This package is required by both the server and client projects. The `Grpc.AspNetCore` metapackage includes a reference to `Grpc.Tools`. Server projects can add `Grpc.AspNetCore` using the Package Manager in Visual Studio or by adding a `<PackageReference>` to the project file:

XML

```xml
<PackageReference Include="Grpc.AspNetCore" Version="2.32.0" />
```

Client projects should directly reference `Grpc.Tools` alongside the other packages required to use the gRPC client. The tooling package isn't required at runtime, so the dependency is marked with `PrivateAssets="All"`:

XML

```xml
<PackageReference Include="Google.Protobuf" Version="3.18.0" />
<PackageReference Include="Grpc.Net.Client" Version="2.52.0" />
<PackageReference Include="Grpc.Tools" Version="2.40.0">
  <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
  <PrivateAssets>all</PrivateAssets>
</PackageReference>
```

Generated C# assets

The tooling package generates the C# types representing the messages defined in the included `.proto` files.

For server-side assets, an abstract service base type is generated. The base type contains the definitions of all the gRPC calls contained in the `.proto` file. Create a concrete service implementation that derives from this base type and implements the logic for the gRPC calls. For the `greet.proto`, the example described previously, an abstract `GreeterBase` type that contains a virtual `SayHello` method is generated. A concrete implementation `GreeterService` overrides the method and implements the logic handling the gRPC call.

C#

```csharp
public class GreeterService : Greeter.GreeterBase
{
    private readonly ILogger<GreeterService> _logger;
    public GreeterService(ILogger<GreeterService> logger)
    {
        _logger = logger;
```

```
    }

    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

For client-side assets, a concrete client type is generated. The gRPC calls in the `.proto` file are translated into methods on the concrete type, which can be called. For the `greet.proto`, the example described previously, a concrete `GreeterClient` type is generated. Call `GreeterClient.SayHelloAsync` to initiate a gRPC call to the server.

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

By default, server and client assets are generated for each `.proto` file included in the `<Protobuf>` item group. To ensure only the server assets are generated in a server project, the `GrpcServices` attribute is set to `Server`.

XML

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>
```

Similarly, the attribute is set to `Client` in client projects.

Additional resources

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

- Call gRPC services with the .NET client

Create gRPC services and methods

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 document explains how to create gRPC services and methods in C#. Topics include:

- How to define services and methods in `.proto` files.
- Generated code using gRPC C# tooling.
- Implementing gRPC services and methods.

Create new gRPC services

gRPC services with C# introduced gRPC's contract-first approach to API development. Services and messages are defined in `.proto` files. C# tooling then generates code from `.proto` files. For server-side assets, an abstract base type is generated for each service, along with classes for any messages.

The following `.proto` file:

- Defines a `Greeter` service.
- The `Greeter` service defines a `SayHello` call.
- `SayHello` sends a `HelloRequest` message and receives a `HelloReply` message

ProtoBuf

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}
```

```
message HelloReply {
  string message = 1;
}
```

C# tooling generates the C# `GreeterBase` base type:

C#

```
public abstract partial class GreeterBase
{
    public virtual Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        throw new RpcException(new Status(StatusCode.Unimplemented, ""));
    }
}

public class HelloRequest
{
    public string Name { get; set; }
}

public class HelloReply
{
    public string Message { get; set; }
}
```

By default the generated `GreeterBase` doesn't do anything. Its virtual `SayHello` method will return an `UNIMPLEMENTED` error to any clients that call it. For the service to be useful an app must create a concrete implementation of `GreeterBase`:

C#

```
public class GreeterService : GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        return Task.FromResult(new HelloReply { Message = $"Hello
{request.Name}" });
    }
}
```

The `ServerCallContext` gives the context for a server-side call.

The service implementation is registered with the app. If the service is hosted by ASP.NET Core gRPC, it should be added to the routing pipeline with the `MapGrpcService`

method.

C#

```
app.MapGrpcService<GreeterService>();
```

See gRPC services with ASP.NET Core for more information.

Implement gRPC methods

A gRPC service can have different types of methods. How messages are sent and received by a service depends on the type of method defined. The gRPC method types are:

- Unary
- Server streaming
- Client streaming
- Bi-directional streaming

Streaming calls are specified with the `stream` keyword in the `.proto` file. `stream` can be placed on a call's request message, response message, or both.

ProtoBuf

```
syntax = "proto3";

service ExampleService {
  // Unary
  rpc UnaryCall (ExampleRequest) returns (ExampleResponse);

  // Server streaming
  rpc StreamingFromServer (ExampleRequest) returns (stream ExampleResponse);

  // Client streaming
  rpc StreamingFromClient (stream ExampleRequest) returns (ExampleResponse);

  // Bi-directional streaming
  rpc StreamingBothWays (stream ExampleRequest) returns (stream
ExampleResponse);
}
```

Each call type has a different method signature. Overriding generated methods from the abstract base service type in a concrete implementation ensures the correct arguments and return type are used.

Unary method

A unary method has the request message as a parameter, and returns the response. A unary call is complete when the response is returned.

C#

```
public override Task<ExampleResponse> UnaryCall(ExampleRequest request,
    ServerCallContext context)
{
    var response = new ExampleResponse();
    return Task.FromResult(response);
}
```

Unary calls are the most similar to actions on web API controllers. One important difference gRPC methods have from actions is gRPC methods are not able to bind parts of a request to different method arguments. gRPC methods always have one message argument for the incoming request data. Multiple values can still be sent to a gRPC service by adding fields to the request message:

ProtoBuf

```
message ExampleRequest {
    int32 pageIndex = 1;
    int32 pageSize = 2;
    bool isDescending = 3;
}
```

Server streaming method

A server streaming method has the request message as a parameter. Because multiple messages can be streamed back to the caller, `responseStream.WriteAsync` is used to send response messages. A server streaming call is complete when the method returns.

C#

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    for (var i = 0; i < 5; i++)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

The client has no way to send additional messages or data once the server streaming method has started. Some streaming methods are designed to run forever. For continuous streaming methods, a client can cancel the call when it's no longer needed. When cancellation happens the client sends a signal to the server and the ServerCallContext.CancellationToken is raised. The `CancellationToken` token should be used on the server with async methods so that:

- Any asynchronous work is canceled together with the streaming call.
- The method exits quickly.

C#

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    while (!context.CancellationToken.IsCancellationRequested)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1),
context.CancellationToken);
    }
}
```

Client streaming method

A client streaming method starts *without* the method receiving a message. The `requestStream` parameter is used to read messages from the client. A client streaming call is complete when a response message is returned:

C#

```
public override async Task<ExampleResponse> StreamingFromClient(
    IAsyncStreamReader<ExampleRequest> requestStream, ServerCallContext
context)
{
    await foreach (var message in requestStream.ReadAllAsync())
    {
        // ...
    }
    return new ExampleResponse();
}
```

Bi-directional streaming method

A bi-directional streaming method starts *without* the method receiving a message. The `requestStream` parameter is used to read messages from the client. The method can choose to send messages with `responseStream.WriteAsync`. A bi-directional streaming call is complete when the method returns:

C#

```
public override async Task
StreamingBothWays(IAsyncStreamReader<ExampleRequest> requestStream,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    await foreach (var message in requestStream.ReadAllAsync())
    {
        await responseStream.WriteAsync(new ExampleResponse());
    }
}
```

The preceding code:

- Sends a response for each request.
- Is a basic usage of bi-directional streaming.

It is possible to support more complex scenarios, such as reading requests and sending responses simultaneously:

C#

```
public override async Task
StreamingBothWays(IAsyncStreamReader<ExampleRequest> requestStream,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    // Read requests in a background task.
    var readTask = Task.Run(async () =>
    {
        await foreach (var message in requestStream.ReadAllAsync())
        {
            // Process request.
        }
    });

    // Send responses until the client signals that it is complete.
    while (!readTask.IsCompleted)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1),
context.CancellationToken);
```

```
    }
}
```

In a bi-directional streaming method, the client and service can send messages to each other at any time. The best implementation of a bi-directional method varies depending upon requirements.

Access gRPC request headers

A request message is not the only way for a client to send data to a gRPC service. Header values are available in a service using `ServerCallContext.RequestHeaders`.

C#

```
public override Task<ExampleResponse> UnaryCall(ExampleRequest request,
    ServerCallContext context)
{
    var userAgent = context.RequestHeaders.GetValue("user-agent");
    // ...

    return Task.FromResult(new ExampleResponse());
}
```

Multi-threading with gRPC streaming methods

There are important considerations to implementing gRPC streaming methods that use multiple threads.

Reader and writer thread safety

`IAsyncStreamReader<TMessage>` and `IServerStreamWriter<TMessage>` can each be used by only one thread at a time. For a streaming gRPC method, multiple threads can't read new messages with `requestStream.MoveNext()` simultaneously. And multiple threads can't write new messages with `responseStream.WriteAsync(message)` simultaneously.

A safe way to enable multiple threads to interact with a gRPC method is to use the producer-consumer pattern with System.Threading.Channels.

C#

```
public override async Task DownloadResults(DataRequest request,
        IServerStreamWriter<DataResult> responseStream, ServerCallContext
context)
```

```
{
    var channel = Channel.CreateBounded<DataResult>(new
BoundedChannelOptions(capacity: 5));

    var consumerTask = Task.Run(async () =>
    {
        // Consume messages from channel and write to response stream.
        await foreach (var message in channel.Reader.ReadAllAsync())
        {
            await responseStream.WriteAsync(message);
        }
    });

    var dataChunks = request.Value.Chunk(size: 10);

    // Write messages to channel from multiple threads.
    await Task.WhenAll(dataChunks.Select(
        async c =>
        {
            var message = new DataResult { BytesProcessed = c.Length };
            await channel.Writer.WriteAsync(message);
        }));

    // Complete writing and wait for consumer to complete.
    channel.Writer.Complete();
    await consumerTask;
}
```

The preceding gRPC server streaming method:

- Creates a bounded channel for producing and consuming `DataResult` messages.
- Starts a task to read messages from the channel and write them to the response stream.
- Writes messages to the channel from multiple threads.

> ⓘ Note
>
> Bidirectional streaming methods take `IAsyncStreamReader<TMessage>` and `IServerStreamWriter<TMessage>` as arguments. It's safe to use these types on separate threads from each other.

Interacting with a gRPC method after a call ends

A gRPC call ends on the server once the gRPC method exits. The following arguments passed to gRPC methods aren't safe to use after the call has ended:

- `ServerCallContext`

- `IAsyncStreamReader<TMessage>`
- `IServerStreamWriter<TMessage>`

If a gRPC method starts background tasks that use these types, it must complete the tasks before the gRPC method exits. Continuing to use the context, stream reader, or stream writer after the gRPC method exists causes errors and unpredictable behavior.

In the following example, the server streaming method could write to the response stream after the call has finished:

C#

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    _ = Task.Run(async () =>
    {
        for (var i = 0; i < 5; i++)
        {
            await responseStream.WriteAsync(new ExampleResponse());
            await Task.Delay(TimeSpan.FromSeconds(1));
        }
    });

    await PerformLongRunningWorkAsync();
}
```

For the previous example, the solution is to await the write task before exiting the method:

C#

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    var writeTask = Task.Run(async () =>
    {
        for (var i = 0; i < 5; i++)
        {
            await responseStream.WriteAsync(new ExampleResponse());
            await Task.Delay(TimeSpan.FromSeconds(1));
        }
    });

    await PerformLongRunningWorkAsync();
```

```
    await writeTask;
}
```

Additional resources

- gRPC services with C#
- Call gRPC services with the .NET client

Create Protobuf messages for .NET apps

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 James Newton-King and Mark Rendle

gRPC uses Protobuf as its Interface Definition Language (IDL). Protobuf IDL is a language neutral format for specifying the messages sent and received by gRPC services. Protobuf messages are defined in `.proto` files. This document explains how Protobuf concepts map to .NET.

Protobuf messages

Messages are the main data transfer object in Protobuf. They are conceptually similar to .NET classes.

ProtoBuf

```
syntax = "proto3";

option csharp_namespace = "Contoso.Messages";

message Person {
    int32 id = 1;
    string first_name = 2;
    string last_name = 3;
}
```

The preceding message definition specifies three fields as name-value pairs. Like properties on .NET types, each field has a name and a type. The field type can be a Protobuf scalar value type, for example `int32`, or another message.

The Protobuf style guide recommends using `underscore_separated_names` for field names. New Protobuf messages created for .NET apps should follow the Protobuf style guidelines. .NET tooling automatically generates .NET types that use .NET naming

standards. For example, a `first_name` Protobuf field generates a `FirstName` .NET property.

In addition to a name, each field in the message definition has a unique number. Field numbers are used to identify fields when the message is serialized to Protobuf. Serializing a small number is faster than serializing the entire field name. Because field numbers identify a field it is important to take care when changing them. For more information about changing Protobuf messages see Versioning gRPC services.

When an app is built the Protobuf tooling generates .NET types from `.proto` files. The `Person` message generates a .NET class:

C#

```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

For more information about Protobuf messages see the Protobuf language guide.

Scalar Value Types

Protobuf supports a range of native scalar value types. The following table lists them all with their equivalent C# type:

Expand table

Protobuf type	C# type
`double`	`double`
`float`	`float`
`int32`	`int`
`int64`	`long`
`uint32`	`uint`
`uint64`	`ulong`
`sint32`	`int`

Protobuf type	C# type
`sint64`	`long`
`fixed32`	`uint`
`fixed64`	`ulong`
`sfixed32`	`int`
`sfixed64`	`long`
`bool`	`bool`
`string`	`string`
`bytes`	`ByteString`

Scalar values always have a default value and can't be set to `null`. This constraint includes `string` and `ByteString` which are C# classes. `string` defaults to an empty string value and `ByteString` defaults to an empty bytes value. Attempting to set them to `null` throws an error.

Nullable wrapper types can be used to support null values.

Dates and times

The native scalar types don't provide for date and time values, equivalent to .NET's DateTimeOffset, DateTime, and TimeSpan. These types can be specified by using some of Protobuf's *Well-Known Types* extensions. These extensions provide code generation and runtime support for complex field types across the supported platforms.

The following table shows the date and time types:

Expand table

.NET type	Protobuf Well-Known Type
`DateTimeOffset`	`google.protobuf.Timestamp`
`DateTime`	`google.protobuf.Timestamp`
`TimeSpan`	`google.protobuf.Duration`

```
syntax = "proto3";

import "google/protobuf/duration.proto";
import "google/protobuf/timestamp.proto";

message Meeting {
    string subject = 1;
    google.protobuf.Timestamp start = 2;
    google.protobuf.Duration duration = 3;
}
```

The generated properties in the C# class aren't the .NET date and time types. The properties use the `Timestamp` and `Duration` classes in the `Google.Protobuf.WellKnownTypes` namespace. These classes provide methods for converting to and from `DateTimeOffset`, `DateTime`, and `TimeSpan`.

C#

```
// Create Timestamp and Duration from .NET DateTimeOffset and TimeSpan.
var meeting = new Meeting
{
    Time = Timestamp.FromDateTimeOffset(meetingTime), // also FromDateTime()
    Duration = Duration.FromTimeSpan(meetingLength)
};

// Convert Timestamp and Duration to .NET DateTimeOffset and TimeSpan.
var time = meeting.Time.ToDateTimeOffset();
var duration = meeting.Duration?.ToTimeSpan();
```

> ⓘ **Note**
>
> The `Timestamp` type works with UTC times. `DateTimeOffset` values always have an offset of zero, and the `DateTime.Kind` property is always `DateTimeKind.Utc`.

Nullable types

The Protobuf code generation for C# uses the native types, such as `int` for `int32`. So the values are always included and can't be `null`.

For values that require explicit `null`, such as using `int?` in C# code, Protobuf's Well-Known Types include wrappers that are compiled to nullable C# types. To use them, import `wrappers.proto` into your `.proto` file, like the following code:

ProtoBuf

```
syntax = "proto3";

import "google/protobuf/wrappers.proto";

message Person {
    // ...
    google.protobuf.Int32Value age = 5;
}
```

`wrappers.proto` types aren't exposed in generated properties. Protobuf automatically maps them to appropriate .NET nullable types in C# messages. For example, a `google.protobuf.Int32Value` field generates an `int?` property. Reference type properties like `string` and `ByteString` are unchanged except `null` can be assigned to them without error.

The following table shows the complete list of wrapper types with their equivalent C# type:

Expand table

C# type	Well-Known Type wrapper
`bool?`	`google.protobuf.BoolValue`
`double?`	`google.protobuf.DoubleValue`
`float?`	`google.protobuf.FloatValue`
`int?`	`google.protobuf.Int32Value`
`long?`	`google.protobuf.Int64Value`
`uint?`	`google.protobuf.UInt32Value`
`ulong?`	`google.protobuf.UInt64Value`
`string`	`google.protobuf.StringValue`
`ByteString`	`google.protobuf.BytesValue`

Bytes

Binary payloads are supported in Protobuf with the `bytes` scalar value type. A generated property in C# uses `ByteString` as the property type.

Use `ByteString.CopyFrom(byte[] data)` to create a new instance from a byte array:

C#

```
var data = await File.ReadAllBytesAsync(path);

var payload = new PayloadResponse();
payload.Data = ByteString.CopyFrom(data);
```

`ByteString` data is accessed directly using `ByteString.Span` or `ByteString.Memory`. Or call `ByteString.ToByteArray()` to convert an instance back into a byte array:

C#

```
var payload = await client.GetPayload(new PayloadRequest());

await File.WriteAllBytesAsync(path, payload.Data.ToByteArray());
```

Decimals

Protobuf doesn't natively support the .NET `decimal` type, just `double` and `float`. There's an ongoing discussion in the Protobuf project about the possibility of adding a standard decimal type to the Well-Known Types, with platform support for languages and frameworks that support it. Nothing has been implemented yet.

It's possible to create a message definition to represent the `decimal` type that works for safe serialization between .NET clients and servers. But developers on other platforms would have to understand the format being used and implement their own handling for it.

Creating a custom decimal type for Protobuf

ProtoBuf

```
package CustomTypes;

// Example: 12345.6789 -> { units = 12345, nanos = 678900000 }
message DecimalValue {

    // Whole units part of the amount
    int64 units = 1;

    // Nano units of the amount (10^-9)
    // Must be same sign as units
    sfixed32 nanos = 2;
}
```

The `nanos` field represents values from `0.999_999_999` to `-0.999_999_999`. For example, the `decimal` value `1.5m` would be represented as `{ units = 1, nanos = 500_000_000 }`. This is why the `nanos` field in this example uses the `sfixed32` type, which encodes more efficiently than `int32` for larger values. If the `units` field is negative, the `nanos` field should also be negative.

> **Note**
>
> Additional algorithms are available for encoding `decimal` values as byte strings. The algorithm used by `DecimalValue`:
>
> - Is easy to understand.
> - Isn't affected by big-endian or little-endian on different platforms.
> - Supports decimal numbers ranging from positive `9,223,372,036,854,775,807.999999999` to negative `9,223,372,036,854,775,808.999999999` with a maximum precision of nine decimal places, which isn't the full range of a `decimal`.

Conversion between this type and the BCL `decimal` type might be implemented in C# like this:

C#

```
namespace CustomTypes
{
    public partial class DecimalValue
    {
        private const decimal NanoFactor = 1_000_000_000;
        public DecimalValue(long units, int nanos)
        {
            Units = units;
            Nanos = nanos;
        }

        public static implicit operator decimal(CustomTypes.DecimalValue
grpcDecimal)
        {
            return grpcDecimal.Units + grpcDecimal.Nanos / NanoFactor;
        }

        public static implicit operator CustomTypes.DecimalValue(decimal
value)
        {
            var units = decimal.ToInt64(value);
            var nanos = decimal.ToInt32((value - units) * NanoFactor);
            return new CustomTypes.DecimalValue(units, nanos);
```

```
        }
    }
}
```

The preceding code:

- Adds a partial class for `DecimalValue`. The partial class is combined with `DecimalValue` generated from the `.proto` file. The generated class declares the `Units` and `Nanos` properties.
- Has implicit operators for converting between `DecimalValue` and the BCL `decimal` type.

Collections

Lists

Lists in Protobuf are specified by using the `repeated` prefix keyword on a field. The following example shows how to create a list:

ProtoBuf

```
message Person {
    // ...
    repeated string roles = 8;
}
```

In the generated code, `repeated` fields are represented by the `Google.Protobuf.Collections.RepeatedField<T>` generic type.

C#

```
public class Person
{
    // ...
    public RepeatedField<string> Roles { get; }
}
```

`RepeatedField<T>` implements IList<T>. So you can use LINQ queries or convert it to an array or a list. `RepeatedField<T>` properties don't have a public setter. Items should be added to the existing collection.

C#

```
var person = new Person();

// Add one item.
person.Roles.Add("user");

// Add all items from another collection.
var roles = new [] { "admin", "manager" };
person.Roles.Add(roles);
```

Dictionaries

The .NET IDictionary<TKey,TValue> type is represented in Protobuf using `map<key_type, value_type>`.

ProtoBuf

```
message Person {
    // ...
    map<string, string> attributes = 9;
}
```

In generated .NET code, `map` fields are represented by the `Google.Protobuf.Collections.MapField<TKey, TValue>` generic type. `MapField<TKey, TValue>` implements IDictionary<TKey,TValue>. Like `repeated` properties, `map` properties don't have a public setter. Items should be added to the existing collection.

C#

```
var person = new Person();

// Add one item.
person.Attributes["created_by"] = "James";

// Add all items from another collection.
var attributes = new Dictionary<string, string>
{
    ["last_modified"] = DateTime.UtcNow.ToString()
};
person.Attributes.Add(attributes);
```

Unstructured and conditional messages

Protobuf is a contract-first messaging format. An app's messages, including its fields and types, must be specified in `.proto` files when the app is built. Protobuf's contract-

first design is great at enforcing message content but can limit scenarios where a strict contract isn't required:

- Messages with unknown payloads. For example, a message with a field that could contain any message.
- Conditional messages. For example, a message returned from a gRPC service might be a success result or an error result.
- Dynamic values. For example, a message with a field that contains an unstructured collection of values, similar to JSON.

Protobuf offers language features and types to support these scenarios.

Any

The `Any` type lets you use messages as embedded types without having their `.proto` definition. To use the `Any` type, import `any.proto`.

ProtoBuf

```
import "google/protobuf/any.proto";

message Status {
    string message = 1;
    google.protobuf.Any detail = 2;
}
```

C#

```
// Create a status with a Person message set to detail.
var status = new ErrorStatus();
status.Detail = Any.Pack(new Person { FirstName = "James" });

// Read Person message from detail.
if (status.Detail.Is(Person.Descriptor))
{
    var person = status.Detail.Unpack<Person>();
    // ...
}
```

Oneof

`oneof` fields are a language feature. The compiler handles the `oneof` keyword when it generates the message class. Using `oneof` to specify a response message that could either return a `Person` or `Error` might look like this:

ProtoBuf

```
message Person {
    // ...
}

message Error {
    // ...
}

message ResponseMessage {
  oneof result {
    Error error = 1;
    Person person = 2;
  }
}
```

Fields within the `oneof` set must have unique field numbers in the overall message declaration.

When using `oneof`, the generated C# code includes an enum that specifies which of the fields has been set. You can test the enum to find which field is set. Fields that aren't set return `null` or the default value, rather than throwing an exception.

C#

```
var response = await client.GetPersonAsync(new RequestMessage());

switch (response.ResultCase)
{
    case ResponseMessage.ResultOneofCase.Person:
        HandlePerson(response.Person);
        break;
    case ResponseMessage.ResultOneofCase.Error:
        HandleError(response.Error);
        break;
    default:
        throw new ArgumentException("Unexpected result.");
}
```

Value

The `Value` type represents a dynamically typed value. It can be either `null`, a number, a string, a boolean, a dictionary of values (`Struct`), or a list of values (`ValueList`). `Value` is a Protobuf Well-Known Type that uses the previously discussed `oneof` feature. To use the `Value` type, import `struct.proto`.

ProtoBuf

```
import "google/protobuf/struct.proto";

message Status {
    // ...
    google.protobuf.Value data = 3;
}
```

C#

```
// Create dynamic values.
var status = new Status();
status.Data = Value.ForStruct(new Struct
{
    Fields =
    {
        ["enabled"] = Value.ForBool(true),
        ["metadata"] = Value.ForList(
            Value.ForString("value1"),
            Value.ForString("value2"))
    }
});

// Read dynamic values.
switch (status.Data.KindCase)
{
    case Value.KindOneofCase.StructValue:
        foreach (var field in status.Data.StructValue.Fields)
        {
            // Read struct fields...
        }
        break;
    // ...
}
```

Using `Value` directly can be verbose. An alternative way to use `Value` is with Protobuf's built-in support for mapping messages to JSON. Protobuf's `JsonFormatter` and `JsonWriter` types can be used with any Protobuf message. `Value` is particularly well suited to being converted to and from JSON.

This is the JSON equivalent of the previous code:

C#

```
// Create dynamic values from JSON.
var status = new Status();
status.Data = Value.Parser.ParseJson(@"{
    ""enabled"": true,
    ""metadata"": [ ""value1"", ""value2"" ]
```

```
}");

// Convert dynamic values to JSON.
// JSON can be read with a library like System.Text.Json or Newtonsoft.Json
var json = JsonFormatter.Default.Format(status.Data);
var document = JsonDocument.Parse(json);
```

Additional resources

- Protobuf language guide
- Versioning gRPC services

Versioning gRPC services

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 James Newton-King

New features added to an app can require gRPC services provided to clients to change, sometimes in unexpected and breaking ways. When gRPC services change:

- Consideration should be given on how changes impact clients.
- A versioning strategy to support changes should be implemented.

Backwards compatibility

The gRPC protocol is designed to support services that change over time. Generally, additions to gRPC services and methods are non-breaking. Non-breaking changes allow existing clients to continue working without changes. Changing or deleting gRPC services are breaking changes. When gRPC services have breaking changes, clients using that service have to be updated and redeployed.

Making non-breaking changes to a service has a number of benefits:

- Existing clients continue to run.
- Avoids work involved with notifying clients of breaking changes, and updating them.
- Only one version of the service needs to be documented and maintained.

Non-breaking changes

These changes are non-breaking at a gRPC protocol level and .NET binary level.

- **Adding a new service**
- **Adding a new method to a service**

- **Adding a field to a request message** - Fields added to a request message are deserialized with the default value on the server when not set. To be a non-breaking change, the service must succeed when the new field isn't set by older clients.
- **Adding a field to a response message** - If an older client hasn't been updated with the new field, the value is deserialized into the response message's unknown fields collection.
- **Adding a value to an enum** - Enums are serialized as a numeric value. New enum values are deserialized on the client to the enum value without an enum name. To be a non-breaking change, older clients must run correctly when receiving the new enum value.

Binary breaking changes

The following changes are non-breaking at a gRPC protocol level, but the client needs to be updated if it upgrades to the latest `.proto` contract or client .NET assembly. Binary compatibility is important if you plan to publish a gRPC library to NuGet.

- **Removing a field** - Values from a removed field are deserialized to a message's unknown fields. This isn't a gRPC protocol breaking change, but the client needs to be updated if it upgrades to the latest contract. It's important that a removed field number isn't accidentally reused in the future. To ensure this doesn't happen, specify deleted field numbers and names on the message using Protobuf's reserved keyword.
- **Renaming a message** - Message names aren't typically sent on the network, so this isn't a gRPC protocol breaking change. The client will need to be updated if it upgrades to the latest contract. One situation where message names **are** sent on the network is with Any fields, when the message name is used to identify the message type.
- **Nesting or unnesting a message** - Message types can be nested. Nesting or unnesting a message changes its message name. Changing how a message type is nested has the same impact on compatibility as renaming.
- **Changing csharp_namespace** - Changing `csharp_namespace` will change the namespace of generated .NET types. This isn't a gRPC protocol breaking change, but the client needs to be updated if it upgrades to the latest contract.

Protocol breaking changes

The following items are protocol and binary breaking changes:

- **Renaming a field** - With Protobuf content, the field names are only used in generated code. The field number is used to identify fields on the network. Renaming a field isn't a protocol breaking change for Protobuf. However, if a server is using JSON content then renaming a field is a breaking change.
- **Changing a field data type** - Changing a field's data type to an incompatible type will cause errors when deserializing the message. Even if the new data type is compatible, it's likely the client needs to be updated to support the new type if it upgrades to the latest contract.
- **Changing a field number** - With Protobuf payloads, the field number is used to identify fields on the network.
- **Renaming a package, service or method** - gRPC uses the package name, service name, and method name to build the URL. The client gets an *UNIMPLEMENTED* status from the server.
- **Removing a service or method** - The client gets an *UNIMPLEMENTED* status from the server when calling the removed method.

Behavior breaking changes

When making non-breaking changes, you must also consider whether older clients can continue working with the new service behavior. For example, adding a new field to a request message:

- Isn't a protocol breaking change.
- Returning an error status on the server if the new field isn't set makes it a breaking change for old clients.

Behavior compatibility is determined by your app-specific code.

Version number services

Services should strive to remain backwards compatible with old clients. Eventually changes to your app may require breaking changes. Breaking old clients and forcing them to be updated along with your service isn't a good user experience. A way to maintain backwards compatibility while making breaking changes is to publish multiple versions of a service.

gRPC supports an optional package specifier, which functions much like a .NET namespace. In fact, the `package` will be used as the .NET namespace for generated .NET types if `option csharp_namespace` is not set in the `.proto` file. The package can be used to specify a version number for your service and its messages:

ProtoBuf

```
syntax = "proto3";

package greet.v1;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

The package name is combined with the service name to identify a service address. A service address allows multiple versions of a service to be hosted side-by-side:

- `greet.v1.Greeter`
- `greet.v2.Greeter`

Implementations of the versioned service are registered in `Startup.cs`:

C#

```
app.UseEndpoints(endpoints =>
{
    // Implements greet.v1.Greeter
    endpoints.MapGrpcService<GreeterServiceV1>();

    // Implements greet.v2.Greeter
    endpoints.MapGrpcService<GreeterServiceV2>();
});
```

Including a version number in the package name gives you the opportunity to publish a *v2* version of your service with breaking changes, while continuing to support older clients who call the *v1* version. Once clients have updated to use the *v2* service, you can choose to remove the old version. When planning to publish multiple versions of a service:

- Avoid breaking changes if reasonable.
- Don't update the version number unless making breaking changes.
- Do update the version number when you make breaking changes.

