gRPC

Table des matières

Chapitre 1 : Introduction 4

1. Introduction 4
2. A qui est destiné ce livre ? 5

Chapitre 2 : API Basics 7

1. Qu'est-ce qu'une API ? 7
   1.1 Quel est le besoin ? (Exposition du problème) 7
   1.2 Quel est le besoin ? (Exposition de la solution) 8
   1.3 Qu'est-ce qu'une Web API et qu'allons-nous utiliser ? 10
2. Les types d'API 11
   2.1 l'API du système d'exploitation. 11
   2.2 l'API Library. 13
   2.3 l'API à distance ou remote API 14
   2.3 Web API 15

Chapitre 3: Web APIs 17

1. Web APIs 17
2. SOAP 18
3. REST 20
4. GraphQL 21
5. gRPC 23

Chapitre 4: gRPC Basics 24

1. Les problèmes des API REST 24
2. Histoire de gRPC 26
3. Les base de gRPC 27
4. RPC 28
5. Les types de communication 29
6. ProtoBuf 32
7. Notions avancées 33

Chapitre 5 : Préparation de l'environnement 35

1. Introduction 35
2. Installer le .NET SDK et Visual Studio 35

Chapitre 6 : Notre application 38

2. Chat App 38

Chapitre 7 : Protobuf 40

1. Introduction 40

2. Flux d’utilisation de Protobuf 41
3. Syntaxe basique des Messages 43
4. Configurer Protobuf avec .NET Core 48
5. Developer avec Protobuf 52
8. Utiliser Oneof 57
9. Maps 59
10. Définir des Services 61
Chapitre 8 : Monter le serveur chat Room 62
1. Introduction 62
2. Créer et configure le projet 63
3. Créer le Chat Room Service 64
4. Tester le Service avec BloomRPC 66
Chapitre 9 : Appels unaires RPC 67
1. Introduction 68
2. Implémentation du Client 68
Chapitre 10 : Client-side Streaming 72
1. Introduction 72
2. Implémenter le stream coté client sur le serveur 73
3. Implémenter le Client news bot 76
Chapitre 11 : Server-side Streaming 80
1. Introduction 80
2. Implémenter le Server-side Streaming sur le serveur 80
3. Connecter le News Bot au serveur de Stream 82
4. Implémenter le Client 85
Chapitre 12 : Streaming Bi-Directionnel 88
1. Introduction 88
2. Implémenter le streaming Bi- Directionnel sur le Server 88
3. Implémenter le Client 95
Chapitre 13 : Sujets avancés 103
1. Introduction 103
2. Deadlines 103
3. Expérimenter les Deadlines 104
4. Gestion des erreurs 105
5. Annuler des requêtes 106
6. Implémenter les Request Cancellation 107

7. Authentification 109
8. Introduction à OAuth 110
9. Ajouter de l'Authentification à l'application de Chat 111
Chapitre 14 : gRPC dans le navigateur 117
1. Introduction 117
2. Utiliser gRPC-Web avec Blazor 118
Chapitre 15 : Conclusion 132
1. Conclusion 132

Chapitre 1 : Introduction

1. Introduction

L'API Web est généralement la partie la plus importante d'une application Web.

C'est ainsi que vous exposez votre application Web au monde et aux autres utilisateurs, et elle doit être rapide, facile à utiliser et à jour.

gRPC est l'une des API Web les plus avancées et les plus intéressantes de l'industrie aujourd'hui, et elle ajoute beaucoup de valeur à n'importe quelle application Web. Des fonctionnalités telles que le streaming serveur et client, des messages fortement typés, des performances ultra-rapides et bien plus encore en font un atout important dans chaque boîte à outils de développeur et d'architecte.

Et ce livre ainsi que le cas pratique feront de vous un connaisseur en gRPC.

Nous allons couvrir tous les aspects de gRPC, des bases aux sujets les plus avancés.

Voici quelques-uns des sujets dont nous allons discuter :

- Comment opposer gRPC face à l'API REST ?
- Concepts de base de gRPC
- Les 4 types de communication de gRPC :
  - RPC unaire

- Diffusion côté client
- Diffusion côté serveur
- Diffusion bidirectionnelle

- Bonnes pratiques de conception d'API gRPC

- Utilisation de Protobuf, le format de sérialisation des messages utilisé par gRPC

- Gestion des erreurs dans gRPC

- Autorisation et sécurité

Et beaucoup plus.

Maintenant, afin de rendre ce livre aussi précieux que possible, je l'ai rendu extrêmement utile et pratique.

Nous allons créer ensemble une application de chat entièrement fonctionnelle et complète basée sur gRPC, en utilisant tous les concepts que nous apprendrons au cours de la lecture, et nous allons utiliser pour cela la plates-forme .NET.

Remarque : Vous n'avez pas besoin d'être un développeur .NET confirmé pour suivre ce cours. Je vais vous guider à travers toutes les étapes du processus de développement et m'assurer que tout fonctionnera comme prévu.

À la fin de cet ouvrage, vous serez compétent sur technologie gRPC, pas seulement en théorie, mais en pratique.

Avant de commencer, je vous propose de récupérer le code source des solutions que je vais vous montrer ici :

https://github.com/AlexCastroAlex/gRPC.FromZeroToHero

2. A qui est destiné ce livre ?

La technologie gRPC est relativement accessible mais elle nécessite quelques prérequis.

À qui ce cours s'adresse-t-il ?

- Les développeurs qui voudraient apprendre gRPC
- Les architectes applicatifs qui souhaiteraient créer de nouvelles API
- N'importe qui dans le monde du développement

Y a-t-il des exigences ou prérequis pour ce cours ?

- Comprendre les concepts basiques des API
- Comprendre les concepts basiques d'HTTP
- Une expérience dans le développement

Chapitre 2 : API Basics

1. Qu'est-ce qu'une API ?

1.1 Quel est le besoin ? (Exposition du problème)

Avant toute chose, nous avons besoin de comprendre pourquoi il est important d'avoir recours à des Web API.

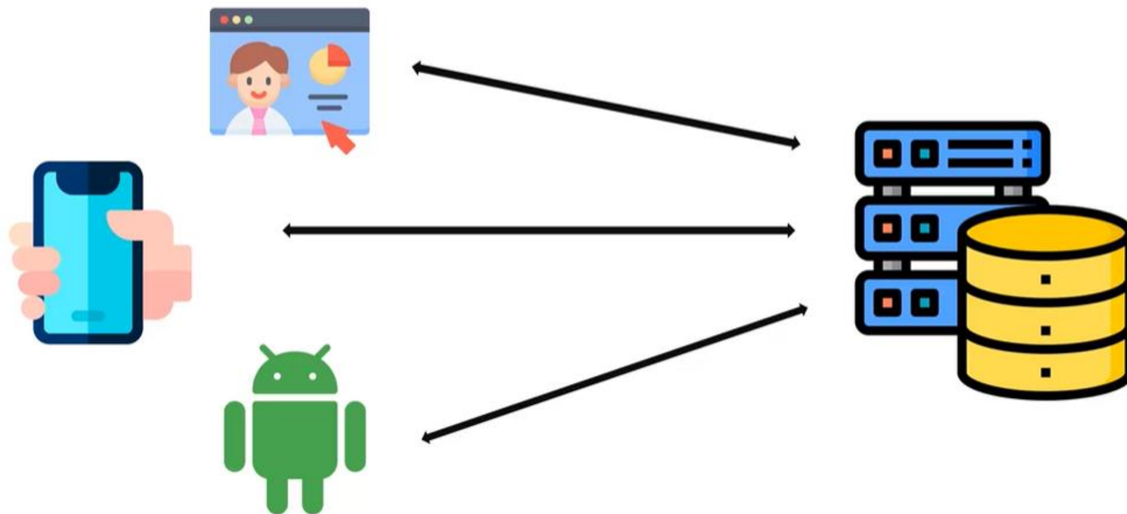
Imaginons que nous ayons un front end dans un langage donné (PHP , ASP MVC , Angular…) qui interroge une base de données comme dans le schéma suivant :

Le besoin est clair et bien défini. En termes d'architecture nous avons une application Web et une base de données qui va contenir les données de l'application.

L'application Web va contenir une multitude services qui vont faire des appels à la base de données.

Cette architecture est certes sommaire mais est totalement fonctionnelle.

Maintenant rajoutons une application IOS et une application Android, ce qui nous donnera le schéma suivant :

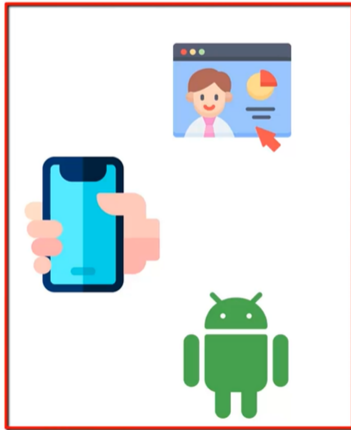


Etant donné que les applications communiquent toutes en même temps vers la même base de données, cela va donc créer des problèmes :

- Duplication de la logique métier dans chaque application
- Code sujet aux erreurs : en lien avec la remarque ci-dessus, la duplication (copier/coller) entraine des erreurs et des différences dans la logique métier implémentée dans chaque application
- Potentiellement, des frameworks front n'auront pas la capacité de communiquer avec votre base de données directement.
- Difficultés de maintenance : en effet, si vous modifiez la logique métier dans une application, il va falloir modifier les mêmes fragments de code dans les autres applications ce qui peut entrainer des erreurs ou des différences mais quoi qu'il arrive c'est une perte de temps sèche.

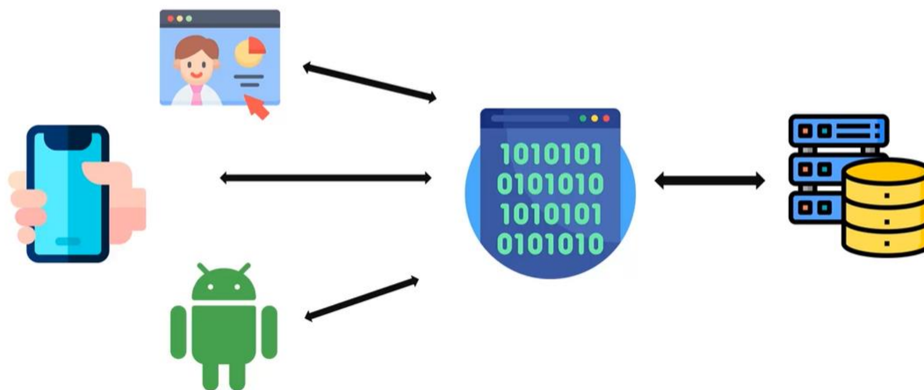
1.2 Quel est le besoin ? (Exposition de la solution)

Maintenant que nous avons les problèmes liés à la non-utilisation de Web API, nous allons exposer une solution et des arguments qui vont aller dans le sens de son utilisation.



Pour rappel, nous avons à gauche nos applications Web métier qui ont toutes besoin de communiquer avec la base de données à droite et nous avons besoin d'une solution pour mutualiser la logique métier et les appels.

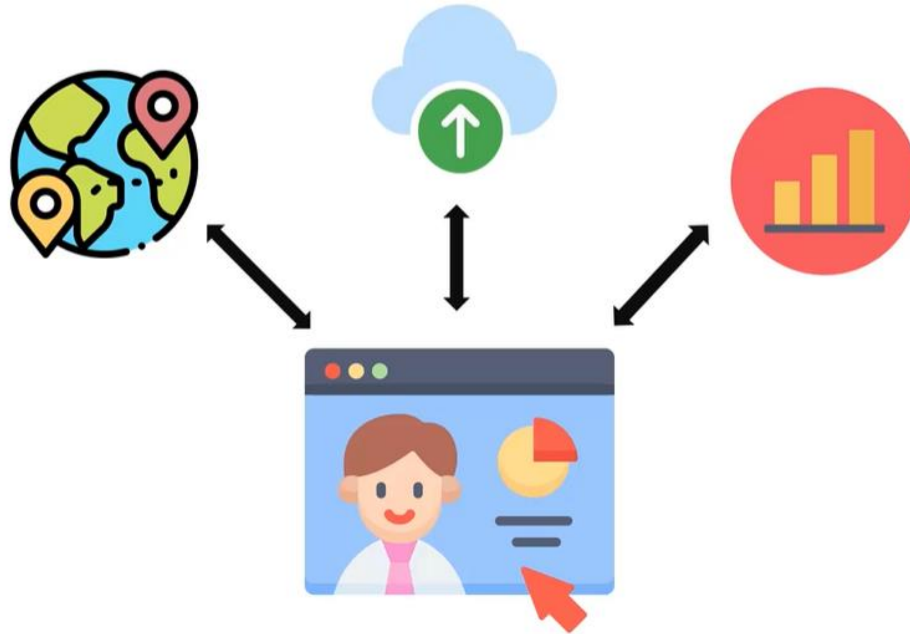
La solution sera donc la suivante :



Nous allons faire communiquer nos applications avec un point central qui lui va communiquer avec la base de données. Ce point central c'est bien sur une Web API vous l'aurez deviné !

La logique sera donc contenue dans ce point central et non plus dans chaque application avec potentiellement des différences.

Ici nous avons mis en avant un exemple simple à comprendre mais l'utilisation des Web API ont d'autres avantages et utilisations.



Dans la capture ci-dessus, nous démontrons que nous pouvons consommer des Web API créées par d'autres et bénéficier de fonctionnalités nouvelles pour nos applications : géolocalisation, données publiques fournies par le gouvernement etc

En résumé :

- Les applications Front end ne doivent pas communiquer directement avec la base de données. Elles ont besoin d'un médiateur (donc d'une API)
- Eviter la duplication de la logique métier
- Etendre les fonctionnalités applicatives : penser au futur et au fait que de nouvelles applications vont se connecter
- Abstraction : l'API est une couche d'abstraction qui cache la logique métier à l'utilisateur final.
- Sécurité : l'API permet de rajouter aussi une couche de sécurité car elle peut restreindre les applications qui interagissent avec elle et lisser les droits également.

1.3 Qu'est-ce qu'une Web API et qu'allons-nous utiliser ?

Nous avons donc établi précédemment que le besoin de Web API était essentiel aujourd'hui et en effet dans les sociétés de service, il représente la majeure partie des développements pour les clients finaux.

Nous allons donc maintenant clarifier ce qu'est une API.

API signifie littéralement : **A**pplication **P**rogramming **I**nterface.

Web API est un concept avant tout (et non pas une technologie) qui fonctionne avec le protocole http et est utilisé pour étendre les fonctionnalités d'une application.

L'extension de fonctionnalités d'une application est matérialisée par le fait qu'une nouvelle application Web peut consommer votre Web API pour bénéficier des fonctionnalités offertes par celle-ci.

Si ce n'est pas une technologie, alors comment va-t-on créer notre Web API ?

Aujourd'hui, différents langages permettent le développement de Web API comme Java , Node, .NET etc

Chaque technologie offre un panel d'outils ou de frameworks afin de développer des API.

2. Les types d'API

Nous avons parlé des principes de base de ce qu'est une API.

Mais une autre chose importante à savoir à propos des API est qu'il en existe plusieurs types, en fait il y en a quatre types principaux d'API.

La première étape est l'API du système d'exploitation, puis l'API library, ensuite l'API distante et enfin l'API Web.

Plongeons plus en détail dans chacun d'eux et expliquons ce que c'est.

2.1 l'API du système d'exploitation.

Disons que vous écrivez une application.

Peu importe la plateforme que vous utilisez pour écrire cette application, cela peut être IT, Java, C++ ou autre.

Maintenant, naturellement, votre application s'exécute à l'intérieur d'un système d'exploitation.

Encore une fois, peu importe quel système d'exploitation, cela peut être Windows, Linux, OS X ou autre.

Dans de nombreux cas, l'application a besoin de certains services du système d'exploitation.

Par exemple, le système de fichiers.

Si l'application souhaite accéder à des fichiers dans le système d'exploitation, elle demande au système d'exploitation de lui donner accès aux fichiers, aux dispositifs réseau…

Si l'application doit accéder aux dispositifs réseau connectés au système d'exploitation, le système d'exploitation doit le permettre.

Eléments de l'interface utilisateur.

Si l'application, par exemple, veut afficher un bouton à l'écran, elle demande au système d'exploitation une représentation visuelle du bouton.

Comment fait-elle cela?

Comment l'application demande-t-elle tous ces types de services au système d'exploitation.

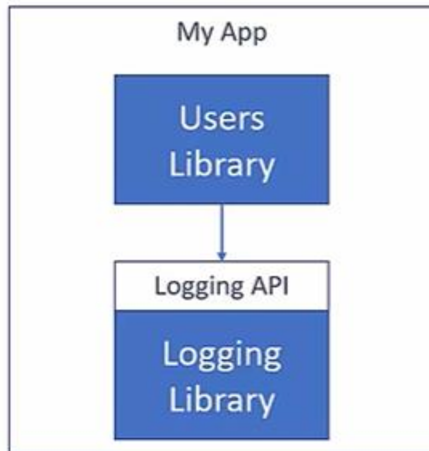
Eh bien, la réponse, comme vous l'avez probablement deviné, est en utilisant l'API du système d'exploitation.

Tous les systèmes d'exploitation exposent une sorte d'API que les applications s'exécutant dans le système d'exploitation peuvent accéder et utiliser.

Et l'une des API de système d'exploitation les plus célèbres est l'API Win32.

Elle est utilisée par toutes les applications qui s'exécutent sous Windows et elle fournit un grand nombre de services de système d'exploitation.

2.2 l'API Library.



Et ce que cela signifie, c'est que lorsque vous développez votre application, vous utilisez probablement une sorte de bibliothèques.

Le concept de bibliothèques est très courant dans toutes les plateformes de développement aujourd'hui.

Donc, vous développez votre bibliothèque, appelons-la bibliothèque utilisateur, et il y a une autre bibliothèque appelée Bibliothèque de journalisation.

Maintenant, la bibliothèque utilisateur veut utiliser la bibliothèque de journalisation.

Comment peut-elle faire cela?

La bibliothèque de journalisation simplement expose une API de journalisation, et ce faisant, elle permet à la bibliothèque de l'utilisateur d'accéder à cette API et d'utiliser les méthodes de la bibliothèque de journalisation telles que l'écriture dans le fichier journal ou le changement de la gravité de la journalisation ou tout autre fonctionnalité.

Mais sans une API, la bibliothèque de l'utilisateur ne pourra pas accéder à la bibliothèque de journalisation.

Notez à nouveau que ces bibliothèques sont toutes deux exécutées dans le même processus unique.

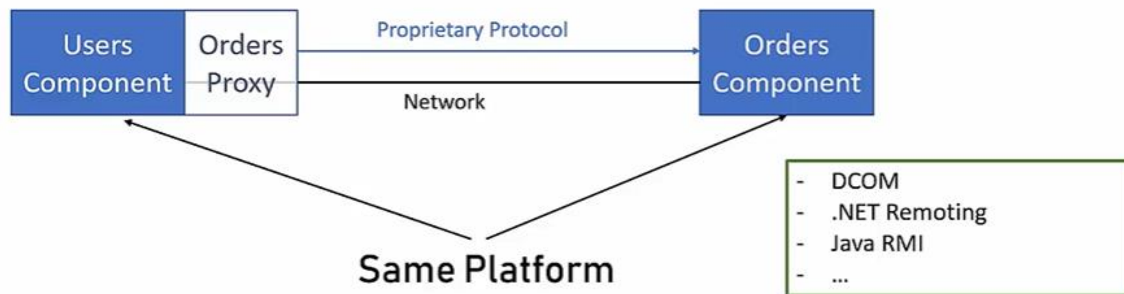
Elles ne communiquent pas via un réseau quelconque.

Comme mentionné précédemment, les bibliothèques de classes sont mises en œuvre dans toutes sortes de plates-formes de développement telles

que, comme nous le disons, dot, net, Java, Python, Node.js et même PHP et ainsi de suite.

2.3 l'API à distance ou remote API

Voici un exemple de remote API que nous allons expliciter :



Disons que ce sont un composant utilisateur et un composant commande.

Notez maintenant que ces deux composants sont distribués sur le réseau, ce qui signifie qu'ils ne sont pas dans le même processus et probablement pas sur le même serveur ou la même machine virtuelle ou PC.

Pour qu'ils puissent communiquer entre eux, ils doivent passer par le réseau.

Comment font-ils alors ?

Ainsi, le composant utilisateur a construit un proxy de commande.

Ce proxy est utilisé pour communiquer avec le composant de commande.

Ce proxy contient tout le fonctionnement interne de l'accès au composant de commande.

Il sait tout ce qu'il y a à savoir sur la façon d'accéder au composant, comme le protocole réseau, les méthodes d'adresse réseau, les signatures, l'authentification, l'autorisation, tout ce qui est important pour accéder à ce composant.

Ainsi, après la construction de ce proxy, le proxy peut accéder au composant commande à l'aide d'un protocole propriétaire et accéder à la méthode et à toutes les autres fonctionnalités exposées par ce composant.

Maintenant, ce qu'il est important de savoir sur cette méthode de communication à distance est qu'elle exige que les deux composants soient basés sur la même plateforme de développement.

Par exemple, il n'est pas possible dans cet exemple que le composant utilisateur soit développé en utilisant dot net et que le composant de commande soit développé en Python.

Cela ne fonctionnera tout simplement pas.

Des exemples de mises en œuvre d'API à distance sont DCOM, .NET Remoting, Java RMI et bien d'autres.

Il est également important de noter que l'API remote perd en popularité et que presque personne ne l'utilise de nos jours.

2.3 Web API

Et cela nous amène au dernier type d'API, qui n'est certainement pas le moindre, à savoir l'API web. Les applications web et les API web peuvent communiquer entre elles via Internet.

Voici comment cela fonctionne :

Nous avons une application publique publiée sur Internet. Cette application web veut utiliser une autre application web, disons une application météo, qui fournit des prévisions météorologiques pour les destinations qui nous intéressent.

Comment mon application web peut-elle communiquer avec l'application météo ?

Il y a donc quelques étapes pour cela.

Tout d'abord, l'application météo doit exposer (et encore une fois, notez l'utilisation du mot "exposer") une API web.

Ensuite, mon application web peut accéder à cette API web, car l'API web utilise toujours un protocole standard pour y accéder.

Et en utilisant ce protocole, mon application web peut accéder à cette API web et interroger l'application météo pour connaître les prévisions météorologiques.

Maintenant, l'une des choses les plus importantes à savoir sur l'API web est qu'elle peut être utilisée sur n'importe quelle plateforme, système d'exploitation ou langage.

En d'autres termes, peu importe comment mon application web est implémentée et comment l'application météo est implémentée.

Si vous vous souvenez, nous avons dit qu'avec une API remote, les deux composants logiciels doivent être développés en utilisant la même plateforme.

Ce n'est pas le cas avec l'API web.

Dans cet exemple, mon application web peut être développée en utilisant dotnet et l'application météo peut être développée en utilisant Java et cela fonctionnera très bien.

C'est juste l'un des facteurs qui rendent l'API web si populaire et si utilisée car elle est agnostique du langage et est uniquement basée sur le protocole http principalement.

Chapitre 3: Web APIs

1. Web APIs

Dans le chapiter précédent, nous avons discuté des différents types d'API et nous avons conclu que les API Web sont les plus populaires et les plus utiles.

Alors plongeons un peu plus en profondeur dans les API Web et expliquons ce que c'est exactement.

Rappelons-nous que la définition d'une API Web est que nous avons utilisé une API exposée par un composant Web permettant à d'autres composants d'interagir avec elle.

Il est important de se rappeler certaines caractéristiques communes des API Web.

Tout d'abord, l'API Web est agnostique de la plate-forme, ce qui signifie que ces composants, par exemple, peuvent facilement communiquer avec un composant Python en utilisant une API Web.

Et bien sûr, cela est également vrai pour d'autres plates-formes.

De plus, les API Web utilisent des protocoles standard, généralement sur HTTP.

Comme vous pouvez le voir dans ce diagramme, les API Web utilisent généralement une demande et une réponse, ce qui signifie que lorsque nous avons un composant appelant, dans ce cas, le composant qui appelle le composant Python, le composant dominant demandera ou enverra une demande basée sur le protocole HTTP au composant Python et recevra une réponse HTTP.

Cela fait partie du protocole HTTP, et presque tous les types d'API Web utiliseront ce paradigme de demande-réponse.

Maintenant, en discutant des types d'API Web, nous allons comprendre quelle est la différence ou quelles sont les principales différences entre eux.

Les API Web sont différenciées principalement par :

- Le format de demande. Cela peut être JSON ou XML ou d'autres formats.
- Le contenu de la demande.
- Le format de réponse est à nouveau JSON ou XML ou texte clair ou autre chose.
- Le contenu de la réponse.

Nous allons donc voir 4 types de Web API :

- SOAP
- REST
- GraphQL
- gRPC.

2. SOAP

Commençons par SOAP.

SOAP est un acronyme qui signifie Simple Object Access Protocol.

Il a été conçu en 1998 pour Microsoft.

Remarquez que ce n'est pas par Microsoft, mais POUR Microsoft.

C'est un protocole basé sur XML.

C'est un protocole qui est obsolete aujourd'hui mais SOAP a plus de 20 ans.

Et au siècle dernier, en fait, XML était le roi.

SOAP utilise une requête de type RPC (Remote Procedure Call), ce qui signifie qu'un client appelle un serveur et en appelant une méthode ou une procédure spécifique.

L'un des attributs importants de SOAP est qu'il est extensible, ce qui signifie que le service est un protocole de base qui est utilisé pour appeler une méthode et obtenir une réponse.

Et il peut être étendu également pour divers attributs tels que l'authentification, le routage, la transaction, la fédération, etc.

Voici un petit exemple de requête SOAP.

```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <NumberToWords xmlns="http://www.dataaccess.com/webservicesserver/">
      <ubiNum>500</ubiNum>
    </NumberToWords>
  </soap:Body>
</soap:Envelope>
```

Remarquez le XML et le format assez strict.

Nous avons d'abord l'enveloppe, qui contient des métadonnées sur la demande, puis nous avons le corps de la demande.

Et dans ce cas, le corps nous indique que nous appelons une méthode appelée NumberToWords et passons un paramètre UbiNum qui permettra de transformer un nombre en lettres.

Voici la réponse que vous recevrez également :

```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <m:NumberToWordsResponse xmlns:m="http://www.dataaccess.com/webservicesserver/">
            <m:NumberToWordsResult>five hundred </m:NumberToWordsResult>
        </m:NumberToWordsResponse>
    </soap:Body>
</soap:Envelope>
```

Maintenant, ce que vous devez savoir sur SOAP, c'est qu'il est assez obsolète et que vous ne devriez généralement pas l'utiliser sauf si vous y êtes vraiment, vraiment oblige dans le cas de systems legacy.

3. REST

Rest signifie "Representational State Transfer", ce qui est un acronyme assez complexe, il faut le reconnaître.

Mais fondamentalement, cela signifie que cette API Web est construite autour des entités et non des opérations.

Avec Rest, vous n'allez pas appeler des fonctions ou des méthodes spécifiques, mais vous allez déclarer des opérations autour des entités, telles que créer, lire, mettre à jour et supprimer.

Rest API a été conçue en 2000 par Roy Fielding et est construite autour de la norme HTTP, elle n'invente donc pas de nouvelles normes utilisant de nouveaux codes de réponse ou de nouvelles flèches, etc.

Mais elle réutilise la norme HTTP déjà utilisée par tous les navigateurs du monde.

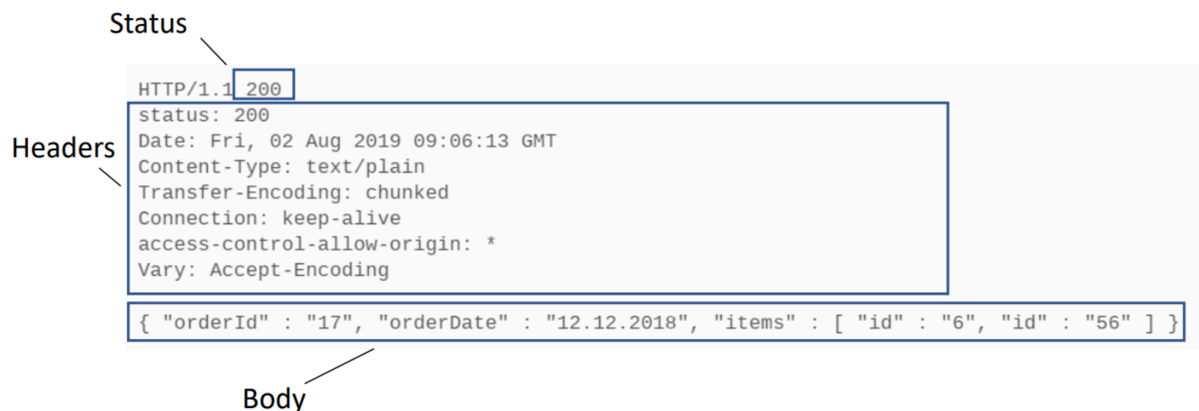
Maintenant, Rest est une API de type message, ce qui signifie qu'elle n'appelle pas de méthodes spécifiques, mais envoie des messages au serveur et fait confiance au serveur pour prendre soin de ces messages.

Elle est extrêmement simple à mettre en œuvre et bénéficie d'un large soutien de toutes les plates-formes de développement disponibles, ce qui en fait l'API Web la plus populaire aujourd'hui.

Maintenant, Rest API utilise une URL pour identifier une entité par exemple.

Ainsi, par exemple, cette URL "/API/V1/order/17/items" identifie en réalité les éléments de la commande numéro 17.

Et si vous voulez récupérer ces éléments, c'est généralement l'URL que vous enverrez au serveur et la réponse de Rest API est généralement un JSON.



Donc, si vous regardez une réponse typique de l'API, vous pouvez trouver la réponse JSON ici, dans le corps de la réponse.

Et si vous examinez de plus près une réponse typique de l'API Rest, nous pouvons voir qu'il y a trois grandes parties qui nous intéressent.

La première est le statut, qui indique si la demande a réussi ou non, et l'état 200 indique que tout va bien et que la demande a réussi.

Ensuite, il y a les en-têtes contenant divers métadonnées sur la réponse.

