Expand table

Short option	Long option	Description
-o	--output	Specifies the download path for the remote protobuf file. This is a required option.
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.
-s	--services	The type of gRPC services that should be generated. If `Default` is specified, `Both` is used for Web projects and `Client` is used for non-Web projects. Accepted values are `Both`, `Client`, `Default`, `None`, `Server`.
-i	--additional-import-dirs	Additional directories to be used when resolving imports for the protobuf files. This is a semicolon separated list of paths.
	--access	The access modifier to use for the generated C# classes. Default value is `Public`. Accepted values are `Internal` and `Public`.

Remove

The `remove` command is used to remove Protobuf references from the `.csproj` file. The command accepts path arguments and source URLs as arguments. The tool:

- Only removes the Protobuf reference.
- Does not delete the `.proto` file, even if it was originally downloaded from a remote URL.

Usage

.NET CLI

```
dotnet-grpc remove [options] <references>...
```

Arguments

Expand table

Argument	Description
references	The URLs or file paths of the protobuf references to remove.

Options

Expand table

Short option	Long option	Description
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.

Refresh

The `refresh` command is used to update a remote reference with the latest content from the source URL. Both the download file path and the source URL can be used to specify the reference to be updated. Note:

- The hashes of the file contents are compared to determine whether the local file should be updated.
- No timestamp information is compared.

The tool always replaces the local file with the remote file if an update is needed.

Usage

.NET CLI

```
dotnet-grpc refresh [options] [<references>...]
```

Arguments

Expand table

Argument	Description
references	The URLs or file paths to remote protobuf references that should be updated. Leave this argument empty to refresh all remote references.

Options

Expand table

Short option	Long option	Description
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.
	--dry-run	Outputs a list of files that would be updated without downloading any new content.

List

The `list` command is used to display all the Protobuf references in the project file. If all values of a column are default values, the column may be omitted.

Usage

.NET CLI

```
dotnet-grpc list [options]
```

Options

Expand table

Short option	Long option	Description
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.

Additional resources

- Overview for gRPC on .NET
- gRPC services with C#
- gRPC services with ASP.NET Core

Test gRPC services with gRPCurl and gRPCui in ASP.NET Core

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 James Newton-King

Tooling is available for gRPC that allows developers to test services without building client apps:

- gRPCurl is an open-source command-line tool that provides interaction with gRPC services.
- gRPCui builds on top of gRPCurl and adds an open-source interactive web UI for gRPC.

This article discusses how to:

- Set up gRPC server reflection with a gRPC ASP.NET Core app.
- Interact with gRPC using test tools:
  - Discover and test gRPC services with `grpcurl`.
  - Interact with gRPC services via a browser using `grpcui`.

> **Note**
>
> To learn how to unit test gRPC services, see Test gRPC services in ASP.NET Core.

Set up gRPC reflection

Tooling must know the Protobuf contract of services before it can call them. There are two ways to do this:

- Set up gRPC reflection on the server. Tools, such as gRPCurl, use reflection to automatically discover service contracts.
- Add `.proto` files to the tool manually.

It's easier to use gRPC reflection. gRPC reflection adds a new gRPC service to the app that clients can call to discover services.

gRPC ASP.NET Core has built-in support for gRPC reflection with the Grpc.AspNetCore.Server.Reflection package. To configure reflection in an app:

- Add a `Grpc.AspNetCore.Server.Reflection` package reference.
- Register reflection in `Program.cs`:
  - `AddGrpcReflection` to register services that enable reflection.
  - `MapGrpcReflectionService` to add a reflection service endpoint.

C#

```
builder.Services.AddGrpc();
builder.Services.AddGrpcReflection();

var app = builder.Build();

app.MapGrpcService<GreeterService>();

IWebHostEnvironment env = app.Environment;

if (env.IsDevelopment())
{
    app.MapGrpcReflectionService();
}
```

When gRPC reflection is set up:

- A gRPC reflection service is added to the server app.
- Client apps that support gRPC reflection can call the reflection service to discover services hosted by the server.
- gRPC services are still called from the client. Reflection only enables service discovery and doesn't bypass server-side security. Endpoints protected by authentication and authorization require the caller to pass credentials for the endpoint to be called successfully.

Reflection service security

gRPC reflection returns a list of available APIs, which could contain sensitive information. Care should be taken to limit access to the gRPC reflection service.

gRPC reflection is usually only required in a local development environment. For local development, the reflection service should only be mapped when IsDevelopment returns true:

```
C#

if (env.IsDevelopment())
{
    app.MapGrpcReflectionService();
}
```

Access to the service can be controlled through standard ASP.NET Core authorization extension methods, such as AllowAnonymous and RequireAuthorization.

For example, if an app has been configured to require authorization by default, configuration the gRPC reflection endpoint with `AllowAnonymous` to skip authentication and authorization.

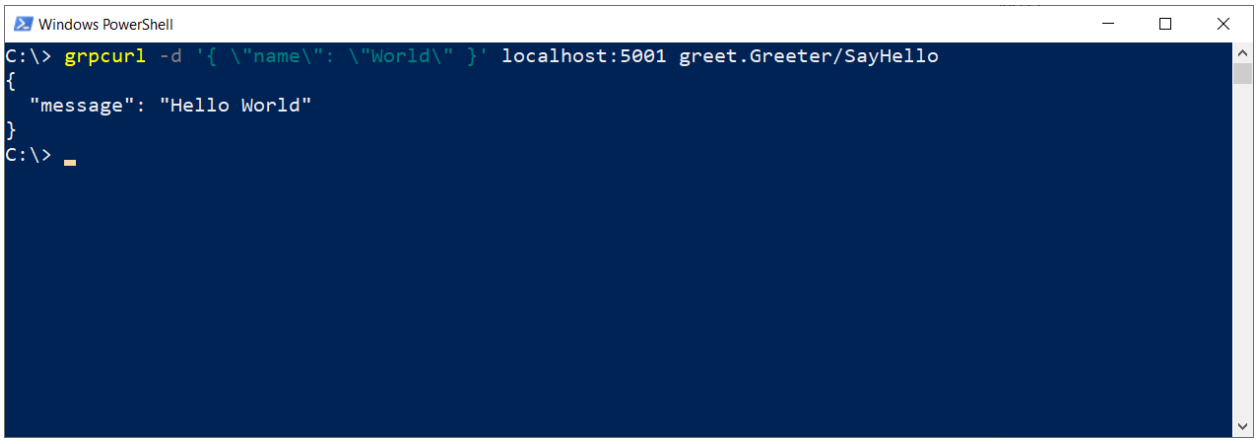
```
C#

if (env.IsDevelopment())
{
    app.MapGrpcReflectionService().AllowAnonymous();
}
```

gRPCurl

gRPCurl is a command-line tool created by the gRPC community. Its features include:

- Calling gRPC services, including streaming services.
- Service discovery using gRPC reflection.
- Listing and describing gRPC services.
- Works with secure (TLS) and insecure (plain-text) servers.

For information about downloading and installing `grpcurl`, see the gRPCurl GitHub homepage.



Use `grpcurl`

The `-help` argument explains `grpcurl` command-line options:

Console

```
$ grpcurl -help
```

Discover services

Use the `describe` verb to view the services defined by the server. Specify `<port>` as the localhost port number of the gRPC server. The port number is randomly assigned when the project is created and set in `Properties/launchSettings.json`:

Console

```
$ grpcurl localhost:<port> describe
greet.Greeter is a service:
service Greeter {
  rpc SayHello ( .greet.HelloRequest ) returns ( .greet.HelloReply );
  rpc SayHellos ( .greet.HelloRequest ) returns ( stream .greet.HelloReply
);
}
grpc.reflection.v1alpha.ServerReflection is a service:
service ServerReflection {
  rpc ServerReflectionInfo ( stream
.grpc.reflection.v1alpha.ServerReflectionRequest ) returns ( stream
.grpc.reflection.v1alpha.ServerReflectionResponse );
}
```

The preceding example:

- Runs the `describe` verb on server `localhost:<port>`. Where `<port>` is randomly assigned when the gRPC server project is created and set in

`Properties/launchSettings.json`

- Prints services and methods returned by gRPC reflection.
  - `Greeter` is a service implemented by the app.
  - `ServerReflection` is the service added by the `Grpc.AspNetCore.Server.Reflection` package.

Combine `describe` with a service, method, or message name to view its detail:

PowerShell

```
$ grpcurl localhost:<port> describe greet.HelloRequest
greet.HelloRequest is a message:
message HelloRequest {
  string name = 1;
}
```

Call gRPC services

Call a gRPC service by specifying a service and method name along with a JSON argument that represents the request message. The JSON is converted into Protobuf and sent to the service.

Console

```
$ grpcurl -d '{ \"name\": \"World\" }' localhost:<port>
greet.Greeter/SayHello
{
  "message": "Hello World"
}
```

In the preceding example:

- The `-d` argument specifies a request message with JSON. This argument must come before the server address and method name.
- Calls the `SayHello` method on the `greeter.Greeter` service.
- Prints the response message as JSON.
- Where `<port>` is randomly assigned when the gRPC server project is created and set in `Properties/launchSettings.json`

The preceding example uses `\` to escape the `"` character. Escaping `"` is required in a PowerShell console but must not be used in some consoles. For example, the previous command for a macOS console:

Console

```
$ grpcurl -d '{ "name": "World" }' localhost:<port> greet.Greeter/SayHello
{
  "message": "Hello World"
}
```

gRPCui

gRPCui is an interactive web UI for gRPC. gRPCui builds on top of gRPCurl. gRPCui offers a GUI for discovering and testing gRPC services, similar to HTTP tools such as Swagger UI.

For information about downloading and installing `grpcui`, see the gRPCui GitHub homepage.

Using `grpcui`

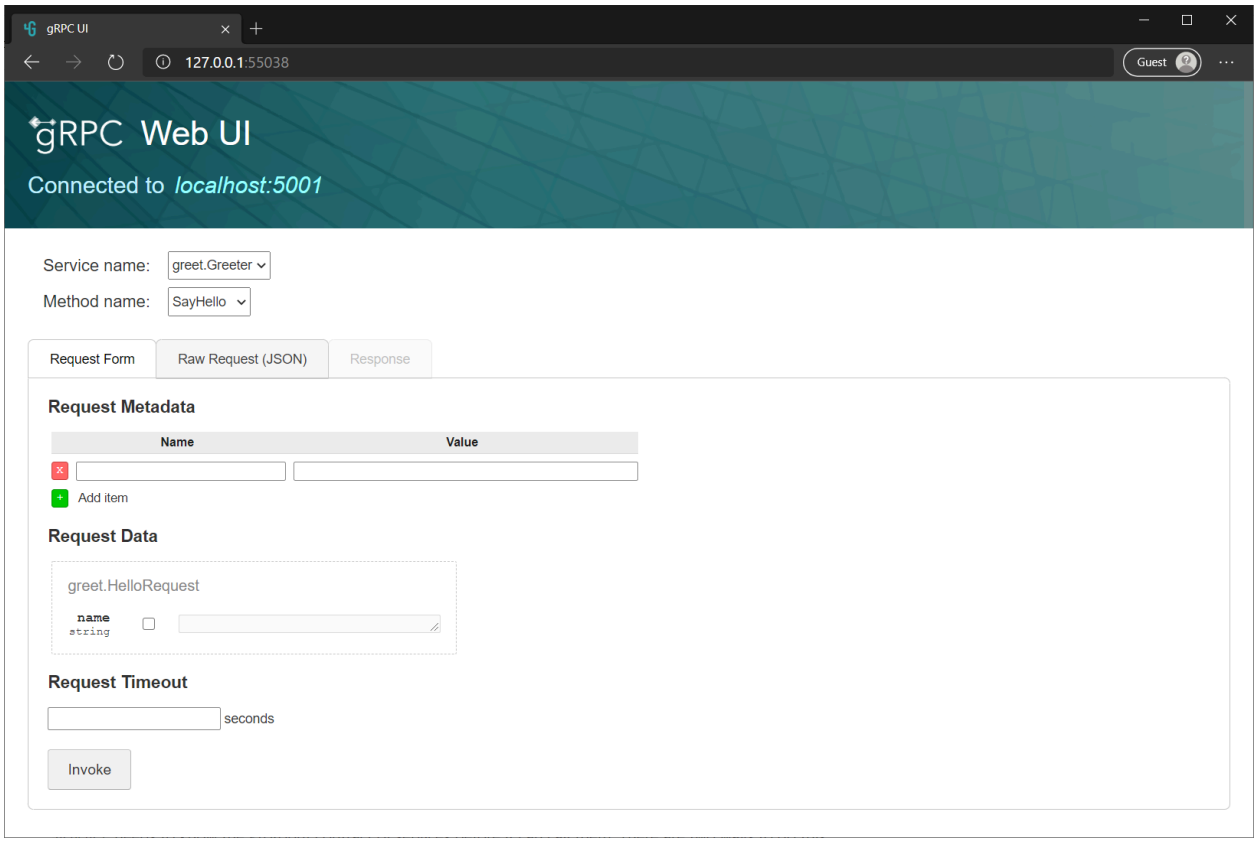
Run `grpcui` with the server address to interact with as an argument:

PowerShell

```
$ grpcui localhost:<port>
gRPC Web UI available at http://127.0.0.1:55038/
```

In the preceding example, specify `<port>` as the localhost port number of the gRPC server. The port number is randomly assigned when the project is created and set in `Properties/launchSettings.json`

The tool launches a browser window with the interactive web UI. gRPC services are automatically discovered using gRPC reflection.



Additional resources

- gRPCurl GitHub homepage
- gRPCui GitHub homepage
- Grpc.AspNetCore.Server.Reflection
- Test gRPC services in ASP.NET Core
- Mock gRPC client in tests

Migrate gRPC from C-core to gRPC for .NET

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.

Due to the implementation of the underlying stack, not all features work in the same way between C-core-based gRPC apps and gRPC for .NET. This document highlights the key differences for migrating between the two stacks.

> **ⓘ Important**
>
> gRPC C-core is in maintenance mode and will be deprecated in favor of gRPC for .NET. gRPC C-core is not recommended for new apps.

Platform support

gRPC C-core and gRPC for .NET have different platform support:

- **gRPC C-core**: A C++ gRPC implementation with its own TLS and HTTP/2 stacks. The `Grpc.Core` package is a .NET wrapper around gRPC C-core and contains a gRPC client and server. It supports .NET Framework, .NET Core, and .NET 5 or later.
- **gRPC for .NET**: Designed for .NET Core 3.x and .NET 5 or later. It uses TLS and HTTP/2 stacks built into modern .NET releases. The `Grpc.AspNetCore` package contains a gRPC server that is hosted in ASP.NET Core and requires .NET Core 3.x or .NET 5 or later. The `Grpc.Net.Client` package contains a gRPC client. The client in `Grpc.Net.Client` has limited support for .NET Framework using WinHttpHandler.

For more information, see gRPC on .NET supported platforms.

Configure server and channel

NuGet packages, configuration, and startup code must be modified when migrating from gRPC C-Core to gRPC for .NET.

gRPC for .NET has separate NuGet packages for its client and server. The packages added depend upon whether an app is hosting gRPC services or calling them:

- `Grpc.AspNetCore`: Services are hosted by ASP.NET Core. For server configuration information, see gRPC services with ASP.NET Core.
- `Grpc.Net.Client`: Clients use `GrpcChannel`, which internally uses networking functionality built into .NET. For client configuration information, see Call gRPC services with the .NET client.

When migration is complete, the `Grpc.Core` package should be removed from the app. `Grpc.Core` contains large native binaries, and removing the package reduces NuGet restore time and app size.

Code generated services and clients

gRPC C-Core and gRPC for .NET share many APIs, and code generated from `.proto` files is compatible with both gRPC implementations. Most clients and service can be migrated from C-Core to gRPC for .NET without changes.

gRPC service implementation lifetime

In the ASP.NET Core stack, gRPC services, by default, are created with a scoped lifetime. In contrast, gRPC C-core by default binds to a service with a singleton lifetime.

A scoped lifetime allows the service implementation to resolve other services with scoped lifetimes. For example, a scoped lifetime can also resolve `DbContext` from the DI container through constructor injection. Using scoped lifetime:

- A new instance of the service implementation is constructed for each request.
- It isn't possible to share state between requests via instance members on the implementation type.
- The expectation is to store shared states in a singleton service in the DI container. The stored shared states are resolved in the constructor of the gRPC service implementation.

For more information on service lifetimes, see Dependency injection in ASP.NET Core.

Add a singleton service

To facilitate the transition from a gRPC C-core implementation to ASP.NET Core, it's possible to change the service lifetime of the service implementation from scoped to singleton. This involves adding an instance of the service implementation to the DI container:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc();
    services.AddSingleton(new GreeterService());
}
```

However, a service implementation with a singleton lifetime is no longer able to resolve scoped services through constructor injection.

Configure gRPC services options

In C-core-based apps, settings such as `grpc.max_receive_message_length` and `grpc.max_send_message_length` are configured with `ChannelOption` when constructing the Server instance.

In ASP.NET Core, gRPC provides configuration through the `GrpcServiceOptions` type. For example, a gRPC service's the maximum incoming message size can be configured via `AddGrpc`. The following example changes the default `MaxReceiveMessageSize` of 4 MB to 16 MB:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.MaxReceiveMessageSize = 16 * 1024 * 1024; // 16 MB
    });
}
```

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

Logging

C-core-based apps rely on the `GrpcEnvironment` to configure the logger for debugging purposes. The ASP.NET Core stack provides this functionality through the

Logging API. For example, a logger can be added to the gRPC service via constructor injection:

```
C#
```

```
public class GreeterService : Greeter.GreeterBase
{
    public GreeterService(ILogger<GreeterService> logger)
    {
    }
}
```

For more information on gRPC logging and diagnostics, see Logging and diagnostics in gRPC on .NET.

HTTPS

C-core-based apps configure HTTPS through the Server.Ports property. A similar concept is used to configure servers in ASP.NET Core. For example, Kestrel uses endpoint configuration for this functionality.

gRPC Interceptors

ASP.NET Core middleware offers similar functionalities compared to interceptors in C-core-based gRPC apps. Both are supported by ASP.NET Core gRPC apps, so there's no need to rewrite interceptors.

For more information on how these features compare to each other, see gRPC Interceptors versus Middleware.

Host gRPC in non-ASP.NET Core projects

A C-core-based server can be added to any project type. gRPC for .NET server requires ASP.NET Core. ASP.NET Core is usually available because the project file specifies `Microsoft.NET.SDK.Web` as the SDK.

A gRPC server can be hosted to non-ASP.NET Core projects by adding `<FrameworkReference Include="Microsoft.AspNetCore.App" />` to a project. The framework reference makes ASP.NET Core APIs available and they can be used to start an ASP.NET Core server.

For more information, see Host gRPC in non-ASP.NET Core projects.

Additional resources

- Overview for gRPC on .NET
- gRPC services with C#
- gRPC services with ASP.NET Core
- Create a .NET Core gRPC client and server in ASP.NET Core

gRPC for Windows Communication Foundation (WCF) developers

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.

This article provides a summary of why ASP.NET Core gRPC is a good fit for Windows Communication Foundation (WCF) developers who want to migrate to modern architectures and platforms.

Comparison to WCF

Although the implementation and approach are different for gRPC, the experience of developing and consuming services with gRPC should be intuitive for WCF developers. WCF and gRPC are RPC (remote procedure call) frameworks with the same goals:

- Make it possible to code as though the client and server are on the same platform.
- Provide a simplified portable networking API.

Both platforms share the requirement of declaring and implementing an interface, although the process for declaring the interface is different. The many types of RPC calls that gRPC supports map well to the bindings available to WCF services. For more information and examples, see Migrate a WCF solution to gRPC.

Benefits of gRPC

gRPC provides a better framework than other approaches for the following reasons.

Performance

gRPC uses HTTP/2. In contrast to HTTP/1.1, HTTP/2:

- Is a smaller, faster binary protocol.

- Is more efficient for computers to parse.
- Supports multiplexing requests over a single connection. Multiplexing enables multiple requests to be sent over one connection without requests blocking each other. In HTTP/1.1, the blocking is known as "head-of-line (HOL) blocking."

gRPC uses Protobuf, an efficient binary format, to serialize messages. Protobuf messages are:

- Fast to serialize and deserialize.
- Use less bandwidth than text-based formats.

gRPC is a good solution for mobile devices and networks with bandwidth restrictions.

Interoperability

There are gRPC tools and libraries for all major programming languages and platforms, including .NET, Java, Python, Go, C++, Node.js, Swift, Dart, Ruby, and PHP. Thanks to the Protobuf binary wire format and the efficient code generation for each platform, developers can build cross-platform, performant apps.

Usability and productivity

gRPC is a comprehensive RPC solution. It works consistently across multiple languages and platforms. It also provides excellent tooling, with much of the boilerplate code automatically generated. Like WCF, gRPC automatically generates messages and a strongly typed client. Developer time is freed up to focus on business logic.

Streaming

gRPC has full bidirectional streaming, which provides similar functionality to WCF's full duplex services. gRPC streaming can operate over regular internet connections, load balancers, and service meshes.

Deadlines, timeouts, and cancellation

gRPC allows clients to specify a maximum time for an RPC to finish. If the specified deadline is exceeded, the server can cancel the operation independently of the client. Deadlines and cancellations can be propagated through subsequent gRPC calls to help enforce resource usage limits. Clients can stop operations when a deadline is exceeded, or earlier if necessary. For example, clients can stop operations because of a user interaction.

Security

gRPC can use TLS and HTTP/2 to provide an end-to-end encrypted connection between the client and the server. Support for client certificate authentication further increases security and trust between client and server.

gRPC as a migration path for WCF to .NET Core and .NET 5

.NET Core and .NET 5 marks a shift in the way that Microsoft delivers remote communication solutions to developers who want to deliver services across a range of platforms. .NET Core and .NET 5 support calling WCF services, but won't offer server-side support for hosting WCF.

There are two recommended paths for modernizing WCF apps:

- gRPC is built on modern technologies and has emerged as the most popular choice across the developer community for RPC apps. Starting with .NET Core 3.0, modern .NET platforms have excellent support for gRPC. Migrating WCF services to use gRPC helps provide the RPC features, performance, an interoperability needed in modern apps.
- CoreWCF is a community effort to bring support for hosting WCF services to .NET Core and .NET 5. A preview release is available and the project is working towards being production ready. CoreWCF only supports a subset of WCF's features, and .NET Framework apps that migrate to use it will need code changes and testing to be successful. CoreWCF is a good choice if an app has to maintain compatibility with existing clients that call WCF services.

Get started

For detailed guidance on building gRPC services in ASP.NET Core for WCF developers, see ASP.NET Core gRPC for WCF Developers

Compare gRPC services with HTTP APIs

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 James Newton-King

This article explains how gRPC services compare to HTTP APIs with JSON (including ASP.NET Core web APIs). The technology used to provide an API for your app is an important choice, and gRPC offers unique benefits compared to HTTP APIs. This article discusses the strengths and weaknesses of gRPC and recommends scenarios for using gRPC over other technologies.

High-level comparison

The following table offers a high-level comparison of features between gRPC and HTTP APIs with JSON.

Expand table

Feature	gRPC	HTTP APIs with JSON
Contract	Required (`.proto`)	Optional (OpenAPI)
Protocol	HTTP/2	HTTP
Payload	Protobuf (small, binary)	JSON (large, human readable)
Prescriptiveness	Strict specification	Loose. Any HTTP is valid.
Streaming	Client, server, bi-directional	Client, server
Browser support	No (requires grpc-web)	Yes
Security	Transport (TLS)	Transport (TLS)
Client code-generation	Yes	OpenAPI + third-party tooling

gRPC strengths

Performance

gRPC messages are serialized using Protobuf, an efficient binary message format. Protobuf serializes very quickly on the server and client. Protobuf serialization results in small message payloads, important in limited bandwidth scenarios like mobile apps.

gRPC is designed for HTTP/2, a major revision of HTTP that provides significant performance benefits over HTTP 1.x:

- Binary framing and compression. HTTP/2 protocol is compact and efficient both in sending and receiving.
- Multiplexing of multiple HTTP/2 calls over a single TCP connection. Multiplexing eliminates head-of-line blocking.

HTTP/2 is not exclusive to gRPC. Many request types, including HTTP APIs with JSON, can use HTTP/2 and benefit from its performance improvements.

Code generation

All gRPC frameworks provide first-class support for code generation. A core file to gRPC development is the .proto file, which defines the contract of gRPC services and messages. From this file, gRPC frameworks generate a service base class, messages, and a complete client.

By sharing the `.proto` file between the server and client, messages and client code can be generated from end to end. Code generation of the client eliminates duplication of messages on the client and server, and creates a strongly-typed client for you. Not having to write a client saves significant development time in applications with many services.

Strict specification

A formal specification for HTTP API with JSON doesn't exist. Developers debate the best format of URLs, HTTP verbs, and response codes.

The gRPC specification is prescriptive about the format a gRPC service must follow. gRPC eliminates debate and saves developer time because gRPC is consistent across platforms and implementations.

Streaming

HTTP/2 provides a foundation for long-lived, real-time communication streams. gRPC provides first-class support for streaming through HTTP/2.

A gRPC service supports all streaming combinations:

- Unary (no streaming)
- Server to client streaming
- Client to server streaming
- Bi-directional streaming

Deadline/timeouts and cancellation

gRPC allows clients to specify how long they are willing to wait for an RPC to complete. The deadline is sent to the server, and the server can decide what action to take if it exceeds the deadline. For example, the server might cancel in-progress gRPC/HTTP/database requests on timeout.

Propagating the deadline and cancellation through child gRPC calls helps enforce resource usage limits.

gRPC recommended scenarios

gRPC is well suited to the following scenarios:

- **Microservices**: gRPC is designed for low latency and high throughput communication. gRPC is great for lightweight microservices where efficiency is critical.
- **Point-to-point real-time communication**: gRPC has excellent support for bi-directional streaming. gRPC services can push messages in real-time without polling.
- **Polyglot environments**: gRPC tooling supports all popular development languages, making gRPC a good choice for multi-language environments.
- **Network constrained environments**: gRPC messages are serialized with Protobuf, a lightweight message format. A gRPC message is always smaller than an equivalent JSON message.
- **Inter-process communication (IPC)**: IPC transports such as Unix domain sockets and named pipes can be used with gRPC to communicate between apps on the same machine. For more information, see Inter-process communication with gRPC.

gRPC weaknesses

Limited browser support

It's impossible to directly call a gRPC service from a browser today. gRPC heavily uses HTTP/2 features and no browser provides the level of control required over web requests to support a gRPC client. For example, browsers do not allow a caller to require that HTTP/2 be used, or provide access to underlying HTTP/2 frames.

gRPC on ASP.NET Core offers two browser-compatible solutions:

- **gRPC-Web** allows browser apps to call gRPC services with the gRPC-Web client and Protobuf. gRPC-Web requires the browser app to generate a gRPC client. gRPC-Web allows browser apps to benefit from the high-performance and low network usage of gRPC.

  .NET has built-in support for gRPC-Web. For more information, see gRPC-Web in ASP.NET Core gRPC apps.

- **gRPC JSON transcoding** allows browser apps to call gRPC services as if they were RESTful APIs with JSON. The browser app doesn't need to generate a gRPC client or know anything about gRPC. RESTful APIs can be automatically created from gRPC services by annotating the `.proto` file with HTTP metadata. Transcoding allows an app to support both gRPC and JSON web APIs without duplicating the effort of building separate services for both.

  .NET has built-in support for creating JSON web APIs from gRPC services. For more information, see gRPC JSON transcoding in ASP.NET Core gRPC apps.

> **Note**
>
> gRPC JSON transcoding requires .NET 7 or later.

