

Лабораторна робота №2

Тема: Реалізація CRUD-операцій на базі сервісу gRPC

Мета: Навчитися розробляти клієнт-серверні застосунки для виконання CRUD-операцій з використанням фреймворку gRPC

Теоретичні відомості

Базові типи proto

Для визначення файлу proto, який визначає формат сервісу та повідомлень, застосовується спеціальна мова Protobuf. Protobuf має власну систему типів даних та визначення повідомлень. Стисло розглянемо синтаксис файлу proto і зіставлення його деяких елементів з елементами мови C#.

Protobuf підтримує використання багатьох стандартних примітивних типів, які застосовуються в ряді найпопулярніших мов програмування:

№ з/п	Тип proto	Тип C#
1	double	double
2	float	float
3	int32 (тільки для додатних чисел)	int
4	int64 (тільки для додатних чисел)	long
5	uint32	uint
6	uint64	ulong
7	sint32	int
8	sint64	long
9	fixed32	uint
10	fixed64	ulong
11	sfixed32	int
12	sfixed64	long
13	bool	bool
14	string	string
15	bytes (довільна послідовність байтів довжиною не більше 2^{32})	ByteString

Типи int32 і int64 застосовуються переважно для додатних чисел, а для від'ємних краще застосовувати відповідно sint32 і sint64.

При парсингу повідомлення, якщо для якогось поля повідомлення відсутнє значення, це поле отримує значення за замовчуванням: для рядків – порожній рядок, для чисел – 0, для bool – false.

Визначення повідомлень

Для передачі даних у Protobuf використовуються повідомлення. У мові C# їм відповідають класи. Визначення повідомлення починається з ключового слова message, за яким вказується ім'я повідомлення.

```
message ім'я_повідомлення { }
```

Далі всередині фігурних дужок розміщуються поля повідомлення. Поля визначаються у форматі:

```
тип ім'я = значення;
```

Наприклад, визначимо повідомлення Person із двома полями name та age:

```
message Person{  
    string name = 1;  
    int32 age = 2;  
}
```

Як значення кожному полю передається унікальний номер для ідентифікації кожного поля під час серіалізації повідомлення. Саме тому кожне поле в рамках повідомлення повинно мати унікальне числове значення. Причому при використанні чисел від 1 до 15 як значення в бінарному поданні до повідомлення додається додатковий байт. Значення від 16 до 2047 додають два додаткові байти. Тому для найбільш часто використовуваних даних краще вказувати значення від 1 до 15. Допустимі значення: від 1 до 536870911 (за винятком діапазону чисел від 19000 до 19999).

За цим повідомленням С# буде створюватися клас Person с двома властивостями Age і Name. Якщо спрощено, він відповідатиме приблизно наступному класу:

```
public class Person  
{  
    public string Name { get; set; }  
    public int Age { get; set; }  
    public Person(string name, int age)  
    {  
        Name = name;  
        Age = age;  
    }  
}
```

Як типи поле повідомлень також можуть виступати не лише примітивні типи Protobuf, а й інші повідомлення.

Варто зазначити, що стайлгайд для Protobuf рекомендує використовувати для іменування полів так званий "snake_case" (змійний реєстр), при якому в складовому імені складові розділяються прочерком, наприклад, "first_name". І Microsoft також рекомендує дотримуватися цього стилю найменування, оскільки при генерації відповідних типів С# автоматично застосовуватимуться стандарти іменування .NET. Наприклад, для поля protobuf first_name С# формується властивість FirstName.

Nullable-munu

Для визначення полів, які допускають значення null, (наприклад, для властивості з типом int? в коді С#), в Protobuf доступні типи-обгортки:

№ з/п	Тип proto	Тип С#
1	google.protobuf.BoolValue	bool?
2	google.protobuf.DoubleValue	double?
3	google.protobuf.FloatValue	float?
4	google.protobuf.Int32Value	int?

5	google.protobuf.Int64Value	long?
6	google.protobuf.UInt32Value	uint?
7	google.protobuf.UInt64Value	ulong?
8	google.protobuf.StringValue	string?
9	google.protobuf.BytesValue	ByteString

Для їх використання у файлі .proto необхідно імпортувати файл wrappers.proto:

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

message Person{
    google.protobuf.StringValue name = 1;
    google.protobuf.Int32Value age = 2;
}
```

Визначення сервісу

Сервіс визначається за допомогою ключового слова service, після якого йде ім'я сервісу:

```
service Greeter {
}
```

Тіло сервісу складають функції, що визначаються за допомогою ключового слова rpc:

```
rpc      ім'я_функції      (вхідне_повідомлення)      returns
(вихідне_повідомлення);
```

Після rpc йде ім'я функції, а потім у дужках тип повідомлення, яке отримує сервіс. Далі після оператора returns – тип повідомлення, яке повертає сервіс.

