration of the project to the local folder. Progress shows in the Output window.

5. Copy the ASP.NET project directory from the Visual Studio computer to the local directory configured for the ASP.NET app (in this example, **C:\Publish**) on the Windows Server computer. In this tutorial, we assume you are copying manually, but you can use other tools like PowerShell, Xcopy, or Robocopy.

> ⊗ **Caution**

If you need to make changes to the code or rebuild, you must republish and repeat this step. The executable you copied to the remote machine must exactly match your local source and symbols. If you do not do this you will receive a `cannot find or open the PDB file` warning in Visual Studio when you attempt to debug the process.

6. On the Windows Server, verify that you can run the app correctly by opening the app in your browser.

   If the app doesn't run correctly, there may be a mismatch between the version of ASP.NET installed on your server and your Visual Studio machine, or you may have an issue with your IIS or Web site configuration. Recheck earlier steps.

Download and Install the remote tools on Windows Server

Download the version of the remote tools that matches your version of Visual Studio.

On the remote device or server that you want to debug on, rather than the Visual Studio machine, download and install the correct version of the remote tools from the links in the following table.

- Download the most recent update of the remote tools for your version of Visual Studio. Earlier remote tools versions aren't compatible with later Visual Studio versions. (For example, if you're using Visual Studio 2019, download the latest update of the remote tools for Visual Studio 2019. In this scenario, don't download the remote tools for Visual Studio 2022.)
- Download the remote tools with the same architecture as the machine you're installing them on. For example, if you want to debug x86 applications on a remote computer running an x64 operating system, install the x64 remote tools. To debug x86, ARM, or x64 applications on an ARM64 operating system, install the ARM64 remote tools.

Expand table

Version	Link	Notes
Visual Studio 2022	Remote tools	Compatible with all Visual Studio 2022 versions. Download the version matching your device operating system (x86, x64 (AMD64), or ARM64). On older versions of Windows Server, see Unblock the file download for help with downloading the remote tools.

Version	Link	Notes
Visual Studio 2019	Remote tools	Remote tools for Visual Studio 2019 are available from My.VisualStudio.com. If prompted, join the free Visual Studio Dev Essentials program, or sign in with your Visual Studio subscription ID. Download the version matching your device operating system (x86, x64 (AMD64), or ARM64). On older versions of Windows Server, see Unblock the file download for help with downloading the remote tools.
Visual Studio 2017	Remote tools	Remote tools for Visual Studio 2017 are available from My.VisualStudio.com. If prompted, join the free Visual Studio Dev Essentials program, or sign in with your Visual Studio subscription ID. Download the version matching your device operating system (x86, x64 (AMD64), or ARM64). On Windows Server, see Unblock the file download for help with downloading the remote tools.
Visual Studio 2015	Remote tools	Remote tools for Visual Studio 2015 are available from My.VisualStudio.com. If prompted, join the free Visual Studio Dev Essentials program, or sign in with your Visual Studio subscription ID. On Windows Server, see Unblock the file download for help with downloading the remote tools.
Visual Studio 2013	Remote tools	Download page in Visual Studio 2013 documentation
Visual Studio 2012	Remote tools	Download page in Visual Studio 2012 documentation

You can run the remote debugger by copying *msvsmon.exe* to the remote computer, rather than installing the remote tools. However, the Remote Debugger Configuration Wizard (*rdbgwiz.exe*) is available only when you install the remote tools. You may need to use the wizard for configuration if you want to run the remote debugger as a service. For more information, see (Optional) Configure the remote debugger as a service.

> **Note**
>
> - To debug Windows 10 or later apps on ARM devices, use ARM64, which is available with the latest version of the remote tools.
> - To debug Windows 10 apps on Windows RT devices, use ARM, which is available only in the Visual Studio 2015 remote tools download.
> - To debug x64 apps on an ARM64 operating system, run the x64 msvsmon.exe that is installed with the ARM64 remote tools.

Set up the remote debugger on Windows Server

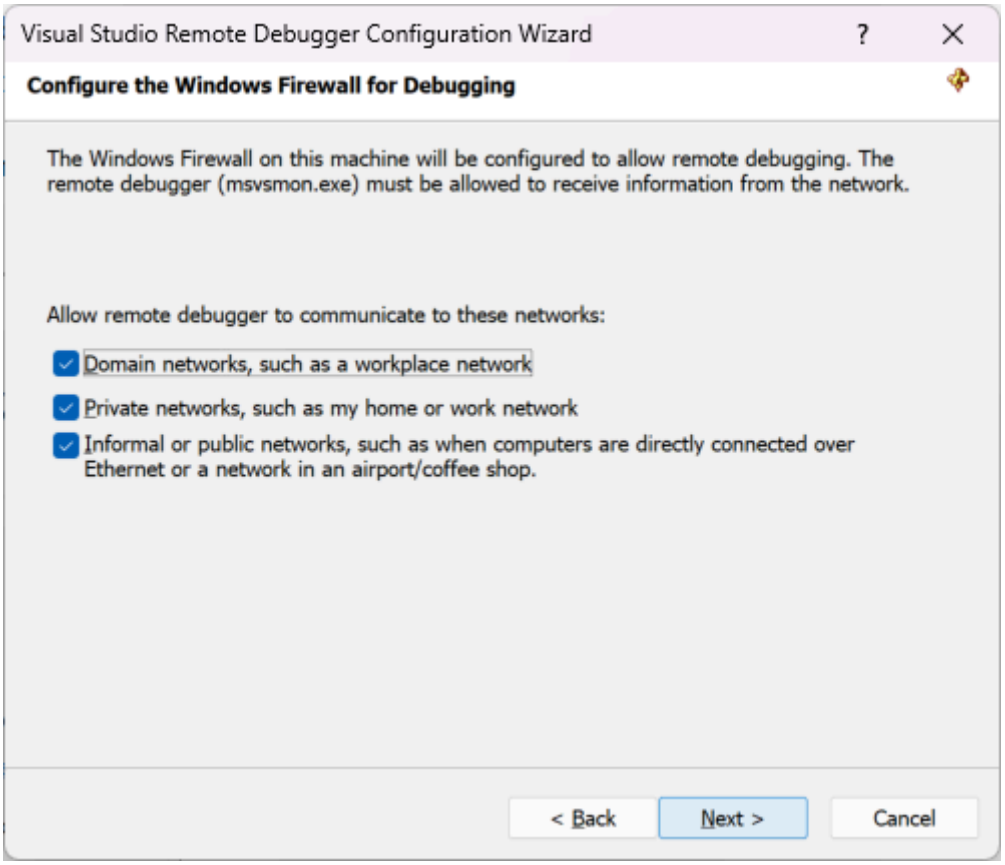
1. On the remote computer, find and start the **Remote Debugger** from the **Start** menu.

   If you don't have administrative permissions on the remote computer, right-click the **Remote Debugger** app and select **Run as administrator**. Otherwise, just start it normally.

   If you are planning to attach to a process which is running as an administrator, or is running under a different user account (such as IIS), right-click the **Remote Debugger** app and select **Run as administrator**. For more information, see Run the remote debugger as an administrator.

2. The first time you start the remote debugger (or before you have configured it), the **Remote Debugging Configuration** wizard appears.

   In most scenarios, choose **Next** until you get to the **Configure the Windows Firewall** page of the wizard.



3. Select at least one network type you want to use the remote tools on. If the computers are connected through a domain, you must choose the first item. If the computers are connected through a workgroup or homegroup, choose the second or third item as appropriate.

Next, select **Finish** to start the remote debugger.

4. When configuration is complete, the **Remote Debugger** window appears.

   The remote debugger is now waiting for a connection. Use the server name and port number shown to set the remote connection configuration in Visual Studio.

To stop the remote debugger, select **File** > **Exit**. You can restart it from the **Start** menu, or from the command line:

Windows Command Prompt

```
<Remote debugger installation directory>\msvsmon.exe
```

> ⓘ **Note**
>
> If you need to add permissions for additional users, change the authentication mode, or port number for the remote debugger, see Configure the remote debugger.

