



TP : Développer un service gRPC

Architecture des composants d'entreprise

Table des matières

I.	Objectif du TP.....	2
II.	Prérequis	2
III.	L'architecture gRPC.....	2
a.	Historique des architectures distribuées client serveur	2
b.	Formats d'échange de données : XML, JSON, ProtoBuf	3
c.	C'est quoi gRPC ?.....	3
d.	Les 04 modèles de communication avec gRPC	4
e.	Protocol Buffer	5
IV.	Développement du serveur gRPC fonctionnant en Unary Model.....	6
a.	Création du projet Maven	6
b.	Création du fichier PROTO.....	6
c.	Ajout des dépendances et plugin PROTOBUF.....	8
d.	Génération du Stub.....	10
e.	Développement du service.....	11
f.	Création du serveur gRPC	12
V.	Tester avec le client BloomRPC	12
VI.	Développement du client gRPC.....	14
VII.	Développement du serveur gRPC fonctionnant en Server Streaming Model	15
a.	Implémenter le service	15
b.	Développer le client.....	17
VIII.	Développement du serveur gRPC fonctionnant en Client Streaming Model	19
a.	Développer le service	19
b.	Développer le client.....	20
IX.	Développement du serveur gRPC fonctionnant en Bidirectional Streaming Model.....	22
a.	Développer le service	22
b.	Développer le client.....	24

Objectif du TP

- Comprendre l'architecture gRPC.
- Développer un serveur qui fonctionne en mode **Unary Streaming**.
- Développer un serveur qui fonctionne en mode **Server Streaming**.
- Développer un serveur qui fonctionne en mode **Client Streaming**.
- Développer un serveur qui fonctionne en mode **Bidirectional Streaming**.
- Développer les clients pour tester les services.

NB : Le code source du TP est disponible sur GITHUB :

- ✓ Partie Serveur : <https://github.com/abbouformations/tpgrpcserver.git>.
- ✓ Partie Client : <https://github.com/abbouformations/tpgrpcclient.git>.

I. Prérequis

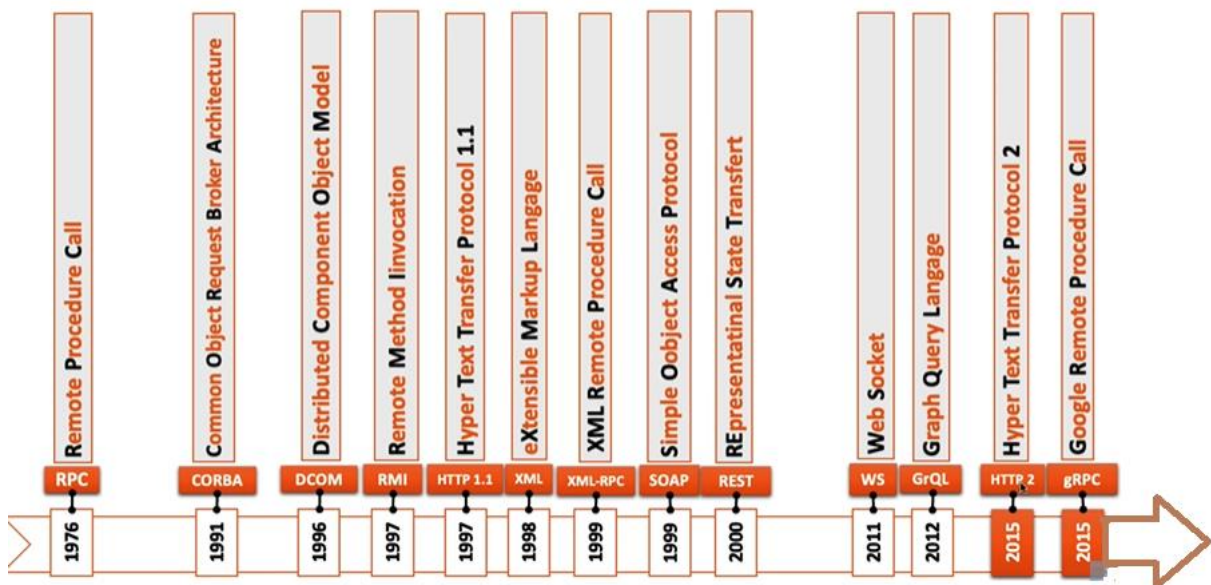
- IntelliJ IDEA ;
- JDK version 17 ;
- Le client BloomRPC ;
- Une connexion Internet pour permettre à Maven de télécharger les librairies.