```
service Greeter {
    rpc SayHello (HelloRequest) returns (HelloReply);
}
message HelloRequest {
    string name = 1;
}
message HelloReply {
    string message = 1;
}
```

Сервіс може містити безліч функцій, але варто враховувати, що вона обов'язково має приймати якесь повідомлення та повертати якесь повідомлення. Але при необхідності можна створювати також пусті повідомлення.

```
service Greeter {
    rpc SayHello (HelloRequest) returns (HelloReply);
    rpc Test (VoidRequest) returns (TestResponse);
}
message HelloRequest {
    string name = 1;
}
```

```
message HelloReply {
    string message = 1;
}
// порожнє повідомлення запиту
message VoidRequest{ }
message TestResponse{
    string text = 1;
}
```

Типи методів

Сервіси gRPC можуть визначати різні типи методів. Від типу методів залежить, як сервіс отримуватиме і надсилатиме повідомлення. Підтримуються такі типи методів:

- унарні (сервіс відправляє та отримує звичайне повідомлення);
- потокова передача сервера (сервер отримує звичайне повідомлення, а надсилає потік);
- потокова передача клієнта (сервер отримує потік даних, а надсилає звичайне повідомлення);
- двонаправлена потокова передача (сервер отримує потік даних та відправляє потік даних).

Потоки визначаються за допомогою ключового слова `stream`, яке ставиться перед назвою повідомлення:

```
syntax = "proto3";

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

    // потокова передача сервера
    rpc StreamingFromServer (ExampleRequest) returns (stream
ExampleResponse);

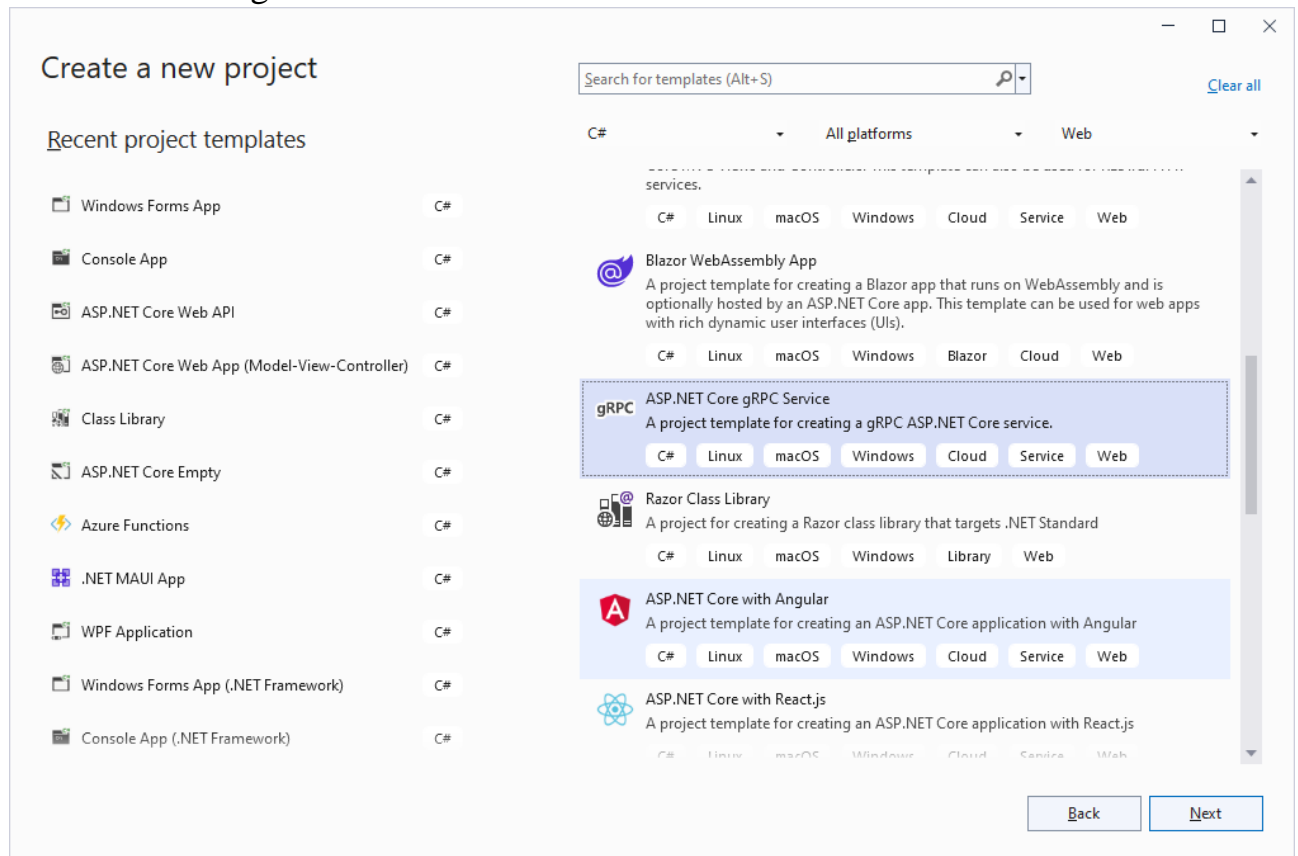
    // потокова передача клієнта
    rpc StreamingFromClient (stream ExampleRequest) returns
(ExampleResponse);