Attach to the ASP.NET Core application from the Visual Studio computer

Starting in Visual Studio 2022 version 17.10 Preview 2, the Attach to Process dialog box has changed. If you need instructions that match the older dialog box, switch to the Visual Studio 2019 view (upper left version selector in the article).

1. On the Visual Studio computer, open the solution that you're trying to debug (**MyASPApp** if you're following all the steps in this article).

2. In Visual Studio, select **Debug** > **Attach to Process** (Ctrl + Alt + P).

   > 💡 **Tip**

> In Visual Studio 2017 and later versions, you can reattach to the same process you previously attached to by using **Debug** > **Reattach to Process...** (Shift + Alt + P).

3. Set the **Connection Type** to **Remote (Windows)**.

   The **Connection Target** option appears.

   Set the **Connection Target** to **<remote computer name>** and press **Enter**.

   Verify that Visual Studio adds the required port to the computer name, which appears in the format: **<remote computer name>:port**

   On Visual Studio 2022, you should see **<remote computer name>:4026**

   The port is required. If you don't see the port number, add it manually.

4. Select **Refresh**.

   You should see some processes appear in the **Available Processes** window.

   If you don't see any processes, try using the IP address instead of the remote computer name (the port is required). You can use `ipconfig` in a command line to get the IPv4 address.

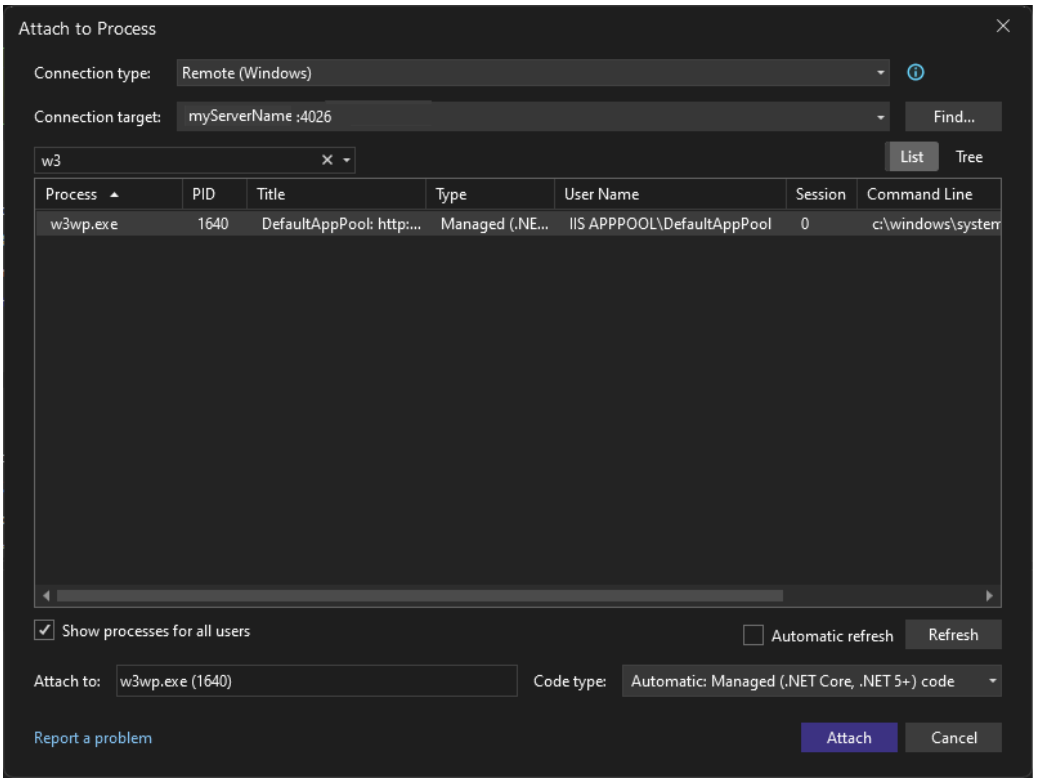
   If you want to use the **Find** button, you might need to open outbound UDP port 3702 on the server.

5. Check **Show processes from all users**.

6. Type the first letter of your process name to quickly find your app.

   - If you're using the in-process hosting model on IIS, select the correct **w3wp.exe** process. Starting in .NET Core 3, this process is the default.
   - Otherwise, select the **dotnet.exe** process. (This is the out-of-process hosting model.)

   If you have multiple processes showing *w3wp.exe* or *dotnet.exe*, check the **User Name** column. In some scenarios, the **User Name** column shows your app pool name, such as **IIS APPPOOL\DefaultAppPool**. If you see the App Pool, but it's not unique, create a new named App Pool for the app instance you want to debug, and then you can find it easily in the **User Name** column.



7. Select **Attach**.

8. Open the remote computer's website. In a browser, go to **http://<remote computer name>**.

   You should see the ASP.NET web page.

9. In the running ASP.NET application, select the link to the **Privacy** page.

   The breakpoint should be hit in Visual Studio.

   If you're unable to attach or hit the breakpoint, see Troubleshoot remote debugging.

Troubleshooting IIS deployment

- If you can't connect to the host using the host name, try the IP address instead.
- Make sure the required ports are open on the remote server.
- For ASP.NET Core, you need to make sure that the Application pool field for the **DefaultAppPool** is set to **No Managed Code**.
- Verify that the version of ASP.NET used in your app is the same as the version you installed on the server. For your app, you can view and set the version in the **Properties** page. To set the app to a different version, that version must be installed.
- If the app tried to open, but you see a certificate warning, choose to trust the site. If you already closed the warning, you can edit the publishing profile, a *.pubxml

file, in your project and add the following element (for test only):
`<AllowUntrustedCertificate>true</AllowUntrustedCertificate>`

- After it's deployed, start the app in IIS to test that it deployed correctly.
- Check the Output window in Visual Studio for status information, and check your error messages.

Open required ports on Windows Server

In most setups, required ports are opened by the installation of ASP.NET and the remote debugger. However, if you're troubleshooting deployment issues and the app is hosted behind a firewall, you might need to verify that the correct ports are open.

On an Azure VM, you must open ports through:

- The Network security group.
- The firewall on Windows Server

Required ports:

- 80 - Required for IIS
- 4026 - Required for remote debugging from Visual Studio 2022 (see Remote Debugger Port Assignments for more information).
- UDP 3702 - (Optional) The Discovery port enables you to the **Find** button when attaching to the remote debugger in Visual Studio. This must be an outbound port (outbound rule).

In addition, these ports should already be opened by the ASP.NET Core installation:

- 8172 - (Optional) Required for Web Deploy to deploy the app from Visual Studio

---

Feedback

Debug exceptions in .NET applications using Snapshot Debugger

Article • 09/11/2024

When enabled, Snapshot Debugger automatically collects a debug snapshot of the source code and variables when an exception occurs in your live .NET application. The Snapshot Debugger in Application Insights:

- Monitors system-generated logs from your web app.
- Collects snapshots on your top-throwing exceptions.
- Provides information you need to diagnose issues in production.

Learn more about the Snapshot Debugger and Snapshot Uploader processes.

Supported applications and environments

This section lists the applications and environments that are supported.

Applications

Snapshot collection is available for:

- .NET Framework 4.6.2 and newer versions.
- .NET 6.0 or later on Windows.

Environments

The following environments are supported:

- Azure App Service
- Azure Functions
- Azure Cloud Services running OS family 4 or later
- Azure Service Fabric running on Windows Server 2012 R2 or later
- Azure Virtual Machines and Azure Virtual Machine Scale Sets running Windows Server 2012 R2 or later
- On-premises virtual or physical machines running Windows Server 2012 R2 or later or Windows 8.1 or later

> **Note**

Client applications (for example, WPF, Windows Forms, or UWP) aren't supported.

Prerequisites for using Snapshot Debugger

Packages and configurations

- Include the Snapshot Collector NuGet package in your application.
- Configure collection parameters in ApplicationInsights.config.