Publishing multiple versions of a service duplicates it. To reduce duplication, consider moving business logic from the service implementations to a centralized location that can be reused by the old and new implementations:

C#

```
using Greet.V1;
using Grpc.Core;
using System.Threading.Tasks;

namespace Services
{
    public class GreeterServiceV1 : Greeter.GreeterBase
    {
        private readonly IGreeter _greeter;
        public GreeterServiceV1(IGreeter greeter)
        {
            _greeter = greeter;
        }

        public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
        {
            return Task.FromResult(new HelloReply
            {
                Message = _greeter.GetHelloMessage(request.Name)
            });
        }
    }
}
```

Services and messages generated with different package names are **different .NET types**. Moving business logic to a centralized location requires mapping messages to common types.

Additional resources

- Create Protobuf messages for .NET apps

Test gRPC services 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

Testing is an important aspect of building stable and maintainable software. This article discusses how to test ASP.NET Core gRPC services.

There are three common approaches for testing gRPC services:

- **Unit testing**: Test gRPC services directly from a unit testing library.
- **Integration testing**: The gRPC app is hosted in TestServer, an in-memory test server from the Microsoft.AspNetCore.TestHost package. gRPC services are tested by calling them using a gRPC client from a unit testing library.
- **Manual testing**: Test gRPC servers with ad hoc calls. For information about how to use command-line and UI tooling with gRPC services, see Test gRPC services with gRPCurl and gRPCui in ASP.NET Core.

In unit testing, only the gRPC service is involved. Dependencies injected into the service must be mocked. In integration testing, the gRPC service and its auxiliary infrastructure are part of the test. This includes app startup, dependency injection, routing and authentication, and authorization.

Example testable service

To demonstrate service tests, review the following service in the sample app.

View or download sample code (how to download)

The `TesterService` returns greetings using gRPC's four method types.

```
C#
```

```
public class TesterService : Tester.TesterBase
{
    private readonly IGreeter _greeter;

    public TesterService(IGreeter greeter)
    {
        _greeter = greeter;
    }

    public override Task<HelloReply> SayHelloUnary(HelloRequest request,
        ServerCallContext context)
    {
        var message = _greeter.Greet(request.Name);
        return Task.FromResult(new HelloReply { Message = message });
    }

    public override async Task SayHelloServerStreaming(HelloRequest request,
        IServerStreamWriter<HelloReply> responseStream, ServerCallContext
context)
    {
        var i = 0;
        while (!context.CancellationToken.IsCancellationRequested)
        {
            var message = _greeter.Greet($"{request.Name} {++i}");
            await responseStream.WriteAsync(new HelloReply { Message =
message });

            await Task.Delay(1000);
        }
    }

    public override async Task<HelloReply> SayHelloClientStreaming(
        IAsyncStreamReader<HelloRequest> requestStream, ServerCallContext
context)
    {
        var names = new List<string>();

        await foreach (var request in requestStream.ReadAllAsync())
        {
            names.Add(request.Name);
        }

        var message = _greeter.Greet(string.Join(", ", names));
        return new HelloReply { Message = message };
    }

    public override async Task SayHelloBidirectionalStreaming(
        IAsyncStreamReader<HelloRequest> requestStream,
        IServerStreamWriter<HelloReply> responseStream,
        ServerCallContext context)
    {
        await foreach (var request in requestStream.ReadAllAsync())
        {
            await responseStream.WriteAsync(
```

```
                new HelloReply { Message = _greeter.Greet(request.Name) });
        }
    }
}
```

The preceding gRPC service:

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

Unit test gRPC services

A unit test library can directly test gRPC services by calling its methods. Unit tests test a gRPC service in isolation.

C#

```
[Fact]
public async Task SayHelloUnaryTest()
{
    // Arrange
    var mockGreeter = new Mock<IGreeter>();
    mockGreeter.Setup(
        m => m.Greet(It.IsAny<string>())).Returns((string s) => $"Hello
{s}");
    var service = new TesterService(mockGreeter.Object);

    // Act
    var response = await service.SayHelloUnary(
        new HelloRequest { Name = "Joe" }, TestServerCallContext.Create());

    // Assert
    mockGreeter.Verify(v => v.Greet("Joe"));
    Assert.Equal("Hello Joe", response.Message);
}
```

The preceding unit test:

- Mocks `IGreeter` using Moq.
- Executes the `SayHelloUnary` method with a request message and a `ServerCallContext`. All service methods have a `ServerCallContext` argument. In

this test, the type is provided using the `TestServerCallContext.Create()` helper method. This helper method is included in the sample code.

- Makes assertions:
  - Verifies the request name is passed to `IGreeter`.
  - The service returns the expected reply message.

Unit test `HttpContext` in gRPC methods

gRPC methods can access a request's HttpContext using the `ServerCallContext.GetHttpContext` extension method. To unit test a method that uses `HttpContext`, the context must be configured in test setup. If HttpContext isn't configured then `GetHttpContext` returns `null`.

To configure a `HttpContext` during test setup, create a new instance and add it to `ServerCallContext.UserState` collection using the `__HttpContext` key.

C#

```
var httpContext = new DefaultHttpContext();

var serverCallContext = TestServerCallContext.Create();
serverCallContext.UserState["__HttpContext"] = httpContext;
```

Execute service methods with this call context to use the configured `HttpContext` instance.

Integration test gRPC services

Integration tests evaluate an app's components on a broader level than unit tests. The gRPC app is hosted in TestServer, an in-memory test server from the Microsoft.AspNetCore.TestHost package.

A unit test library starts the gRPC app and then gRPC services are tested using the gRPC client.

The sample code contains infrastructure to make integration testing possible:

- The `GrpcTestFixture<TStartup>` class configures the ASP.NET Core host and starts the gRPC app in an in-memory test server.
- The `IntegrationTestBase` class is the base type that integration tests inherit from. It contains the fixture state and APIs for creating a gRPC client to call the gRPC app.

C#

```
[Fact]
public async Task SayHelloUnaryTest()
{
    // Arrange
    var client = new Tester.TesterClient(Channel);

    // Act
    var response = await client.SayHelloUnaryAsync(new HelloRequest { Name =
"Joe" });

    // Assert
    Assert.Equal("Hello Joe", response.Message);
}
```

The preceding integration test:

- Creates a gRPC client using the channel provided by `IntegrationTestBase`. This type is included in the sample code.
- Calls the `SayHelloUnary` method using the gRPC client.
- Asserts the service returns the expected reply message.

Inject mock dependencies

Use `ConfigureWebHost` on the fixture to override dependencies. Overriding dependencies is useful when an external dependency is unavailable in the test environment. For example, an app that uses an external payment gateway shouldn't call the external dependency when executing tests. Instead, use a mock gateway for the test.

C#

```
public MockedGreeterServiceTests(GrpcTestFixture<Startup> fixture,
    ITestOutputHelper outputHelper) : base(fixture, outputHelper)
{
    var mockGreeter = new Mock<IGreeter>();
    mockGreeter.Setup(
        m => m.Greet(It.IsAny<string>())).Returns((string s) =>
        {
            if (string.IsNullOrEmpty(s))
            {
                throw new ArgumentException("Name not provided.");
            }
            return $"Test {s}";
        });

    Fixture.ConfigureWebHost(builder =>
    {
        builder.ConfigureServices(
```

```
            services => services.AddSingleton(mockGreeter.Object));
    });
}

[Fact]
public async Task SayHelloUnaryTest_MockGreeter_Success()
{
    // Arrange
    var client = new Tester.TesterClient(Channel);

    // Act
    var response = await client.SayHelloUnaryAsync(
        new HelloRequest { Name = "Joe" });

    // Assert
    Assert.Equal("Test Joe", response.Message);
}
```

The preceding integration test:

- In the test class's (`MockedGreeterServiceTests`) constructor:
  - Mocks `IGreeter` using Moq.
  - Overrides the `IGreeter` registered with dependency injection using `ConfigureWebHost`.
- Calls the `SayHelloUnary` method using the gRPC client.
- Asserts the expected reply message based on the mock `IGreeter` instance.

Additional resources

- Test gRPC services with gRPCurl and gRPCui in ASP.NET Core
- Mock gRPC client in tests

Call gRPC services with the .NET client

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.

A .NET gRPC client library is available in the Grpc.Net.Client NuGet package. This document explains how to:

- Configure a gRPC client to call gRPC services.
- Make gRPC calls to unary, server streaming, client streaming, and bi-directional streaming methods.

Configure gRPC client

gRPC clients are concrete client types that are generated from .proto files. The concrete gRPC client has methods that translate to the gRPC service in the `.proto` file. For example, a service called `Greeter` generates a `GreeterClient` type with methods to call the service.

A gRPC client is created from a channel. Start by using `GrpcChannel.ForAddress` to create a channel, and then use the channel to create a gRPC client:

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greet.GreeterClient(channel);
```

A channel represents a long-lived connection to a gRPC service. When a channel is created, it's configured with options related to calling a service. For example, the `HttpClient` used to make calls, the maximum send and receive message size, and logging can be specified on `GrpcChannelOptions` and used with `GrpcChannel.ForAddress`. For a complete list of options, see client configuration options.

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");

var greeterClient = new Greet.GreeterClient(channel);
var counterClient = new Count.CounterClient(channel);

// Use clients to call gRPC services
```

Configure TLS

A gRPC client must use the same connection-level security as the called service. gRPC client Transport Layer Security (TLS) is configured when the gRPC channel is created. A gRPC client throws an error when it calls a service and the connection-level security of the channel and service don't match.

To configure a gRPC channel to use TLS, ensure the server address starts with `https`. For example, `GrpcChannel.ForAddress("https://localhost:5001")` uses HTTPS protocol. The gRPC channel automatically negotiates a connection secured by TLS and uses a secure connection to make gRPC calls.

> **Tip**
>
> gRPC supports client certificate authentication over TLS. For information on configuring client certificates with a gRPC channel, see **Authentication and authorization in gRPC for ASP.NET Core**.

To call unsecured gRPC services, ensure the server address starts with `http`. For example, `GrpcChannel.ForAddress("http://localhost:5000")` uses HTTP protocol. In .NET Core 3.1, additional configuration is required to call insecure gRPC services with the .NET client.

Client performance

Channel and client performance and usage:

- Creating a channel can be an expensive operation. Reusing a channel for gRPC calls provides performance benefits.
- A channel manages connections to the server. If the connection is closed or lost, the channel automatically reconnects the next time a gRPC call is made.
- gRPC clients are created with channels. gRPC clients are lightweight objects and don't need to be cached or reused.

- Multiple gRPC clients can be created from a channel, including different types of clients.
- A channel and clients created from the channel can safely be used by multiple threads.
- Clients created from the channel can make multiple simultaneous calls.

`GrpcChannel.ForAddress` isn't the only option for creating a gRPC client. If calling gRPC services from an ASP.NET Core app, consider gRPC client factory integration. gRPC integration with `HttpClientFactory` offers a centralized alternative to creating gRPC clients.

> **Note**
>
> Calling gRPC over HTTP/2 with `Grpc.Net.Client` is currently not supported on Xamarin. We are working to improve HTTP/2 support in a future Xamarin release. Grpc.Core and gRPC-Web are viable alternatives that work today.

Make gRPC calls

A gRPC call is initiated by calling a method on the client. The gRPC client will handle message serialization and addressing the gRPC call to the correct service.

gRPC has different types of methods. How the client is used to make a gRPC call depends on the type of method called. The gRPC method types are:

- Unary
- Server streaming
- Client streaming
- Bi-directional streaming

Unary call

A unary call starts with the client sending a request message. A response message is returned when the service finishes.

C#

```
var client = new Greet.GreeterClient(channel);
var response = await client.SayHelloAsync(new HelloRequest { Name = "World"
});
```

```
Console.WriteLine("Greeting: " + response.Message);
// Greeting: Hello World
```

Each unary service method in the `.proto` file will result in two .NET methods on the concrete gRPC client type for calling the method: an asynchronous method and a blocking method. For example, on `GreeterClient` there are two ways of calling `SayHello`:

- `GreeterClient.SayHelloAsync` - calls `Greeter.SayHello` service asynchronously. Can be awaited.
- `GreeterClient.SayHello` - calls `Greeter.SayHello` service and blocks until complete. Don't use in asynchronous code.

Server streaming call

A server streaming call starts with the client sending a request message. `ResponseStream.MoveNext()` reads messages streamed from the service. The server streaming call is complete when `ResponseStream.MoveNext()` returns `false`.

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHellos(new HelloRequest { Name = "World" });

while (await call.ResponseStream.MoveNext())
{
    Console.WriteLine("Greeting: " + call.ResponseStream.Current.Message);
    // "Greeting: Hello World" is written multiple times
}
```

When using C# 8 or later, the `await foreach` syntax can be used to read messages. The `IAsyncStreamReader<T>.ReadAllAsync()` extension method reads all messages from the response stream:

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHellos(new HelloRequest { Name = "World" });

await foreach (var response in call.ResponseStream.ReadAllAsync())
{
    Console.WriteLine("Greeting: " + response.Message);
    // "Greeting: Hello World" is written multiple times
}
```

The type returned from starting a server streaming call implements `IDisposable`. Always dispose a streaming call to ensure it's stopped and all resources are cleaned up.

Client streaming call

A client streaming call starts *without* the client sending a message. The client can choose to send messages with `RequestStream.WriteAsync`. When the client has finished sending messages, `RequestStream.CompleteAsync()` should be called to notify the service. The call is finished when the service returns a response message.

C#

```
var client = new Counter.CounterClient(channel);
using var call = client.AccumulateCount();

for (var i = 0; i < 3; i++)
{
    await call.RequestStream.WriteAsync(new CounterRequest { Count = 1 });
}
await call.RequestStream.CompleteAsync();

var response = await call;
Console.WriteLine($"Count: {response.Count}");
// Count: 3
```

The type returned from starting a client streaming call implements `IDisposable`. Always dispose a streaming call to ensure it's stopped and all resources are cleaned up.

Bi-directional streaming call

A bi-directional streaming call starts *without* the client sending a message. The client can choose to send messages with `RequestStream.WriteAsync`. Messages streamed from the service are accessible with `ResponseStream.MoveNext()` or `ResponseStream.ReadAllAsync()`. The bi-directional streaming call is complete when the `ResponseStream` has no more messages.

C#

```
var client = new Echo.EchoClient(channel);
using var call = client.Echo();

Console.WriteLine("Starting background task to receive messages");
var readTask = Task.Run(async () =>
{
    await foreach (var response in call.ResponseStream.ReadAllAsync())
```

```
    {
        Console.WriteLine(response.Message);
        // Echo messages sent to the service
    }
});

Console.WriteLine("Starting to send messages");
Console.WriteLine("Type a message to echo then press enter.");
while (true)
{
    var result = Console.ReadLine();
    if (string.IsNullOrEmpty(result))
    {
        break;
    }

    await call.RequestStream.WriteAsync(new EchoMessage { Message = result
});
}

Console.WriteLine("Disconnecting");
await call.RequestStream.CompleteAsync();
await readTask;
```

For best performance, and to avoid unnecessary errors in the client and service, try to complete bi-directional streaming calls gracefully. A bi-directional call completes gracefully when the server has finished reading the request stream and the client has finished reading the response stream. The preceding sample call is one example of a bi-directional call that ends gracefully. In the call, the client:

1. Starts a new bi-directional streaming call by calling `EchoClient.Echo`.
2. Creates a background task to read messages from the service using `ResponseStream.ReadAllAsync()`.
3. Sends messages to the server with `RequestStream.WriteAsync`.
4. Notifies the server it has finished sending messages with `RequestStream.CompleteAsync()`.
5. Waits until the background task has read all incoming messages.

During a bi-directional streaming call, the client and service can send messages to each other at any time. The best client logic for interacting with a bi-directional call varies depending upon the service logic.

The type returned from starting a bi-directional streaming call implements `IDisposable`. Always dispose a streaming call to ensure it's stopped and all resources are cleaned up.

Access gRPC headers

gRPC calls return response headers. HTTP response headers pass name/value metadata about a call that isn't related the returned message.

Headers are accessible using `ResponseHeadersAsync`, which returns a collection of metadata. Headers are typically returned with the response message; therefore, you must await them.

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHelloAsync(new HelloRequest { Name = "World" });

var headers = await call.ResponseHeadersAsync;
var myValue = headers.GetValue("my-trailer-name");

var response = await call.ResponseAsync;
```

`ResponseHeadersAsync` usage:

- Must await the result of `ResponseHeadersAsync` to get the headers collection.
- Doesn't have to be accessed before `ResponseAsync` (or the response stream when streaming). If a response has been returned, then `ResponseHeadersAsync` returns headers instantly.
- Will throw an exception if there was a connection or server error and headers weren't returned for the gRPC call.

Access gRPC trailers

gRPC calls may return response trailers. Trailers are used to provide name/value metadata about a call. Trailers provide similar functionality to HTTP headers, but are received at the end of the call.

Trailers are accessible using `GetTrailers()`, which returns a collection of metadata. Trailers are returned after the response is complete. Therefore, you must await all response messages before accessing the trailers.

Unary and client streaming calls must await `ResponseAsync` before calling `GetTrailers()`:

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHelloAsync(new HelloRequest { Name = "World" });
var response = await call.ResponseAsync;

Console.WriteLine("Greeting: " + response.Message);
```

```
// Greeting: Hello World

var trailers = call.GetTrailers();
var myValue = trailers.GetValue("my-trailer-name");
```

Server and bidirectional streaming calls must finish awaiting the response stream before calling `GetTrailers()`:

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHellos(new HelloRequest { Name = "World" });

await foreach (var response in call.ResponseStream.ReadAllAsync())
{
    Console.WriteLine("Greeting: " + response.Message);
    // "Greeting: Hello World" is written multiple times
}

var trailers = call.GetTrailers();
var myValue = trailers.GetValue("my-trailer-name");
```

Trailers are also accessible from `RpcException`. A service may return trailers together with a non-OK gRPC status. In this situation, the trailers are retrieved from the exception thrown by the gRPC client:

C#

```
var client = new Greet.GreeterClient(channel);
string myValue = null;

try
{
    using var call = client.SayHelloAsync(new HelloRequest { Name = "World"
});
    var response = await call.ResponseAsync;

    Console.WriteLine("Greeting: " + response.Message);
    // Greeting: Hello World

    var trailers = call.GetTrailers();
    myValue = trailers.GetValue("my-trailer-name");
}
catch (RpcException ex)
{
    var trailers = ex.Trailers;
    myValue = trailers.GetValue("my-trailer-name");
}
```

Configure deadline

Configuring a gRPC call deadline is recommended because it provides an upper limit on how long a call can run for. It stops misbehaving services from running forever and exhausting server resources. Deadlines are a useful tool for building reliable apps.

Configure `CallOptions.Deadline` to set a deadline for a gRPC call:

C#

```
var client = new Greet.GreeterClient(channel);

try
{
    var response = await client.SayHelloAsync(
        new HelloRequest { Name = "World" },
        deadline: DateTime.UtcNow.AddSeconds(5));

    // Greeting: Hello World
    Console.WriteLine("Greeting: " + response.Message);
}
catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded)
{
    Console.WriteLine("Greeting timeout.");
}
```

For more information, see Reliable gRPC services with deadlines and cancellation.

Additional resources

- gRPC client factory integration in .NET
- Reliable gRPC services with deadlines and cancellation
- gRPC services with C#

gRPC client factory integration in .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.

By James Newton-King

gRPC integration with `HttpClientFactory` offers a centralized way to create gRPC clients. It can be used as an alternative to configuring stand-alone gRPC client instances. Factory integration is available in the Grpc.Net.ClientFactory NuGet package.

The factory offers the following benefits:

- Provides a central location for configuring logical gRPC client instances.
- Manages the lifetime of the underlying `HttpClientMessageHandler`.
- Automatic propagation of deadline and cancellation in an ASP.NET Core gRPC service.

Register gRPC clients

To register a gRPC client, the generic `AddGrpcClient` extension method can be used within an instance of WebApplicationBuilder at the app's entry point in `Program.cs`, specifying the gRPC typed client class and service address:

C#

```
builder.Services.AddGrpcClient<Greeter.GreeterClient>(o =>
{
    o.Address = new Uri("https://localhost:5001");
});
```

The gRPC client type is registered as transient with dependency injection (DI). The client can now be injected and consumed directly in types created by DI. ASP.NET Core MVC controllers, SignalR hubs and gRPC services are places where gRPC clients can automatically be injected:

C#

```
public class AggregatorService : Aggregator.AggregatorBase
{
    private readonly Greeter.GreeterClient _client;

    public AggregatorService(Greeter.GreeterClient client)
    {
        _client = client;
    }

    public override async Task SayHellos(HelloRequest request,
        IServerStreamWriter<HelloReply> responseStream, ServerCallContext
context)
    {
        // Forward the call on to the greeter service
        using (var call = _client.SayHellos(request))
        {
            await foreach (var response in
call.ResponseStream.ReadAllAsync())
            {
                await responseStream.WriteAsync(response);
            }
        }
    }
}
```

Configure HttpHandler

`HttpClientFactory` creates the `HttpMessageHandler` used by the gRPC client. Standard `HttpClientFactory` methods can be used to add outgoing request middleware or to configure the underlying `HttpClientHandler` of the `HttpClient`:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        var handler = new HttpClientHandler();
        handler.ClientCertificates.Add(LoadCertificate());
        return handler;
    });
```

For more information, see Make HTTP requests using IHttpClientFactory.

Configure Interceptors

gRPC interceptors can be added to clients using the `AddInterceptor` method.

```
C#

builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddInterceptor<LoggingInterceptor>();
```

The preceding code:

- Registers the `GreeterClient` type.
- Configures a `LoggingInterceptor` for this client. `LoggingInterceptor` is created once and shared between `GreeterClient` instances.

By default, an interceptor is created once and shared between clients. This behavior can be overridden by specifying a scope when registering an interceptor. The client factory can be configured to create a new interceptor for each client by specifying `InterceptorScope.Client`.

```
C#

builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddInterceptor<LoggingInterceptor>(InterceptorScope.Client);
```

Creating client scoped interceptors is useful when an interceptor requires scoped or transient scoped services from DI.

A gRPC interceptor or channel credentials can be used to send `Authorization` metadata with each request. For more information about configuring authentication, see Send a bearer token with gRPC client factory.

Configure Channel

Additional configuration can be applied to a channel using the `ConfigureChannel` method:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .ConfigureChannel(o =>
    {
        o.Credentials = new CustomCredentials();
    });
```

`ConfigureChannel` is passed a `GrpcChannelOptions` instance. For more information, see configure client options.

> **Note**
>
> Some properties are set on `GrpcChannelOptions` before the `ConfigureChannel` callback is run:
>
> - `HttpHandler` is set to the result from ConfigurePrimaryHttpMessageHandler.
> - `LoggerFactory` is set to the ILoggerFactory resolved from DI.
>
> These values can be overridden by `ConfigureChannel`.

Call credentials

An authentication header can be added to gRPC calls using the `AddCallCredentials` method:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddCallCredentials((context, metadata) =>
    {
        if (!string.IsNullOrEmpty(_token))
        {
            metadata.Add("Authorization", $"Bearer {_token}");
        }
        return Task.CompletedTask;
    });
```

For more information about configuring call credentials, see Bearer token with gRPC client factory.

Deadline and cancellation propagation

gRPC clients created by the factory in a gRPC service can be configured with `EnableCallContextPropagation()` to automatically propagate the deadline and cancellation token to child calls. The `EnableCallContextPropagation()` extension method is available in the Grpc.AspNetCore.Server.ClientFactory NuGet package.

Call context propagation works by reading the deadline and cancellation token from the current gRPC request context and automatically propagating them to outgoing calls made by the gRPC client. Call context propagation is an excellent way of ensuring that complex, nested gRPC scenarios always propagate the deadline and cancellation.

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .EnableCallContextPropagation();
```

By default, `EnableCallContextPropagation` raises an error if the client is used outside the context of a gRPC call. The error is designed to alert you that there isn't a call context to propagate. If you want to use the client outside of a call context, suppress the error when the client is configured with `SuppressContextNotFoundErrors`:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .EnableCallContextPropagation(o => o.SuppressContextNotFoundErrors =
true);
```

For more information about deadlines and RPC cancellation, see Reliable gRPC services with deadlines and cancellation.

Named clients

Typically, a gRPC client type is registered once and then injected directly into a type's constructor by DI. However, there are scenarios where it's useful to have multiple configurations for one client. For example, a client that makes gRPC calls with and without authentication.