NB : Ce TP a été réalisé avec IntelliJ IDEA 2023.1.2 (Community Edition).

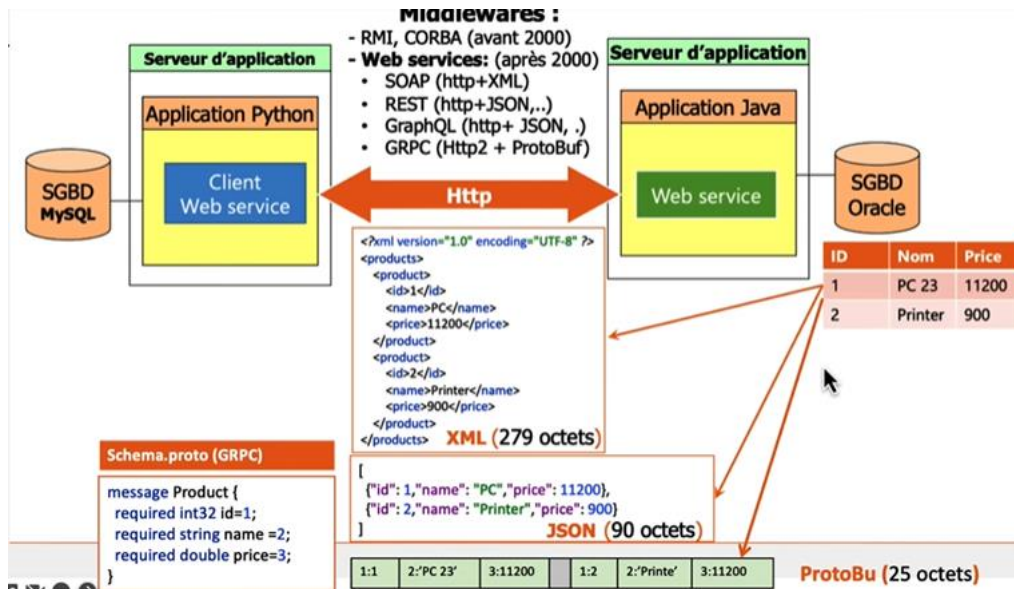
II. L'architecture gRPC

a. Historique des architectures distribuées client serveur

L'image ci-dessous explique l'historique des architectures distribuées client-serveur :

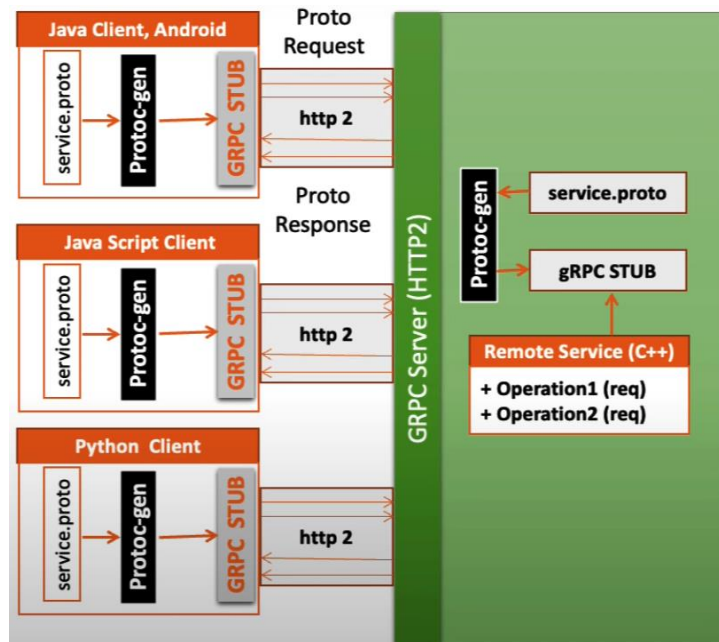


b. Formats d'échange de données : XML, JSON, ProtoBuf



c. C'est quoi gRPC ?