Et enfin, comme nous l'avons vu, il y a le corps contenant le JSON avec les données réelles qui ont été renvoyées au client.

Voilà pour Rest API.

4. GraphQL

GraphQL a été développé en interne en 2012 par Facebook et publié au public en 2015.

Le plus grand avantage de GraphQL est qu'il permet une requête très flexible, une mise à jour et une souscription aux modifications de données.

En contraste avec REST, qui vous permet de récupérer toute l'entité après avoir effectué une requête, GraphQL vous permet de spécifier exactement les champs que vous voulez récupérer, et en plus, vous pouvez spécifier quels paramètres utiliser dans la requête elle-même.

Vous bénéficiez donc de fonctionnalités de requête et d'opération très, très flexibles.

En outre, vous disposez de fonctionnalités de souscription, ce qui signifie que vous pouvez recevoir des notifications de mises à jour pour les entités que vous avez récupérées et fusionner ces mises à jour dans les entités.

GraphQL, comme REST, est basé sur JSON en entrée et en sortie.

Voici un exemple de requête GraphQL.

```
query {
  posts {
    id
    title
    body
    published
    author {
      name

    }
  }
}
```

Comme vous pouvez le voir, cela commence par "query". Ensuite, il nous dit quel type d'entité nous voulons interroger, dans ce cas "posts". Ensuite, nous pouvons spécifier quels champs nous voulons récupérer pour les posts. Dans ce cas, nous voulons récupérer l'ID, le titre, le corps, la date de publication et l'auteur. Pour l'auteur, nous devons préciser quel champ de l'auteur nous voulons récupérer. Dans ce cas, nous voulons récupérer le nom de l'auteur.

Et la réponse ressemble à cela.

```
{
  "data": {
    "posts": [
      {
        "id": "1",
        "title": "Why and how?",
        "body": "this is a post",
        "published": true,
        "author": {
          "name": "Memi"
        }
      },
      {
        "id": "2",
        "title": "How to become wealthy?",
        "body": "Work hard.",
        "published": false,
        "author": {
          "name": "Memi"
        }
      },
```

Et vous pouvez voir que nous avons exactement ce que nous avons demandé. Nous obtenons uniquement les champs que nous avons demandés, y compris le nom de l'auteur, qui est un champ stocké dans une table différente dans ce cas.

Ce que vous devez savoir à propos de GraphQL, c'est qu'il est très flexible, comme nous venons de le démontrer, mais qu'il nécessite un effort de développement initial. Contrairement à REST, qui est extrêmement facile à implémenter, GraphQL n'est pas si facile et nécessite un certain développement pour être efficace. De plus, GraphQL n'est pas aussi optimisé que REST en termes de performances. Vous devez donc examiner de très près les metrics de performances lors de l'utilisation de GraphQL.

GraphQL gagne du terrain dans le monde entier. Par exemple, GitHub a annoncé qu'il migrait toutes ses API pour exposer GraphQL au lieu de REST, car ils ont constaté que la plupart des utilisateurs voulaient récupérer uniquement des champs spécifiques et presque personne ne voulait la totalité des champs exposés par l'API REST de GitHub.

5. gRPC

Et pour finir gRPC mais nous allons aller plus en details dans les chapitres suivants.

Chapitre 4: gRPC Basics

1. Les problèmes des API REST

L’API Rest doit être une des technos les plus répandues d’internet.

Pourtant elle comporte plein de désavantages.

Arguments en vrac :

- Nécessité de valider soi-même les données envoyées par l'app cliente
- Utilisation intensive de la serialisation et déserialisation de JSON (chère en puissance de calcul)
- Nécessité de maintenir des grosses docs pour permettre l'utilisation et la consommation
- Gestion des streams pas facile
- Une connexion TCP par requête ?????
- Besoin de re-décrire toutes les structures de communication pour chaque nouveau langage communiquant avec l'API
- On en code H24 depuis 15 ans ça commence à bien faire

Pour moi ça fait partie de ces technos qui sont parties pour devenir obsolètes, mais qu'on continue d'utiliser parce que c'est "facile" et que "tout le monde connaît".

Si nous rentrons dans le détail, lors de la publication, il a été découvert qu'il y avait trois problèmes majeurs avec REST, à savoir la performance, la demande de réponse uniquement et sa syntaxe limitée.

Commençons par la performance.

Si vous utilisez REST, vous savez probablement que REST utilise des protocoles et des messages basés sur du texte.

Il utilise HTTP 1.1, qui est essentiellement un protocole basé sur du texte, et il transmet des informations en utilisant le protocole JSON, qui est également un format de messagerie basé sur du texte.

Par exemple, si nous examinons une demande ou une réponse typique envoyée par REST, vous pouvez voir le corps, qui est en effet basé sur JSON et qui est un texte clair rendu lisible pour les humains.

Maintenant, il y a deux problèmes avec l'échange de données basé sur du texte.

Le premier est qu'il force le protocole à utiliser de grands paquets car le protocole ou la messagerie basé sur du texte est, par définition, plus grand que le format binaire.

De plus, ce texte doit être transmis par le logiciel, ce qui est une autre opération que le logiciel doit effectuer afin de pouvoir traiter le message, ce qui affecte les performances de l'API.

C'est donc un problème de performance de l'API REST.

Le suivant est la demande de réponse uniquement, et comme vous le savez probablement, REST ne prend en charge que le modèle de demande-réponse, ce qui signifie que nous avons un client et un service, et que ce service expose une API REST, puis le client envoie une demande et attend une réponse avec un certain type de corps ou un code de réponse indiquant la réussite de la demande.

Cependant, les applications Web modernes nécessitent davantage de types de communication client-serveur.

Par exemple, les notifications push deviennent extrêmement utiles et extrêmement courantes.

Et pour cela, l'utilisation classique de la demande-réponse n'est tout simplement pas pertinente.

Nous voulons que le service soit en mesure d'envoyer simplement une notification au client et nous ne voulons pas que le client soit obligé d'envoyer une demande au serveur pour que le serveur ne puisse répondre qu'à la demande mais ne peut pas envoyer de notifications indépendantes au client.

Maintenant, les notifications push peuvent être mises en œuvre dès aujourd'hui en utilisant le polling, par exemple, ou les websockets, mais ceux-ci ne font pas partie du protocole et ne font pas partie de l'API REST, et en plus, ils sont assez complexes à mettre en œuvre.

Comme vous pouvez le voir, le fait que REST ne prenne en charge que le modèle de demande-réponse est définitivement une limitation et un problème dans le protocole.

Le problème suivant est sa syntaxe limitée.

Rest a été conçu pour exécuter des opérations CRUD sur des entités.

Maintenant, ce qui est assez clair, ce sont les quatre opérations que nous effectuons généralement sur des entités et ceux-ci sont créés en utilisant

le verbe post, lus en utilisant le verbe get, mis à jour en utilisant le verbe put et supprimés en utilisant le verbe delete, par exemple.

- `GET /api/v1/orders/17`
- `DELETE /api/v1/employee/84`
- `POST /api/v1/telemetry`
- `GET /api/v1/flights?from=LHR&to=JFK&date=2022-08-05`

Ce sont des opérations typiques dans rest.

Maintenant, étant donné que Rest est basé sur des entités, cela signifie qu'il n'est pas adapté pour exécuter des actions.

Par exemple, comment créer une URL qui doit démarrer un nouveau processus ou effectuer une connexion ou désactiver un périphérique ?

Bien sûr, il existe des solutions pour tous ces problèmes, mais ce sont en fait des contournements et ne sont pas une implémentation classique de Rest API.

Voilà donc les problèmes avec Rest API et ces problèmes ont jeté les bases de la création de gRPC.

2. Histoire de gRPC

gRPC a été initialement créé par Google, qui utilisait une infrastructure RPC polyvalente appelée Stubby pour connecter un grand nombre de microservices s'exécutant à l'intérieur et à travers ses centres de données à partir de 2001.

En mars 2015, Google a décidé de construire la prochaine version de Stubby et de la rendre open source. Le résultat était gRPC, qui est maintenant utilisé dans de nombreuses organisations en dehors de Google pour alimenter des cas d'utilisation allant des microservices à du calcul (mobile, web et Internet des objets).

Il utilise HTTP/2 pour le transport, Protocol Buffers comme langage de description d'interface, et fournit des fonctionnalités telles que l'authentification, la diffusion en continu bidirectionnelle et la gestion de flux, les liaisons bloquantes ou non bloquantes, ainsi que l'annulation et les délais d'attente(deadlines).

Il génère des liaisons client et serveur multiplateformes pour de nombreux langages. Les scénarios d'utilisation les plus courants comprennent la connexion de services dans une architecture de style

microservices, ou la connexion de clients de périphériques mobiles à des services en backend.

L'utilisation complexe de HTTP/2 par gRPC rend impossible la mise en œuvre d'un client gRPC dans le navigateur, nécessitant plutôt un proxy.

Plusieurs organisations différentes ont adopté gRPC, telles que Uber,Square, Netflix, IBM, CoreOS, Docker, CockroachDB, Cisco, Juniper Networks, Spotify, Zalando, Dropbox, et Google en tant que développeur d'origine.

Le projet open source u-bmc utilise gRPC pour remplacer l'interface de gestion de plate-forme intelligente (IPMI). Le 8 janvier 2019, Dropbox a annoncé que la prochaine version de "Courier", leur framework RPC au cœur de leur architecture orientée services (SOA), serait migrée pour être basée sur gRPC, principalement parce qu'elle était bien alignée avec leurs frameworks RPC personnalisés existants.

3. Les base de gRPC

Tout d’abord gRPC c’est tout cela :

Web API

Based on HTTP/2

RPC Style

Multiple Communication Styles

Uses Protobuf as Payload

Donc comme nous l’avons vu, gRPC est bien un type de Web Api.

Comme mentionné précédemment, gRPC est basé sur http/2.

Et pour vraiment comprendre ceci, nous devons d'abord nous souvenir que le protocole HTTP 1.1, qui est un protocole standard à ce jour, a été publié essentiellement dans le millénaire précédent en 1997.

Internet était très différent de ce qu'il est maintenant, et de nombreux problèmes de performance étaient causés par la grande quantité de données et le modèle de request-response, qui n'est tout simplement pas adapté au trafic actuel et au modèle d'utilisation de l'Internet moderne.

Maintenant, le protocole http/2 publié en 2015 vise à résoudre ces problèmes.

Il reprend les possibilités de http/1.1 mais en ajoute de nouvelles.

Le protocole http/2 permet la diffusion bi latérale, en plus de la request-response.

Donc, il ne remplace pas le modèle de request-response, mais ajoute d'autres types de communication.

Et ces types de communication ouvrent de nouvelles possibilités telles que les notifications push, que nous avons discutées précédemment, et bien plus encore.

Notez cependant que, en raison de la complexité de la gestion du protocole HTTP/2 dans gRPC, il ne peut pas être utilisé à partir des navigateurs, donc vous ne pouvez pas envoyer de requêtes gRPC et recevoir des réponses gRPC directement depuis un navigateur.

Cela signifie qu'il est principalement utilisé pour les applications mobiles natives.

Cependant, il existe une solution de contournement pour utiliser gRPC dans le navigateur en arrière-plan, que nous aborderons plus tard dans ce cours et que nous mettrons en œuvre.

4. RPC

Et la prochaine caractéristique de gRPC est qu'il fonctionne selon un style RPC.

Et comprenons exactement ce qu'est RPC.

Maintenant, si vous regardez le nom gRPC, vous pouvez voir qu'il a deux parties G et RPC.

G est pour Google, qui est le créateur de gRPC, et la deuxième partie est RPC.

Alors qu'est-ce que RPC exactement?

Eh bien, RPC signifie Remote Procedure Call, ce qui signifie appeler une méthode sur le serveur à partir du client.

Ainsi, si vous avez un client et un serveur et que le serveur a une méthode nommée obtenir la météo, alors le client peut directement appeler cette méthode spécifique du client au serveur et bien sûr obtenir une réponse.

Maintenant, pour comprendre pourquoi c'est si important et pourquoi c'est si différent de REST, alors nous allons faire un bref rappel :

- REST fonctionne avec des entités, pas des méthodes.
  Par exemple, si vous avez cette même topologie avec client-serveur et la méthode obtenir la météo alors remplie avec REST, nous n'appelons pas la méthode obtenir la météo.
  Au lieu de cela, nous envoyons une URL ou une requête.
- En outre avec REST, nous n'avons aucune idée du nom de la méthode sur le serveur.
  Nous n'avons aucune idée et cela ne nous importe pas. Pour nous, le nom de la méthode est redondant et nous ne sommes pas intéressés par cela.
- gRPC appelle la méthode réelle, donc avec gRPC nous devons connaître le nom de la méthode sur le serveur car c'est la méthode que nous allons appeler.

5. Les types de communication

Et la caractéristique suivante et peut-être la plus importante de gRPC est ses styles de communication et le support de RPC pour les styles de communication, le client / serveur unaire, le streaming client, le streaming serveur et bidirectionnel.

Commençons par passer en revue ces styles de communication et comprenons exactement ce qu'ils sont.

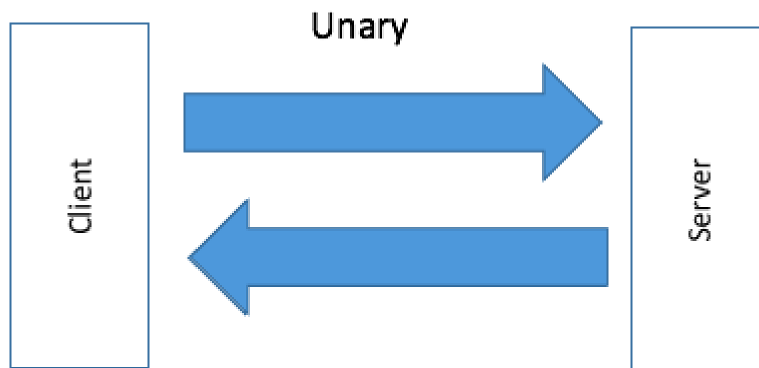
Commençons par le mode client / serveur unaire.

Donc, client / serveur unaire ou unique est essentiellement un modèle de request / response implémenté par des API Web similaires.

Nous avons donc le client et le serveur et le client envoie une demande au serveur.

Et n'oubliez pas qu'avec gRPC, la demande appelle en fait la méthode, la méthode réelle sur le serveur et le serveur reçoit la demande et envoie une réponse.

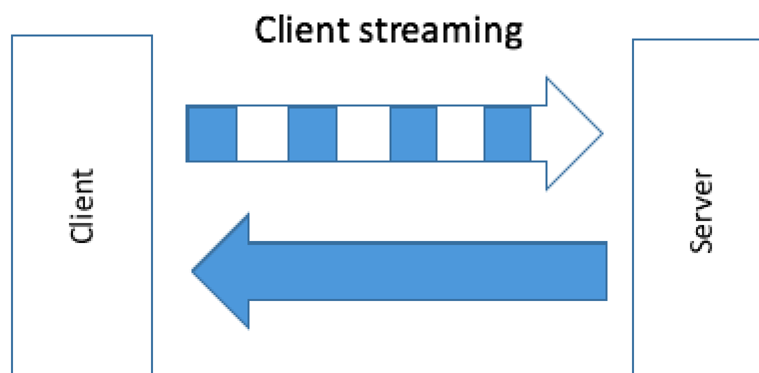
C'est donc un style de communication client / serveur unaire/unique avec lequel vous êtes probablement déjà familier.



Ensuite, le streaming client.

Avec le streaming client, le client ouvre une connexion avec le serveur et envoie ensuite des messages continus au serveur, ce qui est idéal pour les chats, l'envoi de télémétrie, etc.

Donc, cela ressemblera à ceci.



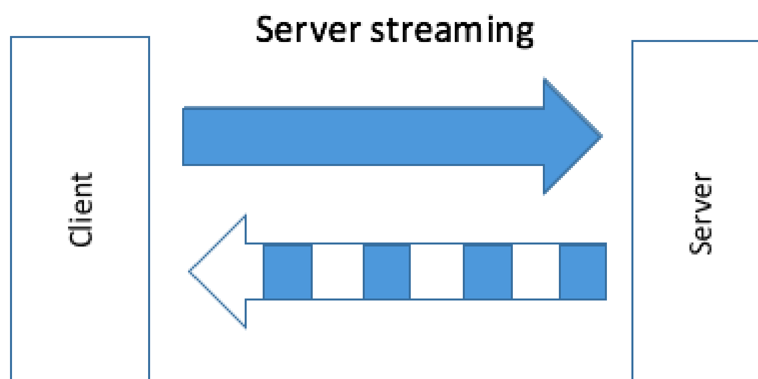
Nous avons à nouveau le client et le serveur, et cette fois nous avons une méthode nommée StoreTelemetry() sur le serveur et le client ouvre un canal ou une connexion vers le serveur et commence à appeler la méthode StoreTelemetry().

Il l'appelle d'abord de cette manière, puis il l'appelle avec un autre paramètre.

Et tout cela est basé sur la même connexion.

Le client envoie donc des données en continu sur la connexion vers le serveur.

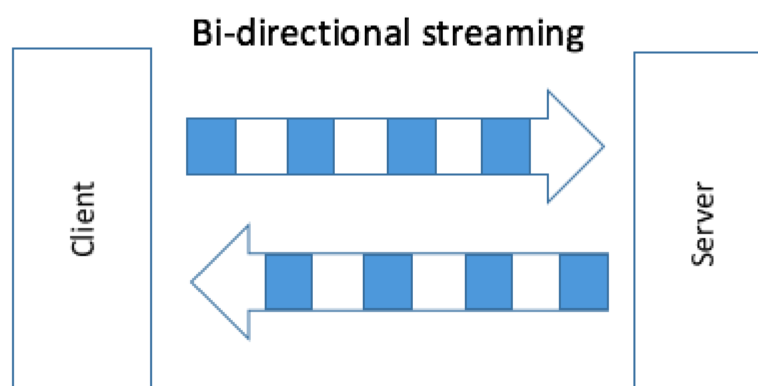
Le streaming serveur signifie que le client ouvre une connexion avec le serveur comme précédemment, et maintenant le serveur envoie des messages continus à l'aide de cette connexion, ce qui est idéal pour les notifications, le chat, etc.



Par exemple, il pourrait envoyer une notification de commentaire, puis une autre notification de commentaire, etc.

Tout cela est basé sur la même connexion qui reste ouverte pendant que le client et le serveur sont en communication.

La prochaine est bidirectionnelle.



Et avec la bidirectionnelle, comme vous pouvez probablement le deviner, le client ouvre une connexion avec le serveur et les deux le client et le serveur envoient des messages continus à l'aide de cette connexion.

C'est donc idéal pour le chat, les données en temps réel, etc. Donc, encore une fois, nous avons le client et le serveur et le client ouvre un canal vers le serveur et commence à envoyer des demandes.

Par exemple, quelle est la température à New York City et à Londres?

Ensuite, il commence à recevoir les réponses, qui sont dans ce cas 38 degrés et 25 degrés.

Ensuite, le client envoie plus de messages demandant la météo à Rome et reçoit une réponse différente, etc.

C'est donc fondamentalement ainsi que fonctionne le streaming bidirectionnel.

6. ProtoBuf

La prochaine caractéristique dont nous allons parler est les protocoles et ce que cela implique exactement.

Un protocol est essentiellement le format de données utilisé par gRPC, par opposition, par exemple, au format JSON utilisé par REST.

Maintenant, théoriquement, gRPC peut utiliser d'autres formats, mais en réalité, personne ne le fait et tout le monde utilise les protocoles.

Le format de données présenté ici est à nouveau un format binaire et non un format basé sur le texte, et il déclare le format des messages dans un fichier proto, avec lequel vous allez travailler beaucoup dans ce cours.

Le protocole génère ensuite le code client dans les langages supportés, et ce code client va sérialiser et désérialiser les messages pour que le code puisse travailler avec eux.

Maintenant, voici un fichier de protocole typique et assez petit qui déclare des messages de protocole.

```
1
2    syntax = "proto3";
3
4  service Converter {
5      rpc convertToGltf(convertReq) returns (convertResp) {}
6    }
7
8  message convertReq {
9      string type = 1;
10     bool isBin = 2;
11     bytes file = 3;
12     bool needDraco = 4;
13     bool noZip = 5;
14   }
15 message convertResp {
16     bytes file = 1;
17   }
```

Et regardez surtout la partie inférieure de ce fichier.

Nous pouvons voir ici un message nommé "convertReq" avec les champs.

Ne vous inquiétez pas si vous ne comprenez pas vraiment ce que vous voyez maintenant, nous allons couvrir le protocole de manière approfondie dans la section suivante et nous allons beaucoup travailler avec.

Je vous promets qu'à la fin de ce cours, vous serez parfaitement à l'aise avec les protocoles.

7. Notions avancées

Maintenant, je souhaite discuter de quelques sujets avancés , dont vous devriez être conscient, et le premier est les canaux (channels). Un canal est essentiellement une définition de la connexion entre le client et le serveur, et il spécifie l'hôte et le port auquel le client se connectera.

Maintenant, un canal a un état, il peut être connecté, inactif, etc., et cela dépend du langage utilisé. Vous pouvez interroger le canal pour connaître son état. Par exemple, en .NET, vous pouvez demander la propriété d'état du canal pour savoir quel est l'état actuel de chaque canal.

Le suivant est le timeout ou deadline qui spécifient essentiellement combien de temps le client attendra une réponse. Le délai d'attente déclare essentiellement la durée pendant laquelle le client attendra, tandis que la deadline est un point fixe dans le temps jusqu'auquel le client attendra. Lorsque ces deux délais sont dépassés, l'appel est interrompu et une erreur est renvoyée.

La spécification du délai d'attente ou de la deadline dépend du langage utilisé, et nous allons beaucoup travailler avec cela dans la section des sujets avancés plus tard.

Le dernier sujet que je souhaite aborder est les métadonnées (metadata). Les métadonnées sont essentiellement des informations sur un appel gRPC, par exemple les détails d'authentification de l'appel. Nous allons utiliser les détails d'authentification dans les métadonnées plus tard dans ce cours pour authentifier les appels. Les métadonnées sont renvoyées et stockées sous la forme d'une liste de paires clé-valeur attachées à la request, et cela dépend du langage utilisé. Par exemple, en .NET, nous allons appeler la méthode ".GetAll()" pour obtenir la liste de toutes les métadonnées de l'appelant.

Voilà pour les bases de gRPC, j'espère que vous comprenez déjà les idées de base derrière cette API Web.

Chapitre 5 : Préparation de l’environnement

1. Introduction

Dans cette section, nous allons préparer notre environnement ou notre ordinateur pour exécuter les applications sur lesquelles nous allons travailler dans ce cours.

Maintenant, pour construire l'application, nous avons besoin d'installer certains logiciels sur notre ordinateur.

Et ce que nous allons installer en premier, c'est le SDK .NET, car l'application est basée sur la plateforme DOTNET.

Enfin, il y a Visual Studio Community Edition.

2. Installer le .NET SDK et Visual Studio

Pour l’installation du SDK .NET , rendez vous sur https://dotnet.microsoft.com/en-us/download et téléchargez la dernière version du SDK (7.0 pour nous) pour votre système d’exploitation.

Lancez ensuite l’exécutable une fois téléchargé et vous devriez avoir un écran ressemblant à ceci :

Download .NET 7.0

Not sure what to download? See recommended downloads for the latest version of .NET.

7.0.2

Release notes **Latest release date** January 10, 2023

Build apps - SDK

SDK 7.0.102

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 | Arm32 Alpine | Arm64 | Arm64 Alpine | x64 | x64 Alpine
macOS	Arm64 | x64	Arm64 | x64
Windows	Arm64 | x64 | x86 | winget instructions	Arm64 | x64 | x86
All	dotnet-install scripts	

Visual Studio support
Visual Studio 2022 (v17.4)
Visual Studio 2022 for Mac (v17.4)

Included in
Visual Studio 17.4.4

Included runtimes
.NET Runtime 7.0.2
ASP.NET Core Runtime 7.0.2
.NET Desktop Runtime 7.0.2

Language support
C# 11.0
F# 7.0
Visual Basic 16.9

Run apps - Runtime

ASP.NET Core Runtime 7.0.2

The ASP.NET Core Runtime enables you to run existing web/server applications. **On Windows, we recommend installing the Hosting Bundle, which includes the .NET Runtime and IIS support.**

IIS runtime support (ASP.NET Core Module v2)
17.0.22341.2

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 | Arm32 Alpine | Arm64 | Arm64 Alpine | x64 | x64 Alpine
macOS		Arm64 | x64
Windows	Hosting Bundle | x64 | x86 | winget instructions	Arm64 | x64 | x86

.NET Desktop Runtime 7.0.2

The .NET Desktop Runtime enables you to run existing Windows desktop applications. **This release includes the .NET Runtime; you don't need to install it separately.**

OS	Installers	Binaries
Windows	Arm64 | x64 | x86 | winget instructions	

.NET Runtime 7.0.2

Suivez les instructions, il ne doit pas y avoir de pièges.

Une fois installé, vous pouvez vérifier que tout est bien installé en ouvrant une invite de commandes et taper : dotnet –version

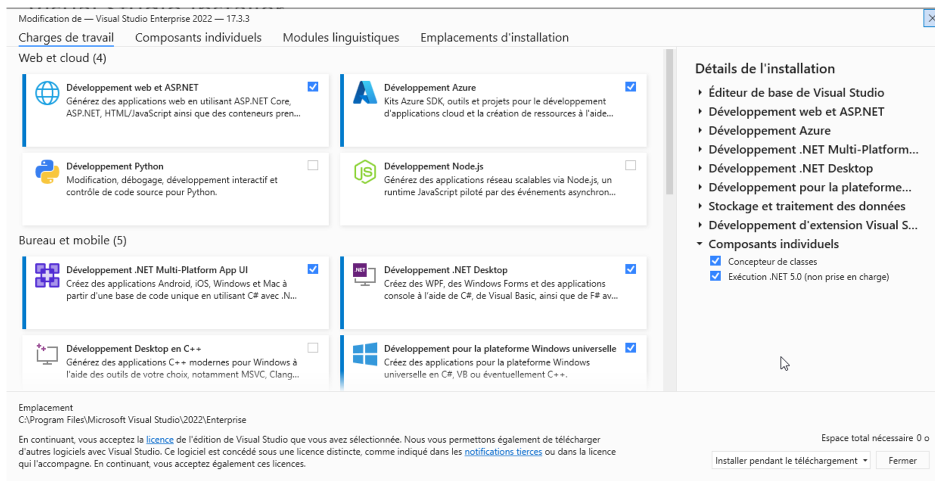
Vous devriez obtenir la version que vous venez d’installer :

```
Invite de commandes
Microsoft Windows [version 10.0.22621.1105]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\alexandre.castro>dotnet --version
7.0.102

C:\Users\alexandre.castro>
```

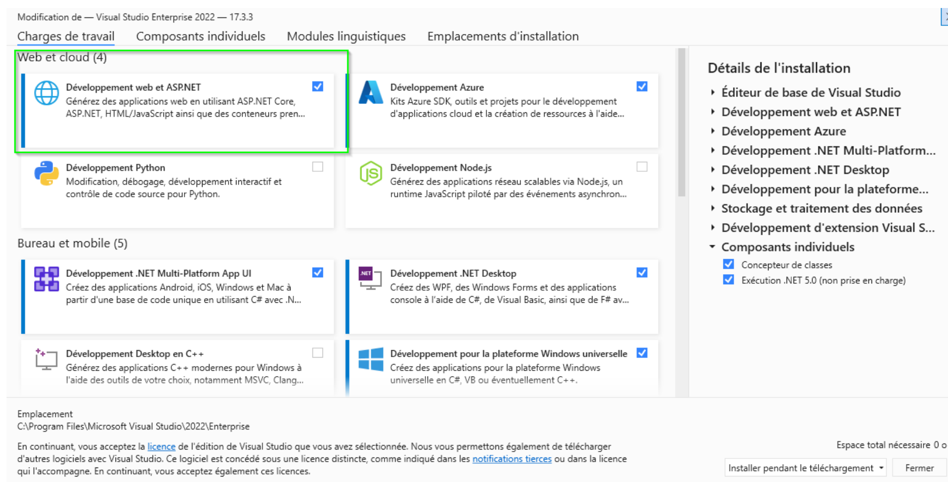
Personnellement je travaille à 90% sous Visual Studio et 10% sous Rider.

Je vous conseille donc de télécharger Visual Studio 2022 et au minimum la version community qui est gratuite pour l’apprentissage ici : https://visualstudio.microsoft.com/fr/vs/

Une fois l'exécutable lancé, vous devriez avoir un écran qui ressemble à celui-ci :



Dans le cadre du développement d'API REST, il faut que vous cochiez au minimum les cases suivantes :



Une fois les options choisies, le programme d'installation va télécharger et installer les modules demandés.

Chapitre 6 : Notre application

2. Chat App

Nous allons donc créer une application de chat puisque ce type d'applications se prête bien à la technologie de gRPC.

Les utilisateurs peuvent discuter entre eux, comme dans des salles de chat classiques.

Cependant, en plus, un bot spécial va envoyer des flashs de news dans les salles de chat, permettant aux utilisateurs de rester au courant des événements importants dans le monde et de veiller à ce qu'il n'y ait pas de comportements inappropriés dans nos salles et dans le flux d'actualités.

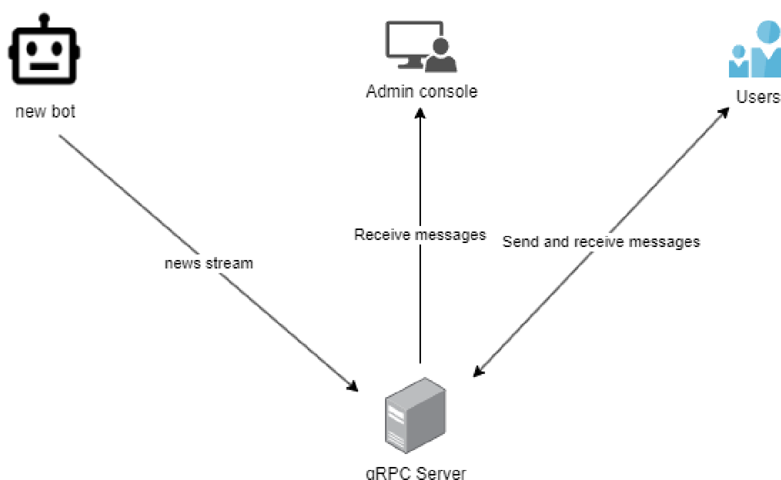
Ensuite, une console de surveillance (admin console) affichera les événements envoyés, permettant aux administrateurs de superviser le bot et l'activité des utilisateurs.

