DOCS

Oleg Sivers

2020

Contents

Website

Website Credentials

Client login: “client1” password: ”111”

Grower login: “grower” password: “111”

Admin login: “sa” password: “111”

Client can get access to auction to place bids, grower can get access to auction to add items. Admin can view both - bid and add items pages.

Articles and growers lists have public access.

Requirements implementation table

Growers/Producers and customers can log in and administer their account	Sing In button at index page. To modify account details use “Account settings” menu or /account Code: \WEB701PRJ\Web701BlazorApp\Components\LoginForm.razor \WEB701PRJ\Web701BlazorApp\Pages\LoginPage.razor \WEB701PRJ\Web701BlazorApp\Auth\*.* \WEB701PRJ\Web701BlazorApp\Components\UserInfoMemberEdit.razor \WEB701PRJ\Web701BlazorApp\Pages\UserDetailsEditPage.razor
Growers/Producers use the auction system to register their products, and customers use the auction system to bid	Add new items for growers and admins: /add Code: \Web701BlazorApp\Components\AddItemsForm.razor \WEB701PRJ\Web701BlazorApp\Components\AddItemsList.razor \WEB701PRJ\Web701BlazorApp\Pages\AddItemsPage.razor Bids for clients and admins: /bid Code: \WEB701PRJ\Web701BlazorApp\Pages\AucBidPage.razor \WEB701PRJ\Web701BlazorApp\Components\AucBidLineCard.razor \WEB701PRJ\Web701BlazorApp\Components\AucBidLine.razor \WEB701PRJ\Web701BlazorApp\Components\AucBidList.razor
A web-based quality calculator that prices the product based on its characteristics	Build-in into AddItemsForm blazor component in method “calculation()” WEB701PRJ\Web701BlazorApp\Components\AddItemsForm.razor
Interactive element(s) that engages the website user	Build-in into index page and default layout (moving leaf at the left side of the screen). Animation done with CSS animation. WEB701PRJ\Web701BlazorApp\Components\CommonContent.razor
gRPC implementation	We can receive article list from a gRPC back-end, gRPC service code located at: WEB701BackEnd\WEB701GRPC\WEB701GRPC\Services\Web701srv.cs gRPC client code can be found here: \WEB701PRJ\Web701BlazorApp\Data\gRPCClient.cs \WEB701PRJ\Web701BlazorApp\Pages\ArticlesPage.razor

Database

Should be restored from a backup file.

Backup database located in the project folder: WEB701A2(MSSQL_DB_BAK).bak

Connection string should be set in appsettings.json file.

gRPC implementation

Description

gRPC is an RPC framework, it could be used for data transfer between client and server within my project.

Net Core 3.1 has a nice implementation of gRPC, which can be used easy and effective, most of code auto generated, based on Proto3 definition of responses, requests and RPC signatures. Developer needs to define messages and a respond logic upon receiving a request. It uses binary transfer over HTTP/2, but it's a higher-level wrap build on top of HTTP/2 blob transfer, to make it easier for developers to transfer data.

Porto3 is a programming language, which used to define typed messages and RPC signatures.

Core benefits of gRPC:

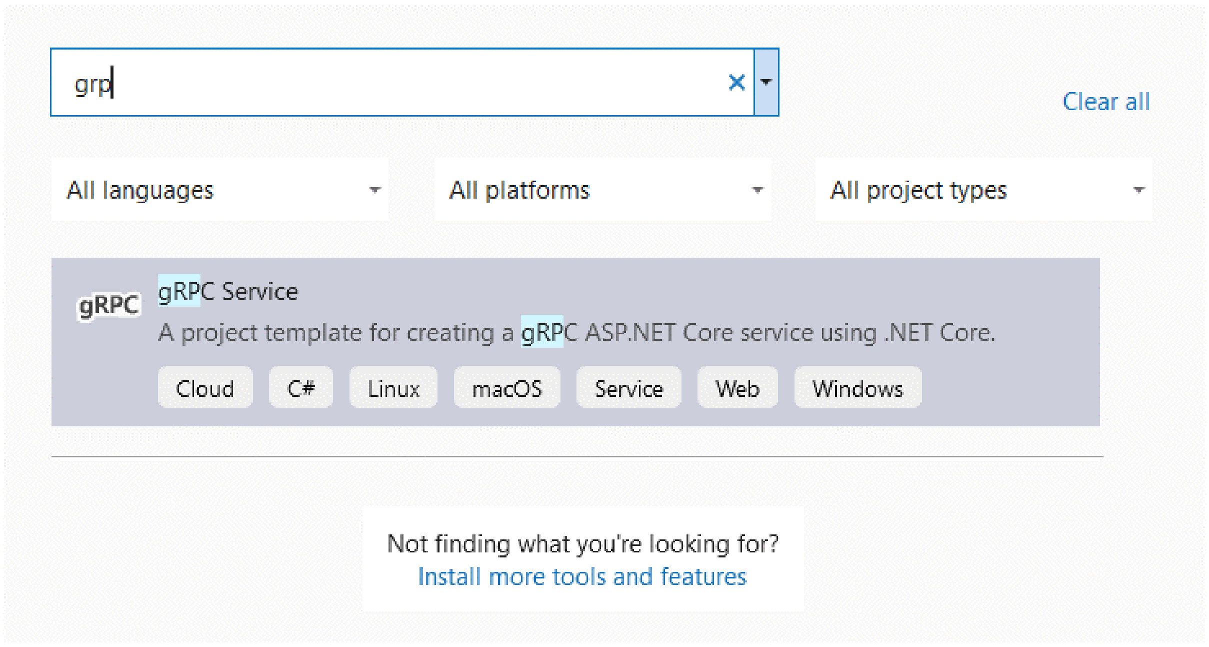
- Scalability
- Works in many programming environments
- Requests and responds are predefined at compilation stage
- It's a high-level wrap over binary blob transfer in HTTP/2