Multiple clients with the same type can be registered by giving each client a name. Each named client can have its own configuration. The generic `AddGrpcClient` extension method has an overload that includes a name parameter:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>("Greeter", o =>
    {
        o.Address = new Uri("https://localhost:5001");
    });

builder.Services
    .AddGrpcClient<Greeter.GreeterClient>("GreeterAuthenticated", o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .ConfigureChannel(o =>
    {
        o.Credentials = new CustomCredentials();
    });
```

The preceding code:

- Registers the `GreeterClient` type twice, specifying a unique name for each.
- Configures different settings for each named client. The `GreeterAuthenticated` registration adds credentials to the channel so that gRPC calls made with it are authenticated.

A named gRPC client is created in app code using `GrpcClientFactory`. The type and name of the desired client is specified using the generic `GrpcClientFactory.CreateClient` method:

C#

```
public class AggregatorService : Aggregator.AggregatorBase
{
    private readonly Greeter.GreeterClient _client;

    public AggregatorService(GrpcClientFactory grpcClientFactory)
    {
        _client = grpcClientFactory.CreateClient<Greeter.GreeterClient>
("GreeterAuthenticated");
```

```
    }
}
```

Additional resources

- Call gRPC services with the .NET client
- Reliable gRPC services with deadlines and cancellation
- Make HTTP requests using IHttpClientFactory in ASP.NET Core

Reliable gRPC services with deadlines and cancellation

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

Deadlines and cancellation are features used by gRPC clients to abort in-progress calls. This article discusses why deadlines and cancellation are important, and how to use them in .NET gRPC apps.

Deadlines

A deadline allows a gRPC client to specify how long it will wait for a call to complete. When a deadline is exceeded, the call is canceled. Setting a deadline is important because it provides an upper limit on how long a call can run for. It stops misbehaving services from running forever and exhausting server resources. Deadlines are a useful tool for building reliable apps and should be configured.

Deadline configuration:

- A deadline is configured using `CallOptions.Deadline` when a call is made.
- There is no default deadline value. gRPC calls aren't time limited unless a deadline is specified.
- A deadline is the UTC time of when the deadline is exceeded. For example, `DateTime.UtcNow.AddSeconds(5)` is a deadline of 5 seconds from now.
- If a past or current time is used then the call immediately exceeds the deadline.
- The deadline is sent with the gRPC call to the service and is independently tracked by both the client and the service. It is possible that a gRPC call completes on one machine, but by the time the response has returned to the client the deadline has been exceeded.

If a deadline is exceeded, the client and service have different behavior:

- The client immediately aborts the underlying HTTP request and throws a `DeadlineExceeded` error. The client app can choose to catch the error and display a timeout message to the user.
- On the server, the executing HTTP request is aborted and ServerCallContext.CancellationToken is raised. Although the HTTP request is aborted, the gRPC call continues to run on the server until the method completes. It's important that the cancellation token is passed to async methods so they are cancelled along with the call. For example, passing a cancellation token to async database queries and HTTP requests. Passing a cancellation token allows the canceled call to complete quickly on the server and free up resources for other calls.

Configure `CallOptions.Deadline` to set a deadline for a gRPC call:

C#

```
var client = new Greet.GreeterClient(channel);

try
{
    var response = await client.SayHelloAsync(
        new HelloRequest { Name = "World" },
        deadline: DateTime.UtcNow.AddSeconds(5));

    // Greeting: Hello World
    Console.WriteLine("Greeting: " + response.Message);
}
catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded)
{
    Console.WriteLine("Greeting timeout.");
}
```

Using `ServerCallContext.CancellationToken` in a gRPC service:

C#

```
public override async Task<HelloReply> SayHello(HelloRequest request,
    ServerCallContext context)
{
    var user = await _databaseContext.GetUserAsync(request.Name,
        context.CancellationToken);

    return new HelloReply { Message = "Hello " + user.DisplayName };
}
```

Deadlines and retries

When a gRPC call is configured with retry fault handling and a deadline, the deadline tracks time across all retries for a gRPC call. If the deadline is exceeded, a gRPC call immediately aborts the underlying HTTP request, skips any remaining retries, and throws a `DeadlineExceeded` error.

Propagating deadlines

When a gRPC call is made from an executing gRPC service, the deadline should be propagated. For example:

1. Client app calls `FrontendService.GetUser` with a deadline.
2. `FrontendService` calls `UserService.GetUser`. The deadline specified by the client should be specified with the new gRPC call.
3. `UserService.GetUser` receives the deadline. It correctly times-out if the client app's deadline is exceeded.

The call context provides the deadline with `ServerCallContext.Deadline`:

C#

```
public override async Task<UserResponse> GetUser(UserRequest request,
    ServerCallContext context)
{
    var client = new User.UserServiceClient(_channel);
    var response = await client.GetUserAsync(
        new UserRequest { Id = request.Id },
        deadline: context.Deadline);

    return response;
}
```

Manually propagating deadlines can be cumbersome. The deadline needs to be passed to every call, and it's easy to accidentally miss. An automatic solution is available with gRPC client factory. Specifying `EnableCallContextPropagation`:

- Automatically propagates the deadline and cancellation token to child calls.
- Doesn't propagate the deadline if the child call specifies a smaller deadline. For example, a propagated deadline of 10 seconds isn't used if a child call specifies a new deadline of 5 seconds using `CallOptions.Deadline`. When multiple deadlines are available, the smallest deadline is used.
- Is an excellent way of ensuring that complex, nested gRPC scenarios always propagate the deadline and cancellation.

C#

```
services
    .AddGrpcClient<User.UserServiceClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .EnableCallContextPropagation();
```

For more information, see gRPC client factory integration in .NET.

Cancellation

Cancellation allows a gRPC client to cancel long running calls that are no longer needed. For example, a gRPC call that streams realtime updates is started when the user visits a page on a website. The stream should be canceled when the user navigates away from the page.

A gRPC call can be canceled in the client by passing a cancellation token with CallOptions.CancellationToken or calling `Dispose` on the call.

C#

```
private AsyncServerStreamingCall<HelloReply> _call;

public void StartStream()
{
    _call = client.SayHellos(new HelloRequest { Name = "World" });

    // Read response in background task.
    _ = Task.Run(async () =>
    {
        await foreach (var response in _call.ResponseStream.ReadAllAsync())
        {
            Console.WriteLine("Greeting: " + response.Message);
        }
    });
}

public void StopStream()
{
    _call.Dispose();
}
```

gRPC services that can be cancelled should:

- Pass `ServerCallContext.CancellationToken` to async methods. Canceling async methods allows the call on the server to complete quickly.

- Propagate the cancellation token to child calls. Propagating the cancellation token ensures that child calls are canceled with their parent. gRPC client factory and `EnableCallContextPropagation()` automatically propagates the cancellation token.

Additional resources

- Call gRPC services with the .NET client
- gRPC client factory integration in .NET

Transient fault handling with gRPC retries

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

gRPC retries is a feature that allows gRPC clients to automatically retry failed calls. This article discusses how to configure a retry policy to make resilient, fault tolerant gRPC apps in .NET.

gRPC retries requires Grpc.Net.Client version 2.36.0 or later.

Transient fault handling

gRPC calls can be interrupted by transient faults. Transient faults include:

- Momentary loss of network connectivity.
- Temporary unavailability of a service.
- Timeouts due to server load.

When a gRPC call is interrupted, the client throws an `RpcException` with details about the error. The client app must catch the exception and choose how to handle the error.

C#

```
var client = new Greeter.GreeterClient(channel);
try
{
    var response = await client.SayHelloAsync(
        new HelloRequest { Name = ".NET" });

    Console.WriteLine("From server: " + response.Message);
}
catch (RpcException ex)
{
```

```
    // Write logic to inspect the error and retry
    // if the error is from a transient fault.
}
```

Duplicating retry logic throughout an app is verbose and error prone. Fortunately the .NET gRPC client has a built-in support for automatic retries.

Configure a gRPC retry policy

A retry policy is configured once when a gRPC channel is created:

C#

```
var defaultMethodConfig = new MethodConfig
{
    Names = { MethodName.Default },
    RetryPolicy = new RetryPolicy
    {
        MaxAttempts = 5,
        InitialBackoff = TimeSpan.FromSeconds(1),
        MaxBackoff = TimeSpan.FromSeconds(5),
        BackoffMultiplier = 1.5,
        RetryableStatusCodes = { StatusCode.Unavailable }
    }
};

var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
{
    ServiceConfig = new ServiceConfig { MethodConfigs = {
defaultMethodConfig } }
});
```

The preceding code:

- Creates a `MethodConfig`. Retry policies can be configured per-method and methods are matched using the `Names` property. This method is configured with `MethodName.Default`, so it's applied to all gRPC methods called by this channel.
- Configures a retry policy. This policy instructs clients to automatically retry gRPC calls that fail with the status code `Unavailable`.
- Configures the created channel to use the retry policy by setting `GrpcChannelOptions.ServiceConfig`.

gRPC clients created with the channel will automatically retry failed calls:

C#

```
var client = new Greeter.GreeterClient(channel);
var response = await client.SayHelloAsync(
    new HelloRequest { Name = ".NET" });

Console.WriteLine("From server: " + response.Message);
```

When retries are valid

Calls are retried when:

- The failing status code matches a value in `RetryableStatusCodes`.
- The previous number of attempts is less than `MaxAttempts`.
- The call hasn't been commited.
- The deadline hasn't been exceeded.

A gRPC call becomes committed in two scenarios:

- The client receives response headers. Response headers are sent by the server when `ServerCallContext.WriteResponseHeadersAsync` is called, or when the first message is written to the server response stream.
- The client's outgoing message (or messages if streaming) has exceeded the client's maximum buffer size. `MaxRetryBufferSize` and `MaxRetryBufferPerCallSize` are configured on the channel.

Committed calls won't retry, regardless of the status code or the previous number of attempts.

Streaming calls

Streaming calls can be used with gRPC retries, but there are important considerations when they are used together:

- **Server streaming**, **bidirectional streaming**: Streaming RPCs that return multiple messages from the server won't retry after the first message has been received. Apps must add additional logic to manually re-establish server and bidirectional streaming calls.
- **Client streaming**, **bidirectional streaming**: Streaming RPCs that send multiple messages to the server won't retry if the outgoing messages have exceeded the client's maximum buffer size. The maximum buffer size can be increased with configuration.

For more information, see When retries are valid.

Retry backoff delay

The backoff delay between retry attempts is configured with `InitialBackoff`, `MaxBackoff`, and `BackoffMultiplier`. More information about each option is available in the gRPC retry options section.

The actual delay between retry attempts is randomized. A randomized delay between 0 and the current backoff determines when the next retry attempt is made. Consider that even with exponential backoff configured, increasing the current backoff between attempts, the actual delay between attempts isn't always larger. The delay is randomized to prevent retries from multiple calls from clustering together and potentially overloading the server.

Detect retries with metadata

gRPC retries can be detected by the presence of `grpc-previous-rpc-attempts` metadata. The `grpc-previous-rpc-attempts` metadata:

- Is automatically added to retried calls and sent to the server.
- Value represents the number of preceding retry attempts.
- Value is always an integer.

Consider the following retry scenario:

1. Client makes a gRPC call to the server.
2. Server fails and returns a retriable status code response.
3. Client retries the gRPC call. Because there was one previous attempt, `grpc-previous-rpc-attempts` metadata has a value of `1`. Metadata is sent to the server with the retry.
4. Server succeeds and returns OK.
5. Client reports success. `grpc-previous-rpc-attempts` is in the response metadata and has a value of `1`.

The `grpc-previous-rpc-attempts` metadata is not present on the initial gRPC call, is `1` for the first retry, `2` for the second retry, and so on.

gRPC retry options

The following table describes options for configuring gRPC retry policies:

Expand table

Option	Description
`MaxAttempts`	The maximum number of call attempts, including the original attempt. This value is limited by `GrpcChannelOptions.MaxRetryAttempts` which defaults to 5. A value is required and must be greater than 1.
`InitialBackoff`	The initial backoff delay between retry attempts. A randomized delay between 0 and the current backoff determines when the next retry attempt is made. After each attempt, the current backoff is multiplied by `BackoffMultiplier`. A value is required and must be greater than zero.
`MaxBackoff`	The maximum backoff places an upper limit on exponential backoff growth. A value is required and must be greater than zero.
`BackoffMultiplier`	The backoff will be multiplied by this value after each retry attempt and will increase exponentially when the multiplier is greater than 1. A value is required and must be greater than zero.
`RetryableStatusCodes`	A collection of status codes. A gRPC call that fails with a matching status will be automatically retried. For more information about status codes, see Status codes and their use in gRPC. At least one retryable status code is required.

Hedging

Hedging is an alternative retry strategy. Hedging enables aggressively sending multiple copies of a single gRPC call without waiting for a response. Hedged gRPC calls may be executed multiple times on the server and the first successful result is used. It's important that hedging is only enabled for methods that are safe to execute multiple times without adverse effect.

Hedging has pros and cons when compared to retries:

- An advantage to hedging is it might return a successful result faster. It allows for multiple simultaneously gRPC calls and will complete when the first successful result is available.
- A disadvantage to hedging is it can be wasteful. Multiple calls could be made and all succeed. Only the first result is used and the rest are discarded.

Configure a gRPC hedging policy

A hedging policy is configured like a retry policy. Note that a hedging policy can't be combined with a retry policy.

```
C#
```

```
var defaultMethodConfig = new MethodConfig
{
    Names = { MethodName.Default },
    HedgingPolicy = new HedgingPolicy
    {
        MaxAttempts = 5,
        NonFatalStatusCodes = { StatusCode.Unavailable }
    }
};

var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
{
    ServiceConfig = new ServiceConfig { MethodConfigs = {
defaultMethodConfig } }
});
```

gRPC hedging options

The following table describes options for configuring gRPC hedging policies:

Expand table

Option	Description
`MaxAttempts`	The hedging policy will send up to this number of calls. `MaxAttempts` represents the total number of all attempts, including the original attempt. This value is limited by `GrpcChannelOptions.MaxRetryAttempts` which defaults to 5. A value is required and must be 2 or greater.
`HedgingDelay`	The first call is sent immediately, subsequent hedging calls are delayed by this value. When the delay is set to zero or `null`, all hedged calls are sent immediately. `HedgingDelay` is optional and defaults to zero. A value must be zero or greater.
`NonFatalStatusCodes`	A collection of status codes which indicate other hedge calls may still succeed. If a non-fatal status code is returned by the server, hedged calls will continue. Otherwise, outstanding requests will be canceled and the error returned to the app. For more information about status codes, see Status codes and their use in gRPC.

Additional resources

- Call gRPC services with the .NET client
- Retry general guidance - Best practices for cloud applications

gRPC client-side load balancing

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

Client-side load balancing is a feature that allows gRPC clients to distribute load optimally across available servers. This article discusses how to configure client-side load balancing to create scalable, high-performance gRPC apps in .NET.

Client-side load balancing requires:

- .NET 5 or later.
- Grpc.Net.Client version 2.45.0 or later.

Configure gRPC client-side load balancing

Client-side load balancing is configured when a channel is created. The two components to consider when using load balancing:

- The resolver, which resolves the addresses for the channel. Resolvers support getting addresses from an external source. This is also known as service discovery.
- The load balancer, which creates connections and picks the address that a gRPC call will use.

Built-in implementations of resolvers and load balancers are included in Grpc.Net.Client. Load balancing can also be extended by writing custom resolvers and load balancers.

Addresses, connections and other load balancing state is stored in a `GrpcChannel` instance. A channel must be reused when making gRPC calls for load balancing to work correctly.

> **Note**

Some load balancing configuration uses dependency injection (DI). Apps that don't use DI can create a ServiceCollection instance.

If an app already has DI setup, like an ASP.NET Core website, then types should be registered with the existing DI instance. `GrpcChannelOptions.ServiceProvider` is configured by getting an IServiceProvider from DI.

Configure resolver

The resolver is configured using the address a channel is created with. The URI scheme of the address specifies the resolver.

Expand table

Scheme	Type	Description
`dns`	`DnsResolverFactory`	Resolves addresses by querying the hostname for DNS address records.
`static`	`StaticResolverFactory`	Resolves addresses that the app has specified. Recommended if an app already knows the addresses it calls.

A channel doesn't directly call a URI that matches a resolver. Instead, a matching resolver is created and used to resolve the addresses.

For example, using `GrpcChannel.ForAddress("dns:///my-example-host", new GrpcChannelOptions { Credentials = ChannelCredentials.Insecure })`:

- The `dns` scheme maps to `DnsResolverFactory`. A new instance of a DNS resolver is created for the channel.
- The resolver makes a DNS query for `my-example-host` and gets two results: `127.0.0.100` and `127.0.0.101`.
- The load balancer uses `127.0.0.100:80` and `127.0.0.101:80` to create connections and make gRPC calls.

DnsResolverFactory

The `DnsResolverFactory` creates a resolver designed to get addresses from an external source. DNS resolution is commonly used to load balance over pod instances that have a Kubernetes headless services.

```
C#
```

```
var channel = GrpcChannel.ForAddress(
    "dns:///my-example-host",
    new GrpcChannelOptions { Credentials = ChannelCredentials.Insecure });
var client = new Greet.GreeterClient(channel);

var response = await client.SayHelloAsync(new HelloRequest { Name = "world"
});
```

The preceding code:

- Configures the created channel with the address `dns:///my-example-host`.
  - The `dns` scheme maps to `DnsResolverFactory`.
  - `my-example-host` is the hostname to resolve.
  - No port is specified in the address, so gRPC calls are sent to port 80. This is the default port for unsecured channels. A port can optionally be specified after the hostname. For example, `dns:///my-example-host:8080` configures gRPC calls to be sent to port 8080.
- Doesn't specify a load balancer. The channel defaults to a pick first load balancer.
- Starts the gRPC call `SayHello`:
  - DNS resolver gets addresses for the hostname `my-example-host`.
  - Pick first load balancer attempts to connect to one of the resolved addresses.
  - The call is sent to the first address the channel successfully connects to.

DNS address caching

Performance is important when load balancing. The latency of resolving addresses is eliminated from gRPC calls by caching the addresses. A resolver will be invoked when making the first gRPC call, and subsequent calls use the cache.

Addresses are automatically refreshed if a connection is interrupted. Refreshing is important in scenarios where addresses change at runtime. For example, in Kubernetes a restarted pod triggers the DNS resolver to refresh and get the pod's new address.

By default, a DNS resolver is refreshed if a connection is interrupted. The DNS resolver can also optionally refresh itself on a periodic interval. This can be useful for quickly detecting new pod instances.

C#

```
services.AddSingleton<ResolverFactory>(
    sp => new DnsResolverFactory(refreshInterval:
TimeSpan.FromSeconds(30)));
```

The preceding code creates a `DnsResolverFactory` with a refresh interval and registers it with dependency injection. For more information on using a custom-configured resolver, see Configure custom resolvers and load balancers.

StaticResolverFactory

A static resolver is provided by `StaticResolverFactory`. This resolver:

- Doesn't call an external source. Instead, the client app configures the addresses.
- Is designed for situations where an app already knows the addresses it calls.

C#

```
var factory = new StaticResolverFactory(addr => new[]
{
    new BalancerAddress("localhost", 80),
    new BalancerAddress("localhost", 81)
});

var services = new ServiceCollection();
services.AddSingleton<ResolverFactory>(factory);

var channel = GrpcChannel.ForAddress(
    "static:///my-example-host",
    new GrpcChannelOptions
    {
        Credentials = ChannelCredentials.Insecure,
        ServiceProvider = services.BuildServiceProvider()
    });
var client = new Greet.GreeterClient(channel);
```

The preceding code:

- Creates a `StaticResolverFactory`. This factory knows about two addresses: `localhost:80` and `localhost:81`.
- Registers the factory with dependency injection (DI).
- Configures the created channel with:
  - The address `static:///my-example-host`. The `static` scheme maps to a static resolver.
  - Sets `GrpcChannelOptions.ServiceProvider` with the DI service provider.

This example creates a new ServiceCollection for DI. Suppose an app already has DI setup, like an ASP.NET Core website. In that case, types should be registered with the existing DI instance. `GrpcChannelOptions.ServiceProvider` is configured by getting an IServiceProvider from DI.

Configure load balancer

A load balancer is specified in a service config using the `ServiceConfig.LoadBalancingConfigs` collection. Two load balancers are built-in and map to load balancer config names:

Expand table

Name	Type	Description
`pick_first`	`PickFirstLoadBalancerFactory`	Attempts to connect to addresses until a connection is successfully made. gRPC calls are all made to the first successful connection.
`round_robin`	`RoundRobinLoadBalancerFactory`	Attempts to connect to all addresses. gRPC calls are distributed across all successful connections using round-robin logic.

`service config` is an abbreviation of service configuration and is represented by the `ServiceConfig` type. There are a couple of ways a channel can get a `service config` with a load balancer configured:

- An app can specify a `service config` when a channel is created using `GrpcChannelOptions.ServiceConfig`.
- Alternatively, a resolver can resolve a `service config` for a channel. This feature allows an external source to specify how its callers should perform load balancing. Whether a resolver supports resolving a `service config` is dependent on the resolver implementation. Disable this feature with `GrpcChannelOptions.DisableResolverServiceConfig`.
- If no `service config` is provided, or the `service config` doesn't have a load balancer configured, the channel defaults to `PickFirstLoadBalancerFactory`.

C#

```
var channel = GrpcChannel.ForAddress(
    "dns:///my-example-host",
    new GrpcChannelOptions
    {
        Credentials = ChannelCredentials.Insecure,
        ServiceConfig = new ServiceConfig { LoadBalancingConfigs = { new
RoundRobinConfig() } }
    });
var client = new Greet.GreeterClient(channel);
```

```
var response = await client.SayHelloAsync(new HelloRequest { Name = "world"
});
```

The preceding code:

- Specifies a `RoundRobinLoadBalancerFactory` in the `service config`.
- Starts the gRPC call `SayHello`:
  - `DnsResolverFactory` creates a resolver that gets addresses for the hostname `my-example-host`.
  - Round-robin load balancer attempts to connect to all resolved addresses.
  - gRPC calls are distributed evenly using round-robin logic.

Configure channel credentials

A channel must know whether gRPC calls are sent using transport security. `http` and `https` are no longer part of the address, the scheme now specifies a resolver, so `Credentials` must be configured on channel options when using load balancing.

- `ChannelCredentials.SecureSsl` - gRPC calls are secured with Transport Layer Security (TLS). Equivalent to an `https` address.
- `ChannelCredentials.Insecure` - gRPC calls don't use transport security. Equivalent to an `http` address.

C#

```
var channel = GrpcChannel.ForAddress(
    "dns:///my-example-host",
    new GrpcChannelOptions { Credentials = ChannelCredentials.Insecure });
var client = new Greet.GreeterClient(channel);

var response = await client.SayHelloAsync(new HelloRequest { Name = "world"
});
```

Use load balancing with gRPC client factory

gRPC client factory can be configured to use load balancing:

C#

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

builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
```

```
    {
        o.Address = new Uri("dns:///my-example-host");
    })
    .ConfigureChannel(o => o.Credentials = ChannelCredentials.Insecure);

builder.Services.AddSingleton<ResolverFactory>(
    sp => new DnsResolverFactory(refreshInterval:
TimeSpan.FromSeconds(30)));

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

The preceding code:

- Configures the client with a load-balancing address.
- Specifies channel credentials.
- Registers DI types with the app's IServiceCollection.

Write custom resolvers and load balancers

Client-side load balancing is extensible:

- Implement `Resolver` to create a custom resolver and resolve addresses from a new data source.
- Implement `LoadBalancer` to create a custom load balancer with new load balancing behavior.

> **ⓘ Important**
>
> The APIs used to extend client-side load balancing are experimental. They can change without notice.

Create a custom resolver

A resolver:

- Implements `Resolver` and is created by a `ResolverFactory`. Create a custom resolver by implementing these types.
- Is responsible for resolving the addresses a load balancer uses.
- Can optionally provide a service configuration.

C#

```
public class FileResolver : PollingResolver
{
```

```
    private readonly Uri _address;
    private readonly int _port;

    public FileResolver(Uri address, int defaultPort, ILoggerFactory
loggerFactory)
        : base(loggerFactory)
    {
        _address = address;
        _port = defaultPort;
    }

    public override async Task ResolveAsync(CancellationToken
cancellationToken)
    {
        // Load JSON from a file on disk and deserialize into endpoints.
        var jsonString = await File.ReadAllTextAsync(_address.LocalPath);
        var results = JsonSerializer.Deserialize<string[]>(jsonString);
        var addresses = results.Select(r => new BalancerAddress(r,
_port)).ToArray();

        // Pass the results back to the channel.
        Listener(ResolverResult.ForResult(addresses));
    }
}

public class FileResolverFactory : ResolverFactory
{
    // Create a FileResolver when the URI has a 'file' scheme.
    public override string Name => "file";

    public override Resolver Create(ResolverOptions options)
    {
        return new FileResolver(options.Address, options.DefaultPort,
options.LoggerFactory);
    }
}
```

In the preceding code:

- `FileResolverFactory` implements `ResolverFactory`. It maps to the `file` scheme and creates `FileResolver` instances.
- `FileResolver` implements `PollingResolver`. `PollingResolver` is an abstract base type that makes it easy to implement a resolver with asynchronous logic by overriding `ResolveAsync`.
- In `ResolveAsync`:
  - The file URI is converted to a local path. For example, `file:///c:/addresses.json` becomes `c:\addresses.json`.
  - JSON is loaded from disk and converted into a collection of addresses.

- Listener is called with results to let the channel know that addresses are available.

Create a custom load balancer

A load balancer:

- Implements `LoadBalancer` and is created by a `LoadBalancerFactory`. Create a custom load balancer and factory by implementing these types.
- Is given addresses from a resolver and creates `Subchannel` instances.
- Tracks state about the connection and creates a `SubchannelPicker`. The channel internally uses the picker to pick addresses when making gRPC calls.

The `SubchannelsLoadBalancer` is:

- An abstract base class that implements `LoadBalancer`.
- Manages creating `Subchannel` instances from addresses.
- Makes it easy to implement a custom picking policy over a collection of subchannels.

C#

```
public class RandomBalancer : SubchannelsLoadBalancer
{
    public RandomBalancer(IChannelControlHelper controller, ILoggerFactory
loggerFactory)
        : base(controller, loggerFactory)
    {
    }

    protected override SubchannelPicker CreatePicker(List<Subchannel>
readySubchannels)
    {
        return new RandomPicker(readySubchannels);
    }

    private class RandomPicker : SubchannelPicker
    {
        private readonly List<Subchannel> _subchannels;

        public RandomPicker(List<Subchannel> subchannels)
        {
            _subchannels = subchannels;
        }

        public override PickResult Pick(PickContext context)
        {
            // Pick a random subchannel.
            return
```

```
PickResult.ForSubchannel(_subchannels[Random.Shared.Next(0,
_subchannels.Count)]);
        }
    }
}

public class RandomBalancerFactory : LoadBalancerFactory
{
    // Create a RandomBalancer when the name is 'random'.
    public override string Name => "random";

    public override LoadBalancer Create(LoadBalancerOptions options)
    {
        return new RandomBalancer(options.Controller,
options.LoggerFactory);
    }
}
```

In the preceding code:

- `RandomBalancerFactory` implements `LoadBalancerFactory`. It maps to the `random` policy name and creates `RandomBalancer` instances.
- `RandomBalancer` implements `SubchannelsLoadBalancer`. It creates a `RandomPicker` that randomly picks a subchannel.

Configure custom resolvers and load balancers

Custom resolvers and load balancers need to be registered with dependency injection (DI) when they are used. There are a couple of options:

- If an app is already using DI, such as an ASP.NET Core web app, they can be registered with the existing DI configuration. An IServiceProvider can be resolved from DI and passed to the channel using `GrpcChannelOptions.ServiceProvider`.
- If an app isn't using DI then create:
  - A ServiceCollection with types registered with it.
  - A service provider using BuildServiceProvider.

C#

```
var services = new ServiceCollection();
services.AddSingleton<ResolverFactory, FileResolverFactory>();
services.AddSingleton<LoadBalancerFactory, RandomLoadBalancerFactory>();

var channel = GrpcChannel.ForAddress(
    "file:///c:/data/addresses.json",
    new GrpcChannelOptions
    {
```

```
        Credentials = ChannelCredentials.Insecure,
        ServiceConfig = new ServiceConfig { LoadBalancingConfigs = { new
LoadBalancingConfig("random") } },
        ServiceProvider = services.BuildServiceProvider()
    });
var client = new Greet.GreeterClient(channel);
```

The preceding code:

- Creates a `ServiceCollection` and registers new resolver and load balancer implementations.
- Creates a channel configured to use the new implementations:
  - `ServiceCollection` is built into an `IServiceProvider` and set to `GrpcChannelOptions.ServiceProvider`.
  - Channel address is `file:///c:/data/addresses.json`. The `file` scheme maps to `FileResolverFactory`.
  - `service config` load balancer name is `random`. Maps to `RandomLoadBalancerFactory`.

Why load balancing is important

HTTP/2 multiplexes multiple calls on a single TCP connection. If gRPC and HTTP/2 are used with a network load balancer (NLB), the connection is forwarded to a server, and all gRPC calls are sent to that one server. The other server instances on the NLB are idle.

Network load balancers are a common solution for load balancing because they are fast and lightweight. For example, Kubernetes by default uses a network load balancer to balance connections between pod instances. However, network load balancers are not effective at distributing load when used with gRPC and HTTP/2.

Proxy or client-side load balancing?

gRPC and HTTP/2 can be effectively load balanced using either an application load balancer proxy or client-side load balancing. Both of these options allow individual gRPC calls to be distributed across available servers. Deciding between proxy and client-side load balancing is an architectural choice. There are pros and cons for each.

- **Proxy**: gRPC calls are sent to the proxy, the proxy makes a load balancing decision, and the gRPC call is sent on to the final endpoint. The proxy is responsible for knowing about endpoints. Using a proxy adds:
  - An additional network hop to gRPC calls.
  - Latency and consumes additional resources.

    - Proxy server must be setup and configured correctly.

- **Client-side load balancing**: The gRPC client makes a load balancing decision when a gRPC call is started. The gRPC call is sent directly to the final endpoint. When using client-side load balancing:
    - The client is responsible for knowing about available endpoints and making load balancing decisions.
    - Additional client configuration is required.
    - High-performance, load balanced gRPC calls eliminate the need for a proxy.

Additional resources

- Call gRPC services with the .NET client

Use gRPC client with .NET Standard 2.0

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 discusses how to use the .NET gRPC client with .NET implementations that support .NET Standard 2.0.

.NET implementations

The following .NET implementations (or later) support Grpc.Net.Client but don't have full support for HTTP/2:

- .NET Core 2.1
- .NET Framework 4.6.1
- Mono 5.4
- Xamarin.iOS 10.14
- Xamarin.Android 8.0
- Universal Windows Platform 10.0.16299
- Unity 2018.1

The .NET gRPC client can call services from these .NET implementations with some additional configuration.

HttpHandler configuration

An HTTP provider must be configured using `GrpcChannelOptions.HttpHandler`. If a handler isn't configured, an error is thrown:

> `System.PlatformNotSupportedException`: gRPC requires extra configuration to successfully make RPC calls on .NET implementations that don't have support for gRPC over HTTP/2. An HTTP provider must be specified using

> `GrpcChannelOptions.HttpHandler`. The configured HTTP provider must either support HTTP/2 or be configured to use gRPC-Web.

.NET implementations that don't support HTTP/2, such as UWP, Xamarin, and Unity, can use gRPC-Web as an alternative.

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
    {
        HttpHandler = new GrpcWebHandler(new HttpClientHandler())
    });

var client = new Greeter.GreeterClient(channel);
var response = await client.SayHelloAsync(new HelloRequest { Name = ".NET"
});
```

Clients can also be created using the gRPC client factory. An HTTP provider is configured using the ConfigurePrimaryHttpMessageHandler extension method.