Not human readable

HTTP API requests are sent as text and can be read and created by humans.

gRPC messages are encoded with Protobuf by default. While Protobuf is efficient to send and receive, its binary format isn't human readable. Protobuf requires the message's interface description specified in the `.proto` file to properly deserialize.

Additional tooling is required to analyze Protobuf payloads on the wire and to compose requests by hand.

Features such as server reflection and the gRPC command line tool exist to assist with binary Protobuf messages. Also, Protobuf messages support conversion to and from JSON. The built-in JSON conversion provides an efficient way to convert Protobuf messages to and from human readable form when debugging.

Alternative framework scenarios

Other frameworks are recommended over gRPC in the following scenarios:

- **Browser accessible APIs**: gRPC isn't fully supported in the browser. gRPC-Web can offer browser support, but it has limitations and introduces a server proxy.
- **Broadcast real-time communication**: gRPC supports real-time communication via streaming, but the concept of broadcasting a message out to registered connections doesn't exist. For example in a chat room scenario where new chat messages should be sent to all clients in the chat room, each gRPC call is required to individually stream new chat messages to the client. SignalR is a useful framework for this scenario. SignalR has the concept of persistent connections and built-in support for broadcasting messages.

Additional resources

- Create a .NET Core gRPC client and server in ASP.NET Core
- Overview for gRPC on .NET
- gRPC services with C#
- Migrate gRPC from C-core to gRPC for .NET

Troubleshoot gRPC on .NET

Article • 09/27/2024

By James Newton-King

> **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.

ASP.NET Core Best Practices

Article • 09/27/2024

> **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 Mike Rousos

This article provides guidelines for maximizing performance and reliability of ASP.NET Core apps.

Cache aggressively

Caching is discussed in several parts of this article. For more information, see Overview of caching in ASP.NET Core.

Understand hot code paths

In this article, a *hot code path* is defined as a code path that is frequently called and where much of the execution time occurs. Hot code paths typically limit app scale-out and performance and are discussed in several parts of this article.

Avoid blocking calls

ASP.NET Core apps should be designed to process many requests simultaneously. Asynchronous APIs allow a small pool of threads to handle thousands of concurrent requests by not waiting on blocking calls. Rather than waiting on a long-running synchronous task to complete, the thread can work on another request.

A common performance problem in ASP.NET Core apps is blocking calls that could be asynchronous. Many synchronous blocking calls lead to Thread Pool starvation and degraded response times.

Do not block asynchronous execution by calling Task.Wait or Task<TResult>.Result. **Do not** acquire locks in common code paths. ASP.NET Core apps perform best when architected to run code in parallel. **Do not** call Task.Run and immediately await it. ASP.NET Core already runs app code on normal Thread Pool threads, so calling `Task.Run` only results in extra unnecessary Thread Pool scheduling. Even if the scheduled code would block a thread, `Task.Run` does not prevent that.

- **Do** make hot code paths asynchronous.
- **Do** call data access, I/O, and long-running operations APIs asynchronously if an asynchronous API is available.
- **Do not** use Task.Run to make a synchronous API asynchronous.
- **Do** make controller/Razor Page actions asynchronous. The entire call stack is asynchronous in order to benefit from async/await patterns.
- **Consider** using message brokers like Azure Service Bus to offload long-running calls

A profiler, such as PerfView, can be used to find threads frequently added to the Thread Pool. The `Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Start` event indicates a thread added to the thread pool.

Return large collections across multiple smaller pages

A webpage shouldn't load large amounts of data all at once. When returning a collection of objects, consider whether it could lead to performance issues. Determine if the design could produce the following poor outcomes:

- OutOfMemoryException or high memory consumption
- Thread pool starvation (see the following remarks on IAsyncEnumerable<T>)
- Slow response times
- Frequent garbage collection

Do add pagination to mitigate the preceding scenarios. Using page size and page index parameters, developers should favor the design of returning a partial result. When an exhaustive result is required, pagination should be used to asynchronously populate batches of results to avoid locking server resources.

For more information on paging and limiting the number of returned records, see:

- Performance considerations
- Add paging to an ASP.NET Core app

Return `IEnumerable<T>` or `IAsyncEnumerable<T>`

Returning `IEnumerable<T>` from an action results in synchronous collection iteration by the serializer. The result is the blocking of calls and a potential for thread pool starvation. To avoid synchronous enumeration, use `ToListAsync` before returning the enumerable.

Beginning with ASP.NET Core 3.0, `IAsyncEnumerable<T>` can be used as an alternative to `IEnumerable<T>` that enumerates asynchronously. For more information, see Controller action return types.

Minimize large object allocations

The .NET Core garbage collector manages allocation and release of memory automatically in ASP.NET Core apps. Automatic garbage collection generally means that developers don't need to worry about how or when memory is freed. However, cleaning up unreferenced objects takes CPU time, so developers should minimize allocating objects in hot code paths. Garbage collection is especially expensive on large objects (>= 85,000 bytes). Large objects are stored on the large object heap and require a full (generation 2) garbage collection to clean up. Unlike generation 0 and generation 1 collections, a generation 2 collection requires a temporary suspension of app execution. Frequent allocation and de-allocation of large objects can cause inconsistent performance.

Recommendations:

- **Do** consider caching large objects that are frequently used. Caching large objects prevents expensive allocations.
- **Do** pool buffers by using an ArrayPool<T> to store large arrays.
- **Do not** allocate many, short-lived large objects on hot code paths.

Memory issues, such as the preceding, can be diagnosed by reviewing garbage collection (GC) stats in PerfView and examining:

- Garbage collection pause time.
- What percentage of the processor time is spent in garbage collection.
- How many garbage collections are generation 0, 1, and 2.

For more information, see Garbage Collection and Performance.

Optimize data access and I/O

Interactions with a data store and other remote services are often the slowest parts of an ASP.NET Core app. Reading and writing data efficiently is critical for good performance.

Recommendations:

- **Do** call all data access APIs asynchronously.
- **Do not** retrieve more data than is necessary. Write queries to return just the data that's necessary for the current HTTP request.
- **Do** consider caching frequently accessed data retrieved from a database or remote service if slightly out-of-date data is acceptable. Depending on the scenario, use a MemoryCache or a DistributedCache. For more information, see Response caching in ASP.NET Core.
- **Do** minimize network round trips. The goal is to retrieve the required data in a single call rather than several calls.
- **Do** use no-tracking queries in Entity Framework Core when accessing data for read-only purposes. EF Core can return the results of no-tracking queries more efficiently.
- **Do** filter and aggregate LINQ queries (with `.Where`, `.Select`, or `.Sum` statements, for example) so that the filtering is performed by the database.
- **Do** consider that EF Core resolves some query operators on the client, which may lead to inefficient query execution. For more information, see Client evaluation performance issues.
- **Do not** use projection queries on collections, which can result in executing "N + 1" SQL queries. For more information, see Optimization of correlated subqueries.

The following approaches may improve performance in high-scale apps:

- DbContext pooling
- Explicitly compiled queries

We recommend measuring the impact of the preceding high-performance approaches before committing the code base. The additional complexity of compiled queries may not justify the performance improvement.

Query issues can be detected by reviewing the time spent accessing data with Application Insights or with profiling tools. Most databases also make statistics available concerning frequently executed queries.

Pool HTTP connections with HttpClientFactory

Although HttpClient implements the `IDisposable` interface, it's designed for reuse. Closed `HttpClient` instances leave sockets open in the `TIME_WAIT` state for a short period of time. If a code path that creates and disposes of `HttpClient` objects is frequently used, the app may exhaust available sockets. `HttpClientFactory` was introduced in ASP.NET Core 2.1 as a solution to this problem. It handles pooling HTTP connections to optimize performance and reliability. For more information, see Use HttpClientFactory to implement resilient HTTP requests.

Recommendations:

- **Do not** create and dispose of `HttpClient` instances directly.
- **Do** use HttpClientFactory to retrieve `HttpClient` instances. For more information, see Use HttpClientFactory to implement resilient HTTP requests.

Keep common code paths fast

You want all of your code to be fast. Frequently-called code paths are the most critical to optimize. These include:

- Middleware components in the app's request processing pipeline, especially middleware run early in the pipeline. These components have a large impact on performance.
- Code that's executed for every request or multiple times per request. For example, custom logging, authorization handlers, or initialization of transient services.

Recommendations:

- **Do not** use custom middleware components with long-running tasks.
- **Do** use performance profiling tools, such as Visual Studio Diagnostic Tools or PerfView), to identify hot code paths.

Complete long-running Tasks outside of HTTP requests

Most requests to an ASP.NET Core app can be handled by a controller or page model calling necessary services and returning an HTTP response. For some requests that involve long-running tasks, it's better to make the entire request-response process asynchronous.

Recommendations:

- **Do not** wait for long-running tasks to complete as part of ordinary HTTP request processing.
- **Do** consider handling long-running requests with background services or out of process possibly with an Azure Function and/or using a message broker like Azure Service Bus. Completing work out-of-process is especially beneficial for CPU-intensive tasks.
- **Do** use real-time communication options, such as SignalR, to communicate with clients asynchronously.

Minify client assets

ASP.NET Core apps with complex front-ends frequently serve many JavaScript, CSS, or image files. Performance of initial load requests can be improved by:

- Bundling, which combines multiple files into one.
- Minifying, which reduces the size of files by removing whitespace and comments.

Recommendations:

- **Do** use the bundling and minification guidelines, which mention compatible tools and show how to use ASP.NET Core's `environment` tag to handle both `Development` and `Production` environments.
- **Do** consider other third-party tools, such as Webpack, for complex client asset management.

Compress responses

Reducing the size of the response usually increases the responsiveness of an app, often dramatically. One way to reduce payload sizes is to compress an app's responses. For more information, see Response compression.

Use the latest ASP.NET Core release

Each new release of ASP.NET Core includes performance improvements. Optimizations in .NET Core and ASP.NET Core mean that newer versions generally outperform older versions. For example, .NET Core 2.1 added support for compiled regular expressions and benefitted from Span<T>. ASP.NET Core 2.2 added support for HTTP/2. ASP.NET Core 3.0 adds many improvements that reduce memory usage and improve throughput. If performance is a priority, consider upgrading to the current version of ASP.NET Core.

Minimize exceptions

Exceptions should be rare. Throwing and catching exceptions is slow relative to other code flow patterns. Because of this, exceptions shouldn't be used to control normal program flow.

Recommendations:

- **Do not** use throwing or catching exceptions as a means of normal program flow, especially in hot code paths.
- **Do** include logic in the app to detect and handle conditions that would cause an exception.
- **Do** throw or catch exceptions for unusual or unexpected conditions.

App diagnostic tools, such as Application Insights, can help to identify common exceptions in an app that may affect performance.

Avoid synchronous read or write on HttpRequest/HttpResponse body

All I/O in ASP.NET Core is asynchronous. Servers implement the `Stream` interface, which has both synchronous and asynchronous overloads. The asynchronous ones should be preferred to avoid blocking thread pool threads. Blocking threads can lead to thread pool starvation.

Do not do this: The following example uses the ReadToEnd. It blocks the current thread to wait for the result. This is an example of sync over async.

C#

```
public class BadStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public ActionResult<ContosoData> Get()
    {
        var json = new StreamReader(Request.Body).ReadToEnd();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }
}
```

In the preceding code, `Get` synchronously reads the entire HTTP request body into memory. If the client is slowly uploading, the app is doing sync over async. The app does sync over async because Kestrel does **NOT** support synchronous reads.

Do this: The following example uses ReadToEndAsync and does not block the thread while reading.

C#

```
public class GoodStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public async Task<ActionResult<ContosoData>> Get()
    {
        var json = await new StreamReader(Request.Body).ReadToEndAsync();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }

}
```

The preceding code asynchronously reads the entire HTTP request body into memory.

> ⚠ **Warning**
>
> If the request is large, reading the entire HTTP request body into memory could lead to an out of memory (OOM) condition. OOM can result in a Denial Of Service. For more information, see Avoid reading large request bodies or response bodies into memory in this article.

Do this: The following example is fully asynchronous using a non-buffered request body:

C#

```
public class GoodStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public async Task<ActionResult<ContosoData>> Get()
    {
        return await JsonSerializer.DeserializeAsync<ContosoData>
(Request.Body);
    }
}
```

The preceding code asynchronously de-serializes the request body into a C# object.

Prefer ReadFormAsync over Request.Form

Use `HttpContext.Request.ReadFormAsync` instead of `HttpContext.Request.Form`. `HttpContext.Request.Form` can be safely read only with the following conditions:

- The form has been read by a call to `ReadFormAsync`, and
- The cached form value is being read using `HttpContext.Request.Form`

Do not do this: The following example uses `HttpContext.Request.Form`. `HttpContext.Request.Form` uses sync over async and can lead to thread pool starvation.

C#

```
public class BadReadController : Controller
{
    [HttpPost("/form-body")]
    public IActionResult Post()
    {
        var form =  HttpContext.Request.Form;

        Process(form["id"], form["name"]);

        return Accepted();
    }
```

Do this: The following example uses `HttpContext.Request.ReadFormAsync` to read the form body asynchronously.

C#

```
public class GoodReadController : Controller
{
    [HttpPost("/form-body")]
    public async Task<IActionResult> Post()
    {
       var form = await HttpContext.Request.ReadFormAsync();

        Process(form["id"], form["name"]);

        return Accepted();
    }
```

Avoid reading large request bodies or response bodies into memory

In .NET, every object allocation greater than or equal to 85,000 bytes ends up in the large object heap (LOH). Large objects are expensive in two ways:

- The allocation cost is high because the memory for a newly allocated large object has to be cleared. The CLR guarantees that memory for all newly allocated objects is cleared.
- LOH is collected with the rest of the heap. LOH requires a full garbage collection or Gen2 collection.

This blog post describes the problem succinctly:

> When a large object is allocated, it's marked as Gen 2 object. Not Gen 0 as for small objects. The consequences are that if you run out of memory in LOH, GC cleans up the whole managed heap, not only LOH. So it cleans up Gen 0, Gen 1 and Gen 2 including LOH. This is called full garbage collection and is the most time-consuming garbage collection. For many applications, it can be acceptable. But definitely not for high-performance web servers, where few big memory buffers are needed to handle an average web request (read from a socket, decompress, decode JSON, and more).

Storing a large request or response body into a single `byte[]` or `string`:

- May result in quickly running out of space in the LOH.
- May cause performance issues for the app because of full GCs running.

Working with a synchronous data processing API

When using a serializer/de-serializer that only supports synchronous reads and writes (for example, Json.NET):

- Buffer the data into memory asynchronously before passing it into the serializer/de-serializer.

> ⚠ **Warning**
>
> If the request is large, it could lead to an out of memory (OOM) condition. OOM can result in a Denial Of Service. For more information, see **Avoid reading large request bodies or response bodies into memory** in this article.

ASP.NET Core 3.0 uses System.Text.Json by default for JSON serialization. System.Text.Json:

- Reads and writes JSON asynchronously.
- Is optimized for UTF-8 text.
- Typically is higher performance than `Newtonsoft.Json`.

Do not store IHttpContextAccessor.HttpContext in a field

The IHttpContextAccessor.HttpContext returns the `HttpContext` of the active request when accessed from the request thread. The `IHttpContextAccessor.HttpContext` should **not** be stored in a field or variable.

Do not do this: The following example stores the `HttpContext` in a field and then attempts to use it later.

C#

```
public class MyBadType
{
    private readonly HttpContext _context;
    public MyBadType(IHttpContextAccessor accessor)
    {
        _context = accessor.HttpContext;
    }

    public void CheckAdmin()
    {
        if (!_context.User.IsInRole("admin"))
        {
            throw new UnauthorizedAccessException("The current user isn't an
admin");
        }
    }
}
```

The preceding code frequently captures a null or incorrect `HttpContext` in the constructor.

Do this: The following example:

- Stores the IHttpContextAccessor in a field.
- Uses the `HttpContext` field at the correct time and checks for `null`.

C#

```
public class MyGoodType
{
    private readonly IHttpContextAccessor _accessor;
    public MyGoodType(IHttpContextAccessor accessor)
    {
        _accessor = accessor;
    }

    public void CheckAdmin()
    {
        var context = _accessor.HttpContext;
        if (context != null && !context.User.IsInRole("admin"))
        {
            throw new UnauthorizedAccessException("The current user isn't an
admin");
        }
    }
}
```

Do not access HttpContext from multiple threads

`HttpContext` is **not** thread-safe. Accessing `HttpContext` from multiple threads in parallel can result in unexpected behavior such as the server to stop responding, crashes, and data corruption.

Do not do this: The following example makes three parallel requests and logs the incoming request path before and after the outgoing HTTP request. The request path is accessed from multiple threads, potentially in parallel.

C#

```
public class AsyncBadSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        var query1 = SearchAsync(SearchEngine.Google, query);
        var query2 = SearchAsync(SearchEngine.Bing, query);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
```

```
    }

    private async Task<SearchResults> SearchAsync(SearchEngine engine,
string query)
    {
        var searchResults = _searchService.Empty();
        try
        {
            _logger.LogInformation("Starting search query from {path}.",
                                   HttpContext.Request.Path);
            searchResults = _searchService.Search(engine, query);
            _logger.LogInformation("Finishing search query from {path}.",
                                   HttpContext.Request.Path);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed query from {path}",
                             HttpContext.Request.Path);
        }

        return await searchResults;
    }
```

Do this: The following example copies all data from the incoming request before making the three parallel requests.

C#

```
public class AsyncGoodSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        string path = HttpContext.Request.Path;
        var query1 = SearchAsync(SearchEngine.Google, query,
                                 path);
        var query2 = SearchAsync(SearchEngine.Bing, query, path);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query, path);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
    }

    private async Task<SearchResults> SearchAsync(SearchEngine engine,
string query,
                                                  string path)
    {
        var searchResults = _searchService.Empty();
```

```
        try
        {
            _logger.LogInformation("Starting search query from {path}.",
                                   path);
            searchResults = await _searchService.SearchAsync(engine, query);
            _logger.LogInformation("Finishing search query from {path}.",
path);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed query from {path}", path);
        }

        return await searchResults;
    }
```

Do not use the HttpContext after the request is complete

`HttpContext` is only valid as long as there is an active HTTP request in the ASP.NET Core pipeline. The entire ASP.NET Core pipeline is an asynchronous chain of delegates that executes every request. When the `Task` returned from this chain completes, the `HttpContext` is recycled.

Do not do this: The following example uses `async void` which makes the HTTP request complete when the first `await` is reached:

- Using `async void` is **ALWAYS** a bad practice in ASP.NET Core apps.
- The example code accesses the `HttpResponse` after the HTTP request is complete.
- The late access crashes the process.

C#

```
public class AsyncBadVoidController : Controller
{
    [HttpGet("/async")]
    public async void Get()
    {
        await Task.Delay(1000);

        // The following line will crash the process because of writing
after the
        // response has completed on a background thread. Notice async void
Get()

        await Response.WriteAsync("Hello World");
```

```
    }
}
```

Do this: The following example returns a `Task` to the framework, so the HTTP request doesn't complete until the action completes.

C#

```
public class AsyncGoodTaskController : Controller
{
    [HttpGet("/async")]
    public async Task Get()
    {
        await Task.Delay(1000);

        await Response.WriteAsync("Hello World");
    }
}
```

Do not capture the HttpContext in background threads

Do not do this: The following example shows a closure is capturing the `HttpContext` from the `Controller` property. This is a bad practice because the work item could:

- Run outside of the request scope.
- Attempt to read the wrong `HttpContext`.

C#

```
[HttpGet("/fire-and-forget-1")]
public IActionResult BadFireAndForget()
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        var path = HttpContext.Request.Path;
        Log(path);
    });

    return Accepted();
}
```

Do this: The following example:

- Copies the data required in the background task during the request.
- Doesn't reference anything from the controller.

C#

```
[HttpGet("/fire-and-forget-3")]
public IActionResult GoodFireAndForget()
{
    string path = HttpContext.Request.Path;
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        Log(path);
    });

    return Accepted();
}
```

Background tasks should be implemented as hosted services. For more information, see Background tasks with hosted services.

Do not capture services injected into the controllers on background threads

Do not do this: The following example shows a closure that is capturing the `DbContext` from the `Controller` action parameter. This is a bad practice. The work item could run outside of the request scope. The `ContosoDbContext` is scoped to the request, resulting in an `ObjectDisposedException`.

C#

```
[HttpGet("/fire-and-forget-1")]
public IActionResult FireAndForget1([FromServices]ContosoDbContext context)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        context.Contoso.Add(new Contoso());
        await context.SaveChangesAsync();
    });

    return Accepted();
}
```

Do this: The following example:

- Injects an IServiceScopeFactory in order to create a scope in the background work item. `IServiceScopeFactory` is a singleton.
- Creates a new dependency injection scope in the background thread.
- Doesn't reference anything from the controller.
- Doesn't capture the `ContosoDbContext` from the incoming request.

C#

```
[HttpGet("/fire-and-forget-3")]
public IActionResult FireAndForget3([FromServices]IServiceScopeFactory
                                    serviceScopeFactory)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        await using (var scope = serviceScopeFactory.CreateAsyncScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<ContosoDbContext>();

            context.Contoso.Add(new Contoso());

            await context.SaveChangesAsync();
        }
    });

    return Accepted();
}
```

The following highlighted code:

- Creates a scope for the lifetime of the background operation and resolves services from it.
- Uses `ContosoDbContext` from the correct scope.

C#

```
[HttpGet("/fire-and-forget-3")]
public IActionResult FireAndForget3([FromServices]IServiceScopeFactory
                                    serviceScopeFactory)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);
```

```
        await using (var scope = serviceScopeFactory.CreateAsyncScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<ContosoDbContext>();

            context.Contoso.Add(new Contoso());

            await context.SaveChangesAsync();
        }
    });

    return Accepted();
}
```

Do not modify the status code or headers after the response body has started

ASP.NET Core does not buffer the HTTP response body. The first time the response is written:

- The headers are sent along with that chunk of the body to the client.
- It's no longer possible to change response headers.

Do not do this: The following code tries to add response headers after the response has already started:

C#

```
app.Use(async (context, next) =>
{
    await next();

    context.Response.Headers["test"] = "test value";
});
```

In the preceding code, `context.Response.Headers["test"] = "test value";` will throw an exception if `next()` has written to the response.

Do this: The following example checks if the HTTP response has started before modifying the headers.

C#

```
app.Use(async (context, next) =>
{
    await next();
```

```
    if (!context.Response.HasStarted)
    {
        context.Response.Headers["test"] = "test value";
    }
});
```

Do this: The following example uses `HttpResponse.OnStarting` to set the headers before the response headers are flushed to the client.

Checking if the response has not started allows registering a callback that will be invoked just before response headers are written. Checking if the response has not started:

- Provides the ability to append or override headers just in time.
- Doesn't require knowledge of the next middleware in the pipeline.

C#

```
app.Use(async (context, next) =>
{
    context.Response.OnStarting(() =>
    {
        context.Response.Headers["someheader"] = "somevalue";
        return Task.CompletedTask;
    });

    await next();
});
```

Do not call next() if you have already started writing to the response body

Components only expect to be called if it's possible for them to handle and manipulate the response.

Use In-process hosting with IIS

Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter. The loopback adapter is a network interface that returns outgoing network traffic back to the same machine. IIS handles process management with the Windows Process Activation Service (WAS).

Projects default to the in-process hosting model in ASP.NET Core 3.0 and later.

For more information, see Host ASP.NET Core on Windows with IIS

Don't assume that HttpRequest.ContentLength is not null

`HttpRequest.ContentLength` is null if the `Content-Length` header is not received. Null in that case means the length of the request body is not known; it doesn't mean the length is zero. Because all comparisons with null (except `==`) return false, the comparison `Request.ContentLength > 1024`, for example, might return `false` when the request body size is more than 1024. Not knowing this can lead to security holes in apps. You might think you're protecting against too-large requests when you aren't.

For more information, see this StackOverflow answer.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Web server implementations in ASP.NET Core

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 Tom Dykstra, Steve Smith, Stephen Halter, and Chris Ross

An ASP.NET Core app runs with an in-process HTTP server implementation. The server implementation listens for HTTP requests and surfaces them to the app as a set of request features composed into an HttpContext.

Windows

ASP.NET Core ships with the following:

- Kestrel server is the default, cross-platform HTTP server implementation. Kestrel provides the best performance and memory utilization, but it doesn't have some of the advanced features in HTTP.sys. For more information, see Kestrel vs. HTTP.sys in the Windows tab.
- IIS HTTP Server is an in-process server for IIS.
- HTTP.sys server is a Windows-only HTTP server based on the HTTP.sys kernel driver and HTTP Server API.

When using IIS or IIS Express, the app either runs:

- In the same process as the IIS worker process (the in-process hosting model) with the IIS HTTP Server. *In-process* is the recommended configuration.
- In a process separate from the IIS worker process (the out-of-process hosting model) with the Kestrel server.

The ASP.NET Core Module is a native IIS module that handles native IIS requests between IIS and the in-process IIS HTTP Server or Kestrel. For more information, see ASP.NET Core Module (ANCM) for IIS.

Kestrel vs. HTTP.sys

Kestrel has the following advantages over HTTP.sys:

- Better performance and memory utilization.
- Cross platform
- Agility, it's developed and patched independent of the OS.
- Programmatic port and TLS configuration
- Extensibility that allows for protocols like PPv2 and alternate transports.

Http.Sys operates as a shared kernel mode component with the following features that kestrel does not have:

- Port sharing
- Kernel mode windows authentication. Kestrel supports only user-mode authentication.
- Fast proxying via queue transfers
- Direct file transmission
- Response caching

Hosting models

Several hosting models are available:

- Kestrel self-hosting: The Kestrel web server runs without requiring any other external web server such as IIS or HTTP.sys.

- HTTP.sys self-hosting is an alternative to Kestrel. Kestrel is recommended over HTTP.sys unless the app requires features not available in Kestrel.

- IIS in-process hosting: An ASP.NET Core app runs in the same process as its IIS worker process. IIS in-process hosting provides improved performance over IIS out-of-process hosting because requests aren't proxied over the loopback adapter, a network interface that returns outgoing network traffic back to the same machine. IIS handles process management with the Windows Process Activation Service (WAS).

- IIS out-of-process hosting: ASP.NET Core apps run in a process separate from the IIS worker process, and the module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the

Windows Process Activation Service (WAS). Using a separate process also enables hosting more than one app from the same app pool.

For more information, see the following:

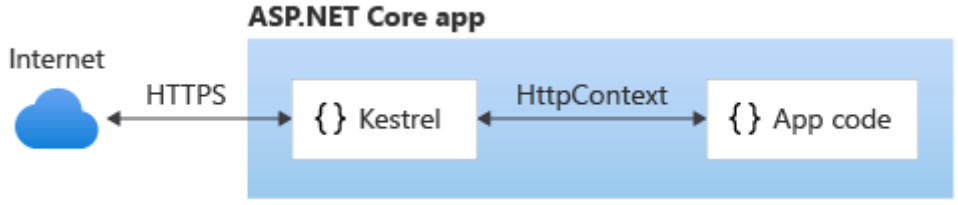
- Kestrel vs. HTTP.sys
- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module (ANCM) for IIS

Kestrel

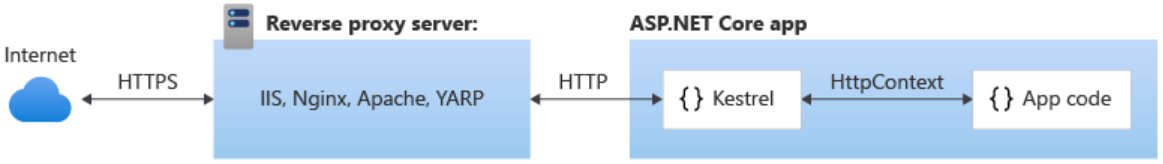
Kestrel server is the default, cross-platform HTTP server implementation. Kestrel provides the best performance and memory utilization, but it doesn't have some of the advanced features in HTTP.sys. For more information, see Kestrel vs. HTTP.sys in this document.