    // двонаправлена потокова передача
    rpc StreamingBothWays (stream ExampleRequest) returns (stream
ExampleResponse);
}
```

Хід виконання роботи:

У попередній роботі проєкт сервісу gRPC розглядалося створювався за допомогою консольних команд `.NET CLI`. Тепер розглянемо, як використовувати Visual Studio для створення аналогічного проєкту.

Visual Studio підтримує спеціальний шаблон створення сервісів gRPC - ASP.NET Core gRPC Service.



Точкою входу в програму за промовчанням є клас Program, який неявно визначений у файлі Program.cs (оператори верхнього рівня, top-level statements). І саме тут і відбувається підключення всієї інфраструктури gRPC:

```
using GrpcService1.Services;

var builder = WebApplication.CreateBuilder(args);

// 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...");

app.Run();
```

Для того щоб задіяти сервіс gRPC, по-перше, додаються необхідні сервіси:

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

По-друге, далі сервіс gRPC вбудовується в систему маршрутизації для обробки запиту:

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

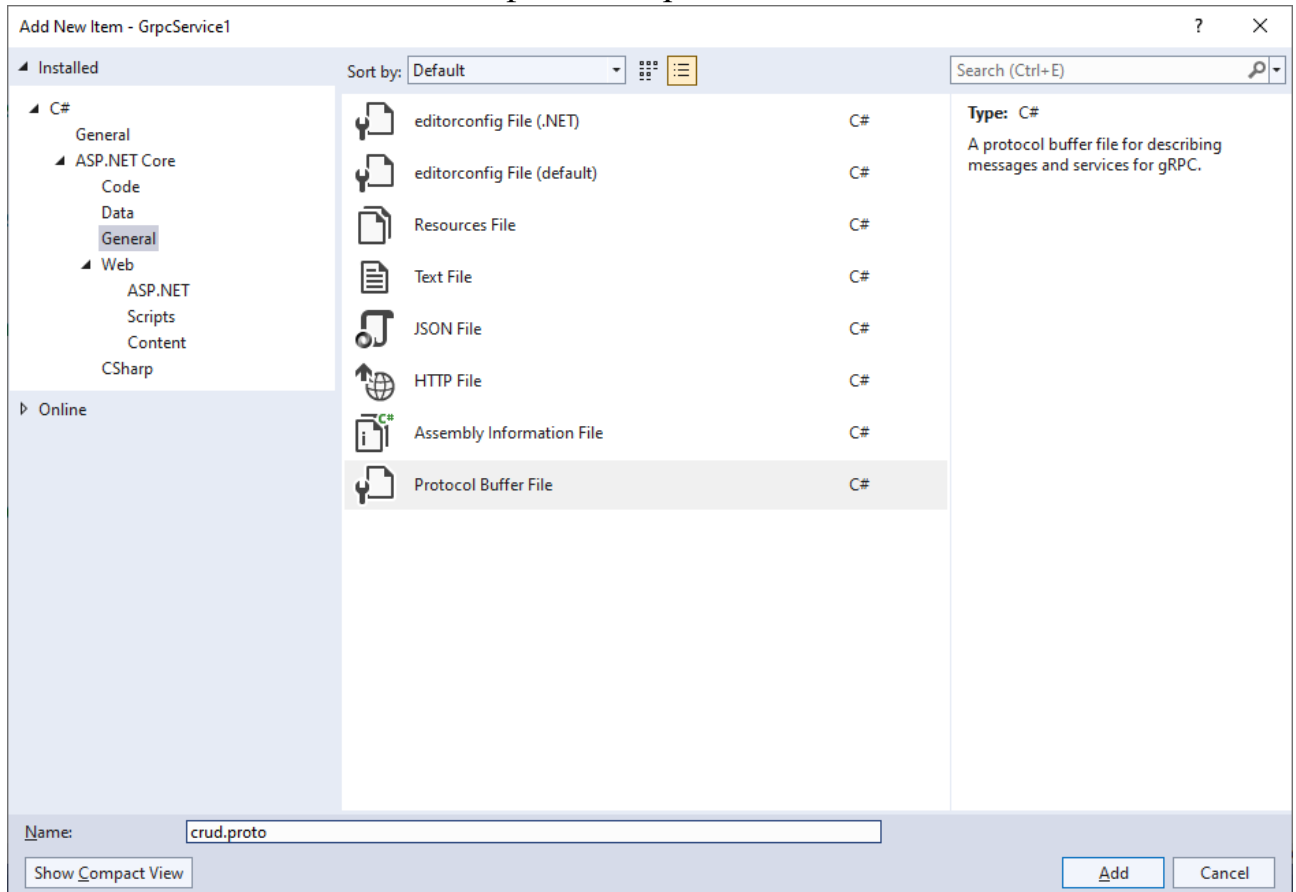
Для вбудовування сервісу застосовується метод MapGrpcService(), який типизується типом сервісу, що вбудовується.

Реалізація CRUD-операцій

Робота з даними в стилі CRUD, зокрема додавання, отримання, оновлення та видалення даних, є досить поширеним завданням веб-додатків. І додаток на gRPC також може визначати функціонал для роботи з даними.

Спочатку створимо проект сервісу типу ASP.NET Core gRPC Service.

У папці Protos визначимо файл crud.proto



Додамо у файл наступний опис gRPC сервісу та повідомлень:

```
syntax = "proto3";
option csharp_namespace = "GrpcService1.Protos";
package crudexample;
import "google/protobuf/empty.proto";
service UserService{

    rpc ListUsers(google.protobuf.Empty) returns (ListReply);
    rpc GetUser(GetUserRequest) returns (UserReply);
    rpc CreateUser(CreateUserRequest) returns (UserReply);
    rpc UpdateUser(UpdateUserRequest) returns (UserReply);
    rpc DeleteUser(DeleteUserRequest) returns (UserReply);
}
message CreateUserRequest{
    string name=1;
    int32 age = 2;
}
message GetUserRequest{
    int32 id =1;
}
```

```

message UpdateUserRequest{
    int32 id=1;
    string name=2;
    int32 age = 3;
}
message DeleteUserRequest{
    int32 id =1;
}
message ListReply{
    repeated UserReply Users = 1;
}
message UserReply{
    int32 id = 1;
    string name=2;
    int32 age = 3;
}

```

Тут визначено сервіс UserService, який отримуватиме дані від клієнта. У сервісі визначено низку методів:

- ListUsers: призначений для надсилання списку об'єктів із умовної бд. Він отримує як параметр порожню відповідь. Для цього використовуємо спеціально певний тип google.protobuf.Empty із пакета google/protobuf/empty.proto, який на початку файлу імпортується.

- GetUser: призначений для надсилання одного об'єкта. Як параметр він отримує повідомлення GetUserRequest, яке містить ID запрошеного об'єкта.

- CreateUser: щоб додати об'єкт. Як параметр він отримує повідомлення CreateUserRequest, яке містить дані, що додаються у вигляді рядка name і числа age. І повертає створений об'єкт.

- UpdateUser: призначено для оновлення об'єкта. Як параметр він отримує повідомлення UpdateUserRequest з новими даними. Як результат повертає новий стан об'єкта.

- DeleteUser: призначений для видалення об'єкта. Як параметр він отримує повідомлення DeleteUserRequest, яке містить id об'єкта, що видаляється. Як результат повертає видалений об'єкт.

Як результат майже всі методи повертають об'єкт UserReply, який представляє самі дані:

```

message UserReply{
    int32 id = 1;
    string name=2;
    int32 age = 3;
}

```

І метод ListUsers, який по суті представляє набір об'єктів UserReply:

```

message ListReply{
    repeated UserReply Users = 1;
}

```

Оператор repeated показує, що це буде набір об'єктів типу UserReply.

Найперший рядок визначає тип синтаксису, що використовується:

```
syntax = "proto3";
```

У цьому випадку застосовується синтаксис "proto3". Якщо явно не вказати версію, то за умовчанням буде використовуватися більш стара версія - proto2.

Далі вказується простір імен, який використовуватиметься з цим сервісом:

```
option csharp_namespace = "GrpcService1.Protos";
```

І відповідно генеровані класи поміщатимуться у цей простір імен. Якщо не вказати цю настройку, то як простір імен буде використовуватися назва пакета PascalCase.

Наступний рядок за допомогою оператора package визначає назву пакета:

```
package crudexample;
```

У даному випадку пакет називається " crudexample ". Встановлення імені пакета дозволяє вирішити конфлікти імен за наявності сутностей з однаковими іменами.

Для додавання файлу .proto до проекту у файлі проекту .csproj повинен бути доданий елемент <Protobuf>:

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

У проекті сервісу grpc, який створюється за шаблоном, цей елемент додається автоматично. Для проектів клієнтів цей пункт слід додавати вручну.

За допомогою атрибута Include вказується шлях до файлу proto. А атрибут GrpcServices описує тип класів, що генеруються, на основі файлу proto. GrpcServices може приймати такі значення:

- Server (генеруються файли сторони сервера)
- Client (генеруються файли сторони клієнта)
- Both (значення за умовчанням, при якому генеруються файли як сервера, так клієнта)
- None (файли не генеруються).

Наприклад, значення GrpcServices="Server" вказує, що генеруватимуться файли, які необхідні для роботи сервісу на стороні сервера.

Proto-файл визначає контракт і залежить від конкретної мови. Однак у додатку C# ми працюємо саме з типами мови C#. І для роботи сервісу та клієнта інструменти .NET самостійно на підставі proto-файлу при необхідності автоматично генерують ряд класів при кожній побудові проекту. Причому генерація здійснюється як у проекті сервера, і у проекті клієнта.

Для генерації подібних класів у проект додається пакет Grpc.Tools (як у проект сервера, і у проект клієнта). Але на стороні сервера ASP.NET також застосовується метапакет Grpc.AspNetCore, який включає зокрема Grpc.Tools і який додається за замовчуванням в проект grpc-сервісу.

```
<ItemGroup>
  <PackageReference Include="Grpc.AspNetCore" Version="2.49.0" />
</ItemGroup>
```


Далі у проекті в папці Services визначимо клас сервісу, який назовемо UserApiService і який матиме наступний код:

```
using Google.Protobuf.WellKnownTypes;
using Grpc.Core;
using GrpcService1.Protos;

namespace GrpcService1.Services
{
    public class UserApiService : UserService.UserServiceBase
    {
        static int id = 0; // лічильник для генерації id
                          // умовна база даних
        static List<User> users = new() { new User(++id, "Tom", 38), new User(++id,
"Bob", 42) };

        // відправка списку користувачів
        public override Task<ListReply> ListUsers(Empty request, ServerCallContext
context)
        {
            var listReply = new ListReply(); // відправка списку
            // перетворюємо кожний об'єкт зі списку users у об'єкт UserReply
            var userList = users.Select(item => new UserReply { Id = item.Id, Name =
item.Name, Age = item.Age }).ToList();
            listReply.Users.AddRange(userList);
            return Task.FromResult(listReply);
        }
        // відправка одного користувача за id
        public override Task<UserReply> GetUser(GetUserRequest request,
ServerCallContext context)
        {
            var user = users.Find(u => u.Id == request.Id);
            // якщо користувача не знайдено, генеруємо виключення
            if (user == null)
            {
                throw new RpcException(new Status(StatusCode.NotFound, "User not
found"));
            }
            UserReply userReply = new UserReply() { Id = user.Id, Name = user.Name,
Age = user.Age };
            return Task.FromResult(userReply);
        }
        // додавання користувача
        public override Task<UserReply> CreateUser(CreateUserRequest request,
ServerCallContext context)
        {
            // формуємо з даних об'єкт User та додаємо його у список users
            var user = new User(++id, request.Name, request.Age);
            users.Add(user);
            var reply = new UserReply() { Id = user.Id, Name = user.Name, Age =
user.Age };
            return Task.FromResult(reply);
        }
        // оновлення користувача
        public override Task<UserReply> UpdateUser(UpdateUserRequest request,
ServerCallContext context)
        {
            var user = users.Find(u => u.Id == request.Id);

            if (user == null)
            {
                throw new RpcException(new Status(StatusCode.NotFound, "User not
found"));
            }
            // оновлюємо дані
            user.Name = request.Name;
            user.Age = request.Age;
        }
    }
}
```

```

        var reply = new UserReply() { Id = user.Id, Name = user.Name, Age =
user.Age };
        return Task.FromResult(reply);
    }
    // видалення користувача
    public override Task<UserReply> DeleteUser(DeleteUserRequest request,
ServerCallContext context)
    {
        var user = users.Find(u => u.Id == request.Id);

        if (user == null)
        {
            throw new RpcException(new Status(StatusCode.NotFound, "User not
found"));
        }

        users.Remove(user);
        var reply = new UserReply() { Id = user.Id, Name = user.Name, Age =
user.Age };
        return Task.FromResult(reply);
    }
}
class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public User(int id, string name, int age)
    {
        Id = id;
        Name = name;
        Age = age;
    }
}
}

```

Як модель даних тут виступає клас User з трьома властивостями, який представляє умовного користувача. У класі UserApiService для простоти імітуємо базу даних як статичного списку users з двома початковими об'єктами.