C#

```
builder.Services
    .AddGrpcClient<Greet.GreeterClient>(options =>
    {
        options.Address = new Uri("https://localhost:5001");
    })
    .ConfigurePrimaryHttpMessageHandler(
        () => new GrpcWebHandler(new HttpClientHandler()));
```

For more information, see Configure gRPC-Web with the .NET gRPC client.

> ⓘ **Important**
>
> gRPC-Web requires the client ***and*** server to support it. gRPC-Web can be **quickly configured by an ASP.NET Core gRPC server**. Other gRPC server implementations require a proxy to support gRPC-Web.

.NET Framework

.NET Framework has limited support for gRPC over HTTP/2. To enable gRPC over HTTP/2 on .NET Framework, configure the channel to use WinHttpHandler.

Requirements and restrictions to using `WinHttpHandler`:

- Windows 11 or later, Windows Server 2019 or later.
  - gRPC client is fully supported on Windows 11 or later.
  - gRPC client is partially supported on Windows Server 2019 and Windows Server 2022. Unary and server streaming methods are supported. Client and bidirectional streaming methods are ***not*** supported.
- A reference to System.Net.Http.WinHttpHandler version 6.0.1 or later.
- Configure `WinHttpHandler` on the channel using `GrpcChannelOptions.HttpHandler`.
- .NET Framework 4.6.1 or later.
- Only gRPC calls over TLS are supported.

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
    {
        HttpHandler = new WinHttpHandler()
    });

var client = new Greeter.GreeterClient(channel);
var response = await client.SayHelloAsync(new HelloRequest { Name = ".NET"
});
```

gRPC C# core-library

An alternative option for .NET Framework and Xamarin has been to use gRPC C# core-library to make gRPC calls. gRPC C# core-library is:

- A third party library that supports making gRPC calls over HTTP/2 on .NET Framework and Xamarin.
- Not supported by Microsoft.
- In maintenance mode and will be deprecated in favour of gRPC for .NET.
- Not recommended for new apps.

Additional resources

- Call gRPC services with the .NET client
- Use gRPC in browser apps
- gRPC C# core-library

Mock gRPC client in tests

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

Testing is an important aspect of building stable and maintainable software. Part of writing high-quality tests is removing external dependencies. This article discusses using mock gRPC clients in tests to remove gRPC calls to external servers.

Example testable client app

To demonstrate client app tests, review the following type in the sample app.

View or download sample code (how to download)

The `Worker` is a BackgroundService that makes calls to a gRPC server.

C#

```
public class Worker : BackgroundService
{
    private readonly Tester.TesterClient _client;
    private readonly IGreetRepository _greetRepository;

    public Worker(Tester.TesterClient client, IGreetRepository
greetRepository)
    {
        _client = client;
        _greetRepository = greetRepository;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        var count = 0;
        while (!stoppingToken.IsCancellationRequested)
        {
```

```
            count++;

            var reply = await _client.SayHelloUnaryAsync(
                new HelloRequest { Name = $"Worker {count}" });

            _greetRepository.SaveGreeting(reply.Message);

            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

The preceding type:

- Follows the Explicit Dependencies Principle.
- `TesterClient` is generated automatically by the tooling package Grpc.Tools based on the *test.proto* file, during the build process.
- Expects dependency injection (DI) to provide instances of `TesterClient` and `IGreetRepository`. The app is configured to use the gRPC client factory to create `TesterClient`.
- Can be tested with a mocked `IGreetRepository` service and `TesterClient` client using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Integration tests in ASP.NET Core.

For more information on the C# assets automatically generated by Grpc.Tools, see gRPC services with C#: Generated C# assets.

Mock a gRPC client

gRPC clients are concrete client types that are generated from .proto files. The concrete gRPC client has methods that translate to the gRPC service in the `.proto` file. For example, a service called `Greeter` generates a `GreeterClient` type with methods to call the service.

A mocking framework can mock a gRPC client type. When a mocked client is passed to the type, the test uses the mocked method instead of sending a gRPC call to a server.

C#

```
[Fact]
public async Task Greeting_Success_RepositoryCalled()
{
    // Arrange
    var mockRepository = new Mock<IGreetRepository>();
```

```
    var mockCall = CallHelpers.CreateAsyncUnaryCall(new HelloReply { Message
= "Test" });
    var mockClient = new Mock<Tester.TesterClient>();
    mockClient
        .Setup(m => m.SayHelloUnaryAsync(
            It.IsAny<HelloRequest>(), null, null, CancellationToken.None))
        .Returns(mockCall);

    var worker = new Worker(mockClient.Object, mockRepository.Object);

    // Act
    await worker.StartAsync(CancellationToken.None);

    // Assert
    mockRepository.Verify(v => v.SaveGreeting("Test"));
}
```

The preceding unit test:

- Mocks `IGreetRepository` and `TesterClient` using Moq.
- Starts the worker.
- Verifies `SaveGreeting` is called with the greeting message returned by the mocked `TesterClient`.

Additional resources

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

gRPC services with ASP.NET Core

Article • 07/31/2024

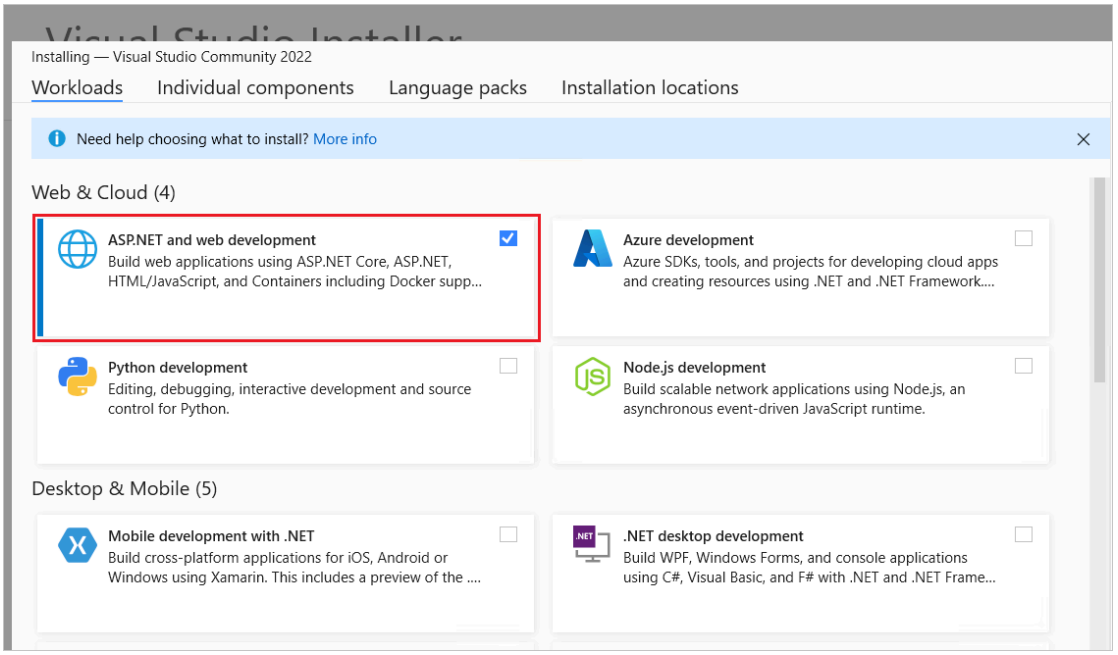
> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

This document shows how to get started with gRPC services using ASP.NET Core.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Get started with gRPC service in ASP.NET Core

View or download sample code (how to download).

Visual Studio

See Get started with gRPC services for detailed instructions on how to create a gRPC project.

Add gRPC services to an ASP.NET Core app

gRPC requires the Grpc.AspNetCore package.

Configure gRPC

In `Program.cs`:

- gRPC is enabled with the `AddGrpc` method.
- Each gRPC service is added to the routing pipeline through the `MapGrpcService` method.

C#

```
using GrpcGreeter.Services;

var builder = WebApplication.CreateBuilder(args);

// Additional configuration is required to successfully run gRPC on macOS.
// For instructions on how to configure Kestrel and gRPC clients on macOS,
visit https://go.microsoft.com/fwlink/?linkid=2099682

// Add services to the container.
builder.Services.AddGrpc();

var app = builder.Build();

// Configure the HTTP request pipeline.
app.MapGrpcService<GreeterService>();
app.MapGet("/", () => "Communication with gRPC endpoints must be made
through a gRPC client. To learn how to create a client, visit:
https://go.microsoft.com/fwlink/?linkid=2086909");

app.Run();
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

ASP.NET Core middleware and features share the routing pipeline, therefore an app can be configured to serve additional request handlers. The additional request handlers,

such as MVC controllers, work in parallel with the configured gRPC services.

Server options

gRPC services can be hosted by all built-in ASP.NET Core servers.

- ✓ Kestrel
- ✓ TestServer
- ✓ IIS†
- ✓ HTTP.sys†

†Requires .NET 5 and Windows 11 Build 22000 or Windows Server 2022 Build 20348 or later.

For more information about choosing the right server for an ASP.NET Core app, see Web server implementations in ASP.NET Core.

Kestrel

Kestrel is a cross-platform web server for ASP.NET Core. Kestrel focuses on high performance and memory utilization, but it doesn't have some of the advanced features in HTTP.sys such as port sharing.

Kestrel gRPC endpoints:

- Require HTTP/2.
- Should be secured with Transport Layer Security (TLS).

HTTP/2

gRPC requires HTTP/2. gRPC for ASP.NET Core validates HttpRequest.Protocol is `HTTP/2`.

Kestrel supports HTTP/2 on most modern operating systems. Kestrel endpoints are configured to support HTTP/1.1 and HTTP/2 connections by default.

TLS

Kestrel endpoints used for gRPC should be secured with TLS. In development, an endpoint secured with TLS is automatically created at `https://localhost:5001` when the ASP.NET Core development certificate is present. No configuration is required. An `https` prefix verifies the Kestrel endpoint is using TLS.

In production, TLS must be explicitly configured. In the following `appsettings.json` example, an HTTP/2 endpoint secured with TLS is provided:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "HttpsInlineCertFile": {
        "Url": "https://localhost:5001",
        "Protocols": "Http2",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      }
    }
  }
}
```

Alternatively, Kestrel endpoints can be configured in `Program.cs`:

C#

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

builder.WebHost.ConfigureKestrel(options =>
{
    options.Listen(IPAddress.Any, 5001, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http2;
        listenOptions.UseHttps("<path to .pfx file>",
            "<certificate password>");
    });
});
```

For more information on enabling TLS with Kestrel, see Kestrel HTTPS endpoint configuration.

Protocol negotiation

TLS is used for more than securing communication. 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. This negotiation determines whether the connection uses HTTP/1.1 or HTTP/2.

If an HTTP/2 endpoint is configured without TLS, the endpoint's ListenOptions.Protocols must be set to `HttpProtocols.Http2`. An endpoint with multiple protocols, such as `HttpProtocols.Http1AndHttp2` for example, can't be used without TLS because there's no negotiation. All connections to the unsecured endpoint default to HTTP/1.1, and gRPC calls fail.

For more information on enabling HTTP/2 and TLS with Kestrel, see Kestrel endpoint configuration.

> **Note**
>
> macOS doesn't support ASP.NET Core gRPC with TLS before .NET 8. Additional configuration is required to successfully run gRPC services on macOS when using .NET 7 or earlier. For more information, see **Unable to start ASP.NET Core gRPC app on macOS**.

IIS

Internet Information Services (IIS) is a flexible, secure and manageable Web Server for hosting web apps, including ASP.NET Core. .NET 5 and Windows 11 Build 22000 or Windows Server 2022 Build 20348 or later are required to host gRPC services with IIS.

IIS must be configured to use TLS and HTTP/2. For more information, see Use ASP.NET Core with HTTP/2 on IIS.

HTTP.sys

HTTP.sys is a web server for ASP.NET Core that only runs on Windows. .NET 5 and Windows 11 Build 22000 or Windows Server 2022 Build 20348 or later are required to host gRPC services with HTTP.sys.

HTTP.sys must be configured to use TLS and HTTP/2. For more information, see HTTP.sys web server HTTP/2 support.

Host gRPC in non-ASP.NET Core projects

An ASP.NET Core gRPC server is typically created from the gRPC template. The project file created by the template uses `Microsoft.NET.SDK.Web` as the SDK:

```
XML
```

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

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.47.0" />
    <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
  </ItemGroup>

</Project>
```

The `Microsoft.NET.SDK.Web` SDK value automatically adds a reference to the ASP.NET Core framework. The reference allows the app to use ASP.NET Core types required to host a server.

You can add a gRPC server to non-ASP.NET Core projects with the following project file settings:

XML

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

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.47.0" />
    <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />

    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

The preceding project file:

- Doesn't use `Microsoft.NET.SDK.Web` as the SDK.
- Adds a framework reference to `Microsoft.AspNetCore.App`.
  - The framework reference allows non-ASP.NET Core apps, such as Windows Services, WPF apps, or WinForms apps to use ASP.NET Core APIs.
  - The app can now use ASP.NET Core APIs to start an ASP.NET Core server.
- Adds gRPC requirements:
  - NuGet package reference to Grpc.AspNetCore.
  - `.proto` file.

For more information about using the `Microsoft.AspNetCore.App` framework reference, see Use the ASP.NET Core shared framework.

Integration with ASP.NET Core APIs

gRPC services have full access to the ASP.NET Core features such as Dependency Injection (DI) and Logging. For example, the service implementation can resolve a logger service from the DI container via the constructor:

C#

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

By default, the gRPC service implementation can resolve other DI services with any lifetime (Singleton, Scoped, or Transient).

Resolve HttpContext in gRPC methods

The gRPC API provides access to some HTTP/2 message data, such as the method, host, header, and trailers. Access is through the `ServerCallContext` argument passed to each gRPC method:

C#

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(
        HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

`ServerCallContext` doesn't provide full access to `HttpContext` in all ASP.NET APIs. The `GetHttpContext` extension method provides full access to the `HttpContext` representing the underlying HTTP/2 message in ASP.NET APIs:

C#

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(
        HelloRequest request, ServerCallContext context)
    {
```

```
        var httpContext = context.GetHttpContext();
        var clientCertificate = httpContext.Connection.ClientCertificate;

        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name + " from " +
clientCertificate.Issuer
        });
    }
}
```

Additional resources

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

gRPC on .NET supported platforms

Article • 11/06/2024

> **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 discusses the requirements and supported platforms for using gRPC with .NET. There are different requirements for the two major gRPC workloads:

- Hosting gRPC services in ASP.NET Core
- Calling gRPC from .NET client apps

Wire-formats

gRPC takes advantage of advanced features available in HTTP/2. HTTP/2 isn't supported everywhere, but a second wire-format using HTTP/1.1 is available for gRPC:

- application/grpc - gRPC over HTTP/2 is how gRPC is typically used.
- application/grpc-web - gRPC-Web modifies the gRPC protocol to be compatible with HTTP/1.1. gRPC-Web can be used in more places. gRPC-Web can be used by browser apps and in networks without complete support for HTTP/2. Two advanced gRPC features are no longer supported: client streaming and bidirectional streaming.

gRPC on .NET supports both wire-formats. `application/grpc` is used by default. gRPC-Web must be configured on the client and the server for successful gRPC-Web calls. For information on setting up gRPC-Web, see gRPC-Web in ASP.NET Core gRPC apps.

ASP.NET Core gRPC server requirements

Hosting gRPC services with ASP.NET Core requires .NET Core 3.x or later.

✔ .NET 5 or later

- ✔ .NET Core 3

ASP.NET Core gRPC services can be hosted on all operating system that .NET Core supports.

- ✔ Windows
- ✔ Linux
- ✔ macOS

Supported ASP.NET Core servers

All built-in ASP.NET Core servers are supported.

- ✔ Kestrel
- ✔ TestServer
- ✔ IIS†
- ✔ HTTP.sys†

†Requires .NET 5 and Windows 11 Build 22000 or Windows Server 2022 Build 20348 or later.

For information about configuring ASP.NET Core servers to run gRPC, see gRPC services with ASP.NET Core.

Azure services

- ✔ Azure Kubernetes Service (AKS)
- ✔ Azure Container Apps
- ✔ Azure App Service †

†gRPC requires a Linux-based environment on Azure App Service. See How-to deploy a .NET 6 gRPC app on App Service for Azure App Service deployment information.

.NET gRPC client requirements

The Grpc.Net.Client package supports gRPC calls over HTTP/2 on .NET Core 3 and .NET 5 or later.

Limited support is available for gRPC over HTTP/2 on .NET Framework. Other .NET versions such as UWP, Xamarin and Unity don't have required HTTP/2 support, and must use gRPC-Web instead.

The following table lists .NET implementations and their gRPC client support:

Expand table

.NET implementation	gRPC over HTTP/2	gRPC-Web
.NET 5 or later	✔	✔
.NET Core 3	✔	✔
.NET Core 2.1	✘	✔
.NET Framework 4.6.1	⚠†	✔
Blazor WebAssembly	✘	✔
Mono 5.4	✘	✔
Xamarin.iOS 10.14	✘	✔
Xamarin.Android 8.0	✘	✔
Universal Windows Platform 10.0.16299	✘	✔
Unity 2018.1	✘	✔

†.NET Framework requires configuration of WinHttpHandler and Windows 11 or later, Windows Server 2019 or later. For more information, see Make gRPC calls on .NET Framework.

Using `Grpc.Net.Client` with gRPC-Web requires additional configuration. For more information:

- Configure gRPC-Web with the .NET gRPC client
- Use gRPC client with .NET Standard 2.0

> **Important**
>
> gRPC-Web requires the client ***and*** server to support it. gRPC-Web can be **quickly configured by an ASP.NET Core gRPC server**. Other gRPC server implementations require a proxy to support gRPC-Web.

Additional resources

- Use gRPC client with .NET Standard 2.0
- gRPC C# core-library

Use gRPC in browser apps

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

It's not possible to directly call a gRPC service from a browser. gRPC uses HTTP/2 features, and no browser provides the level of control required over web requests to support a gRPC client.

gRPC on ASP.NET Core offers two browser-compatible solutions, gRPC-Web and gRPC JSON transcoding.

gRPC-Web

gRPC-Web allows browser apps to call gRPC services with the gRPC-Web client and Protobuf.

- Similar to normal gRPC, but it has a slightly different wire-protocol, which makes it compatible with HTTP/1.1 and browsers.
- Requires the browser app to generate a gRPC client from a `.proto` file.
- Allows browser apps to benefit from the high-performance and low network usage of binary messages.

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

gRPC JSON transcoding

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 are automatically created from gRPC services by annotating the `.proto` file with HTTP metadata.
- 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.

Additional resources

- gRPC-Web in ASP.NET Core gRPC apps
- gRPC JSON transcoding in ASP.NET Core gRPC apps

gRPC-Web in ASP.NET Core gRPC apps

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

Learn how to configure an existing ASP.NET Core gRPC service to be callable from browser apps, using the gRPC-Web protocol. gRPC-Web allows browser JavaScript and Blazor apps to call gRPC services. It's not possible to call an HTTP/2 gRPC service from a browser-based app. gRPC services hosted in ASP.NET Core can be configured to support gRPC-Web alongside HTTP/2 gRPC.

For instructions on adding a gRPC service to an existing ASP.NET Core app, see Add gRPC services to an ASP.NET Core app.

For instructions on creating a gRPC project, see Create a .NET Core gRPC client and server in ASP.NET Core.

ASP.NET Core gRPC-Web versus Envoy

There are two choices for how to add gRPC-Web to an ASP.NET Core app:

- Support gRPC-Web alongside gRPC HTTP/2 in ASP.NET Core. This option uses middleware provided by the Grpc.AspNetCore.Web package.
- Use the Envoy proxy's gRPC-Web support to translate gRPC-Web to gRPC HTTP/2. The translated call is then forwarded onto the ASP.NET Core app.

There are pros and cons to each approach. If an app's environment is already using Envoy as a proxy, it might make sense to also use Envoy to provide gRPC-Web support. For a basic solution for gRPC-Web that only requires ASP.NET Core, `Grpc.AspNetCore.Web` is a good choice.

Configure gRPC-Web in ASP.NET Core

gRPC services hosted in ASP.NET Core can be configured to support gRPC-Web alongside HTTP/2 gRPC. gRPC-Web doesn't require any changes to services. The only modification is in setting the middleware in `Program.cs`.

To enable gRPC-Web with an ASP.NET Core gRPC service:

- Add a reference to the Grpc.AspNetCore.Web package.
- Configure the app to use gRPC-Web by adding `UseGrpcWeb` and `EnableGrpcWeb` to `Program.cs`:

C#

```
using GrpcGreeter.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();

var app = builder.Build();

app.UseGrpcWeb();

app.MapGrpcService<GreeterService>().EnableGrpcWeb();
app.MapGet("/", () => "This gRPC service is gRPC-Web enabled and is callable
from browser apps using the gRPC-Web protocol");

app.Run();
```

The preceding code:

- Adds the gRPC-Web middleware, `UseGrpcWeb`, after routing and before endpoints.
- Specifies that the `endpoints.MapGrpcService<GreeterService>()` method supports gRPC-Web with `EnableGrpcWeb`.

Alternatively, the gRPC-Web middleware can be configured so that all services support gRPC-Web by default and `EnableGrpcWeb` isn't required. Specify `new GrpcWebOptions { DefaultEnabled = true }` when the middleware is added.

C#

```
using GrpcGreeter.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();

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

```
app.UseGrpcWeb(new GrpcWebOptions { DefaultEnabled = true });

app.MapGrpcService<GreeterService>().EnableGrpcWeb();
app.MapGet("/", () => "All gRPC service are supported by default in this
example, and are callable from browser apps using the gRPC-Web protocol");

app.Run();
```

> **Note**
>
> There is a known issue that causes gRPC-Web to fail when hosted by HTTP.sys in .NET Core 3.x.
>
> A workaround to get gRPC-Web working on HTTP.sys is available in Grpc-web experimental and UseHttpSys()? (grpc/grpc-dotnet #853).

gRPC-Web and CORS

Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction applies to making gRPC-Web calls with browser apps. For example, a browser app served by `https://www.contoso.com` is blocked from calling gRPC-Web services hosted on `https://services.contoso.com`. Cross-Origin Resource Sharing (CORS) can be used to relax this restriction.

To allow a browser app to make cross-origin gRPC-Web calls, set up CORS in ASP.NET Core. Use the built-in CORS support, and expose gRPC-specific headers with WithExposedHeaders.