Use Kestrel:

- By itself as an edge server processing requests directly from a network, including the Internet.



- With a *reverse proxy server*, such as Internet Information Services (IIS), Nginx, or Apache. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel.



Either hosting configuration—with or without a reverse proxy server—is supported.

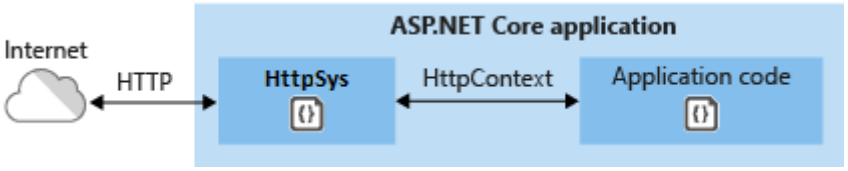
For Kestrel configuration guidance and information on when to use Kestrel in a reverse proxy configuration, see Kestrel web server in ASP.NET Core.

Nginx with Kestrel

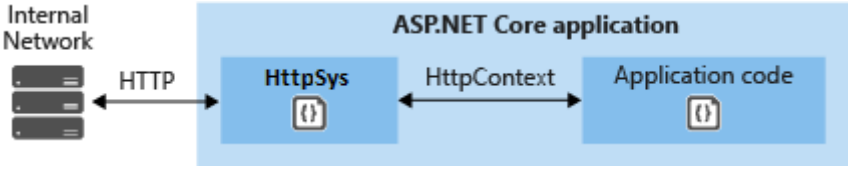
For information on how to use Nginx on Linux as a reverse proxy server for Kestrel, see Host ASP.NET Core on Linux with Nginx.

HTTP.sys

If ASP.NET Core apps are run on Windows, HTTP.sys is an alternative to Kestrel. Kestrel is recommended over HTTP.sys unless the app requires features not available in Kestrel. For more information, see HTTP.sys web server implementation in ASP.NET Core.



HTTP.sys can also be used for apps that are only exposed to an internal network.



For HTTP.sys configuration guidance, see HTTP.sys web server implementation in ASP.NET Core.

ASP.NET Core server infrastructure

The IApplicationBuilder available in the `Startup.Configure` method exposes the ServerFeatures property of type IFeatureCollection. Kestrel and HTTP.sys only expose a single feature each, IServerAddressesFeature, but different server implementations may expose additional functionality.

`IServerAddressesFeature` can be used to find out which port the server implementation has bound at runtime.

Custom servers

If the built-in servers don't meet the app's requirements, a custom server implementation can be created. The Open Web Interface for .NET (OWIN) guide demonstrates how to write a Nowin -based IServer implementation. Only the feature interfaces that the app uses require implementation, though at a minimum IHttpRequestFeature and IHttpResponseFeature must be supported.

Server startup

The server is launched when the Integrated Development Environment (IDE) or editor starts the app:

- Visual Studio: Launch profiles can be used to start the app and server with either IIS Express/ASP.NET Core Module or the console.
- Visual Studio Code: The app and server are started by Omnisharp, which activates the CoreCLR debugger.
- Visual Studio for Mac: The app and server are started by the Mono Soft-Mode Debugger.

When launching the app from a command prompt in the project's folder, dotnet run launches the app and server (Kestrel and HTTP.sys only). The configuration is specified by the `-c|--configuration` option, which is set to either `Debug` (default) or `Release`.

A `launchSettings.json` file provides configuration when launching an app with `dotnet run` or with a debugger built into tooling, such as Visual Studio. If launch profiles are present in a `launchSettings.json` file, use the `--launch-profile {PROFILE NAME}` option with the `dotnet run` command or select the profile in Visual Studio. For more information, see dotnet run and .NET Core distribution packaging.

HTTP/2 support

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

- Kestrel
  - Operating system
    - Windows Server 2016/Windows 10 or later†
    - Linux with OpenSSL 1.0.2 or later (for example, Ubuntu 16.04 or later)
    - macOS 10.15 or later
  - Target framework: .NET Core 2.2 or later
- HTTP.sys
  - Windows Server 2016/Windows 10 or later
  - Target framework: Not applicable to HTTP.sys deployments.
- IIS (in-process)
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - Target framework: .NET Core 2.2 or later
- IIS (out-of-process)
  - Windows Server 2016/Windows 10 or later; IIS 10 or later

    - Public-facing edge server connections use HTTP/2, but the reverse proxy connection to Kestrel uses HTTP/1.1.
    - Target framework: Not applicable to IIS out-of-process deployments.

†Kestrel has limited support for HTTP/2 on Windows Server 2012 R2 and Windows 8.1. Support is limited because the list of supported TLS cipher suites available on these operating systems is limited. A certificate generated using an Elliptic Curve Digital Signature Algorithm (ECDSA) may be required to secure TLS connections.

An HTTP/2 connection must use Application-Layer Protocol Negotiation (ALPN) and TLS 1.2 or later. For more information, see the topics that pertain to your server deployment scenarios.

Additional resources

- Kestrel web server in ASP.NET Core
- ASP.NET Core Module (ANCM) for IIS
- Host ASP.NET Core on Windows with IIS
- Deploy ASP.NET Core apps to Azure App Service
- Host ASP.NET Core on Linux with Nginx
- HTTP.sys web server implementation in ASP.NET Core

Kestrel web server in ASP.NET Core

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 Tom Dykstra, Chris Ross, and Stephen Halter

Kestrel is a cross-platform web server for ASP.NET Core. Kestrel is the recommended server for ASP.NET Core, and it's configured by default in ASP.NET Core project templates.

Kestrel's features include:

- **Cross-platform:** Kestrel is a cross-platform web server that runs on Windows, Linux, and macOS.
- **High performance:** Kestrel is optimized to handle a large number of concurrent connections efficiently.
- **Lightweight:** Optimized for running in resource-constrained environments, such as containers and edge devices.
- **Security hardened:** Kestrel supports HTTPS and is hardened against web server vulnerabilities.
- **Wide protocol support:** Kestrel supports common web protocols, including:
  - HTTP/1.1, HTTP/2 and HTTP/3
  - WebSockets
- **Integration with ASP.NET Core:** Seamless integration with other ASP.NET Core components, such as the middleware pipeline, dependency injection, and configuration system.
- **Flexible workloads**: Kestrel supports many workloads:
  - ASP.NET app frameworks such as Minimal APIs, MVC, Razor pages, SignalR, Blazor, and gRPC.
  - Building a reverse proxy with YARP.
- **Extensibility:** Customize Kestrel through configuration, middleware, and custom transports.

- **Performance diagnostics:** Kestrel provides built-in performance diagnostics features, such as logging and metrics.

Get started

ASP.NET Core project templates use Kestrel by default when not hosted with IIS. In the following template-generated `Program.cs`, the WebApplication.CreateBuilder method calls UseKestrel internally:

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

For more information on configuring `WebApplication` and `WebApplicationBuilder`, see Minimal APIs quick reference.

Optional client certificates

For information on apps that must protect a subset of the app with a certificate, see Optional client certificates.

Behavior with debugger attached

The following timeouts and rate limits aren't enforced when a debugger is attached to a Kestrel process:

- KestrelServerLimits.KeepAliveTimeout
- KestrelServerLimits.RequestHeadersTimeout
- KestrelServerLimits.MinRequestBodyDataRate
- KestrelServerLimits.MinResponseDataRate
- IConnectionTimeoutFeature
- IHttpMinRequestBodyDataRateFeature
- IHttpMinResponseDataRateFeature

Additional resources

- Configure endpoints for the ASP.NET Core Kestrel web server
- Source for WebApplication.CreateBuilder method call to UseKestrel
- Configure options for the ASP.NET Core Kestrel web server
- Use HTTP/2 with the ASP.NET Core Kestrel web server
- When to use a reverse proxy with the ASP.NET Core Kestrel web server
- Host filtering with ASP.NET Core Kestrel web server
- Troubleshoot and debug ASP.NET Core projects
- Enforce HTTPS in ASP.NET Core
- Configure ASP.NET Core to work with proxy servers and load balancers
- RFC 9110: HTTP Semantics (Section 7.2: Host and :authority)
- When using UNIX sockets on Linux, the socket isn't automatically deleted on app shutdown. For more information, see this GitHub issue.

> **Note**
>
> As of ASP.NET Core 5.0, Kestrel's libuv transport is obsolete. The libuv transport doesn't receive updates to support new OS platforms, such as Windows ARM64, and will be removed in a future release. Remove any calls to the obsolete **UseLibuv** method and use Kestrel's default Socket transport instead.

Configure endpoints for the ASP.NET Core Kestrel web server

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.

> **ⓘ 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.

Kestrel endpoints provide the infrastructure for listening to incoming requests and routing them to the appropriate middleware. The combination of an address and a protocol defines an endpoint.

- The address specifies the network interface that the server listens on for incoming requests, such as a TCP port.
- The protocol specifies the communication between the client and server, such as HTTP/1.1, HTTP/2, or HTTP/3.
- An endpoint can be secured using the `https` URL scheme or `UseHttps` method.

Endpoints can be configured using URLs, JSON in `appsettings.json`, and code. This article discusses how to use each option to configure an endpoint:

- Configure endpoints
- Configure HTTPS
- Configure HTTP protocols

Default endpoint

New ASP.NET Core projects are configured to bind to a random HTTP port between 5000-5300 and a random HTTPS port between 7000-7300. The selected ports are stored in the generated `Properties/launchSettings.json` file and can be modified by the developer. The `launchSetting.json` file is only used in local development.

If there's no endpoint configuration, then Kestrel binds to `http://localhost:5000`.

Configure endpoints

Kestrel endpoints listen for incoming connections. When an endpoint is created, it must be configured with the address it will listen to. Usually, this is a TCP address and port number.

There are several options for configuring endpoints:

- Configure endpoints with URLs
- Specify ports only
- Configure endpoints in appsettings.json
- Configure endpoints in code

Configure endpoints with URLs

The following sections explain how to configure endpoints using the:

- `ASPNETCORE_URLS` environment variable.
- `--urls` command-line argument.
- `urls` host configuration key.
- UseUrls extension method.
- WebApplication.Urls property.

URL formats

The URLs indicate the IP or host addresses with ports and protocols the server should listen on. The port can be omitted if it's the default for the protocol (typically 80 and 443). URLs can be in any of the following formats.

- IPv4 address with port number

```
http://65.55.39.10:80/
```

`0.0.0.0` is a special case that binds to all IPv4 addresses.

- IPv6 address with port number

  ```
  http://[0:0:0:0:0:ffff:4137:270a]:80/
  ```

  `[::]` is the IPv6 equivalent of IPv4 `0.0.0.0`.

- Wildcard host with port number

  ```
  http://contoso.com:80/
  http://*:80/
  ```

  Anything not recognized as a valid IP address or `localhost` is treated as a wildcard that binds to all IPv4 and IPv6 addresses. Some people like to use `*` or `+` to be more explicit. To bind different host names to different ASP.NET Core apps on the same port, use HTTP.sys or a reverse proxy server.

  Reverse proxy server examples include IIS, YARP, Nginx, and Apache.

- Host name `localhost` with port number or loopback IP with port number

  ```
  http://localhost:5000/
  http://127.0.0.1:5000/
  http://[::1]:5000/
  ```

  When `localhost` is specified, Kestrel attempts to bind to both IPv4 and IPv6 loopback interfaces. If the requested port is in use by another service on either loopback interface, Kestrel fails to start. If either loopback interface is unavailable for any other reason (most commonly because IPv6 isn't supported), Kestrel logs a warning.

Multiple URL prefixes can be specified by using a semicolon (`;`) delimiter:

```
http://*:5000;http://localhost:5001;https://hostname:5002
```

For more information, see Override configuration.

HTTPS URL prefixes

HTTPS URL prefixes can be used to define endpoints only if a default certificate is provided in the HTTPS endpoint configuration. For example, use KestrelServerOptions configuration or a configuration file, as shown later in this article.

For more information, see Configure HTTPS.

Specify ports only

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://*:8081/
```

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.

Configure endpoints in appsettings.json

Kestrel can load endpoints from an IConfiguration instance. By default, Kestrel configuration is loaded from the `Kestrel` section and endpoints are configured in `Kestrel:Endpoints`:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "MyHttpEndpoint": {
        "Url": "http://localhost:8080"
      }
    }
  }
}
```

The preceding example:

- Uses `appsettings.json` as the configuration source. However, any `IConfiguration` source can be used.
- Adds an endpoint named `MyHttpEndpoint` on port 8080.

For more information about configuring endpoints with JSON, see later sections in this article that discuss configuring HTTPS and configuring HTTP protocols in appsettings.json.

Reloading endpoints from configuration

Reloading endpoint configuration when the configuration source changes is enabled by default. It can be disabled using KestrelServerOptions.Configure(IConfiguration, Boolean).

If a change is signaled, the following steps are taken:

- The new configuration is compared to the old one, and any endpoint without configuration changes isn't modified.
- Removed or modified endpoints are given 5 seconds to complete processing requests and shut down.
- New or modified endpoints are started.

Clients connecting to a modified endpoint may be disconnected or refused while the endpoint is restarted.

ConfigurationLoader

KestrelServerOptions.Configure returns a KestrelConfigurationLoader. The loader's Endpoint(String, Action<EndpointConfiguration>) method that can be used to supplement a configured endpoint's settings:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    var kestrelSection = context.Configuration.GetSection("Kestrel");

    serverOptions.Configure(kestrelSection)
        .Endpoint("HTTPS", listenOptions =>
        {
            // ...
        });
});
```

`KestrelServerOptions.ConfigurationLoader` can be directly accessed to continue iterating on the existing loader, such as the one provided by WebApplicationBuilder.WebHost.

- The configuration section for each endpoint is available on the options in the Endpoint method so that custom settings may be read.
- KestrelServerOptions.Configure(IConfiguration) can be called multiple times, but only the last configuration is used unless `Load` is explicitly called on prior instances. The default host doesn't call `Load` so that its default configuration section may be replaced.
- `KestrelConfigurationLoader` mirrors the `Listen` family of APIs from `KestrelServerOptions` as `Endpoint` overloads, so code and config endpoints can be configured in the same place. These overloads don't use names and only consume default settings from configuration.

Configure endpoints in code

KestrelServerOptions provides methods for configuring endpoints in code:

- Listen
- ListenLocalhost
- ListenAnyIP
- ListenUnixSocket
- ListenNamedPipe

When both the `Listen` and UseUrls APIs are used simultaneously, the `Listen` endpoints override the `UseUrls` endpoints.

Bind to a TCP socket

The Listen, ListenLocalhost, and ListenAnyIP methods bind to a TCP socket:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

The preceding example:

- Configures endpoints that listen on port 5000 and 5001.
- Configures HTTPS for an endpoint with the UseHttps extension method on ListenOptions. For more information, see Configure HTTPS in code.

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

On macOS, Linux, and Windows, certificates can be created using OpenSSL.

Bind to a Unix socket

Listen on a Unix socket with ListenUnixSocket for improved performance with Nginx, as shown in this example:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.ListenUnixSocket("/tmp/kestrel-test.sock");
});
```

- In the Nginx configuration file, set the `server` > `location` > `proxy_pass` entry to `http://unix:/tmp/{KESTREL SOCKET}:/;`. `{KESTREL SOCKET}` is the name of the socket provided to ListenUnixSocket (for example, `kestrel-test.sock` in the preceding example).

- Ensure that the socket is writeable by Nginx (for example, `chmod go+w /tmp/kestrel-test.sock`).

Configure endpoint defaults

ConfigureEndpointDefaults(Action<ListenOptions>) specifies configuration that runs for each specified endpoint. Calling `ConfigureEndpointDefaults` multiple times replaces previous configuration.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureEndpointDefaults(listenOptions =>
    {
        // ...
    });
});
```

> ⓘ Note
>
> Endpoints created by calling Listen **before** calling ConfigureEndpointDefaults won't have the defaults applied.

Dynamic port binding

When port number `0` is specified, Kestrel dynamically binds to an available port. The following example shows how to determine which port Kestrel bound at runtime:

C#

```
app.Run(async (context) =>
{
    var serverAddressFeature = context.Features.Get<IServerAddressesFeature>
();

    if (serverAddressFeature is not null)
    {
        var listenAddresses = string.Join(", ",
serverAddressFeature.Addresses);

        // ...
```

```
    }
});
```

Dynamically binding a port isn't available in some situations:

- KestrelServerOptions.ListenLocalhost
- Binding TCP-based HTTP/1.1 or HTTP/2, and QUIC-based HTTP/3 together.

Configure HTTPS

Kestrel supports securing endpoints with HTTPS. Data sent over HTTPS is encrypted using Transport Layer Security (TLS) to increase the security of data transferred between the client and server.

HTTPS requires a TLS certificate. The TLS certificate is stored on the server, and Kestrel is configured to use it. An app can use the ASP.NET Core HTTPS development certificate in a local development environment. The development certificate isn't installed in nondevelopment environments. In production, a TLS certificate must be explicitly configured. At a minimum, a default certificate must be provided.

The way HTTPS and the TLS certificate is configured depends on how endpoints are configured:

- If URL prefixes or specify ports only are used to define endpoints, HTTPS can be used only if a default certificate is provided in HTTPS endpoint configuration. A default certificate can be configured with one of the following options:
- Configure HTTPS in appsettings.json
- Configure HTTPS in code

Configure HTTPS in appsettings.json

A default HTTPS app settings configuration schema is available for Kestrel. Configure multiple endpoints, including the URLs and the certificates to use, either from a file on disk or from a certificate store.

Any HTTPS endpoint that doesn't specify a certificate (`HttpsDefaultCert` in the example that follows) falls back to the certificate defined under `Certificates:Default` or the development certificate.

The following example is for `appsettings.json`, but any configuration source can be used:

```
JSON
```

```
{
  "Kestrel": {
    "Endpoints": {
      "Http": {
        "Url": "http://localhost:5000"
      },
      "HttpsInlineCertFile": {
        "Url": "https://localhost:5001",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "$CREDENTIAL_PLACEHOLDER$"
        }
      },
      "HttpsInlineCertAndKeyFile": {
        "Url": "https://localhost:5002",
        "Certificate": {
          "Path": "<path to .pem/.crt file>",
          "KeyPath": "<path to .key file>",
          "Password": "$CREDENTIAL_PLACEHOLDER$"
        }
      },
      "HttpsInlineCertStore": {
        "Url": "https://localhost:5003",
        "Certificate": {
          "Subject": "<subject; required>",
          "Store": "<certificate store; required>",
          "Location": "<location; defaults to CurrentUser>",
          "AllowInvalid": "<true or false; defaults to false>"
        }
      },
      "HttpsDefaultCert": {
        "Url": "https://localhost:5004"
      }
    },
    "Certificates": {
      "Default": {
        "Path": "<path to .pfx file>",
        "Password": "$CREDENTIAL_PLACEHOLDER$"
      }
    }
  }
}
```

> **⚠ Warning**
>
> In the preceding example, the certificate password is stored in plain-text in `appsettings.json`. The `$CREDENTIAL_PLACEHOLDER$` token is used as a placeholder for the certificate's password. To store certificate passwords securely in development environments, see Protect secrets in development. To store certificate passwords

securely in production environments, see Azure Key Vault configuration provider. Development secrets shouldn't be used for production or test.

Schema notes

- Endpoint names are case-insensitive. For example, `HTTPS` and `Https` are equivalent.
- The `Url` parameter is required for each endpoint. The format for this parameter is the same as the top-level `Urls` configuration parameter except that it's limited to a single value. See URL formats earlier in this article.
- These endpoints replace the ones defined in the top-level `Urls` configuration rather than adding to them. Endpoints defined in code via `Listen` are cumulative with the endpoints defined in the configuration section.
- The `Certificate` section is optional. If the `Certificate` section isn't specified, the defaults defined in `Certificates:Default` are used. If no defaults are available, the development certificate is used. If there are no defaults and the development certificate isn't present, the server throws an exception and fails to start.
- The `Certificate` section supports multiple certificate sources.
- Any number of endpoints may be defined in `Configuration`, as long as they don't cause port conflicts.

Certificate sources

Certificate nodes can be configured to load certificates from a number of sources:

- `Path` and `Password` to load *.pfx* files.
- `Path`, `KeyPath` and `Password` to load *.pem*/*.crt* and *.key* files.
- `Subject` and `Store` to load from the certificate store.

For example, the `Certificates:Default` certificate can be specified as:

JSON

```
"Default": {
  "Subject": "<subject; required>",
  "Store": "<cert store; required>",
  "Location": "<location; defaults to CurrentUser>",
  "AllowInvalid": "<true or false; defaults to false>"
}
```

Configure client certificates in appsettings.json

ClientCertificateMode is used to configure client certificate behavior.

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "MyHttpsEndpoint": {
        "Url": "https://localhost:5001",
        "ClientCertificateMode": "AllowCertificate",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "$CREDENTIAL_PLACEHOLDER$"
        }
      }
    }
  }
}
```

> ⚠ **Warning**
>
> In the preceding example, the certificate password is stored in plain-text in `appsettings.json`. The `$CREDENTIAL_PLACEHOLDER$` token is used as a placeholder for the certificate's password. To store certificate passwords securely in development environments, see Protect secrets in development. To store certificate passwords securely in production environments, see Azure Key Vault configuration provider. Development secrets shouldn't be used for production or test.

The default value is `ClientCertificateMode.NoCertificate`, where Kestrel doesn't request or require a certificate from the client.

For more information, see Configure certificate authentication in ASP.NET Core.

Configure SSL/TLS protocols in appsettings.json

SSL Protocols are protocols used for encrypting and decrypting traffic between two peers, traditionally a client and a server.

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "MyHttpsEndpoint": {
        "Url": "https://localhost:5001",
        "SslProtocols": ["Tls12", "Tls13"],
```

```
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "$CREDENTIAL_PLACEHOLDER$"
        }
      }
    }
  }
}
```

> ⚠ **Warning**
>
> In the preceding example, the certificate password is stored in plain-text in `appsettings.json`. The `$CREDENTIAL_PLACEHOLDER$` token is used as a placeholder for the certificate's password. To store certificate passwords securely in development environments, see Protect secrets in development. To store certificate passwords securely in production environments, see Azure Key Vault configuration provider. Development secrets shouldn't be used for production or test.

The default value, `SslProtocols.None`, causes Kestrel to use the operating system defaults to choose the best protocol. Unless you have a specific reason to select a protocol, use the default.

Configure HTTPS in code

When using the `Listen` API, the UseHttps extension method on ListenOptions is available to configure HTTPS.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

`ListenOptions.UseHttps` parameters:

- `filename` is the path and file name of a certificate file, relative to the directory that contains the app's content files.

- `password` is the password required to access the X.509 certificate data.
- `configureOptions` is an `Action` to configure the `HttpsConnectionAdapterOptions`. Returns the `ListenOptions`.
- `storeName` is the certificate store from which to load the certificate.
- `subject` is the subject name for the certificate.
- `allowInvalid` indicates if invalid certificates should be considered, such as self-signed certificates.
- `location` is the store location to load the certificate from.
- `serverCertificate` is the X.509 certificate.

For a complete list of `UseHttps` overloads, see UseHttps.

Configure client certificates in code

ClientCertificateMode configures the client certificate requirements.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        listenOptions.ClientCertificateMode =
ClientCertificateMode.AllowCertificate;
    });
});
```

The default value is NoCertificate, where Kestrel doesn't request or require a certificate from the client.

For more information, see Configure certificate authentication in ASP.NET Core.

Configure HTTPS defaults in code

ConfigureHttpsDefaults(Action<HttpsConnectionAdapterOptions>) specifies a configuration `Action` to run for each HTTPS endpoint. Calling `ConfigureHttpsDefaults` multiple times replaces prior `Action` instances with the last `Action` specified.

C#

```
var builder = WebApplication.CreateBuilder(args);
```

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        // ...
    });
});
```

> ⓘ **Note**
>
> Endpoints created by calling **Listen** **before** calling **ConfigureHttpsDefaults** won't have the defaults applied.

Configure SSL/TLS protocols in code

SSL protocols are protocols used for encrypting and decrypting traffic between two peers, traditionally a client and a server.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        listenOptions.SslProtocols = SslProtocols.Tls13;
    });
});
```

Configure TLS cipher suites filter in code

On Linux, CipherSuitesPolicy can be used to filter TLS handshakes on a per-connection basis:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        listenOptions.OnAuthenticate = (context, sslOptions) =>
        {
            sslOptions.CipherSuitesPolicy = new CipherSuitesPolicy(
```

```
                new[]
                {
                    TlsCipherSuite.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
                    TlsCipherSuite.TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
                    // ...
                });
        };
    });
});
```

Configure Server Name Indication

Server Name Indication (SNI) can be used to host multiple domains on the same IP address and port. SNI can be used to conserve resources by serving multiple sites from one server.

For SNI to function, the client sends the host name for the secure session to the server during the TLS handshake so that the server can provide the correct certificate. The client uses the furnished certificate for encrypted communication with the server during the secure session that follows the TLS handshake.

All websites must run on the same Kestrel instance. Kestrel doesn't support sharing an IP address and port across multiple instances without a reverse proxy.

SNI can be configured in two ways:

- Configure a mapping between host names and HTTPS options in Configuration. For example, JSON in the `appsettings.json` file.
- Create an endpoint in code and select a certificate using the host name with the ServerCertificateSelector callback.

Configure SNI in appsettings.json

Kestrel supports SNI defined in configuration. An endpoint can be configured with an `Sni` object that contains a mapping between host names and HTTPS options. The connection host name is matched to the options and they're used for that connection.

The following configuration adds an endpoint named `MySniEndpoint` that uses SNI to select HTTPS options based on the host name:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
```

```
      "MySniEndpoint": {
        "Url": "https://*",
        "SslProtocols": ["Tls11", "Tls12"],
        "Sni": {
          "a.example.org": {
            "Protocols": "Http1AndHttp2",
            "SslProtocols": ["Tls11", "Tls12", "Tls13"],
            "Certificate": {
              "Subject": "<subject; required>",
              "Store": "<certificate store; required>",
            },
            "ClientCertificateMode" : "NoCertificate"
          },
          "*.example.org": {
            "Certificate": {
              "Path": "<path to .pfx file>",
              "Password": "$CREDENTIAL_PLACEHOLDER$"
            }
          },
          "*": {
            // At least one subproperty needs to exist per SNI section or it
            // cannot be discovered via IConfiguration
            "Protocols": "Http1",
          }
        }
      }
    },
    "Certificates": {
      "Default": {
        "Path": "<path to .pfx file>",
        "Password": "$CREDENTIAL_PLACEHOLDER$"
      }
    }
  }
}
```

> ⚠ **Warning**
>
> In the preceding example, the certificate password is stored in plain-text in `appsettings.json`. The `$CREDENTIAL_PLACEHOLDER$` token is used as a placeholder for the certificate's password. To store certificate passwords securely in development environments, see Protect secrets in development. To store certificate passwords securely in production environments, see Azure Key Vault configuration provider. Development secrets shouldn't be used for production or test.

HTTPS options that can be overridden by SNI:

- `Certificate` configures the certificate source.
- `Protocols` configures the allowed HTTP protocols.

- `SslProtocols` configures the allowed SSL protocols.
- `ClientCertificateMode` configures the client certificate requirements.