I'm going to implement gRPC for communication with a back-end. Article module will receive data from the gRPC back-end.

Implementation

Server

I've created a project based on gRPC template in Visual Studio:



gRPC Service

A project template for creating a gRPC ASP.NET Core service using .NET Core.

Not finding what you're looking for?

After creating the project, I've defined messages and one remote procedure, for my articles list, with protobuf:

```
syntax = "proto3";

option csharp_namespace = "WEB701GRPC.Protos";

package greet;

// service definition.
service Web701 {
  // Sends a greeting
  rpc GetArticleList (GetArticleListRequest) returns (ArticleListReply);
}

// The request message
message GetArticleListRequest {

}
// The response message containing the list of articles
message ArticleListReply {
  repeated Article articles = 1;
  string errorMsg = 2;
}
message Article {
    int32 id = 1;
    string title = 2;
    string text = 3;
    string imageLink =4;
}
```

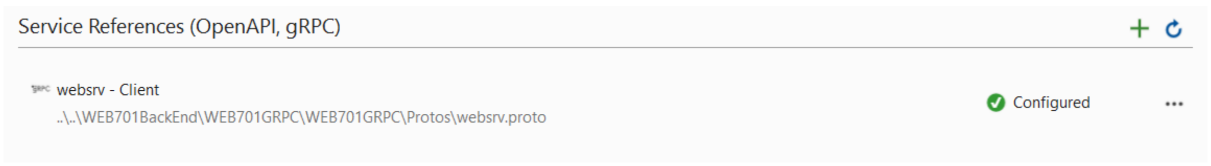
Basically, I've defined the list of articles and article information fields. "`ArticleListReply`" contains a repeated field with "`Article`". After creating proto3 definition of messages and RPCs Visual Studio generated support C# classes for me. I had to create service file and define request-response logic there:

```
public override Task<ArticleListReply> GetArticleList(GetArticleListRequest request
, ServerCallContext context)
{
   string text = @"Lorem ipsum dolor sit amet";
   var response = new ArticleListReply();
   int id = 1;
   response.Articles.Add(new Article { Id = id++, Title = "Nam finibus eget", Text
= text, ImageLink= $"https://picsum.photos/200/200?random={id}>" });
   response.Articles.Add(new Article { Id = id++, Title = "Aenean sit amet", Text =
 text, ImageLink = $"https://picsum.photos/200/200?random={id}>" });
   return Task.FromResult(response);
}
```

The code is just generating several "dummy" articles and random pictures. This code is called upon calling "`GetArticleList`" rpc and generates a response according to proto3 definition.

Client

I had to include proto3 file with RPCs and messages definition from server into client project. Visual studio has GUI for that:


Service References (OpenAPI, gRPC)

websrv - Client
..\..\WEB701BackEnd\WEB701GRPC\WEB701GRPC\Protos\websrv.proto

Configured

Its important to add existing server-side proto3 file here. I've also had to install several NuGet packages, including:

- grpc.AspNetCore
- grpc.Net.Clietn
- grpc.Tools

These packages are needed to support usage of gRPC and C# classes auto generation.

