gRPC Services in Blazor Applications

---

1. Introduction to gRPC

What is gRPC?

gRPC (Google Remote Procedure Call) is a **high-performance, contract-first, binary communication framework** built on:

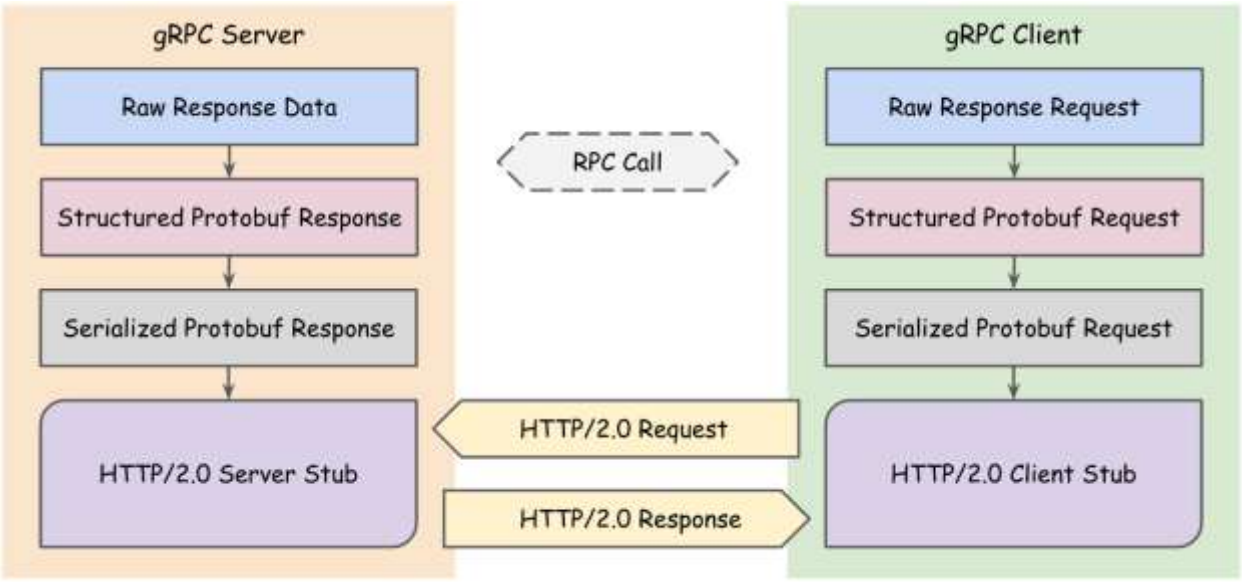
- **HTTP/2**
- **Protocol Buffers (Protobuf)** for serialization
- **Strongly typed service contracts**

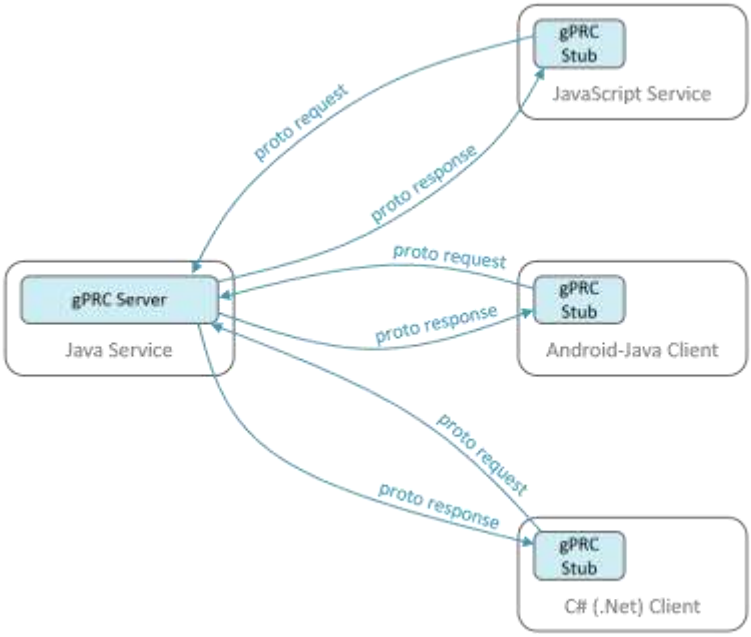
Instead of REST endpoints (/api/products/1), gRPC exposes **methods**:

GetProduct(ProductRequest) → ProductResponse

Key Characteristics

Feature	**Description**
Transport	HTTP/2
Payload	Binary (Protobuf)
Contract	.proto file
Performance	Very high
Streaming	Client, Server & Bi-Directional
Typing	Strongly typed





Why gRPC in Blazor?