Permissions

- Verify you're added to the Application Insights Snapshot Debugger role for the target **Application Insights Snapshot**.

How Snapshot Debugger works

The Snapshot Debugger is implemented as an Application Insights telemetry processor. When your application runs, the Snapshot Debugger telemetry processor is added to your application's system-generated logs pipeline.

> **Important**
>
> Snapshots might contain personal data or other sensitive information in variable and parameter values. Snapshot data is stored in the same region as your Application Insights resource.

Snapshot Debugger process

The Snapshot Debugger process starts and ends with the `TrackException` method. A process snapshot is a suspended clone of the running process, so that your users experience little to no interruption. In a typical scenario:

1. Your application throws the TrackException.
2. The Snapshot Debugger monitors exceptions as they're thrown by subscribing to the AppDomain.CurrentDomain.FirstChanceException event.
3. A counter is incremented for the problem ID.

- When the counter reaches the `ThresholdForSnapshotting` value, the problem ID is added to a collection plan.

> ⓘ **Note**
>
> The `ThresholdForSnapshotting` default minimum value is 1. With this value, your app has to trigger the same exception *twice* before a snapshot is created.

4. The exception event's problem ID is computed and compared against the problem IDs in the collection plan.

5. If there's a match between problem IDs, a **snapshot** of the running process is created.

   - The snapshot is assigned a unique identifier and the exception is stamped with that identifier.

   > ⓘ **Note**
   >
   > The snapshot creation rate is limited by the `SnapshotsPerTenMinutesLimit` setting. By default, the limit is one snapshot every 10 minutes.

6. After the `FirstChanceException` handler returns, the thrown exception is processed as normal.

7. The exception reaches the `TrackException` method again and is reported to Application Insights, along with the snapshot identifier.

> ⓘ **Note**
>
> Set `IsEnabledInDeveloperMode` to `true` if you want to generate snapshots while you debug in Visual Studio.

Snapshot Uploader process

While the Snapshot Debugger process continues to run and serve traffic to users with little interruption, the snapshot is handed off to the Snapshot Uploader process. In a typical scenario, the Snapshot Uploader:

1. Creates a minidump.

2. Uploads the minidump to Application Insights, along with any relevant symbol (*.pdb*) files.

> ⓘ **Note**
>
> No more than 50 snapshots per day can be uploaded.

If you enabled the Snapshot Debugger but you aren't seeing snapshots, see the Troubleshooting guide.

Upgrading Snapshot Debugger

Snapshot Debugger auto-upgrades via the built-in, preinstalled Application Insights site extension.

Manually adding an Application Insights site extension to keep Snapshot Debugger up-to-date is deprecated.

Overhead

The Snapshot Debugger is designed for use in production environments. The default settings include rate limits to minimize the impact on your applications.

However, you may experience small CPU, memory, and I/O overhead associated with the Snapshot Debugger, such as:

- When an exception is thrown in your application
- If the exception handler decides to create a snapshot
- When `TrackException` is called

There is **no additional cost** for storing data captured by Snapshot Debugger.

See example scenarios in which you may experience Snapshot Debugger overhead.

Limitations

This section discusses limitations for the Snapshot Debugger.

- **Data retention**

  Debug snapshots are stored for 15 days. The default data retention policy is set on a per-application basis. If you need to increase this value, you can request an

increase by opening a support case in the Azure portal. For each Application Insights instance, a maximum number of 50 snapshots are allowed per day.

- **Publish symbols**

  The Snapshot Debugger requires symbol files on the production server to:
  - Decode variables
  - Provide a debugging experience in Visual Studio

  By default, Visual Studio 2017 versions 15.2+ publishes symbols for release builds when it publishes to App Service.

  In prior versions, you must add the following line to your publish profile `.pubxml` file so that symbols are published in release mode:

  ```
  XML

  <ExcludeGeneratedDebugSymbol>False</ExcludeGeneratedDebugSymbol>
  ```

  For Azure Compute and other types, make sure that the symbol files are either:
  - In the same folder of the main application `.dll` (typically, `wwwroot/bin`), or
  - Available on the current path.

  For more information on the different symbol options that are available, see the Visual Studio documentation. For best results, we recommend that you use *Full*, *Portable*, or *Embedded*.

- **Optimized builds**

  In some cases, local variables can't be viewed in release builds because of optimizations applied by the JIT compiler.

  However, in App Service, the Snapshot Debugger can deoptimize throwing methods that are part of its collection plan.

  > **Tip**
  >
  > Install the Application Insights Site extension in your instance of App Service to get deoptimization support.

Next steps

Enable the Application Insights Snapshot Debugger for your application:

- Azure App Service
- Azure Functions
- Azure Cloud Services
- Azure Service Fabric
- Azure Virtual Machines and Virtual Machine Scale Sets
- On-premises virtual or physical machines

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Debug live ASP.NET Azure apps using the Snapshot Debugger

Article • 10/20/2022

The Snapshot Debugger takes a snapshot of your in-production apps when code that you're interested in executes. To instruct the debugger to take a snapshot, you set snappoints and logpoints in your code. The debugger lets you see exactly what went wrong, without impacting traffic of your production application. The Snapshot Debugger can help you dramatically reduce the time it takes to resolve issues that occur in production environments.

Snappoints and logpoints are similar to breakpoints, but unlike breakpoints, snappoints don't halt the application when hit. Typically, capturing a snapshot at a snappoint takes 10-20 milliseconds.

In this tutorial, you will:

- ✓ Start the Snapshot Debugger
- ✓ Set a snappoint and view a snapshot
- ✓ Set a logpoint

Prerequisites

- Snapshot Debugger is only available starting in Visual Studio 2017 Enterprise version 15.5 or higher with the **Azure development workload**. (Under the **Individual components** tab, you find it under **Debugging and testing** > **Snapshot debugger**.)

  If it's not already installed, install Visual Studio 2019. If you're updating from a previous Visual Studio installation, run the Visual Studio Installer and check the Snapshot Debugger component in the **ASP.NET and web development workload**.

- Basic or higher Azure App Service plan.

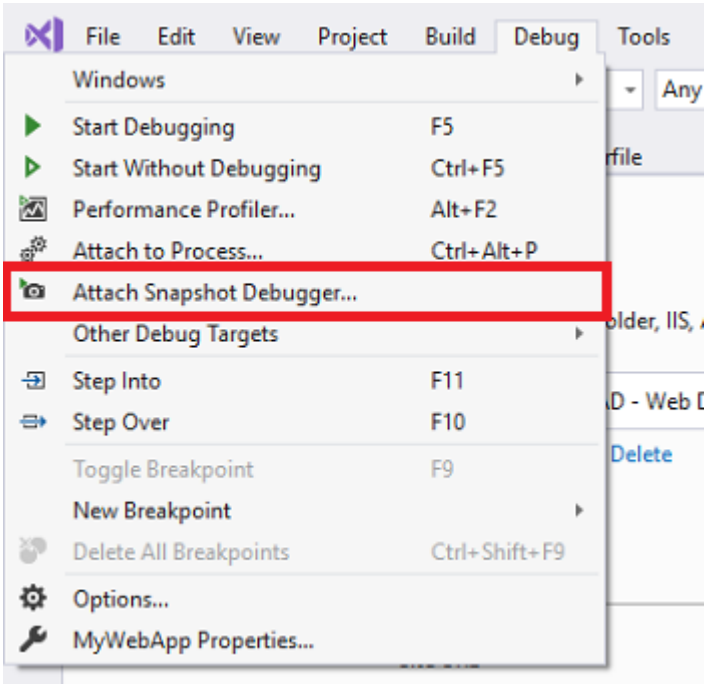
- Snapshot collection is available for the following web apps running in Azure App Service:
  - ASP.NET applications running on .NET Framework 4.6.1 or later.
  - ASP.NET Core applications running on .NET Core 2.0 or later on Windows.