```

static int id = 0; // лічильник для генерації id
static List<User> users = new() { new User(++id, "Tom", 38),
    new User(++id, "Bob", 42) };

```

Для генерації ідентифікаторів визначено допоміжну статичну змінну id.

У методі ListUsers повертаємо список об'єктів:

```

public override Task<ListReply> ListUsers(Empty request, ServerCallContext context)
{
    var listReply = new ListReply(); // відправка списку
    // перетворюємо кожний об'єкт зі списку users у об'єкт UserReply
    var userList = users.Select(item => new UserReply { Id = item.Id, Name =
item.Name, Age = item.Age }).ToList();
    listReply.Users.AddRange(userList);
    return Task.FromResult(listReply);
}

```

Як параметр приймаємо порожній запит, для якої використовуємо спеціальний тип Google.Protobuf.WellKnownTypes.Empty. У методі визначаємо об'єкт ListReply, у його властивість Users передаємо всі об'єкти зі списку users, перетворюючи їх у тип UserReply. І надсилаємо сформований об'єкт ListReply клієнту.

У методі GetUser клієнту відправляємо одного користувача:

```

public override Task<UserReply> GetUser(GetUserRequest request, ServerCallContext
context)
{

```

```

var user = users.Find(u => u.Id == request.Id);
// якщо користувача не знайдено, генеруємо виключення
if (user == null)
{
    throw new RpcException(new Status(StatusCode.NotFound, "User not found"));
}
UserReply userReply = new UserReply() { Id = user.Id, Name = user.Name, Age =
user.Age };
return Task.FromResult(userReply);
}

```

З параметра request отримуємо id запитаного користувача та шукаємо відповідний об'єкт у списку users. Цілком можливо, що користувач із зазначеним id не буде знайдено. І тут генеруємо виняток RpcException. У конструктор виключення ми можемо передати об'єкт Status, у якому за допомогою властивості StatusCode можна вказати статусний код, аналогічно тому, як це робиться для встановлення статусних кодів http. Крім того, можна вказати рядок статусу із зазначенням причини помилки. І якщо буде згенеровано виняток, клієнт отримає ці дані.

Якщо користувача знайдено у списку users, то перетворимо його на тип UserReply і відправляємо клієнту.

Інші методи однотипні. Метод CreateUser додає одного користувача до списку. Метод UpdateUser за отриманими даними оновлює певного користувача. І метод DeleteUser видаляє по отриманому id користувача зі списку users. У всіх випадках результат методу надсилається користувачеві.

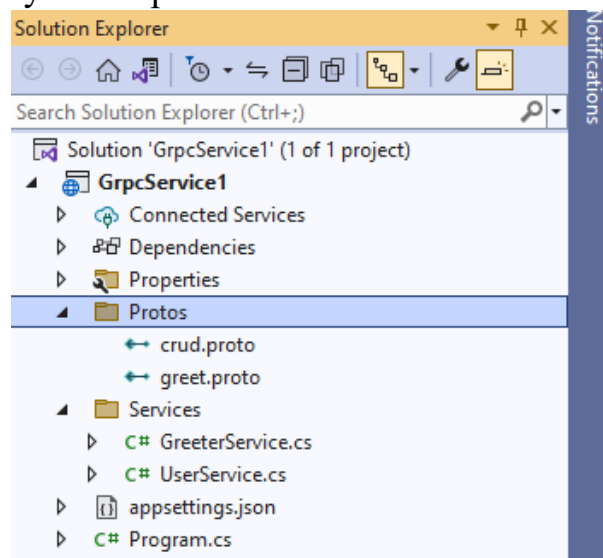
Наприкінці включимо сервіс UserApiService до конвеєра обробки запиту у файлі Program.cs:

```

using GrpcService1.Services;
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddGrpc();
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapGrpcService<UserApiService>();
app.MapGet("/", () => "Hello KN-22m!");
app.Run();

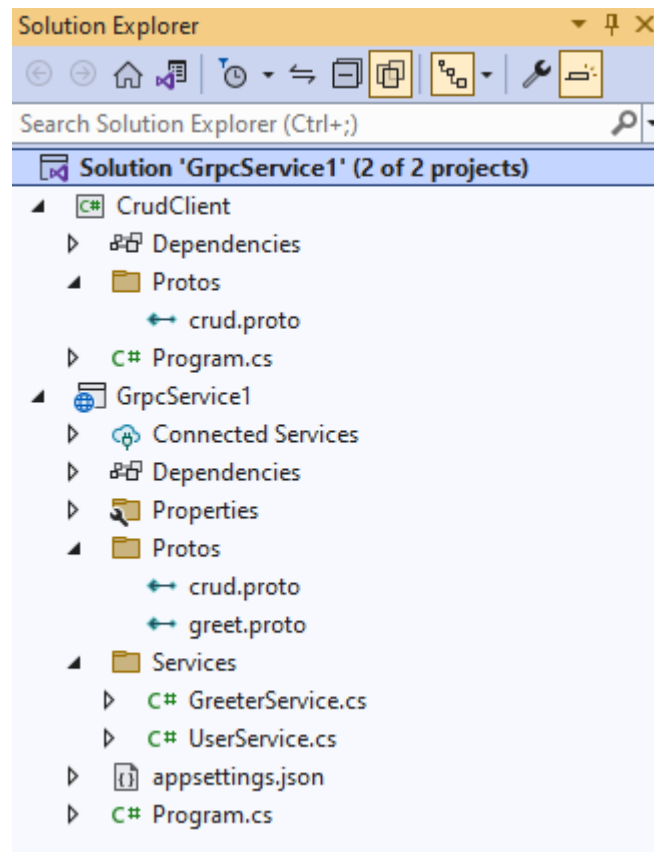
```

Тобто вийде наступний проект:



Створення клієнта

Для тестування сервісу визначимо проект консольної програми. Вийде наступний проект:



Додамо до нього через Nuget всі необхідні пакети:

- Grpc.Net.Client: містить функціонал клієнта.
- Google.Protobuf: містить API для повідомлень protobuf для мови C#.
- Grpc.Tools: містить інструменти для підтримки protobuf-файлів у C#

Тому додамо до проекту консольного клієнта ці пакети, послідовно виконавши такі команди:

```
dotnet add package Grpc.Net.Client
dotnet add package Google.Protobuf
dotnet add package Grpc.Tools
```

Далі створимо в проекті консольної програми нову папку Protos і в неї скопіюємо з проекту сервісу файл crud.proto

Далі нам треба відредагувати файл проекту з розширенням csproj, який називається на ім'я проекту – вузол <Project> додамо наступний елемент:

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

Для отримання списку користувачів у файлі Program.cs визначимо наступний код:

```
using Grpc.Net.Client;
using GrpcService1.Protos;
// створюємо канал для обміну повідомленнями з сервером
// параметр – адрес серверу gRPC
```

```
using var channel = GrpcChannel.ForAddress("https://localhost:7203");
// створюємо клієнта
var client = new UserService.UserServiceClient(channel);
// отримання списку
ListReply users = await client.ListUsersAsync(new
Google.Protobuf.WellKnownTypes.Empty());
foreach (var user in users.Users)
{
    Console.WriteLine($"{user.Id}. {user.Name} - {user.Age}");
}
```

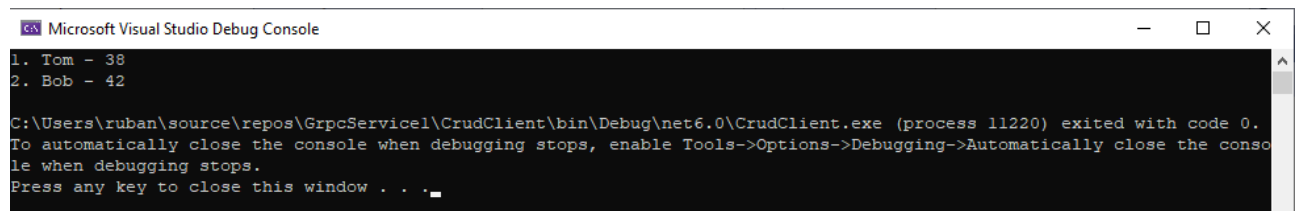
Оскільки сервер повертає об'єкт ListReply, відповідно на клієнті ми отримуємо цей об'єкт і за допомогою його властивості Users можемо отримати всі дані користувача у вигляді об'єкта UserReply.

Для тестування отримання одного об'єкта з id визначимо наступний код:

```
using Grpc.Net.Client;
using GrpcService1.Protos;
using var channel = GrpcChannel.ForAddress("https://localhost:7203");
// створюємо клієнта
var client = new UserService.UserServiceClient(channel);
// отримання одного об'єкту за id = 1
UserReply user1 = await client.GetUserAsync(new GetUserRequest { Id = 1 });
Console.WriteLine($"{user1.Id}. {user1.Name} - {user1.Age}");
```

Однак, якщо користувача за певним id не виявиться, то сервер генерує виняток RpcException, надсилає клієнту відповідний статус. Якщо на стороні сервера генерується виняток, то на клієнті теж автоматично генерується виняткова ситуація. І в цьому випадку, щоб програма не завершилася аварійно, ми можемо обробити виняток, отримавши статус відповіді:

```
using Grpc.Core;
using Grpc.Net.Client;
using GrpcService1.Protos;
using var channel = GrpcChannel.ForAddress("https://localhost:7203");
var client = new UserService.UserServiceClient(channel);
try
{
    // отримання одного об'єкту за id = 3
    UserReply user = await client.GetUserAsync(new GetUserRequest { Id = 3 });
    Console.WriteLine($"{user.Id}. {user.Name} - {user.Age}");
}
catch (RpcException ex)
{
    Console.WriteLine(ex.Status.Detail);    // отримуємо статус відповіді
}
```