The host name supports wildcard matching:

- Exact match. For example, `a.example.org` matches `a.example.org`.
- Wildcard prefix. If there are multiple wildcard matches, then the longest pattern is chosen. For example, `*.example.org` matches `b.example.org` and `c.example.org`.
- Full wildcard. `*` matches everything else, including clients that aren't using SNI and don't send a host name.

The matched SNI configuration is applied to the endpoint for the connection, overriding values on the endpoint. If a connection doesn't match a configured SNI host name, then the connection is refused.

Configure SNI with code

Kestrel supports SNI with several callback APIs:

- `ServerCertificateSelector`
- `ServerOptionsSelectionCallback`
- `TlsHandshakeCallbackOptions`

SNI with `ServerCertificateSelector`

Kestrel supports SNI via the `ServerCertificateSelector` callback. The callback is invoked once per connection to allow the app to inspect the host name and select the appropriate certificate:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenAnyIP(5005, listenOptions =>
    {
        listenOptions.UseHttps(httpsOptions =>
        {
            var localhostCert = CertificateLoader.LoadFromStoreCert(
                "localhost", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var exampleCert = CertificateLoader.LoadFromStoreCert(
                "example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var subExampleCert = CertificateLoader.LoadFromStoreCert(
```

```
                "sub.example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var certs = new Dictionary<string, X509Certificate2>(
                StringComparer.OrdinalIgnoreCase)
            {
                ["localhost"] = localhostCert,
                ["example.com"] = exampleCert,
                ["sub.example.com"] = subExampleCert
            };

            httpsOptions.ServerCertificateSelector = (connectionContext,
name) =>
            {
                if (name is not null && certs.TryGetValue(name, out var
cert))
                {
                    return cert;
                }

                return exampleCert;
            };
        });
    });
});
```

SNI with `ServerOptionsSelectionCallback`

Kestrel supports additional dynamic TLS configuration via the `ServerOptionsSelectionCallback` callback. The callback is invoked once per connection to allow the app to inspect the host name and select the appropriate certificate and TLS configuration. Default certificates and `ConfigureHttpsDefaults` aren't used with this callback.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenAnyIP(5005, listenOptions =>
    {
        listenOptions.UseHttps(httpsOptions =>
        {
            var localhostCert = CertificateLoader.LoadFromStoreCert(
                "localhost", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var exampleCert = CertificateLoader.LoadFromStoreCert(
                "example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
```

```
            listenOptions.UseHttps((stream, clientHelloInfo, state,
cancellationToken) =>
            {
                if (string.Equals(clientHelloInfo.ServerName, "localhost",
                    StringComparison.OrdinalIgnoreCase))
                {
                    return new ValueTask<SslServerAuthenticationOptions>(
                        new SslServerAuthenticationOptions
                        {
                            ServerCertificate = localhostCert,
                            // Different TLS requirements for this host
                            ClientCertificateRequired = true
                        });
                }

                return new ValueTask<SslServerAuthenticationOptions>(
                    new SslServerAuthenticationOptions
                    {
                        ServerCertificate = exampleCert
                    });
            }, state: null!);
        });
    });
});
```

SNI with `TlsHandshakeCallbackOptions`

Kestrel supports additional dynamic TLS configuration via the `TlsHandshakeCallbackOptions.OnConnection` callback. The callback is invoked once per connection to allow the app to inspect the host name and select the appropriate certificate, TLS configuration, and other server options. Default certificates and `ConfigureHttpsDefaults` aren't used with this callback.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenAnyIP(5005, listenOptions =>
    {
        listenOptions.UseHttps(httpsOptions =>
        {
            var localhostCert = CertificateLoader.LoadFromStoreCert(
                "localhost", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var exampleCert = CertificateLoader.LoadFromStoreCert(
                "example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
```

```
            listenOptions.UseHttps(new TlsHandshakeCallbackOptions
            {
                OnConnection = context =>
                {
                    if (string.Equals(context.ClientHelloInfo.ServerName,
"localhost",
                        StringComparison.OrdinalIgnoreCase))
                    {
                        // Different TLS requirements for this host
                        context.AllowDelayedClientCertificateNegotation =
true;

                        return new ValueTask<SslServerAuthenticationOptions>
(
                            new SslServerAuthenticationOptions
                            {
                                ServerCertificate = localhostCert
                            });
                    }

                    return new ValueTask<SslServerAuthenticationOptions>(
                        new SslServerAuthenticationOptions
                        {
                            ServerCertificate = exampleCert
                        });
                }
            });
        });
    });
});
```

Configure HTTP protocols

Kestrel supports all commonly used HTTP versions. Endpoints can be configured to support different HTTP versions using the HttpProtocols enum, which specifies available HTTP version options.

TLS is required to support more than one HTTP version. The TLS Application-Layer Protocol Negotiation (ALPN) handshake is used to negotiate the connection protocol between the client and the server when an endpoint supports multiple protocols.

Expand table

`HttpProtocols` value	Connection protocol permitted
`Http1`	HTTP/1.1 only. Can be used with or without TLS.
`Http2`	HTTP/2 only. May be used without TLS only if the client supports a Prior Knowledge mode.

`HttpProtocols` value	Connection protocol permitted
`Http3`	HTTP/3 only. Requires TLS. The client may need to be configured to use HTTP/3 only.
`Http1AndHttp2`	HTTP/1.1 and HTTP/2. HTTP/2 requires the client to select HTTP/2 in the TLS Application-Layer Protocol Negotiation (ALPN) handshake; otherwise, the connection defaults to HTTP/1.1.
`Http1AndHttp2AndHttp3`	HTTP/1.1, HTTP/2 and HTTP/3. The first client request normally uses HTTP/1.1 or HTTP/2, and the alt-svc response header prompts the client to upgrade to HTTP/3. HTTP/2 and HTTP/3 requires TLS; otherwise, the connection defaults to HTTP/1.1.

The default protocol value for an endpoint is `HttpProtocols.Http1AndHttp2`.

TLS restrictions for HTTP/2:

- TLS version 1.2 or later
- Renegotiation disabled
- Compression disabled
- Minimum ephemeral key exchange sizes:
  - Elliptic curve Diffie-Hellman (ECDHE) [RFC4492]: 224 bits minimum
  - Finite field Diffie-Hellman (DHE) [`TLS12`]: 2048 bits minimum
- Cipher suite not prohibited.

`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` [`TLS-ECDHE`] with the P-256 elliptic curve [`FIPS186`] is supported by default.

Configure HTTP protocols in appsettings.json

The following `appsettings.json` example establishes the HTTP/1.1 connection protocol for a specific endpoint:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "HttpsDefaultCert": {
        "Url": "https://localhost:5001",
        "Protocols": "Http1"
      }
    }
  }
}
```

A default protocol can be configured in the `Kestrel:EndpointDefaults` section. The following `appsettings.json` example establishes HTTP/1.1 as the default connection protocol for all endpoints:

JSON

```
{
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1"
    }
  }
}
```

Protocols specified in code override values set by configuration.

Configure HTTP protocols in code

ListenOptions.Protocols is used to specify protocols with the HttpProtocols enum.

The following example configures an endpoint for HTTP/1.1, HTTP/2, and HTTP/3 connections on port 8000. Connections are secured by TLS with a supplied certificate:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
        listenOptions.Protocols = HttpProtocols.Http1AndHttp2AndHttp3;
    });
});
```

See also

- Kestrel web server in ASP.NET Core
- Configure options for the ASP.NET Core Kestrel web server

Configure options for the ASP.NET Core Kestrel web server

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.

The Kestrel web server has constraint configuration options that are especially useful in Internet-facing deployments. To configure Kestrel configuration options, call ConfigureKestrel in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    // ...
});
```

Set constraints on the KestrelServerOptions.Limits property. This property holds an instance of the KestrelServerLimits class.

General limits

Keep-alive timeout

KeepAliveTimeout gets or sets the keep-alive timeout:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
});
```

This timeout is not enforced when a debugger is attached to the Kestrel process.

Maximum client connections

MaxConcurrentConnections gets or sets the maximum number of open connections:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
});
```

MaxConcurrentUpgradedConnections gets or sets the maximum number of open, upgraded connections:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
});
```

An upgraded connection is one that has been switched from HTTP to another protocol, such as WebSockets. After a connection is upgraded, it isn't counted against the `MaxConcurrentConnections` limit.

Maximum request body size

MaxRequestBodySize gets or sets the maximum allowed size of any request body in bytes.

The recommended approach to override the limit in an ASP.NET Core MVC app is to use the RequestSizeLimitAttribute attribute on an action method:

```
C#

[RequestSizeLimit(100_000_000)]
public IActionResult Get()
```

The following example configures `MaxRequestBodySize` for all requests:

```
C#
```

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxRequestBodySize = 100_000_000;
});
```

The following example configures `MaxRequestBodySize` for a specific request using IHttpMaxRequestBodySizeFeature in a custom middleware:

C#

```
app.Use(async (context, next) =>
{
    var httpMaxRequestBodySizeFeature =
context.Features.Get<IHttpMaxRequestBodySizeFeature>();

    if (httpMaxRequestBodySizeFeature is not null)
        httpMaxRequestBodySizeFeature.MaxRequestBodySize = 10 * 1024;

    // ...

    await next(context);
});
```

If the app attempts to configure the limit on a request after it starts to read the request, an exception is thrown. Ue the IHttpMaxRequestBodySizeFeature.IsReadOnly property to check if it's safe to set the `MaxRequestBodySize` property.

When an app runs out-of-process behind the ASP.NET Core Module, IIS sets the limit and Kestrel's request body size limit is disabled.

Minimum request body data rate

Kestrel checks every second if data is arriving at the specified rate in bytes/second. If the rate drops below the minimum, the connection is timed out. The grace period is the amount of time Kestrel allows the client to increase its send rate up to the minimum. The rate isn't checked during that time. The grace period helps avoid dropping connections that are initially sending data at a slow rate because of TCP slow-start. A minimum rate also applies to the response.

MinRequestBodyDataRate gets or sets the request body minimum data rate in bytes/second. MinResponseDataRate gets or sets the response minimum data rate in bytes/second.

The following example configures `MinRequestBodyDataRate` and `MinResponseDataRate` for all requests:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MinRequestBodyDataRate = new MinDataRate(
        bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate = new MinDataRate(
        bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
});
```

The following example configures `MinRequestBodyDataRate` and `MinResponseDataRate` for a specific request using IHttpMinRequestBodyDataRateFeature and IHttpMinResponseDataRateFeature in a custom middleware:

C#

```
app.Use(async (context, next) =>
{
    var httpMinRequestBodyDataRateFeature = context.Features
        .Get<IHttpMinRequestBodyDataRateFeature>();

    if (httpMinRequestBodyDataRateFeature is not null)
    {
        httpMinRequestBodyDataRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    var httpMinResponseDataRateFeature = context.Features
        .Get<IHttpMinResponseDataRateFeature>();

    if (httpMinResponseDataRateFeature is not null)
    {
        httpMinResponseDataRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    // ...

    await next(context);
});
```

`IHttpMinResponseDataRateFeature` isn't present in HttpContext.Features for HTTP/2 requests. Modifying rate limits on a per-request basis isn't generally supported for HTTP/2 because of the protocol's support for request multiplexing. However, `IHttpMinRequestBodyDataRateFeature` is still present in `HttpContext.Features` for HTTP/2 requests, because the read rate limit can still be *disabled entirely* on a per-request basis by setting IHttpMinResponseDataRateFeature.MinDataRate to `null`, even for an HTTP/2 request. Attempts to read `IHttpMinRequestBodyDataRateFeature.MinDataRate` or attempts

to set it to a value other than `null` result in a NotSupportedException for HTTP/2 requests.

Server-wide rate limits configured via KestrelServerOptions.Limits still apply to both HTTP/1.x and HTTP/2 connections.

Request headers timeout

RequestHeadersTimeout gets or sets the maximum amount of time the server spends receiving request headers:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(1);
});
```

This timeout is not enforced when a debugger is attached to the Kestrel process.

HTTP/2 limits

The limits in this section are set on KestrelServerLimits.Http2.

Maximum streams per connection

MaxStreamsPerConnection limits the number of concurrent request streams per HTTP/2 connection. Excess streams are refused:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.MaxStreamsPerConnection = 100;
});
```

Header table size

HeaderTableSize limits the size of the header compression tables, in octets, the HPACK encoder and decoder on the server can use. The HPACK decoder decompresses HTTP headers for HTTP/2 connections:

```
```

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.HeaderTableSize = 4096;
});
```

Maximum frame size

MaxFrameSize indicates the size of the largest frame payload that is allowed to be received, in octets:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.MaxFrameSize = 16_384;
});
```

Maximum request header size

MaxRequestHeaderFieldSize indicates the size of the maximum allowed size of a request header field sequence. This limit applies to both name and value sequences in their compressed and uncompressed representations:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.MaxRequestHeaderFieldSize = 8192;
});
```

Initial connection window size

InitialConnectionWindowSize indicates how much request body data the server is willing to receive and buffer at a time aggregated across all requests (streams) per connection:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.InitialConnectionWindowSize = 131_072;
});
```

Requests are also limited by InitialStreamWindowSize.

Initial stream window size

InitialStreamWindowSize indicates how much request body data the server is willing to receive and buffer at a time per stream:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.InitialStreamWindowSize = 98_304;
});
```

Requests are also limited by InitialConnectionWindowSize.

HTTP/2 keep alive ping configuration

Kestrel can be configured to send HTTP/2 pings to connected clients. HTTP/2 pings serve multiple purposes:

- Keep idle connections alive. Some clients and proxy servers close connections that are idle. HTTP/2 pings are considered as activity on a connection and prevent the connection from being closed as idle.
- Close unhealthy connections. Connections where the client doesn't respond to the keep alive ping in the configured time are closed by the server.

There are two configuration options related to HTTP/2 keep alive pings:

- KeepAlivePingDelay is a TimeSpan that configures the ping interval. The server sends a keep alive ping to the client if it doesn't receive any frames for this period of time. Keep alive pings are disabled when this option is set to TimeSpan.MaxValue.
- KeepAlivePingTimeout is a TimeSpan that configures the ping timeout. If the server doesn't receive any frames, such as a response ping, during this timeout then the connection is closed. Keep alive timeout is disabled when this option is set to TimeSpan.MaxValue.

The following example sets `KeepAlivePingDelay` and `KeepAlivePingTimeout`:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
```

```
    serverOptions.Limits.Http2.KeepAlivePingDelay =
TimeSpan.FromSeconds(30);
    serverOptions.Limits.Http2.KeepAlivePingTimeout =
TimeSpan.FromMinutes(1);
});
```

Other options

Synchronous I/O

AllowSynchronousIO controls whether synchronous I/O is allowed for the request and response.

> **⚠ Warning**
>
> A large number of blocking synchronous I/O operations can lead to thread pool starvation, which makes the app unresponsive. Only enable `AllowSynchronousIO` when using a library that doesn't support asynchronous I/O.

The following example enables synchronous I/O:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.AllowSynchronousIO = true;
});
```

For information about other Kestrel options and limits, see:

- KestrelServerOptions
- KestrelServerLimits
- ListenOptions

Behavior with debugger attached

Certain timeouts and rate limits aren't enforced when a debugger is attached to a Kestrel process. For more information, see Behavior with debugger attached.

Diagnostics in Kestrel

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 Sourabh Shirhatti

This article provides guidance for gathering diagnostics from Kestrel to help troubleshoot issues. Topics covered include:

- **Logging**: Structured logs written to .NET Core logging. ILogger is used by app frameworks to write logs, and by users for their own logging in an app.
- **Metrics**: Representation of data measures over intervals of time, for example, requests per second. Metrics are emitted using `EventCounter` and can be observed using the dotnet-counters command line tool or with Application Insights.
- **DiagnosticSource**: `DiagnosticSource` is a mechanism for production-time logging with rich data payloads for consumption within the process. Unlike logging, which assumes data will leave the process and expects serializable data, `DiagnosticSource` works well with complex data.

Logging

Like most components in ASP.NET Core, Kestrel uses `Microsoft.Extensions.Logging` to emit log information. Kestrel employs the use of multiple categories which allows you to be selective on which logs you listen to.

Expand table

Logging Category Name	Logging Events
`Microsoft.AspNetCore.Server.Kestrel`	`ApplicationError`, `ConnectionHeadResponseBodyWrite`, `ApplicationNeverCompleted`, `RequestBodyStart`, `RequestBodyDone`, `RequestBodyNotEntirelyRead`,

Logging Category Name	Logging Events
	`RequestBodyDrainTimedOut`, `ResponseMinimumDataRateNotSatisfied`, `InvalidResponseHeaderRemoved`, `HeartbeatSlow`
`Microsoft.AspNetCore.Server.Kestrel.BadRequests`	`ConnectionBadRequest`, `RequestProcessingError`, `RequestBodyMinimumDataRateNotSatisfied`
`Microsoft.AspNetCore.Server.Kestrel.Connections`	`ConnectionAccepted`, `ConnectionStart`, `ConnectionStop`, `ConnectionPause`, `ConnectionResume`, `ConnectionKeepAlive`, `ConnectionRejected`, `ConnectionDisconnect`, `NotAllConnectionsClosedGracefully`, `NotAllConnectionsAborted`, `ApplicationAbortedConnection`
`Microsoft.AspNetCore.Server.Kestrel.Http2`	`Http2ConnectionError`, `Http2ConnectionClosing`, `Http2ConnectionClosed`, `Http2StreamError`, `Http2StreamResetAbort`, `HPackDecodingError`, `HPackEncodingError`, `Http2FrameReceived`, `Http2FrameSending`, `Http2MaxConcurrentStreamsReached`
`Microsoft.AspNetCore.Server.Kestrel.Http3`	`Http3ConnectionError`, `Http3ConnectionClosing`, `Http3ConnectionClosed`, `Http3StreamAbort`, `Http3FrameReceived`, `Http3FrameSending`

Connection logging

Kestrel also supports the ability to emit `Debug` level logs for byte-level communication and can be enabled on a per-endpoint basis. To enable connection logging, see configure endpoints for Kestrel

Metrics

Metrics is a representation of data measures over intervals of time, for example, requests per second. Metrics data allows observation of the state of an app at a high-level. Kestrel metrics are emitted using `EventCounter`.

> ⓘ **Note**

The `connections-per-second` and `tls-handshakes-per-second` counters are named incorrectly. The counters:

- Do ***not*** always contain the number of new connections or TLS handshakes per second
- Display the number of new connection or TLS handshakes in the last update interval as requested as the consumer of Events via the `EventCounterIntervalSec` argument in the `filterPayload` to `KestrelEventSource`.

We **recommend** consumers of these counters scale the metric value based on the `DisplayRateTimeScale` of one second.

Expand table

Name	Display Name	Description
`connections-per-second`	Connection Rate	The number of new incoming connections per update interval
`total-connections`	Total Connections	The total number of connections
`tls-handshakes-per-second`	TLS Handshake Rate	The number of new TLS handshakes per update interval
`total-tls-handshakes`	Total TLS Handshakes	The total number of TLS handshakes
`current-tls-handshakes`	Current TLS Handshakes	The number of TLS handshakes in process
`failed-tls-handshakes`	Failed TLS Handshakes	The total number of failed TLS handshakes
`current-connections`	Current Connections	The total number of connections, including idle connections
`connection-queue-length`	Connection Queue Length	The total number connections queued to the thread pool. In a healthy system at steady state, this number should always be close to zero
`request-queue-length`	Request Queue Length	The total number requests queued to the thread pool. In a healthy system at steady state, this number should

Name	Display Name	Description
		always be close to zero. This metric is unlike the IIS/Http.Sys request queue and cannot be compared
`current-upgraded-requests`	Current Upgraded Requests (WebSockets)	The number of active WebSocket requests

DiagnosticSource

Kestrel emits a `DiagnosticSource` event for HTTP requests rejected at server layer such as malformed requests and protocols violations. As such, these requests never make it into the hosting layer of ASP.NET Core.

Kestrel emits these events with the `Microsoft.AspNetCore.Server.Kestrel.BadRequest` event name and an `IFeatureCollection` as the object payload. The underlying exception can be retrieved by accessing the `IBadRequestExceptionFeature` on the feature collection.

Resolving these events is a two-step process. An observer for `DiagnosticListener` must be created:

C#

```
class BadRequestEventListener : IObserver<KeyValuePair<string, object>>,
IDisposable
{
    private readonly IDisposable _subscription;
    private readonly Action<IBadRequestExceptionFeature> _callback;

    public BadRequestEventListener(DiagnosticListener diagnosticListener,
Action<IBadRequestExceptionFeature> callback)
    {
        _subscription = diagnosticListener.Subscribe(this!, IsEnabled);
        _callback = callback;
    }
    private static readonly Predicate<string> IsEnabled = (provider) =>
provider switch
    {
        "Microsoft.AspNetCore.Server.Kestrel.BadRequest" => true,
        _ => false
    };
    public void OnNext(KeyValuePair<string, object> pair)
    {
        if (pair.Value is IFeatureCollection featureCollection)
        {
            var badRequestFeature =
```

```
featureCollection.Get<IBadRequestExceptionFeature>();

            if (badRequestFeature is not null)
            {
                _callback(badRequestFeature);
            }
        }
    }
    public void OnError(Exception error) { }
    public void OnCompleted() { }
    public virtual void Dispose() => _subscription.Dispose();
}
```

Subscribe to the ASP.NET Core `DiagnosticListener` with the observer. In this example, we create a callback that logs the underlying exception.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
var diagnosticSource = app.Services.GetRequiredService<DiagnosticListener>
();
using var badRequestListener = new BadRequestEventListener(diagnosticSource,
(badRequestExceptionFeature) =>
{
    app.Logger.LogError(badRequestExceptionFeature.Error, "Bad request
received");
});
app.MapGet("/", () => "Hello world");
app.Run();
```

Behavior with debugger attached

Certain timeouts and rate limits aren't enforced when a debugger is attached to a Kestrel process. For more information, see Behavior with debugger attached.

Use HTTP/2 with the ASP.NET Core Kestrel web server

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.

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

- Operating system
  - Windows Server 2016/Windows 10 or later‡
  - Linux with OpenSSL 1.0.2 or later (for example, Ubuntu 16.04 or later)
  - macOS 10.15 or later
- Target framework: .NET Core 2.2 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

‡Kestrel has limited support for HTTP/2 on Windows Server 2012 R2 and Windows 8.1. Support is limited because the list of supported TLS cipher suites available on these operating systems is limited. A certificate generated using an Elliptic Curve Digital Signature Algorithm (ECDSA) may be required to secure TLS connections.

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

Starting with .NET Core 3.0, HTTP/2 is enabled by default. For more information on configuration, see the Kestrel HTTP/2 limits and ListenOptions.Protocols sections.

Advanced HTTP/2 features

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

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 HTTP/3 with the ASP.NET Core Kestrel web server

Article • 03/14/2024

> **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.

HTTP/3 is an approved standard and the third major version of HTTP. This article discusses the requirements for HTTP/3. HTTP/3 is fully supported in ASP.NET Core 7.0 and later.

> **Important**
>
> Apps configured to take advantage of HTTP/3 should be designed to also support HTTP/1.1 and HTTP/2.

HTTP/3 requirements

HTTP/3 has different requirements depending on the operating system. If the platform that Kestrel is running on doesn't have all the requirements for HTTP/3, then it's disabled, and Kestrel will fall back to other HTTP protocols.

Windows

- Windows 11 Build 22000 or later OR Windows Server 2022.
- TLS 1.3 or later connection.

Linux

- `libmsquic` package installed.

`libmsquic` is published via Microsoft's official Linux package repository at `packages.microsoft.com`. To install this package:

1. Add the `packages.microsoft.com` repository. See Linux Software Repository for Microsoft Products for instructions.
2. Install the `libmsquic` package using the distro's package manager. For example, `apt install libmsquic=1.9*` on Ubuntu.

Note: .NET 6 is only compatible with the 1.9.x versions of libmsquic. Libmsquic 2.x is not compatible due to breaking changes. Libmsquic receives updates to 1.9.x when needed to incorporate security fixes.

macOS

HTTP/3 isn't currently supported on macOS and may be available in a future release.

Getting started

HTTP/3 is not enabled by default. Add configuration to `Program.cs` to enable HTTP/3.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, options) =>
{
    options.ListenAnyIP(5001, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http1AndHttp2AndHttp3;
        listenOptions.UseHttps();
    });
});
```

The preceding code configures port 5001 to:

- Use HTTP/3 alongside HTTP/1.1 and HTTP/2 by specifying `HttpProtocols.Http1AndHttp2AndHttp3`.
- Enable HTTPS with `UseHttps`. HTTP/3 requires HTTPS.

Because not all routers, firewalls, and proxies properly support HTTP/3, HTTP/3 should be configured together with HTTP/1.1 and HTTP/2. This can be done by specifying HttpProtocols.Http1AndHttp2AndHttp3 as an endpoint's supported protocols.

For more information, see Configure endpoints for the ASP.NET Core Kestrel web server.

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. Kestrel automatically adds the `alt-svc` header if HTTP/3 is enabled.

Localhost testing

- Browsers don't allow self-signed certificates on HTTP/3, such as the Kestrel development certificate.
- `HttpClient` can be used for localhost/loopback testing in .NET 6 or later. Extra configuration is required when using `HttpClient` to make an HTTP/3 request:
  - Set HttpRequestMessage.Version to 3.0, or
  - Set HttpRequestMessage.VersionPolicy to HttpVersionPolicy.RequestVersionOrHigher.

HTTP/3 benefits

HTTP/3 uses the same semantics as HTTP/1.1 and HTTP/2: the same request methods, status codes, and message fields apply to all versions. The differences are in the underlying transport. Both HTTP/1.1 and HTTP/2 use TCP as their transport. HTTP/3 uses a new transport technology developed alongside HTTP/3 called QUIC.

HTTP/3 and QUIC have a number of benefits compared to HTTP/1.1 and HTTP/2:

- Faster response time of the first request. QUIC and HTTP/3 negotiates the connection in fewer round-trips between the client and the server. The first request reaches the server faster.
- Improved experience when there is connection packet loss. HTTP/2 multiplexes multiple requests via one TCP connection. Packet loss on the connection affects all requests. This problem is called "head-of-line blocking". Because QUIC provides native multiplexing, lost packets only impact the requests where data has been lost.
- Supports transitioning between networks. This feature is useful for mobile devices where it is common to switch between WIFI and cellular networks as a mobile device changes location. Currently, HTTP/1.1 and HTTP/2 connections fail with an error when switching networks. An app or web browsers must retry any failed HTTP requests. HTTP/3 allows the app or web browser to seamlessly continue when a network changes. Kestrel doesn't support network transitions in .NET 8. It may be available in a future release.

Connection middleware

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.

Kestrel supports connection middleware. Connection middleware is software that is assembled into a connection pipeline and runs when Kestrel receives a new connection. Each component:

- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline.

Connection delegates are used to build the connection pipeline. Connection delegates are configured with the ListenOptions.Use method.

Connection middleware is different from ASP.NET Core Middleware. Connection middleware runs per-connection instead of per-request.

Connection logging

Connection logging is connection middleware that is included with ASP.NET Core. Call UseConnectionLogging to emit Debug level logs for byte-level communication on a connection.