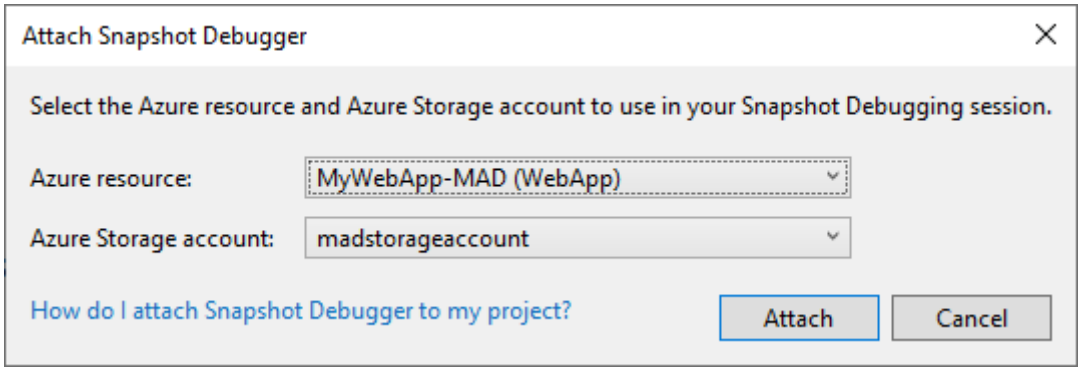
Open your project and start the Snapshot Debugger

1. Open the project you would like to snapshot debug.

   > **ⓘ Important**
   >
   > To snapshot debug, you need to open the *same version of source code* that is published to your Azure App Service.

2. Choose **Debug** > **Attach Snapshot Debugger...**. Select the Azure App Service your project is deployed to and an Azure storage account, and then click **Attach**. Snapshot Debugger also supports Azure Kubernetes Service and Azure Virtual Machines (VM) & Virtual Machine Scale Sets.





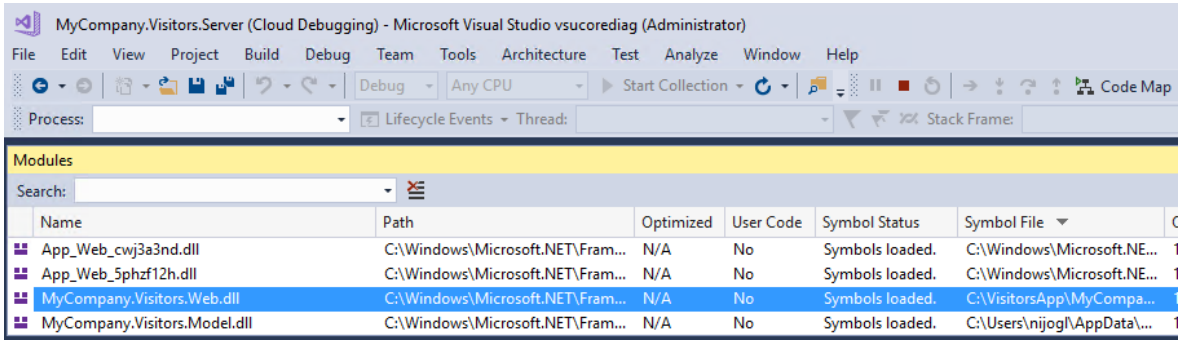
> **ⓘ Important**

The first time you select **Attach Snapshot Debugger**, you're prompted to install the Snapshot Debugger site extension on your Azure App Service. This installation requires a restart of your Azure App Service.

> **Note**
>
> (Visual Studio 2019 version 16.2 and above) Snapshot Debugger has enabled Azure cloud support. Make sure that both the Azure resource and Azure Storage account you select are from the same cloud. Please contact your Azure administrator if you have questions about your enterprise's Azure compliance configurations.

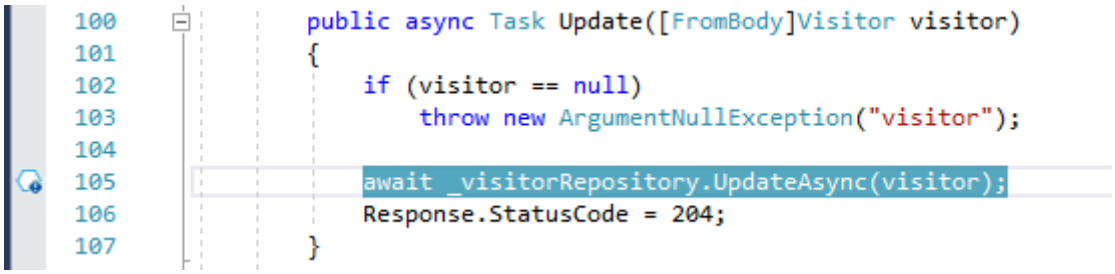
Visual Studio is now in snapshot debugging mode.

The **Modules** window shows you when all the modules have loaded for the Azure App Service (choose **Debug** > **Windows** > **Modules** to open this window).

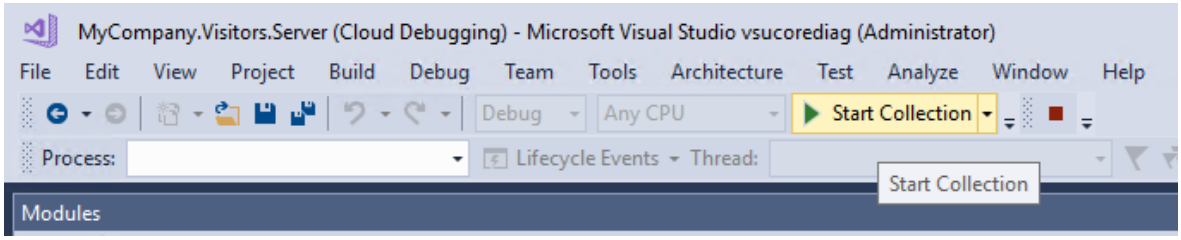


Set a snappoint

1. In the code editor, click the left gutter next to a line of code you're interested in to set a snappoint. Make sure it's code that you know will execute.



2. Click **Start Collection** to turn on the snappoint.



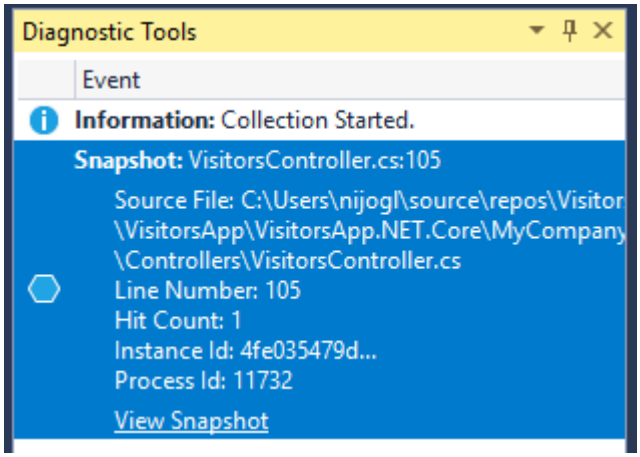
> **Tip**
>
> You can't step when viewing a snapshot, but you can place multiple snappoints in your code to follow execution at different lines of code. If you have multiple snappoints in your code, the Snapshot Debugger makes sure that the corresponding snapshots are from the same end-user session. The Snapshot Debugger does this even if there are many users hitting your app.