La première(la chat room), qui est plus classique, est une salle de chat où les utilisateurs peuvent discuter entre eux et s'inscrire pour la salle de chat qui les intéresse.

En outre, nous avons le bot d'actualités qui envoie des flashs d'actualités dans les salles et bien sûr la console d'administration permettant aux administrateurs de voir ce qui se passe dans la salle.

Ce sont donc les principales composantes de la salle.

Et maintenant, examinons l'architecture du système que nous allons construire.



Tous ces composants seront construits à l’aide du framework .NET Core 7.0.

So l’on regarde attentivement les flèches nous voyons donc du streaming client(news) , du streaming serveur (à destination de l’admin console) , du streaming bidirectionnel pour les users et enfin nous aurons des appels unaires pour se connecter à des chat rooms définies en appellant une méthode prenant en paramètre l’identifiant de la salle à laquelle l’utilisateur veut se connecter.

Chapitre 7 : Protobuf

1. Introduction

Le protocole est essentiellement une abréviation de "protocol buffer", et c'est le format de sérialisation utilisé par gRPC pour sérialiser les messages entre le client et le serveur.

En réalité, le protocole n'est pas strictement lié à gRPC et peut être utilisé comme un format de sérialisation général dans de nombreux autres scénarios, pas nécessairement liés à gRPC.

Cependant, gRPC utilise presque toujours le protocole, bien que théoriquement il n'ait pas besoin de le faire.

Premièrement, et peut-être la chose la plus importante à savoir sur le protocole, c'est qu'il s'agit d'un format binaire par opposition à JSON et XML qui sont basés sur du texte, et cela signifie que le protocole n'est pas lisible par les humains.

Donc si vous regardez un fichier généré par le compilateur de protocole, vous ne pourrez pas le lire.

Le fait que ce soit un format binaire le rend extrêmement efficace et rapide, car bien sûr les ordinateurs fonctionnent mieux avec un format binaire et non avec un format basé sur du texte.

Maintenant, le protocole est compatible avec la plupart des plateformes de développement modernes, et la façon dont il fonctionne est qu'il génère des classes client dans la plateforme respective pour accéder aux messages du protocole.

Et l'une des choses les plus agréables à propos du protocole, c'est que c'est un format inter-langage, ce qui signifie que si vous générez des messages en, disons, dot net, alors votre code Java pourra lire ce message.

Maintenant, quels sont les langues supportées par le protocole?

Eh bien, voici la liste la plus à jour des langues supportées par le protocole.

Nous avons ici C++, C-sharp, Java, Kotlin, Objective-C, PHP, Python et Ruby, et ce ne sont que les langues officiellement supportées.

Il y a plus de langues supportées, mais pas officiellement.

Donc vous pouvez utiliser le protocole dans d'autres langues, mais vous n'aurez pas de support.

2. Flux d’utilisation de Protobuf

Si vous souhaitez utiliser Protobuf, il y a une séquence ou flux d’utilisation à respecter :

Define messages and services

Generate code

Use the generated code

Commençons par la définition des messages et des services.

Il s'agit d'un fichier avec une extension .proto.

Le proto est en fait un simple fichier texte, facile à créer et très lisible.

Vous pouvez avoir plus d'un fichier, et vous pouvez également définir plusieurs messages et services dans le même fichier.

Voici un exemple de message :

```
syntax = "proto3";

service Converter {
  rpc convertToGltf(convertReq) returns (convertResp) {}
}

message convertReq {
  string type = 1;
  bool isBin = 2;
  bytes file = 3;
  bool needDraco = 4;
  bool noZip = 5;
}
message convertResp {
  bytes file = 1;
}
```

Ensuite avec ce fichier proto, vous allez pouvoir générer du code : cela se fait à l'aide du compilateur de protocole, et ce compilateur génère du code client dans le langage approprié. (pour nous ce sera le compilateur proto dotnet)

Les chiffres à coté des champs représentent l'ordre des champs tout simplement.

Dans notre exemple, nous avons un accès à un service rpc , un message d'entrée et un message de réponse.

À la suite de cela, nous pourrons accéder au code généré.

Voici un bout de ce qui va être généré :

```
static readonly string __ServiceName = "Converter";

[global::System.CodeDom.Compiler.GeneratedCode("grpc_csharp_plugin", null)]
s       __ lper_SerializeMessage(global::Google.Protobuf.IMessage message, grpc::SerializationContext context)
{
  #if !GRPC_DISABLE_PROTOBUF_BUFFER_SERIALIZATION
  if (message is global::Google.Protobuf.IBufferMessage)
  {
    context.SetPayloadLength(message.CalculateSize());
    global::Google.Protobuf.MessageExtensions.WriteTo(message, context.GetBufferWriter());
    context.Complete();
    return;
  }
  #endif
  context.Complete(global::Google.Protobuf.MessageExtensions.ToByteArray(message));
}
```

Et comment nous l'utiliserions :

```
using var channel = GrpcChannel.ForAddress("http://127.0.0.1:8999",new GrpcChannelOptions
{
    MaxReceiveMessageSize = 50 * 1024 * 1024, // 5 MB
    MaxSendMessageSize = 50 * 1024 * 1024 // 2 MB
});
var client = new Converter.ConverterClient(channel);
var fileBytes = File.ReadAllBytes(filePath);
var req = new convertReq();
req.Type = "stp";
req.File = ByteString.CopyFrom(fileBytes);
req.IsBin = true;
req.NeedDraco = true;
req.NoZip = true;
var resp = await client.convertToGltfAsync(req);
return resp.File.ToByteArray();
```

C’est un exemple très sommaire mais nous verrons cela en détails dans les chapitres suivants.

3. Syntaxe basique des Messages

Le premier élément dont nous allons discuter est l'élément message, dont nous avons vu un exemple auparavant, et l'élément message définit la structure du message envoyé entre les composants gRPC, qui sont le client et le serveur et le message est défini en utilisant un élément message, comme nous venons de le dire.

Maintenant, en tant que bonne pratique, un message a un nom formaté avec un modèle camel case, ce qui signifie que la première lettre est une majuscule, et pour chaque mot qui fait partie du nom, nous allons également capitaliser la première lettre.

Maintenant, le message contient les champs typés qui composent ensemble le contenu du message.

Un détail technique important est que vous devez toujours spécifier la version de la syntaxe, et nous allons travailler avec la version 3 de proto, qui est, au moment de l’écriture, la version la plus à jour des protocoles.

Donc voici le modèle du fichier de protocole que nous allons utiliser.

```
syntax = "proto3";

message convertReq {
  string type = 1;
  bool isBin = 2;
  bytes file = 3;
  bool needDraco = 4;
  bool noZip = 5;
}
```

Comme vous pouvez le voir, la première ligne définit la version de la syntaxe, qui est, comme nous l'avons dit, la version 3. Donc la ligne se présente essentiellement comme suit : syntaxe égale à proto3, puis nous définissons le message.

Dans ce cas, le message est nommé ConvertReq.

Vous êtes bien sûr libre d'utiliser le nom de message que vous voulez.

Ensuite, après avoir défini le message, il est temps de définir les champs de message, qui contiendront les valeurs réelles du message.

Le message contient des champs contenant les données, les valeurs du message.

Et chaque champ a au moins un nom, un type de données.

Et c'est une partie importante, un numéro d'identification unique.

Rappelez-vous, un protocole est essentiellement un format binaire, il doit savoir quel champ a quel numéro.

Il est donc extrêmement important que chaque champ dans le message ait un numéro d'identification unique.

Maintenant, quels sont les types de données que nous pouvons utiliser lors de la définition des champs de message?

Vous pouvez les trouver ici : https://protobuf.dev/programming-guides/proto3/#scalar ainsi que leur correspondance dans le langage que vous utilisez.

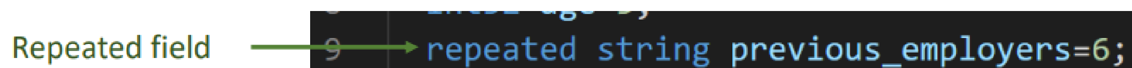
Voici la correspondance avec les types scalaires C# :

.proto Type	C# Type
double	double
float	float
int32	int
int64	long
uint32	uint
uint64	ulong
sint32	int
sint64	long
fixed32	uint
fixed64	ulong
sfixed32	int
sfixed64	long
bool	bool
string	string
bytes	ByteString

Maintenant, une chose que vous pourriez remarquer qui n'est pas dans la liste est la date, et c'est parce qu'il n'y a pas de type de date intégré, mais comme nous le verrons plus tard, ce type peut être importé.

La bonne pratique pour le nommage des champs est que le nom du champ ne contient que des lettres minuscules, et lorsque le nom du champ contient plus d'un mot, alors nous allons le concaténer avec un trait de soulignement.

Il existe aussi ce que l'on appelle des « repeated fields » qui sont par essence des listes comme par exemple :



Ici nous aurons donc une liste de chaine de caractères.

Nous pouvons aussi avoir des types complexes dans les messages :

```
1  syntax="proto3";
2
3  message Employee {
4    int32 id=1;
5    string first_name=2;
6    string last_name=3;
7    bool is_retired=4;
8    int32 age=5;
9    Address current_address=6;
10   repeated string previous_employers=7;
11 }
12
13
14 message Address {
15   string street_name=1;
16   int32 house_number=2;
17   string city=3;
18   string zip_code=4;
19 }
```

Message type

Address message

Nous pouvons aussi avoir des enums si elles sont bien déclarées au préalable dans le message comme suit :

```
1  syntax="proto3";
2
3  message Employee {
4    int32 id=1;
5    string first_name=2;
6    string last_name=3;
7    bool is_retired=4;
8    google.protobuf.Timestamp birth_date=5;
9    Address current_address=6;
10   repeated string previous_employers=7;
11   enum MaritalStatus  {
12     SINGLE=0;
13     MARRIED=1;
14     DIVORCED=2;
15     OTHER=3;
16   }
17   MaritalStatus marital_status=8;
18 }
```

Enum

Enum field

La prochaine chose que je souhaite aborder est le package.

Un package fournit un préfixe de nom unique à tous les messages et services dans le fichier, et il est utilisé pour éviter les conflits de noms entre les messages et services dans le même projet.

Par exemple, si vous avez deux messages nommés "entity" parce que l'un appartient au module X.X et l'autre au groupe de Y.Y de l'organisation,

vous ne voulez pas qu'ils soient en conflit et vous les définissez donc dans des packages différents.

Le package est donc important.

Cependant, il est facultatif et vous n'êtes pas obligé de définir un package.

C'est la meilleure pratique à suivre.

La définition du package doit être placée après la déclaration de syntaxe.

```
syntax = "proto3";

package mypackage.convert;

message convertReq {
  string type = 1;
  bool isBin = 2;
  bytes file = 3;
  bool needDraco = 4;
  bool noZip = 5;
}
```

La prochaine chose que nous allons aborder dans le protocole est l'importation de définitions et d'autres types de messages à partir d'autres fichiers proto, qui peuvent être importés et utilisés.

Cela est extrêmement utile, par exemple, pour utiliser des types externes développés par des tiers ou pour rendre les protocoles modulaires en plaçant des types partagés dans des fichiers dédiés, puis simplement en partageant et en réutilisant ces types au lieu de les définir à partir de zéro à chaque fois.

Si vous regardez à nouveau notre fichier d'exemple, notez la déclaration d'importation ici et ce que nous faisons ici est d'importer le type "timestamp" de la bibliothèque de protobuf Google.

```
syntax = "proto3";

package mypackage.convert;

import "google/protobuf/timestamp.proto";

message convertReq {
  string type = 1;
  bool isBin = 2;
  bytes file = 3;
  bool needDraco = 4;
  bool noZip = 5;
}
```

Nous importons ce type et ici, dans ce champ, nous allons simplement utiliser ce type.

Donc, si vous vous souvenez, nous avons dit avant qu'il n'y a pas de type de date intégré pour le protocole, et c'est ainsi que nous implémentons DateTime dans le protocole.

Nous importons le type "timestamp" de la bibliothèque de protocole Google et l'utilisons ensuite comme type de champ dans notre définition de champ du message "convertReq".

```
syntax = "proto3";

package mypackage.convert;

import "google/protobuf/timestamp.proto";

message convertReq {
  string type = 1;
  bool isBin = 2;
  bytes file = 3;
  bool needDraco = 4;
  bool noZip = 5;
  google.protobuf.timestamp mydate = 6 ;
}
```

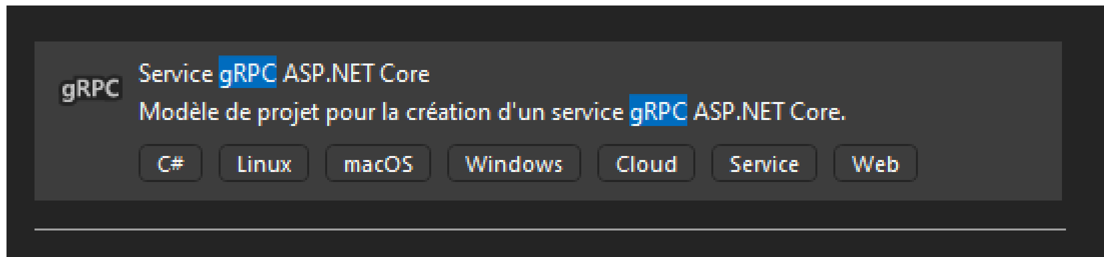
Maintenant, une autre chose que nous pouvons ajouter à notre message est des commentaires et des commentaires peuvent être ajoutés au protocole.

Ils sont la même syntaxe que dans les fichiers csharp c'est-à-dire «// » ou « /**/ ».

Il y a beaucoup de choses que nous allons discuter sur la syntaxe du protocole, mais maintenant il est temps de mettre toute la théorie en pratique.

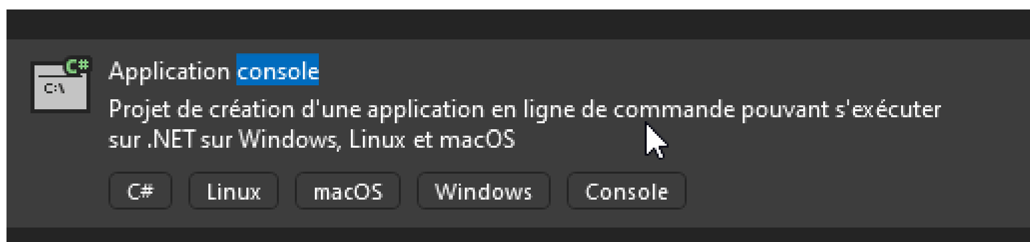
4. Configurer Protobuf avec .NET Core

Nous pourrions créer un projet gRPC directement dans Visual Studio :



Mais nous allons créer un projet Console dans lequel nous allons implémenter un fichier .proto et générer le code avec tout ce que nous avons vu précédemment afin de bien assimiler les concepts que nous avons énuméré.

A la place nous aller créer une application console :



Nous allons créer un dossier Protos en dehors du projet pour avoir nos fichiers .proto accessibles à tous les projets.

Nous allons donc y ajouter un fichier proto bien sur et il aura cette définition :

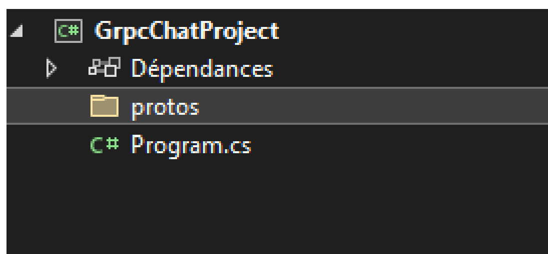
```
syntax="proto3";

package hr.entities;

import "google/protobuf/timestamp.proto";

option csharp_namespace = "Grpc.Exemple.Project";

/* Represents the Employee entity, with a list of previous employers.
Also, this definition contains the current address of the employee. */
message Employee {
  int32 id=1;
  string first_name=2;
  string last_name=3;
  bool is_retired=4;
  google.protobuf.Timestamp birth_date=5;
  Address current_address=6;
  repeated string previous_employers=7; // Leave empty if unknown
  enum MaritalStatus  {
    SINGLE=0;
```

```
    MARRIED=1;
    DIVORCED=2;
    OTHER=3;
  }
  MaritalStatus marital_status=8;
}


message Address {
  string street_name=1;
  int32 house_number=2;
  string city=3;
  string zip_code=4;
}
```

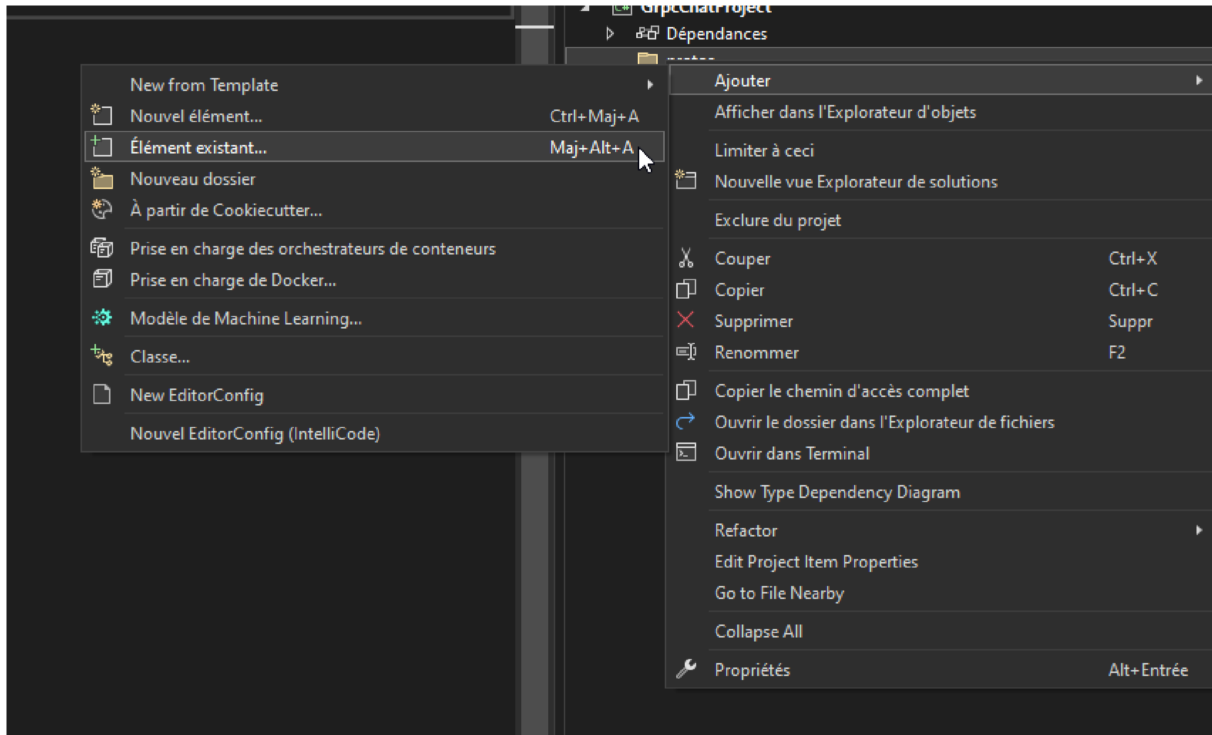
Notez que j'ai ajouté un namespace pour pouvoir ajouter facilement une référence à mon modèle :

```
option csharp_namespace = "Grpc.Exemple.Project";
```

Dans notre projet, nous allons maintenant ajouter un dossier « Protos » comme suit :

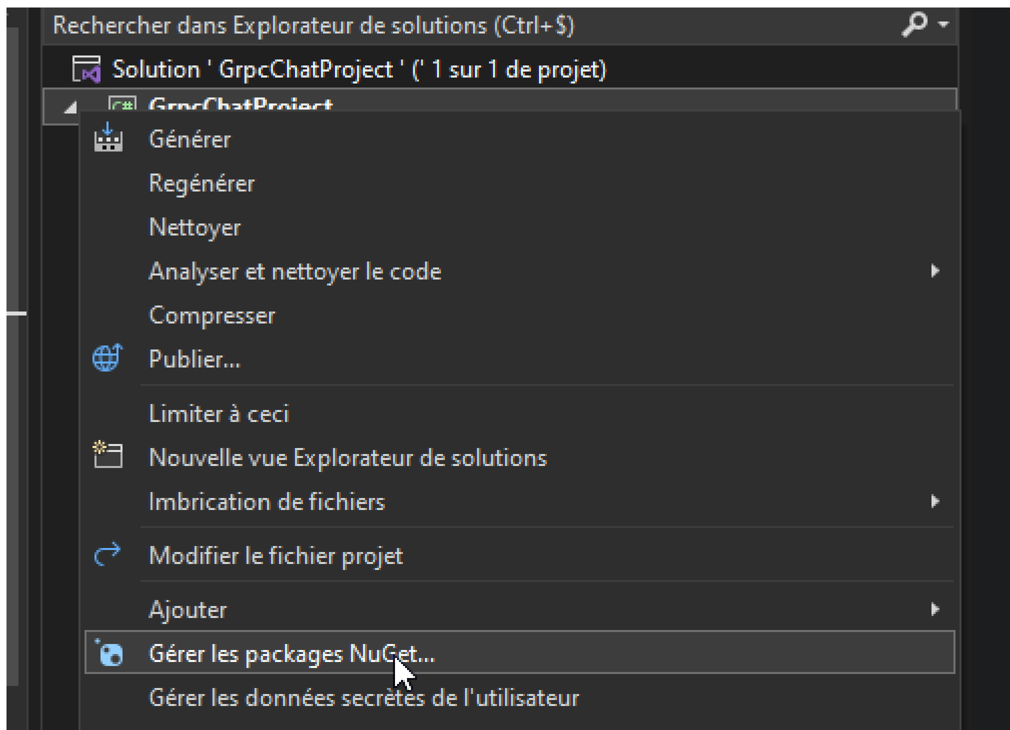


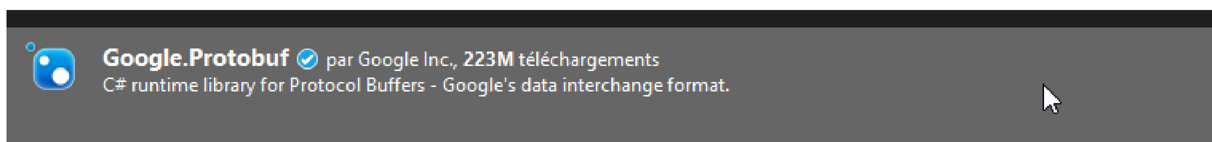
Et y ajouter notre fichier .proto :



Nous avons nos fichiers mais il nous manque encore les librairies gRPC pour .NET Core et le compilateur gRPC .NET Core.

Nous allons donc installer les deux librairies dont nous avons besoin :





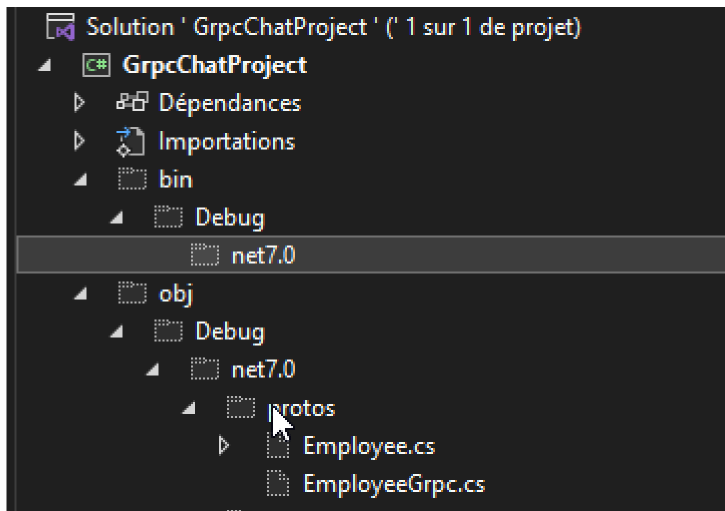
Une fois que ces deux packages sont installés, nous allons pouvoir paramétrer notre projet pour générer le code à partir d'un fichier .proto.

Pour cela, rien de très compliqué, il suffit de double cliquer sur le nom de votre projet pour avoir accès au fichier « csproj » de votre projet et rajouter cette ligne :

```
<ItemGroup>
   <Protobuf Include="protos\employee.proto"/>
</ItemGroup>
```

Le projet va automatique être régénéré et générer par la même occasion les fichiers csharp à partir de votre fichier .proto.

Vous pouvez maintenant observer qu'un dossier « protos » a été généré et qu'il contient vos fichier générés à partir de votre fichier .proto :



Je ne vais pas rentrer dans le détail de ces fichiers car ils sont assez complexes.

Nous allons passer dans le chapitre suivant à l'utilisation de cette entité générée.

5. Developer avec Protobuf

Maintenant que nous avons notre modèle, nous allons pouvoir l'utiliser pour l'afficher dans la console.

Pour cela, nous allons le faire dans le fichier « Program.cs ».

Avant tout, il va falloir ajouter les références aux classes nécessaires :

```
using Google.Protobuf;
using Google.Protobuf.WellKnownTypes;
using Grpc.Exemple.Project;
```

Les deux premières références sont issues des packages Nuget que nous avons installé et la dernière est le namespace de notre fichier .proto.

Nous pouvons donc maintenant utiliser notre classe Employee afin de créer des instances de celle-ci :

```
using Google.Protobuf;
using Google.Protobuf.WellKnownTypes;
using Grpc.Exemple.Project;

// See https://aka.ms/new-console-template for more information
Console.WriteLine("Beginning Proto");

var emp = new Employee();
emp.FirstName = "Alex";
emp.LastName = "CASTRO";
emp.IsRetired = false;
var birthdate = new DateTime(1980, 1, 1);
birthdate = DateTime.SpecifyKind(birthdate,DateTimeKind.Utc);
emp.BirthDate = Timestamp.FromDateTime(birthdate);
emp.MaritalStatus = Employee.Types.MaritalStatus.Other;
emp.PreviousEmployers.Add("XXXX");
emp.PreviousEmployers.Add("YYYYYY");
```

Notez que nous avons dû créer une date de naissance en spécifiant son format en UTC car Protobuf gère les timestamps et ceux-ci doivent être des dates en UTC.

Si nous essayer d'écrire cette instance d'Employee dans un fichier, elle ne sera pas lisible car elle est dans un format qui n'est pas lisible.

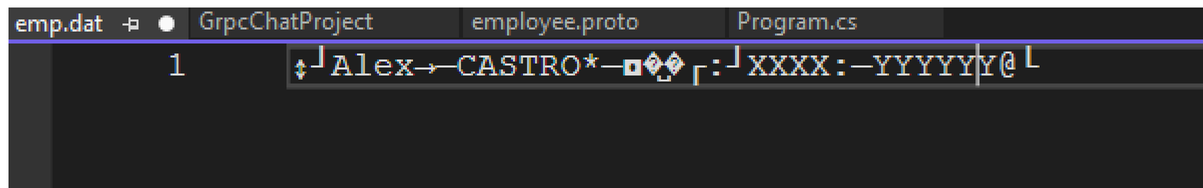
Il existe donc un Parser qui a été généré à partir de notre fichier Proto.

```
using Google.Protobuf;
using Google.Protobuf.WellKnownTypes;
using Grpc.Exemple.Project;

// See https://aka.ms/new-console-template for more information
Console.WriteLine("Beginning Proto");

var emp = new Employee();
```

```
emp.FirstName = "Alex";
emp.LastName = "CASTRO";
emp.IsRetired = false;
var birthdate = new DateTime(1980, 1, 1);
birthdate = DateTime.SpecifyKind(birthdate,DateTimeKind.Utc);
emp.BirthDate = Timestamp.FromDateTime(birthdate);
emp.MaritalStatus = Employee.Types.MaritalStatus.Other;
emp.PreviousEmployers.Add("XXXX");
emp.PreviousEmployers.Add("YYYYYY");

using (var output = File.Create("emp.dat"))
{
    emp.WriteTo(output);
}

Employee employeeFromfile;
using (var input = File.OpenRead("emp.dat"))
{
    employeeFromfile = Employee.Parser.ParseFrom(input);
}

Console.WriteLine(employeeFromfile);
```

Ici nous poussons notre entité vers un fichier et nous lisons ensuite ce fichier pour parser celui-ci et créer une nouvelle entité.

Si nous regardons le fichier généré, il n'est pas lisible :



Mais si nous regardons la console après le parsing, nous avons bien nos valeurs :

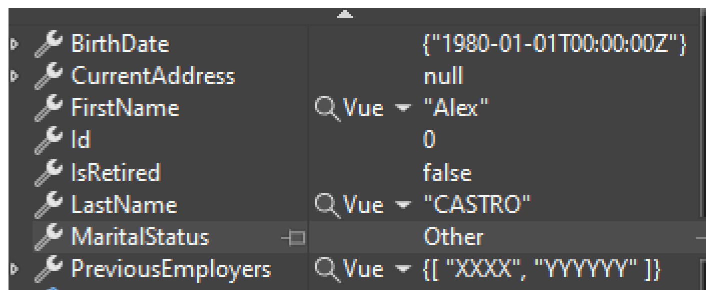
```
Beginning Proto
{ "firstName": "Alex", "lastName": "CASTRO", "birthDate": "1980-01-01T00:00:00Z", "previousEmployers": [ "XXXX", "YYYYYY
" ], "maritalStatus": "OTHER" }
```

Nous avons donc vu un exemple simple.