- Extremely fast **service-to-service** calls
- Ideal for **Blazor Server**
- Excellent for **microservices**
- Compile-time safety

⚠️ **Important limitation**

- **Blazor WebAssembly (browser)** cannot call gRPC directly (browser HTTP/2 restrictions)
- Solution: **gRPC-Web**

---

2. Comparing gRPC and Web API

REST (Web API) vs gRPC

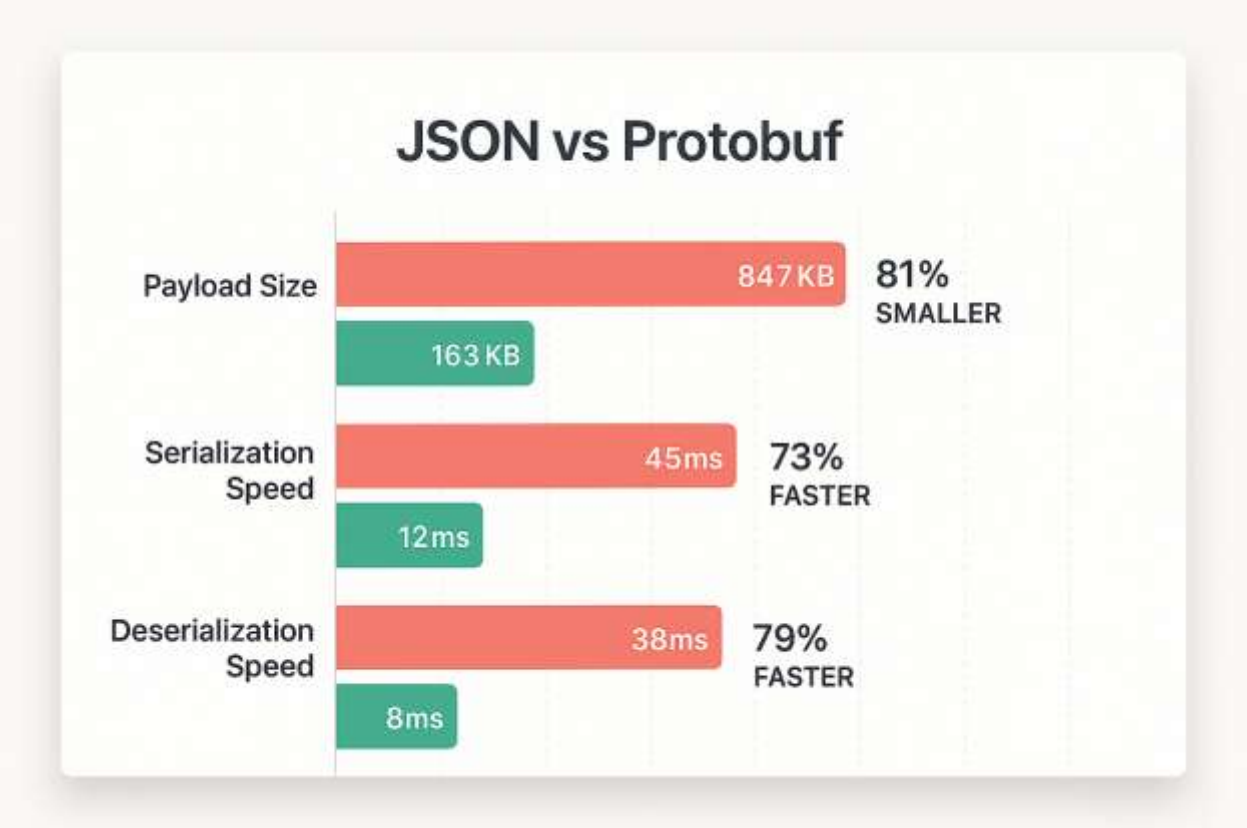
Aspect	**Web API (REST)**	**gRPC**
Protocol	HTTP/1.1 / HTTP/2	HTTP/2
Format	JSON (text)	Protobuf (binary)
Contract	Optional (Swagger)	Mandatory (.proto)
Speed	Moderate	Very fast
Payload Size	Large	Very small
Streaming	Limited	Native
Browser Friendly	Yes	No (needs gRPC-Web)