```
Microsoft Visual Studio Debug Console
1. Tom - 38
2. Bob - 42
C:\Users\ruban\source\repos\GrpcService1\CrudClient\bin\Debug\net6.0\CrudClient.exe (process 11220) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Створення нового об'єкта:

```
using Grpc.Core;
using Grpc.Net.Client;
using GrpcService1.Protos;
using var channel = GrpcChannel.ForAddress("https://localhost:7203");
var client = new UserService.UserServiceClient(channel);

// додавання одного об'єкту
```

```
UserReply user1 = await client.CreateUserAsync(new CreateUserRequest { Name = "Sam",
Age = 28 });
Console.WriteLine($"{user1.Id}. {user1.Name} - {user1.Age}");
```

Оновлення об'єкту:

```
using Grpc.Core;
using Grpc.Net.Client;
using GrpcService1.Protos;
using var channel = GrpcChannel.ForAddress("https://localhost:7203");
var client = new UserService.UserServiceClient(channel);
try
{
    //оновлення одного об'єкту - змінимо ім'я у об'єкту з id = 1 на Tomas
    UserReply user = await client.UpdateUserAsync(new UpdateUserRequest { Id = 1, Name
= "Anna", Age = 38 });
    Console.WriteLine($"{user.Id}. {user.Name} - {user.Age}");
}
catch (RpcException ex)
{
    Console.WriteLine(ex.Status.Detail);
}
```

І тестування видалення об'єкта:

```
using Grpc.Core;
using Grpc.Net.Client;
using GrpcService1.Protos;
using var channel = GrpcChannel.ForAddress("https://localhost:7203");
var client = new UserService.UserServiceClient(channel);
try
{
    // видалення об'єкту з id = 2
    UserReply user = await client.DeleteUserAsync(new DeleteUserRequest { Id = 2 });
    Console.WriteLine($"{user.Id}. {user.Name} - {user.Age}");
}
catch (RpcException ex)
{
    Console.WriteLine(ex.Status.Detail);
}
```

Завдання

З використанням шаблону проєкту ASP.NET Core gRPC Service створити сервіс gRPC для виконання CRUD-операцій з об'єктами заданого класу. Розробити тестовий клієнт. Тип клієнту обрати самостійно (консольний, десктопний або веб-додаток).

№ варіанту	Заданий клас
1	Книга (Id, автор, назва, рік видання, ціна)
2	Поточне значення технологічного параметру (Id, назва параметру, дата та час вимірювання, одиниця вимірювання)
3	Грошовий переказ (Id, код валюти згідно ISO 4217, сума переказу, IBAN отримувача)
4	Товар (Id, виробник, назва, вартість)
5	Автомобіль (Id, марка, модель, рік випуску, потужність двигуна, вартість)
6	Складова рецепту (Id, назва компонента, кількість, одиниця вимірювання)

7	Вакансія (Id, назва посади, назва компанії, перелік вимог)
8	Співробітник (Id, ПІБ, посада, дата прийняття на роботу)
9	Музичний альбом (Id, виконавець, назва альбому, рік виходу, кількість пісень, тираж)
10	Музична група (Id, назва, рік заснування, список учасників)
11	Місто (Id, назва, область, рік заснування, кількість мешканців, площа)
12	Електродвигун (Id, марка, рік випуску, номінальна напруга, номінальний струм, номінальна частота обертання, дата сервісного обслуговування)
13	Студент (Id, ПІБ, код спеціальності, рік вступу, код академічної групи)
14	Музичний інструмент (Id, назва, країна, тип (струнний, духовий, ударний тощо), опис)
15	Процесор (Id, бренд, серія, модель, частота, кількість ядер, тепловиділення)
16	Відеокарта (Id, бренд, виробник чіпа, назва графічного процесора, інтерфейс, форм-фактор, тип пам'яті, частота пам'яті, об'єм вбудованої пам'яті, частота роботи GPU)
17	Фільм (Id, назва, рік виходу на екран, кіностудія, тривалість, бюджет, касові збори, рейтинг IMDb)
18	Картина (Id, назва, автор, рік, ринкова вартість, місце зберігання)
19	Країна (Id, назва, столиця, населення, площа, бюджет, офіційні мови, код валюти згідно ISO 4217)
20	Мова програмування (Id, назва, дата появи, автор, перелік відомих проєктів)

Звіт повинен включати:

1. Тему, мету та опис ходу виконання завдання.
2. Код реалізації gRPC-сервісу.
3. Код реалізації клієнту.
4. Скріни з демонстрацією роботи додатку.
5. Архів з кодом сервісу та клієнту.

Перелік рекомендованих джерел

1. <https://protobuf.dev/programming-guides/proto/>
2. <https://learn.microsoft.com/en-us/dotnet/architecture/grpc-for-wcf-developers/protobuf-data-types>
3. <https://learn.microsoft.com/ru-ru/aspnet/core/grpc/test-tools?view=aspnetcore-7.0>