Take a snapshot

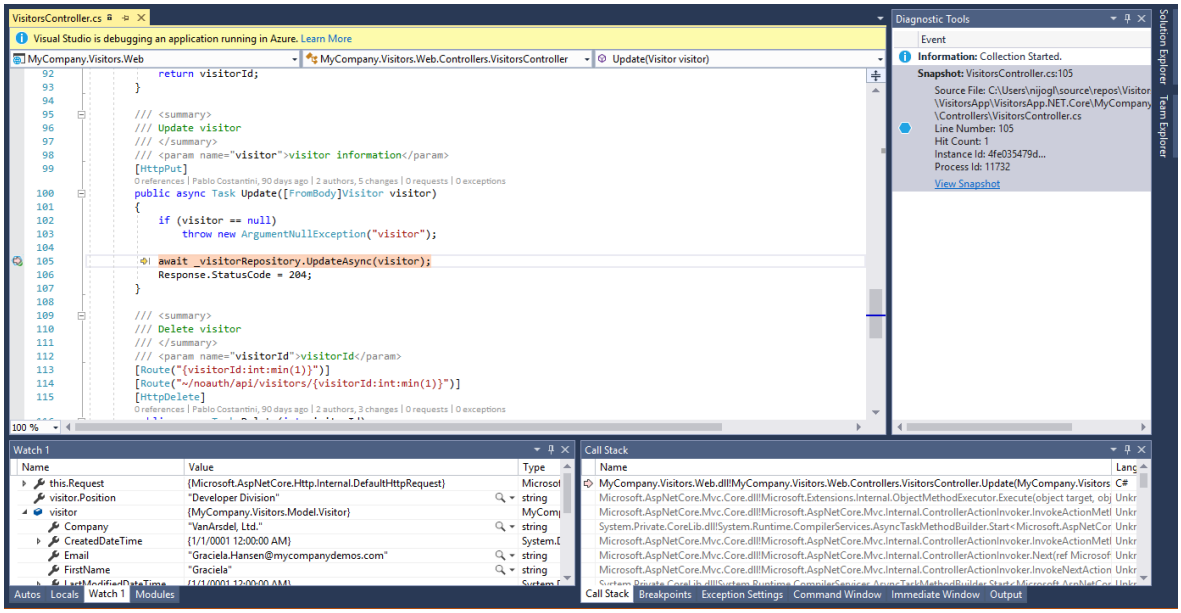
Once a snappoint is set, you can either manually generate a snapshot by going to the browser view of your web site and running the line of code marked or wait for your users to generate one from their usage of the site.

Inspect snapshot data

1. When the snappoint is hit, a snapshot appears in the Diagnostic Tools window. To open this window, choose **Debug** > **Windows** > **Show Diagnostic Tools**.



2. Double-click the snappoint to open the snapshot in the code editor.



From this view, you can hover over variables to view DataTips, use the **Locals**, **Watches**, and **Call Stack** windows, and also evaluate expressions.

The website itself is still live and end users aren't affected. Only one snapshot is captured per snappoint by default: after a snapshot is captured the snappoint turns off. If you want to capture another snapshot at the snappoint, you can turn the snappoint back on by clicking **Update Collection**.

You can also add more snappoints to your app and turn them on with the **Update Collection** button.

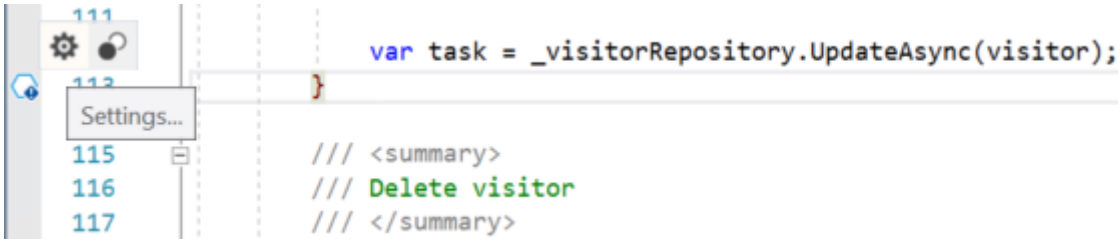
Need help? See the Troubleshooting and known issues and FAQ for snapshot debugging pages.

Set a conditional snappoint

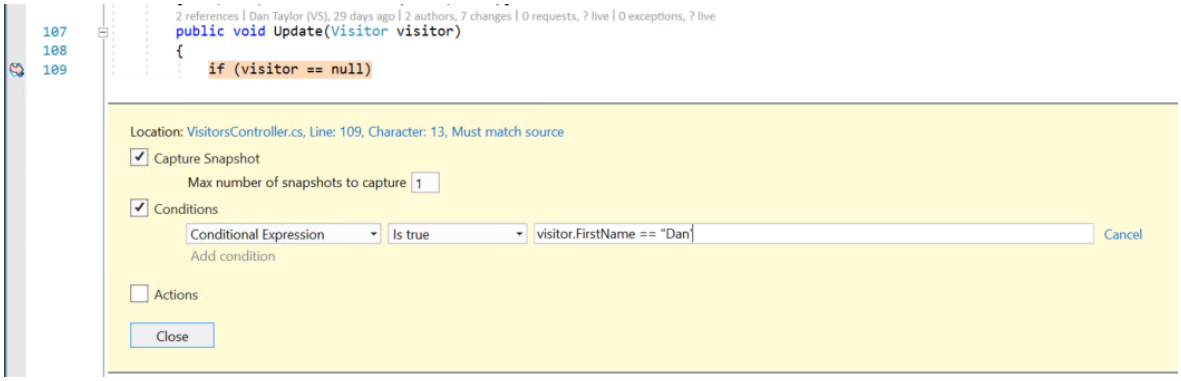
If it's difficult to recreate a particular state in your app, consider using a conditional snappoint. Conditional snappoints help you control when to take a snapshot such as when a variable contains a particular value that you want to inspect. You can set conditions using expressions, filters, or hit counts.

To create a conditional snappoint

1. Right-click a snappoint icon (the hollow sphere) and choose **Settings**.



2. In the snappoint settings window, type an expression.



In the preceding illustration, the snapshot is only taken for the snappoint when `visitor.FirstName == "Dan"`.

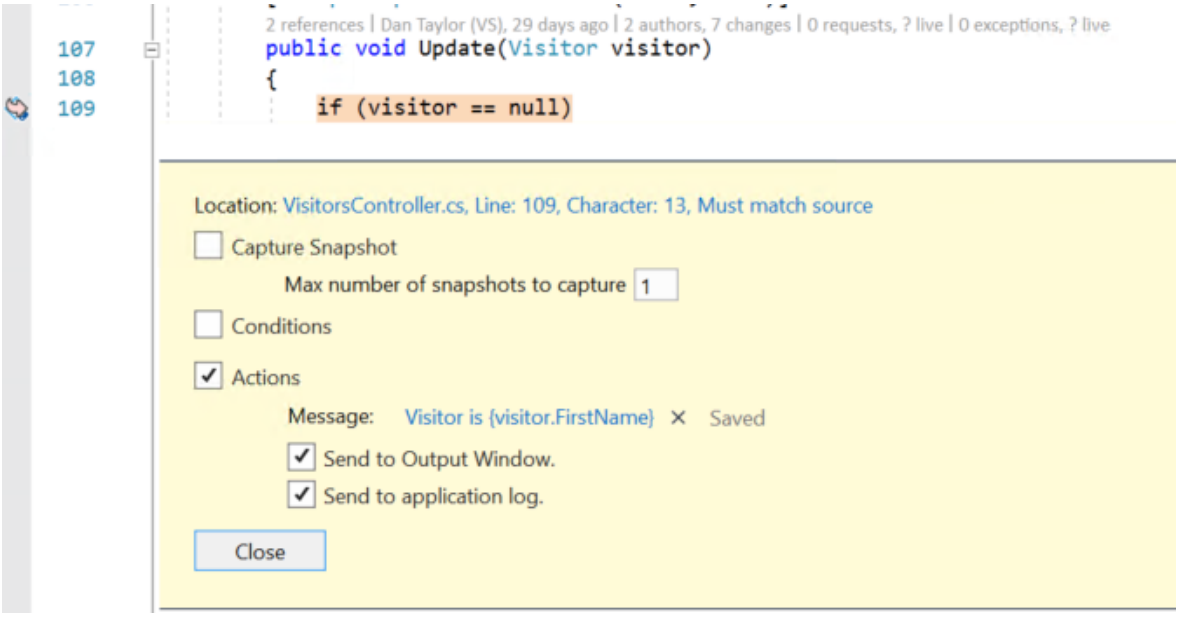
Set a logpoint

In addition to taking a snapshot when a snappoint is hit, you can also configure a snappoint to log a message (that is, create a logpoint). You can set logpoints without having to redeploy your app. Logpoints are executed virtually and cause no impact or side effects to your running application.