Nous n'avons pas rempli toutes les valeurs de notre entité mais il existe des valeurs par défaut quoi qu'il arrive :

Field Type	Default Value
string	Empty string
bool	false
Numeric types	Zero
Enums	The first defined enum value, which must be 0
bytes	Empty bytes

L'id que nous n'avons pas rempli est également vide :

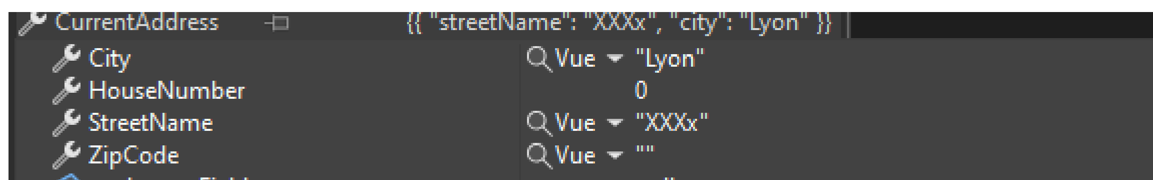


Dans notre exemple nous avons également un type complexe que nous n'avons pas utilisé et qui était donc vide, au dessus c'est le champ CurrentAdress

Dans ce cas pour l'utiliser, c'est exactement le même principe que pour notre entité Employee, nous allons devoir créer une instance et remplir les propriétés.

```
18  var address = new Address();
19  address.City = "Lyon";
20  address.StreetName = "XXXx";
21  emp.CurrentAddress = address;
22
```

Je crée donc une adresse et l'assigne à ma « CurrentAddress » de mon entité.J'ai laissé volontairement des champs non remplis pour observer le comportement des valeurs par défaut.



Nous observons que les propriétés « HouseNumber » et « ZipCode » qui étaient vides ont été remplacées par les valeurs par défaut de leur type de données.

Maintenant, naturellement, les messages seront modifiés au cours de la durée de vie de ce système.

En général, les changements peuvent être apportés aux messages, mais il est important de suivre les règles pour s'assurer que tout le code puisse toujours fonctionner avec les nouveaux messages et vice versa.

Il y a donc certaines règles qu'il est important de suivre.

Et voici la première : ne changez pas les numéros de champ.

Donc, si dans le message d'origine, le champ du prénom, par exemple, est assigné au numéro un, alors ne changez pas ce numéro car ces numéros sont utilisés pour identifier les champs dans le message sérialisé et les changer perturbera les choses.

Car dans votre code généré, il est codé en dur que le numéro un est assigné au champ du prénom.

Et si vous allez maintenant changer ce numéro dans votre profil pour un autre numéro, alors le code généré ne saura pas comment sérialiser ce champ.

Donc ne changez pas les numéros des champs.

Règle numéro deux : si vous ajoutez de nouveaux champs, les messages sérialisés par le code basé sur tous les messages fonctionneront toujours.

Et cela a du sens car comme nous venons de le voir, si vous ne donnez pas de valeur à divers champs dans le message, alors ils obtiendront simplement les valeurs par défaut, ce qui se produira exactement avec les nouveaux champs qui ne sont pas assignés de valeurs.

Il n'y a donc aucun problème à ne pas assigner de valeurs aux nouveaux champs.

Règle numéro trois : les champs peuvent être supprimés.

Il est simplement important de ne pas réutiliser les numéros de champs, et cela pour la raison dont nous venons de discuter dans la règle numéro un, ces numéros sont déjà utilisés par le code généré.

Vous ne pouvez donc pas réutiliser ces numéros.

Si vous supprimez des champs et que vous ajoutez de nouveaux champs, utilisez des numéros non utilisés pour les nouveaux champs.

La règle suivante, lors du changement de types de champs, notez que la plupart des types numériques sont compatibles.

Par exemple, vous pouvez passer de int32 à int64 à bool etc. Les chaînes de caractères et bool sont également compatibles, mais seulement si les valeurs en octets sont des caractères UTF-8 valides.

Voilà les choses à retenir lors du changement de types de champs.

Ce sont donc les règles pour la mise à jour d'un type de message.

Et si vous suivez et retenez ces règles, vous n'aurez probablement aucun problème lors de la mise à jour de votre type de message.

8. Utiliser Oneof

Maintenant nous allons voir le mot clé OneOf.

OneOf n'est pas le package utilisé pour renvoyer plusieurs types de réponse en .NET mais le mot clé pour un type de données Protobuf utilisé pour renvoyer seulement un champ sur plusieurs.

Il existe des cas où vous n'aurez besoin de renvoyer qu'une seule propriété et pour cela vous devrez utiliser le mot clé OneOf.

Ce n'est peut-être pas clair tout de suite mais ça va le devenir 😊

Donc si nous revenons sur le code précédent, nous pourrions remplacer le champ « birth_date » par une propriété de type « oneof » comme suit :

```
syntax="proto3";

package hr.entities;

import "google/protobuf/timestamp.proto";

option csharp_namespace = "Grpc.Exemple.Project";

/* Represents the Employee entity, with a list of previous employers.
Also, this definition contains the current address of the employee. */
message Employee {
  int32 id=1;
  string first_name=2;
  string last_name=3;
  bool is_retired=4;
  //google.protobuf.Timestamp birth_date=5;
  oneof birth_data {
      google.protobuf.Timestamp birth_date=5;
      int32 age = 9;
```

```
  }
  Address current_address=6;
  repeated string previous_employers=7; // Leave empty if unknown
  enum MaritalStatus  {
    SINGLE=0;
    MARRIED=1;
    DIVORCED=2;
    OTHER=3;
  }
  MaritalStatus marital_status=8;
}


message Address {
  string street_name=1;
  int32 house_number=2;
  string city=3;
  string zip_code=4;
}
```

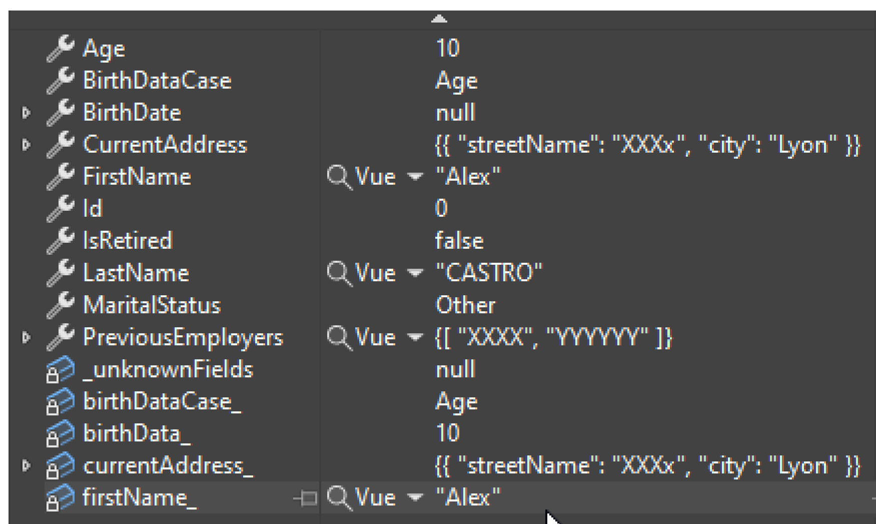
Ceci induit que soit la date de naissance sera rempli mais pas les deux, un seul champ peut avoir une valeur malgré le fait que vous puissiez remplir les deux..

Donc dans notre code nous pouvons remplir l'age juste après la date de naissance :

```
using Google.Protobuf;
using Google.Protobuf.WellKnownTypes;
using Grpc.Exemple.Project;

// See https://aka.ms/new-console-template for more information
Console.WriteLine("Beginning Proto");

var emp = new Employee();
emp.FirstName = "Alex";
emp.LastName = "CASTRO";
emp.IsRetired = false;
var birthdate = new DateTime(1980, 1, 1);
birthdate = DateTime.SpecifyKind(birthdate,DateTimeKind.Utc);
emp.BirthDate = Timestamp.FromDateTime(birthdate);
emp.Age = 10;
emp.MaritalStatus = Employee.Types.MaritalStatus.Other;
emp.PreviousEmployers.Add("XXXX");
emp.PreviousEmployers.Add("YYYYYY");
var address = new Address();
address.City = "Lyon";
```

```
address.StreetName = "XXXx";
emp.CurrentAddress = address;

using (var output = File.Create("emp.dat"))
{
   emp.WriteTo(output);
}

Employee employeeFromfile;
using (var input = File.OpenRead("emp.dat"))
{
   employeeFromfile = Employee.Parser.ParseFrom(input);
}

Console.WriteLine(employeeFromfile);
```

Nous avons bien la possibilité de remplir les deux valeurs comme décrit ci-dessus mais si nous inspectons la valeur, nous voyons deux choses :



Tout d'abord la date de naissance est NULL et seul l'age est rempli.

Ensuite, nous avons une nouvelle propriété qui est apparue « BirthDataCase » qui est la pour nous dire quelle valeur est remplie et dans notre cas c'est l'age.

L'utilisation de OneOf correspond à des cas très particuliers mais il est important de connaitre cette notion.

9. Maps

Maintenant, les **Maps** peuvent être définies comme un champ spécialisé dans un message, et les **Maps** sont essentiellement un HashTable ou un dictionnaire.

Et vous connaissez probablement au moins l'un de ces deux types.

Tout comme dans le dictionnaire ou le HashTable, les clés doivent être uniques sinon le parsing va générer une exception.

Voyons maintenant comment l’utiliser.

Dans notre solution, rajoutons une nouvelle propriété de type « Map ».

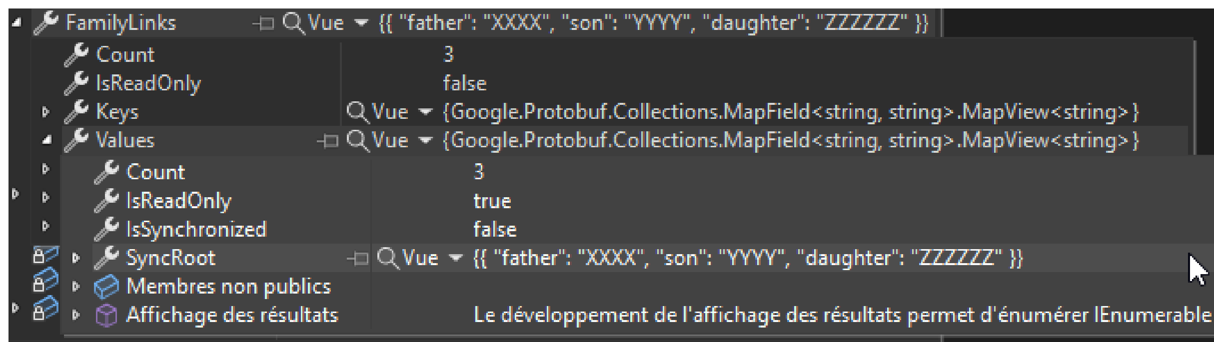
```
syntax="proto3";

package hr.entities;

import "google/protobuf/timestamp.proto";

option csharp_namespace = "Grpc.Exemple.Project";

/* Represents the Employee entity, with a list of previous employers.
Also, this definition contains the current address of the employee. */
message Employee {
  int32 id=1;
  string first_name=2;
  string last_name=3;
  bool is_retired=4;
  //google.protobuf.Timestamp birth_date=5;
  oneof birth_data {
      google.protobuf.Timestamp birth_date=5;
      int32 age = 9;
  }
  Address current_address=6;
  repeated string previous_employers=7; // Leave empty if unknown
  enum MaritalStatus  {
    SINGLE=0;
    MARRIED=1;
    DIVORCED=2;
    OTHER=3;
  }
  MaritalStatus marital_status=8;
  map<string,string> familyLinks = 10;
}


message Address {
  string street_name=1;
  int32 house_number=2;
  string city=3;
  string zip_code=4;
}
```

Cette propriété nous servira à rajouter les liens de parenté pour notre employé.

Voici donc comment l'utiliser en C# :

```
emp.FamilyLinks.Add("father", "XXXX");
emp.FamilyLinks.Add("son", "YYYY");
emp.FamilyLinks.Add("daughter", "ZZZZZZ");
```

Et voyons son comportement avec un point d'arrêt :



Le champ se comporte comme un dictionnaire C# mais est de type Google.Protobuf.Collections.MapField<string, string>.

Nous avons fait le tour des champs et comment utiliser les entités générées.

10. Définir des Services

Maintenant que nous savons utiliser les messages, il est temps de créer un service car gRPC est une Web API orientée méthode / action et donc service par extension.

La définition d'un service est extrêmement simple , il suffit de créer un service avec un nom, un paramètre d'entrée et un paramètre de réponse.

Tout cela se fait dans le fichier proto.

Tout d'abord nous allons créer le message d'entrée :

```
message EmployeeId {
   int32 id=1;
}
```

Ce sera donc un message contenant l'identifiant d'un employé.

Et ensuite le service :

```
service EmployeeService {
   rpc GetEmployee(EmployeeId) returns (Employee);
}
```

Donc notre service renverra un employé via son identifiant.

Une fois que vous aurez regénéré, le code correspondant sera généré et utilisable !

Le code de cette section est disponible ici : « Exercices\01»

Chapitre 8 : Monter le serveur chat Room

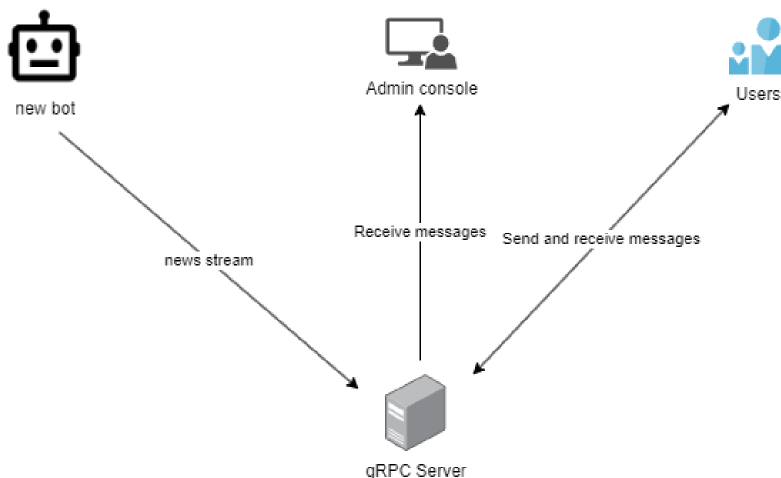
1. Introduction

Maintenant que nous avons les bases pour travailler avec gRPC et Protobuf, nous allons pouvoir passer à la création de notre serveur.

Celui-ci sera le point central de notre projet car il contiendra toutes les méthodes pour les autres composants :

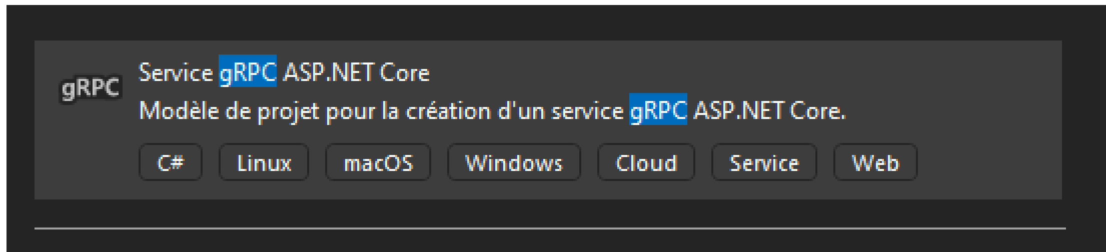
- S'enregistrer dans une chat room
- La méthode de streaming client pour le news bot
- La méthode de streaming server pour la console admin
- Les méthodes pour envoyer et recevoir des messages

Je vous remets pour mémo l'architecture des composants :



2. Créer et configure le projet

Nous allons donc maintenant pouvoir créer notre solution et pour cela nous allons utiliser le template gRPC :



Comme vous pourrez le constater, le projet contient déjà un fichier proto et un service mais nous allons créer nos propres services et fichiers proto.

Commençons par le fichier .proto et nous allons premièrement gérer l'enregistrement à une chat room :

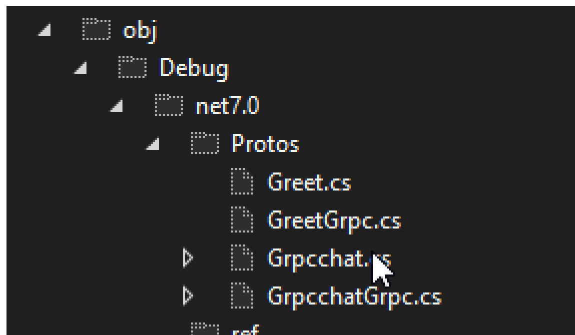
```
syntax = "proto3";

option csharp_namespace = "ChatServer.Protos";

package chatgrpc;

message RoomRegistrationRequest {
	string roomId = 1;
}

message RoomRegistrationResponse {
	int32 room_id = 1;
}

service ChatGrpc {
	rpc RegisterToRoom(RoomRegistrationRequest) returns
(RoomRegistrationResponse);
}
```

Nous avons donc ici une requête d'entrée et une requête de réponse et un service.

Maintenant nous pouvons nettoyer le service et supprimer le GreeterService et ensuite commenter la ligne suivante car nous en aurons besoin par la suite :

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

Maintenant si nous regénérons la solution, nous voyons que le code à partir de notre fichier proto a été généré :



Il ne nous reste plus qu'à implémenter la logique de notre service comme c'était fait auparavant pour le GreeterService.

3. Créer le Chat Room Service

Maintenant que nous avons un service « appelable » par l'extérieur, nous allons pouvoir implémenter la logique de notre service comme le faisant le GreeterService avant sa suppression.

Pour cela, nous allons créer un ChatGrpcService qui va hériter de notre service généré et qui implémentera une méthode assez basique afin de pouvoir renvoyer une réponse et pouvoir tester ce service :

```
using ChatServer.Protos;
using Grpc.Core;

namespace ChatServer.Services
{
    public class ChatGrpcService :
ChatServer.Protos.ChatGrpcService.ChatGrpcServiceBase
    {
        private readonly ILogger<ChatGrpcService> _logger;
        public ChatGrpcService(ILogger<ChatGrpcService> logger)
        {
            _logger = logger;
        }

        public override Task<RoomRegistrationResponse>
RegisterToRoom(RoomRegistrationRequest request, ServerCallContext
context)
        {
```

```
            _logger.LogInformation("Service called...");
            var rnd = new Random();
            var roomNum = rnd.Next(1, 100);
            _logger.LogInformation($"Room no. {roomNum}");
            var resp = new RoomRegistrationResponse { RoomId = roomNum
};
            return Task.FromResult(resp);
        }
    }
}
```

En jaune vous avez la classe d'héritage de notre service, elle nous permettra de « surcharger » (override) pour pouvoir coder la logique de notre méthode qui devra porter le même nom et les mêmes paramètres de requête et réponse que celui défini dans le fichier .proto.

A noter que nous avons introduit la notion de ServerCallContext en paramètre mais nous en parlerons plus tard.

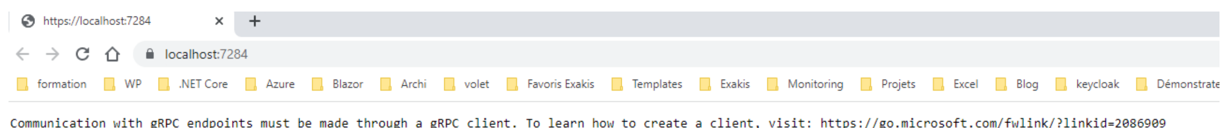
Une fois que vous avez implémenté cela, nous allons pouvoir remettre une référence à ce service dans le fichier « Program.cs » et remplacer
//app.MapGrpcService<GreeterService>(); par
app.MapGrpcService<ChatGrpcService>();

Vous pouvez même lancer votre projet et voir une page web s'afficher avec une erreur qui est normale.

Pour cela il va falloir changer le fichier « launchsettings.json » dans le dossier Properties :

```json
{
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5049",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7284;http://localhost:5049",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Et passer les valeurs à **true.**



L'erreur est normale car on doit consommer via un client gRPC 😊

4. Tester le Service avec BloomRPC

Pour tester notre service, nous allons donc utiliser POSTMAN qui inclut le support de gRPC !

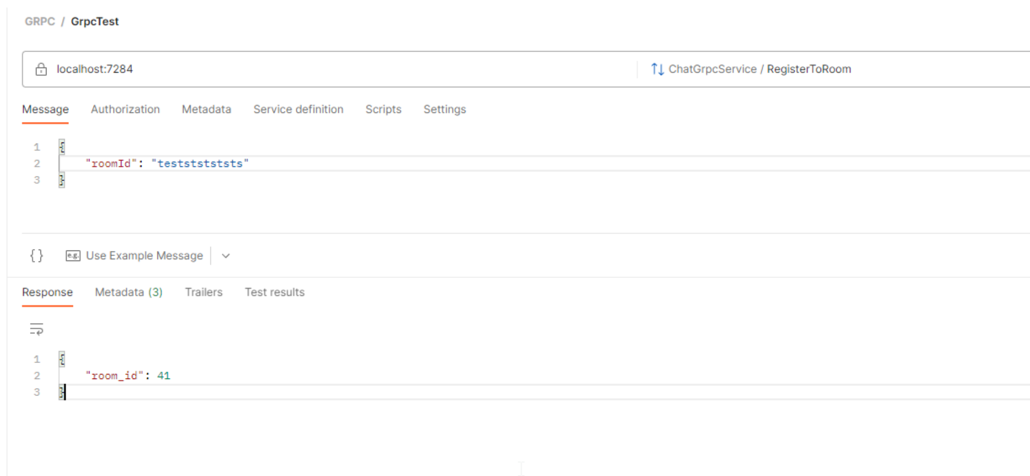
Je vous laisse donc installer POSTMAN : https://www.postman.com/downloads/

Ensuite une fois POSTMAN installé, vous allez pouvoir configurer une requête :

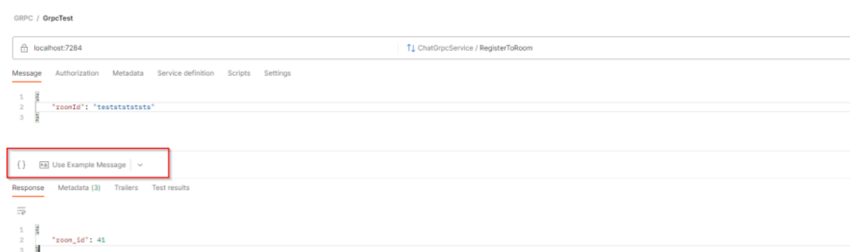
- **1-** Tout d'abord l'url du serveur avec le port mais sans « https:// ».
- **2-** Ensuite il va falloir importer votre fichier proto pour la découverte des méthodes existantes.

- **3-** Pour finir le contenu de la requête en suivant la définition de votre fichier proto :
  J’ai volontairement mis un champ qui n’a pas le même nom que le champ d’origine car seul le numéro du champ compte mais il faut bien que le type soit le même !

Une fois tout cela effectué, vous pouvez cliquer pour lancer votre appel et voir la réponse :



POSTMAN est capable également de générer des messages de requête pour vous en cliquant sur ce bouton :



Vous savez maintenant comment tester vos services gRPC avec POSTMAN.

Le code de cette section est disponible ici : « Exercices\02 - Server»

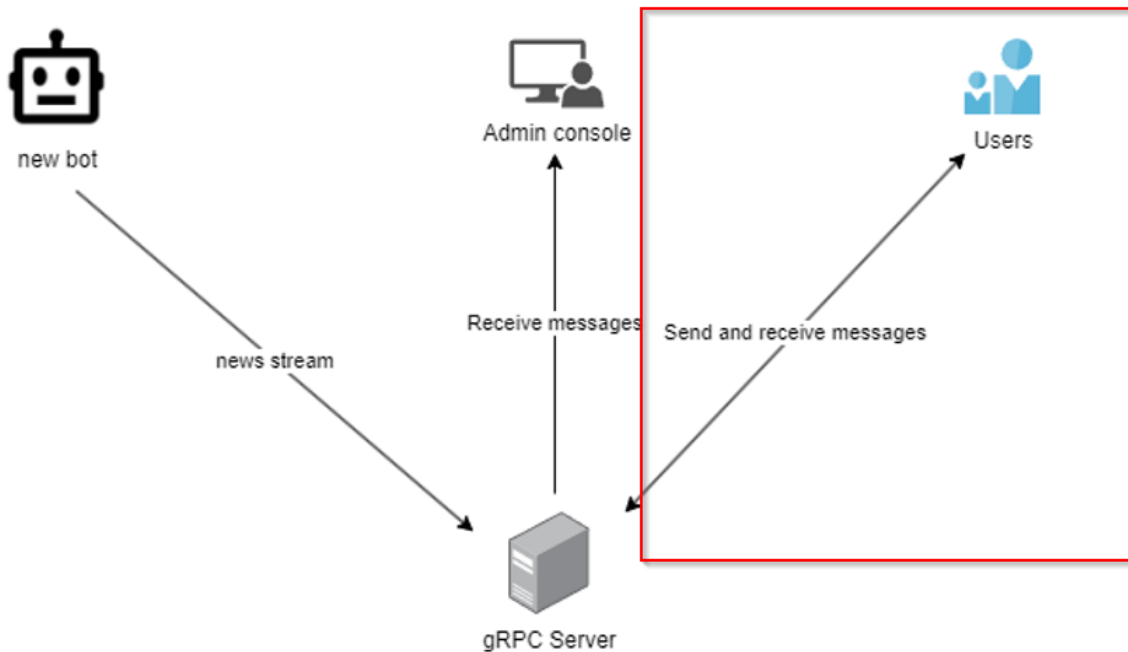
Chapitre 9 : Appels unaires RPC

1. Introduction

Nous avons implémenté une méthode sur le serveur qui permet de se connecter à une chat room mais personne ne la consommé.

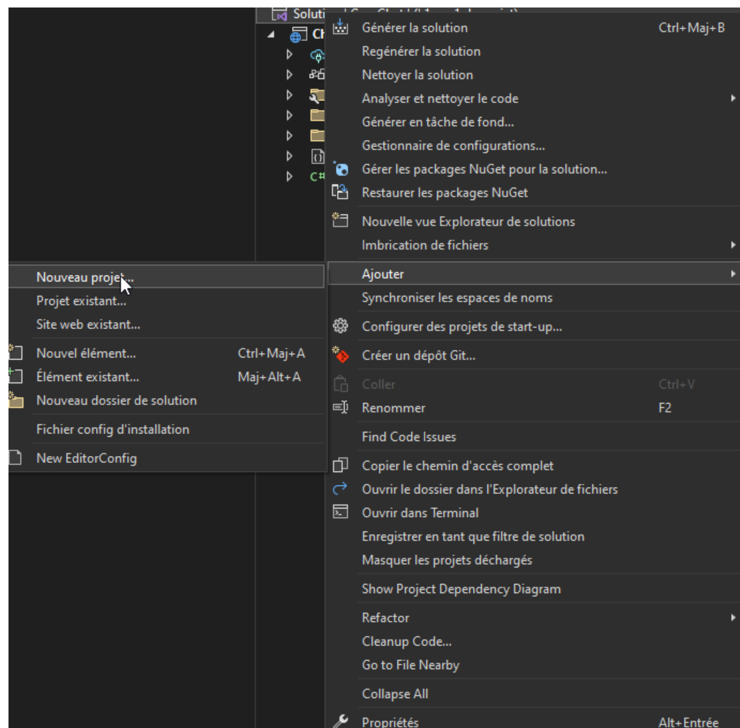
Nous allons donc maintenant créer le client pour consommer cette méthode !

Cela correspond à cette partie :



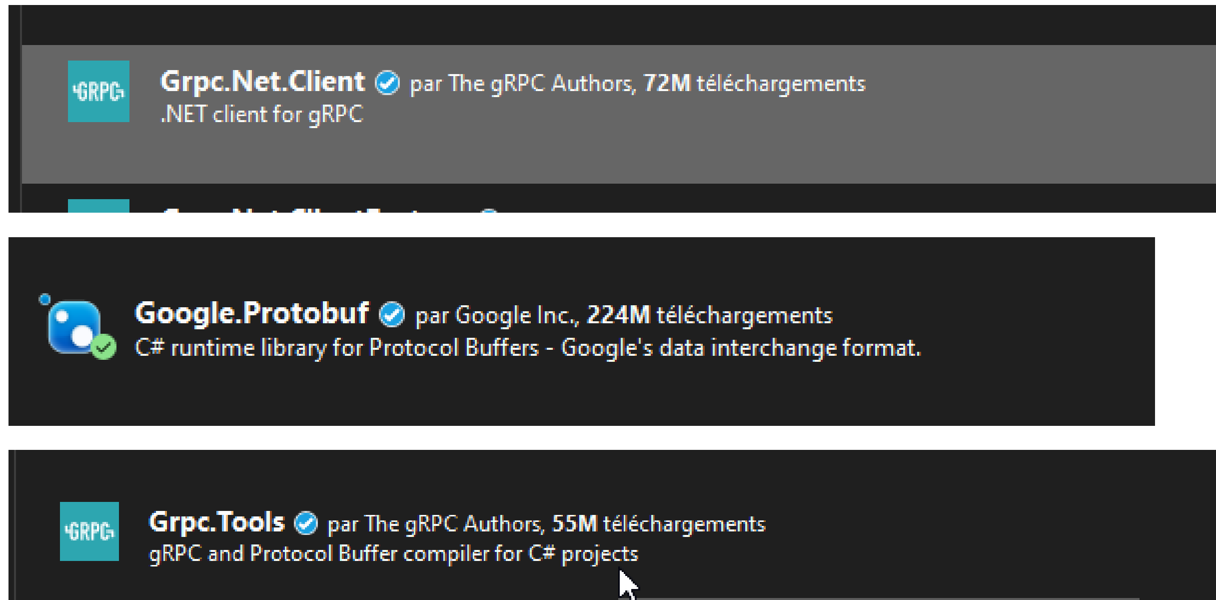
2. Implémentation du Client

Pour implementer le client, nous allons commencer par créer un nouveau projet dans notre solution :



Et nous allons ajouter un projet de type Console.

Pour ce projet, nous allons rajouter les packages Nuget necessaires.



Voici donc les 3 packages à ajouter pour pouvoir communiquer avec le serveur et rajouter le même fichier .proto.

Nous allons également, comme pour le serveur, créer un dossier “Protos”.

Et oui nous rajoutons également le même fichier car pour pouvoir communiquer, le serveur et le client doivent avoir exactement le même fichier. N'oublions pas que gRPC est un langage qui définit un "contrat" entre deux composants et le contrat doit être exactement le même pour que cela fonctionne.

Et pour finir nous allons modifier le fichier "csproj" et double cliquant sur le nom du projet puis en rajoutant ces lignes :

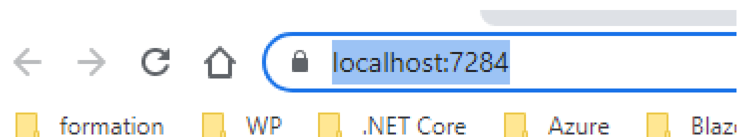
```
<ItemGroup>
	<Protobuf Include="Protos\grpcchat.proto" GrpcServices="Client" />
</ItemGroup>
```

La seule difference avec le serveur est que nous nous positionnons en tant que client et non en tant que serveur.

Lorsque vous aurez sauvegardé, le code va être généré pour pouvoir consommer les méthodes du serveur via le contrat (le fichier .proto)

Ensuite dans le fichier Program.cs, nous allons rajouter les références nécessaires pour utiliser le code généré :