C#

```
using GrpcGreeter.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();

builder.Services.AddCors(o => o.AddPolicy("AllowAll", builder =>
{
    builder.AllowAnyOrigin()
           .AllowAnyMethod()
           .AllowAnyHeader()
           .WithExposedHeaders("Grpc-Status", "Grpc-Message", "Grpc-
Encoding", "Grpc-Accept-Encoding");
}));

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

```
app.UseGrpcWeb();
app.UseCors();

app.MapGrpcService<GreeterService>().EnableGrpcWeb()
                                    .RequireCors("AllowAll");

app.MapGet("/", () => "This gRPC service is gRPC-Web enabled, CORS enabled,
and is callable from browser apps using the gRPC-Web protocol");

app.Run();
```

The preceding code:

- Calls `AddCors` to add CORS services and configure a CORS policy that exposes gRPC-specific headers.
- Calls `UseCors` to add the CORS middleware after routing configuration and before endpoints configuration.
- Specifies that the `endpoints.MapGrpcService<GreeterService>()` method supports CORS with `RequireCors`.

gRPC-Web and streaming

Traditional gRPC over HTTP/2 supports client, server and bidirectional streaming. gRPC-Web offers limited support for streaming:

- gRPC-Web browser clients don't support calling client streaming and bidirectional streaming methods.
- gRPC-Web .NET clients don't support calling client streaming and bidirectional streaming methods over HTTP/1.1.
- ASP.NET Core gRPC services hosted on Azure App Service and IIS don't support bidirectional streaming.

When using gRPC-Web, we only recommend the use of unary methods and server streaming methods.

HTTP protocol

The ASP.NET Core gRPC service template, included in the .NET SDK, creates an app that's only configured for HTTP/2. This is a good default when an app only supports traditional gRPC over HTTP/2. gRPC-Web, however, works with both HTTP/1.1 and HTTP/2. Some platforms, such as UWP or Unity, can't use HTTP/2. To support all client apps, configure the server to enable HTTP/1.1 and HTTP/2.

Update the default protocol in `appsettings.json`:

JSON

```
{
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1AndHttp2"
    }
  }
}
```

Alternatively, configure Kestrel endpoints in startup code.

Enabling HTTP/1.1 and HTTP/2 on the same port requires TLS for protocol negotiation. For more information, see ASP.NET Core gRPC protocol negotiation.

Call gRPC-Web from the browser

Browser apps can use gRPC-Web to call gRPC services. There are some requirements and limitations when calling gRPC services with gRPC-Web from the browser:

- The server must contain configuration to support gRPC-Web.
- Client streaming and bidirectional streaming calls aren't supported. Server streaming is supported.
- Calling gRPC services on a different domain requires CORS configuration on the server.

JavaScript gRPC-Web client

A JavaScript gRPC-Web client exists. For instructions on how to use gRPC-Web from JavaScript, see write JavaScript client code with gRPC-Web.

Configure gRPC-Web with the .NET gRPC client

The .NET gRPC client can be configured to make gRPC-Web calls. This is useful for Blazor WebAssembly apps, which are hosted in the browser and have the same HTTP limitations of JavaScript code. Calling gRPC-Web with a .NET client is the same as HTTP/2 gRPC. The only modification is how the channel is created.

To use gRPC-Web:

- Add a reference to the Grpc.Net.Client.Web package.

- Ensure the reference to Grpc.Net.Client package is version 2.29.0 or later.
- Configure the channel to use the `GrpcWebHandler`:

```
C#
```

```
var channel = GrpcChannel.ForAddress("https://localhost:53305", new
GrpcChannelOptions
{
    HttpHandler = new GrpcWebHandler(new HttpClientHandler())
});

var client = new Greeter.GreeterClient(channel);
var response = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
```

The preceding code:

- Configures a channel to use gRPC-Web.
- Creates a client and makes a call using the channel.

`GrpcWebHandler` has the following configuration options:

- `InnerHandler`: The underlying HttpMessageHandler that makes the gRPC HTTP request, for example, `HttpClientHandler`.
- `GrpcWebMode`: An enumeration type that specifies whether the gRPC HTTP request `Content-Type` is `application/grpc-web` or `application/grpc-web-text`.
  - `GrpcWebMode.GrpcWeb` configures sending content without encoding. Default value.
  - `GrpcWebMode.GrpcWebText` configures base64-encoded content. Required for server streaming calls in browsers.
- `HttpVersion`: HTTP protocol `Version` used to set HttpRequestMessage.Version on the underlying gRPC HTTP request. gRPC-Web doesn't require a specific version and doesn't override the default unless specified.

> **ⓘ Important**
>
> Generated gRPC clients have synchronous and asynchronous methods for calling unary methods. For example, `SayHello` is synchronous, and `SayHelloAsync` is asynchronous. Asynchronous methods are always required in Blazor WebAssembly. Calling a synchronous method in a Blazor WebAssembly app causes the app to become unresponsive.

Use gRPC client factory with gRPC-Web

Create a .NET client compatible with gRPC-Web using the gRPC client factory:

- Add package references to the project file for the following packages:
  - Grpc.Net.Client.Web
  - Grpc.Net.ClientFactory
- Register a gRPC client with dependency injection (DI) using the generic `AddGrpcClient` extension method. In a Blazor WebAssembly app, services are registered with DI in `Program.cs`.
- Configure `GrpcWebHandler` using the ConfigurePrimaryHttpMessageHandler extension method.

C#

```
builder.Services
    .AddGrpcClient<Greet.GreeterClient>(options =>
    {
        options.Address = new Uri("https://localhost:5001");
    })
    .ConfigurePrimaryHttpMessageHandler(
        () => new GrpcWebHandler(new HttpClientHandler()));
```

For more information, see gRPC client factory integration in .NET.

Additional resources

- gRPC for Web Clients GitHub project
- Enable Cross-Origin Requests (CORS) in ASP.NET Core
- gRPC JSON transcoding in ASP.NET Core gRPC apps

gRPC JSON transcoding 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

gRPC is a high-performance Remote Procedure Call (RPC) framework. gRPC uses HTTP/2, streaming, Protobuf, and message contracts to create high-performance, real-time services.

One limitation with gRPC is that not every platform can use it. Browsers don't fully support HTTP/2, making REST APIs and JSON the primary way to get data into browser apps. Despite the benefits that gRPC brings, REST APIs and JSON have an important place in modern apps. Building gRPC ***and*** JSON Web APIs adds unwanted overhead to app development.

This document discusses how to create JSON Web APIs using gRPC services.

Overview

gRPC JSON transcoding is an extension for ASP.NET Core that creates RESTful JSON APIs for gRPC services. Once configured, transcoding allows apps to call gRPC services with familiar HTTP concepts:

- HTTP verbs
- URL parameter binding
- JSON requests/responses

gRPC can still be used to call services.

> **Note**

gRPC JSON transcoding replaces gRPC HTTP API, an alternative experimental extension.

Usage

1. Add a package reference to Microsoft.AspNetCore.Grpc.JsonTranscoding.
2. Register transcoding in server startup code by adding `AddJsonTranscoding`: In the `Program.cs` file, change `builder.Services.AddGrpc();` to `builder.Services.AddGrpc().AddJsonTranscoding();`.
3. Add `<IncludeHttpRuleProtos>true</IncludeHttpRuleProtos>` to the property group in the project file (`.csproj`):

JSON

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>true</InvariantGlobalization>
    <IncludeHttpRuleProtos>true</IncludeHttpRuleProtos>
  </PropertyGroup>
```

4. Annotate gRPC methods in your `.proto` files with HTTP bindings and routes:

ProtoBuf

```
syntax = "proto3";

option csharp_namespace = "GrpcServiceTranscoding";
import "google/api/annotations.proto";

package greet;

// The greeting service definition.
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
      get: "/v1/greeter/{name}"
    };
  }
}

// The request message containing the user's name.
```

```
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```

The `SayHello` gRPC method can now be invoked as gRPC and as a JSON Web API:

- Request: `GET /v1/greeter/world`
- Response: `{ "message": "Hello world" }`

If the server is configured to write logs for each request, server logs show that a gRPC service executes the HTTP call. Transcoding maps the incoming HTTP request to a gRPC message and converts the response message to JSON.

text

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:5001/v1/greeter/world
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /v1/greeter/{name}'
info: Server.GreeterService[0]
      Sending hello to world
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /v1/greeter/{name}'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 1.996ms 200 application/json
```

Annotate gRPC methods

gRPC methods must be annotated with an HTTP rule before they support transcoding. The HTTP rule includes information about how to call the gRPC method, such as the HTTP method and route.

ProtoBuf

```
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
      get: "/v1/greeter/{name}"
    };
  }
}
```

The proceeding example:

- Defines a `Greeter` service with a `SayHello` method. The method has an HTTP rule specified using the name `google.api.http`.
- The method is accessible with `GET` requests and the `/v1/greeter/{name}` route.
- The `name` field on the request message is bound to a route parameter.

Many options are available for customizing how a gRPC method binds to a RESTful API. For more information about annotating gRPC methods and customizing JSON, see Configure HTTP and JSON for gRPC JSON transcoding.

Streaming methods

Traditional gRPC over HTTP/2 supports streaming in all directions. Transcoding is limited to server streaming only. Client streaming and bidirectional streaming methods aren't supported.

Server streaming methods use line-delimited JSON. Each message written using `WriteAsync` is serialized to JSON and followed by a new line.

The following server streaming method writes three messages:

C#

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    for (var i = 1; i <= 3; i++)
    {
        await responseStream.WriteAsync(new ExampleResponse { Text =
$"Message {i}" });
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

The client receives three line-delimited JSON objects:

JSON

```
{"Text":"Message 1"}
{"Text":"Message 2"}
{"Text":"Message 3"}
```

Note that the `WriteIndented` JSON setting doesn't apply to server streaming methods. Pretty printing adds new lines and whitespace to JSON, which can't be used with line-delimited JSON.

View or download an ASP.NET Core gPRC transcoding and streaming app sample.

HTTP protocol

The ASP.NET Core gRPC service template, included in the .NET SDK, creates an app that's only configured for HTTP/2. HTTP/2 is a good default when an app only supports traditional gRPC over HTTP/2. Transcoding, however, works with both HTTP/1.1 and HTTP/2. Some platforms, such as UWP or Unity, can't use HTTP/2. To support all client apps, configure the server to enable HTTP/1.1 and HTTP/2.

Update the default protocol in `appsettings.json`:

JSON

```
{
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1AndHttp2"
    }
  }
}
```

Alternatively, configure Kestrel endpoints in startup code.

Enabling HTTP/1.1 and HTTP/2 on the same port requires TLS for protocol negotiation. For more information about configuring HTTP protocols in a gRPC app, see ASP.NET Core gRPC protocol negotiation.

gRPC JSON transcoding vs gRPC-Web

Both transcoding and gRPC-Web allow gRPC services to be called from a browser. However, the way each does this is different:

- gRPC-Web lets browser apps call gRPC services from the browser with the gRPC-Web client and Protobuf. gRPC-Web requires the browser app to generate a gRPC client and has the advantage of sending small, fast Protobuf messages.
- 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.

The previous `Greeter` service can be called using browser JavaScript APIs:

JavaScript

```
var name = nameInput.value;

fetch('/v1/greeter/' + name)
  .then((response) => response.json())
  .then((result) => {
    console.log(result.message);
    // Hello world
  });
```

grpc-gateway

grpc-gateway is another technology for creating RESTful JSON APIs from gRPC services. It uses the same `.proto` annotations to map HTTP concepts to gRPC services.

grpc-gateway uses code generation to create a reverse-proxy server. The reverse proxy translates RESTful calls into gRPC+Protobuf and sends the calls over HTTP/2 to the gRPC service. The benefit of this approach is the gRPC service doesn't know about the RESTful JSON APIs. Any gRPC server can use grpc-gateway.

Meanwhile, gRPC JSON transcoding runs inside an ASP.NET Core app. It deserializes JSON into Protobuf messages, then invokes the gRPC service directly. Transcoding in ASP.NET Core offers advantages to .NET app developers:

- Less complex: Both gRPC services and mapped RESTful JSON API run out of one ASP.NET Core app.
- Better performance: Transcoding deserializes JSON to Protobuf messages and invokes the gRPC service directly. There are significant performance benefits in doing this in-process versus making a new gRPC call to a different server.
- Lower cost: Fewer servers result in a smaller monthly hosting bill.

For installation and usage of grpc-gateway, see the grpc-gateway README.

Additional resources

- Configure HTTP and JSON for gRPC JSON transcoding ASP.NET Core apps
- Use OpenAPI with gRPC JSON transcoding ASP.NET Core apps
- Use gRPC in browser apps
- gRPC-Web in ASP.NET Core gRPC apps

Configure HTTP and JSON for gRPC JSON transcoding

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

gRPC JSON transcoding creates RESTful JSON web APIs from gRPC methods. It uses annotations and options for customizing how a RESTful API maps to the gRPC methods.

HTTP rules

gRPC methods must be annotated with an HTTP rule before they support transcoding. The HTTP rule includes information about calling the gRPC method as a RESTful API, such as the HTTP method and route.

ProtoBuf

```
import "google/api/annotations.proto";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
      get: "/v1/greeter/{name}"
    };
  }
}
```

An HTTP rule is:

- An annotation on gRPC methods.
- Identified by the name `google.api.http`.
- Imported from the `google/api/annotations.proto` file. The google/api/http.proto and google/api/annotations.proto files need to be in the project.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

HTTP method

The HTTP method is specified by setting the route to the matching HTTP method field name:

- `get`
- `put`
- `post`
- `delete`
- `patch`

The `custom` field allows for other HTTP methods.

In the following example, the `CreateAddress` method is mapped to `POST` with the specified route:

ProtoBuf

```
service Address {
  rpc CreateAddress (CreateAddressRequest) returns (CreateAddressReply) {
    option (google.api.http) = {
      post: "/v1/address",
      body: "*"
    };
  }
}
```

Route

gRPC JSON transcoding routes support route parameters. For example, `{name}` in a route binds to the `name` field on the request message.

To bind a field on a nested message, specify the path to the field. In the following example, `{params.org}` binds to the `org` field on the `IssueParams` message:

ProtoBuf

```
service Repository {
  rpc GetIssue (GetIssueRequest) returns (GetIssueReply) {
    option (google.api.http) = {
      get: "/{apiVersion}/{params.org}/{params.repo}/issue/{params.issueId}"
    };
  }
}

message GetIssueRequest {
  int32 api_version = 1;
  IssueParams params = 2;
}
message IssueParams {
  string org = 1;
  string repo = 2;
  int32 issueId = 3;
}
```

Transcoding routes and ASP.NET Core routes have a similar syntax and feature set. However, some ASP.NET Core routing features aren't supported by transcoding. These include:

- Route constraints
- Default values
- Optional parameters
- Complex segments

Request body

Transcoding deserializes the request body JSON to the request message. The `body` field specifies how the HTTP request body maps to the request message. The value is either the name of the request field whose value is mapped to the HTTP request body or `*` for mapping all request fields.

In the following example, the HTTP request body is deserialized to the `address` field:

ProtoBuf

```
service Address {
  rpc AddAddress (AddAddressRequest) returns (AddAddressReply) {
    option (google.api.http) = {
      post: "/{apiVersion}/address",
      body: "address"
    };
  }
}
```

```
message AddAddressRequest {
  int32 api_version = 1;
  Address address = 2;
}
message Address {
  string street = 1;
  string city = 2;
  string country = 3;
}
```

Query parameters

Any fields in the request message that aren't bound by route parameters or the request body can be set using HTTP query parameters.

ProtoBuf

```
service Repository {
  rpc GetIssues (GetIssuesRequest) returns (GetIssuesReply) {
    option (google.api.http) = {
      get: "/v1/{org}/{repo}/issue"
    };
  }
}

message GetIssuesRequest {
  string org = 1;
  string repo = 2;
  string text = 3;
  PageParams page = 4;
}
message PageParams {
  int32 index = 1;
  int32 size = 2;
}
```

In the preceding example:

- `org` and `repo` fields are bound from route parameters.
- Other fields, such as `text` and the nested fields from `page`, can be bound from the query string: `?text=value&page.index=0&page.size=10`

Response body

By default, transcoding serializes the entire response message as JSON. The `response_body` field allows serialization of a subset of the response message.

ProtoBuf

```
service Address {
  rpc GetAddress (GetAddressRequest) returns (GetAddressReply) {
    option (google.api.http) = {
      get: "/v1/address/{id}",
      response_body: "address"
    };
  }
}

message GetAddressReply {
  int32 version = 1;
  Address address = 2;
}
message Address {
  string street = 1;
  string city = 2;
  string country = 3;
}
```

In the preceding example, the `address` field is serialized to the response body as JSON.

Specification

For more information about customizing gRPC transcoding, see the HttpRule specification.

Customize JSON

Messages are converted to and from JSON using the JSON mapping in the Protobuf specification. Protobuf's JSON mapping is a standardized way to convert between JSON and Protobuf, and all serialization follows these rules.

However, gRPC JSON transcoding offers some limited options for customizing JSON with GrpcJsonSettings, as shown in the following table.

Expand table

Option	Default Value	Description
IgnoreDefaultValues	`false`	If set to `true`, fields with default values are ignored during serialization.
WriteEnumsAsIntegers	`false`	If set to `true`, enum values are written as integers instead of

Option	Default Value	Description
		strings.
WriteInt64sAsStrings	`false`	If set to `true`, `Int64` and `UInt64` values are written as strings instead of numbers.
WriteIndented	`false`	If set to `true`, JSON is written using pretty printing. This option doesn't affect streaming methods, which write line-delimited JSON messages and can't use pretty printing.

C#

```
builder.Services.AddGrpc().AddJsonTranscoding(o =>
{
    o.JsonSettings.WriteIndented = true;
});
```

In the `.proto` file, the `json_name` field option customizes a field's name when it's serialized as JSON, as in the following example:

ProtoBuf

```
message TestMessage {
  string my_field = 1 [json_name="customFieldName"];
}
```

Transcoding doesn't support advanced JSON customization. Apps requiring precise JSON structure control should consider using ASP.NET Core Web API.

Additional resources

- gRPC JSON transcoding in ASP.NET Core gRPC apps
- HttpRule specification

gRPC JSON transcoding documentation with Swagger / OpenAPI

Article • 09/23/2024

> **ⓘ 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

OpenAPI (Swagger) is a language-agnostic specification for describing REST APIs. gRPC JSON transcoding supports generating OpenAPI from transcoded RESTful APIs. The Microsoft.AspNetCore.Grpc.Swagger package:

- Integrates gRPC JSON transcoding with Swashbuckle.
- Is experimental in .NET 7 to allow us to explore the best way to provide OpenAPI support.

Get started

To enable OpenAPI with gRPC JSON transcoding:

1. Add a package reference to Microsoft.AspNetCore.Grpc.Swagger. The version must be 0.3.0-xxx or later.
2. Configure Swashbuckle in startup. The `AddGrpcSwagger` method configures Swashbuckle to include gRPC endpoints.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddGrpc().AddJsonTranscoding();
builder.Services.AddGrpcSwagger();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1",
        new OpenApiInfo { Title = "gRPC transcoding", Version = "v1" });
});
```

```
var app = builder.Build();
app.UseSwagger();
if (app.Environment.IsDevelopment())
{
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });
}
app.MapGrpcService<GreeterService>();

app.Run();
```

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Add OpenAPI descriptions from `.proto` comments

Generate OpenAPI descriptions from comments in the `.proto` contract, as in the following example:

ProtoBuf

```
// My amazing greeter service.
service Greeter {
  // Sends a greeting.
  rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
      get: "/v1/greeter/{name}"
    };
  }
}

message HelloRequest {
  // Name to say hello to.
  string name = 1;
}
message HelloReply {
  // Hello reply message.
  string message = 1;
}
```

To enable gRPC OpenAPI comments:

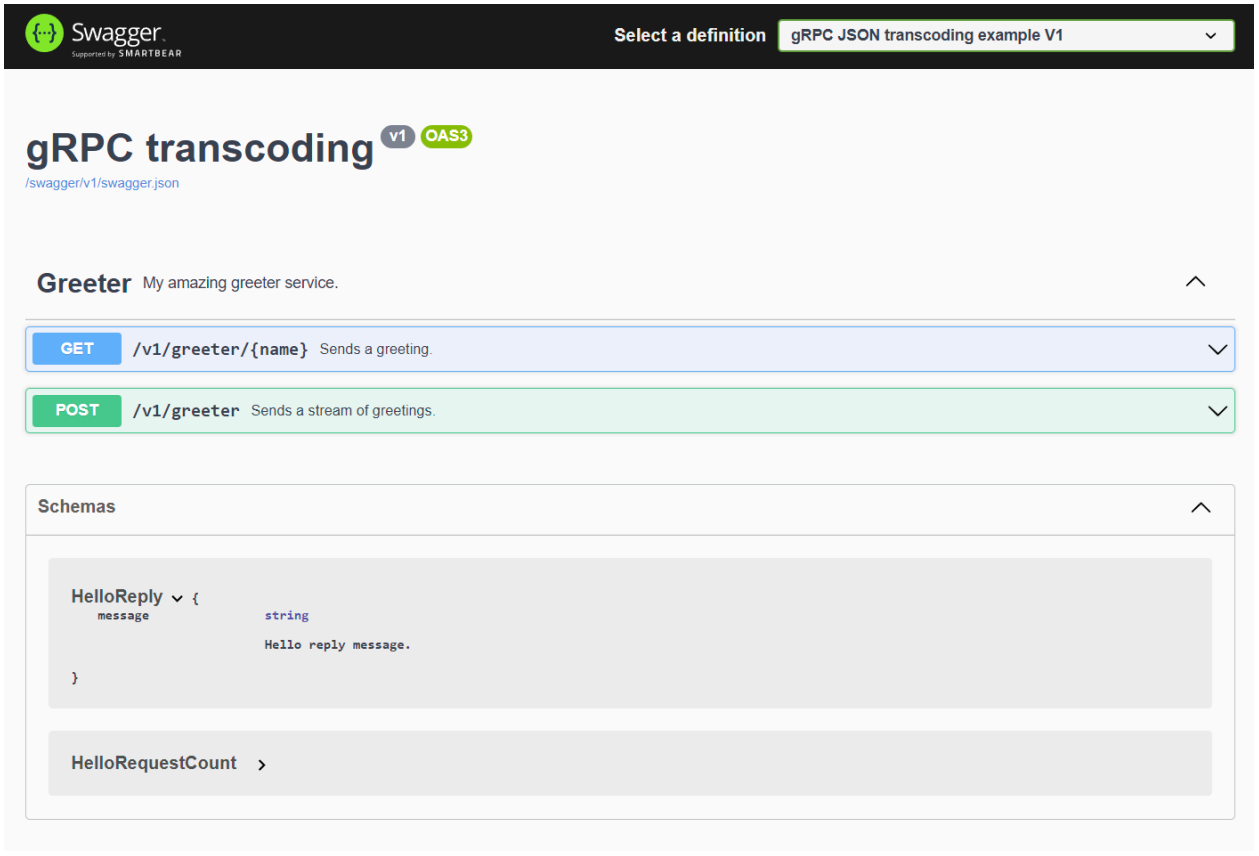
1. Enable the XML documentation file in the server project with `<GenerateDocumentationFile>true</GenerateDocumentationFile>`.
2. Configure `AddSwaggerGen` to read the generated XML file. Pass the XML file path to `IncludeXmlComments` and `IncludeGrpcXmlComments`, as in the following example:

C#

```
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1",
        new OpenApiInfo { Title = "gRPC transcoding", Version = "v1" });

    var filePath = Path.Combine(System.AppContext.BaseDirectory,
"Server.xml");
    c.IncludeXmlComments(filePath);
    c.IncludeGrpcXmlComments(filePath, includeControllerXmlComments: true);
});
```

To confirm that Swashbuckle is generating OpenAPI with descriptions for the RESTful gRPC services, start the app and navigate to the Swagger UI page:



Additional resources

- gRPC JSON transcoding in ASP.NET Core gRPC apps

- OpenAPI homepage
- Swashbuckle.AspNetCore GitHub repository

gRPC for .NET configuration

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.

Configure services options

gRPC services are configured with `AddGrpc` in `Startup.cs`. Configuration options are in the Grpc.AspNetCore.Server package.

The following table describes options for configuring gRPC services:

Expand table

Option	Default Value	Description
`MaxSendMessageSize`	`null`	The maximum message size in bytes that can be sent from the server. Attempting to send a message that exceeds the configured maximum message size results in an exception. When set to `null`, the message size is unlimited.
`MaxReceiveMessageSize`	4 MB	The maximum message size in bytes that can be received by the server. If the server receives a message that exceeds this limit, it throws an exception. Increasing this value allows the server to receive larger messages, but can negatively impact memory consumption. When set to `null`, the message size is unlimited.
`EnableDetailedErrors`	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a service method. The default is `false`. Setting `EnableDetailedErrors` to `true` can leak sensitive information.

Option	Default Value	Description
`CompressionProviders`	gzip	A collection of compression providers used to compress and decompress messages. Custom compression providers can be created and added to the collection. The default configured providers support **gzip** compression.
`ResponseCompressionAlgorithm`	`null`	The compression algorithm used to compress messages sent from the server. The algorithm must match a compression provider in `CompressionProviders`. For the algorithm to compress a response, the client must indicate it supports the algorithm by sending it in the **grpc-accept-encoding** header.
`ResponseCompressionLevel`	`null`	The compress level used to compress messages sent from the server.
`Interceptors`	None	A collection of interceptors that are run with each gRPC call. Interceptors are run in the order they are registered. Globally configured interceptors are run before interceptors configured for a single service. Interceptors have a per-request lifetime by default. The interceptor constructor is called and parameters are resolved from dependency injection (DI). An interceptor type can also be registered with DI to override how it is created and its lifetime. Interceptors offer similar functionalities compared to ASP.NET Core middleware. For more information, see gRPC Interceptors vs. Middleware.
`IgnoreUnknownServices`	`false`	If `true`, calls to unknown services and methods don't return an **UNIMPLEMENTED** status, and the request passes to the next registered middleware in ASP.NET Core.

Options can be configured for all services by providing an options delegate to the `AddGrpc` call in `Startup.ConfigureServices`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.EnableDetailedErrors = true;
```

```
        options.MaxReceiveMessageSize = 2 * 1024 * 1024; // 2 MB
        options.MaxSendMessageSize = 5 * 1024 * 1024; // 5 MB
    });
}
```

Options for a single service override the global options provided in `AddGrpc` and can be configured using `AddServiceOptions<TService>`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc().AddServiceOptions<MyService>(options =>
    {
        options.MaxReceiveMessageSize = 2 * 1024 * 1024; // 2 MB
        options.MaxSendMessageSize = 5 * 1024 * 1024; // 5 MB
    });
}
```

Service interceptors have a per-request lifetime by default. Registering the interceptor type with DI overrides how an interceptor is created and its lifetime.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.Interceptors.Add<LoggingInterceptor>();
    });
    services.AddSingleton<LoggingInterceptor>();
}
```

ASP.NET Core server options

`Grpc.AspNetCore.Server` is hosted by an ASP.NET Core web server. There are a number of options for ASP.NET Core servers, including Kestrel, IIS and HTTP.sys. Each server offers additional options for how HTTP requests are served.

The server used by an ASP.NET Core app is configured in app startup code. The default server is Kestrel.

For more information about the different servers and their configuration options, see:

- Kestrel web server in ASP.NET Core
- HTTP.sys web server implementation in ASP.NET Core

- Host ASP.NET Core on Windows with IIS

Configure client options

gRPC client configuration is set on `GrpcChannelOptions`. Configuration options are in the Grpc.Net.Client package.

The following table describes options for configuring gRPC channels:

Expand table

Option	Default Value	Description
`HttpHandler`	New instance	The `HttpMessageHandler` used to make gRPC calls. A client can be set to configure a custom `HttpClientHandler` or add additional handlers to the HTTP pipeline for gRPC calls. If no `HttpMessageHandler` is specified, a new `HttpClientHandler` instance is created for the channel with automatic disposal.
`HttpClient`	`null`	The `HttpClient` used to make gRPC calls. This setting is an alternative to `HttpHandler`.
`DisposeHttpClient`	`false`	If set to `true` and an `HttpMessageHandler` or `HttpClient` is specified, then either the `HttpHandler` or `HttpClient`, respectively, is disposed when the `GrpcChannel` is disposed.
`LoggerFactory`	`null`	The `LoggerFactory` used by the client to log information about gRPC calls. A `LoggerFactory` instance can be resolved from dependency injection or created using `LoggerFactory.Create`. For examples of configuring logging, see Logging and diagnostics in gRPC on .NET.
`MaxSendMessageSize`	`null`	The maximum message size in bytes that can be sent from the client. Attempting to send a message that exceeds the configured maximum message size results in an exception. When set to `null`, the message size is unlimited.

Option	Default Value	Description
`MaxReceiveMessageSize`	4 MB	The maximum message size in bytes that can be received by the client. If the client receives a message that exceeds this limit, it throws an exception. Increasing this value allows the client to receive larger messages, but can negatively impact memory consumption. When set to `null`, the message size is unlimited.
`Credentials`	`null`	A `ChannelCredentials` instance. Credentials are used to add authentication metadata to gRPC calls.
`CompressionProviders`	gzip	A collection of compression providers used to compress and decompress messages. Custom compression providers can be created and added to the collection. The default configured providers support **gzip** compression.
`ThrowOperationCanceledOnCancellation`	`false`	If set to `true`, clients throw OperationCanceledException when a call is canceled or its deadline is exceeded.
`UnsafeUseInsecureChannelCallCredentials`	`false`	If set to `true`, `CallCredentials` are applied to gRPC calls made by an insecure channel. Sending authentication headers over an insecure connection has security implications and shouldn't be done in production environments.
`MaxRetryAttempts`	5	The maximum retry attempts. This value limits any retry and hedging attempt values specified in the service config. Setting this value alone doesn't enable retries. Retries are enabled in the service config, which can be done using `ServiceConfig`. A `null` value removes the maximum retry attempts limit. For more information about retries, see Transient fault handling with gRPC retries.
`MaxRetryBufferSize`	16 MB	The maximum buffer size in bytes that can be used to store sent messages when retrying or hedging calls. If the buffer limit is exceeded, then no more retry attempts are made and all hedging calls but one

Option	Default Value	Description
		will be canceled. This limit is applied across all calls made using the channel. A `null` value removes the maximum retry buffer size limit.
`MaxRetryBufferPerCallSize`	1 MB	The maximum buffer size in bytes that can be used to store sent messages when retrying or hedging calls. If the buffer limit is exceeded, then no more retry attempts are made and all hedging calls but one will be canceled. This limit is applied to one call. A `null` value removes the maximum retry buffer size limit per call.
`ServiceConfig`	`null`	The service config for a gRPC channel. A service config can be used to configure gRPC retries.