To create a logpoint

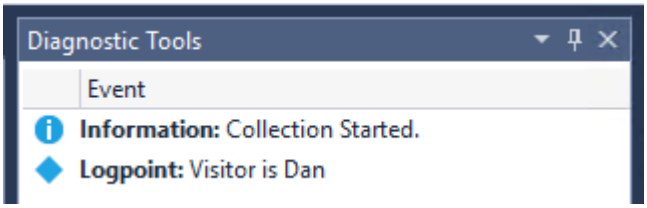
1. Right-click a snappoint icon (the blue hexagon) and choose **Settings**.

2. In the snappoint settings window, select **Actions**.



3. In the **Message** field, you can enter the new log message you want to log. You can also evaluate variables in your log message by placing them inside curly braces.

   If you choose **Send to Output Window**, when the logpoint is hit, the message appears in the Diagnostic Tools window.



   If you choose **Send to application log**, when the logpoint is hit, the message appears anywhere that you can see messages from `System.Diagnostics.Trace` (or `ILogger` in .NET Core), such as App Insights.

Related content

In this tutorial, you've learned how to use the Snapshot Debugger for App Services. You may want to read more details about this feature.

FAQ for snapshot debugging

Feedback

Was this page helpful? Yes No

Use dev tunnels in Visual Studio to debug your web APIs

Article • 11/14/2023

To quickly debug and test your web APIs within Microsoft Power Automate or Power Apps, use *dev tunnels* in Visual Studio. Dev tunnels enables ad-hoc connections between machines that can't directly connect to each other. Once you enable this feature, you'll see that debugging (F5) automatically creates a dev tunnel URL that you can use to connect to Power Apps or Power Automate.

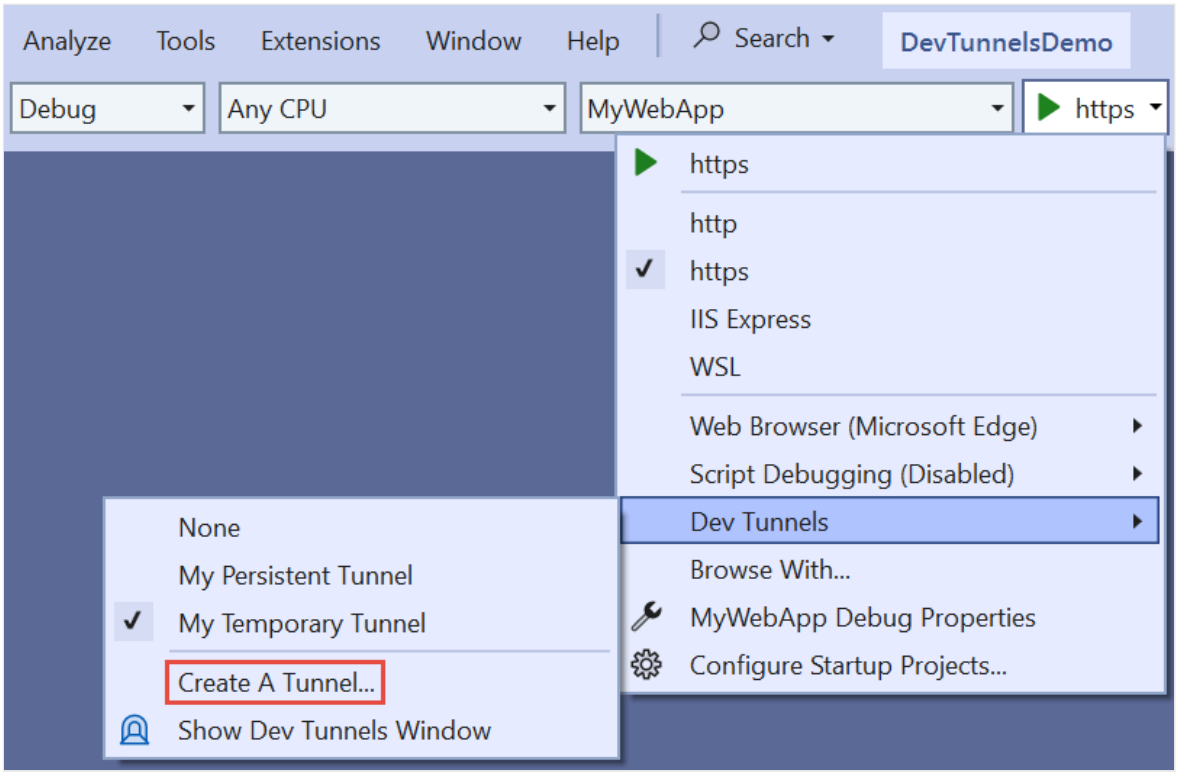
Prerequisites

- Download Visual Studio 2022 Preview version 17.6 or later with the **ASP.NET and web development workload** installed. You need to sign in to Visual Studio to create and use dev tunnels. The feature isn't available in Visual Studio for Mac.
- One of the following Power Platform environments:
  - Power Automate
  - Power Apps

> **Note**
>
> If you need help getting started with Microsoft Power Platform, go to Create a developer environment.

Step 1: Configure your ASP.NET Core project in Visual Studio

1. In the debug dropdown menu, select **Dev Tunnels** > **Create A Tunnel**.



2. The tunnel creation dialog opens and you can configure dev tunnels. Make sure to set authentication type to **Public**.

   To learn more, go to How to use dev tunnels in Visual Studio 2022 with ASP.NET Core apps.

3. Select **OK**. Visual Studio displays confirmation of tunnel creation. The tunnel is now enabled and appears in the debug dropdown **Dev Tunnels** flyout.

4. Select **F5** (**Debug** > **Start Debugging**), or the **Start Debugging** button to see the dev tunnel URL.

URL with and without dev tunnels

To learn more, go to Use a tunnel.

- **Before debugging:** `https://localhost:7223/swagger/index.html`
- **After debugging:** `https://50tt58xr-7223.usw2.devtunnels.ms/swagger/indexf.html`

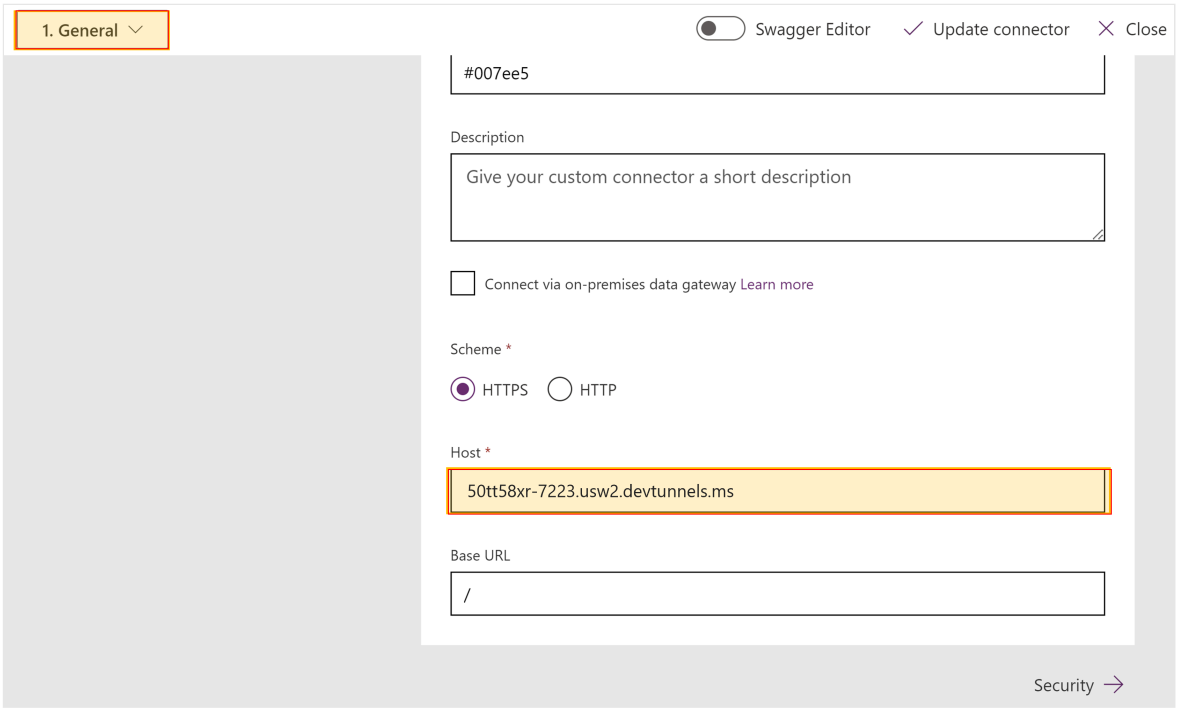
Step 2: Create a custom connector for your web API using the dev tunnel URL