When to use REST

- Public APIs
- Browser-only clients
- Third-party integrations

When to use gRPC

- Internal microservices
- High-performance systems
- Blazor Server
- Service-to-service communication



3. Implementing gRPC Service in a Blazor Application (.NET 10.0)

We will build:

1. **gRPC Service**
2. **Blazor Server client**
3. **Blazor WebAssembly client (via gRPC-Web)**

3.1 Create a gRPC Service (.NET 10.0)

Step 1: Create gRPC Project

dotnet new grpc -n ProductGrpcService

cd ProductGrpcService

Step 2: Define Protobuf Contract

📁 Protos/product.proto

```
syntax = "proto3";

option csharp_namespace = "ProductGrpcService";

service ProductService {
  rpc GetProduct (ProductRequest) returns (ProductReply);
}

message ProductRequest {
  int32 id = 1;
}

message ProductReply {
  int32 id = 1;
  string name = 2;
  double price = 3;
}
```

Step 3: Implement gRPC Service

📁 Services/ProductServiceImpl.cs

```
using Grpc.Core;

namespace ProductGrpcService.Services;

public class ProductServiceImpl : ProductService.ProductServiceBase
{
```

```
    public override Task<ProductReply> GetProduct(
        ProductRequest request,
        ServerCallContext context)
    {
        return Task.FromResult(new ProductReply
        {
            Id = request.Id,
            Name = "Laptop",
            Price = 75000
        });
    }
}
```

Step 4: Configure gRPC

📁 Program.cs

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

builder.Services.AddGrpc();

var app = builder.Build();

app.MapGrpcService<ProductServiceImpl>();
app.MapGet("/", () => "gRPC Server Running");

app.Run();
```

3.2 Consume gRPC in Blazor Server (.NET 10.0)

Step 1: Create Blazor Server App

dotnet new blazorserver -n BlazorGrpcClient

Step 2: Add gRPC Client Packages

dotnet add package Grpc.Net.Client

dotnet add package Google.Protobuf

dotnet add package Grpc.Tools

Step 3: Add product.proto to Client

📁 Protos/product.proto

```
<ItemGroup>
  <Protobuf Include="Protos\product.proto" GrpcServices="Client" />
</ItemGroup>
```

Step 4: Configure gRPC Client

📁 Program.cs

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

Step 5: Use gRPC in Blazor Component

📁 Pages/Product.razor

```
@inject ProductService.ProductServiceClient Client
```

```
<h3>Product</h3>

<button class="btn btn-primary" @onclick="LoadProduct">
    Load Product
</button>

@if (product != null)
{
    <p>@product.Name - ₹@product.Price</p>
}

@code {
    ProductReply? product;

    async Task LoadProduct()
    {
        product = await Client.GetProductAsync(
            new ProductRequest { Id = 1 });
    }
}
```

3.3 Consume gRPC in Blazor WebAssembly (gRPC-Web)

Key Concept

Browsers **cannot use native gRPC**, so we enable **gRPC-Web**.

Step 1: Enable gRPC-Web on Server

dotnet add package Grpc.AspNetCore.Web

📁 Program.cs

```
app.UseGrpcWeb();


app.MapGrpcService<ProductServiceImpl>()
  .EnableGrpcWeb();
```

Step 2: Configure Blazor WASM Client

```
builder.Services.AddScoped(sp =>
{
   var handler = new GrpcWebHandler(
      GrpcWebMode.GrpcWebText,
      new HttpClientHandler());

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

   return new ProductService.ProductServiceClient(channel);
});
```

Step 3: Use Same Razor Code

The component code remains **unchanged**, which is a major benefit of gRPC.

---

4. Summary

Architecture Recommendation

Scenario	Recommendation
Blazor Server	Native gRPC
Blazor WASM	gRPC-Web
Public APIs	REST
Internal Microservices	gRPC

Key Takeaways

- gRPC is **faster and safer** than REST
- .proto defines the contract
- Ideal for **Blazor Server & microservices**
- Blazor WASM requires **gRPC-Web**