The following code:

- Sets the maximum send and receive message size on the channel.
- Creates a client.

C#

```
static async Task Main(string[] args)
{
    var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
    {
        MaxReceiveMessageSize = 5 * 1024 * 1024, // 5 MB
        MaxSendMessageSize = 2 * 1024 * 1024 // 2 MB
    });
    var client = new Greeter.GreeterClient(channel);

    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
}
```

Note that client interceptors aren't configured with `GrpcChannelOptions`. Instead, client interceptors are configured using the `Intercept` extension method with a channel. This extension method is in the `Grpc.Core.Interceptors` namespace.

C#

```
static async Task Main(string[] args)
{
    var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var callInvoker = channel.Intercept(new LoggingInterceptor());
    var client = new Greeter.GreeterClient(callInvoker);

    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
}
```

System.Net handler options

`Grpc.Net.Client` uses a HTTP transport derived from `HttpMessageHandler` to make HTTP requests. Each handler offers additional options for how HTTP requests are made.

The handler is configured on a channel and can be overridden by setting `GrpcChannelOptions.HttpHandler`. .NET Core 3 and .NET 5 or later uses SocketsHttpHandler by default. gRPC client apps on .NET Framework should configure WinHttpHandler.

For more information about the different handlers and their configuration options, see:

- System.Net.Http.SocketsHttpHandler
- System.Net.Http.WinHttpHandler

Additional resources

- gRPC services with ASP.NET Core
- Call gRPC services with the .NET client
- Logging and diagnostics in gRPC on .NET
- Create a .NET Core gRPC client and server in ASP.NET Core

Authentication and authorization in gRPC 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 James Newton-King

View or download sample code (how to download)

Authenticate users calling a gRPC service

gRPC can be used with ASP.NET Core authentication to associate a user with each call.

The following is an example of `Program.cs` which uses gRPC and ASP.NET Core authentication:

C#

```
app.UseRouting();

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

app.MapGrpcService<GreeterService>();
```

> **Note**
>
> The order in which you register the ASP.NET Core authentication middleware matters. Always call `UseAuthentication` and `UseAuthorization` after `UseRouting` and before `UseEndpoints`.

The authentication mechanism your app uses during a call needs to be configured. Authentication configuration is added in `Program.cs` and will be different depending

upon the authentication mechanism your app uses.

Once authentication has been setup, the user can be accessed in a gRPC service methods via the `ServerCallContext`.

C#

```
public override Task<BuyTicketsResponse> BuyTickets(
    BuyTicketsRequest request, ServerCallContext context)
{
    var user = context.GetHttpContext().User;

    // ... access data from ClaimsPrincipal ...
}
```

Bearer token authentication

The client can provide an access token for authentication. The server validates the token and uses it to identify the user.

On the server, bearer token authentication is configured using the JWT Bearer middleware.

In the .NET gRPC client, the token can be sent with calls by using the `Metadata` collection. Entries in the `Metadata` collection are sent with a gRPC call as HTTP headers:

C#

```
public bool DoAuthenticatedCall(
    Ticketer.TicketerClient client, string token)
{
    var headers = new Metadata();
    headers.Add("Authorization", $"Bearer {token}");

    var request = new BuyTicketsRequest { Count = 1 };
    var response = await client.BuyTicketsAsync(request, headers);

    return response.Success;
}
```

Set the bearer token with `CallCredentials`

Configuring `ChannelCredentials` on a channel is an alternative way to send the token to the service with gRPC calls. A `ChannelCredentials` can include `CallCredentials`, which

provide a way to automatically set `Metadata`.

Benefits of using `CallCredentials`:

- Authentication is centrally configured on the channel. The token doesn't need to be manually provided to the gRPC call.
- The `CallCredentials.FromInterceptor` callback is asynchronous. Call credentials can fetch a credential token from an external system if required. Asynchronous methods inside the callback should use the `CancellationToken` on `AuthInterceptorContext`.

> **Note**
>
> `CallCredentials` are only applied if the channel is secured with TLS. Sending authentication headers over an insecure connection has security implications and shouldn't be done in production environments. An app can configure a channel to ignore this behavior and always use `CallCredentials` by setting `UnsafeUseInsecureChannelCallCredentials` on a channel.

The credential in the following example configures the channel to send the token with every gRPC call:

C#

```
private static GrpcChannel CreateAuthenticatedChannel(ITokenProvder
tokenProvider)
{
    var credentials = CallCredentials.FromInterceptor(async (context,
metadata) =>
    {
        var token = await
tokenProvider.GetTokenAsync(context.CancellationToken);
        metadata.Add("Authorization", $"Bearer {token}");
    });

    var channel = GrpcChannel.ForAddress(address, new GrpcChannelOptions
    {
        Credentials = ChannelCredentials.Create(new SslCredentials(),
credentials)
    });
    return channel;
}
```

Bearer token with gRPC client factory

gRPC client factory can create clients that send a bearer token using `AddCallCredentials`. This method is available in Grpc.Net.ClientFactory version 2.46.0 or later.

The delegate passed to `AddCallCredentials` is executed for each gRPC call:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddCallCredentials((context, metadata) =>
    {
        if (!string.IsNullOrEmpty(_token))
        {
            metadata.Add("Authorization", $"Bearer {_token}");
        }
        return Task.CompletedTask;
    });
```

Dependency injection (DI) can be combined with `AddCallCredentials`. An overload passes `IServiceProvider` to the delegate, which can be used to get a service constructed from DI using scoped and transient services.

Consider an app that has:

- A user-defined `ITokenProvider` for getting a bearer token. `ITokenProvider` is registered in DI with a scoped lifetime.
- gRPC client factory is configured to create clients that are injected into gRPC services and Web API controllers.
- gRPC calls should use `ITokenProvider` to get a bearer token.

C#

```
public interface ITokenProvider
{
    Task<string> GetTokenAsync(CancellationToken cancellationToken);
}

public class AppTokenProvider : ITokenProvider
{
    private string _token;

    public async Task<string> GetTokenAsync(CancellationToken
cancellationToken)
    {
```

```
        if (_token == null)
        {
            // App code to resolve the token here.
        }

        return _token;
    }
}
```

C#

```
builder.Services.AddScoped<ITokenProvider, AppTokenProvider>();

builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddCallCredentials(async (context, metadata, serviceProvider) =>
    {
        var provider = serviceProvider.GetRequiredService<ITokenProvider>();
        var token = await provider.GetTokenAsync(context.CancellationToken);
        metadata.Add("Authorization", $"Bearer {token}");
    }));
```

The preceding code:

- Defines `ITokenProvider` and `AppTokenProvider`. These types handle resolving the authentication token for gRPC calls.
- Registers the `AppTokenProvider` type with DI in a scoped lifetime. `AppTokenProvider` caches the token so that only the first call in the scope is required to calculate it.
- Registers the `GreeterClient` type with client factory.
- Configures `AddCallCredentials` for this client. The delegate is executed each time a call is made and adds the token returned by `ITokenProvider` to the metadata.

Client certificate authentication

A client could alternatively provide a client certificate for authentication. Certificate authentication happens at the TLS level, long before it ever gets to ASP.NET Core. When the request enters ASP.NET Core, the client certificate authentication package allows you to resolve the certificate to a `ClaimsPrincipal`.

> **Note**

Configure the server to accept client certificates. For information on accepting client certificates in Kestrel, IIS, and Azure, see Configure certificate authentication in ASP.NET Core.

In the .NET gRPC client, the client certificate is added to `HttpClientHandler` that is then used to create the gRPC client:

C#

```
public Ticketer.TicketerClient CreateClientWithCert(
    string baseAddress,
    X509Certificate2 certificate)
{
    // Add client cert to the handler
    var handler = new HttpClientHandler();
    handler.ClientCertificates.Add(certificate);

    // Create the gRPC channel
    var channel = GrpcChannel.ForAddress(baseAddress, new GrpcChannelOptions
    {
        HttpHandler = handler
    });

    return new Ticketer.TicketerClient(channel);
}
```

Other authentication mechanisms

Many ASP.NET Core supported authentication mechanisms work with gRPC:

- Microsoft Entra ID
- Client Certificate
- IdentityServer
- JWT Token
- OAuth 2.0
- OpenID Connect
- WS-Federation

For more information on configuring authentication on the server, see ASP.NET Core authentication.

Configuring the gRPC client to use authentication will depend on the authentication mechanism you are using. The previous bearer token and client certificate examples show a couple of ways the gRPC client can be configured to send authentication metadata with gRPC calls:

- Strongly typed gRPC clients use `HttpClient` internally. Authentication can be configured on HttpClientHandler, or by adding custom HttpMessageHandler instances to the `HttpClient`.
- Each gRPC call has an optional `CallOptions` argument. Custom headers can be sent using the option's headers collection.

> ⓘ Note
>
> Windows Authentication (NTLM/Kerberos/Negotiate) can't be used with gRPC. gRPC requires HTTP/2, and HTTP/2 doesn't support Windows Authentication.

Authorize users to access services and service methods

By default, all methods in a service can be called by unauthenticated users. To require authentication, apply the [Authorize] attribute to the service:

C#

```
[Authorize]
public class TicketerService : Ticketer.TicketerBase
{
}
```

You can use the constructor arguments and properties of the `[Authorize]` attribute to restrict access to only users matching specific authorization policies. For example, if you have a custom authorization policy called `MyAuthorizationPolicy`, ensure that only users matching that policy can access the service using the following code:

C#

```
[Authorize("MyAuthorizationPolicy")]
public class TicketerService : Ticketer.TicketerBase
{
}
```

Individual service methods can have the `[Authorize]` attribute applied as well. If the current user doesn't match the policies applied to **both** the method and the class, an error is returned to the caller:

C#

```
[Authorize]
public class TicketerService : Ticketer.TicketerBase
{
    public override Task<AvailableTicketsResponse> GetAvailableTickets(
        Empty request, ServerCallContext context)
    {
        // ... buy tickets for the current user ...
    }

    [Authorize("Administrators")]
    public override Task<BuyTicketsResponse> RefundTickets(
        BuyTicketsRequest request, ServerCallContext context)
    {
        // ... refund tickets (something only Administrators can do) ..
    }
}
```

Additional resources

- Bearer Token authentication in ASP.NET Core
- Configure Client Certificate authentication in ASP.NET Core
- Configure interceptors in a gRPC client factory in .NET

Error handling with gRPC

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 discusses error handling and gRPC:

- Built-in error handling capabilities using gRPC status codes and error messages.
- Sending complex, structured error information using rich error handling.

Built-in error handling

gRPC calls communicate success or failure with a status code. When a gRPC call completes successfully the server returns an `OK` status to the client. If an error occurs, gRPC returns:

- An error status code, such as `CANCELLED` or `UNAVAILABLE`.
- An optional string error message.

The types commonly used with error handling are:

- `StatusCode`: An enumeration of gRPC status codes. `OK` signals success; other values are failure.
- `Status`: A `struct` that combines a `StatusCode` and an optional string error message. The error message provides further details about what happened.
- `RpcException`: An exception type that has `Status` value. This exception is thrown in gRPC server methods and caught by gRPC clients.

Built-in error handling only supports a status code and string description. To send complex error information from the server to the client, use rich error handling.

Throw server errors

A gRPC server call always returns a status. The server automatically returns `OK` when a method completes successfully.

cs

```
public class GreeterService : GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        return Task.FromResult(new HelloReply { Message = $"Hello
{request.Name}" });
    }

    public override async Task SayHelloStreaming(HelloRequest request,
        IServerStreamWriter<HelloReply> responseStream, ServerCallContext
context)
    {
        for (var i = 0; i < 5; i++)
        {
            await responseStream.WriteAsync(new HelloReply { Message =
$"Hello {request.Name} {i}" });
            await Task.Delay(TimeSpan.FromSeconds(1));
        }
    }
}
```

The preceding code:

- Implements the unary `SayHello` method that completes successfully when it returns a response message.
- Implements the server streaming `SayHelloStreaming` method that completes successfully when the method finished.

Server error status

gRPC methods return an error status code by throwing an exception. When an `RpcException` is thrown on the server, its status code and description is returned to the client:

cs

```
public class GreeterService : GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        if (string.IsNullOrEmpty(request.Name))
```

```
        {
            throw new RpcException(new Status(StatusCode.InvalidArgument,
"Name is required."));
        }
        return Task.FromResult(new HelloReply { Message = $"Hello
{request.Name}" });
    }
}
```

Thrown exception types that aren't `RpcException` also cause the call to fail, but with an `UNKNOWN` status code and a generic message `Exception was thrown by handler`.

`Exception was thrown by handler` is sent to the client rather than the exception message to prevent exposing potentially sensitive information. To see a more descriptive error message in a development environment, configure EnableDetailedErrors.

Handle client errors

When a gRPC client makes a call, the status code is automatically validated when accessing the response. For example, awaiting a unary gRPC call returns the message sent by the server if the call is successful, and throws an `RpcException` if there's a failure. Catch `RpcException` to handle errors in a client:

cs

```
var client = new Greet.GreeterClient(channel);

try
{
    var response = await client.SayHelloAsync(new HelloRequest { Name =
"World" });
    Console.WriteLine("Greeting: " + response.Message);
}
catch (RpcException ex)
{
    Console.WriteLine("Status code: " + ex.Status.StatusCode);
    Console.WriteLine("Message: " + ex.Status.Detail);
}
```

The preceding code:

- Makes a unary gRPC call to the `SayHello` method.
- Writes the response message to the console if it's successful.
- Catches `RpcException` and writes out the error details on failure.

Error scenarios

Errors are represented by `RpcException` with an error status code and optional detail message. `RpcException` is thrown in many scenarios:

- The call failed on the server and the server sent an error status code. For example, the gRPC client started a call that was missing required data from the request message and the server returns an `INVALID_ARGUMENT` status code.
- An error occurred inside the client when making the gRPC call. For example, a client makes a gRPC call, can't connect to the server, and throws an error with a status of `UNAVAILABLE`.
- The CancellationToken passed to the gRPC call is canceled. The gRPC call is stopped and the client throws an error with a status of `CANCELLED`.
- A gRPC call exceeds its configured deadline. The gRPC call is stopped and the client throws an error with a status of `DEADLINE_EXCEEDED`.

Rich error handling

Rich error handling allows complex, structured information to be sent with error messages. For example, validation of incoming message fields that returns a list of invalid field names and descriptions. The google.rpc.Status error model is often used to send complex error information between gRPC apps.

gRPC on .NET supports a rich error model using the Grpc.StatusProto package. This package has methods for creating rich error models on the server and reading them by a client. The rich error model builds on top of gRPC's built-in handling capabilities and they can be used side-by-side.

> **Important**
>
> Errors are included in headers, and total headers in responses are often limited to 8 KB (8,192 bytes). Ensure that the headers containing errors do not exceed 8 KB.

Creating rich errors on the server

Rich errors are created from `Google.Rpc.Status`. This type is ***different*** from `Grpc.Core.Status`.

`Google.Rpc.Status` has status, message, and details fields. The most important field is details, which is a repeating field of Any values. Details are where complex payloads are

added.

Although any message type can be used as a payload, it's recommended to use one of the standard error payloads:

- `BadRequest`
- `PreconditionFailure`
- `ErrorInfo`
- `ResourceInfo`
- `QuotaFailure`

Grpc.StatusProto includes the `ToRpcException` a helper method to convert `Google.Rpc.Status` to an error. Throw the error from the gRPC server method:

cs

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        ArgumentNotNullOrEmpty(request.Name);

        return Task.FromResult(new HelloReply { Message = "Hello " +
request.Name });
    }

    public static void ArgumentNotNullOrEmpty(string value,
[CallerArgumentExpression(nameof(value))] string? paramName = null)
    {
        if (string.IsNullOrEmpty(value))
        {
            var status = new Google.Rpc.Status
            {
                Code = (int)Code.InvalidArgument,
                Message = "Bad request",
                Details =
                {
                    Any.Pack(new BadRequest
                    {
                        FieldViolations =
                        {
                            new BadRequest.Types.FieldViolation { Field =
paramName, Description = "Value is null or empty" }
                        }
                    })
                }
            };
            throw status.ToRpcException();
        }
```

```
    }
}
```

Reading rich errors by a client

Rich errors are read from the `RpcException` caught in the client. Catch the exception and use helper methods provided by `Grpc.StatusCode` to get its `Google.Rpc.Status` instance:

cs

```
var client = new Greet.GreeterClient(channel);

try
{
    var reply = await client.SayHelloAsync(new HelloRequest { Name = name
});
    Console.WriteLine("Greeting: " + reply.Message);
}
catch (RpcException ex)
{
    Console.WriteLine($"Server error: {ex.Status.Detail}");
    var badRequest = ex.GetRpcStatus()?.GetDetail<BadRequest>();
    if (badRequest != null)
    {
        foreach (var fieldViolation in badRequest.FieldViolations)
        {
            Console.WriteLine($"Field: {fieldViolation.Field}");
            Console.WriteLine($"Description: {fieldViolation.Description}");
        }
    }
}
```

The preceding code:

- Makes a gRPC call inside a try/catch that catches `RpcException`.
- Calls `GetRpcStatus()` to attempt to get the rich error model from the exception.
- Calls `GetDetail<BadRequest>()` to attempt to get a `BadRequest` payload from the rich error.

Additional resources

- Create gRPC services and methods
- Call gRPC services with the .NET client
- gRPC status codes

gRPC interceptors on .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.

By Ernest Nguyen

Interceptors are a gRPC concept that allows apps to interact with incoming or outgoing gRPC calls. They offer a way to enrich the request processing pipeline.

Interceptors are configured for a channel or service and executed automatically for each gRPC call. Since interceptors are transparent to the user's application logic, they're an excellent solution for common cases, such as logging, monitoring, authentication, and validation.

`Interceptor` type

Interceptors can be implemented for both gRPC servers and clients by creating a class that inherits from the `Interceptor` type:

C#

```
public class ExampleInterceptor : Interceptor
{
}
```

By default, the `Interceptor` base class doesn't do anything. Add behavior to an interceptor by overriding the appropriate base class methods in an interceptor implementation.

Client interceptors

gRPC client interceptors intercept outgoing RPC invocations. They provide access to the sent request, the incoming response, and the context for a client-side call.

`Interceptor` methods to override for client:

- `BlockingUnaryCall`: Intercepts a blocking invocation of an unary RPC.
- `AsyncUnaryCall`: Intercepts an asynchronous invocation of an unary RPC.
- `AsyncClientStreamingCall`: Intercepts an asynchronous invocation of a client-streaming RPC.
- `AsyncServerStreamingCall`: Intercepts an asynchronous invocation of a server-streaming RPC.
- `AsyncDuplexStreamingCall`: Intercepts an asynchronous invocation of a bidirectional-streaming RPC.

> ⚠ **Warning**
>
> Although both `BlockingUnaryCall` and `AsyncUnaryCall` refer to unary RPCs, they aren't interchangeable. A blocking invocation isn't intercepted by `AsyncUnaryCall`, and an asynchronous invocation isn't intercepted by a `BlockingUnaryCall`.

Create a client gRPC interceptor

The following code presents a basic example of intercepting an asynchronous invocation of a unary call:

C#

```
public class ClientLoggingInterceptor : Interceptor
{
    private readonly ILogger _logger;

    public ClientLoggingInterceptor(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<ClientLoggingInterceptor>();
    }

    public override AsyncUnaryCall<TResponse> AsyncUnaryCall<TRequest,
TResponse>(
        TRequest request,
        ClientInterceptorContext<TRequest, TResponse> context,
        AsyncUnaryCallContinuation<TRequest, TResponse> continuation)
    {
        _logger.LogInformation("Starting call. Type/Method: {Type} /
{Method}",
            context.Method.Type, context.Method.Name);
        return continuation(request, context);
    }
}
```

Overriding `AsyncUnaryCall`:

- Intercepts an asynchronous unary call.
- Logs details about the call.
- Calls the `continuation` parameter passed into the method. This invokes the next interceptor in the chain or the underlying call invoker if this is the last interceptor.

Methods on `Interceptor` for each kind of service method have different signatures. However, the concept behind `continuation` and `context` parameters remains the same:

- `continuation` is a delegate which invokes the next interceptor in the chain or the underlying call invoker (if there is no interceptor left in the chain). It isn't an error to call it zero or multiple times. Interceptors aren't required to return a call representation (`AsyncUnaryCall` in case of unary RPC) returned from the `continuation` delegate. Omitting the delegate call and returning your own instance of call representation breaks the interceptors' chain and returns the associated response immediately.
- `context` carries scoped values associated with the client-side call. Use `context` to pass metadata, such as security principals, credentials, or tracing data. Moreover, `context` carries information about deadlines and cancellation. For more information, see Reliable gRPC services with deadlines and cancellation.

Awaiting response in client interceptor

An interceptor can await the response in unary and client streaming calls by updating the `AsyncUnaryCall<TResponse>.ResponseAsync` or `AsyncClientStreamingCall<TRequest, TResponse>.ResponseAsync` value.

C#

```
public class ErrorHandlerInterceptor : Interceptor
{
    public override AsyncUnaryCall<TResponse> AsyncUnaryCall<TRequest,
TResponse>(
        TRequest request,
        ClientInterceptorContext<TRequest, TResponse> context,
        AsyncUnaryCallContinuation<TRequest, TResponse> continuation)
    {
        var call = continuation(request, context);

        return new AsyncUnaryCall<TResponse>(
            HandleResponse(call.ResponseAsync),
            call.ResponseHeadersAsync,
            call.GetStatus,
            call.GetTrailers,
```

```
            call.Dispose);
    }

    private async Task<TResponse> HandleResponse<TResponse>(Task<TResponse>
inner)
    {
        try
        {
            return await inner;
        }
        catch (Exception ex)
        {
            throw new InvalidOperationException("Custom error", ex);
        }
    }
}
```

The preceding code:

- Creates a new interceptor that overrides `AsyncUnaryCall`.
- Overriding `AsyncUnaryCall`:
  - Calls the `continuation` parameter to invoke the next item in the interceptor chain.
  - Creates a new `AsyncUnaryCall<TResponse>` instance based on the result of the continuation.
  - Wraps the `ResponseAsync` task using the `HandleResponse` method.
  - Awaits the response with `HandleResponse`. Awaiting the response allows logic to be added after the client received the response. By awaiting the response in a try-catch block, errors from calls can be logged.

For more information on how to create a client interceptor, see the ClientLoggerInterceptor.cs example in the grpc/grpc-dotnet GitHub repository.

Configure client interceptors

gRPC client interceptors are configured on a channel.

The following code:

- Creates a channel by using `GrpcChannel.ForAddress`.
- Uses the `Intercept` extension method to configure the channel to use the interceptor. Note that this method returns a `CallInvoker`. Strongly-typed gRPC clients can be created from an invoker just like a channel.
- Creates a client from the invoker. gRPC calls made by the client automatically execute the interceptor.

```
C#
```

```
using var channel = GrpcChannel.ForAddress("https://localhost:5001");
var invoker = channel.Intercept(new ClientLoggerInterceptor());

var client = new Greeter.GreeterClient(invoker);
```

The `Intercept` extension method can be chained to configure multiple interceptors for a channel. Alternatively, there is an `Intercept` overload that accepts multiple interceptors. Any number of interceptors can be executed for a single gRPC call, as the following example demonstrates:

```
C#
```

```
var invoker = channel
    .Intercept(new ClientTokenInterceptor())
    .Intercept(new ClientMonitoringInterceptor())
    .Intercept(new ClientLoggerInterceptor());
```

Interceptors are invoked in reverse order of the chained `Intercept` extension methods. In the preceding code, interceptors are invoked in the following order:

1. `ClientLoggerInterceptor`
2. `ClientMonitoringInterceptor`
3. `ClientTokenInterceptor`

For information on how to configure interceptors with gRPC client factory, see gRPC client factory integration in .NET.

Server interceptors

gRPC server interceptors intercept incoming RPC requests. They provide access to the incoming request, the outgoing response, and the context for a server-side call.

`Interceptor` methods to override for server:

- `UnaryServerHandler`: Intercepts a unary RPC.
- `ClientStreamingServerHandler`: Intercepts a client-streaming RPC.
- `ServerStreamingServerHandler`: Intercepts a server-streaming RPC.
- `DuplexStreamingServerHandler`: Intercepts a bidirectional-streaming RPC.

Create a server gRPC interceptor

The following code presents an example of an intercepting an incoming unary RPC:

C#

```
public class ServerLoggerInterceptor : Interceptor
{
    private readonly ILogger _logger;

    public ServerLoggerInterceptor(ILogger<ServerLoggerInterceptor> logger)
    {
        _logger = logger;
    }

    public override async Task<TResponse> UnaryServerHandler<TRequest,
TResponse>(
        TRequest request,
        ServerCallContext context,
        UnaryServerMethod<TRequest, TResponse> continuation)
    {
        _logger.LogInformation("Starting receiving call. Type/Method: {Type}
/ {Method}",
            MethodType.Unary, context.Method);
        try
        {
            return await continuation(request, context);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, $"Error thrown by {context.Method}.");
            throw;
        }
    }
}
```

Overriding `UnaryServerHandler`:

- Intercepts an incoming unary call.
- Logs details about the call.
- Calls the `continuation` parameter passed into the method. This invokes the next interceptor in the chain or the service handler if this is the last interceptor.
- Logs any exceptions. Awaiting the continuation allows logic to be added after the service method has executed. By awaiting the continuation in a try-catch block, errors from methods can be logged.

The signature of both client and server interceptors methods are similar:

- `continuation` stands for a delegate for an incoming RPC calling the next interceptor in the chain or the service handler (if there is no interceptor left in the chain). Similar to client interceptors, you can call it any time and there's no need to

return a response directly from the continuation delegate. Outbound logic can be added after a service handler has executed by awaiting the continuation.
- `context` carries metadata associated with the server-side call, such as request metadata, deadlines and cancellation, or RPC result.

For more information on how to create a server interceptor, see the ServerLoggerInterceptor.cs example in the grpc/grpc-dotnet GitHub repository.

Configure server interceptors

gRPC server interceptors are configured at startup. The following code:

- Adds gRPC to the app with `AddGrpc`.
- Configures `ServerLoggerInterceptor` for all services by adding it to the service option's `Interceptors` collection.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.Interceptors.Add<ServerLoggerInterceptor>();
    });
}
```

An interceptor can also be configured for a specific service by using `AddServiceOptions` and specifying the service type.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services
        .AddGrpc()
        .AddServiceOptions<GreeterService>(options =>
        {
            options.Interceptors.Add<ServerLoggerInterceptor>();
        });
}
```

Interceptors are run in the order that they're added to the `InterceptorCollection`. If both global and single service interceptors are configured, then globally-configured interceptors are run before those configured for a single service.