A custom connector is a wrapper around a REST API and allows Power Automate or Power Apps solutions to communicate with your web API. There are many ways to

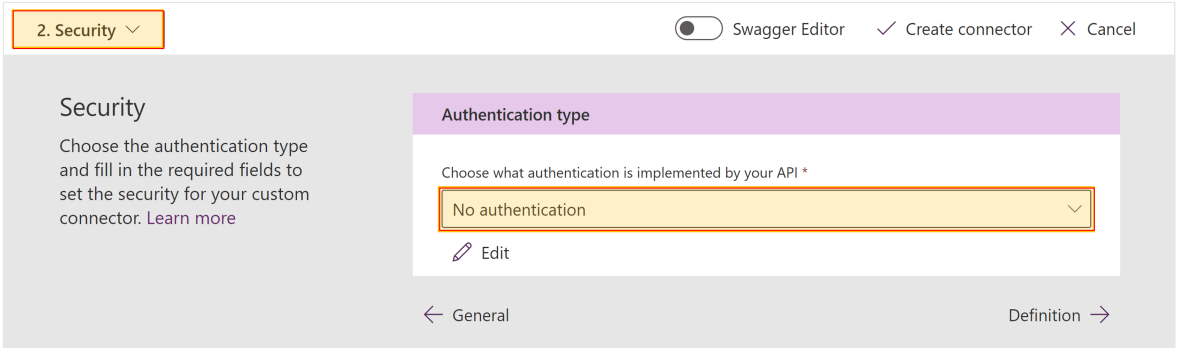
create a custom connector. The following sections explain how to use the dev tunnel URL and create a custom connector from scratch, or with API Management.

Create a custom connector from scratch

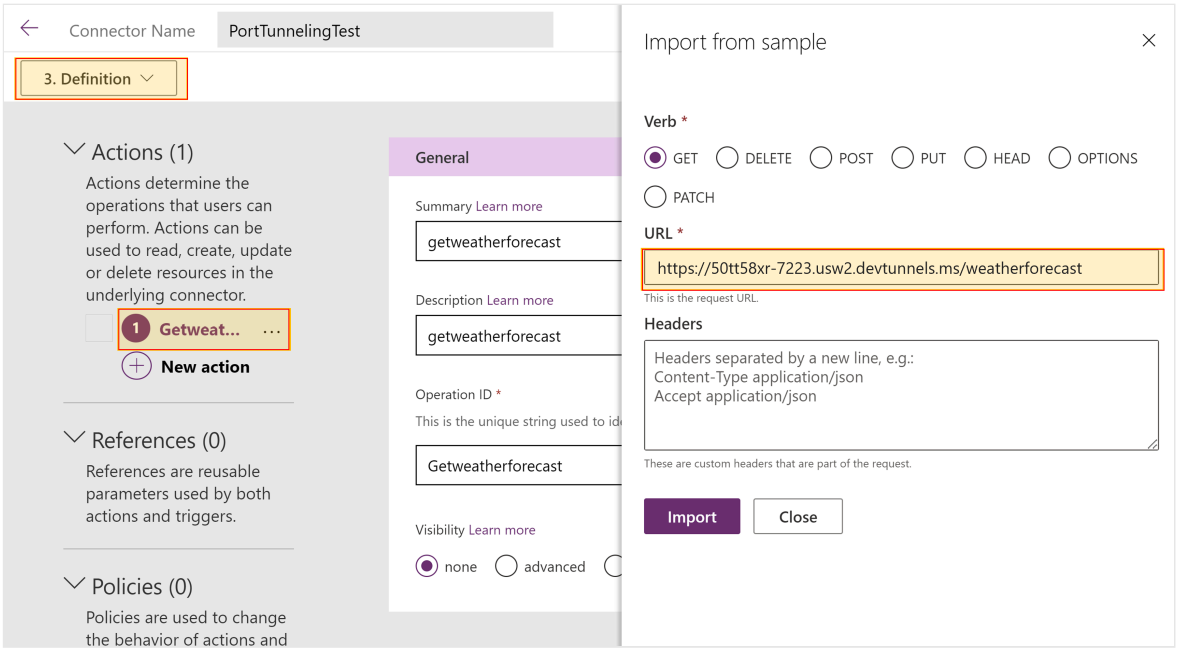
1. On the **General** tab, post the dev tunnel URL into the **Host** field.



2. On the **Security** tab, select **No authentication** from the dropdown menu.



3. On the **Definition** tab, define your HTTP methods by adding actions. For the URL action, use the dev tunnel base URL + /actionName. For an example, go to How to use dev tunnels.

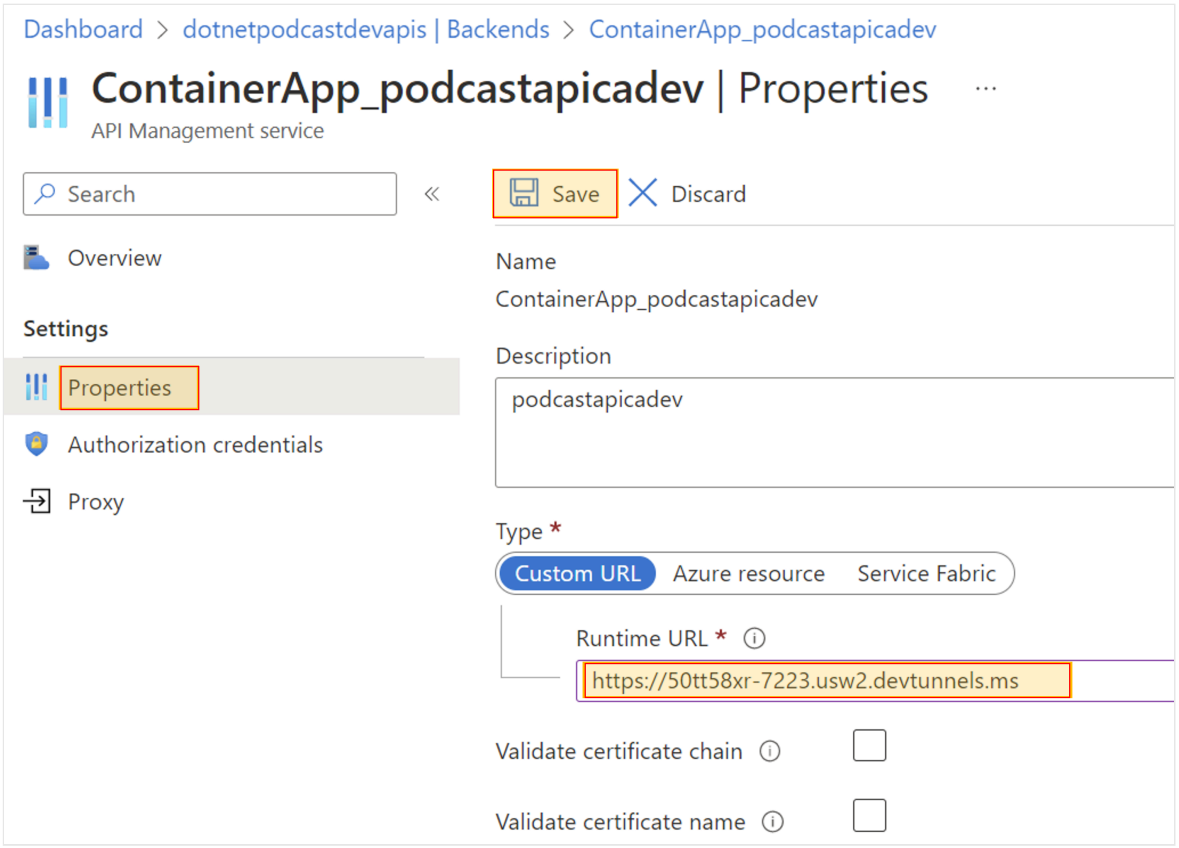


4. You can now test your custom connector. To do this, select the **Test** tab. After adding your connection, you can test your web API.