Connection logging is helpful for troubleshooting problems in low-level communication, such as during TLS encryption and behind proxies. If `UseConnectionLogging` is placed before `UseHttps`, encrypted traffic is logged. If `UseConnectionLogging` is placed after `UseHttps`, decrypted traffic is logged.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
```

```
    {
        listenOptions.UseConnectionLogging();
    });
});
```

Create custom connection middleware

The following example shows a custom connection middleware that can filter TLS handshakes on a per-connection basis for specific ciphers if necessary. The middleware throws NotSupportedException for any cipher algorithm that the app doesn't support. Alternatively, define and compare ITlsHandshakeFeature.CipherAlgorithm to a list of acceptable cipher suites.

No encryption is used with a CipherAlgorithmType.Null cipher algorithm.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");

        listenOptions.Use((context, next) =>
        {
            var tlsFeature = context.Features.Get<ITlsHandshakeFeature>()!;

            if (tlsFeature.CipherAlgorithm == CipherAlgorithmType.Null)
            {
                throw new NotSupportedException(
                    $"Prohibited cipher: {tlsFeature.CipherAlgorithm}");
            }

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

See also

- Configure endpoints for the ASP.NET Core Kestrel web server

When to use Kestrel with a reverse proxy

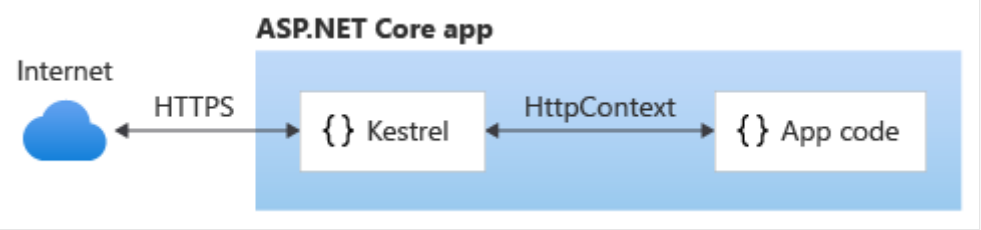
Article • 09/27/2024

> **ⓘ 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.

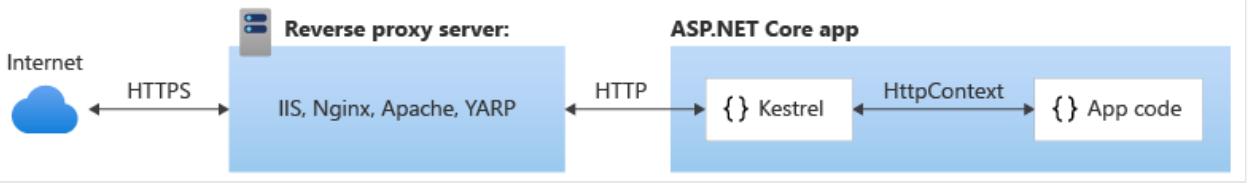
Kestrel can be used by itself or with a *reverse proxy server*. A reverse proxy server receives HTTP requests from the network and forwards them to Kestrel. Examples of a reverse proxy server include:

- Internet Information Services (IIS)
- Nginx
- Apache
- YARP: Yet Another Reverse Proxy

Kestrel used as an edge (Internet-facing) web server:



Kestrel used in a reverse proxy configuration:



Either configuration, with or without a reverse proxy server, is a supported hosting configuration.

When Kestrel is used as an edge server without a reverse proxy server, sharing of the same IP address and port among multiple processes is unsupported. When Kestrel is configured to listen on a port, Kestrel handles all traffic for that port regardless of

requests' `Host` headers. A reverse proxy that can share ports can forward requests to Kestrel on a unique IP and port.

Even if a reverse proxy server isn't required, using a reverse proxy server might be a good choice.

A reverse proxy:

- Can limit the exposed public surface area of the apps that it hosts.
- Provides an additional layer of configuration and defense-in-depth cybersecurity.
- Might integrate better with existing infrastructure.
- Simplifies load balancing and secure communication (HTTPS) configuration. Only the reverse proxy server requires the X.509 certificate for the public domain(s). That server can communicate with the app's servers on the internal network using plain HTTP or HTTPS with locally managed certificates. Internal HTTPS increases security but adds significant overhead.

> **⚠ Warning**
>
> Hosting in a reverse proxy configuration requires **host filtering**.

Additional resources

Configure ASP.NET Core to work with proxy servers and load balancers

Host filtering with ASP.NET Core Kestrel web server

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.

While Kestrel supports configuration based on prefixes such as `http://example.com:5000`, Kestrel largely ignores the host name. Host `localhost` is a special case used for binding to loopback addresses. Any host other than an explicit IP address binds to all public IP addresses. `Host` headers aren't validated.

As a workaround, use Host Filtering Middleware. The middleware is added by CreateDefaultBuilder, which calls AddHostFiltering:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Host Filtering Middleware is disabled by default. To enable the middleware, define an `AllowedHosts` key in `appsettings.json`/`appsettings.{Environment}.json`. The value is a semicolon-delimited list of host names without port numbers:

`appsettings.json`:

JSON

```
{
  "AllowedHosts": "example.com;localhost"
}
```

> ⓘ Note
>
> Forwarded Headers Middleware also has an AllowedHosts option. Forwarded Headers Middleware and Host Filtering Middleware have similar functionality for different scenarios. Setting `AllowedHosts` with Forwarded Headers Middleware is appropriate when the `Host` header isn't preserved while forwarding requests with a reverse proxy server or load balancer. Setting `AllowedHosts` with Host Filtering Middleware is appropriate when Kestrel is used as a public-facing edge server or when the `Host` header is directly forwarded.
>
> For more information on Forwarded Headers Middleware, see Configure ASP.NET Core to work with proxy servers and load balancers.

Request draining with ASP.NET Core Kestrel web server

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.

Opening HTTP connections is time consuming. For HTTPS, it's also resource intensive. Therefore, Kestrel tries to reuse connections per the HTTP/1.1 protocol. A request body must be fully consumed to allow the connection to be reused. The app doesn't always consume the request body, such as HTTP POST requests where the server returns a redirect or 404 response. In the HTTP POST redirect case:

- The client may already have sent part of the POST data.
- The server writes the 301 response.
- The connection can't be used for a new request until the POST data from the previous request body has been fully read.
- Kestrel tries to drain the request body. Draining the request body means reading and discarding the data without processing it.

The draining process makes a tradeoff between allowing the connection to be reused and the time it takes to drain any remaining data:

- Draining has a timeout of five seconds, which isn't configurable.
- If all of the data specified by the `Content-Length` or `Transfer-Encoding` header hasn't been read before the timeout, the connection is closed.

Sometimes you may want to terminate the request immediately, before or after writing the response. For example, clients may have restrictive data caps. Limiting uploaded data might be a priority. In such cases to terminate a request, call HttpContext.Abort from a controller, Razor Page, or middleware.

There are caveats to calling `Abort`:

- Creating new connections can be slow and expensive.

- There's no guarantee that the client has read the response before the connection closes.
- Calling `Abort` should be rare and reserved for severe error cases, not common errors.
  - Only call `Abort` when a specific problem needs to be solved. For example, call `Abort` if malicious clients are trying to POST data or when there's a bug in client code that causes large or several requests.
  - Don't call `Abort` for common error situations, such as HTTP 404 (Not Found).

Calling HttpResponse.CompleteAsync before calling `Abort` ensures that the server has completed writing the response. However, client behavior isn't predictable and they may not read the response before the connection is aborted.

This process is different for HTTP/2 because the protocol supports aborting individual request streams without closing the connection. The five-second drain timeout doesn't apply. If there's any unread request body data after completing a response, then the server sends an HTTP/2 RST frame. Additional request body data frames are ignored.

If possible, it's better for clients to use the Expect: 100-continue request header and wait for the server to respond before starting to send the request body. That gives the client an opportunity to examine the response and abort before sending unneeded data.

Host ASP.NET Core on Windows with 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.

Internet Information Services (IIS) is a flexible, secure and manageable Web Server for hosting web apps, including ASP.NET Core.

Supported platforms

The following operating systems are supported:

- Windows 7 or later
- Windows Server 2012 R2 or later

Apps published for 32-bit (x86) or 64-bit (x64) deployment are supported. Deploy a 32-bit app with a 32-bit (x86) .NET Core SDK unless the app:

- Requires the larger virtual memory address space available to a 64-bit app.
- Requires the larger IIS stack size.
- Has 64-bit native dependencies.

Install the ASP.NET Core Module/Hosting Bundle

Download the latest installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

For more details instructions on how to install the ASP.NET Core Module, or installing different versions, see Install the .NET Core Hosting Bundle.

To download previous versions of the hosting bundle, see this GitHub issue.

Get started

For getting started with hosting a website on IIS, see our getting started guide.

For getting started with hosting a website on Azure App Services, see our deploying to Azure App Service guide.

Configuration

For configuration guidance, see Advanced configuration.

Deployment resources for IIS administrators

- IIS documentation
- Getting Started with the IIS Manager in IIS
- .NET Core application deployment
- ASP.NET Core Module (ANCM) for IIS
- ASP.NET Core directory structure
- IIS modules with ASP.NET Core
- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core

Overlapped recycle

In general, we recommend using a pattern like blue-green deployments for zero-downtime deployments. Features like Overlapped Recycle help, but don't guarantee that you can do a zero-downtime deployment. For more information, see this GitHub issue.

Optional client certificates

For information on apps that must protect a subset of the app with a certificate, see Optional client certificates.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Overview of ASP.NET Core
- The Official Microsoft IIS Site

- Windows Server technical content library
- HTTP/2 on IIS
- Transform web.config

ASP.NET Core Module (ANCM) for IIS

Article • 09/27/2024

> **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**.

The ASP.NET Core Module (ANCM) is a native IIS module that plugs into the IIS pipeline, allowing ASP.NET Core applications to work with IIS. Run ASP.NET Core apps with IIS by either:

- Hosting an ASP.NET Core app inside of the IIS worker process (`w3wp.exe`), called the in-process hosting model.
- Forwarding web requests to a backend ASP.NET Core app running the Kestrel server, called the out-of-process hosting model.

There are trade-offs between each of the hosting models. By default, the in-process hosting model is used due to better performance and diagnostics.

For more information and configuration guidance, see the following topics:

- Web server implementations in ASP.NET Core

Install ASP.NET Core Module (ANCM)

The ASP.NET Core Module (ANCM) is installed with the .NET Core Runtime from the .NET Core Hosting Bundle. The ASP.NET Core Module is forward and backward compatible with in-support releases of .NET.

Breaking changes and security advisories are reported on the Announcements repo. Announcements can be limited to a specific version by selecting a **Label** filter.

Download the installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

For more information, including installing an earlier version of the module, see Hosting Bundle.

For a tutorial experience on publishing an ASP.NET Core app to an IIS server, see Publish an ASP.NET Core app to IIS.

Additional resources

- Host ASP.NET Core on Windows with IIS
- Deploy ASP.NET Core apps to Azure App Service
- ASP.NET Core Module reference source [default branch (main)] : Use the **Branch** drop down list to select a specific release (for example, `release/3.1`).
- IIS modules with ASP.NET Core

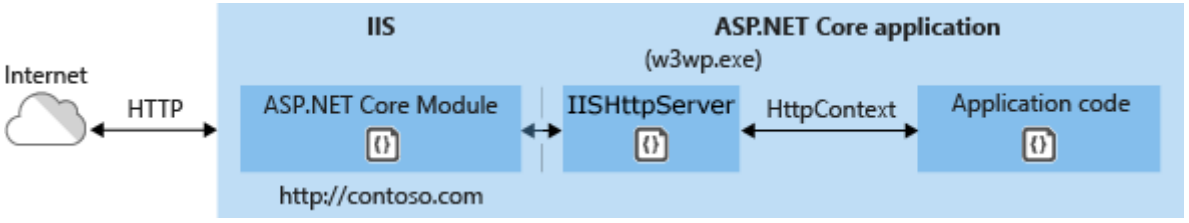
In-process hosting with IIS and ASP.NET Core

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.

In-process hosting runs an ASP.NET Core app in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter, a network interface that returns outgoing network traffic back to the same machine.

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted in-process:



Enable in-process hosting

Since ASP.NET Core 3.0, in-process hosting has been enabled by default for all app deployed to IIS.

To explicitly configure an app for in-process hosting, set the value of the `<AspNetCoreHostingModel>` property to `InProcess` in the project file (`.csproj`):

XML

```xml
<PropertyGroup>
  <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

General architecture

The general flow of a request is as follows:

1. A request arrives from the web to the kernel-mode HTTP.sys driver.
2. The driver routes the native request to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS).
3. The ASP.NET Core Module receives the native request and passes it to IIS HTTP Server (`IISHttpServer`). IIS HTTP Server is an in-process server implementation for IIS that converts the request from native to managed.

After the IIS HTTP Server processes the request:

1. The request is sent to the ASP.NET Core middleware pipeline.
2. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic.
3. The app's response is passed back to IIS through IIS HTTP Server.
4. IIS sends the response to the client that initiated the request.

`CreateDefaultBuilder` adds an IServer instance by calling the UseIIS method to boot the CoreCLR and host the app inside of the IIS worker process (`w3wp.exe` or `iisexpress.exe`). Performance tests indicate that hosting a .NET Core app in-process delivers significantly higher request throughput compared to hosting the app out-of-process and proxying requests to Kestrel.

Apps published as a single file executable can't be loaded by the in-process hosting model.

Application configuration

To configure IIS options, include a service configuration for IISServerOptions in `Program.cs`. The following example disables AutomaticAuthentication:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Server.IIS;
using Microsoft.EntityFrameworkCore;
using RPauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
```

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.Configure<IISServerOptions>(options =>
{
    options.AutomaticAuthentication = false;
});

builder.Services.AddTransient<IClaimsTransformation, MyClaimsTransformation>
();
builder.Services.AddAuthentication(IISServerDefaults.AuthenticationScheme);

builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

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

app.MapRazorPages();

app.Run();
```

Expand table

Option	Default	Setting
`AutomaticAuthentication`	`true`	If `true`, IIS Server sets the `HttpContext.User` authenticated by Windows Authentication. If `false`, the server only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication

Option	Default	Setting
		must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see Windows Authentication.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.
`AllowSynchronousIO`	`false`	Whether synchronous I/O is allowed for the `HttpContext.Request` and the `HttpContext.Response`.
`MaxRequestBodySize`	`30000000`	Gets or sets the max request body size for the `HttpRequest`. Note that IIS itself has the limit `maxAllowedContentLength` which will be processed before the `MaxRequestBodySize` set in the `IISServerOptions`. Changing the `MaxRequestBodySize` won't affect the `maxAllowedContentLength`. To increase `maxAllowedContentLength`, add an entry in the `web.config` to set `maxAllowedContentLength` to a higher value. For more details, see Configuration.

Differences between in-process and out-of-process hosting

The following characteristics apply when hosting in-process:

- IIS HTTP Server (`IISHttpServer`) is used instead of Kestrel server. For in-process, CreateDefaultBuilder calls UseIIS to:
  - Register the `IISHttpServer`.
  - Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
  - Configure the host to capture startup errors.
- The requestTimeout attribute doesn't apply to in-process hosting.
- Sharing an app pool among apps isn't supported. Use one app pool per app.
- The architecture (bitness) of the app and installed runtime (x64 or x86) must match the architecture of the app pool. For example, apps published for 32-bit (x86) must have 32-bit enabled for their IIS Application Pools. For more information, see the Create the IIS site section.
- Client disconnects are detected. The HttpContext.RequestAborted cancellation token is cancelled when the client disconnects.

- When hosting in-process, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. When transforming claims with an IClaimsTransformation implementation, call AddAuthentication to add authentication services:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Server.IIS;
using Microsoft.EntityFrameworkCore;
using RPauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.Configure<IISServerOptions>(options =>
{
    options.AutomaticAuthentication = false;
});

builder.Services.AddTransient<IClaimsTransformation, MyClaimsTransformation>
();
builder.Services.AddAuthentication(IISServerDefaults.AuthenticationScheme);

builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
```

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

app.MapRazorPages();

app.Run();
```

- Web Package (single-file) deployments aren't supported.

Get timing information

See Get detailed timing information with IHttpSysRequestTimingFeature.

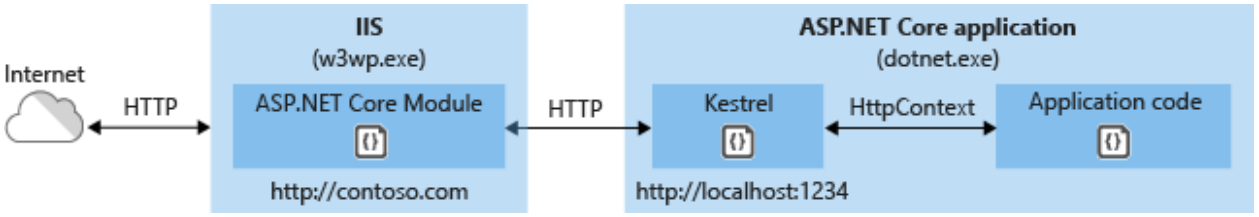
Out-of-process hosting with IIS and ASP.NET Core

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.

Because ASP.NET Core apps run in a process separate from the IIS worker process, the ASP.NET Core Module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the Windows Process Activation Service (WAS).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted out-of-process:



1. Requests arrive from the web to the kernel-mode HTTP.sys driver.
2. The driver routes the requests to IIS on the website's configured port. The configured port is usually 80 (HTTP) or 443 (HTTPS).
3. The module forwards the requests to Kestrel on a random port for the app. The random port isn't 80 or 443.

The ASP.NET Core Module specifies the port via an environment variable at startup. The UseIISIntegration extension configures the server to listen on `http://localhost:{PORT}`. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding. Requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is forwarded into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and

passes it on as an `HttpContext` instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which forwards it back to the HTTP client that initiated the request.

For ASP.NET Core Module configuration guidance, see ASP.NET Core Module (ANCM) for IIS.

For more information on hosting, see Host in ASP.NET Core.

Application configuration

Enable the IISIntegration components

When building a host in `CreateHostBuilder` (`Program.cs`), call CreateDefaultBuilder to enable IIS integration:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        ...
```

For more information on `CreateDefaultBuilder`, see .NET Generic Host in ASP.NET Core.

Out-of-process hosting model

To configure IIS options, include a service configuration for IISOptions in ConfigureServices. The following example prevents the app from populating `HttpContext.Connection.ClientCertificate`:

C#

```
services.Configure<IISOptions>(options =>
{
    options.ForwardClientCertificate = false;
});
```

Expand table

Option	Default	Setting
`AutomaticAuthentication`	`true`	If `true`, IIS Integration Middleware sets the `HttpContext.User` authenticated by Windows

Option	Default	Setting
		Authentication. If `false`, the middleware only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see the Windows Authentication topic.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.
`ForwardClientCertificate`	`true`	If `true` and the `MS-ASPNETCORE-CLIENTCERT` request header is present, the `HttpContext.Connection.ClientCertificate` is populated.

Proxy server and load balancer scenarios

The IIS Integration Middleware and the ASP.NET Core Module are configured to forward the:

- Scheme (HTTP/HTTPS).
- Remote IP address where the request originated.

The IIS Integration Middleware configures Forwarded Headers Middleware.

Additional configuration might be required for apps hosted behind additional proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Out-of-process hosting model

To configure an app for out-of-process hosting, set the value of the `<AspNetCoreHostingModel>` property to `OutOfProcess` in the project file (`.csproj`):

XML

```
<PropertyGroup>
  <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

In-process hosting is set with `InProcess`, which is the default value.

The value of `<AspNetCoreHostingModel>` is case insensitive, so `inprocess` and `outofprocess` are valid values.

Kestrel server is used instead of IIS HTTP Server (`IISHttpServer`).

For out-of-process, CreateDefaultBuilder calls UseIISIntegration to:

- Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
- Configure the host to capture startup errors.

Process name

`Process.GetCurrentProcess().ProcessName` reports `w3wp`/`iisexpress` (in-process) or `dotnet` (out-of-process).

Many native modules, such as Windows Authentication, remain active. To learn more about IIS modules active with the ASP.NET Core Module, see IIS modules with ASP.NET Core.

The ASP.NET Core Module can also:

- Set environment variables for the worker process.
- Log stdout output to file storage for troubleshooting startup issues.
- Forward Windows authentication tokens.

The .NET Core Hosting Bundle

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**.

The .NET Core Hosting bundle is an installer for the .NET Core Runtime and the ASP.NET Core Module. The bundle allows ASP.NET Core apps to run with IIS.

Install the .NET Core Hosting Bundle

> **Important**
>
> If the Hosting Bundle is installed before IIS, the bundle installation must be repaired. Run the Hosting Bundle installer again after installing IIS.
>
> If the Hosting Bundle is installed after installing the 64-bit (x64) version of .NET Core, SDKs might appear to be missing (**No .NET Core SDKs were detected**). To resolve the problem, see **Troubleshoot and debug ASP.NET Core projects**.

Breaking changes and security advisories are reported on the Announcements repo. Announcements can be limited to a specific version by selecting a **Label** filter.

Direct download

Download the installer using the following links:

- Current version:.NET Core Hosting Bundle installer (direct download)
- Previous and pre-release versions

Visual C++ Redistributable Requirement

On older versions of Windows, for example Windows Server 2012 R2, install the Visual Studio C++ 2015, 2017, 2019 Redistributable. Otherwise, a confusing error message in the Windows Event Log reports that `The data is the error.`

Current x64 VS C++ redistributable Current x86 VS C++ redistributable

Earlier versions of the installer

To obtain an earlier version of the installer:

1. Navigate to the Download .NET Core page.
2. Select the desired .NET Core version.
3. In the **Run apps - Runtime** column, find the row of the .NET Core runtime version desired.
4. Download the installer using the **Hosting Bundle** link.

> ⚠ **Warning**
>
> Some installers contain release versions that have reached their end of life (EOL) and are no longer supported by Microsoft. For more information, see the **support policy**.
>
> The **ASP.NET Core Module** is forward and backward compatible with **in-support releases of .NET**.

Options

1. The following parameters are available when running the installer from an administrator command shell:

   - `OPT_NO_ANCM=1`: Skip installing the ASP.NET Core Module.
   - `OPT_NO_RUNTIME=1`: Skip installing the .NET Core runtime. Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_SHAREDFX=1`: Skip installing the ASP.NET Shared Framework (ASP.NET runtime). Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_X86=1`: Skip installing x86 runtimes. Use this parameter when you know that you won't be hosting 32-bit apps. If there's any chance that you will host both 32-bit and 64-bit apps in the future, don't use this parameter and install both runtimes.

- `OPT_NO_SHARED_CONFIG_CHECK=1`: Disable the check for using an IIS Shared Configuration when the shared configuration (`applicationHost.config`) is on the same machine as the IIS installation. *Only available for ASP.NET Core 2.2 or later Hosting Bundler installers.* For more information, see Advanced configuration.

> **Note**
>
> For information on IIS Shared Configuration, see **ASP.NET Core Module with IIS Shared Configuration**.

> **Note**
>
> When running the Hosting Bundle installer with options set, the value for each option is saved in the registry. Subsequent installs from the same Major.Minor version band use the same options, unless another set of options is explicitly passed from the command line. If the first install of the hosting bundle has no options passed, each option gets a default value of `0` written in to the registry. A value of `0` implies that the option is off, meaning the user is not opting out of the given component.

Restart IIS

After the Hosting Bundle is installed, a manual IIS restart may be required. For example, the `dotnet` CLI tooling (command) might not exist on the PATH for running IIS worker processes.

To manually restart IIS, stop the Windows Process Activation Service (WAS) and then restart the World Wide Web Publishing Service (W3SVC) and any dependent services. Execute the following commands in an elevated command shell:

Console

```
net stop was /y
net start w3svc
```

Module version and Hosting Bundle installer logs

To determine the version of the installed ASP.NET Core Module:

1. On the hosting system, navigate to `%PROGRAMFILES%\IIS\Asp.Net Core Module\V2`.
2. Locate the `aspnetcorev2.dll` file.
3. Right-click the file and select **Properties** from the contextual menu.
4. Select the **Details** tab. The **File version** and **Product version** represent the installed version of the module.

The Hosting Bundle installer logs for the module are found at `C:\Users\%UserName%\AppData\Local\Temp`. The file is named `dd_DotNetCoreWinSvrHosting__{TIMESTAMP}_000_AspNetCoreModule_x64.log`, where the placeholder `{TIMESTAMP}` is the timestamp of the file.

`web.config` file

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.

The `web.config` is a file that is read by IIS and the ASP.NET Core Module to configure an app hosted with IIS.

`web.config` file location

In order to set up the ASP.NET Core Module correctly, the `web.config` file must be present at the content root path (typically the app base path) of the deployed app. This is the same location as the website physical path provided to IIS. The `web.config` file is required at the root of the app to enable the publishing of multiple apps using Web Deploy.

Sensitive files exist on the app's physical path, such as `{ASSEMBLY}.runtimeconfig.json`, `{ASSEMBLY}.xml` (XML Documentation comments), and `{ASSEMBLY}.deps.json`, where the placeholder `{ASSEMBLY}` is the assembly name. When the `web.config` file is present and the site starts normally, IIS doesn't serve these sensitive files if they're requested. If the `web.config` file is missing, incorrectly named, or unable to configure the site for normal startup, IIS may serve sensitive files publicly.

The `web.config` file must be present in the deployment at all times, correctly named, and able to configure the site for normal start up. Never remove the `web.config` file from a production deployment.

If a `web.config` file isn't present in the project, the file is created with the correct `processPath` and `arguments` to configure the ASP.NET Core Module and moved to published output.

If a `web.config` file is present in the project, the file is transformed with the correct `processPath` and `arguments` to configure the ASP.NET Core Module and moved to

published output. The transformation doesn't modify IIS configuration settings in the file.

The `web.config` file may provide additional IIS configuration settings that control active IIS modules. For information on IIS modules that are capable of processing requests with ASP.NET Core apps, see the IIS modules topic.