```
using Grpc.Net.Client;
using ChatServer.Protos;
```

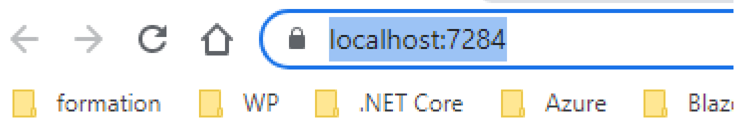


Voici le code dans le fichier Program.cs :

```
using Grpc.Net.Client;
using ChatServer.Protos;


using var channel = GrpcChannel.ForAddress("https://localhost:7284");
var client = new ChatGrpcService.ChatGrpcServiceClient(channel);
Console.WriteLine("Enter a room name to register :");
var roomName = Console.ReadLine();

var response = client.RegisterToRoom(new RoomRegistrationRequest {
RoomName = roomName});
```

```
Console.WriteLine($"RoomId : {response.RoomId}");
Console.Read();
```

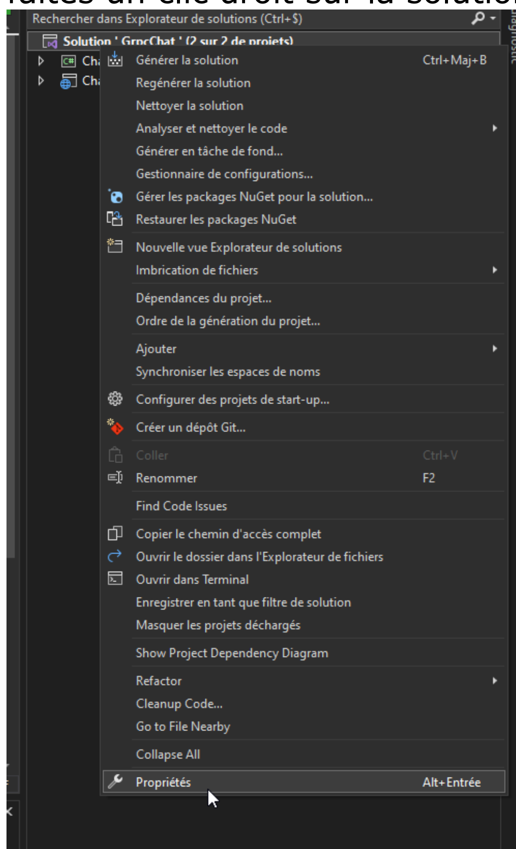
Ici nous nous connectons au serveur via using var channel = GrpcChannel.ForAddress("https://localhost:7284");
Pour connaitre le port de votre application ;il suffit de lancer le serveur :



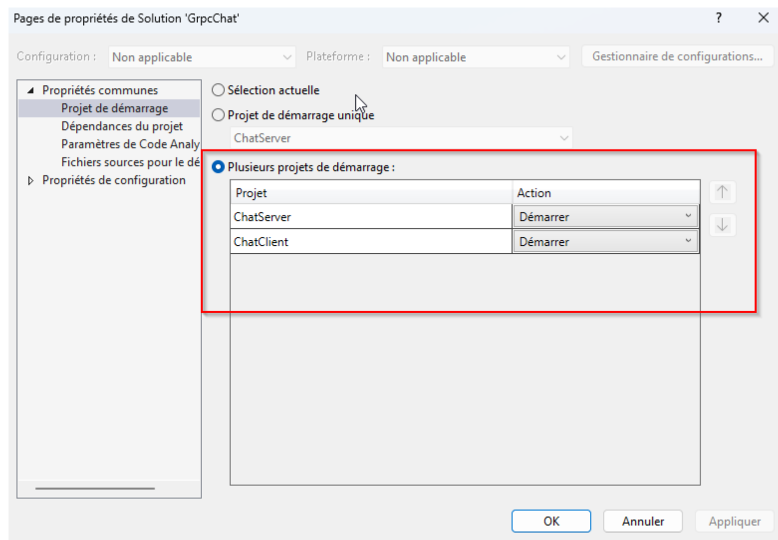
Ensuite nous invitions l'utilisateur à saisir le nom de la room à laquelle il veut se connecter et le serveur va nous renvoyer l'identifiant de la room.

Voila ! nous avons consommé un service gRPC via un client avec une requête unaire 😊

Afin de tester vous allez devoir démarrer les deux composants , pour cela faites un clic droit sur la solution pour voir les propriétés :



Et passez en mode multi projets de démarrage avec le serveur en premier (donc au-dessus du client) comme ceci :



Le code de cette section est disponible ici : « Exercices\03 - Unary»

Chapitre 10 : Client-side Streaming

1. Introduction

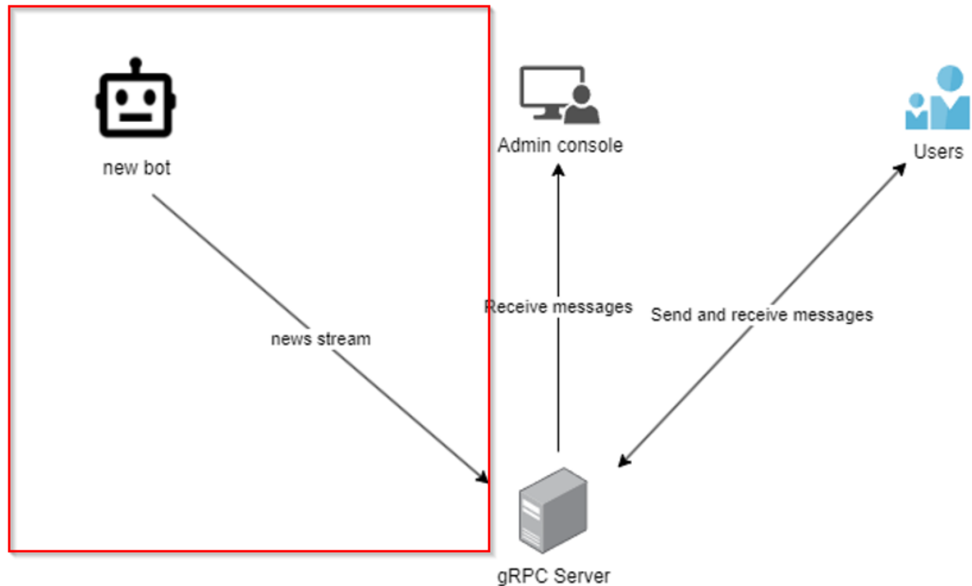
Nous savons maintenant consommer une méthode gRPC sur un serveur et créer des clients pour effectuer des appels unaires (des appels classiques request-response)

Il est temps de passer au réel intérêt de gRPC qui est le streaming que ce soit unidirectionnel ou bidirectionnel.

Nous allons donc commencer pour cela par implémenter le streaming client c'est à dire le streaming du client vers le serveur.

Pour cela le client ouvre une connexion au serveur et envoie des messages en continu.

Nous allons pour cela implémenter un composant qui effectue tout cela : le news bot :



2. Implémenter le stream coté client sur le serveur

Pour cela il va falloir que nous ajoutions du code sur notre serveur pour qu'il puisse gérer tout cela.

Commençons par ouvrir notre projet et allons sur le fichier .proto dans la solution serveur.

Nous allons importer le type de données timestamp :

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

Et créer deux nouveaux messages :

```
message NewsFlash {
	google.protobuf.Timestamp news_time = 1 ;
	string news_item = 2 ;
}

message NewsStreamStatus {
	bool success =1 ;
}
```

Le premier sera le stream qui sera envoyé par le news bot et le second sera la réponse retournée par le serveur.

Pour finir, il nous faut une méthode de service pour gérer cela :

```
rpc SendNewsFlash(stream NewsFlash) returns (NewsStreamStatus);
```

Notez qu'elle ne prend par une requête en entrée mais un stream !

Notre fichier .proto ressemble donc maintenant à ceci :

```
syntax = "proto3";

option csharp_namespace = "ChatServer.Protos";
import "google/protobuf/timestamp.proto";
package chatgrpc;

message RoomRegistrationRequest {
	string room_name = 1;
}

message RoomRegistrationResponse {
	int32 room_id = 1;
}

message NewsFlash {
	google.protobuf.Timestamp news_time = 1 ;
	string news_item = 2 ;
}

message NewsStreamStatus {
	bool success =1 ;
}

service ChatGrpcService {
	rpc RegisterToRoom(RoomRegistrationRequest) returns
(RoomRegistrationResponse);
	rpc SendNewsFlash(stream NewsFlash) returns
(NewsStreamStatus);
}
```

Maintenant que notre code a été généré à partir du fichier proto, il va falloir implémenter la logique dans le service :

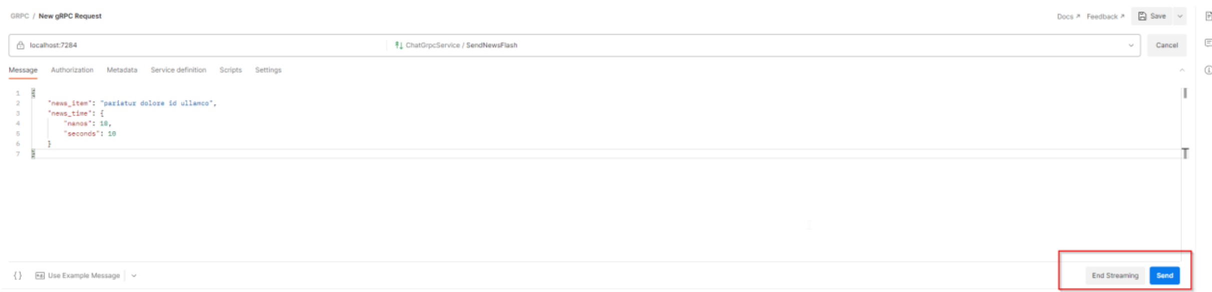
```
public override async Task<NewsStreamStatus>
SendNewsFlash(IAsyncStreamReader<NewsFlash> newsStream,
ServerCallContext context)
```

```
{
        while (await newsStream.MoveNext())
        {
            var news = newsStream.Current;
            Console.WriteLine($"Here is new flash info : {news.NewsItem}
at {news.NewsTime.ToDateTime()}");
        }

        return new NewsStreamStatus {Success = true};
 }
```

La méthode cette fois-ci prend en entrée un IAsyncStreamReader et donc tant que la connection est maintenue, nous parcourons le stream pour renvoyer les données vers la console et une fois la connexion terminée, nous renvoyons notre réponse !

Il est temps de tester tout cela !

Pour cela retournons dans POSTMAN et mettons à jour notre fichier proto pour accéder à ce nouveau service :



Comme vous pouvez le constater, POSTMAN a bien détecté que nous étions sur du streaming client !

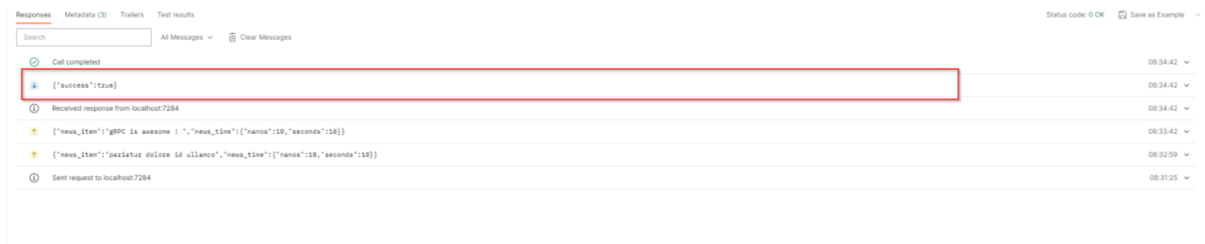
Il suffit donc de cliquer sur « Send » pour envoyer notre message et avoir notre retour dans la console :

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7284
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5049
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\alexandre.castro\OneDrive - Magellan Partners\Perso\Grpc\Exercices\04 - Client Streami
ng\GrpcChat\ChatServer
Here is new flash info : pariatur dolore id ullamco at 01/01/1970 00:00:10
```

Nous pouvons modifier le message ensuite et le renvoyer.

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7284
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5049
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\alexandre.castro\OneDrive - Magellan Partners\Perso\Grpc\Exercices\04 - Client Streami
ng\GrpcChat\ChatServer
Here is new flash info : pariatur dolore id ullamco at 01/01/1970 00:00:10
Here is new flash info : gRPC is awesome !  at 01/01/1970 00:00:10
|
```

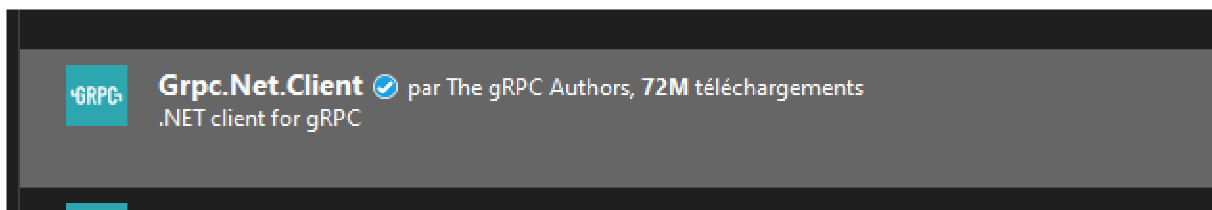
Et dès que nous terminerons la connexion au serveur via le bouton « End Streaming », nous aurons notre statut de retour renvoyé par le serveur.

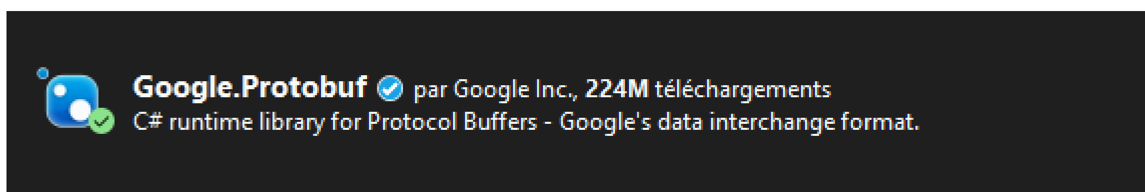


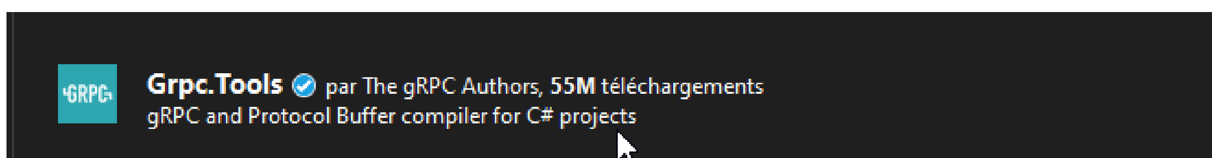
Maintenant que notre serveur est prêt à recevoir des messages, il est temps d'implémenter le news bot !

3. Implémenter le Client news bot

Nous allons donc maintenant créer un nouveau projet console et ajouter les mêmes packages que pour le client :







Nous allons aussi créer un dossier « Protos » et importer le fichier proto du serveur et rajouter les mêmes lignes que sur le client :

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

Voila nous sommes prêts à streamer des données vers le serveur.

Pour info, nous pourrions nettoyer le fichier proto en ne laissant que la méthode et les messages qui nous intéressent, à savoir l'envoi des news.

Nous allons implémenter cela dans le fichier Program.cs.

Pour commencer nous allons définir une liste de news à streamer :

```
var news = new List<NewsFlash>
{
   new() { NewsItem = "Weather is sunny !" , NewsTime =
Timestamp.FromDateTime(DateTime.Now)},
   new() { NewsItem = "Time to learn gRPC !" , NewsTime =
Timestamp.FromDateTime(DateTime.Now)},
   new() { NewsItem = "Go on with gRPC !" , NewsTime =
Timestamp.FromDateTime(DateTime.Now)}
};
```

Initialiser la connexion comme nous l'avons fait sur le client :

```
using var channel = GrpcChannel.ForAddress("https://localhost:7284");
var client = new ChatGrpcService.ChatGrpcServiceClient(channel);
```

Et la ou notre va différer c'est que nous allons appeler la méthode pour envoyer les news sans paramètre pour initier le streaming et ensuite dans une boucle allons envoyer nos news avec un délai pour se rendre compte de l'envoi.

```
Thread.Sleep(2000);
var call = client.SendNewsFlash();
foreach (var newItem in news)
{
   await call.RequestStream.WriteAsync(newItem);
   await Task.Delay(2000);
}
```

La méthode en jaune est la méthode qui permet de streamer du client vers le serveur.

Lorsque le streaming est terminé et que nous avons envoyé toutes nos news, nous pouvons fermer la connexion et récupérer la réponse du serveur :

```
await call.RequestStream.CompleteAsync();
var response = await call;

Console.WriteLine($"Streaming status : {(response.Success ?  "Success" : "Failed")}");
Console.Read();
```

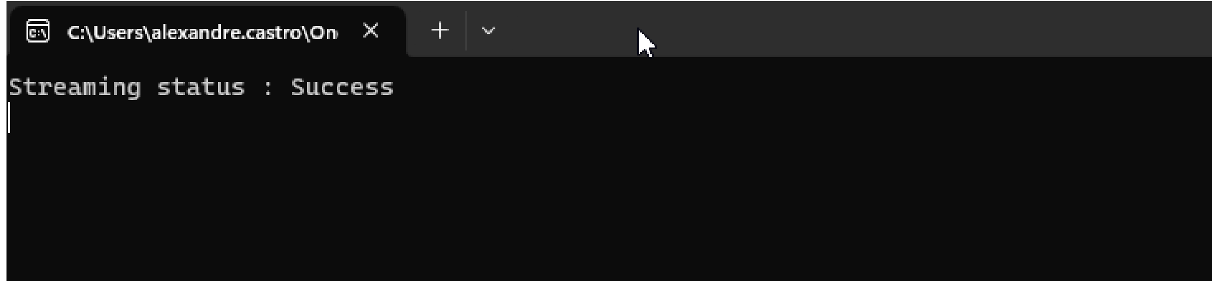
Donc notre fichier donne ce code-là :

```
using Grpc.Net.Client;
using ChatServer.Protos;
using Google.Protobuf.WellKnownTypes;


var news = new List<NewsFlash>
{
    new() { NewsItem = "Weather is sunny !" , NewsTime = Timestamp.FromDateTime(DateTime.UtcNow)},
    new() { NewsItem = "Time to learn gRPC !" , NewsTime = Timestamp.FromDateTime(DateTime.UtcNow)},
    new() { NewsItem = "Go on with gRPC !" , NewsTime = Timestamp.FromDateTime(DateTime.UtcNow)}
};


using var channel = GrpcChannel.ForAddress("https://localhost:7284");
var client = new ChatGrpcService.ChatGrpcServiceClient(channel);

Thread.Sleep(2000);

var call = client.SendNewsFlash();
foreach (var newItem in news)
{
    await call.RequestStream.WriteAsync(newItem);
    Thread.Sleep(2000);
}
await call.RequestStream.CompleteAsync();
var response = await call;

Console.WriteLine($"Streaming status : {(response.Success ?  "Success" : "Failed")}");
Console.Read();
```

Et si nous observons la console du serveur :

```
Here is new flash info : Weather is sunny ! at 15/04/2023 07:05:21
Here is new flash info : Time to learn gRPC ! at 15/04/2023 07:05:21
Here is new flash info : Go on with gRPC ! at 15/04/2023 07:05:21
```

Et lorsque le streaming est terminé :



Nous avons donc mis en place maintenant du streaming client !

Le code complet de ce chapitre est disponible ici : « Exercices\04 - Client Streaming »

Chapitre 11 : Server-side Streaming

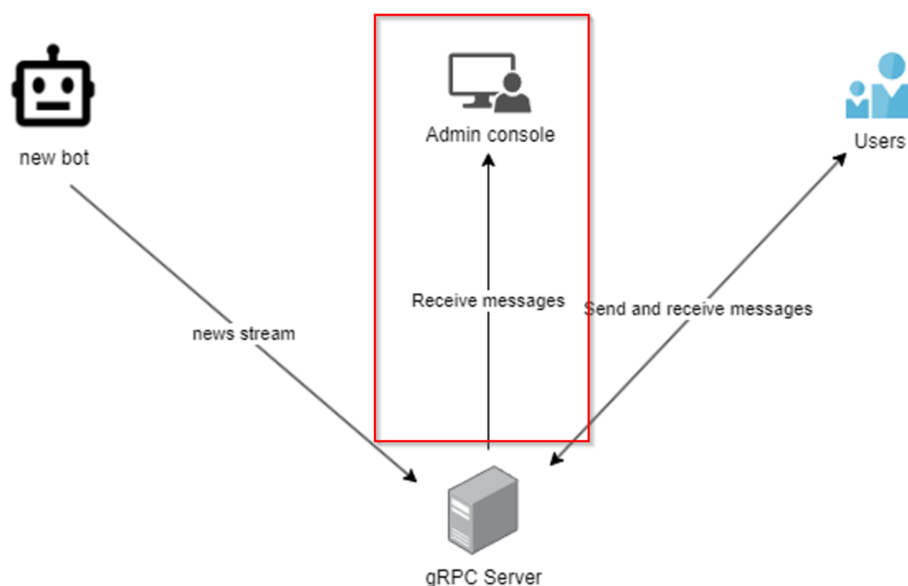
1. Introduction

Nous savons maintenant consommer une méthode gRPC sur un serveur et créer des clients pour effectuer des appels unaires (des appels classiques request-response), créer un client fera du streaming.

Nous allons donc pour cela par implémenter le streaming server c'est à dire le streaming du serveur vers le client.

Pour cela le client ouvre une connexion au serveur et envoie des messages en continu.

Nous allons pour cela implémenter un composant qui effectue tout cela : l'admin console.



2. Implémenter le Server-side Streaming sur le serveur

Reprenons notre projet et commençons par la partie Serveur et allons modifier le fichier proto du serveur pour qu'il puisse exposer une méthode de streaming serveur.

Pour cela nous allons importer un nouveau type de données :

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

Et rajouter un nouveau message :

```
message ReceivedMessage {
	google.protobuf.Timestamp msg_time = 1 ;
```

```
	string content = 2;
	string user = 3;
}
```

Celui-ci sera un message reçu par le serveur.

Maintenant nous allons implémenter la méthode qui utilisera ce message :

```
rpc StartMonitoring(google.protobuf.Empty) returns (stream
ReceivedMessage);
```

Cette méthode reçoit un message vide en entrée et retourne un stream avec des messages reçus et renvoyés par le serveur.

Nous pouvons donc maintenant implémenter la méthode de service qui contiendra la logique !

```
public override async Task StartMonitoring(Empty request,
IServerStreamWriter<ReceivedMessage> streamWriter,
            ServerCallContext context)
        {
            while (true)
            {
                await streamWriter.WriteAsync(new ReceivedMessage
                {
                    MsgTime = Timestamp.FromDateTime(DateTime.UtcNow),
User = "UserMe!",
                    Content = "Content of the message"
                });
                await Task.Delay(1000);
            }
        }
```

Ici nous streamons des messages vers le client connecté donc à l’inverse du chapitre précédent ou le client envoyait des messages.

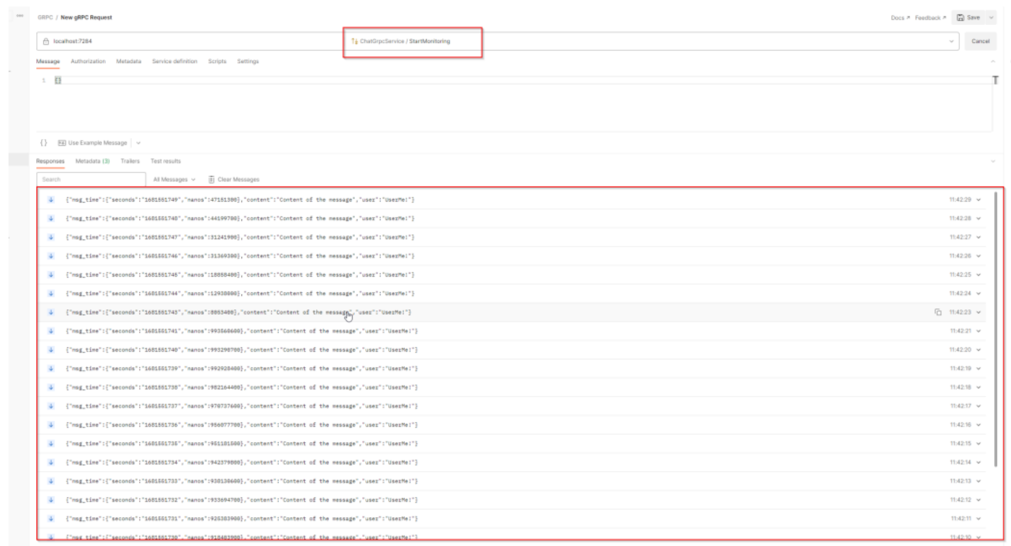
Pour tester, il suffit de lancer le serveur et de lancer un appel vers la méthode : StartMonitoring avec une requête vide pour recevoir des messages toutes les secondes.

Ici nous ne terminons jamais le streaming et en plus les messages sont hard codés mais ne vous inquiétez pas cela va changer dans les chapitres suivants. Le but ici est de comprendre comment fonctionne chaque type de communication en gRPC et comment l’implémenter.

Revenons à nos tests POSTMAN :

Nous avons mis à jour notre fichier proto dans POSTMAN et avons donc maintenant accès à la méthode que nous venons de créer et si nous

lançons le streaming server vers le client, nous voyons bien arriver des messages toutes les secondes 😊



3. Connecter le News Bot au serveur de Stream

Afin de ne pas envoyer des messages hard codés du serveur vers la console Admin, nous allons ajouter les messages provenant du news bot dans une message queue.

En informatique, une message queue (file de messages) est une structure de données qui permet de stocker des messages dans une file d'attente jusqu'à ce qu'ils soient traités. En .NET Core, une message queue est une fonctionnalité qui permet aux applications de communiquer entre elles de manière asynchrone en utilisant des messages envoyés à travers une file d'attente.

Dans .NET Core, la classe principale pour la gestion des message queues est "System.Messaging.MessageQueue". Cette classe fournit une interface pour envoyer, recevoir et gérer des messages dans une file d'attente. Elle prend en charge les messages de différentes tailles et formats, y compris les messages binaires et XML.

Les message queues peuvent être utilisées pour implémenter une architecture basée sur les microservices, où chaque service peut publier et consommer des messages à partir de file d'attente partagées. Cela permet une communication asynchrone entre les services, ce qui peut améliorer les performances et la scalabilité de l'application.

Pour cela nous allons rajouter un fichier de classe MessageQueue contenant 3 méthodes :

- Une méthode pour rajouter des messages dans une queue
- Une méthode pour pouvoir lire le prochain message dans la queue
- Une méthode pour connaitre le nombre de messages dans la queue afin de ne pas lire des messages qui n'existent pas.

Dans un nouveau dossier nommé « Utils », créez une classe MessageQueue avec le contenu suivant :

```
public class MessageQueue
{
    private static readonly Queue<ReceivedMessage> _queue;

    static MessageQueue()  {
        _queue = new Queue<ReceivedMessage>();
    }

    public static void AddNewsToQueue(NewsFlash news)  {
        var msg = new ReceivedMessage();
        msg.Content = news.NewsItem;
        msg.User = "NewsBot";
        msg.MsgTime = Timestamp.FromDateTime(DateTime.UtcNow);
        _queue.Enqueue(msg);
    }

    public static ReceivedMessage GetNextMessage()  {
        return _queue.Dequeue();
    }

    public static int GetMessagesCount()  {
        return _queue.Count;
    }
}
```

Elle contient donc les 3méthodes que nous avons décrit aussi.

Nous allons maintenant modifier notre service pour qu'il utilise cette file de messages.

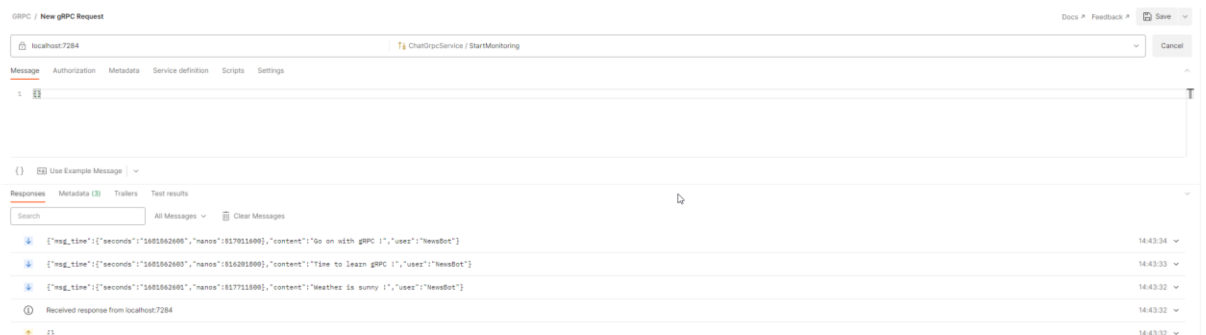
Nous allons donc rajouter le message de « newsFlash » à la queue lors de sa réception :