   For instructions, go to Create a custom connector from scratch.

Create a custom connector with API Management

1. Go to your Azure API Management instance in the Azure portal.
2. Modify the runtime URL of your API in the menu under **Backends** and select your API instance.
3. On the **Properties** tab, replace the **Runtime URL** with the dev tunnel URL and select **Save**.



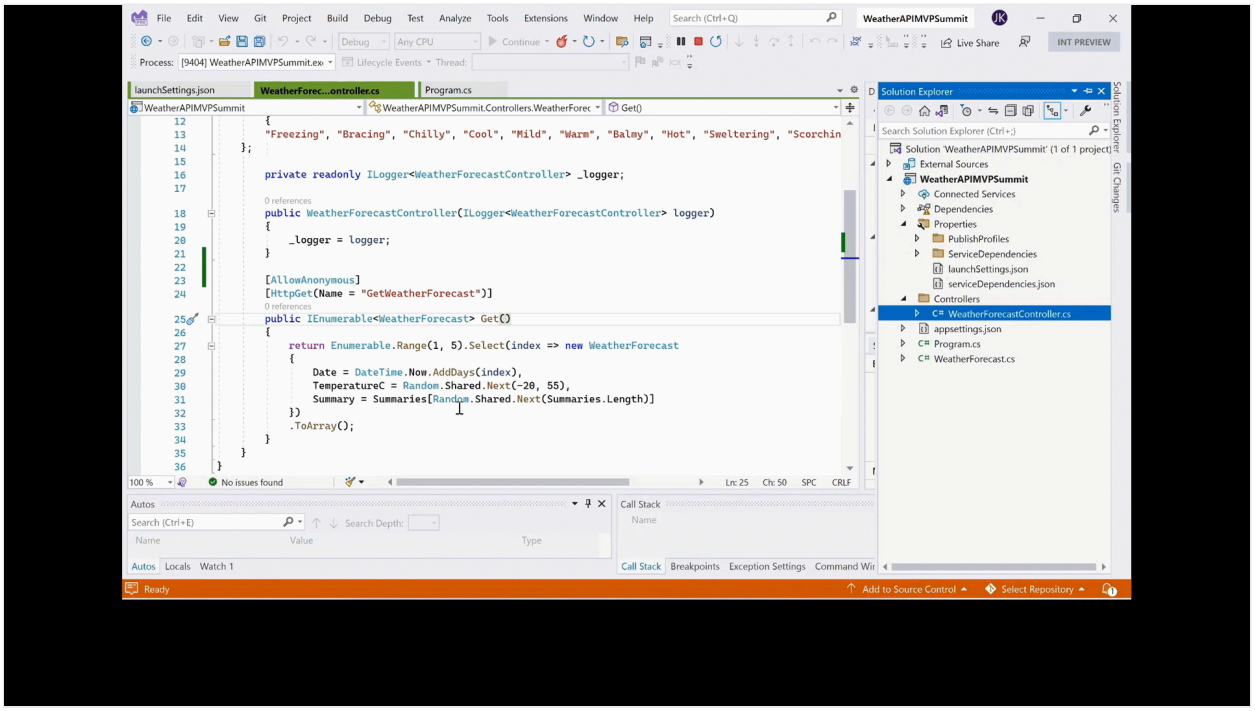
4. On the **Power Platform** tab, you can now create a custom connector. For instructions, go to Export APIs from Azure API Management to the Power Platform.

Step 3: Add the custom connector to Power Apps or Power Automate

To debug your web API, use a custom connector from a Power Apps app or a Power Automate flow.

When your custom connector is integrated in your Power Platform solution, you can set a *breakpoint*, and debug your Power Apps app or Power Automate flow.

> **Note**
>
> Breakpoints are the most basic and essential feature of reliable debugging. A breakpoint indicates where Visual Studio should suspend your running code so you review the values of variables, behavior of memory, or if a branch of code is getting run.



Provide feedback

We greatly appreciate feedback on issues with our connector platform, or new feature ideas. To provide feedback, go to Submit issues or get help with connectors and select your feedback type.

Troubleshoot and debug ASP.NET Core projects

Article • 09/17/2024

By Rick Anderson

The following links provide troubleshooting guidance:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- NDC Conference (London, 2018): Diagnosing issues in ASP.NET Core Applications
- ASP.NET Blog: Troubleshooting ASP.NET Core Performance Problems

.NET Core SDK warnings

Both the 32-bit and 64-bit versions of the .NET Core SDK are installed

In the **New Project** dialog for ASP.NET Core, you may see the following warning:

> Both 32-bit and 64-bit versions of the .NET Core SDK are installed. Only templates from the 64-bit versions installed at 'C:\Program Files\dotnet\sdk\' are displayed.

This warning appears when both 32-bit (x86) and 64-bit (x64) versions of the .NET Core SDK are installed. Common reasons both versions may be installed include:

- You originally downloaded the .NET Core SDK installer using a 32-bit machine but then copied it across and installed it on a 64-bit machine.
- The 32-bit .NET Core SDK was installed by another application.
- The wrong version was downloaded and installed.

Uninstall the 32-bit .NET Core SDK to prevent this warning. Uninstall from **Control Panel** > **Programs and Features** > **Uninstall or change a program**. If you understand why the warning occurs and its implications, you can ignore the warning.

The .NET Core SDK is installed in multiple locations

In the **New Project** dialog for ASP.NET Core, you may see the following warning:

> The .NET Core SDK is installed in multiple locations. Only templates from the SDKs installed at 'C:\Program Files\dotnet\sdk\' are displayed.

You see this message when you have at least one installation of the .NET Core SDK in a directory outside of *C:\Program Files\dotnet\sdk*. Usually this happens when the .NET Core SDK has been deployed on a machine using copy/paste instead of the MSI installer.

Uninstall all 32-bit .NET Core SDKs and runtimes to prevent this warning. Uninstall from **Control Panel** > **Programs and Features** > **Uninstall or change a program**. If you understand why the warning occurs and its implications, you can ignore the warning.

No .NET Core SDKs were detected

- In the Visual Studio **New Project** dialog for ASP.NET Core, you may see the following warning:

  > No .NET Core SDKs were detected, ensure they are included in the environment variable `PATH`.

- When executing a `dotnet` command, the warning appears as:

  > It was not possible to find any installed dotnet SDKs.

These warnings appear when the environment variable `PATH` doesn't point to any .NET Core SDKs on the machine. To resolve this problem:

- Install the .NET Core SDK. Obtain the latest installer from .NET Downloads.
- Verify that the `PATH` environment variable points to the location where the SDK is installed (`C:\Program Files\dotnet\` for 64-bit/x64 or `C:\Program Files (x86)\dotnet\` for 32-bit/x86). The SDK installer normally sets the `PATH`. Always install the same bitness SDKs and runtimes on the same machine.

Missing SDK after installing the .NET Core Hosting Bundle

Installing the .NET Core Hosting Bundle modifies the `PATH` when it installs the .NET Core runtime to point to the 32-bit (x86) version of .NET Core (`C:\Program Files (x86)\dotnet\`). This can result in missing SDKs when the 32-bit (x86) .NET Core `dotnet` command is used (No .NET Core SDKs were detected). To resolve this problem, move