My gRPC client code:

```
public class gRPCClient
{
    public static string backEndAddress = "http://localhost:5001";
    public static async Task<List<Article>> GetArticleList()
    {
        AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);
        using var channel = GrpcChannel.ForAddress(backEndAddress);
        var client = new Web701Client(channel);
        ArticleListReply res = await client.GetArticleListAsync(new GetArticleListRequest());
        if (res.ErrorMsg != "")
           throw new Exception(res.ErrorMsg);
        return res.Articles.ToList();
    }
}
```

It is simply requesting "`GetArticleListAsync`" and returns results as a list of article objects. I had to add non-HTTPS support, to make my life a bit easier without HTTPS configuration. Code is also using Async \ Await, because its an external API call and we don't want to lock a thread.

Last step is to display list of articles within a blazor component:

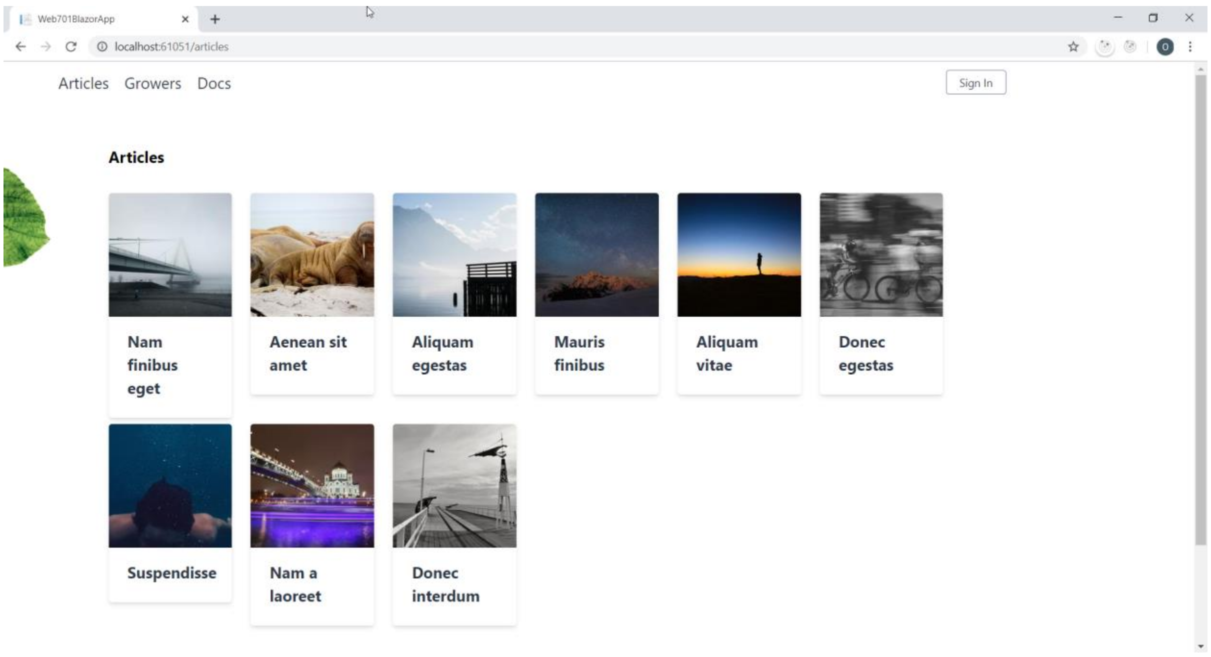
```
@using Web701BlazorApp.Data
@using WEB701GRPC.Protos
@page "/articles"
<Web701BlazorApp.Components.ContentAndFooter>
    <div class="font-bold text-xl mb-8">Articles</div>
    <Web701BlazorApp.Components.CardList PageHandler="article" Lst="@lst"></Web701BlazorApp.Components.CardList>
</Web701BlazorApp.Components.ContentAndFooter>

@code {
    List<Card> lst = new List<Card>();
    protected override async Task OnInitializedAsync()
    {
        try
        {
            lst.Clear();
            List<Article> articles = await gRPCClient.GetArticleList();
            articles.ForEach(a => lst.Add(new Card { Text = a.Text, Id = a.Id, ImageLink = a.ImageLink, Title = a.Title }));
        }
        catch (Exception e)
        {
            Console.WriteLine(e.GetBaseException().Message);
        }
        this.StateHasChanged();
    }
}
```

In my application I have a component to display card-based items, but we have to convert our articles, received from gRPC back-end into suitable card objects. gRPC request is:

```
List<Article> articles = await gRPCClient.GetArticleList();
```

Rest of code is just supporting display of the list. A resulting web page:



Conclusion

It was relatively easy to implement gRPC data transfer within Microsoft stack, because I'm using both server and client based on .Net Core. There were some configuration issues, as developer should manually add ".proto" file link to project file, by manually editing Visual studio project file. Still, official documentation covers almost all aspects of adding gRPC functionality to .Net Core based projects.