By default, gRPC server interceptors have a per-request lifetime. Overriding this behavior is possible through registering the interceptor type with dependency injection. The following example registers the `ServerLoggerInterceptor` with a singleton lifetime:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.Interceptors.Add<ServerLoggerInterceptor>();
    });

    services.AddSingleton<ServerLoggerInterceptor>();
}
```

gRPC Interceptors versus Middleware

ASP.NET Core middleware offers similar functionalities compared to interceptors in C-core-based gRPC apps. ASP.NET Core middleware and interceptors are conceptually similar. Both:

- Are used to construct a pipeline that handles a gRPC request.
- Allow work to be performed before or after the next component in the pipeline.
- Provide access to `HttpContext`:
  - In middleware, the `HttpContext` is a parameter.
  - In interceptors, the `HttpContext` can be accessed using the `ServerCallContext` parameter with the `ServerCallContext.GetHttpContext` extension method. This feature is specific to interceptors running in ASP.NET Core.

gRPC Interceptor differences from ASP.NET Core Middleware:

- Interceptors:
  - Operate on the gRPC layer of abstraction using the ServerCallContext.
  - Provide access to:
    - The deserialized message sent to a call.
    - The message returned from the call before it's serialized.
  - Can catch and handle exceptions thrown from gRPC services.
- Middleware:
  - Runs for all HTTP requests.
  - Runs before gRPC interceptors.
  - Operates on the underlying HTTP/2 messages.
  - Can only access bytes from the request and response streams.

Additional resources

- Overview for gRPC on .NET
- Create gRPC services and methods
- Call gRPC services with the .NET client
- Example of how to use gRPC on the client and server (grpc/grpc-dotnet GitHub repository)
- Configure interceptors in a gRPC client factory in .NET

Logging and diagnostics in gRPC on .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**.

By James Newton-King

This article provides guidance for gathering diagnostics from a gRPC app 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.
- **Tracing** - Events related to an operation written using `DiaganosticSource` and `Activity`. Traces from diagnostic source are commonly used to collect app telemetry by libraries such as Application Insights and OpenTelemetry.
- **Metrics** - Representation of data measures over intervals of time, for example, requests per second. Metrics are emitted using `EventCounter` and can be observed using dotnet-counters command-line tool or with Application Insights.

Logging

gRPC services and the gRPC client write logs using .NET Core logging. Logs are a good place to start when debugging unexpected behavior in service and client apps.

gRPC services logging

> **⚠ Warning**
>
> Server-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

Since gRPC services are hosted on ASP.NET Core, it uses the ASP.NET Core logging system. In the default configuration, gRPC logs minimal information, but logging can be configured. See the documentation on ASP.NET Core logging for details on configuring ASP.NET Core logging.

gRPC adds logs under the `Grpc` category. To enable detailed logs from gRPC, configure the `Grpc` prefixes to the `Debug` level in the `appsettings.json` file by adding the following items to the `LogLevel` subsection in `Logging`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information",
      "Grpc": "Debug"
    }
  }
}
```

Logging can also be configured in `Program.cs` with `ConfigureLogging`:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.AddFilter("Grpc", LogLevel.Debug);
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

When not using JSON-based configuration, set the following configuration value in the configuration system:

- `Logging:LogLevel:Grpc` = `Debug`

Check the documentation for your configuration system to determine how to specify nested configuration values. For example, when using environment variables, two `_` characters are used instead of the `:` (for example, `Logging__LogLevel__Grpc`).

We recommend using the `Debug` level when gathering detailed diagnostics for an app. The `Trace` level produces low-level diagnostics and is rarely needed to diagnose issues.

Sample logging output

Here is an example of console output at the `Debug` level of a gRPC service:

Console

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST
https://localhost:5001/Greet.Greeter/SayHello application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /Greet.Greeter/SayHello'
dbug: Grpc.AspNetCore.Server.ServerCallHandler[1]
      Reading message.
info: GrpcService.GreeterService[0]
      Hello World
dbug: Grpc.AspNetCore.Server.ServerCallHandler[6]
      Sending message.
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 1.4113ms 200 application/grpc
```

Access server-side logs

How server-side logs are accessed depends on the app's environment.

As a console app

If you're running in a console app, the Console logger should be enabled by default. gRPC logs will appear in the console.

Other environments

If the app is deployed to another environment (for example, Docker, Kubernetes, or Windows Service), see Logging in .NET Core and ASP.NET Core for more information on how to configure logging providers suitable for the environment.

gRPC client logging

> ⚠ **Warning**

Client-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

To get logs from the .NET client, set the `GrpcChannelOptions.LoggerFactory` property when the client's channel is created. When calling a gRPC service from an ASP.NET Core app, the logger factory can be resolved from dependency injection (DI):

C#

```
[ApiController]
[Route("[controller]")]
public class GreetingController : ControllerBase
{
    private ILoggerFactory _loggerFactory;

    public GreetingController(ILoggerFactory loggerFactory)
    {
        _loggerFactory = loggerFactory;
    }

    [HttpGet]
    public async Task<ActionResult<string>> Get(string name)
    {
        var channel = GrpcChannel.ForAddress("https://localhost:5001",
            new GrpcChannelOptions { LoggerFactory = _loggerFactory });
        var client = new Greeter.GreeterClient(channel);

        var reply = await client.SayHelloAsync(new HelloRequest { Name =
name });
        return Ok(reply.Message);
    }
}
```

An alternative way to enable client logging is to use the gRPC client factory to create the client. A gRPC client registered with the client factory and resolved from DI will automatically use the app's configured logging.

If the app isn't using DI, then create a new `ILoggerFactory` instance with LoggerFactory.Create. To access this method, add the Microsoft.Extensions.Logging package to your app.

C#

```
var loggerFactory = LoggerFactory.Create(logging =>
{
    logging.AddConsole();
    logging.SetMinimumLevel(LogLevel.Debug);
});
```

```
var channel = GrpcChannel.ForAddress("https://localhost:5001",
    new GrpcChannelOptions { LoggerFactory = loggerFactory });

var client = Greeter.GreeterClient(channel);
```

gRPC client log scopes

The gRPC client adds a logging scope to logs made during a gRPC call. The scope has metadata related to the gRPC call:

- **GrpcMethodType** - The gRPC method type. Possible values are names from `Grpc.Core.MethodType` enum. For example, `Unary`.
- **GrpcUri** - The relative URI of the gRPC method. For example, */greet.Greeter/SayHellos*.

Sample logging output

Here is an example of console output at the `Debug` level of a gRPC client:

Console

```
dbug: Grpc.Net.Client.Internal.GrpcCall[1]
      Starting gRPC call. Method type: 'Unary', URI:
'https://localhost:5001/Greet.Greeter/SayHello'.
dbug: Grpc.Net.Client.Internal.GrpcCall[6]
      Sending message.
dbug: Grpc.Net.Client.Internal.GrpcCall[1]
      Reading message.
dbug: Grpc.Net.Client.Internal.GrpcCall[4]
      Finished gRPC call.
```

Tracing

gRPC services and the gRPC client provide information about gRPC calls using DiagnosticSource and Activity.

- .NET gRPC uses an activity to represent a gRPC call.
- Tracing events are written to the diagnostic source at the start and stop of the gRPC call activity.
- Tracing doesn't capture information about when messages are sent over the lifetime of gRPC streaming calls.

gRPC service tracing

gRPC services are hosted on ASP.NET Core, which reports events about incoming HTTP requests. gRPC specific metadata is added to the existing HTTP request diagnostics that ASP.NET Core provides.

- Diagnostic source name is `Microsoft.AspNetCore`.
- Activity name is `Microsoft.AspNetCore.Hosting.HttpRequestIn`.
  - Name of the gRPC method invoked by the gRPC call is added as a tag with the name `grpc.method`.
  - Status code of the gRPC call when it is complete is added as a tag with the name `grpc.status_code`.

gRPC client tracing

The .NET gRPC client uses `HttpClient` to make gRPC calls. Although `HttpClient` writes diagnostic events, the .NET gRPC client provides a custom diagnostic source, activity, and events so that complete information about a gRPC call can be collected.

- Diagnostic source name is `Grpc.Net.Client`.
- Activity name is `Grpc.Net.Client.GrpcOut`.
  - Name of the gRPC method invoked by the gRPC call is added as a tag with the name `grpc.method`.
  - Status code of the gRPC call when it is complete is added as a tag with the name `grpc.status_code`.

Collecting tracing

The easiest way to use `DiagnosticSource` is to configure a telemetry library such as Application Insights or OpenTelemetry in your app. The library will process information about gRPC calls along-side other app telemetry.

Tracing can be viewed in a managed service like Application Insights, or run as your own distributed tracing system. OpenTelemetry supports exporting tracing data to Jaeger and Zipkin.

`DiagnosticSource` can consume tracing events in code using `DiagnosticListener`. For information about listening to a diagnostic source with code, see the DiagnosticSource user's guide.

> **Note**

Telemetry libraries do not capture gRPC specific `Grpc.Net.Client.GrpcOut` telemetry currently. Work to improve telemetry libraries capturing this tracing is ongoing.

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. .NET gRPC metrics are emitted using `EventCounter`.

gRPC service metrics

gRPC server metrics are reported on `Grpc.AspNetCore.Server` event source.

Expand table

Name	Description
`total-calls`	Total Calls
`current-calls`	Current Calls
`calls-failed`	Total Calls Failed
`calls-deadline-exceeded`	Total Calls Deadline Exceeded
`messages-sent`	Total Messages Sent
`messages-received`	Total Messages Received
`calls-unimplemented`	Total Calls Unimplemented

ASP.NET Core also provides its own metrics on `Microsoft.AspNetCore.Hosting` event source.

gRPC client metrics

gRPC client metrics are reported on `Grpc.Net.Client` event source.

Expand table

Name	Description
`total-calls`	Total Calls

Name	Description
`current-calls`	Current Calls
`calls-failed`	Total Calls Failed
`calls-deadline-exceeded`	Total Calls Deadline Exceeded
`messages-sent`	Total Messages Sent
`messages-received`	Total Messages Received

Observe metrics

dotnet-counters is a performance monitoring tool for ad-hoc health monitoring and first-level performance investigation. Monitor a .NET app with either `Grpc.AspNetCore.Server` or `Grpc.Net.Client` as the provider name.

Console

```
> dotnet-counters monitor --process-id 1902 Grpc.AspNetCore.Server

Press p to pause, r to resume, q to quit.
    Status: Running
[Grpc.AspNetCore.Server]
    Total Calls                                300
    Current Calls                              5
    Total Calls Failed                         0
    Total Calls Deadline Exceeded              0
    Total Messages Sent                        295
    Total Messages Received                    300
    Total Calls Unimplemented                  0
```

Another way to observe gRPC metrics is to capture counter data using Application Insights's Microsoft.ApplicationInsights.EventCounterCollector package. Once setup, Application Insights collects common .NET counters at runtime. gRPC's counters are not collected by default, but App Insights can be customized to include additional counters.

Specify the gRPC counters for Application Insight to collect in `Startup.cs`:

C#

```
    using Microsoft.ApplicationInsights.Extensibility.EventCounterCollector;

    public void ConfigureServices(IServiceCollection services)
    {
        //... other code...
```

```
        services.ConfigureTelemetryModule<EventCounterCollectionModule>(
            (module, o) =>
            {
                // Configure App Insights to collect gRPC counters gRPC
services hosted in an ASP.NET Core app
                module.Counters.Add(new
EventCounterCollectionRequest("Grpc.AspNetCore.Server", "current-calls"));
                module.Counters.Add(new
EventCounterCollectionRequest("Grpc.AspNetCore.Server", "total-calls"));
                module.Counters.Add(new
EventCounterCollectionRequest("Grpc.AspNetCore.Server", "calls-failed"));
            }
        );
    }
```

Additional resources

- Logging in .NET Core and ASP.NET Core
- gRPC for .NET configuration
- gRPC client factory integration in .NET

Security considerations in gRPC 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 James Newton-King

This article provides information on securing gRPC with .NET Core.

Transport security

gRPC messages are sent and received using HTTP/2. We recommend:

- Transport Layer Security (TLS) be used to secure messages in production gRPC apps.
- gRPC services should only listen and respond over secured ports.

TLS is configured in Kestrel. For more information on configuring Kestrel endpoints, see Kestrel endpoint configuration.

A TLS termination proxy can be combined with TLS. The benefits of using TLS termination should be considered against the security risks of sending unsecured HTTP requests between apps in the private network.

Exceptions

Exception messages are generally considered sensitive data that shouldn't be revealed to a client. By default, gRPC doesn't send the details of an exception thrown by a gRPC service to the client. Instead, the client receives a generic message indicating an error occurred. Exception message delivery to the client can be overridden (for example, in development or test) with EnableDetailedErrors. Exception messages shouldn't be exposed to the client in production apps.

Message size limits

Incoming messages to gRPC clients and services are loaded into memory. Message size limits are a mechanism to help prevent gRPC from consuming excessive resources.

gRPC uses per-message size limits to manage incoming and outgoing messages. By default, gRPC limits incoming messages to 4 MB. There is no limit on outgoing messages.

On the server, gRPC message limits can be configured for all services in an app with `AddGrpc`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.MaxReceiveMessageSize = 1 * 1024 * 1024; // 1 MB
        options.MaxSendMessageSize = 1 * 1024 * 1024; // 1 MB
    });
}
```

Limits can also be configured for an individual service using `AddServiceOptions<TService>`. For more information on configuring message size limits, see gRPC configuration.

Client certificate validation

Client certificates are initially validated when the connection is established. By default, Kestrel doesn't perform additional validation of a connection's client certificate.

We recommend that gRPC services secured by client certificates use the Microsoft.AspNetCore.Authentication.Certificate package. ASP.NET Core certification authentication will perform additional validation on a client certificate, including:

- Certificate has a valid extended key use (EKU)
- Is within its validity period
- Check certificate revocation

Performance best practices with gRPC

Article • 10/04/2024

> **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

gRPC is designed for high-performance services. This document explains how to get the best performance possible from gRPC.

Reuse gRPC channels

A gRPC channel should be reused when making gRPC calls. Reusing a channel allows calls to be multiplexed through an existing HTTP/2 connection.

If a new channel is created for each gRPC call then the amount of time it takes to complete can increase significantly. Each call will require multiple network round-trips between the client and the server to create a new HTTP/2 connection:

1. Opening a socket
2. Establishing TCP connection
3. Negotiating TLS
4. Starting HTTP/2 connection
5. Making the gRPC call

Channels are safe to share and reuse between gRPC calls:

- gRPC clients are created with channels. gRPC clients are lightweight objects and don't need to be cached or reused.
- Multiple gRPC clients can be created from a channel, including different types of clients.
- A channel and clients created from the channel can safely be used by multiple threads.
- Clients created from the channel can make multiple simultaneous calls.

gRPC client factory offers a centralized way to configure channels. It automatically reuses underlying channels. For more information, see gRPC client factory integration in .NET.

Connection concurrency

HTTP/2 connections typically have a limit on the number of maximum concurrent streams (active HTTP requests) on a connection at one time. By default, most servers set this limit to 100 concurrent streams.

A gRPC channel uses a single HTTP/2 connection, and concurrent calls are multiplexed on that connection. When the number of active calls reaches the connection stream limit, additional calls are queued in the client. Queued calls wait for active calls to complete before they are sent. Applications with high load, or long running streaming gRPC calls, could see performance issues caused by calls queuing because of this limit.

.NET 5 introduces the SocketsHttpHandler.EnableMultipleHttp2Connections property. When set to `true`, additional HTTP/2 connections are created by a channel when the concurrent stream limit is reached. When a `GrpcChannel` is created its internal `SocketsHttpHandler` is automatically configured to create additional HTTP/2 connections. If an app configures its own handler, consider setting `EnableMultipleHttp2Connections` to `true`:

C#

```
var channel = GrpcChannel.ForAddress("https://localhost", new
GrpcChannelOptions
{
    HttpHandler = new SocketsHttpHandler
    {
        EnableMultipleHttp2Connections = true,

        // ...configure other handler settings
    }
});
```

.NET Framework apps that make gRPC calls must be configured to use WinHttpHandler. .NET Framework apps can set the WinHttpHandler.EnableMultipleHttp2Connections property to `true` to create additional connections.

There are a couple of workarounds for .NET Core 3.1 apps:

- Create separate gRPC channels for areas of the app with high load. For example, the `Logger` gRPC service might have a high load. Use a separate channel to create

the `LoggerClient` in the app.

- Use a pool of gRPC channels, for example, create a list of gRPC channels. `Random` is used to pick a channel from the list each time a gRPC channel is needed. Using `Random` randomly distributes calls over multiple connections.

> **Important**
>
> Increasing the maximum concurrent stream limit on the server is another way to solve this problem. In Kestrel this is configured with MaxStreamsPerConnection.
>
> Increasing the maximum concurrent stream limit is not recommended. Too many streams on a single HTTP/2 connection introduces new performance issues:
>
> - Thread contention between streams trying to write to the connection.
> - Connection packet loss causes all calls to be blocked at the TCP layer.

`ServerGarbageCollection` in client apps

The .NET garbage collector has two modes: workstation garbage collection (GC) and server garbage collection. Each is each tuned for different workloads. ASP.NET Core apps use server GC by default.

Highly concurrent apps generally perform better with server GC. If a gRPC client app is sending and receiving a high number of gRPC calls at the same time, then there may be a performance benefit in updating the app to use server GC.

To enable server GC, set `<ServerGarbageCollection>` in the app's project file:

XML

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

For more information about garbage collection, see Workstation and server garbage collection.

> **Note**
>
> ASP.NET Core apps use server GC by default. Enabling `<ServerGarbageCollection>` is only useful in non-server gRPC client apps, for example in a gRPC client console

app.

Load balancing

Some load balancers don't work effectively with gRPC. L4 (transport) load balancers operate at a connection level, by distributing TCP connections across endpoints. This approach works well for loading balancing API calls made with HTTP/1.1. Concurrent calls made with HTTP/1.1 are sent on different connections, allowing calls to be load balanced across endpoints.

Because L4 load balancers operate at a connection level, they don't work well with gRPC. gRPC uses HTTP/2, which multiplexes multiple calls on a single TCP connection. All gRPC calls over that connection go to one endpoint.

There are two options to effectively load balance gRPC:

- Client-side load balancing
- L7 (application) proxy load balancing

> ⓘ **Note**
>
> Only gRPC calls can be load balanced between endpoints. Once a streaming gRPC call is established, all messages sent over the stream go to one endpoint.

Client-side load balancing

With client-side load balancing, the client knows about endpoints. For each gRPC call, it selects a different endpoint to send the call to. Client-side load balancing is a good choice when latency is important. There's no proxy between the client and the service, so the call is sent to the service directly. The downside to client-side load balancing is that each client must keep track of the available endpoints that it should use.

Lookaside client load balancing is a technique where load balancing state is stored in a central location. Clients periodically query the central location for information to use when making load balancing decisions.

For more information, see gRPC client-side load balancing.

Proxy load balancing

An L7 (application) proxy works at a higher level than an L4 (transport) proxy. L7 proxies understand HTTP/2. The proxy receives gRPC calls multiplexed on one HTTP/2 connection and distributes them across multiple backend endpoints. Using a proxy is simpler than client-side load balancing, but adds extra latency to gRPC calls.

There are many L7 proxies available. Some options are:

- Envoy - A popular open source proxy.
- Linkerd - Service mesh for Kubernetes.
- YARP: Yet Another Reverse Proxy - An open source proxy written in .NET.

Inter-process communication

gRPC calls between a client and service are usually sent over TCP sockets. TCP is great for communicating across a network, but inter-process communication (IPC) is more efficient when the client and service are on the same machine.

Consider using a transport like Unix domain sockets or named pipes for gRPC calls between processes on the same machine. For more information, see Inter-process communication with gRPC.

Keep alive pings

Keep alive pings can be used to keep HTTP/2 connections alive during periods of inactivity. Having an existing HTTP/2 connection ready when an app resumes activity allows for the initial gRPC calls to be made quickly, without a delay caused by the connection being reestablished.

Keep alive pings are configured on SocketsHttpHandler:

C#

```
var handler = new SocketsHttpHandler
{
    PooledConnectionIdleTimeout = Timeout.InfiniteTimeSpan,
    KeepAlivePingDelay = TimeSpan.FromSeconds(60),
    KeepAlivePingTimeout = TimeSpan.FromSeconds(30),
    EnableMultipleHttp2Connections = true
};

var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
{
```

```
    HttpHandler = handler
});
```

The preceding code configures a channel that sends a keep alive ping to the server every 60 seconds during periods of inactivity. The ping ensures the server and any proxies in use won't close the connection because of inactivity.

> ⓘ **Note**
>
> Keep alive pings only help keep the connection alive. Long-running gRPC calls on the connection may still be terminated by the server or intermediary proxies for inactivity.

Flow control

HTTP/2 flow control is a feature that prevents apps from being overwhelmed with data. When using flow control:

- Each HTTP/2 connection and request has an available buffer window. The buffer window is how much data the app can receive at once.
- Flow control activates if the buffer window is filled up. When activated, the sending app pauses sending more data.
- Once the receiving app has processed data, then space in the buffer window is available. The sending app resumes sending data.

Flow control can have a negative impact on performance when receiving large messages. If the buffer window is smaller than incoming message payloads or there's latency between the client and server, then data can be sent in start/stop bursts.

Flow control performance issues can be fixed by increasing buffer window size. In Kestrel, this is configured with InitialConnectionWindowSize and InitialStreamWindowSize at app startup:

C#

```
builder.WebHost.ConfigureKestrel(options =>
{
    var http2 = options.Limits.Http2;
    http2.InitialConnectionWindowSize = 1024 * 1024 * 2; // 2 MB
    http2.InitialStreamWindowSize = 1024 * 1024; // 1 MB
});
```

Recommendations:

- If a gRPC service often receives messages larger than 768 KB, Kestrel's default stream window size, then consider increasing the connection and stream window size.
- The connection window size should always be equal to or greater than the stream window size. A stream is part of the connection, and the sender is limited by both.

For more information about how flow control works, see HTTP/2 Flow Control (blog post).

> **Important**
>
> Increasing Kestrel's window size allows Kestrel to buffer more data on behalf of the app, which possibly increases memory usage. Avoid configuring an unnecessarily large window size.

Gracefully complete streaming calls

Try to complete streaming calls gracefully. Gracefully completing calls avoids unnecessary errors and allow servers to reuse internal data structures between requests.

A call is completed gracefully when the client and server have finished sending messages and the peer has read all the messages.

Client request stream:

1. The client has finished writing messages to the request stream and completes the stream with `call.RequestStream.CompleteAsync()`.
2. The server has read all messages from the request stream. Depending on how you're reading messages, either `requestStream.MoveNext()` returns `false` or `requestStream.ReadAllAsync()` has finished.

Server response stream:

1. The server has finished writing messages to the response stream and the server method has exited.
2. The client has read all messages from the response stream. Depending on how you're reading messages, either `call.ResponseStream.MoveNext()` returns `false` or `call.ResponseStream.ReadAllAsync()` has finished.

For an example of gracefully completing a bi-direction streaming call, see making a bi-directional streaming call.

Server streaming calls don't have a request stream. This means that the only way a client can communicate to the server that the stream should stop is by canceling it. If the overhead from canceled calls is impacting the app then consider changing the server streaming call to a bi-directional streaming call. In a bi-directional streaming call the client completing the request stream can be a signal to the server to end the call.

Dispose streaming calls

Always dispose streaming calls once they're no longer needed. The type returned when starting streaming calls implements `IDisposable`. Disposing a call once it is no longer needed ensures it is stopped and all resources are cleaned up.

In the following example, the using declaration on the `AccumulateCount()` call ensures it's always disposed if an unexpected error occurs.

C#

```
var client = new Counter.CounterClient(channel);
using var call = client.AccumulateCount();

for (var i = 0; i < 3; i++)
{
    await call.RequestStream.WriteAsync(new CounterRequest { Count = 1 });
}
await call.RequestStream.CompleteAsync();

var response = await call;
Console.WriteLine($"Count: {response.Count}");
// Count: 3
```

Ideally streaming calls should be completed gracefully. Disposing the call ensures the HTTP request between the client and the server is canceled if an unexpected error occurs. Streaming calls that are accidentally left running don't just leak memory and resources on the client, but are left running on the server as well. Many leaked streaming calls could impact the stability of the app.

Disposing of a streaming call that has already completed gracefully doesn't have any negative impact.

Replace unary calls with streaming

gRPC bidirectional streaming can be used to replace unary gRPC calls in high-performance scenarios. Once a bidirectional stream has started, streaming messages back and forth is faster than sending messages with multiple unary gRPC calls. Streamed

messages are sent as data on an existing HTTP/2 request and eliminates the overhead of creating a new HTTP/2 request for each unary call.

Example service:

C#

```
public override async Task SayHello(IAsyncStreamReader<HelloRequest>
requestStream,
    IServerStreamWriter<HelloReply> responseStream, ServerCallContext
context)
{
    await foreach (var request in requestStream.ReadAllAsync())
    {
        var helloReply = new HelloReply { Message = "Hello " + request.Name
};

        await responseStream.WriteAsync(helloReply);
    }
}
```

Example client:

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHello();

Console.WriteLine("Type a name then press enter.");
while (true)
{
    var text = Console.ReadLine();

    // Send and receive messages over the stream
    await call.RequestStream.WriteAsync(new HelloRequest { Name = text });
    await call.ResponseStream.MoveNext();

    Console.WriteLine($"Greeting: {call.ResponseStream.Current.Message}");
}
```

Replacing unary calls with bidirectional streaming for performance reasons is an advanced technique and is not appropriate in many situations.

Using streaming calls is a good choice when:

1. High throughput or low latency is required.
2. gRPC and HTTP/2 are identified as a performance bottleneck.
3. A worker in the client is sending or receiving regular messages with a gRPC service.

Be aware of the additional complexity and limitations of using streaming calls instead of unary:

1. A stream can be interrupted by a service or connection error. Logic is required to restart stream if there is an error.
2. `RequestStream.WriteAsync` is not safe for multi-threading. Only one message can be written to a stream at a time. Sending messages from multiple threads over a single stream requires a producer/consumer queue like Channel<T> to marshall messages.
3. A gRPC streaming method is limited to receiving one type of message and sending one type of message. For example, `rpc StreamingCall(stream RequestMessage) returns (stream ResponseMessage)` receives `RequestMessage` and sends `ResponseMessage`. Protobuf's support for unknown or conditional messages using `Any` and `oneof` can work around this limitation.

Binary payloads

Binary payloads are supported in Protobuf with the `bytes` scalar value type. A generated property in C# uses `ByteString` as the property type.

ProtoBuf

```
syntax = "proto3";

message PayloadResponse {
    bytes data = 1;
}
```

Protobuf is a binary format that efficiently serializes large binary payloads with minimal overhead. Text based formats like JSON require encoding bytes to base64 and add 33% to the message size.

When working with large `ByteString` payloads there are some best practices to avoid unnecessary copies and allocations that are discussed below.

Send binary payloads

`ByteString` instances are normally created using `ByteString.CopyFrom(byte[] data)`. This method allocates a new `ByteString` and a new `byte[]`. Data is copied into the new byte array.

Additional allocations and copies can be avoided by using `UnsafeByteOperations.UnsafeWrap(ReadOnlyMemory<byte> bytes)` to create `ByteString` instances.

C#

```
var data = await File.ReadAllBytesAsync(path);

var payload = new PayloadResponse();
payload.Data = UnsafeByteOperations.UnsafeWrap(data);
```

Bytes are not copied with `UnsafeByteOperations.UnsafeWrap` so they must not be modified while the `ByteString` is in use.

`UnsafeByteOperations.UnsafeWrap` requires Google.Protobuf version 3.15.0 or later.

Read binary payloads

Data can be efficiently read from `ByteString` instances by using `ByteString.Memory` and `ByteString.Span` properties.

C#

```
var byteString = UnsafeByteOperations.UnsafeWrap(new byte[] { 0, 1, 2 });
var data = byteString.Span;

for (var i = 0; i < data.Length; i++)
{
    Console.WriteLine(data[i]);
}
```

These properties allow code to read data directly from a `ByteString` without allocations or copies.

Most .NET APIs have `ReadOnlyMemory<byte>` and `byte[]` overloads, so `ByteString.Memory` is the recommended way to use the underlying data. However, there are circumstances where an app might need to get the data as a byte array. If a byte array is required then the MemoryMarshal.TryGetArray method can be used to get an array from a `ByteString` without allocating a new copy of the data.

C#

```
var byteString = GetByteString();

ByteArrayContent content;
```

```
if (MemoryMarshal.TryGetArray(byteString.Memory, out var segment))
{
    // Success. Use the ByteString's underlying array.
    content = new ByteArrayContent(segment.Array, segment.Offset,
segment.Count);
}
else
{
    // TryGetArray didn't succeed. Fall back to creating a copy of the data
with ToByteArray.
    content = new ByteArrayContent(byteString.ToByteArray());
}

var httpRequest = new HttpRequestMessage();
httpRequest.Content = content;
```

The preceding code:

- Attempts to get an array from `ByteString.Memory` with MemoryMarshal.TryGetArray.
- Uses the `ArraySegment<byte>` if it was successfully retrieved. The segment has a reference to the array, offset and count.
- Otherwise, falls back to allocating a new array with `ByteString.ToByteArray()`.

gRPC services and large binary payloads

gRPC and Protobuf can send and receive large binary payloads. Although binary Protobuf is more efficient than text-based JSON at serializing binary payloads, there are still important performance characteristics to keep in mind when working with large binary payloads.

gRPC is a message-based RPC framework, which means:

- The entire message is loaded into memory before gRPC can send it.
- When the message is received, the entire message is deserialized into memory.

Binary payloads are allocated as a byte array. For example, a 10 MB binary payload allocates a 10 MB byte array. Messages with large binary payloads can allocate byte arrays on the large object heap. Large allocations impact server performance and scalability.

Advice for creating high-performance applications with large binary payloads:

- **Avoid** large binary payloads in gRPC messages. A byte array larger than 85,000 bytes is considered a large object. Keeping below that size avoids allocating on the large object heap.

- **Consider** splitting large binary payloads using gRPC streaming. Binary data is chunked and streamed over multiple messages. For more information on how to stream files, see examples in the grpc-dotnet repository:
  - gRPC streaming file download.
  - gRPC streaming file upload.
- **Consider** not using gRPC for large binary data. In ASP.NET Core, Web APIs can be used alongside gRPC services. An HTTP endpoint can access the request and response stream body directly:
  - Read the request body using minimal web API
  - Return stream response using minimal web API

gRPC and Native AOT

Article • 07/31/2024

By James Newton-King

gRPC supports .NET native ahead-of-time (AOT) in .NET 8. Native AOT enables publishing gRPC client and server apps as small, fast native executables.

> ⚠ **Warning**
>
> In .NET 8, not all ASP.NET Core features are compatible with Native AOT. For more information, see ASP.NET Core and Native AOT compatibility.

Get started

AOT compilation happens when the app is published. Native AOT is enabled with the `PublishAot` option.

1. Add `<PublishAot>true</PublishAot>` to the gRPC client or server app's project file. This will enable Native AOT compilation during publish and enable dynamic code usage analysis during build and editing.

   XML

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

     <PropertyGroup>
       <TargetFramework>net8.0</TargetFramework>
       <PublishAot>true</PublishAot>
     </PropertyGroup>

     <ItemGroup>
       <PackageReference Include="Grpc.AspNetCore" Version="2.51.0" />
       <PackageReference Include="Google.Protobuf" Version="3.22.0" />
     </ItemGroup>

   </Project>
   ```

   Native AOT can also be enabled by specifying the `-aot` option with the ASP.NET Core gRPC template:

   .NET CLI

```
dotnet new grpc -aot
```

2. Publish the app for a specific runtime identifier (RID) using `dotnet publish -r <RID>`.

The app is available in the publish directory and contains all the code needed to run in it.