Creating, transforming, and publishing the `web.config` file is handled by an MSBuild target (`_TransformWebConfig`) when the project is published. This target is present in the Web SDK targets (`Microsoft.NET.Sdk.Web`). The SDK is set at the top of the project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

To prevent the Web SDK from transforming the `web.config` file, use the `<IsTransformWebConfigDisabled>` property in the project file:

XML

```
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

When disabling the Web SDK from transforming the file, the `processPath` and `arguments` should be manually set by the developer. For more information, see ASP.NET Core Module (ANCM) for IIS.

Configuration of ASP.NET Core Module with `web.config`

The ASP.NET Core Module is configured with the `aspNetCore` section of the `system.webServer` node in the site's `web.config` file.

The following `web.config` file is published for a framework-dependent deployment and configures the ASP.NET Core Module to handle site requests:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
```

```
      <handlers>
        <add name="aspNetCore" path="*" verb="*"
modules="AspNetCoreModuleV2" resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet"
                  arguments=".\MyApp.dll"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The following `web.config` is published for a self-contained deployment:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*"
modules="AspNetCoreModuleV2" resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath=".\MyApp.exe"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the app.

When an app is deployed to Azure App Service, the `stdoutLogFile` path is set to `\\?\%home%\LogFiles\stdout`. The path saves stdout logs to the `LogFiles` folder, which is a location automatically created by the service.

For information on IIS sub-application configuration, see Advanced configuration.

Attributes of the `aspNetCore` element

Expand table

Attribute	Description	Default
`arguments`	Optional string attribute. Arguments to the executable specified in `processPath`.	
`disableStartUpErrorPage`	Optional Boolean attribute. If true, the **502.5 - Process Failure** page is suppressed, and the 502 status code page configured in the `web.config` takes precedence.	`false`
`forwardWindowsAuthToken`	Optional Boolean attribute. If true, the token is forwarded to the child process listening on `%ASPNETCORE_PORT%` as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It's the responsibility of that process to call CloseHandle on this token per request.	`true`
`hostingModel`	Optional string attribute. Specifies the hosting model as in-process (`InProcess`/`inprocess`) or out-of-process (`OutOfProcess`/`outofprocess`).	`OutOfProcess`/`outofprocess` when not present
`processesPerApplication`	Optional integer attribute. Specifies the number of instances of the process specified in the `processPath` setting that can be spun up per app. †For in-process hosting, the value is limited to `1`. Setting `processesPerApplication` is discouraged. This attribute will be removed in a future release.	Default: `1` Min: `1` Max: `100`†
`processPath`	Required string attribute. Path to the executable that launches a process listening for HTTP requests. Relative paths are supported. If the path begins with `.`, the path is	

Attribute	Description	Default
	considered to be relative to the site root.	
`rapidFailsPerMinute`	Optional integer attribute. Specifies the number of times the process specified in `processPath` is allowed to crash per minute. If this limit is exceeded, the module stops launching the process for the remainder of the minute. Not supported with in-process hosting.	Default: `10` Min: `0` Max: `100`
`requestTimeout`	Optional timespan attribute. Specifies the duration for which the ASP.NET Core Module waits for a response from the process listening on %ASPNETCORE_PORT%. In versions of the ASP.NET Core Module that shipped with the release of ASP.NET Core 2.1 or later, the `requestTimeout` is specified in hours, minutes, and seconds. Doesn't apply to in-process hosting. For in-process hosting, the module waits for the app to process the request. Valid values for minutes and seconds segments of the string are in the range 0-59. Use of `60` in the value for minutes or seconds results in a *500 - Internal Server Error*.	Default: `00:02:00` Min: `00:00:00` Max: `360:00:00`
`shutdownTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to gracefully shutdown when the `app_offline.htm` file is detected.	Default: `10` Min: `0` Max: `600`
`startupTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to start a process listening on the port. If this	Default: `120` Min: `0` Max: `3600`

Attribute	Description	Default
	time limit is exceeded, the module kills the process. When hosting *in-process*: The process is **not** restarted and does **not** use the `rapidFailsPerMinute` setting. When hosting *out-of-process*: The module attempts to relaunch the process when it receives a new request and continues to attempt to restart the process on subsequent incoming requests unless the app fails to start `rapidFailsPerMinute` number of times in the last rolling minute. A value of 0 (zero) is **not** considered an infinite timeout.	
`stdoutLogEnabled`	Optional Boolean attribute. If true, `stdout` and `stderr` for the process specified in `processPath` are redirected to the file specified in `stdoutLogFile`.	`false`
`stdoutLogFile`	Optional string attribute. Specifies the relative or absolute file path for which `stdout` and `stderr` from the process specified in `processPath` are logged. Relative paths are relative to the root of the site. Any path starting with `.` are relative to the site root and all other paths are treated as absolute paths. Any folders provided in the path are created by the module when the log file is created. Using underscore delimiters, a timestamp, process ID, and file extension (`.log`) are added to the last segment of the `stdoutLogFile` path. If `.\logs\stdout` is supplied as a value, an example stdout log is saved as `stdout_20180205194132_1934.log` in the `logs` folder when saved on February 5, 2018 at 19:41:32 with a process ID of 1934.	`aspnetcore-stdout`

Set environment variables

Environment variables can be specified for the process in the `processPath` attribute. Specify an environment variable with the `<environmentVariable>` child element of an `<environmentVariables>` collection element. Environment variables set in this section take precedence over system environment variables.

The following example sets two environment variables in `web.config`. `ASPNETCORE_ENVIRONMENT` configures the app's environment to `Development`. A developer may temporarily set this value in the `web.config` file in order to force the Developer Exception Page to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that reads the value on startup to form a path for loading the app's configuration file.

XML

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="inprocess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development"
/>
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

> **Note**
>
> An alternative to setting the environment directly in `web.config` is to include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in `web.config` when the project is published:
>
> XML
>
> ```
> <PropertyGroup>
>   <EnvironmentName>Development</EnvironmentName>
> </PropertyGroup>
> ```

> **Warning**

> Only set the `ASPNETCORE_ENVIRONMENT` environment variable to `Development` on staging and testing servers that aren't accessible to untrusted networks, such as the Internet.

Configuration of IIS with `web.config`

IIS configuration is influenced by the `<system.webServer>` section of `web.config` for IIS scenarios that are functional for ASP.NET Core apps with the ASP.NET Core Module. For example, IIS configuration is functional for dynamic compression. If IIS is configured at the server level to use dynamic compression, the `<urlCompression>` element in the app's `web.config` file can disable it for an ASP.NET Core app.

For more information, see the following topics:

- Configuration reference for <system.webServer>
- ASP.NET Core Module (ANCM) for IIS
- IIS modules with ASP.NET Core

To set environment variables for individual apps running in isolated app pools (supported for IIS 10.0 or later), see the *`AppCmd.exe` command* section of the Environment Variables <environmentVariables> topic in the IIS reference documentation.

Configuration sections of `web.config`

Configuration sections of ASP.NET 4.x apps in `web.config` aren't used by ASP.NET Core apps for configuration:

- `<system.web>`
- `<appSettings>`
- `<connectionStrings>`
- `<location>`

ASP.NET Core apps are configured using other configuration providers. For more information, see Configuration.

Transform web.config

If you need to transform `web.config` on publish, see Transform web.config. You might need to transform `web.config` on publish to set environment variables based on the

configuration, profile, or environment.

Additional resources

- IIS <system.webServer>
- IIS modules with ASP.NET Core
- Transform web.config

Development-time IIS support in Visual Studio for ASP.NET Core

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 Sourabh Shirhatti

This article describes Visual Studio support for debugging ASP.NET Core apps running with IIS on Windows Server. This topic walks through enabling this scenario and setting up a project.

Prerequisites

- Visual Studio for Windows
- **ASP.NET and web development** workload
- **.NET Core cross-platform development** workload
- X.509 security certificate (for HTTPS support)

Enable IIS

1. In Windows, navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Select the **Internet Information Services** checkbox. Select **OK**.

The IIS installation may require a system restart.

Configure IIS

IIS must have a website configured with the following:

- **Host name**: Typically, the **Default Web Site** is used with a **Host name** of `localhost`. However, any valid IIS website with a unique host name works.

- **Site Binding**
  - For apps that require HTTPS, create a binding to port 443 with a certificate. Typically, the **IIS Express Development Certificate** is used, but any valid certificate works.
  - For apps that use HTTP, confirm the existence of a binding to port 80 or create a binding to port 80 for a new site.
  - Use a single binding for either HTTP or HTTPS. **Binding to both HTTP and HTTPS ports simultaneously isn't supported.**

Configure the project

HTTPS redirection

For a new project that requires HTTPS, select the checkbox to **Configure for HTTPS** in the **Create a new ASP.NET Core Web Application** window. Selecting the checkbox adds HTTPS Redirection and HSTS Middleware to the app when it's created.

For an existing project that requires HTTPS, use HTTPS Redirection and HSTS Middleware in `Startup.Configure`. For more information, see Enforce HTTPS in ASP.NET Core.

For a project that uses HTTP, HTTPS Redirection and HSTS Middleware aren't added to the app. No app configuration is required.

IIS launch profile

Create a new launch profile to add development-time IIS support:

1. Right-click the project in **Solution Explorer**. Select **Properties**. Open the **Debug** tab.

2. For **Profile**, select the **New** button. Name the profile "IIS" in the popup window. Select **OK** to create the profile.

3. For the **Launch** setting, select **IIS** from the list.

4. Select the checkbox for **Launch browser** and provide the endpoint URL.

   When the app requires HTTPS, use an HTTPS endpoint (`https://`). For HTTP, use an HTTP (`http://`) endpoint.

Provide the same host name and port as the IIS configuration specified earlier uses, typically `localhost`.

Provide the name of the app at the end of the URL.

For example, `https://localhost/WebApplication1` (HTTPS) or `http://localhost/WebApplication1` (HTTP) are valid endpoint URLs.

5. In the **Environment variables** section, select the **Add** button. Provide an environment variable with a **Name** of `ASPNETCORE_ENVIRONMENT` and a **Value** of `Development`.

6. In the **Web Server Settings** area, set the **App URL** to the same value used for the **Launch browser** endpoint URL.

7. For the **Hosting Model** setting in Visual Studio 2019 or later, select **Default** to use the hosting model used by the project. If the project sets the `<AspNetCoreHostingModel>` property in its project file, the value of the property (`InProcess` or `OutOfProcess`) is used. If the property isn't present, the default hosting model of the app is used, which is in-process. If the app requires an explicit hosting model setting different from the app's normal hosting model, set the **Hosting Model** to either `In Process` or `Out Of Process` as needed.

8. Save the profile.

When not using Visual Studio, manually add a launch profile to the launchSettings.json file in the *Properties* folder. The following example configures the profile to use the HTTPS protocol:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iis": {
      "applicationUrl": "https://localhost/WebApplication1",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS": {
      "commandName": "IIS",
      "launchBrowser": true,
      "launchUrl": "https://localhost/WebApplication1",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
```

```
      }
    }
  }
}
```

Confirm that the `applicationUrl` and `launchUrl` endpoints match and use the same protocol as the IIS binding configuration, either HTTP or HTTPS.

Run the project

Run Visual Studio as an administrator:

- Confirm that the build configuration drop-down list is set to **Debug**.
- Set the Start Debugging button to the **IIS** profile and select the button to start the app.

Visual Studio may prompt a restart if not running as an administrator. If prompted, restart Visual Studio.

If an untrusted development certificate is used, the browser may require you to create an exception for the untrusted certificate.

> ⓘ **Note**
>
> Debugging a Release build configuration with Just My Code and compiler optimizations results in a degraded experience. For example, break points aren't hit.

Additional resources

- Getting Started with the IIS Manager in IIS
- Enforce HTTPS in ASP.NET Core

IIS modules with ASP.NET Core

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.

Some of the native IIS modules and all of the IIS managed modules aren't able to process requests for ASP.NET Core apps. In many cases, ASP.NET Core offers an alternative to the scenarios addressed by IIS native and managed modules.

Native modules

The table indicates native IIS modules that are functional with ASP.NET Core apps and the ASP.NET Core Module.

Expand table

Module	Functional with ASP.NET Core apps	ASP.NET Core Option
Anonymous Authentication `AnonymousAuthenticationModule`	Yes	
Basic Authentication `BasicAuthenticationModule`	Yes	
Client Certification Mapping Authentication `CertificateMappingAuthenticationModule`	Yes	
CGI `CgiModule`	No	
Configuration Validation `ConfigurationValidationModule`	Yes	
HTTP Errors `CustomErrorModule`	No	Status Code Pages Middleware

Module	Functional with ASP.NET Core apps	ASP.NET Core Option
Custom Logging `CustomLoggingModule`	Yes	
Default Document `DefaultDocumentModule`	No	Default Files Middleware
Digest Authentication `DigestAuthenticationModule`	Yes	
Directory Browsing `DirectoryListingModule`	No	Directory Browsing Middleware
Dynamic Compression `DynamicCompressionModule`	Yes	Response Compression Middleware
Failed Requests Tracing `FailedRequestsTracingModule`	Yes	ASP.NET Core Logging
File Caching `FileCacheModule`	No	Response Caching Middleware
HTTP Caching `HttpCacheModule`	No	Response Caching Middleware
HTTP Logging `HttpLoggingModule`	Yes	ASP.NET Core Logging
HTTP Redirection `HttpRedirectionModule`	Yes	URL Rewriting Middleware
HTTP Tracing `TracingModule`	Yes	
IIS Client Certificate Mapping Authentication `IISCertificateMappingAuthenticationModule`	Yes	
IP and Domain Restrictions `IpRestrictionModule`	Yes	
ISAPI Filters `IsapiFilterModule`	Yes	Middleware
ISAPI `IsapiModule`	Yes	Middleware
Protocol Support `ProtocolSupportModule`	Yes	

Module	Functional with ASP.NET Core apps	ASP.NET Core Option
Request Filtering `RequestFilteringModule`	Yes	URL Rewriting Middleware IRule
Request Monitor `RequestMonitorModule`	Yes	
URL Rewriting† `RewriteModule`	Yes	URL Rewriting Middleware
Server-Side Includes `ServerSideIncludeModule`	No	
Static Compression `StaticCompressionModule`	No	Response Compression Middleware
Static Content `StaticFileModule`	No	Static File Middleware
Token Caching `TokenCacheModule`	Yes	
URI Caching `UriCacheModule`	Yes	
URL Authorization `UrlAuthorizationModule`	Yes	ASP.NET Core Identity
WebDav `WebDAV`	No	
Windows Authentication `WindowsAuthenticationModule`	Yes	

†The URL Rewrite Module's `isFile` and `isDirectory` match types don't work with ASP.NET Core apps due to the changes in directory structure.

Managed modules

Managed modules are *not* functional with hosted ASP.NET Core apps when the app pool's .NET CLR version is set to **No Managed Code**. ASP.NET Core offers middleware alternatives in several cases.

Expand table

Module	ASP.NET Core Option
AnonymousIdentification	
DefaultAuthentication	
FileAuthorization	
FormsAuthentication	Cookie Authentication Middleware
OutputCache	Response Caching Middleware
Profile	
RoleManager	
ScriptModule-4.0	
Session	Session Middleware
UrlAuthorization	
UrlMappingsModule	URL Rewriting Middleware
UrlRoutingModule-4.0	ASP.NET Core Identity
WindowsAuthentication	

IIS Manager application changes

When using IIS Manager to configure settings, the *web.config* file of the app is changed. If deploying an app and including *web.config*, any changes made with IIS Manager are overwritten by the deployed *web.config* file. If changes are made to the server's *web.config* file, copy the updated *web.config* file on the server to the local project immediately.

Disabling IIS modules

If an IIS module is configured at the server level that must be disabled for an app, an addition to the app's *web.config* file can disable the module. Either leave the module in place and deactivate it using a configuration setting (if available) or remove the module from the app.

Module deactivation

Many modules offer a configuration setting that allows them to be disabled without removing the module from the app. This is the simplest and quickest way to deactivate a module. For example, the HTTP Redirection Module can be disabled with the `<httpRedirect>` element in *web.config*:

XML

```
<configuration>
  <system.webServer>
    <httpRedirect enabled="false" />
  </system.webServer>
</configuration>
```

For more information on disabling modules with configuration settings, follow the links in the *Child Elements* section of IIS <system.webServer>.

Module removal

If opting to remove a module with a setting in *web.config*, unlock the module and unlock the `<modules>` section of *web.config* first:

1. Unlock the module at the server level. Select the IIS server in the IIS Manager **Connections** sidebar. Open the **Modules** in the **IIS** area. Select the module in the list. In the **Actions** sidebar on the right, select **Unlock**. If the action entry for the module appears as **Lock**, the module is already unlocked, and no action is required. Unlock as many modules as you plan to remove from *web.config* later.

2. Deploy the app without a `<modules>` section in *web.config*. If an app is deployed with a *web.config* containing the `<modules>` section without having unlocked the section first in the IIS Manager, the Configuration Manager throws an exception when attempting to unlock the section. Therefore, deploy the app without a `<modules>` section.

3. Unlock the `<modules>` section of *web.config*. In the **Connections** sidebar, select the website in **Sites**. In the **Management** area, open the **Configuration Editor**. Use the navigation controls to select the `system.webServer/modules` section. In the **Actions** sidebar on the right, select to **Unlock** the section. If the action entry for the module section appears as **Lock Section**, the module section is already unlocked, and no action is required.

4. Add a `<modules>` section to the app's local *web.config* file with a `<remove>` element to remove the module from the app. Add multiple `<remove>` elements to remove multiple modules. If *web.config* changes are made on the server, immediately make

the same changes to the project's *web.config* file locally. Removing a module using this approach doesn't affect the use of the module with other apps on the server.

```
XML

<configuration>
 <system.webServer>
   <modules>
     <remove name="MODULE_NAME" />
   </modules>
 </system.webServer>
</configuration>
```

In order to add or remove modules for IIS Express using *web.config*, modify *applicationHost.config* to unlock the `<modules>` section:

1. Open *{APPLICATION ROOT}\.vs\config\applicationhost.config*.
2. Locate the `<section>` element for IIS modules and change `overrideModeDefault` from `Deny` to `Allow`:

   ```
   XML

   <section name="modules"
            allowDefinition="MachineToApplication"
            overrideModeDefault="Allow" />
   ```

3. Locate the `<location path="" overrideMode="Allow"><system.webServer><modules>` section. For any modules that you wish to remove, set `lockItem` from `true` to `false`. In the following example, the CGI Module is unlocked:

   ```
   XML

   <add name="CgiModule" lockItem="false" />
   ```

4. After the `<modules>` section and individual modules are unlocked, you're free to add or remove IIS modules using the app's *web.config* file for running the app on IIS Express.

An IIS module can also be removed with *Appcmd.exe*. Provide the `MODULE_NAME` and `APPLICATION_NAME` in the command:

```
Console
```

```
Appcmd.exe delete module MODULE_NAME /app.name:APPLICATION_NAME
```

For example, remove the `DynamicCompressionModule` from the Default Web Site:

Console

```
%windir%\system32\inetsrv\appcmd.exe delete module DynamicCompressionModule
/app.name:"Default Web Site"
```

Minimum module configuration

The only modules required to run an ASP.NET Core app are the Anonymous Authentication Module and the ASP.NET Core Module.

The URI Caching Module (`UriCacheModule`) allows IIS to cache website configuration at the URL level. Without this module, IIS must read and parse configuration on every request, even when the same URL is repeatedly requested. Parsing the configuration every request results in a significant performance penalty. *Although the URI Caching Module isn't strictly required for a hosted ASP.NET Core app to run, we recommend that the URI Caching Module be enabled for all ASP.NET Core deployments.*

The HTTP Caching Module (`HttpCacheModule`) implements the IIS output cache and also the logic for caching items in the HTTP.sys cache. Without this module, content is no longer cached in kernel mode, and cache profiles are ignored. Removing the HTTP Caching Module usually has adverse effects on performance and resource usage. *Although the HTTP Caching Module isn't strictly required for a hosted ASP.NET Core app to run, we recommend that the HTTP Caching Module be enabled for all ASP.NET Core deployments.*

Additional resources

- Introduction to IIS Architectures: Modules in IIS
- IIS Modules Overview
- Customizing IIS 7.0 Roles and Modules
- IIS <system.webServer>

IIS log creation and redirection

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.

The ASP.NET Core Module redirects stdout and stderr console output to disk if the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element are set. Any folders in the `stdoutLogFile` path are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\{APP POOL NAME}` to provide write permission, where the placeholder `{APP POOL NAME}` is the app pool name).

Logs aren't rotated, unless process recycling/restart occurs. It's the responsibility of the hoster to limit the disk space the logs consume.

Using the stdout log is only recommended for troubleshooting app startup issues when hosting on IIS or when using development-time support for IIS with Visual Studio, not while debugging locally and running the app with IIS Express.

Don't use the stdout log for general app logging purposes. For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

A timestamp and file extension are added automatically when the log file is created. The log file name is composed by appending the timestamp, process ID, and file extension (`.log`) to the last segment of the `stdoutLogFile` path (typically `stdout`) delimited by underscores. If the `stdoutLogFile` path ends with `stdout`, a log for an app with a PID of 1934 created on 2/5/2018 at 19:42:32 has the file name `stdout_20180205194132_1934.log`.

If `stdoutLogEnabled` is false, errors that occur on app startup are captured and emitted to the event log up to 30 KB. After startup, all additional logs are discarded.

The following sample `aspNetCore` element configures stdout logging at the relative path `.\log\`. Confirm that the AppPool user identity has permission to write to the path

provided.

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile=".\logs\stdout"
    hostingModel="inprocess">
</aspNetCore>
```

When publishing an app for Azure App Service deployment, the Web SDK sets the `stdoutLogFile` value to `\\?\%home%\LogFiles\stdout`. The `%home` environment variable is predefined for apps hosted by Azure App Service.

To create logging filter rules, see the Apply log filter rules in code section of the ASP.NET Core logging documentation.

For more information on path formats, see File path formats on Windows systems.

Enhanced diagnostic logs

The ASP.NET Core Module is configurable to provide enhanced diagnostics logs. Add the `<handlerSettings>` element to the `<aspNetCore>` element in `web.config`. Setting the `debugLevel` to `TRACE` exposes a higher fidelity of diagnostic information:

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="debugFile" value=".\logs\aspnetcore-debug.log" />
    <handlerSetting name="debugLevel" value="FILE,TRACE" />
  </handlerSettings>
</aspNetCore>
```

Any folders in the path (`logs` in the preceding example) are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\{APP POOL NAME}` to provide write permission, where the placeholder `{APP POOL NAME}` is the app pool name).

Debug level (`debugLevel`) values can include both the level and the location.

Levels (in order from least to most verbose):

- ERROR
- WARNING
- INFO
- TRACE

Locations (multiple locations are permitted):

- CONSOLE
- EVENTLOG
- FILE

The handler settings can also be provided via environment variables:

- `ASPNETCORE_MODULE_DEBUG_FILE`: Path to the debug log file. (Default: `aspnetcore-debug.log`)
- `ASPNETCORE_MODULE_DEBUG`: Debug level setting.

> ⚠ **Warning**
>
> Do **not** leave debug logging enabled in the deployment for longer than required to troubleshoot an issue. The size of the log isn't limited. Leaving the debug log enabled can exhaust the available disk space and crash the server or app service.

See Configuration of ASP.NET Core Module with web.config for an example of the `aspNetCore` element in the `web.config` file.

Troubleshoot ASP.NET Core on Azure App Service and IIS

Article • 09/27/2024

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, the ASP.NET Core project default server is Kestrel. Visual studio can be configured to use IIS Express. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally with IIS Express can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:

    - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
    - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
    - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.

3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module (ANCM) for IIS.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

This error also may occur when the .NET Core Hosting Bundle isn't installed or is corrupted. Installing or repairing the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express) may fix the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

An unknown error occurred loading ASP.NET Core Module components. Take one of the following actions:

- Contact Microsoft Support (select **Developer Tools** then **ASP.NET Core**).
- Ask a question on Stack Overflow.
- File an issue on our GitHub repository.

500.30 In-Process Startup Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core CLR in-process, but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

Common failure conditions:

- The app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine.
- Using Azure Key Vault, lack of permissions to the Key Vault. Check the access policies in the targeted Key Vault to ensure that the correct permissions are granted.

500.31 ANCM Failed to Find Native Dependencies

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core runtime in-process, but it fails to start. The most common cause of this startup failure is when the `Microsoft.NETCore.App` or `Microsoft.AspNetCore.App` runtime isn't installed. If the app is deployed to target ASP.NET Core 3.0 and that version doesn't exist on the machine, this error occurs. An example error message follows:

```
```

```
The specified framework 'Microsoft.NETCore.App', version '3.0.0' was not
found.
  - The following frameworks were found:
      2.2.1 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview5-27626-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27713-13 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27714-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27723-08 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
```

The error message lists all the installed .NET Core versions and the version requested by the app. To fix this error, either:

- Install the appropriate version of .NET Core on the machine.
- Change the app to target a version of .NET Core that's present on the machine.
- Publish the app as a self-contained deployment.

When running in development (the `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development`), the specific error is written to the HTTP response. The cause of a process startup failure is also found in the Application Event Log.

500.32 ANCM Failed to Load dll

The worker process fails. The app doesn't start.

The most common cause for this error is that the app is published for an incompatible processor architecture. If the worker process is running as a 32-bit app and the app was published to target 64-bit, this error occurs.

To fix this error, either:

- Republish the app for the same processor architecture as the worker process.
- Publish the app as a framework-dependent deployment.

500.33 ANCM Request Handler Load Failure

The worker process fails. The app doesn't start.

The app didn't reference the `Microsoft.AspNetCore.App` framework. Only apps targeting the `Microsoft.AspNetCore.App` framework can be hosted by the ASP.NET Core Module.

To fix this error, confirm that the app is targeting the `Microsoft.AspNetCore.App` framework. Check the `.runtimeconfig.json` to verify the framework targeted by the app.

500.34 ANCM Mixed Hosting Models Not Supported

The worker process can't run both an in-process app and an out-of-process app in the same process.

To fix this error, run apps in separate IIS application pools.

500.35 ANCM Multiple In-Process Applications in same Process

The worker process can't run multiple in-process apps in the same process.

To fix this error, run apps in separate IIS application pools.

500.36 ANCM Out-Of-Process Handler Load Failure

The out-of-process request handler, *aspnetcorev2_outofprocess.dll*, isn't next to the *aspnetcorev2.dll* file. This indicates a corrupted installation of the ASP.NET Core Module.

To fix this error, repair the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express).

500.37 ANCM Failed to Start Within Startup Time Limit

ANCM failed to start within the provided startup time limit. By default, the timeout is 120 seconds.

This error can occur when starting a large number of apps on the same machine. Check for CPU/Memory usage spikes on the server during startup. You may need to stagger the startup process of multiple apps.

500.38 ANCM Application DLL Not Found

ANCM failed to locate the application DLL, which should be next to the executable.

This error occurs when hosting an app packaged as a single-file executable using the in-process hosting model. The in-process model requires that the ANCM load the .NET Core app into the existing IIS process. This scenario isn't supported by the single-file

deployment model. Use **one** of the following approaches in the app's project file to fix this error:

1. Disable single-file publishing by setting the `PublishSingleFile` MSBuild property to `false`.
2. Switch to the out-of-process hosting model by setting the `AspNetCoreHostingModel` MSBuild property to `OutOfProcess`.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.