```
public override async Task<NewsStreamStatus>
SendNewsFlash(IAsyncStreamReader<NewsFlash> newsStream,
ServerCallContext context)
{
        while (await newsStream.MoveNext())
        {
         var news = newsStream.Current;
         MessageQueue.AddNewsToQueue(news);
         Console.WriteLine($"Here is new flash info : {news.NewsItem} at
{news.NewsTime.ToDateTime()}");
      }

      return new NewsStreamStatus {Success = true};
}
```

Et au lieu d’écrire des messages hard codés à destination de la console admin, nous allons lire les messages dans la message queue :

```
public override async Task StartMonitoring(Empty request,
IServerStreamWriter<ReceivedMessage> streamWriter,
          ServerCallContext context)
{
    while (true)
    {
       if (MessageQueue.GetMessagesCount() > 0)
       {
          await
streamWriter.WriteAsync(MessageQueue.GetNextMessage());
       }
       await Task.Delay(1000);
    }
}
```

Si nous lançons tous les projets et que nous lançons un appel à StartMonitoring à partir de POSTMAN alors nous allons recevoir les messages mis en file d’attente par le serveur à partir des messages du news bot :



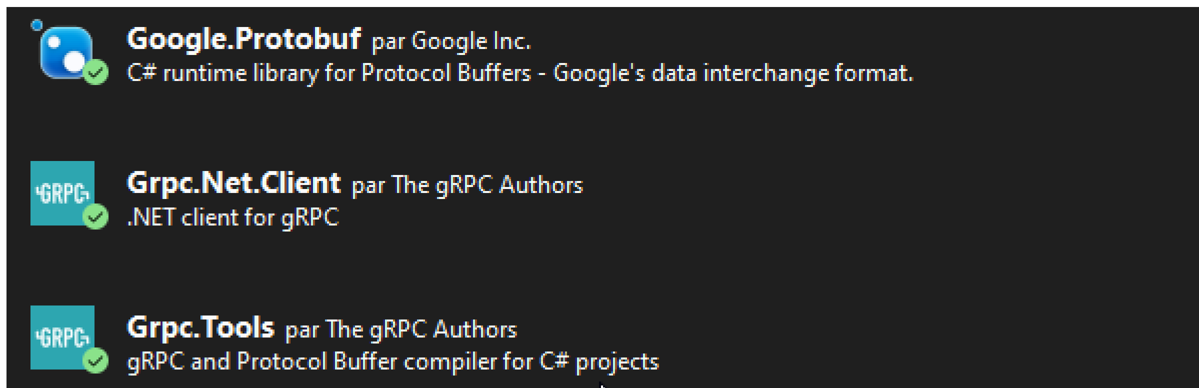
Effectivement, ce sont bien les messages envoyés par le news bot 😊

4. **Implémenter le Client**

Maintenant au lieu de faire des appels avec POSTMAN , nous allons créer un composant « Admin console » qui sera en écoute des données streamées par le serveur.

Encore une fois, nous allons ajouter une application console nommée « AdminConsole ».

Installons les packages nécessaires :



Rajoutons le fichier proto dans un dossier « Protos » à partir du fichier du serveur et la référence dans le fichier « csproj » :

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

Nous pouvons nettoyer le fichier proto et ne garder que la partie monitoring :

```
syntax = "proto3";

option csharp_namespace = "ChatServer.Protos";

package chatgrpc;

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


message ReceivedMessage {
	google.protobuf.Timestamp msg_time = 1 ;
	string content = 2;
	string user = 3;
}
```

```
service ChatGrpcService {
	rpc StartMonitoring(google.protobuf.Empty) returns (stream
ReceivedMessage);
}
```

Maintenant dans le fichier « Program.cs », nous allons pouvoir implémenter la logique d'écoute de streaming server :

```
using ChatServer.Protos;
using Google.Protobuf.WellKnownTypes;
using Grpc.Net.Client;

using var channel = GrpcChannel.ForAddress("https://localhost:7284");
var client = new ChatGrpcService.ChatGrpcServiceClient(channel);

Console.WriteLine("*** Admin Console started ***");
Console.WriteLine("Listening...");

using var call = client.StartMonitoring(new Empty() );
var cts = new CancellationTokenSource();
while (await call.ResponseStream.MoveNext(cts.Token))
{
   Console.WriteLine($"Message received from :
{call.ResponseStream.Current.User} at
{call.ResponseStream.Current.MsgTime.ToDateTime()} with content :
{call.ResponseStream.Current.Content}");
}
```

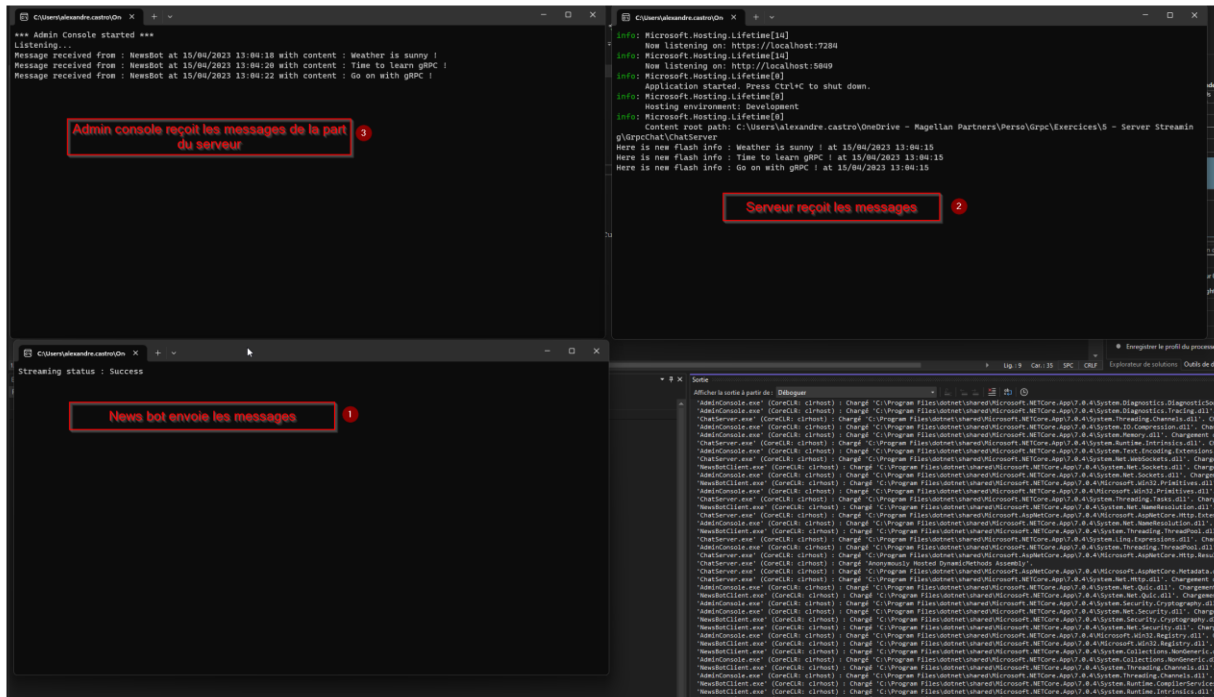
Nous avons rajouté une notion que nous verrons plus tard et qui concerne les demandes d'annulation en jaune dans le code ci-dessus 😊

Nous configurons maintenant le démarrage multi projets pour tout faire démarrer sauf le client de chat des utilisateurs car il n'est pas terminé et nous allons le terminer dans la section suivante.

Plusieurs projets de démarrage :

Projet	Action
ChatServer	Démarrer
NewsBotClient	Démarrer
ChatClient	Aucun
AdminConsole	Démarrer

Observons le résultat :



Nous avons donc le new bot qui envoie des news flashes au serveur.

Nous avons bien sûr le serveur lui-même qui reçoit ces messages et les met dans une file d'attente, puis il diffuse ces messages en continu vers la console d'administration et nous avons la console d'administration qui écoute ce flux de serveur et les affiche.

Et voilà comment le streaming côté serveur fonctionne avec gRPC.

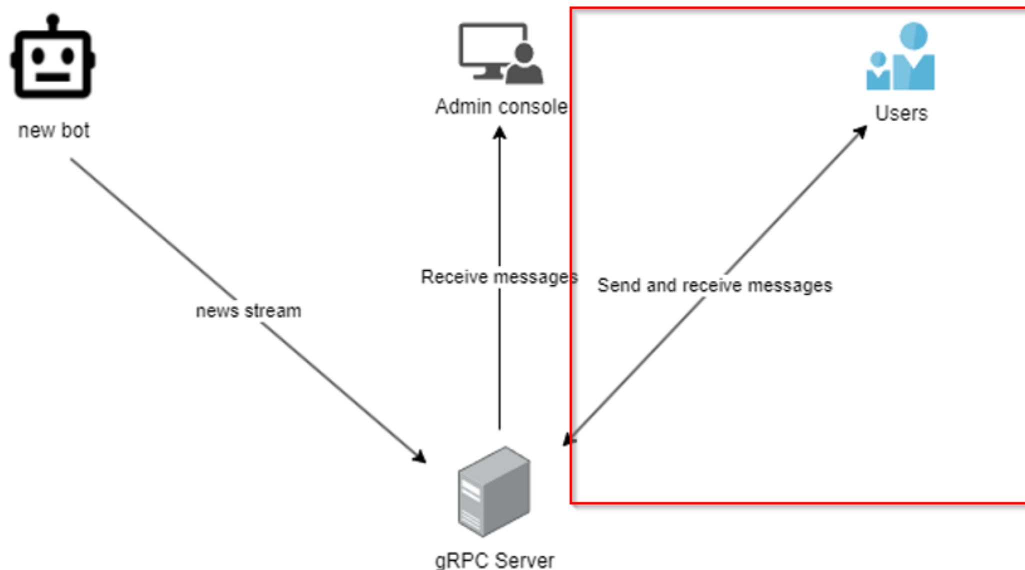
Le contenu de ce chapitre est disponible ici : « Exercices\5 - Server Streaming »

Chapitre 12 : Streaming Bi-Directionnel

1. Introduction

Il ne nous reste plus qu'un type de communication à voir et c'est le streaming bidirectionnel.

Ce type de communications va concerner le composant des utilisateurs afin d'envoyer et recevoir des messages :



2. Implémenter le streaming Bi- Directionnel sur le Server

Pour cela nous allons modifier le fichier proto du serveur afin d'avoir un nouveau type de messages :

```
message ChatMessage {
	google.protobuf.Timestamp msg_time = 1 ;
	string content = 2 ;
	string username = 3 ;
	string roomname = 4;
}
```

Ici nous avons tout ce dont nous avons besoin pour créer des messages et les recevoir car dans notre cas de streaming bidirectionnel , les ChatMessage seront streamés en entrée et en sortie.

Nous allons aussi modifier les messages d'enregistrement à une chat room et la réponse :

```
message RoomRegistrationRequest {
	string room_name = 1;
```

```
	string user_name = 2;
}

message RoomRegistrationResponse {
	bool joined_room = 1;
}
```

Et pour finir nous allons créer une méthode de streaming bidirectionnel :

```
rpc StartChatting(stream ChatMessage) returns (stream ChatMessage);
```

Si nous tentons de relancer notre serveur, il ne va plus fonctionner car nous avons changé des messages.

Nous allons donc corriger cela tout de suite.

Avant de corriger cela, nous allons gérer les connexions à une chat room dans une Message queue également.

Nous allons retourner dans notre dossier « Utils » et créer une classe « UserQueue » et implémenter la création de la queue pour un user.

Nous avons donc une liste de UserQueue car nous pouvons avoir plusieurs utilisateurs.

Nous avons la Queue de ReceivedMessage pour scruter les messages provenant de l'extérieur(l'admin console).

Dans les détails, nous avons des méthodes pour créer des queues pour chaque user, rajouter des messages dans la queue , récupérer les messages d'un utilisateur , récupérer le nombre de messages dans la queue de l'admin console et récupérer le prochain message de l'admin console.

Nous aurions pu utiliser des technologies comme RabbitMQ ou Azure ServiceBus mais pour notre usage, les message queues .NET suffiront largement.

```
public static class UsersQueue
    {
        private static List<UserQueue> _queues;
        private static Queue<ReceivedMessage> _adminQueue;

        static UsersQueue()  {
            _queues = new List<UserQueue>();
            _adminQueue = new Queue<ReceivedMessage>();
        }

        public static void CreateUserQueue(String room, String user)  {
```

```
            _queues.Add(new UserQueue(room, user));
        }

        public static void AddMessageToRoom(ReceivedMessage msg, string
room)  {
            // Add message only to users in this room
            foreach (var queue in _queues.Where(q=>q.Room==room))  {
                queue.AddMessageToQueue(msg);
            }
            _adminQueue.Enqueue(msg);
        }

        public static ReceivedMessage GetMessageForUser(string user)  {
            var userQueue = _queues.First(q => q.User == user);
            if (userQueue.GetMessagesCount()>0)  {
                return userQueue.GetNextMessage();
            }
            else  {
                return null;
            }
        }

        public static int GetAdminQueueMessageCount()  {
            return _adminQueue.Count;
        }

        public static ReceivedMessage GetNextAdminMessage()  {
            return _adminQueue.Dequeue();
        }
    }

    class UserQueue  {
        private Queue<ReceivedMessage> queue  { get; }
        public string Room { get; }
        public string User { get; }

        public UserQueue(string room, string user)  {
            Room = room;
            User = user;
            this.queue = new Queue<ReceivedMessage>();
        }

        public void AddMessageToQueue(ReceivedMessage msg)  {
            this.queue.Enqueue(msg);
        }

        public ReceivedMessage GetNextMessage()  {
            return this.queue.Dequeue();
```

```
        }

        public int GetMessagesCount()  {
            return this.queue.Count;
        }
    }
```

Maintenant nous pouvons modifier notre service 😉

```
  public override async Task<RoomRegistrationResponse>
RegisterToRoom(RoomRegistrationRequest request, ServerCallContext
context)
 {
        UsersQueue.CreateUserQueue(request.RoomName,
request.UserName);
        var resp = new RoomRegistrationResponse { JoinedRoom = true };
        return await Task.FromResult(resp);
 }
```

Et voila, notre méthode d’abonnement à une chat room crée la message queue pour un utilisateur donnée.

Comme vous avez pu le voir dans la classe « UsersQueue » , nous avons deux typologies de messages :

- ReceivedMessage
- ChatMessage

Nous allons donc devoir implémenter des méthodes privées dans notre service afin de les convertir car ils passent de la file de message admin vers la file de messages UsersQueue.

```
private ReceivedMessage ConvertToReceivedMessage(ChatMessage
chatMsg)  {
            var rcMsg = new ReceivedMessage
            {
                Content = chatMsg.Content,
                MsgTime = chatMsg.MsgTime,
                User = chatMsg.UserName
            };
            return rcMsg;
        }

        private ChatMessage ConvertToChatMessage(ReceivedMessage
rcMsg, string room)  {
```

```
            var chatMsg = new ChatMessage
            {
                Content = rcMsg.Content,
                MsgTime = rcMsg.MsgTime,
                UserName = rcMsg.User,
                RoomName = room
            };
            return chatMsg;
        }
```

Maintenant que nous avons créé toutes ces classes, nous allons pouvoir créer notre méthode StartChat dans le service qui sera une surcharge de la méthode que nous avons écrit dans notre fichier proto :

```
public override async Task
StartChatting(IAsyncStreamReader<ChatMessage> incomingStream,
IServerStreamWriter<ChatMessage> outgoingStream, ServerCallContext
context)  {

            // Wait for the first message to get the user name
            while (!await incomingStream.MoveNext())  {
                await Task.Delay(100);
            }

            string userName = incomingStream.Current.UserName;
            string room = incomingStream.Current.RoomName;
            Console.WriteLine($"User {userName} connected to room
{incomingStream.Current.RoomName}");

            // TO USE ONLY WHEN TESTING WITH POSTMAN -- WILL BE
REMOVED LATER
            UsersQueue.CreateUserQueue(room, userName);
            // END TEST END TEST END TEST

            // Get messages from the user
            var reqTask = Task.Run(async () =>
            {
                while (await incomingStream.MoveNext())
                {
                    Console.WriteLine($"Message received:
{incomingStream.Current.Content}");

UsersQueue.AddMessageToRoom(ConvertToReceivedMessage(incomingStr
eam.Current), incomingStream.Current.RoomName);
                }
            });
```

```
        // Check for messages to send to the user from users queue and
admin queue
        var respTask = Task.Run(async () =>
        {
            while (true)
            {
                var userMsg = UsersQueue.GetMessageForUser(userName);
                if (userMsg != null)
                {
                    var userMessage = ConvertToChatMessage(userMsg,
room);
                    await outgoingStream.WriteAsync(userMessage);
                }
                if (MessageQueue.GetMessagesCount() > 0)
                {
                    var news = MessageQueue.GetNextMessage();
                    var newsMessage = ConvertToChatMessage(news,
room);
                    await outgoingStream.WriteAsync(newsMessage);
                }

                await Task.Delay(200);
            }
        });

        // Keep the method running
        while (true)  {
            await Task.Delay(10000);
        }
    }
```

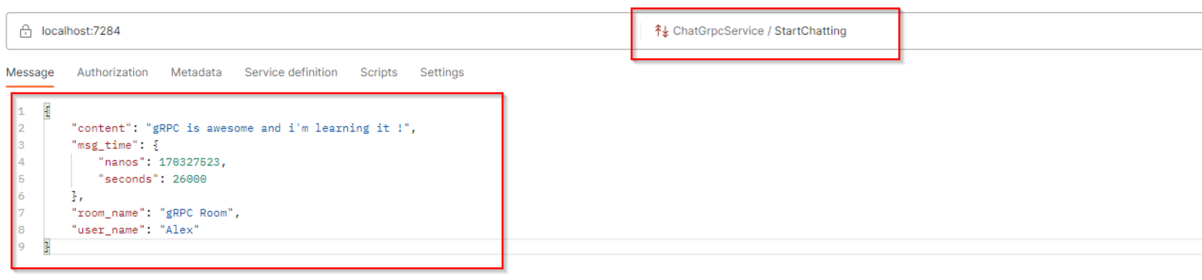
Nous avons fini notre implémentation du serveur et allons pouvoir le tester !

Encore une fois nous allons tout lancer sauf le client que , comme vous pouvez vous en douter, nous allons implémenter dans le chapitre suivant et sans l'admin console qui consommerait les messages de news sans que la méthode StartChatting puisse les consommer :

Plusieurs projets de démarrage :

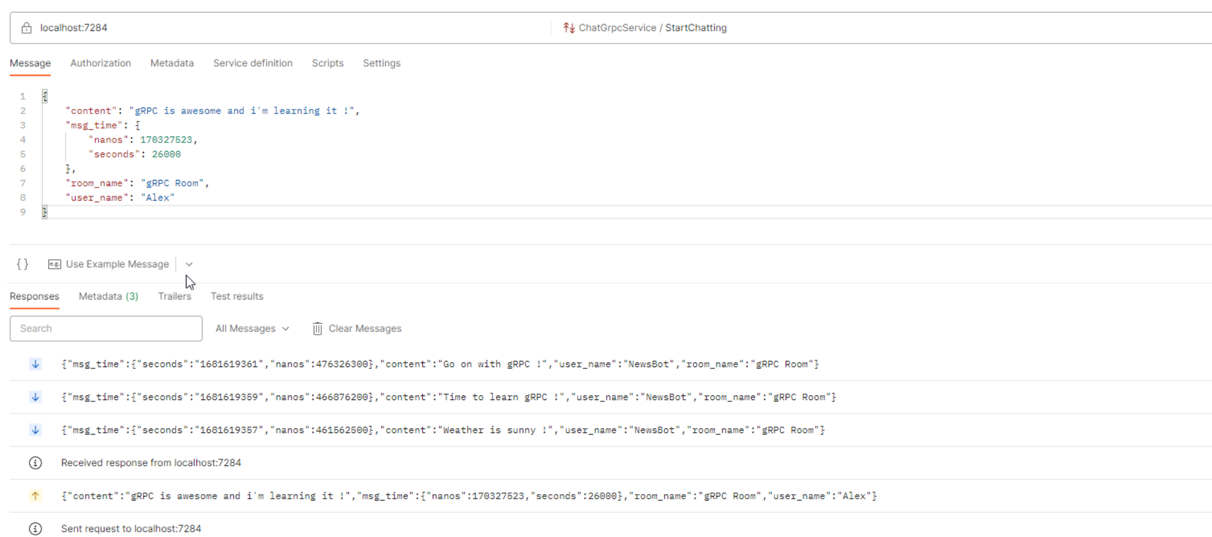
Projet	Action
ChatServer	Démarrer
NewsBotClient	Démarrer
AdminConsole	Aucun
ChatClient	Aucun

Lançons POSTMAN et mettons à jour notre fichier proto pour avoir accès à la méthode StartChatting :



POSTMAN a bien détecté que notre méthode est de type streaming bidirectionnel et voici le contenu que je vais envoyer.

Si tout se déroule comme il faut je devrais recevoir les messages du newsbot / admin console en retour.



En effet, tout fonctionne parfaitement !

3. Implémenter le Client

Maintenant que tout est prêt pour que nous puissions chatter, nous allons donc implémenter le client pour envoyer et recevoir des messages.

Nous allons donc reprendre le premier projet que nous avions créé pour le client de chat et le modifier.

Tout d'abord nous allons mettre à jour le fichier proto avec le fichier provenant du serveur :

```
syntax = "proto3";

option csharp_namespace = "ChatServer.Protos";

package chatgrpc;

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

message RoomRegistrationRequest {
	string room_name = 1;
	string user_name = 2;
}

message RoomRegistrationResponse {
	bool joined_room = 1;
}


message ChatMessage {
	google.protobuf.Timestamp msg_time = 1 ;
	string content = 2 ;
	string user_name = 3 ;
	string room_name = 4;
}

service ChatGrpcService {
	rpc RegisterToRoom(RoomRegistrationRequest) returns
(RoomRegistrationResponse);
	rpc StartChatting(stream ChatMessage) returns (stream
ChatMessage);
}
```

Nous n'aurons besoin que de nous enregistrer à une chat room et pouvoir envoyer et recevoir des messages.

Maintenant dans le fichier « Program.cs » nous allons devoir :

- Saisir un username
- Saisir une chat room
- Pouvoir recevoir des messages et les afficher
- Pouvoir envoyer des messages dans la chat room
- En plus de tout cela, il faudra bien replacer le curseurs lors de la réception et l'envoi des messages.

Le contenu est assez long concernant le fichier « Program.cs » :

```
using ChatServer.Protos;
using Google.Protobuf.WellKnownTypes;
using Grpc.Net.Client;

using var channel = GrpcChannel.ForAddress("https://localhost:7284");
var client = new ChatGrpcService.ChatGrpcServiceClient(channel);

Console.WriteLine("Welcome the the gRPC chat!");
Console.Write("Please type your user name: ");
var username = Console.ReadLine();

Console.Write("Please type the name of the room you want to join : ");
var room = Console.ReadLine();

Console.WriteLine($"Joining room {room}...");

try
{
    var joinResponse = client.RegisterToRoom(new
RoomRegistrationRequest() { RoomName = room, UserName = username
});
    if (joinResponse.JoinedRoom)
    {
        Console.WriteLine("Joined successfully!");
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine($"Error joining room {room}.");
        Console.ForegroundColor = ConsoleColor.Gray;
        Console.WriteLine("Press any key to close the window.");
        Console.Read();
        return;
    }
}
catch (Exception ex)
{
```

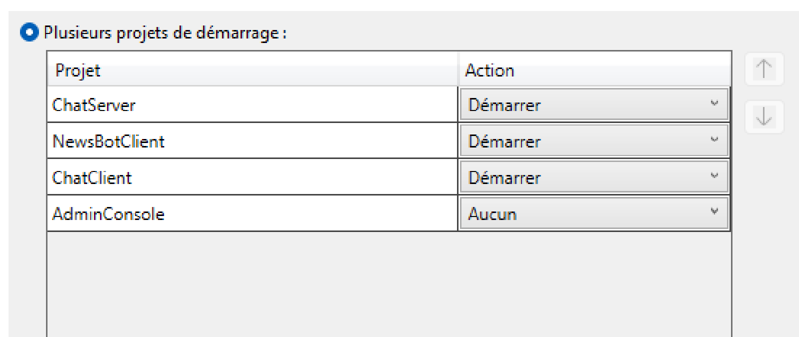
```
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"Error joining room {room}. Error:
{ex.Message}");
    Console.ForegroundColor = ConsoleColor.Gray;
    Console.WriteLine("Press any key to close the window.");
    Console.Read();
    return;
}

Console.WriteLine($"Press any key to enter the {room} room.");
Console.Read();
Console.Clear();

var call = client.StartChatting();
var cts = new CancellationTokenSource();

var promptText = "Type your message: ";
var row = 2;

var task = Task.Run(async () =>
{

    while (true)
    {
        if (await call.ResponseStream.MoveNext(cts.Token))
        {
            var message  = call.ResponseStream.Current;
            //cursor position to stay at same position when we receive a
message
            var left = Console.CursorLeft - promptText.Length;
            PrintMessage(message);
        }
        //wait for 0.5 sec before checking for new message
        await Task.Delay(500);
    }
});

Console.Write(promptText);
while (true)
{
    var input = Console.ReadLine();
    RestoreInputCursor();

    var reqMessage = new ChatMessage
    {
        Content = input,
        MsgTime = Timestamp.FromDateTime(DateTime.UtcNow),
        RoomName = room,
```

```
        UserName = username
    };
    await call.RequestStream.WriteAsync(reqMessage);
}

void PrintMessage(ChatMessage msg)
{
    var left = Console.CursorLeft - promptText.Length;
    Console.SetCursorPosition(0, row++);
    Console.Write($"{msg.UserName}: {msg.Content}");
    Console.SetCursorPosition(promptText.Length + left, 0);
}

void RestoreInputCursor()  {
    Console.SetCursorPosition(promptText.Length - 1, 0);
    Console.Write("                                        ");
    Console.SetCursorPosition(promptText.Length - 1, 0);
}
```

Il est temps maintenant de tester tout cela en prenant soin de commenter la ligne qui nous servait pour les tests POSTMAN dans le service sur la partie serveur :

```
// TO USE ONLY WHEN TESTING WITH POSTMAN -- WILL BE REMOVED LATER
//UsersQueue.CreateUserQueue(room, userName);
// END TEST END TEST END TEST
```

Voici ce que nous allons lancer :



Une fois que vous aurez suivi les prompts, vous pourrez chatter 😉

J’ai lancé deux clients sur la même room et voici le résultat !

```
Type your message:

NewsBot: Weather is sunny !
NewsBot: Time to learn gRPC !
NewsBot: Go on with gRPC !
Bot: Hello
Alex: Hi how are you ?
Bot: Fine thanks
```

```
Type your message:

Bot: Hello
Alex: Hi how are you ?
Bot: Fine thanks
```

```
Type your message:

NewsBot: Weather is sunny !
NewsBot: Time to learn gRPC !
NewsBot: Go on with gRPC !
Alex: i love gRPC
Bot: I love gRPC
Bot: are you learning gRPC
Alex: Yes and you ?
```

```
Type your message:

Bot: I love gRPC
Bot: are you learning gRPC
Alex: Yes and you ?
```

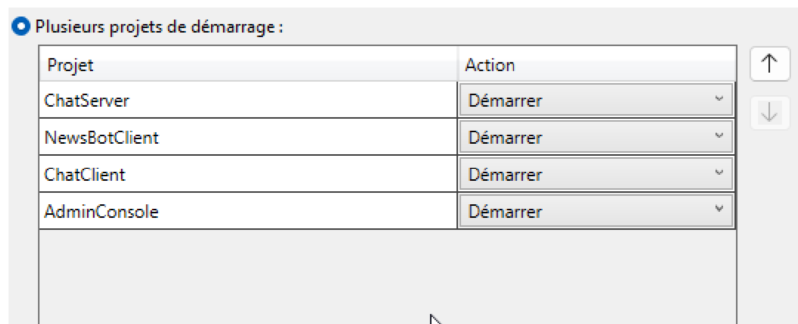
Nous allons maintenant modifier la console admin pour qu'elle puisse aussi recevoir des messages des utilisateurs.

```
public override async Task StartMonitoring(Empty request,
IServerStreamWriter<ReceivedMessage> streamWriter,
            ServerCallContext context)
        {
            while (true)
            {
                if (MessageQueue.GetMessagesCount() > 0)
                {
                    await
streamWriter.WriteAsync(MessageQueue.GetNextMessage());
                }

                if (UsersQueue.GetAdminQueueMessageCount() > 0)
                {
                    await
streamWriter.WriteAsync(UsersQueue.GetNextAdminMessage());
                }

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

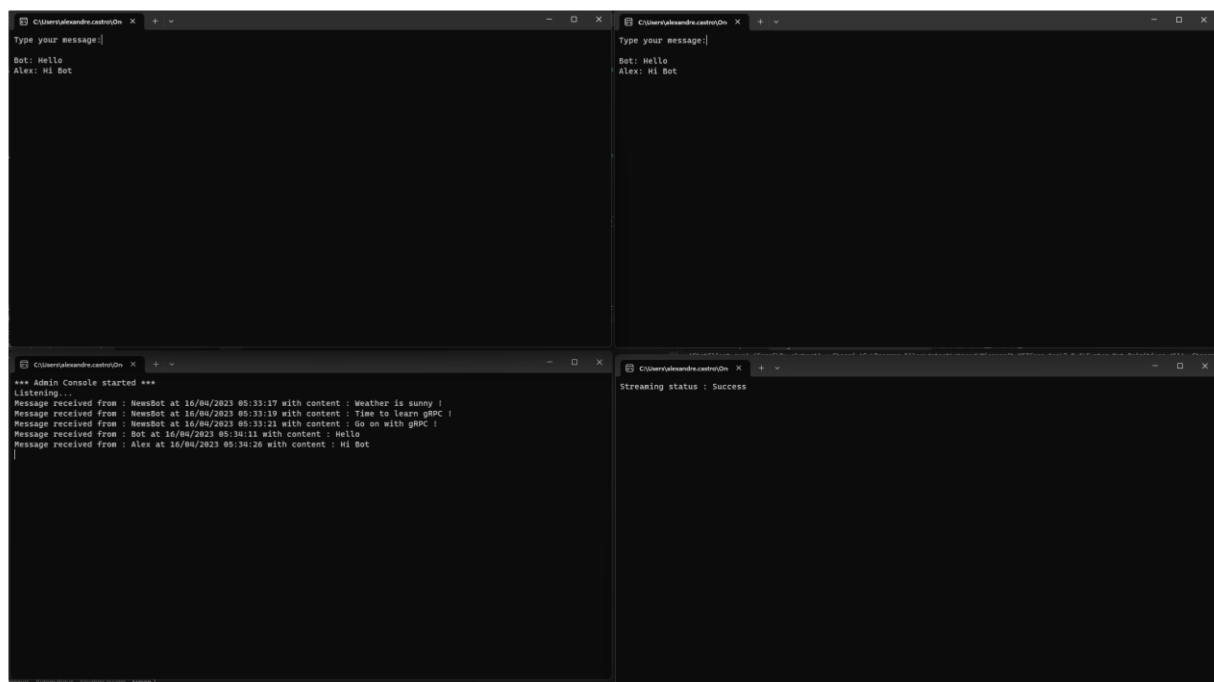
Nous avons donc rajouté les messages provenant des files d’attente des messages des utilisateurs !