Native AOT analysis includes all of the app's code and the libraries the app depends on. Review Native AOT warnings and take corrective steps. It's a good idea to test publishing apps frequently to discover issues early in the development lifecycle.

Optimize publish size

A Native AOT executable contains just the code from external dependencies required to support the app. Unused code is automatically trimmed away.

The publish size of an ASP.NET Core gRPC service can be optimized by creating the host builder with `WebApplication.CreateSlimBuilder()`. This builder provides a minimal list of features required to run an ASP.NET Core app.

C#

```
var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.AddGrpc();

var app = builder.Build();
app.MapGrpcService<GreeterService>();
app.Run();
```

Benefits of using Native AOT

Apps published with Native AOT have:

- Minimized disk footprint
- Reduced startup time
- Reduce memory demand

For more information and examples of the benefits that Native AOT provides, see Benefits of using Native AOT with ASP.NET Core.

Additional resources

- ASP.NET Core support for Native AOT
- Native AOT deployment

Inter-process communication with gRPC

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.

Processes running on the same machine can be designed to communicate with each other. Operating systems provide technologies for enabling fast and efficient inter-process communication (IPC). Popular examples of IPC technologies are Unix domain sockets and Named pipes.

.NET provides support for inter-process communication using gRPC.

Built-in support for Named pipes in ASP.NET Core requires .NET 8 or later.

Get started

IPC calls are sent from a client to a server. To communicate between apps on a machine with gRPC, at least one app must host an ASP.NET Core gRPC server.

An ASP.NET Core gRPC server is usually created from the gRPC template. The project file created by the template uses `Microsoft.NET.SDK.Web` as the SDK:

XML

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

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.47.0" />
    <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
  </ItemGroup>

</Project>
```

The `Microsoft.NET.SDK.Web` SDK value automatically adds a reference to the ASP.NET Core framework. The reference allows the app to use ASP.NET Core types required to host a server.

It's also possible to add a server to existing non-ASP.NET Core projects, such as Windows Services, WPF apps, or WinForms apps. See Host gRPC in non-ASP.NET Core projects for more information.

Inter-process communication (IPC) transports

gRPC calls between a client and server on different machines are usually sent over TCP sockets. TCP is a good choice for communicating across a network or the Internet. However, IPC transports offer advantages when communicating between processes on the same machine:

- Less overhead and faster transfer speeds.
- Integration with OS security features.
- Doesn't use TCP ports, which are a limited resource.

.NET supports multiple IPC transports:

- Unix domain sockets (UDS) is a widely supported IPC technology. UDS is the best choice for building cross-platform apps, and it's usable on Linux, macOS, and Windows 10/Windows Server 2019 or later.
- Named pipes are supported by all versions of Windows. Named pipes integrate well with Windows security, which can control client access to the pipe.
- Additional IPC transports by implementing IConnectionListenerFactory and registering the implementation at app startup.

Depending on the OS, cross-platform apps may choose different IPC transports. An app can check the OS on startup and choose the desired transport for that platform:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    if (OperatingSystem.IsWindows())
    {
        serverOptions.ListenNamedPipe("MyPipeName");
    }
    else
    {
        var socketPath = Path.Combine(Path.GetTempPath(), "socket.tmp");
        serverOptions.ListenUnixSocket(socketPath);
    }

    serverOptions.ConfigureEndpointDefaults(listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http2;
```

```
    });
});
```

Security considerations

IPC apps send and receive RPC calls. External communication is a potential attack vector for IPC apps and must be properly secured.

Secure IPC server app against unexpected callers

The IPC server app hosts RPC services for other apps to call. Incoming callers should be authenticated to prevent untrusted clients from making RPC calls to the server.

Transport security is one option for securing a server. IPC transports, such as Unix domain sockets and named pipes, support limiting access based on operating system permissions:

- Named pipes supports securing a pipe with the Windows access control model. Access rights can be configured in .NET when a server is started using the PipeSecurity class.
- Unix domain sockets support securing a socket with file permissions.

Another option for securing an IPC server is to use authentication and authorization built into ASP.NET Core. For example, the server could be configured to require certificate authentication. RPC calls made by client apps without the required certificate fail with an unauthorized response.

Validate the server in the IPC client app

It's important for the client app to validate the identity of the server it is calling. Validation is necessary to protect against a malicious actor from stopping the trusted server, running their own, and accepting incoming data from clients.

Named pipes provides support for getting the account that a server is running under. A client can validate the server was launched by the expected account:

cs

```
internal static bool CheckPipeConnectionOwnership(
    NamedPipeClientStream pipeStream, SecurityIdentifier expectedOwner)
{
    var remotePipeSecurity = pipeStream.GetAccessControl();
    var remoteOwner =
```

```
remotePipeSecurity.GetOwner(typeof(SecurityIdentifier));
    return expectedOwner.Equals(remoteOwner);
}
```

Another option for validating the server is to secure its endpoints with HTTPS inside ASP.NET Core. The client can configure `SocketsHttpHandler` to validate the server is using the expected certificate when the connection is established.

cs

```
var socketsHttpHandler = new SocketsHttpHandler()
{
    SslOptions = new SslOptions()
    {
        RemoteCertificateValidationCallback = (sender, certificate, chain,
sslPolicyErrors) =>
        {
            if (sslPolicyErrors != SslPolicyErrors.None)
            {
                return false;
            }

            // Validate server cert thumbprint matches the expected
thumbprint.
        }
    }
};
```

Protect against named pipe privilege escalation

Named pipes supports a feature called impersonation. Using impersonation, the named pipes server can execute code with the privileges of the client user. This is a powerful feature but can allow a low-privilege server to impersonate a high-privilege caller and then run malicious code.

Client's can protect against this attack by not allowing impersonation when connecting to a server. Unless required by a server, a TokenImpersonationLevel value of `None` or `Anonymous` should be used when creating a client connection:

cs

```
using var pipeClient = new NamedPipeClientStream(
    serverName: ".", pipeName: "testpipe", PipeDirection.In,
PipeOptions.None, TokenImpersonationLevel.None);
await pipeClient.ConnectAsync();
```

`TokenImpersonationLevel.None` is the default value in `NamedPipeClientStream` constructors that don't have an `impersonationLevel` parameter.

Configure client and server

The client and server must be configured to use an inter-process communication (IPC) transport. For more information about configuring Kestrel and SocketsHttpHandler to use IPC:

- Inter-process communication with gRPC and Unix domain sockets
- Inter-process communication with gRPC and Named pipes

> **Note**
>
> Built-in support for Named pipes in ASP.NET Core requires .NET 8 or later.

Inter-process communication with gRPC and Unix domain sockets

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

.NET supports inter-process communication (IPC) using gRPC. For more information about getting started with using gRPC to communicate between processes, see Inter-process communication with gRPC.

Unix domain sockets (UDS) is a widely supported IPC transport that's more efficient than TCP when the client and server are on the same machine. This article discusses how to configure gRPC communication over UDS.

Prerequisites

- .NET 5 or later
- Linux, macOS, or Windows 10/Windows Server 2019 or later

Server configuration

Unix domain sockets are supported by Kestrel, which is configured in `Program.cs`:

C#

```
var socketPath = Path.Combine(Path.GetTempPath(), "socket.tmp");

var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenUnixSocket(socketPath, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http2;
```

```
    });
});
```

The preceding example:

- Configures Kestrel's endpoints in ConfigureKestrel.
- Calls ListenUnixSocket to listen to a UDS with the specified path.
- Creates a UDS endpoint that isn't configured to use HTTPS. For information about enabling HTTPS, see Kestrel HTTPS endpoint configuration.

Client configuration

`GrpcChannel` supports making gRPC calls over custom transports. When a channel is created, it can be configured with a SocketsHttpHandler that has a custom ConnectCallback. The callback allows the client to make connections over custom transports and then send HTTP requests over that transport.

> **Note**
>
> Some connectivity features of `GrpcChannel`, such as client side load balancing and channel status, can't be used together with Unix domain sockets.

Unix domain sockets connection factory example:

C#

```
public class UnixDomainSocketsConnectionFactory
{
    private readonly EndPoint endPoint;

    public UnixDomainSocketsConnectionFactory(EndPoint endPoint)
    {
        this.endPoint = endPoint;
    }

    public async ValueTask<Stream> ConnectAsync(SocketsHttpConnectionContext
_,
        CancellationToken cancellationToken = default)
    {
        var socket = new Socket(AddressFamily.Unix, SocketType.Stream,
ProtocolType.Unspecified);

        try
        {
            await socket.ConnectAsync(this.endPoint,
cancellationToken).ConfigureAwait(false);
```

```
            return new NetworkStream(socket, true);
        }
        catch
        {
            socket.Dispose();
            throw;
        }
    }
}
```

Using the custom connection factory to create a channel:

C#

```
public static readonly string SocketPath = Path.Combine(Path.GetTempPath(),
"socket.tmp");

public static GrpcChannel CreateChannel()
{
    var udsEndPoint = new UnixDomainSocketEndPoint(SocketPath);
    var connectionFactory = new
UnixDomainSocketsConnectionFactory(udsEndPoint);
    var socketsHttpHandler = new SocketsHttpHandler
    {
        ConnectCallback = connectionFactory.ConnectAsync
    };

    return GrpcChannel.ForAddress("http://localhost", new GrpcChannelOptions
    {
        HttpHandler = socketsHttpHandler
    });
}
```

Channels created using the preceding code send gRPC calls over Unix domain sockets.

Inter-process communication with gRPC and Named pipes

Article • 07/31/2024

By James Newton-King

.NET supports inter-process communication (IPC) using gRPC. For more information about getting started with using gRPC to communicate between processes, see Inter-process communication with gRPC.

Named pipes is an IPC transport that is supported on all versions of Windows. Named pipes integrate well with Windows security to control client access to the pipe. This article discusses how to configure gRPC communication over named pipes.

Prerequisites

- .NET 8 or later
- Windows

Server configuration

Named pipes are supported by Kestrel, which is configured in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenNamedPipe("MyPipeName", listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http2;
    });
});
```

The preceding example:

- Configures Kestrel's endpoints in ConfigureKestrel.
- Calls `ListenNamedPipe` to listen to a named pipe with the specified name.
- Creates a named pipe endpoint that isn't configured to use HTTPS. For information about enabling HTTPS, see Kestrel HTTPS endpoint configuration.

Client configuration

`GrpcChannel` supports making gRPC calls over custom transports. When a channel is created, it can be configured with a SocketsHttpHandler that has a custom ConnectCallback. The callback allows the client to make connections over custom transports and then send HTTP requests over that transport.

> **Note**
>
> Some connectivity features of `GrpcChannel`, such as client side load balancing and channel status, can't be used together with named pipes.

Named pipes connection factory example:

C#

```
public class NamedPipesConnectionFactory
{
    private readonly string pipeName;

    public NamedPipesConnectionFactory(string pipeName)
    {
        this.pipeName = pipeName;
    }

    public async ValueTask<Stream> ConnectAsync(SocketsHttpConnectionContext
_,
        CancellationToken cancellationToken = default)
    {
        var clientStream = new NamedPipeClientStream(
            serverName: ".",
            pipeName: this.pipeName,
            direction: PipeDirection.InOut,
            options: PipeOptions.WriteThrough | PipeOptions.Asynchronous,
            impersonationLevel: TokenImpersonationLevel.Anonymous);

        try
        {
            await
clientStream.ConnectAsync(cancellationToken).ConfigureAwait(false);
            return clientStream;
        }
        catch
        {
            clientStream.Dispose();
            throw;
        }
```

```
    }
}
```

Using the custom connection factory to create a channel:

C#

```
public static GrpcChannel CreateChannel()
{
    var connectionFactory = new NamedPipesConnectionFactory("MyPipeName");
    var socketsHttpHandler = new SocketsHttpHandler
    {
        ConnectCallback = connectionFactory.ConnectAsync
    };

    return GrpcChannel.ForAddress("http://localhost", new GrpcChannelOptions
    {
        HttpHandler = socketsHttpHandler
    });
}
```

Channels created using the preceding code send gRPC calls over named pipes.

Code-first gRPC services and clients with .NET

Article • 11/05/2024

> **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 and Marc Gravell

Code-first gRPC uses .NET types to define service and message contracts.

Code-first is a good choice when an entire system uses .NET:

- .NET service and data contract types can be shared between the .NET server and clients.
- Avoids the need to define contracts in `.proto` files and code generation.

Code-first isn't recommended in polyglot systems with multiple languages. .NET service and data contract types can't be used with non-.NET platforms. To call a gRPC service written using code-first, other platforms must create a `.proto` contract that matches the service.

protobuf-net.Grpc

> **Important**
>
> For help with protobuf-net.Grpc, visit the protobuf-net.Grpc website or create an issue on the protobuf-net.Grpc GitHub repository.

protobuf-net.Grpc is a community project and isn't supported by Microsoft. It adds code-first support to `Grpc.AspNetCore` and `Grpc.Net.Client`. It uses .NET types annotated with attributes to define an app's gRPC services and messages.

The first step to creating a code-first gRPC service is defining the code contract:

- Create a new project that will be shared by the server and client.
- Add a protobuf-net.Grpc package reference.
- Create service and data contract types.

C#

```
using ProtoBuf.Grpc;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Threading.Tasks;

namespace Shared.Contracts;

[DataContract]
public class HelloReply
{
    [DataMember(Order = 1)]
    public string Message { get; set; }
}

[DataContract]
public class HelloRequest
{
    [DataMember(Order = 1)]
    public string Name { get; set; }
}

[ServiceContract]
public interface IGreeterService
{
    [OperationContract]
    Task<HelloReply> SayHelloAsync(HelloRequest request,
        CallContext context = default);
}
```

The preceding code:

- Defines `HelloRequest` and `HelloReply` messages.
- Defines the `IGreeterService` contract interface with the unary `SayHelloAsync` gRPC method.

The service contract is implemented on the server and called from the client.

Methods defined on service interfaces must match certain signatures depending on whether they're:

- Unary
- Server streaming
- Client streaming

- Bidirectional streaming

For more information on defining service contracts, see the protobuf-net.Grpc getting started documentation.

Create a code-first gRPC service

To add gRPC code-first service to an ASP.NET Core app:

- Add a protobuf-net.Grpc.AspNetCore package reference.
- Add a reference to the shared code-contract project.

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="protobuf-net.Grpc.AspNetCore"
Version="1.0.152" />
  </ItemGroup>

  <ItemGroup>
      <ProjectReference Include="..\Shared\Shared.Contracts.csproj" />
  </ItemGroup>

</Project>
```

- Create a new `GreeterService.cs` file and implement the `IGreeterService` service interface:

```csharp
using Shared.Contracts;
using ProtoBuf.Grpc;

public class GreeterService : IGreeterService
{
    public Task<HelloReply> SayHelloAsync(HelloRequest request,
CallContext context = default)
    {
        return Task.FromResult(
                new HelloReply
```

```
                {
                    Message = $"Hello {request.Name}"
                });
    }
}
```

- Update the `Program.cs` file:

```C#
using ProtoBuf.Grpc.Server;

var builder = WebApplication.CreateBuilder(args);

// Additional configuration is required to successfully run gRPC on
macOS.
// For instructions on how to configure Kestrel and gRPC clients on
macOS, visit https://go.microsoft.com/fwlink/?linkid=2099682

// Add services to the container.
builder.Services.AddCodeFirstGrpc();

var app = builder.Build();

// Configure the HTTP request pipeline.
app.MapGrpcService<GreeterService>();
app.MapGet("/", () => "Communication with gRPC endpoints must be made
through a gRPC client. To learn how to create a client, visit:
https://go.microsoft.com/fwlink/?linkid=2086909");

app.Run();
```

  The preceding highlighted code updates the following:
  - `AddCodeFirstGrpc` registers services that enable code-first.
  - `MapGrpcService<GreeterService>` adds the code-first service endpoint.

gRPC services implemented with code-first and `.proto` files can co-exist in the same app. All gRPC services use gRPC service configuration.

Create a code-first gRPC client

A code-first gRPC client uses the service contract to call gRPC services.

- In the gRPC client `.csproj` file:
  - Add a protobuf-net.Grpc package reference.
  - Add a Grpc.Net.Client package reference.
  - Add a reference to the shared code-contract project.

XML

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Grpc.Net.Client" Version="2.52.0" />
    <PackageReference Include="protobuf-net.Grpc" Version="1.0.152" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\Shared\Shared.Contracts.csproj" />
  </ItemGroup>

</Project>
```

- Update the client `program.cs`

C#

```csharp
// See https://aka.ms/new-console-template for more information
using Grpc.Net.Client;
using ProtoBuf.Grpc.Client;
using Shared.Contracts;

namespace GrpcGreeterClient;

internal class Program
{
    private static async Task Main(string[] args)
    {
        using var channel =
GrpcChannel.ForAddress("https://localhost:7184");
        var client = channel.CreateGrpcService<IGreeterService>();

        var reply = await client.SayHelloAsync(
            new HelloRequest { Name = "GreeterClient" });

        Console.WriteLine($"Greeting: {reply.Message}");
        Console.WriteLine("Press any key to exit...");
        Console.ReadKey();
    }
}
```

The preceding gRPC client `Program.cs` code:

- Creates a gRPC channel.
- Creates a code-first client from the channel with the `CreateGrpcService<IGreeterService>` extension method.
- Calls the gRPC service with `SayHelloAsync`.

A code-first gRPC client is created from a channel. Just like a regular client, a code-first client uses its channel configuration.

View or download sample code (how to download)

Additional resources

- protobuf-net.Grpc website
- protobuf-net.Grpc GitHub repository

gRPC health checks in ASP.NET Core

Article • 08/28/2024

> **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

The gRPC health checking protocol is a standard for reporting the health of gRPC server apps.

Health checks are exposed by an app as a gRPC service. They're typically used with an external monitoring service to check the status of an app. The service can be configured for various real-time monitoring scenarios:

- Health probes can be used by container orchestrators and load balancers to check an app's status. For example, Kubernetes supports gRPC liveness, readiness and startup probes. Kubernetes can be configured to reroute traffic or restart unhealthy containers based on gRPC health check results.
- Use of memory, disk, and other physical server resources can be monitored for healthy status.
- Health checks can test an app's dependencies, such as databases and external service endpoints, to confirm availability and normal functioning.

Set up gRPC health checks

gRPC ASP.NET Core has built-in support for gRPC health checks with the Grpc.AspNetCore.HealthChecks package. Results from .NET health checks are reported to callers.

To set up gRPC health checks in an app:

- Add a `Grpc.AspNetCore.HealthChecks` package reference.
- Register gRPC health checks service:
  - `AddGrpcHealthChecks` to register services that enable health checks.
  - `MapGrpcHealthChecksService` to add a health checks service endpoint.

- Add health checks by implementing IHealthCheck or using the AddCheck method.

C#

```
using GrpcServiceHC.Services;
using Microsoft.Extensions.Diagnostics.HealthChecks;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();
builder.Services.AddGrpcHealthChecks()
                .AddCheck("Sample", () => HealthCheckResult.Healthy());

var app = builder.Build();

app.MapGrpcService<GreeterService>();
app.MapGrpcHealthChecksService();

// Code removed for brevity.
```

When health checks is set up:

- The health checks service is added to the server app.
- .NET health checks registered with the app are periodically executed for health results. By default, there's a 5-second delay after app startup, and then health checks are executed every 30 seconds. Health check execution interval can be customized with HealthCheckPublisherOptions.
- Health results determine what the gRPC service reports:
  - `Unknown` is reported when there are no health results.
  - `NotServing` is reported when there are any health results of HealthStatus.Unhealthy.
  - Otherwise, `Serving` is reported.

Health checks service security

gRPC health checks returns health status about an app, which could be sensitive information. Care should be taken to limit access to the gRPC health checks service.

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, configure the gRPC health checks endpoint with `AllowAnonymous` to skip authentication and authorization.

C#

```
app.MapGrpcHealthChecksService().AllowAnonymous();
```

Configure `Grpc.AspNetCore.HealthChecks`

By default, the gRPC health checks service uses all registered health checks to determine health status. gRPC health checks can be customized when registered to use a subset of health checks. The `MapService` method is used to map health results to service names, along with a predicate for filtering health results:

C#

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

builder.Services.AddGrpc();
builder.Services.AddGrpcHealthChecks(o =>
{
    o.Services.MapService("", r => r.Tags.Contains("public"));
});

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

The preceding code overrides the default service (`""`) to only use health results with the "public" tag.

gRPC health checks supports the client specifying a service name argument when checking health. Multiple services are supported by providing a service name to `MapService`:

C#

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

builder.Services.AddGrpc();
builder.Services.AddGrpcHealthChecks(o =>
{
    o.Services.MapService("greet.Greeter", r => r.Tags.Contains("greeter"));
    o.Services.MapService("count.Counter", r => r.Tags.Contains("counter"));
});

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

The service name specified by the client is usually the default (`""`) or a package-qualified name of a service in your app. However, nothing prevents the client using arbitrary values to check app health.

Configure health checks execution interval

Health checks are run immediately when `Check` is called. `Watch` is a streaming method and has a different behavior than `Check`: The long running stream reports health checks results over time by periodically executing IHealthCheckPublisher to gather health results. By default, the publisher:

- Waits 5 seconds after app startup before running health checks.
- Runs health checks every 30 seconds.

Publisher intervals can be configured using HealthCheckPublisherOptions at startup:

C#

```
builder.Services.Configure<HealthCheckPublisherOptions>(options =>
{
    options.Delay = TimeSpan.Zero;
    options.Period = TimeSpan.FromSeconds(10);
});
```

Call gRPC health checks service

The Grpc.HealthCheck package includes a client for gRPC health checks:

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Health.HealthClient(channel);

var response = await client.CheckAsync(new HealthCheckRequest());
var status = response.Status;
```

There are two methods on the `Health` service:

- `Check` is a unary method for getting the current health status. Health checks are executed immediately when `Check` is called. The server returns a `NOT_FOUND` error response if the client requests an unknown service name. This can happen at app startup if health results haven't been published yet.
- `Watch` is a streaming method that reports changes in health status over time. IHealthCheckPublisher is periodically executed to gather health results. The server returns an `Unknown` status if the client requests an unknown service name.

The `Grpc.HealthCheck` client can be used in a client factory approach:

C#

```
builder.Services
    .AddGrpcClient<Health.HealthClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    });
```

In the previous example, a client factory for `Health.HealthClient` instances is registered with the dependency injection system. Then, these instances are injected into services for executing health check calls.

For more information, see gRPC client factory integration in .NET.

Additional resources

- Health checks in ASP.NET Core
- gRPC health checking protocol
- Grpc.AspNetCore.HealthChecks
- Grpc.HealthCheck

Manage Protobuf references with dotnet-grpc

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.

`dotnet-grpc` is a .NET Core Global Tool for managing Protobuf (.proto) references within a .NET gRPC project. The tool can be used to add, refresh, remove, and list Protobuf references.

Installation

To install the `dotnet-grpc` .NET Core Global Tool, run the following command:

.NET CLI

```
dotnet tool install -g dotnet-grpc
```

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

Add references

`dotnet-grpc` can be used to add Protobuf references as `<Protobuf />` items to the `.csproj` file:

XML

```
<Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
```

The Protobuf references are used to generate the C# client and/or server assets. The `dotnet-grpc` tool can:

- Create a Protobuf reference from local files on disk.
- Create a Protobuf reference from a remote file specified by a URL.
- Ensure the correct gRPC package dependencies are added to the project.

For example, the `Grpc.AspNetCore` package is added to a web app. `Grpc.AspNetCore` contains gRPC server and client libraries and tooling support. Alternatively, the `Grpc.Net.Client`, `Grpc.Tools` and `Google.Protobuf` packages, which contain only the gRPC client libraries and tooling support, are added to a Console app.

Add file

The `add-file` command is used to add local files on disk as Protobuf references. The file paths provided:

- Can be relative to the current directory or absolute paths.
- May contain wild cards for pattern-based file globbing.

If any files are outside the project directory, a `Link` element is added to display the file under the folder `Protos` in Visual Studio.

Usage

.NET CLI

```
dotnet-grpc add-file [options] <files>...
```

Arguments

Expand table

Argument	Description
files	The protobuf file references. These can be a path to glob for local protobuf files.

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.
-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. The default value is `Public`. Accepted values are `Internal` and `Public`.

Add URL

The `add-url` command is used to add a remote file specified by an source URL as Protobuf reference. A file path must be provided to specify where to download the remote file. The file path can be relative to the current directory or an absolute path. If the file path is outside the project directory, a `Link` element is added to display the file under the virtual folder `Protos` in Visual Studio.

Usage

.NET CLI

```
dotnet-grpc add-url [options] <url>
```

Arguments

Expand table

Argument	Description
url	The URL to a remote protobuf file.

Options