2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:

    - If deploying a 32-bit (x86) app, set the value to `True`.
    - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Failed to start application (ErrorCode '0x800701b1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x800701b1'.
```

The app failed to start because a Windows Service failed to load.

One common service that needs to be enabled is the "null" service. The following command enables the `null` Windows Service:

Windows Command Prompt

```
sc.exe start null
```

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

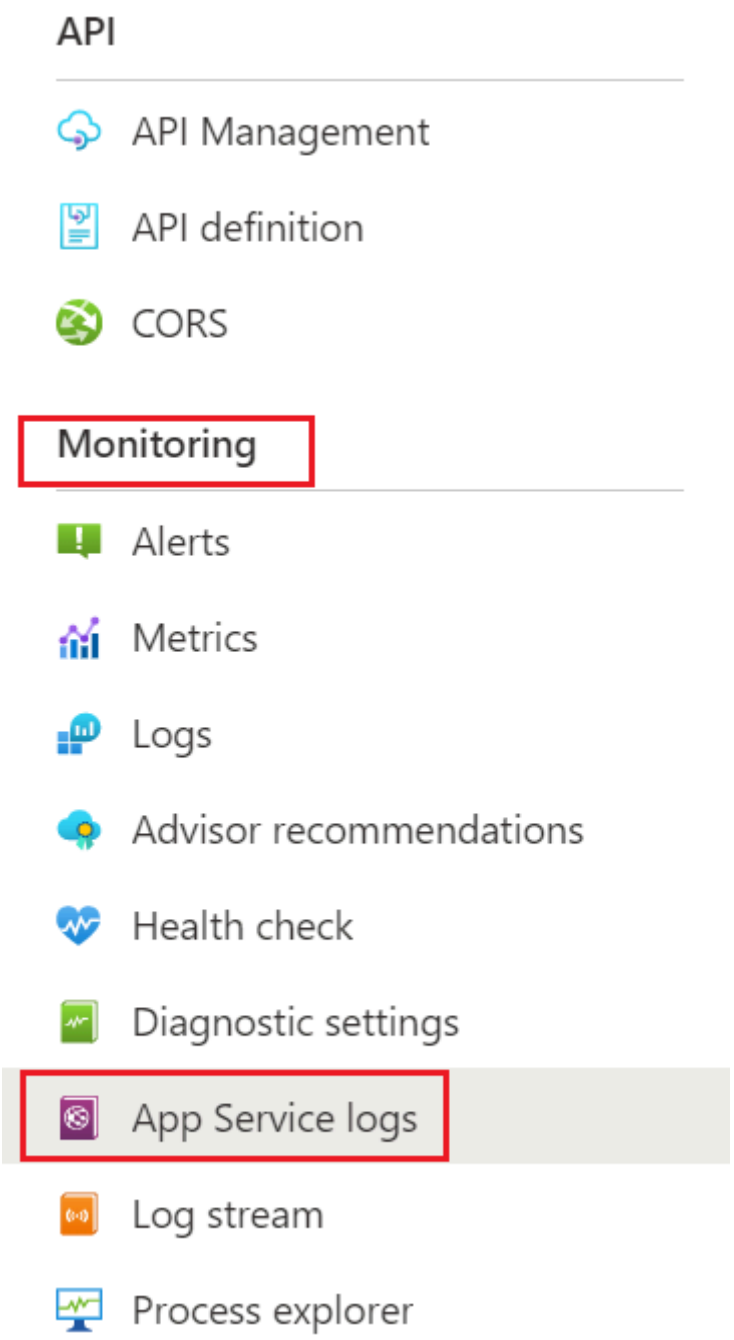
> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Azure App Services Log stream

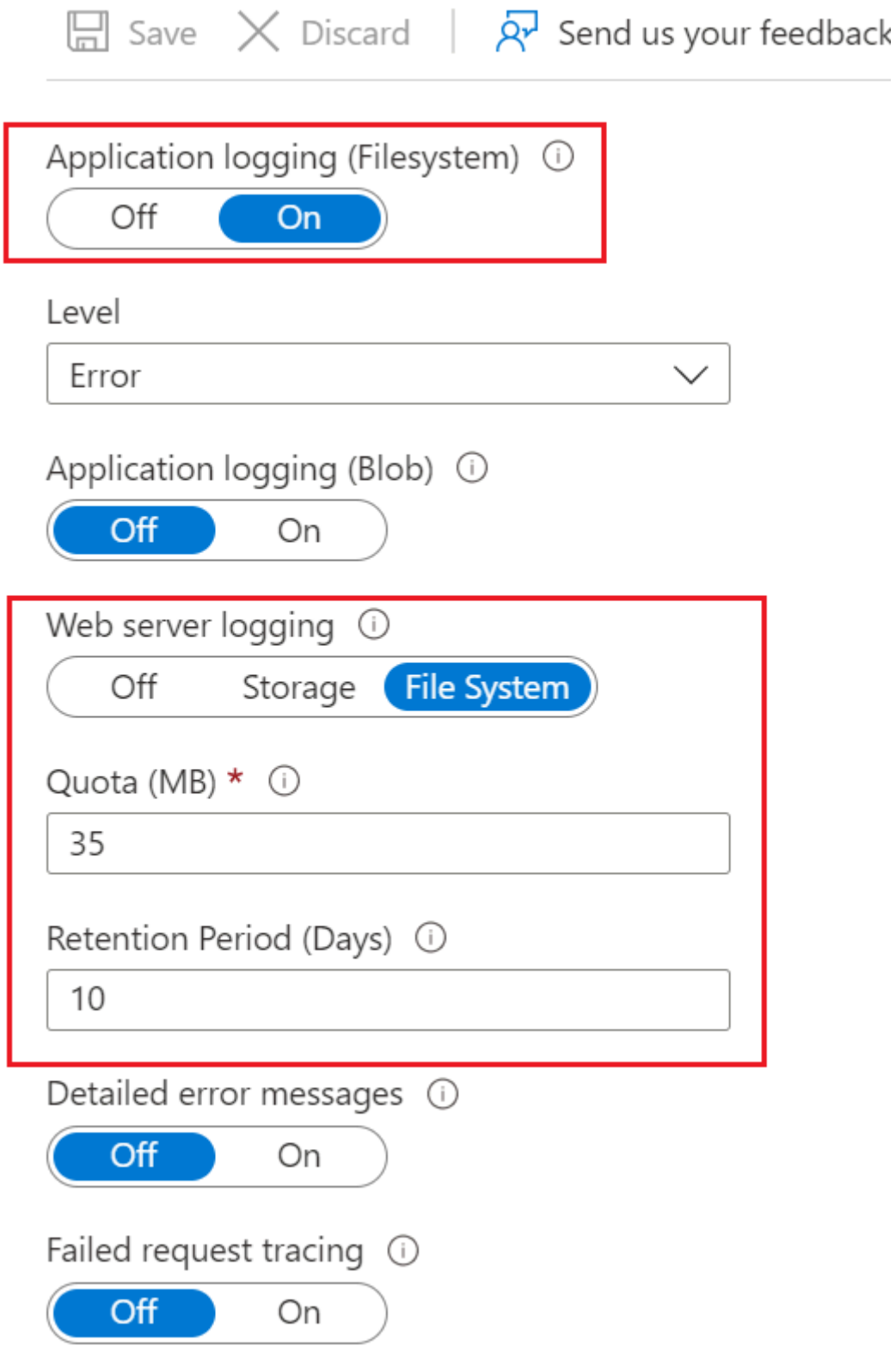
The Azure App Services Log streams logging information as it occurs. To view streaming logs:

1. In the Azure portal, open the app in **App Services**.

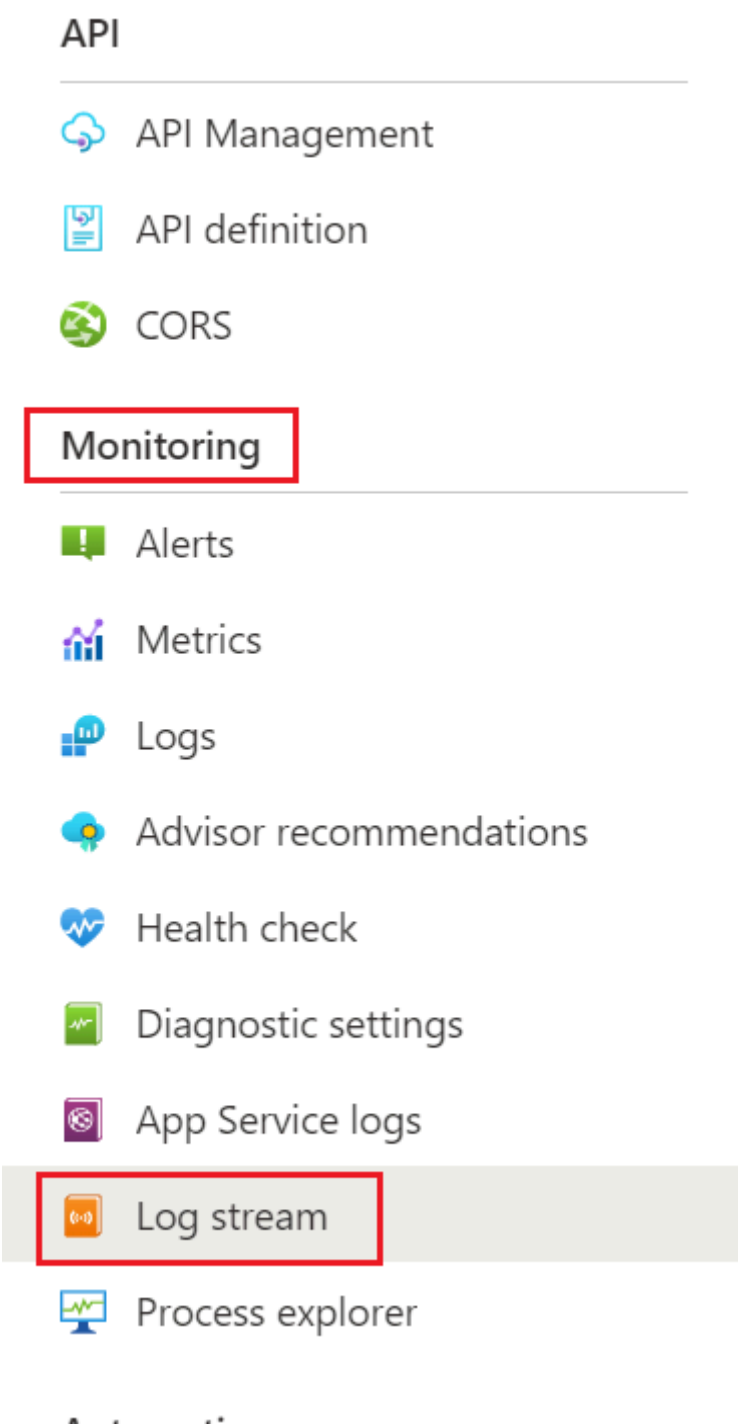
2. In the left pane, navigate to **Monitoring** > **App Service Logs**.



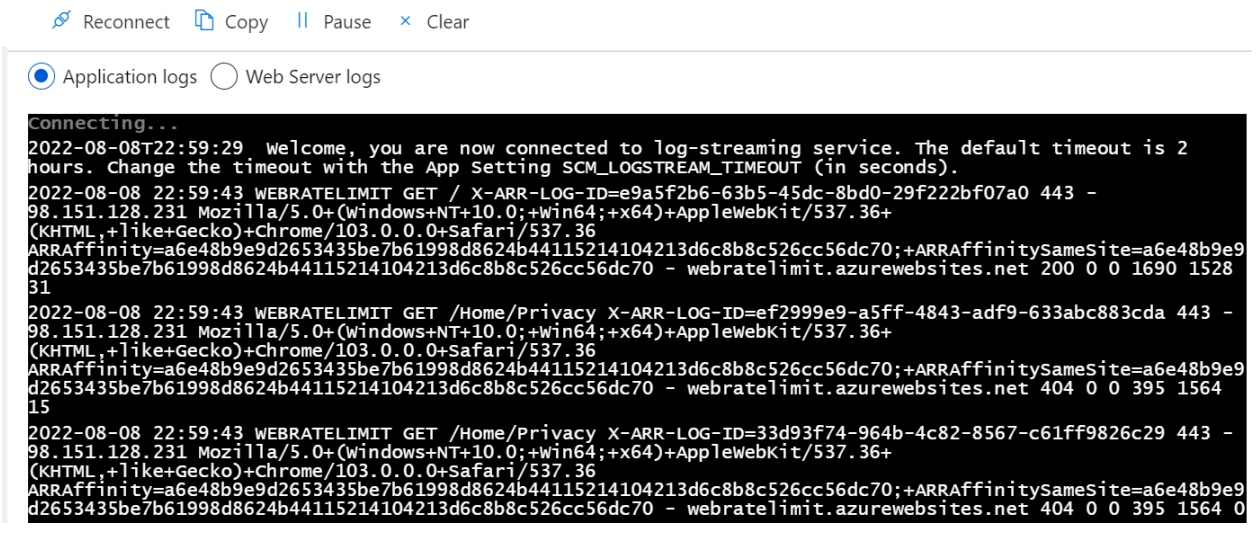
3. Select **File System** for **Web Server Logging**. Optionally enable **Application logging**.



4. In the left pane, navigate to **Monitoring** > **Log stream**, and then select **Application logs** or **Web Server Logs**.



The following images shows the application logs output:



Streaming logs have some latency and might not display immediately.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the `eventlog.xml` file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:

- If the app is a framework-dependent deployment:

  .NET CLI

  ```
  dotnet .\{ASSEMBLY NAME}.dll
  ```

- If the app is a self-contained deployment:

  Console

  ```
  {ASSEMBLY NAME}.exe
  ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:

  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

- If the app is a self-contained deployment:

  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. In the Azure Portal, navigate to the web app.
2. In the **App Service** blade, enter **kudu** in the search box.
3. Select **Advanced Tools** > **Go**.
4. Select **Debug console** > **CMD**.
5. Navigate to *site/wwwroot*
6. Select the pencil icon to edit the *web.config* file.
7. In the `<aspNetCore />` element, set `stdoutLogEnabled="true"` and select **Save**.

Disable stdout logging when troubleshooting is complete by setting `stdoutLogEnabled="false"`.

For more information, see ASP.NET Core Module (ANCM) for IIS.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:

- Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic logging. Redeploy the app.
- Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
  a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
  b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
  c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.

2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see Log creation and redirection with the ASP.NET Core Module.

> ⚠ **Warning**
>
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or nonresponsive app (Azure App Service)

When an app responds slowly or doesn't respond to a request, see Troubleshoot slow web app performance issues in Azure App Service.

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.

5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`).

`stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.

4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module (ANCM) for IIS.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

XML

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see Log creation and redirection with the ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

XML

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development"
/>
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or non-responding app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.

2. Run the EnableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.

4. After the crash has occurred, run the DisableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> ⚠ **Warning**
>
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App doesn't respond, fails during startup, or runs normally

When an app stops responding but doesn't crash, fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Debug .NET and ASP.NET Core source code with Visual Studio
- Troubleshoot and debug ASP.NET Core projects
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module (ANCM) for IIS

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

Common error troubleshooting for Azure App Service and IIS with ASP.NET Core

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**.

This topic describes the most common errors and provides troubleshooting advice when hosting ASP.NET Core apps on Azure Apps Service and IIS.

See Troubleshoot ASP.NET Core on Azure App Service and IIS information on common app startup errors and instructions on how to diagnose errors.

Collect the following information:

- Browser behavior such as status code and error message.
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS

    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.

- ASP.NET Core Module stdout and debug log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> ⓘ **Important**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff. Could not find inprocess request handler. It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site extension, restart the app. Wait several seconds for the app to restart.
- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.
- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure

- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely

- **ASP.NET Core Module stdout Log:** The log file is created but empty.

- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8007023e

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure

- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.

- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED **--OR--** Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied **--OR--** 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. --**OR**-- This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections>` `<sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  XML
  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0. Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed to start application '/LM/W3SVC/2/ROOT', ErrorCode '0x8007023e'.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** Event Log: 'Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed HRESULT returned: 0x8007023e

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

Current .NET Core Hosting Bundle installer (direct download)

For more information, see Install the .NET Core Hosting Bundle.

If a specific runtime is required, download the runtime from the .NET Downloads page and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed HRESULT returned: 0x8000ffff

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`) for a self-contained deployment (SCD).

Missing .NET Core shared framework

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.

Failed to start application '/LM/W3SVC/5/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8000ffff

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.
- **ASP.NET Core Module Debug Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section or that the sub-app doesn't inherit the parent app's handlers.

The parent app's `<system.webServer>` section of *web.config* is placed inside of a `<location>` element. The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the parent app. For more information, see ASP.NET Core Module (ANCM) for IIS.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module debug Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core

Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure **--OR--** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Variable
- **ASP.NET Core Module stdout Log:** The log file is created but empty or created with normal entries until the point of the app failing.
- **ASP.NET Core Module Debug Log:** Variable

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

Advanced configuration of the ASP.NET Core Module and IIS

Article • 09/27/2024

> **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.

This article covers advanced configuration options and scenarios for the ASP.NET Core Module and IIS.

Modify the stack size

Only applies when using the in-process hosting model.

Configure the managed stack size using the `stackSize` setting in bytes in the `web.config` file. The default size is 1,048,576 bytes (1 MB). The following example changes the stack size to 2 MB (2,097,152 bytes):

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="stackSize" value="2097152" />
  </handlerSettings>
</aspNetCore>
```

Disallow rotation on config

The `disallowRotationOnConfigChange` setting is intended for blue/green scenarios where a change to global config should not cause all sites to recycle. When this flag is true, only changes relevant to the site itself will cause it to recycle. For example, a site recycles

if its *web.config* changes or something changes that is relevant to the site's path from IIS's perspective. But a general change to *applicationHost.config* would not cause an app to recycle. The following example sets this setting to true:

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="disallowRotationOnConfigChange" value="true" />
  </handlerSettings>
</aspNetCore>
```

This setting corresponds to the API ApplicationPoolRecycling.DisallowRotationOnConfigChange

Reduce 503 likelihood during app recycle

By default, there is a one second delay between when IIS is notified of a recycle or shutdown and when ANCM tells the managed server to initiate shutdown. The delay is configurable via the `ANCM_shutdownDelay` environment variable or by setting the `shutdownDelay` handler setting. Both values are in milliseconds. The delay is primarily to reduce the likelihood of a race where:

- IIS hasn't started queuing requests to go to the new app.
- ANCM starts rejecting new requests that come into the old app.

This setting doesn't mean incoming requests will be delayed by this amount. The setting indicates that the old app instance will start shutting down after the timeout occurs. Slower machines or machines with heavier CPU usage might need to adjust this value to reduce 503 likelihood.

The following example sets the delay to 5 seconds:

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="shutdownDelay" value="5000" />
```

```
  </handlerSettings>
</aspNetCore>
```

Proxy configuration uses HTTP protocol and a pairing token

Only applies to out-of-process hosting.

The proxy created between the ASP.NET Core Module and Kestrel uses the HTTP protocol. There's no risk of eavesdropping the traffic between the module and Kestrel from a location off of the server.

A pairing token is used to guarantee that the requests received by Kestrel were proxied by IIS and didn't come from some other source. The pairing token is created and set into an environment variable (`ASPNETCORE_TOKEN`) by the module. The pairing token is also set into a header (`MS-ASPNETCORE-TOKEN`) on every proxied request. IIS Middleware checks each request it receives to confirm that the pairing token header value matches the environment variable value. If the token values are mismatched, the request is logged and rejected. The pairing token environment variable and the traffic between the module and Kestrel aren't accessible from a location off of the server. Without knowing the pairing token value, a cyberattacker can't submit requests that bypass the check in the IIS Middleware.

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the `TrustedInstaller` account. Because the local system account doesn't have modify permission for the share path used by the IIS Shared Configuration, the installer throws an access denied error when attempting to configure the module settings in the `applicationHost.config` file on the share.

When using an IIS Shared Configuration on the same machine as the IIS installation, run the ASP.NET Core Hosting Bundle installer with the `OPT_NO_SHARED_CONFIG_CHECK` parameter set to `1`:

Console