Allons donc tester tout cela maintenant pour voir si les messages arrivent dans la console admin !

Nous allons donc tout démarrer pour voir cela :



Tout fonctionne donc parfaitement :



Il y a juste un souci, c’est que les news sont poussées uniquement vers la console admin car une fois que les messages sont consommés, ils n’existent plus. Je ne vais pas rentrer dans le détail du fonctionnement des message bus car ce n’est pas le but de ce guide.

Nous allons donc modifier cela et les pousser également dans les UsersQueue.

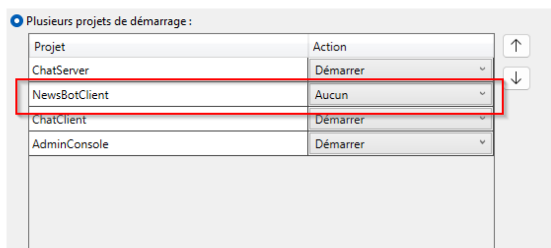
Dans la classe « UsersQueue », nous allons rajouter une méthode pour récupérer la liste des files de messages créées :

```
public static List<UserQueue> GetUsersQueues()
        {
            return _queues;
        }
```

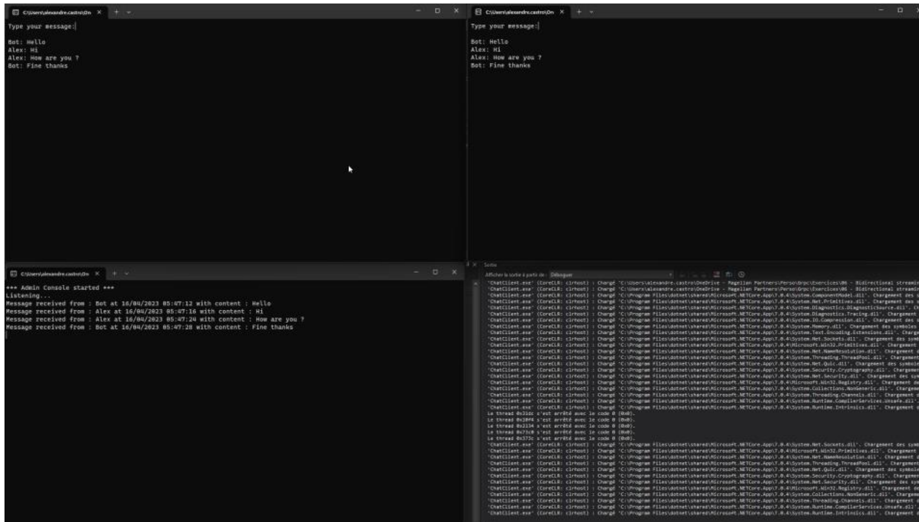
Et ensuite dans le service nous allons modifier la méthode SendNewsFlash pour qu'elle envoie aussi les messages dans les files de messages des utilisateurs !

```
public override async Task<NewsStreamStatus>
SendNewsFlash(IAsyncStreamReader<NewsFlash> newsStream,
ServerCallContext context)
        {
            while (await newsStream.MoveNext())
            {
                var news = newsStream.Current;
                MessageQueue.AddNewsToQueue(news);
                foreach (var queue in UsersQueue.GetUsersQueues())
                {
                    UsersQueue.AddMessageToRoom(new ReceivedMessage {
User = "NewBot" , Content = news.NewsItem , MsgTime =
news.NewsTime} , queue.Room);
                }
                Console.WriteLine($"Here is new flash info : {news.NewsItem}
at {news.NewsTime.ToDateTime()}");
            }

            return new NewsStreamStatus {Success = true};
        }
```

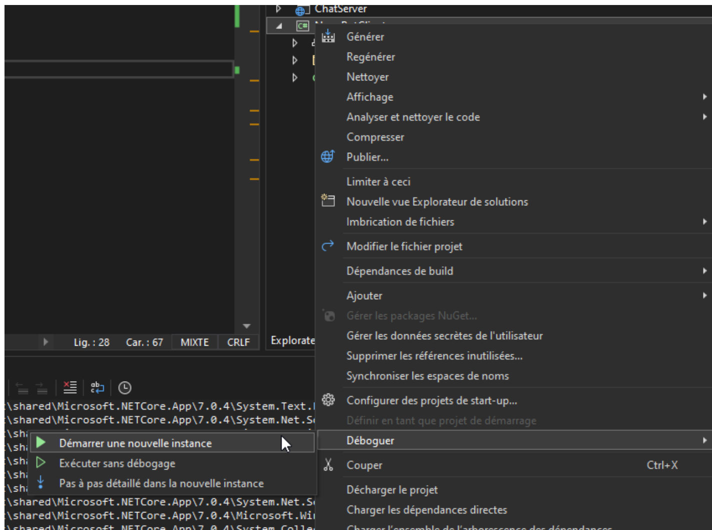
Afin de tester, nous n'allons pas lancer le news bot par défaut car sinon il n'enverra pas les messages vers les files de messages utilisateur car elles ne seront pas créées et nous le démarrerons « à la main » :

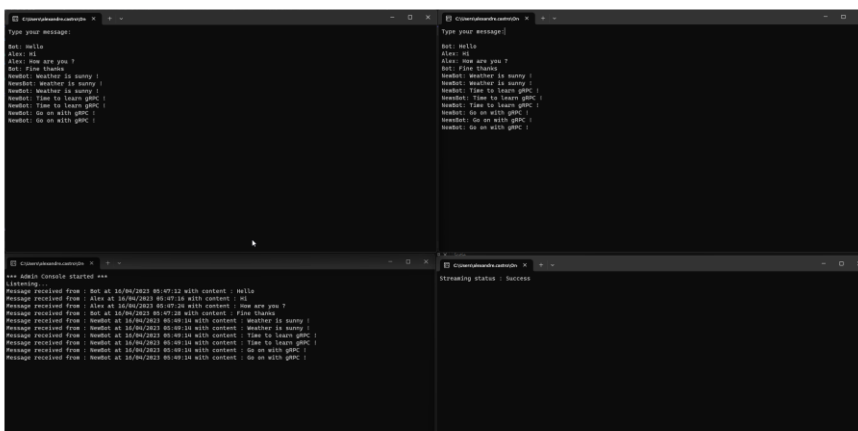


Tout fonctionne parfaitement sans le news bot :



Lançons maintenant le news bot pour voir si tout fonctionne :





Les messages sont doublés car il existe une queue pour chaque user donc il double forcément les messages mais le principe est là.

Nous avons fait le tour des types de communications gRPC et la solution complète se trouve ici : « Exercices\06 - Bidirectional streaming ».

Chapitre 13 : Sujets avancés

1. Introduction

Maintenant que nous avons fait le tour des concepts de base dans un projet qui utilise ceux-ci, nous allons pouvoir passer aux concepts avancés.

Nous allons donc voir ce qui concerne :

- Les deadlines
- La gestion des erreurs
- Les cancellation token
- La sécurisation de votre API et comment la consommer en utilisant une couche d’authentification

2. Deadlines

gRPC est un système de communication à distance qui permet à des applications d'échanger des données entre elles via des appels de méthode distant. Dans ce contexte, une "deadline" (échéance) se réfère à une durée maximale autorisée pour qu'un appel de méthode distant s'exécute avant qu'il ne soit considéré comme ayant échoué.

Plus précisément, la deadline représente le temps maximal que le client attendra une réponse de la part du serveur pour une requête donnée. Cette durée est généralement définie par le client qui émet la requête.

Il est important de noter que la notion de deadline est intégrée dans le protocole gRPC et que sa gestion est automatique. Lorsqu'un client émet une requête, il spécifie une durée de timeout pour cette requête. Si le serveur répond avant l'échéance, le client recevra la réponse normalement. En revanche, si la réponse du serveur n'est pas reçue avant l'échéance, le client recevra une erreur indiquant que la requête a expiré.

Les deadlines dans gRPC sont essentielles pour éviter les blocages prolongés des connexions en attente et pour garantir une réponse rapide aux clients. Les développeurs peuvent configurer les timeouts selon les besoins de leur application, en veillant à ce que la durée soit suffisante pour permettre le traitement de la requête par le serveur, tout en étant courte pour éviter des temps d'attente inutiles.

En somme, les deadlines dans gRPC permettent aux clients de définir un temps maximal pour recevoir une réponse de la part du serveur, évitant ainsi les temps d'attente prolongés et les blocages de connexions. Cette fonctionnalité est cruciale pour les applications qui ont besoin de performances élevées et de latences minimales.

3. Expérimenter les Deadlines

Afin d'expérimenter les deadlines, nous allons implémenter ce concept dans le client gRPC qui envoie et reçoit des messages.

Comme vous avez pu le constater nous avons déjà implémenté un « try catch » donc c'est ici que nous allons rajouter notre deadline sur la méthode qui permet de s'enregistrer à une chat room.

Si nous observons le code généré par les Grpc.Tools :

```
}
[global::System.CodeDom.Compiler.GeneratedCode(tool: "grpc_csharp_plugin", version: null)]
1 référence
public virtual global::ChatServer.Protos.RoomRegistrationResponse RegisterToRoom(global::ChatServer.Protos.RoomRegistrationRequest request, grpc::CallOptions options)
{
  return CallInvoker.BlockingUnaryCall(__Method_RegisterToRoom, host: null, options, request);
}
```

Nous avons accès à des CallOptions.

Voici la définition des CallOptions avec la définition de la deadline :

```
public struct CallOptions
{
    Metadata? headers;
    DateTime? deadline;
    CancellationToken cancellationToken;
    WriteOptions? writeOptions;
    ContextPropagationToken? propagationToken;
    CallCredentials? credentials;
    CallFlags flags;

    /// <summary>
    /// Creates a new instance of <c>CallOptions</c> struct.
    /// </summary>
    /// <param name="headers">Headers to be sent with the call.</param>
    /// <param name="deadline">Deadline for the call to finish. null means no deadline.</param>
    /// <param name="cancellationToken">Can be used to request cancellation of the call.</param>
    /// <param name="writeOptions">Write options that will be used for this call.</param>
    /// <param name="propagationToken">Context propagation token obtained from <see cref="ServerCallContext"/>.</param>
    /// <param name="credentials">Credentials to use for this call.</param>
    public CallOptions(Metadata? headers = null, DateTime? deadline = null, CancellationToken cancellationToken = default(CancellationToken),
                       WriteOptions? writeOptions = null, ContextPropagationToken? propagationToken = null, CallCredentials? credentials = null)
    {
        this.headers = headers;
        this.deadline = deadline;
        this.cancellationToken = cancellationToken;
        this.writeOptions = writeOptions;
        this.propagationToken = propagationToken;
        this.credentials = credentials;
        this.flags = default(CallFlags);
    }
```

Il nous suffit de modifier le code comme suit :

```
var joinResponse = client.RegisterToRoom(new RoomRegistrationRequest() { RoomName = room, UserName = username }, deadline : DateTime.UtcNow.AddSeconds(5));
```

Nous venons de rajouter une deadline à 5secondes donc si le serveur ne répond pas au bout de 5 secondes nous devrions avoir une exception.

Dans notre cas, il est impossible de déclencher cette exception naturellement.

A des fins des test, nous allons donc modifier le service pour qu'il crée une attente afin de déclencher cette exception :

```
public override async Task<RoomRegistrationResponse>
RegisterToRoom(RoomRegistrationRequest request, ServerCallContext
context)
{
    await Task.Delay(7000);
    UsersQueue.CreateUserQueue(request.RoomName,
request.UserName);
    var resp = new RoomRegistrationResponse { JoinedRoom = true };
    return await Task.FromResult(resp);
}
```

Nous créons donc un délai de 7secondes pour être large et relançons notre solution avec un point d'arrêt dans le « catch ».

```
catch (Exception ex)  ex = {"Status(StatusCode=\"DeadlineExceeded\", Detail=\"\")"}
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"Error joining room {room}. Error: {ex.Message}");
    Console.ForegroundColor = ConsoleColor.Gray;
    Console.WriteLine("Press any key to close the window.");
    Console.Read();
    return;
}
```

```
Welcome the the gRPC chat!
Please type your user name: Alex
Please type the name of the room you want to join : gRPC
Joining room gRPC...
Error joining room gRPC. Error: Status(StatusCode="DeadlineExceeded", Detail="")
Press any key to close the window.
```

Nous héritons d'une exception de type DeadlineExceeded » qui conforme que la mécanique fonctionne.

4. Gestion des erreurs

Maintenant passons à la gestion des erreurs.

La gestion des erreurs en gRPC .NET est importante car elle permet de détecter et de corriger les erreurs dans le système, ce qui permet de garantir le bon fonctionnement de l'application.

Il existe deux types d'erreurs en gRPC .NET : les erreurs de transport et les erreurs de service. Les erreurs de transport se produisent lorsqu'il y a une erreur dans la communication entre le client et le serveur, par exemple lorsque la connexion est perdue ou lorsque le délai d'attente est dépassé. Les erreurs de service se produisent lorsque le serveur ne peut pas traiter la demande du client en raison d'une erreur interne.

Pour gérer les erreurs de transport, gRPC utilise les mécanismes de contrôle de flux et de fenêtrage de flux du protocole HTTP/2.0. Ces

mécanismes permettent de détecter les erreurs de communication et de les signaler au client.

Pour gérer les erreurs de service, gRPC utilise des codes d'erreur standardisés qui sont définis dans le protocole. Ces codes d'erreur permettent de détecter et de signaler les erreurs qui se produisent dans le service. Les codes d'erreur standardisés comprennent des codes tels que "INVALID_ARGUMENT", "NOT_FOUND", "ALREADY_EXISTS", "UNIMPLEMENTED", "UNAVAILABLE", "INTERNAL", etc.

En outre, gRPC permet également aux développeurs de définir leurs propres codes d'erreur personnalisés, en fonction des besoins de leur application. Les codes d'erreur personnalisés peuvent être utilisés pour signaler des erreurs spécifiques à l'application, qui ne sont pas couvertes par les codes d'erreur standardisés.

En résumé, la gestion des erreurs en gRPC .NET est essentielle pour garantir le bon fonctionnement de l'application. gRPC utilise des mécanismes standardisés pour détecter et signaler les erreurs de transport et de service, ainsi que la possibilité de définir des codes d'erreur personnalisés pour répondre aux besoins spécifiques de l'application.

Si vous souhaitez consulter la liste des codes statut disponibles, vous pouvez consulter le lien ci-dessous :

https://grpc.io/docs/guides/error/#error-status-codes

5. Annuler des requêtes

En gRPC .NET, la cancellation de requête est gérée en utilisant des objets CancellationToken. Lorsqu'un client envoie une requête à un serveur, il peut fournir un CancellationToken qui est utilisé pour suivre l'état de la requête. Si le client décide d'annuler la requête, il peut appeler la méthode Cancel() sur l'objet CancellationToken, ce qui entraîne l'annulation de la requête en cours de traitement.

Le serveur peut également annuler une requête en cours de traitement en utilisant un CancellationTokenSource. Lorsqu'un serveur commence à traiter une requête, il peut créer un CancellationTokenSource qui est utilisé pour suivre l'état de la requête. Si le serveur décide d'annuler la requête, il peut appeler la méthode Cancel() sur l'objet

CancellationTokenSource, ce qui entraîne l'annulation de la requête en cours de traitement.

Lorsqu'une requête est annulée, le client reçoit une exception RpcException qui contient le code d'erreur Canceled. Le serveur peut également détecter l'annulation de la requête en vérifiant l'état du CancellationToken ou du CancellationTokenSource, et peut agir en conséquence en libérant les ressources utilisées pour traiter la requête.

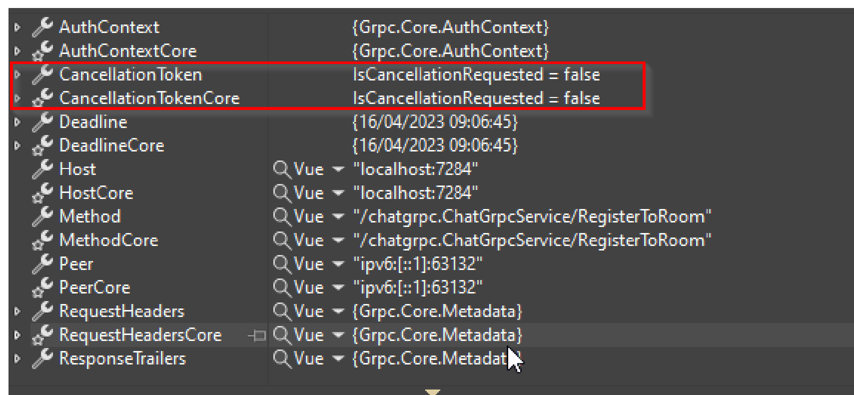
En résumé, la cancellation de requête est une fonctionnalité importante de gRPC qui permet à un client d'annuler une requête en cours de traitement par un serveur. En gRPC .NET, la cancellation de requête est gérée en utilisant des objets CancellationToken et CancellationTokenSource, et l'annulation de la requête entraîne la génération d'une exception RpcException avec le code d'erreur Canceled.

6. Implémenter les Request Cancellation

Nous allons implémenter un exemple pour que vous puissiez à votre tour l'utiliser et le mettre en place dans vos projets.

Dans votre service gRPC dans le projet « serveur » , nous avons rajouté plusieurs paramètres à nos requêtes dont ServerCallContext.

Ce type de données contient des métadonnées dont la demande d'annulation de la requête :



Nous allons maintenant implémenter dans le client une fonctionnalité qui va permettre d'annuler le streaming bidirectionnel ou en d'autres termes, quitter la chat room.

Imaginons que si je tape « QUIT » à la place d'un message, je quitte la chat room.

Pour cela allons dans le projet client et le fichier « Program.cs » :

```
if (input == "QUIT")
```

```
    {
        cts.Cancel();
        Console.WriteLine("You left the chat room!");
    }
```

Nous rajoutons seulement ce bout de code pour que lorsque nous tapons « QUIT », cela envoie une demande d’annulation.

Nous devons également gérer l’exception pour qu’elle ne casse pas l’application lorsque nous demandons à quitter la chat room.

Pour cela nous allons rajouter un « try catch » qui va « catcher » spécifiquement l’exception que nous ne souhaitons pas renvoyer :

```
while (true)
    {
        try
        {
            if (await call.ResponseStream.MoveNext(cts.Token))
            {
                var message  = call.ResponseStream.Current;
                //cursor position to stay at same position when we receive a
message
                var left = Console.CursorLeft - promptText.Length;
                PrintMessage(message);
            }
            //wait for 0.5 sec before checking for new message
            await Task.Delay(500);
        }
        catch (Grpc.Core.RpcException e)
        {
            if (e.StatusCode == StatusCode.Cancelled)
            {
                Console.WriteLine("Chat Cancelled");
                break;
            }
        }

    }
```

Et dans le service sur le serveur :

```
  public override async Task StartMonitoring(Empty request,
IServerStreamWriter<ReceivedMessage> streamWriter,
            ServerCallContext context)
        {
            while (true)
            {
                if (context.CancellationToken.IsCancellationRequested)
                {
```

```
                return;
            }
            if (MessageQueue.GetMessagesCount() > 0)
            {
                await
streamWriter.WriteAsync(MessageQueue.GetNextMessage());
            }

            if (UsersQueue.GetAdminQueueMessageCount() > 0)
            {
                await
streamWriter.WriteAsync(UsersQueue.GetNextAdminMessage());
            }

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

Tout fonctionne parfaitement !

```
Type your message:You left the chat room!
Chat Cancelled
Alex: Hello
Alex: Is there anybody here ?
Alex: Well I leave !
```

7. Authentification

L'authentification est une fonctionnalité importante de gRPC qui permet de sécuriser les communications entre les clients et les serveurs. En gRPC .NET, l'authentification est prise en charge en utilisant le protocole OAuth2, qui permet aux clients de s'authentifier auprès du serveur en utilisant un jeton d'accès.

La première étape de l'authentification en gRPC .NET consiste à configurer le serveur pour qu'il vérifie l'identité des clients qui accèdent aux ressources. Le serveur peut être configuré pour utiliser plusieurs mécanismes d'authentification, tels que l'authentification basée sur les certificats SSL, l'authentification basée sur les informations d'identification de l'utilisateur ou l'authentification basée sur les jetons d'accès OAuth2.

Pour l'authentification basée sur les jetons d'accès OAuth2, le serveur est configuré pour vérifier le jeton d'accès fourni par le client. Le serveur peut utiliser une bibliothèque de validation de jetons pour vérifier que le jeton d'accès est valide et qu'il a été émis par une autorité de confiance. Si le

jeton est valide, le serveur peut accorder l'accès aux ressources demandées par le client.

Du côté du client, l'authentification est effectuée en envoyant le jeton d'accès avec chaque requête RPC. Le client peut obtenir le jeton d'accès en utilisant un processus d'authentification préalable, tel qu'une page de connexion, ou en utilisant un processus d'authentification automatique, tel que le flux d'autorisation OAuth2.

En résumé, l'authentification en gRPC .NET est prise en charge en utilisant le protocole OAuth2, qui permet aux clients de s'authentifier auprès du serveur en utilisant un jeton d'accès. Le serveur est configuré pour vérifier l'identité des clients qui accèdent aux ressources, tandis que le client envoie le jeton d'accès avec chaque requête RPC.

8. Introduction à OAuth

Le mécanisme d'authentification JWT (JSON Web Token) est un moyen de transmettre de manière sécurisée des informations entre deux parties, généralement pour l'authentification. Les JWT sont utilisés pour s'assurer que les informations transmises sont fiables et n'ont pas été modifiées en cours de route.

Voici comment cela fonctionne :

1. L'application cliente envoie une demande d'authentification au serveur avec les informations d'identification de l'utilisateur, comme le nom d'utilisateur et le mot de passe.

2. Si les informations d'identification sont valides, le serveur génère un JWT et l'envoie à l'application cliente. Un JWT est une chaîne de caractères encodée qui contient des informations sur l'utilisateur, comme son nom d'utilisateur et son rôle dans l'application. Le JWT est généralement composé de trois parties :

- Un en-tête : qui spécifie le type de JWT (typ) et l'algorithme utilisé pour signer le JWT (alg).

- Un corps : qui contient les informations sur l'utilisateur, appelées "claims". Les claims peuvent être de différents types, comme l'identifiant de l'utilisateur (sub), le nom de l'utilisateur (name) ou le rôle de l'utilisateur dans l'application (role).

- Une signature : qui est générée en codant l'en-tête et le corps du JWT à l'aide de l'algorithme spécifié dans l'en-tête, et en utilisant une clé secrète connue uniquement par le serveur.

3. L'application cliente stocke le JWT et l'envoie avec chaque demande future au serveur.

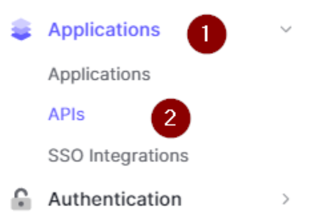
4. Lorsque le serveur reçoit une demande avec un JWT, il décode le JWT et vérifie sa signature avec la clé secrète. Si la signature est valide, cela signifie que le JWT n'a pas été modifié en cours de route et que les informations qu'il contient sont fiables. Le serveur peut alors autoriser l'accès à l'application ou à certaines fonctionnalités de l'application en fonction des claims du JWT.

Il est important de noter que les JWT ne sont pas cryptés, seulement signés. Cela signifie que le contenu du JWT est accessible en clair et peut être lu par n'importe qui qui a accès au JWT. Par conséquent, les JWT ne doivent pas être utilisés pour stocker des informations sensibles, comme des mots de passe ou des informations de carte de crédit. Ils sont plutôt utilisés pour vérifier l'identité de l'utilisateur et ses autorisations d'accès

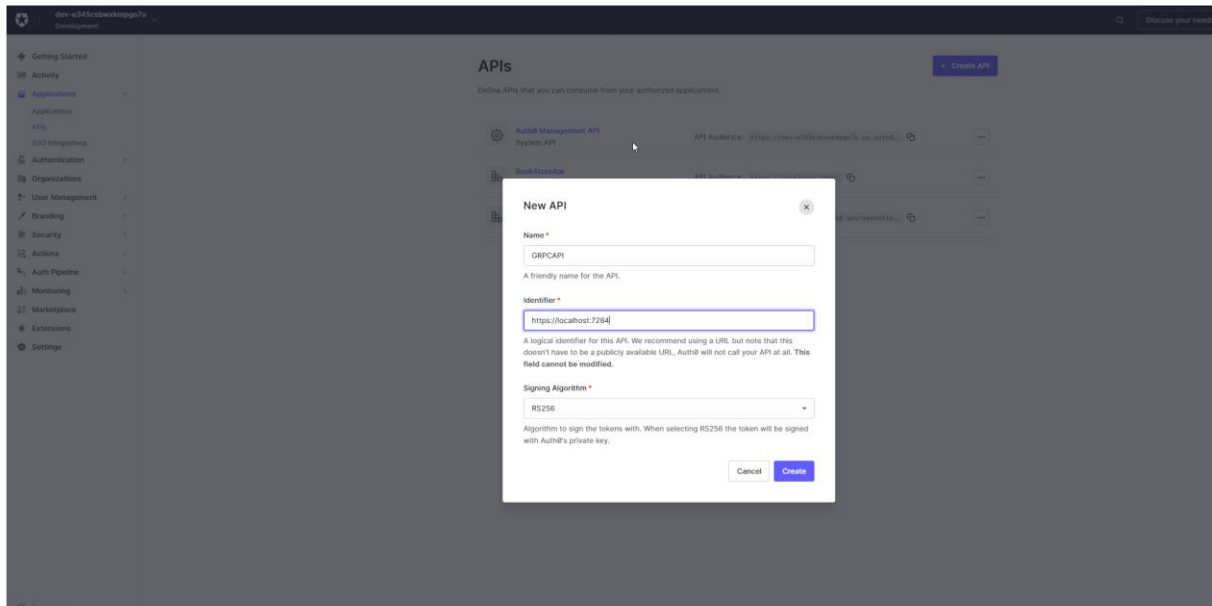
9. Ajouter de l'Authentification à l'application de Chat

Nous allons donc maintenant sécuriser notre API avec le fournisseur d'identité Auth0 https://auth0.com/

Ce fournisseur contient un tiers gratuit donc inscrivez vous et accédez au dashboard afin de créer votre application :



Ensuite créez votre API dont l' « identifier » est arbitraire mais sera nécessaire pour configurer votre authentification et autorisation.



Nous allons maintenant pouvoir implémenter le flux de sécurisation des méthodes de service par authentification.

Pour cela dans le fichier « Program.cs » de l’application serveur, ajoutons :

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options =>
{
    options.Authority = builder.Configuration["Auth0:Domain"];
    options.Audience = builder.Configuration["Auth0:Audience"];
});
builder.Services.AddAuthorization();
```

et en bas du fichier juste avant app.MapGrpcService<ChatGrpcService>(); :

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

Je ne vais pas rentrer dans les détails des fichiers de configuration mais vous allez devoir rajouter dans votre fichier appsettings.json , la configuration Auth0 de l’application que vous venez de créer :

```
"Auth0": {
  "Domain": "https://dev-e345csbwxkmpgo7u.us.auth0.com/",
  "Audience": "https://localhost:7284"
}
```

Maintenant que le setup est effectué, protégeons la méthode qui permet d'accéder à une chat room :

```
        [Authorize]
public override async Task<RoomRegistrationResponse>
RegisterToRoom(RoomRegistrationRequest request, ServerCallContext
context)
{
    UsersQueue.CreateUserQueue(request.RoomName,
request.UserName);
    var resp = new RoomRegistrationResponse { JoinedRoom = true };
    return await Task.FromResult(resp);
}
```

Si nous faisons un test entre le client et le serveur, et que nous essayons de nous connecter :

```
catch (Exception ex)   ex = {"Status(StatusCode=\"Unauthenticated\", Detail=\"Bad gRPC response. HTTP status code: 401\")"}
{
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"Error joining room {room}. Error: {ex.Message}");
    Console.ForegroundColor = ConsoleColor.Gray;
    Console.WriteLine("Press any key to close the window.");
    Console.Read();
    return;
}
```

```
Welcome the the gRPC chat!
Please type your user name: Alex
Please type the name of the room you want to join : rpc
Joining room rpc...
Error joining room rpc. Error: Status(StatusCode="Unauthenticated", Detail="Bad gRPC response. HTTP status code: 401")
Press any key to close the window.
```

Tout fonctionne comme prévu et notre endpoint est protégé.

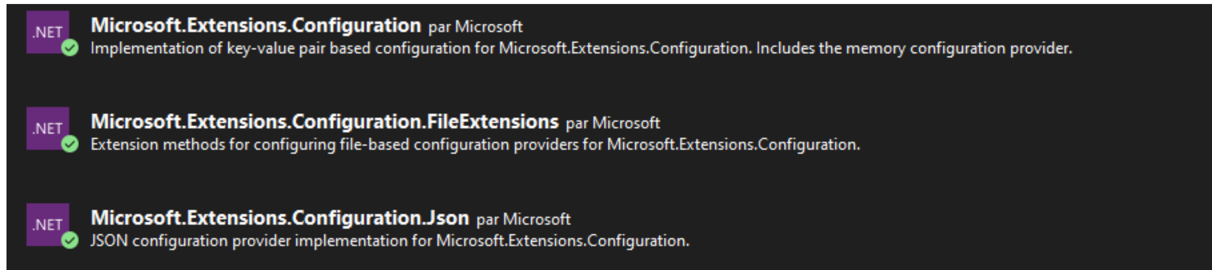
Maintenant que le serveur est protégé, il va falloir que le client s'authentifie et envoie un bearer token afin de pouvoir accéder aux méthodes.