```
dotnet-hosting-{VERSION}.exe OPT_NO_SHARED_CONFIG_CHECK=1
```

When the path to the shared configuration isn't on the same machine as the IIS installation, follow these steps:

1. Disable the IIS Shared Configuration.
2. Run the installer.
3. Export the updated `applicationHost.config` file to the file share.
4. Re-enable the IIS Shared Configuration.

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including middleware used in authentication. Even if Data Protection APIs aren't called by user code, data protection should be configured with a deployment script or in user code to create a persistent cryptographic key store. If data protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the Data Protection key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection under IIS to persist the key ring, use **one** of the following approaches:

- **Create Data Protection Registry keys**

  Data Protection keys used by ASP.NET Core apps are stored in the registry external to the apps. To persist the keys for a given app, create Registry keys for the app pool.

  For standalone, non-webfarm IIS installations, the Data Protection Provision-AutoGenKeys.ps1 PowerShell script can be used for each app pool used with an ASP.NET Core app. This script creates a Registry key in the HKLM registry that's accessible only to the worker process account of the app's app pool. Keys are encrypted at rest using DPAPI with a machine-wide key.

  In web farm scenarios, an app can be configured to use a UNC path to store its Data Protection key ring. By default, the keys aren't encrypted. Ensure that the file permissions for the network share are limited to the Windows account that the app runs under. An X509 certificate can be used to protect keys at rest. Consider a mechanism to allow users to upload certificates. Place certificates into the user's

trusted certificate store and ensure they're available on all machines where the user's app runs. For more information, see Configure ASP.NET Core Data Protection.

- **Configure the IIS Application Pool to load the user profile**

  This setting is in the **Process Model** section under the **Advanced Settings** for the app pool. Set **Load User Profile** to `True`. When set to `True`, keys are stored in the user profile directory and protected using DPAPI with a key specific to the user account. Keys are persisted to the `%LOCALAPPDATA%/ASP.NET/DataProtection-Keys` folder.

  The app pool's setProfileEnvironment attribute must also be enabled. The default value of `setProfileEnvironment` is `true`. In some scenarios (for example, Windows OS), `setProfileEnvironment` is set to `false`. If keys aren't stored in the user profile directory as expected:

  1. Navigate to the `%windir%/system32/inetsrv/config` folder.
  2. Open the `applicationHost.config` file.
  3. Locate the `<system.applicationHost><applicationPools><applicationPoolDefaults><processModel>` element.
  4. Confirm that the `setProfileEnvironment` attribute isn't present, which defaults the value to `true`, or explicitly set the attribute's value to `true`.

- **Use the file system as a key ring store**

  Adjust the app code to use the file system as a key ring store. Use an X509 certificate to protect the key ring and ensure the certificate is a trusted certificate. If the certificate is self-signed, place the certificate in the Trusted Root store.

  When using IIS in a web farm:
  - Use a file share that all machines can access.
  - Deploy an X509 certificate to each machine. Configure Data Protection in code.

- **Set a machine-wide policy for Data Protection**

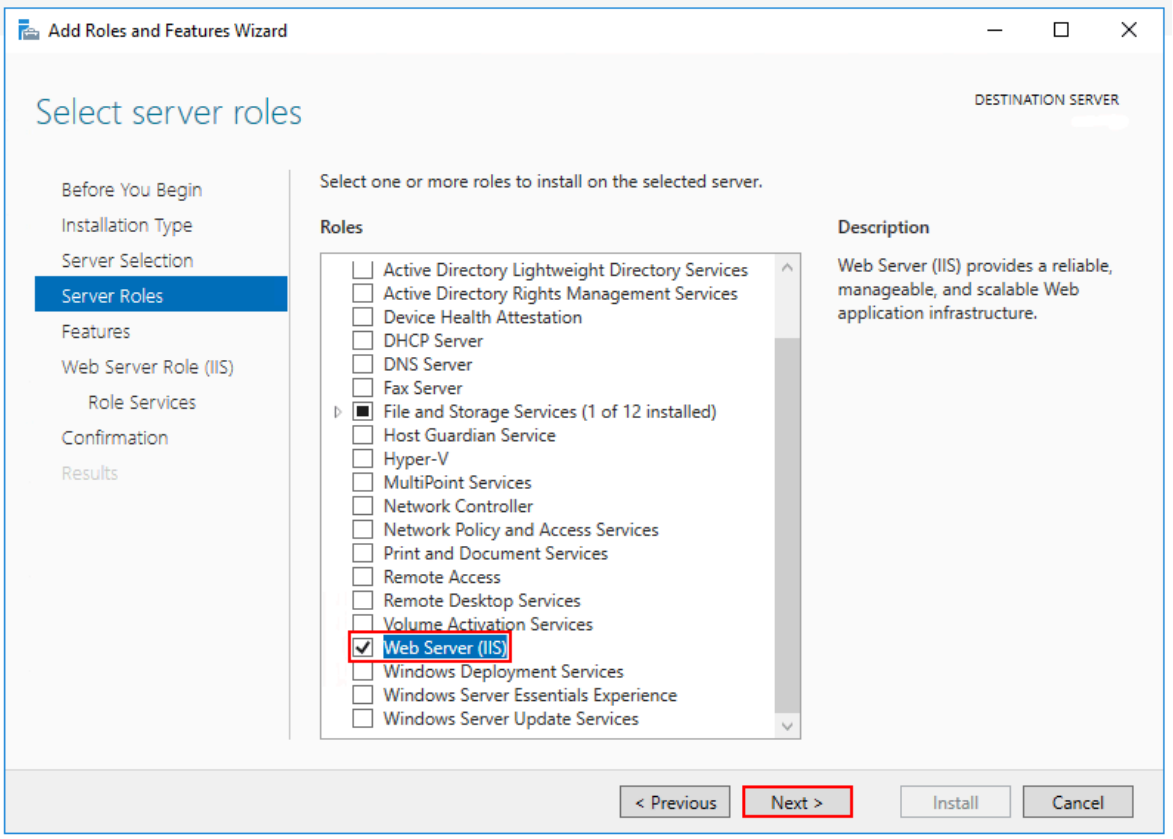
  The Data Protection system has limited support for setting a default machine-wide policy for all apps that consume the Data Protection APIs. For more information, see ASP.NET Core Data Protection Overview.

IIS configuration

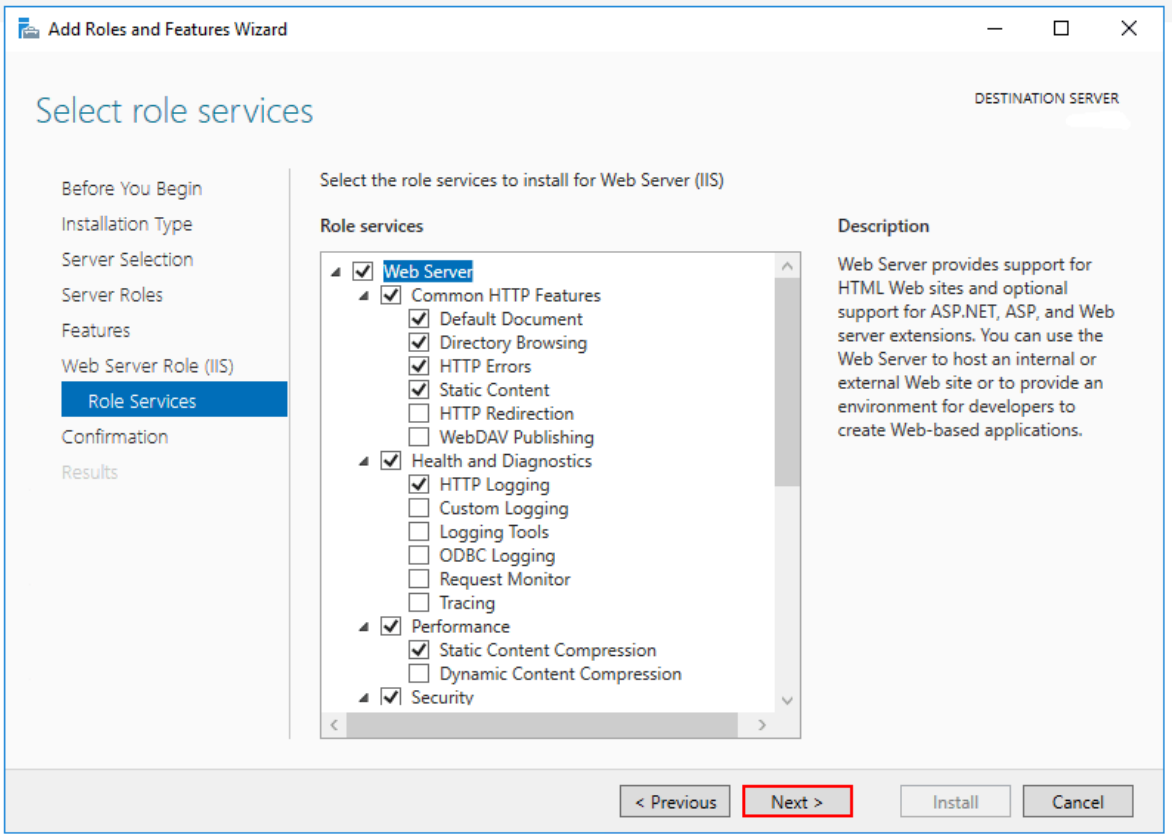
Windows Server operating systems

Enable the **Web Server (IIS)** server role and establish role services.

1. Use the **Add Roles and Features** wizard from the **Manage** menu or the link in **Server Manager**. On the **Server Roles** step, check the box for **Web Server (IIS)**.



2. After the **Features** step, the **Role services** step loads for Web Server (IIS). Select the IIS role services desired or accept the default role services provided.



Windows Authentication (Optional)
To enable Windows Authentication, expand the following nodes: **Web Server** > **Security**. Select the **Windows Authentication** feature. For more information, see Windows Authentication <windowsAuthentication> and Configure Windows authentication.

WebSockets (Optional)
WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **Web Server** > **Application Development**. Select the **WebSocket Protocol** feature. For more information, see WebSockets.

3. Proceed through the **Confirmation** step to install the web server role and services. A server/IIS restart isn't required after installing the **Web Server (IIS)** role.

Windows desktop operating systems

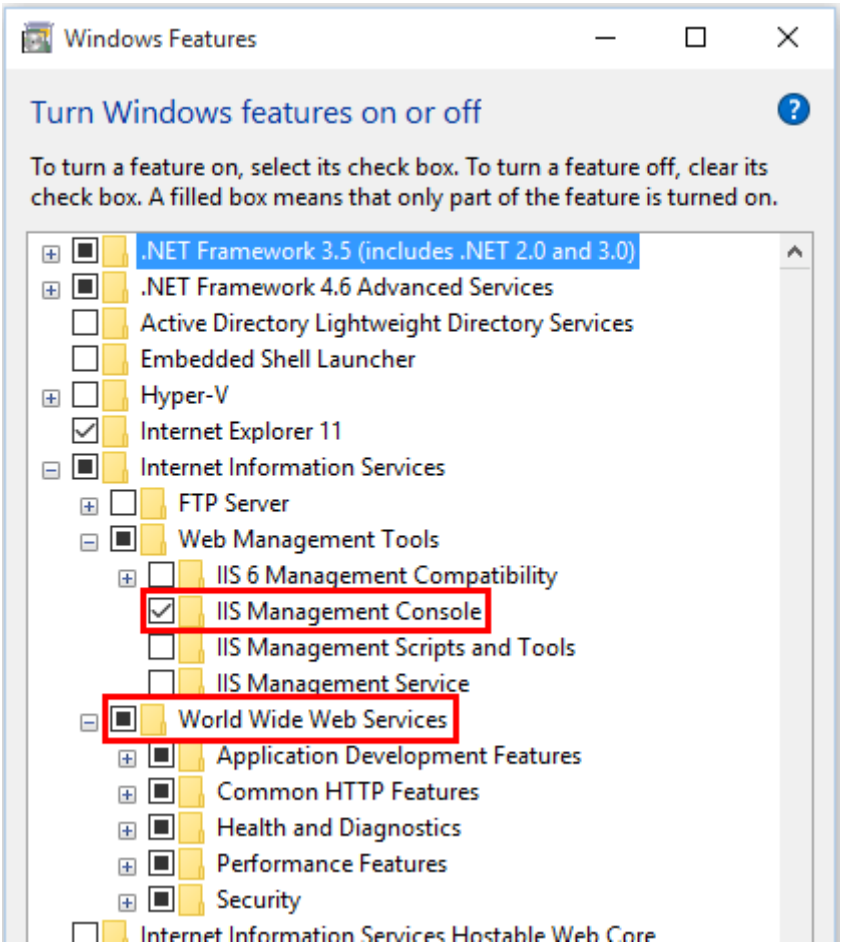
Enable the **IIS Management Console** and **World Wide Web Services**.

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open the **Internet Information Services** node. Open the **Web Management Tools** node.
3. Check the box for **IIS Management Console**.
4. Check the box for **World Wide Web Services**.
5. Accept the default features for **World Wide Web Services** or customize the IIS features.

   Windows Authentication (Optional)
   To enable Windows Authentication, expand the following nodes: **World Wide Web Services** > **Security**. Select the **Windows Authentication** feature. For more information, see Windows Authentication <windowsAuthentication> and Configure Windows authentication.

   WebSockets (Optional)
   WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **World Wide Web Services** > **Application Development Features**. Select the **WebSocket Protocol** feature. For more information, see WebSockets.

6. If the IIS installation requires a restart, restart the system.



Virtual Directories

IIS Virtual Directories aren't supported with ASP.NET Core apps. An app can be hosted as a sub-application.

Sub-applications

An ASP.NET Core app can be hosted as an IIS sub-application (sub-app). The sub-app's path becomes part of the root app's URL.

Static asset links within the sub-app should use tilde-slash (`~/`) notation in MVC and Razor Pages. Tilde-slash notation triggers a Tag Helper to prepend the sub-app's pathbase to the rendered relative link. For a sub-app at `/subapp_path`, an image linked with `src="~/image.png"` is rendered as `src="/subapp_path/image.png"`. The root app's Static File Middleware doesn't process the static file request. The request is processed by the sub-app's Static File Middleware.

> ⓘ Note

Razor components (`.razor`) shouldn't use tilde-slash notation. For more information, see the **Blazor app base path documentation**.

If a static asset's `src` attribute is set to an absolute path (for example, `src="/image.png"`), the link is rendered without the sub-app's pathbase. The root app's Static File Middleware attempts to serve the asset from the root app's web root, which results in a *404 - Not Found* response unless the static asset is available from the root app.

To host an ASP.NET Core app as a sub-app under another ASP.NET Core app:

1. Establish an app pool for the sub-app. Set the **.NET CLR Version** to **No Managed Code** because the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process, not the desktop CLR (.NET CLR).
2. Add the root site in IIS Manager with the sub-app in a folder under the root site.
3. Right-click the sub-app folder in IIS Manager and select **Convert to Application**.
4. In the **Add Application** dialog, use the **Select** button for the **Application Pool** to assign the app pool that you created for the sub-app. Select **OK**.

The assignment of a separate app pool to the sub-app is a requirement when using the in-process hosting model.

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

Application Pools

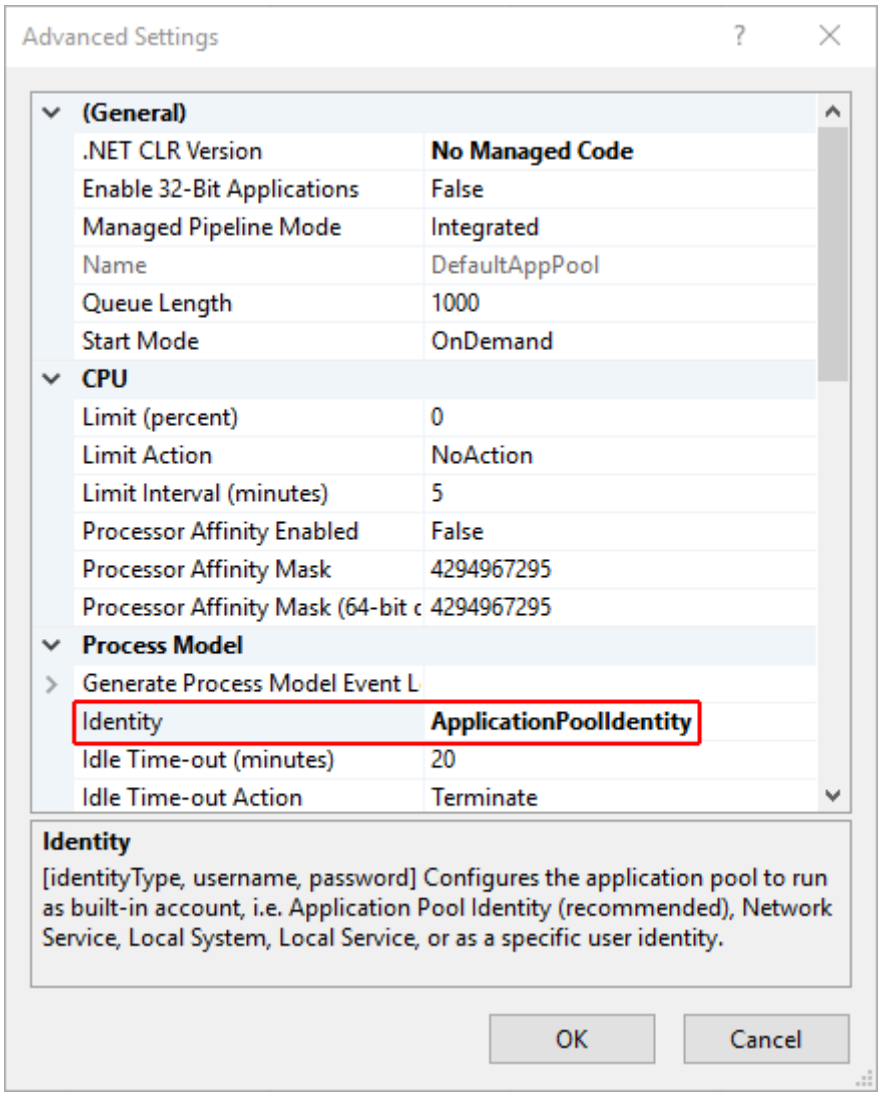
App pool isolation is determined by the hosting model:

- In-process hosting: Apps are required to run in separate app pools.
- Out-of-process hosting: We recommend isolating the apps from each other by running each app in its own app pool.

The IIS **Add Website** dialog defaults to a single app pool per app. When a **Site name** is provided, the text is automatically transferred to the **Application pool** textbox. A new app pool is created using the site name when the site is added.

Application Pool Identity

An app pool identity account allows an app to run under a unique account without having to create and manage domains or local accounts. On IIS 8.0 or later, the IIS Admin Worker Process (WAS) creates a virtual account with the name of the new app pool and runs the app pool's worker processes under this account by default. In the IIS Management Console under **Advanced Settings** for the app pool, ensure that the **Identity** is set to use `ApplicationPoolIdentity`:



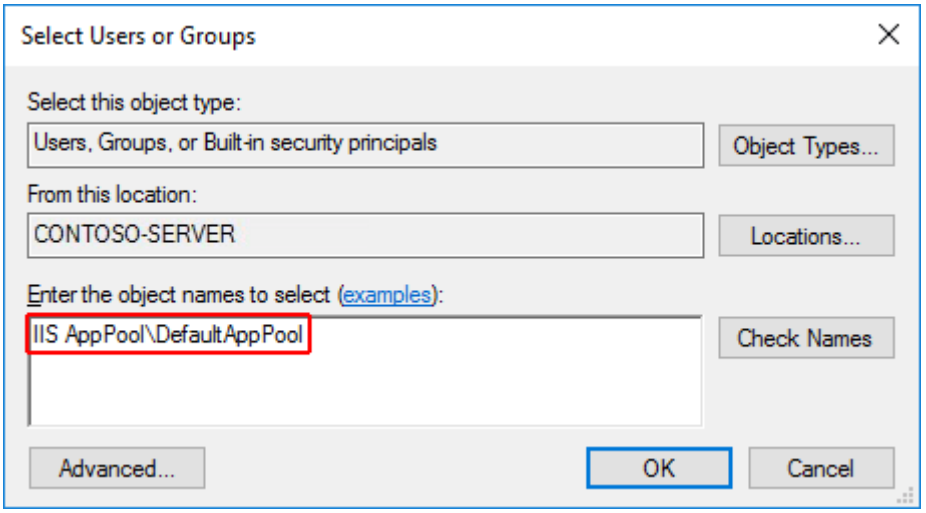
The IIS management process creates a secure identifier with the name of the app pool in the Windows Security System. Resources can be secured using this identity. However, this identity isn't a real user account and doesn't show up in the Windows User Management Console.

If the IIS worker process requires elevated access to the app, modify the Access Control List (ACL) for the directory containing the app:

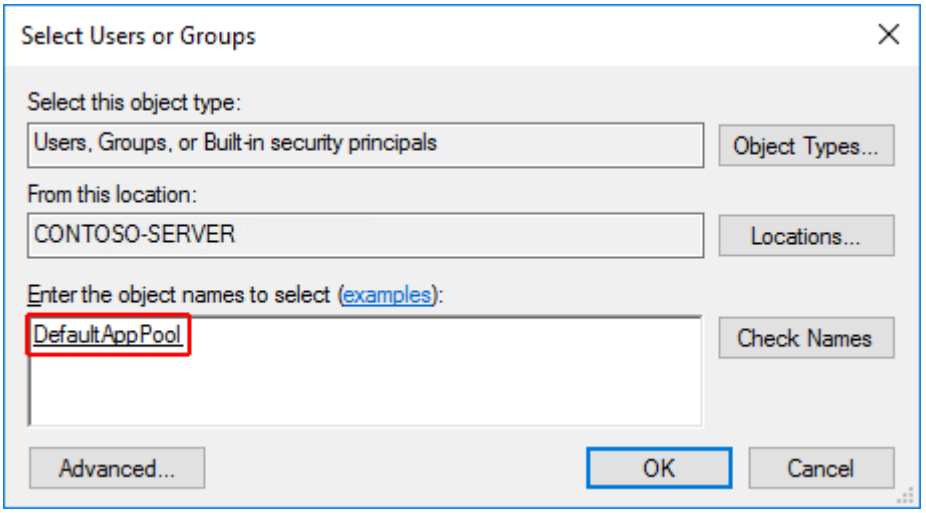
1. Open Windows Explorer and navigate to the directory.
2. Right-click on the directory and select **Properties**.
3. Under the **Security** tab, select the **Edit** button and then the **Add** button.

4. Select the **Locations** button and make sure the system is selected.

5. Enter `IIS AppPool\{APP POOL NAME}` format, where the placeholder `{APP POOL NAME}` is the app pool name, in **Enter the object names to select** area. Select the **Check Names** button. For the *DefaultAppPool* check the names using `IIS AppPool\DefaultAppPool`. When the **Check Names** button is selected, a value of `DefaultAppPool` is indicated in the object names area. It isn't possible to enter the app pool name directly into the object names area. Use the `IIS AppPool\{APP POOL NAME}` format, where the placeholder `{APP POOL NAME}` is the app pool name, when checking for the object name.



6. Select **OK**.



7. Read & execute permissions should be granted by default. Provide additional permissions as needed.

Access can also be granted at a command prompt using the **ICACLS** tool. Using the *DefaultAppPool* as an example, the following command is used to grant read and execute permissions to the `MyWebApp` folder, subfolders, and files:

```
```

Console

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool:(OI)(CI)RX"
```

For more information, see the icacls topic.

HTTP/2 support

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

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

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. 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.

CORS preflight requests

This section only applies to ASP.NET Core apps that target the .NET Framework.

For an ASP.NET Core app that targets the .NET Framework, OPTIONS requests aren't passed to the app by default in IIS. To learn how to configure the app's IIS handlers in `web.config` to pass OPTIONS requests, see Enable cross-origin requests in ASP.NET Web API 2: How CORS Works.

Application Initialization Module and Idle Timeout

When hosted in IIS by the ASP.NET Core Module version 2:

- Application Initialization Module: App's hosted in-process or out-of-process can be configured to start automatically on a worker process restart or server restart.
- Idle Timeout: App's hosted in-process can be configured not to time out during periods of inactivity.

Application Initialization Module

Applies to apps hosted in-process and out-of-process.

IIS Application Initialization is an IIS feature that sends an HTTP request to the app when the app pool starts or is recycled. The request triggers the app to start. By default, IIS issues a request to the app's root URL (`/`) to initialize the app (see the additional resources for more details on configuration).

Confirm that the IIS Application Initialization role feature in enabled:

On Windows 7 or later desktop systems when using IIS locally:

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open **Internet Information Services** > **World Wide Web Services** > **Application Development Features**.
3. Select the checkbox for **Application Initialization**.

On Windows Server 2008 R2 or later:

1. Open the **Add Roles and Features Wizard**.
2. In the **Select role services** panel, open the **Application Development** node.
3. Select the checkbox for **Application Initialization**.

Use either of the following approaches to enable the Application Initialization Module for the site:

- Using IIS Manager:

    1. Select **Application Pools** in the **Connections** panel.
    2. Right-click the app's app pool in the list and select **Advanced Settings**.

3. The default **Start Mode** is `OnDemand`. Set the **Start Mode** to `AlwaysRunning`. Select **OK**.
4. Open the **Sites** node in the **Connections** panel.
5. Right-click the app and select **Manage Website** > **Advanced Settings**.
6. The default **Preload Enabled** setting is `False`. Set **Preload Enabled** to `True`. Select **OK**.

- Using `web.config`, add the `<applicationInitialization>` element with `doAppInitAfterRestart` set to `true` to the `<system.webServer>` elements in the app's `web.config` file:

```
XML
```

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <applicationInitialization doAppInitAfterRestart="true" />
    </system.webServer>
  </location>
</configuration>
```

Idle Timeout

Only applies to apps hosted in-process.

To prevent the app from idling, set the app pool's idle timeout using IIS Manager:

1. Select **Application Pools** in the **Connections** panel.
2. Right-click the app's app pool in the list and select **Advanced Settings**.
3. The default **Idle Time-out (minutes)** is `20` minutes. Set the **Idle Time-out (minutes)** to `0` (zero). Select **OK**.
4. Recycle the worker process.

To prevent apps hosted out-of-process from timing out, use either of the following approaches:

- Ping the app from an external service in order to keep it running.
- If the app only hosts background services, avoid IIS hosting and use a Windows Service to host the ASP.NET Core app.

Application Initialization Module and Idle Timeout additional resources

- IIS 8.0 Application Initialization
- Application Initialization <applicationInitialization>.
- Process Model Settings for an Application Pool <processModel>.

Module, schema, and configuration file locations

Module

IIS (x86/amd64):

- `%windir%\System32\inetsrv\aspnetcore.dll`
- `%windir%\SysWOW64\inetsrv\aspnetcore.dll`
- `%ProgramFiles%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll`
- `%ProgramFiles(x86)%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll`

IIS Express (x86/amd64):

- `%ProgramFiles%\IIS Express\aspnetcore.dll`
- `%ProgramFiles(x86)%\IIS Express\aspnetcore.dll`
- `%ProgramFiles%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll`
- `%ProgramFiles(x86)%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll`

Schema

IIS

- `%windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml`
- `%windir%\System32\inetsrv\config\schema\aspnetcore_schema_v2.xml`

IIS Express

- `%ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml`
- `%ProgramFiles%\IIS Express\config\schema\aspnetcore_schema_v2.xml`

Configuration

IIS

- `%windir%\System32\inetsrv\config\applicationHost.config`

IIS Express

- Visual Studio: `{APPLICATION ROOT}\.vs\config\applicationHost.config`
- *iisexpress.exe* CLI: `%USERPROFILE%\Documents\IISExpress\config\applicationhost.config`

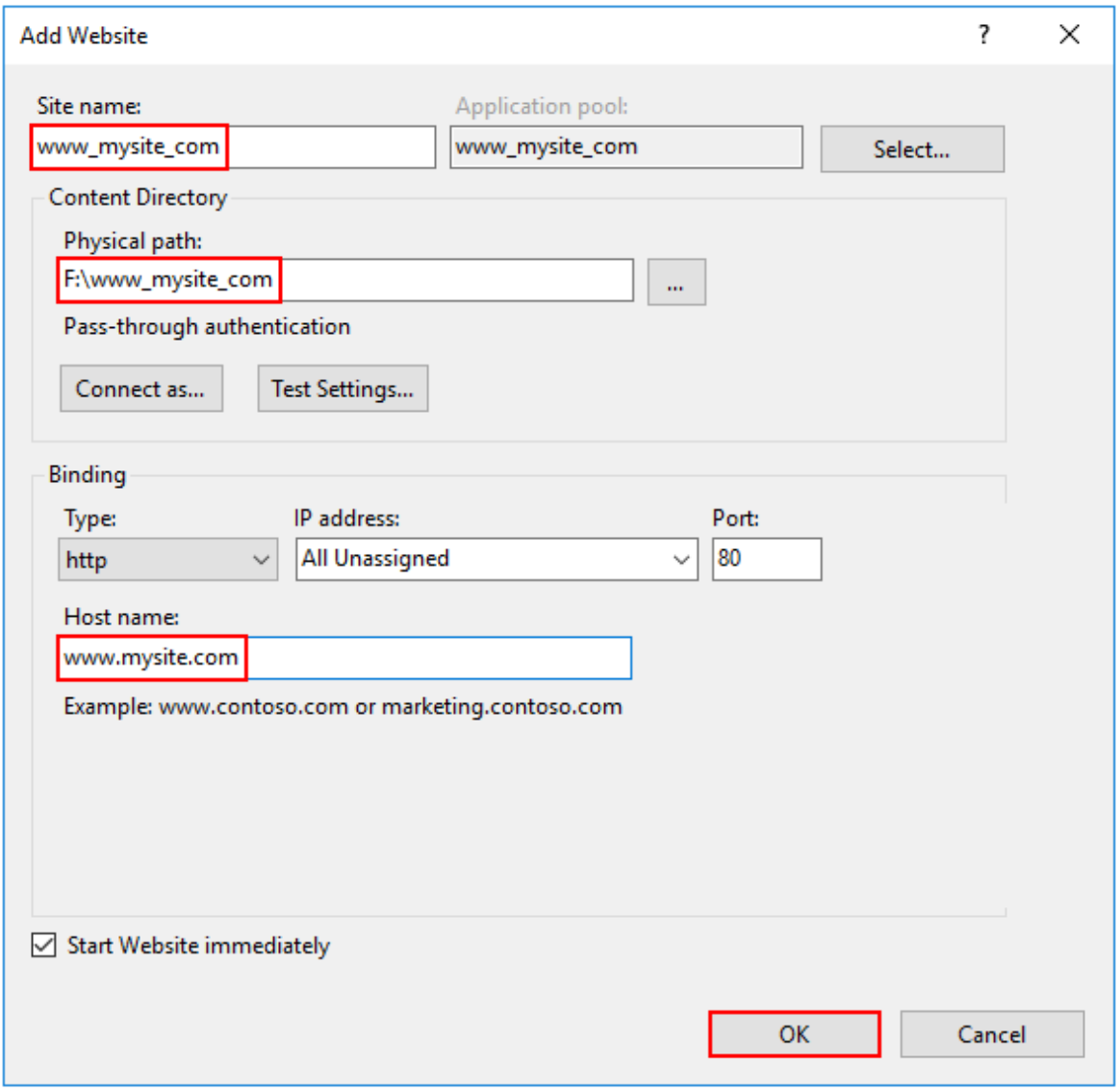
The files can be found by searching for `aspnetcore` in the `applicationHost.config` file.

Install Web Deploy when publishing with Visual Studio

When deploying apps to servers with Web Deploy, install the latest version of Web Deploy on the server. To install Web Deploy, see IIS Downloads: Web Deploy.

Create the IIS site

1. On the hosting system, create a folder to contain the app's published folders and files. In a following step, the folder's path is provided to IIS as the physical path to the app. For more information on an app's deployment folder and file layout, see ASP.NET Core directory structure.
2. In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.
3. Provide a **Site name** and set the **Physical path** to the app's deployment folder. Provide the **Binding** configuration and create the website by selecting **OK**:

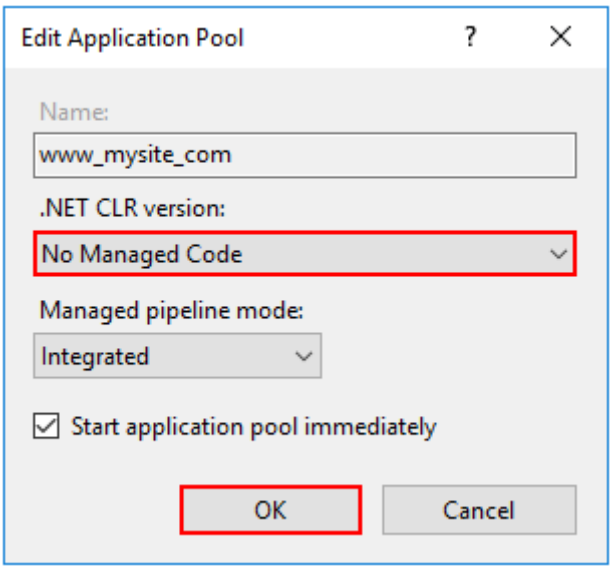


> ⚠ **Warning**
>
> Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings can open up your app to security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) doesn't have this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). See RFC 9110: HTTP Semantics (Section 7.2: Host and :authority) for more information.

4. Under the server's node, select **Application Pools**.

5. Right-click the site's app pool and select **Basic Settings** from the contextual menu.

6. In the **Edit Application Pool** window, set the **.NET CLR version** to **No Managed Code**:



ASP.NET Core runs in a separate process and manages the runtime. ASP.NET Core doesn't rely on loading the desktop CLR (.NET CLR). The Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process. Setting the **.NET CLR version** to **No Managed Code** is optional but recommended.

- For a 32-bit (x86) self-contained deployment published with a 32-bit SDK that uses the in-process hosting model, enable the Application Pool for 32-bit. In IIS Manager, navigate to **Application Pools** in the **Connections** sidebar. Select the app's Application Pool. In the **Actions** sidebar, select **Advanced Settings**. Set **Enable 32-Bit Applications** to `True`.

- For a 64-bit (x64) self-contained deployment that uses the in-process hosting model, disable the app pool for 32-bit (x86) processes. In IIS Manager, navigate to **Application Pools** in the **Connections** sidebar. Select the app's Application Pool. In the **Actions** sidebar, select **Advanced Settings**. Set **Enable 32-Bit Applications** to `False`.

7. Confirm the process model identity has the proper permissions.

   If the default identity of the app pool (**Process Model** > **Identity**) is changed from **ApplicationPoolIdentity** to another identity, verify that the new identity has the required permissions to access the app's folder, database, and other required resources. For example, the app pool requires read and write access to folders where the app reads and writes files.

Windows Authentication configuration (Optional)
For more information, see Configure Windows authentication.

Shadow copy

Shadow copying app assemblies to the ASP.NET Core Module (ANCM) for IIS can provide a better end user experience than stopping the app by deploying an app offline file.

When an ASP.NET Core app is running on Windows, the binaries are locked so that they can't be modified or replaced. Shadow copying enables the app assemblies to be updated while the app is running by making a copy of the assemblies.

Shadow copy isn't intended to enable zero-downtime deployment, so its expected that IIS will still recycle the app, and some requests may get an 503 Service Unavailable response. We recommend using a pattern like blue-green deployments or Azure deployment slots for zero-downtime deployments. Shadow copy helps minimize downtime on deployments, but can't completely eliminate it.

Shadow copying is enabled by customizing the ANCM handler settings in `web.config`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <remove name="aspNetCore"/>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified"/>
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%" arguments="%LAUNCHER_ARGS%"
stdoutLogEnabled="false" stdoutLogFile=".logsstdout">
      <handlerSettings>
        <handlerSetting name="enableShadowCopy" value="true" />
        <!-- Ensure that the IIS ApplicationPool identity has permission to
this directory -->
        <handlerSetting name="shadowCopyDirectory"
value="../ShadowCopyDirectory/" />
      </handlerSettings>
    </aspNetCore>
  </system.webServer>
</configuration>
```

Transform web.config

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 Vijay Ramakrishnan

Transformations to the *web.config* file can be applied automatically when an app is published based on:

- Build configuration
- Profile
- Environment
- Custom

These transformations occur for either of the following *web.config* generation scenarios:

- Generated automatically by the `Microsoft.NET.Sdk.Web` SDK.
- Provided by the developer in the content root of the app.

Build configuration

Build configuration transforms are run first.

Include a *web.{CONFIGURATION}.config* file for each build configuration (Debug|Release) requiring a *web.config* transformation.

In the following example, a configuration-specific environment variable is set in *web.Release.config*:

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="Configuration_Specific"
                               value="Configuration_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the configuration is set to *Release*:

.NET CLI

```
dotnet publish --configuration Release
```

The MSBuild property for the configuration is `$(Configuration)`.

Profile

Profile transformations are run second, after Build configuration transforms.

Include a *web.{PROFILE}.config* file for each profile configuration requiring a *web.config* transformation.

In the following example, a profile-specific environment variable is set in *web.FolderProfile.config* for a folder publish profile:

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="Profile_Specific"
                               value="Profile_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the profile is *FolderProfile*:

.NET CLI

```
dotnet publish --configuration Release /p:PublishProfile=FolderProfile
```

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

If no profile is passed, the default profile name is **FileSystem** and *web.FileSystem.config* is applied if the file is present in the app's content root.

Environment

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

Include a *web.{ENVIRONMENT}.config* file for each environment requiring a *web.config* transformation.

In the following example, an environment-specific environment variable is set in *web.Production.config* for the Production environment:

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="Environment_Specific"
                               value="Environment_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the environment is *Production*:

.NET CLI

```
dotnet publish --configuration Release /p:EnvironmentName=Production
```