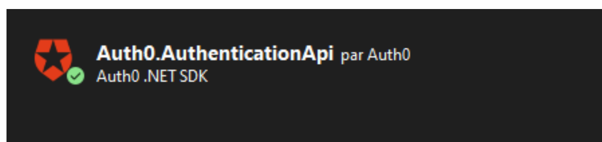
Nous allons maintenant créer une application de type « machine to machine » qui accèdera à notre client et nous lui donnerons accès à notre Api créée précédemment :

Rajoutons ensuite un fichier json de configuration dans l'application client contenant vos settings fournies lors de la création de l'application « machine to machine »:

```json
{
  "Auth0": {
    "Domain": "dev-e345csbwxkmpgo7u.us.auth0.com",
    "Audience": "https://localhost:7284",
    "ClientId": "HXXdBMw9MzgHFyJvtmDd3dXK78OG93Xo",
    "ClientSecret": 
  }
}
```

Rajoutons ensuite ces 4 packages pour gérer la configuration et le fichier json et l'authentification:





Rajoutons cette méthode pour gérer la configuration :

```
static IConfiguration GetAppSettings()
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json");

    return builder.Build();
}
```

Et celle-ci pour l'authentification de notre client :

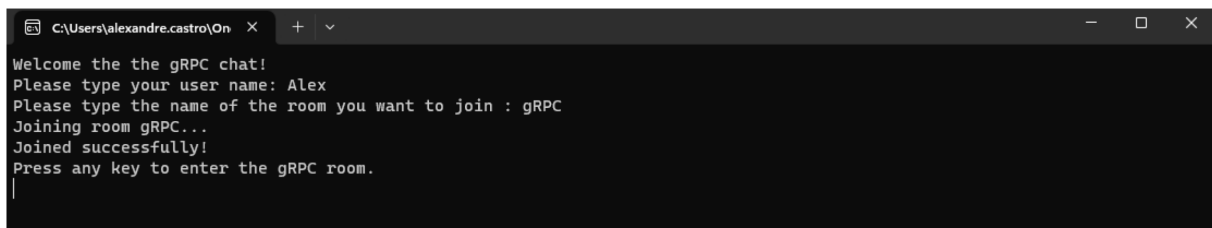
```
static async Task<string> GetAccessToken()
{
    var appAuth0Settings = GetAppSettings().GetSection("Auth0");
    var auth0Client = new
AuthenticationApiClient(appAuth0Settings["Domain"]);
    var tokenRequest = new ClientCredentialsTokenRequest()
    {
        ClientId = appAuth0Settings["ClientId"],
        ClientSecret = appAuth0Settings["ClientSecret"],
        Audience = appAuth0Settings["Audience"]
    };
    var tokenResponse = await auth0Client.GetTokenAsync(tokenRequest);
```

```
    return tokenResponse.AccessToken;
}
```

Nous allons maintenant pouvoir l’utiliser dans notre code d’appel lors de l’accès à la chat room !

```
var accessToken = await GetAccessToken();
var headers = new Metadata
    {
        { "Authorization", $"Bearer {accessToken}" }
    };

    var joinResponse = client.RegisterToRoom(new
RoomRegistrationRequest() { RoomName = room, UserName = username
}, headers ,deadline : DateTime.UtcNow.AddSeconds(5));
```

Vous pouvez maintenant tester de joindre une chat room :



Et voila notre service est sécurisé et nous pouvons appeler à partir du client en toute sécurité.

Dans le cas ou nous souhaitons créer un canal sécurisé en passant le bearer token par défaut lors de tous les appels, nous pouvons aller plus loin et créer une méthode qui va instancier un canal avec le header contenant le token bearer par défaut.

Rajoutons donc cette méthode à la fin du fichier :

```
async static Task<GrpcChannel> CreateAuthorizedChannel(string
address)
{
    var accessToken = await GetAccessToken();

    var credentials = CallCredentials.FromInterceptor((context, metadata)
=>
    {
        if (!string.IsNullOrEmpty(accessToken))
        {
            metadata.Add("Authorization", $"Bearer {accessToken}");
        }
        return Task.CompletedTask;
    });
```

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

Cette méthode nous servira à créer le canal sécurisé avec le bearer en remplaçant : `using var channel = GrpcChannel.ForAddress("https://localhost:7284");`

Par :

```
var channel = await CreateAuthorizedChannel("https://localhost:7284");
```

et en supprimant le code que nous avions rajouté précédemment pour gérer le token dans l'appel avec le client.

Voila nous avons maintenant une application serveur gRPC sécurisée et une application client qui fait ses appels de manière sécurisée également.

Le contenu de ce chapitre est disponible ici : « Exercices\07 - Advanced topics »

Chapitre 14 : gRPC dans le navigateur

1. Introduction

Rappelez-vous qu'à l'instar des API REST et GraphQL, gRPC n'est pas pris en charge nativement dans un navigateur.

Il peut facilement être appelé directement depuis un navigateur, et cela signifie que gRPC est principalement utilisé dans les applications mobiles natives et les appels back-end entre les services.

Maintenant, quelle est la raison de cela?

Pourquoi est-il si compliqué de mettre en œuvre gRPC dans le navigateur?

Eh bien, il y a certaines lacunes dans la prise en charge de gRPC par le navigateur.

Tout d'abord, il n'y a aucun moyen de forcer l'utilisation de http/2 dans le navigateur.

De plus, il n'y a aucun moyen d'attacher des trailers ou des footers spécifiques à gRPC à la fin de chaque demande et réponse.

Et ces footers sont requis par gRPC afin qu'il puisse transmettre la demande.

Enfin, il n'y a aucun moyen de forcer l'utilisation d'un proxy pour la traduction de gRPC sur le web vers les réponses gRPC http/2.

Est-ce que cela signifie qu'il n'y a aucun moyen d'utiliser gRPC dans le navigateur?

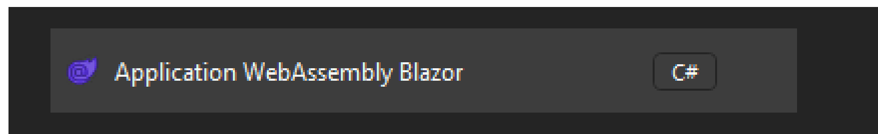
Eh bien, heureusement, nous pouvons utiliser gRPC dans le navigateur et il existe divers contournements pour ces limitations !

Nous allons donc créer une application Blazor afin de faire des appels gRPC vers notre serveur car ce serait dommage de rester sur des applications console.

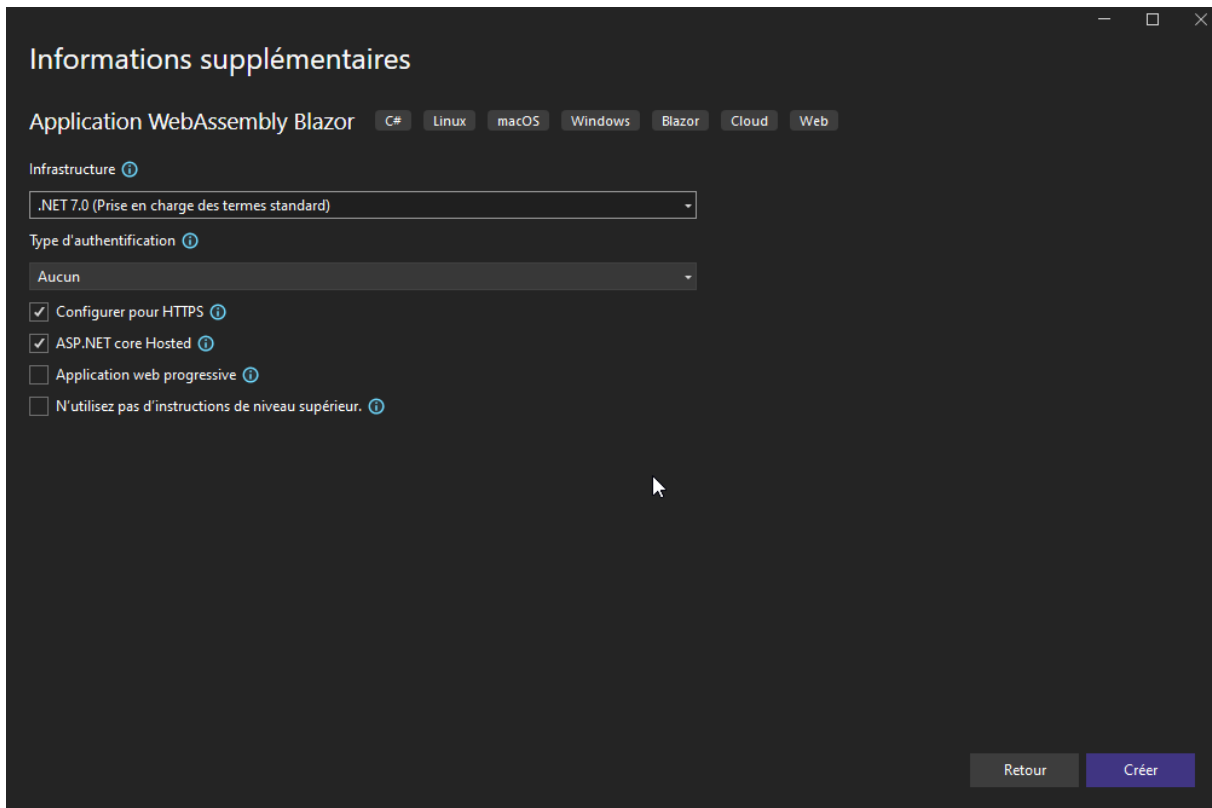
2. Utiliser gRPC-Web avec Blazor 💥💖💖💓💞💕

Nous allons passer à la partie qui nous intéresse maintenant c'est-à-dire appeler notre serveur à partir d'une application Web.

Nous allons donc rajouter un nouveau projet à notre solution :

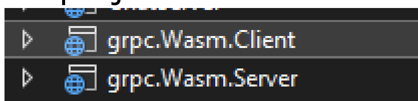


Avec le setup suivant :

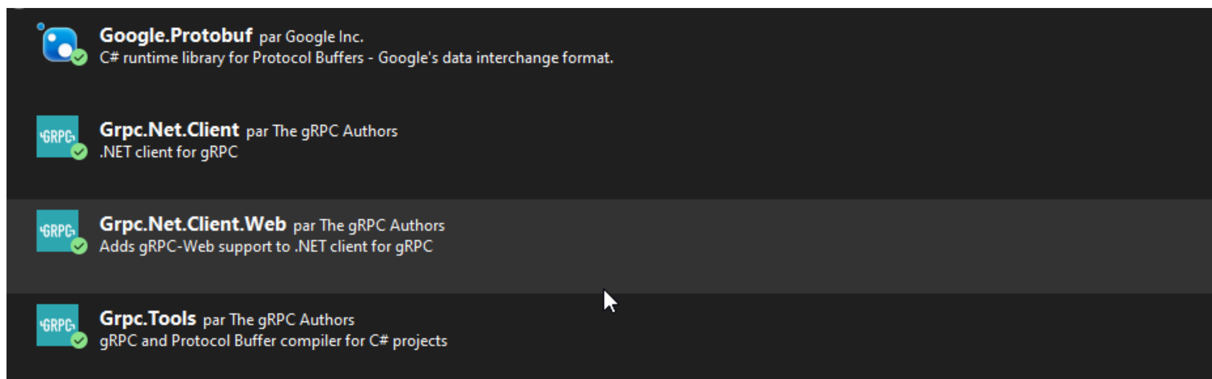


En effet, l'option ASP .NET core Hosted nous permettra d'avoir le client et le serveur qui communiquent et cette configuration va générer deux projets :

- Un projet client
- Un projet serveur

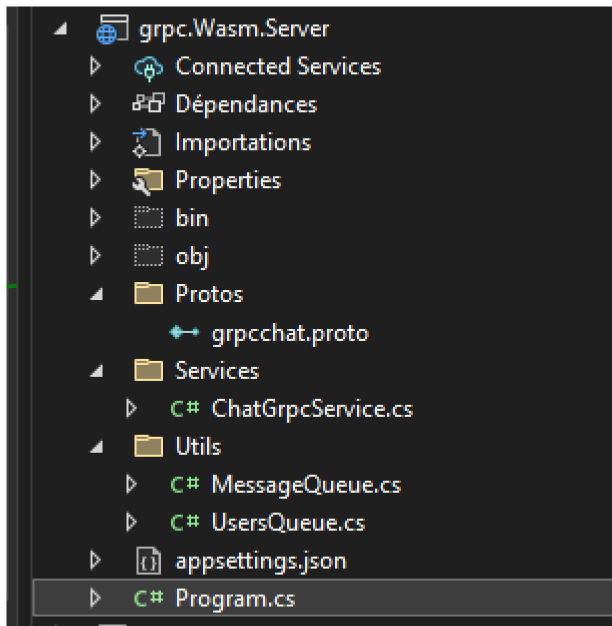


Commençons par installer les 4 packages Nuget suivants sur la partie client:



Nous allons donc commencer par le serveur et pour cela nous allons commencer par rajouter le package suivant :

Nous allons donc reproduire ce que nous avions sur notre serveur mais sur notre projet Blazor WASM Serveur :



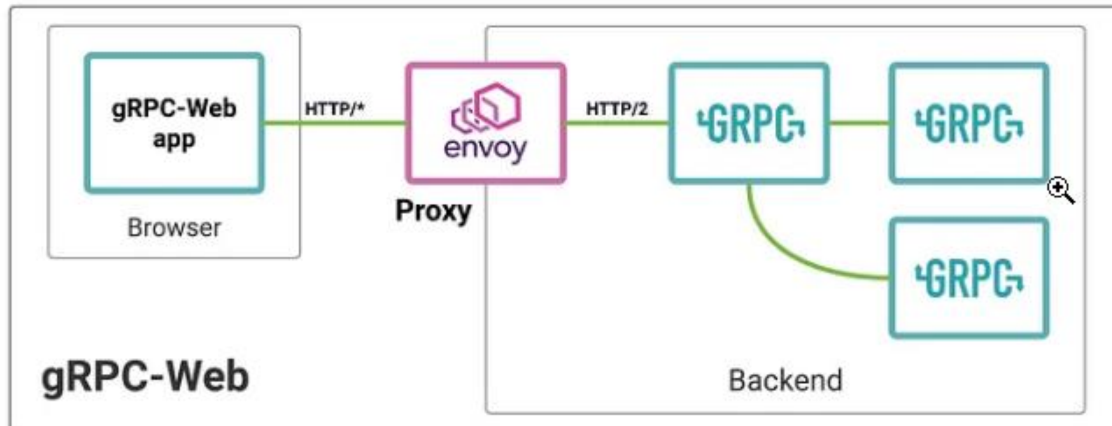
Ensuite allons dans le fichier « Program.cs » et nous allons devoir rajouter de la configuration pour gRPC Web :

Rajoutons ces lignes :

```
builder.Services.AddGrpc();

app.MapGrpcService<ChatGrpcService>().EnableGrpcWeb();

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

Cette ligne active gRPC Web par défaut sur tous les endpoints ! Quelle magie et merci ASP .NET Core 😊

Cela veut surtout dire que ASP .NET Core rajoute un proxy (Envoy) qui va traduire les appels http/1.1 en appels http/2 vers le serveur et va faire de même pour les réponses dans l'autre sens.



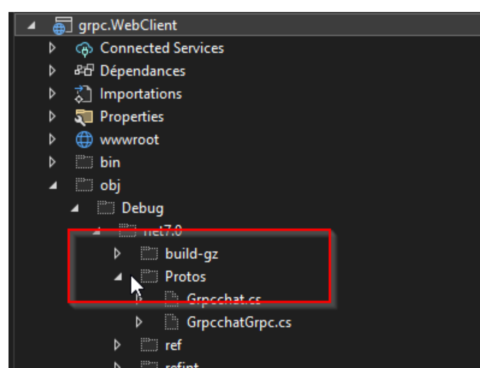
Comme nous l'avons vu précédemment, nous allons recevoir des appels http/1.1 donc nous devons les accepter dans notre fichier de configuration en rajoutant cette ligne:

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

Ensuite sur la partie client nous allons comme pour tous les projets, créer un dossier « Protos » et rajouter le fichier proto de l'application client avec l'itemgroup dans le fichier csproj :

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

Notre fichier proto a bien généré du code C# :



Actuellement les librairies gRPC, au mieux, prennent en charge les appels unaires et le streaming serveur mais ne prennent pas en charge le streaming client et le streaming bi directionnel comme vous pouvez le constater :

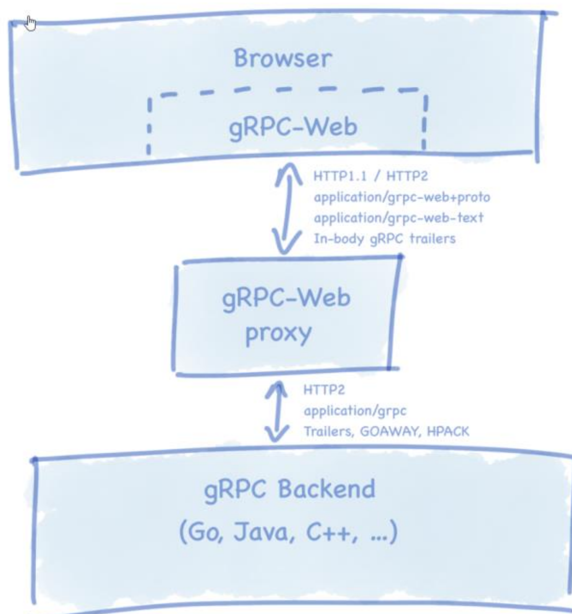
Client / Feature	Transport	Unary	Server-side streams	Client-side & bi-directional streaming
Improbable	Fetch/XHR	✓	✓	✗ [19]
Google (`grpcwebtext`)	XHR	✓	✓	✗
Google (`grpcweb`)	XHR	✓	✗ [20]	✗

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.

Ceci est dû à une limitation actuelle des proxy qui convertissent les appels http/1.1 en appels http/2 :



Nous allons donc devoir réécrire notre partie serveur pour pouvoir exploiter la partie Chat.

En effet, nous allons devoir couper en deux la méthode StartChatting() qui nous servait à envoyer et recevoir les messages.

Pour cela nous allons créer :

- Un appel unaire pour envoyer les messages et les rajouter à la queue
- Un appel streaming server pour recevoir les messages en continu.

Notre fichier proto va donc ressembler à cela sur la partie serveur et client :

```
syntax = "proto3";

option csharp_namespace = "ChatServer.Protos";

package chatgrpc;

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

message RoomRegistrationRequest {
	string room_name = 1;
	string user_name = 2;
}

message RoomRegistrationResponse {
	bool joined_room = 1;
}

message ChatMessage {
	google.protobuf.Timestamp msg_time = 1 ;
	string content = 2 ;
	string user_name = 3 ;
	string room_name = 4;
}

message CounterRequest {
  int32 start = 1;
}

message CounterResponse {
  int32 count = 1;
}

message ChatMessageResponse {
	bool success =1 ;
}

message RoomRequestMessage {
	string user_name = 1 ;
	string room_name = 2;
}
```

```
message ReceivedMessage {
	google.protobuf.Timestamp msg_time = 1 ;
	string content = 2;
	string user = 3;
}

service ChatGrpcService {
	rpc RegisterToRoom(RoomRegistrationRequest) returns
(RoomRegistrationResponse);
	rpc StartCounter (CounterRequest) returns (stream
CounterResponse);
	rpc SendMessage(ChatMessage) returns (ChatMessageResponse);
	rpc PullMessages(RoomRequestMessage) returns (stream
ChatMessage);
}
```

Avec une méthode SendMessage et une méthode PullMessages.

Notre implémentation du service sur le service a donc changé aussi et aura le code suivant :

```
using ChatServer.Protos;
using Google.Protobuf.WellKnownTypes;
using Grpc.Core;
using grpc.Wasm.Server.Utils;
using Microsoft.AspNetCore.Authorization;

namespace Chagrpc.Wasm.Server.Services
{
    public class ChatGrpcService :
ChatServer.Protos.ChatGrpcService.ChatGrpcServiceBase
    {
        private readonly ILogger<ChatGrpcService> _logger;
        public ChatGrpcService(ILogger<ChatGrpcService> logger)
        {
            _logger = logger;
        }

        public override async Task StartCounter(CounterRequest request,
IServerStreamWriter<CounterResponse> responseStream,
ServerCallContext context)
        {
            var count = request.Start;

            while (!context.CancellationToken.IsCancellationRequested)
            {
                await responseStream.WriteAsync(new CounterResponse
                {
```

```
                Count = ++count
            });

            await Task.Delay(TimeSpan.FromSeconds(1));
        }
    }




    public override async Task<RoomRegistrationResponse>
RegisterToRoom(RoomRegistrationRequest request, ServerCallContext
context)
    {
        UsersQueue.CreateUserQueue(request.RoomName,
request.UserName);
        var resp = new RoomRegistrationResponse { JoinedRoom = true
};
        return await Task.FromResult(resp);
    }

    public override async Task<ChatMessageResponse>
SendMessage(ChatMessage request, ServerCallContext context)
    {
        string userName = request.UserName;
        string room = request.RoomName;

UsersQueue.AddMessageToRoom(ConvertToReceivedMessage(request),
room);
        var resp = new ChatMessageResponse { Success = true };
        return await Task.FromResult(resp);
    }

    public override async Task PullMessages(RoomRequestMessage
request, IServerStreamWriter<ChatMessage> responseStream,
ServerCallContext context)
    {
        var respTask = Task.Run(async () =>
        {
            while (true)
            {
                var userMsg =
UsersQueue.GetMessageForUser(request.UserName);
                if (userMsg != null)
                {
                    var userMessage = ConvertToChatMessage(userMsg,
request.RoomName);
                    await responseStream.WriteAsync(userMessage);
```

```
                }
                if (MessageQueue.GetMessagesCount() > 0)
                {
                    var news = MessageQueue.GetNextMessage();
                    var newsMessage = ConvertToChatMessage(news,
request.RoomName);
                    await responseStream.WriteAsync(newsMessage);
                }

                await Task.Delay(200);
            }
        });

        // Keep the method running
        while (true)  {
            await Task.Delay(10000);
        }
    }


    private ReceivedMessage ConvertToReceivedMessage(ChatMessage
chatMsg)  {
        var rcMsg = new ReceivedMessage
        {
            Content = chatMsg.Content,
            MsgTime = chatMsg.MsgTime,
            User = chatMsg.UserName
        };
        return rcMsg;
    }

    private ChatMessage ConvertToChatMessage(ReceivedMessage
rcMsg, string room)  {
        var chatMsg = new ChatMessage
        {
            Content = rcMsg.Content,
            MsgTime = rcMsg.MsgTime,
            UserName = rcMsg.User,
            RoomName = room
        };
        return chatMsg;
    }

  }
}
```

Pour tester tout cela dans notre application Blazor, nous allons devoir injecter notre client gRPC dans le « Program.cs » sur la partie client car sinon, nous ne pourrons pas l'utiliser dans nos pages :

```
builder.Services.AddSingleton(services =>
{
    // Get the service address from appsettings.json
    var config = services.GetRequiredService<IConfiguration>();

    var navigationManager = services.GetRequiredService<NavigationManager>();
    var backendUrl = navigationManager.BaseUri;

    // Create a channel with a GrpcWebHandler that is addressed to the backend
    server.
    //
    // GrpcWebText is used because server streaming requires it. If server
streaming is not used in your app
    // then GrpcWeb is recommended because it produces smaller messages.
    var httpHandler = new GrpcWebHandler(GrpcWebMode.GrpcWebText, new
HttpClientHandler());

    return GrpcChannel.ForAddress(backendUrl, new GrpcChannelOptions {
HttpHandler = httpHandler });
});
```

Pour finir,nous allons remplacer le code de la page « Index.cshtml » par celui-ci :

```
@page "/"
@using Microsoft.AspNetCore.Components
@using ChatServer.Protos
@using Google.Protobuf.WellKnownTypes
@using Grpc.Core
@using System.Threading.Channels
@using Grpc.Net.Client
@inject GrpcChannel Channel

<PageTitle>Index</PageTitle>

<h1>Hello, world!</h1>


Welcome to the chat app !
<br />

<EditForm Model="@roomRegistrationRequest"
OnSubmit="@ConnectToChatRoom">
    <InputText id="roomname" @bind-
Value="roomRegistrationRequest.RoomName" />
    <InputText id="username" @bind-
Value="roomRegistrationRequest.UserName" />
    <button type="submit" disabled="@Connected">Submit</button>
```

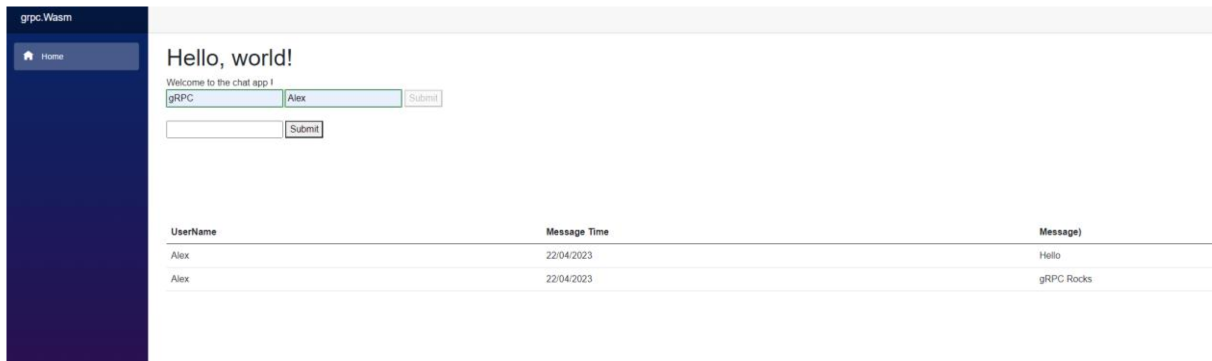
```
</EditForm>
<br />
<InputText id="message" @bind-Value="message" />
<button type="submit" @onclick="SendMessage">Submit</button>
<br />
<br />
<br />
<br />
<br />
<br />
<br />

<table class="table">
    <thead>
    <tr>
        <th>UserName</th>
        <th>Message Time</th>
        <th>Message)</th>
    </tr>
    </thead>
    <tbody>
    @foreach (var mess in listMessage)
    {
        <tr>
            <td>@mess.UserName</td>
            <td>@mess.MsgTime.ToDateTime().ToShortDateString()</td>
            <td>@mess.Content</td>
        </tr>
    }
    </tbody>
</table>


@code {
    private RoomRegistrationRequest roomRegistrationRequest = new
RoomRegistrationRequest();
    private string message = string.Empty;
    private List<ChatMessage> listMessage = new();
    private CancellationTokenSource cts;
    private bool Connected = false;

    private async Task ConnectToChatRoom()
    {
        cts = new CancellationTokenSource();
        var client = new ChatGrpcService.ChatGrpcServiceClient(Channel);

        try
        {
```

```
            var res = await
client.RegisterToRoomAsync(roomRegistrationRequest);
            if (res.JoinedRoom)
            {
                Connected = true;
                    await PullMessages();

            }

        }
        catch (Exception e)
        {
            Console.WriteLine(e);
            throw;
        }

    }

    private async Task SendMessage()
    {
        try
        {
            var client = new
ChatGrpcService.ChatGrpcServiceClient(Channel);
            var msg = message;
            RestoreInputCursor();
            var reqMessage = new ChatMessage
                {
                    Content = msg,
                    MsgTime = Timestamp.FromDateTime(DateTime.UtcNow),
                    RoomName = roomRegistrationRequest.RoomName,
                    UserName = roomRegistrationRequest.UserName
                };
            await client.SendMessageAsync(reqMessage);
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
            throw;
        }

    }

    void PrintMessage(ChatMessage msg)
    {
        listMessage.Add(msg);
        StateHasChanged();
    }
```

```
    async Task PullMessages()
    {

        var client = new ChatGrpcService.ChatGrpcServiceClient(Channel);
        var call = client.PullMessages(new RoomRequestMessage {
RoomName = roomRegistrationRequest.RoomName , UserName =
roomRegistrationRequest.UserName});
        var task = Task.Run(async () =>
        {

            while (true)
            {
                try
                {
                    if (await call.ResponseStream.MoveNext())
                    {
                        var message  = call.ResponseStream.Current;
                        Console.WriteLine(message);
                        PrintMessage(message);
                    }
                }
                catch (Grpc.Core.RpcException e)
                {
                    if (e.StatusCode == StatusCode.Cancelled)
                    {
                        Console.WriteLine("Chat Cancelled");
                        break;
                    }
                }

            }
        });


    }

    void RestoreInputCursor()
    {
        message = string.Empty;
    }
}
```

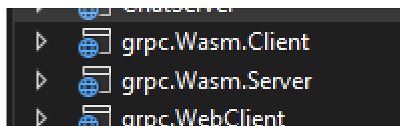
Nous constatons que lorsque que nous nous connectons au chat room, si la requête est en succès, nous allons lancer le streaming serveur.

Le streaming serveur reçoit des messages en continu et lorsqu'un nouveau message arrive, il le rajoute à la liste des messages et met à jour l'interface avec la méthode StateHasChanged().



Voilà c'est tout pour cette partie applicative Web qui mérite encore des améliorations de la part de gRPC mais qui est déjà plus performante pour les appels unaires que les web api REST.

Vous trouverez l'exemple codé ici : « Exercices\08 - gRPC Web » et pour tester lancez ces 2 projets :



Chapitre 15 : Conclusion

1. Conclusion

Nous avons donc vu gRPC des bases vers les sujets plus avancés.

- Les bases des API et WEB API
- Les bases de gRPC
- Les bases de protobuf
- La creation d’un serveur gRPC
- Tous les types de communications possibles via gRPC
- Des sujets plus avancés comme les deadlines ou les cancellation tokens
- L’utilisation de gRPC sur une application Web