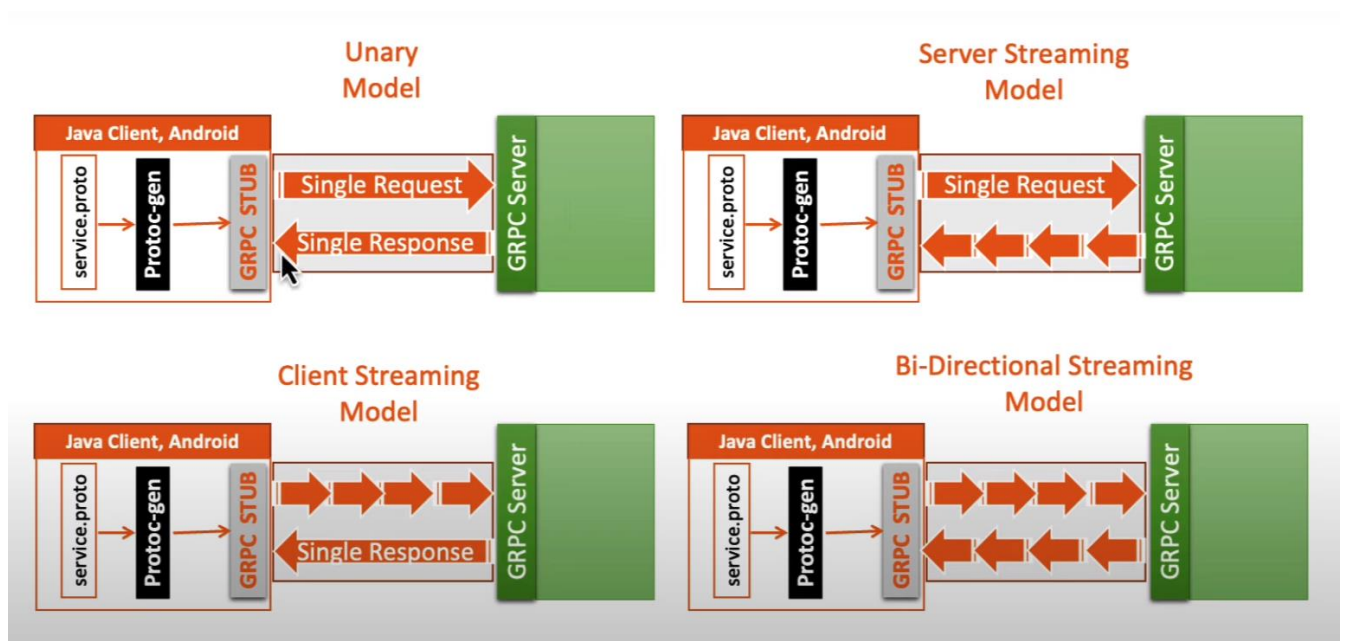
- Le schéma suivant montre l'architecture gRPC :



- **gRPC** est une Framework qui permet d'exposer les fonctionnalités d'un service pour un accès distant via HTTP 2. Même titre de que ces prédécesseurs RPC, CORBA, RMI, SOAP, REST et GraphQL.
- Proche de CORBA, **gRPC** permet d'assurer la communication entre plusieurs applications distribuées multi langages et multi plateformes.
- gRPC utilise « Protocol Buffers » (ProtoBuff) comme mécanisme de sérialisation privilégié, développé par Google comme alternative plus performante qu'IDL, XML, JSON.
- Il bénéficie des performances du protocole http 2 à savoir :
 - o Communication bidirectionnelles (Full Duplex) entre le client et le serveur. Ce qui permet de créer des applications distribuées réactives.
 - o Une connexion permanente pour toute la conversation (possibilité d'envoyer plusieurs requêtes simultanées pour une même connexion TCP).
 - o Optimisation du contenu échangé.
 - o Les clients et serveurs peuvent transmettre des données en continu ce qui évite d'avoir à demander de nouvelles données ou à créer d'autres connexions.

d. Les 04 modèles de communication avec gRPC

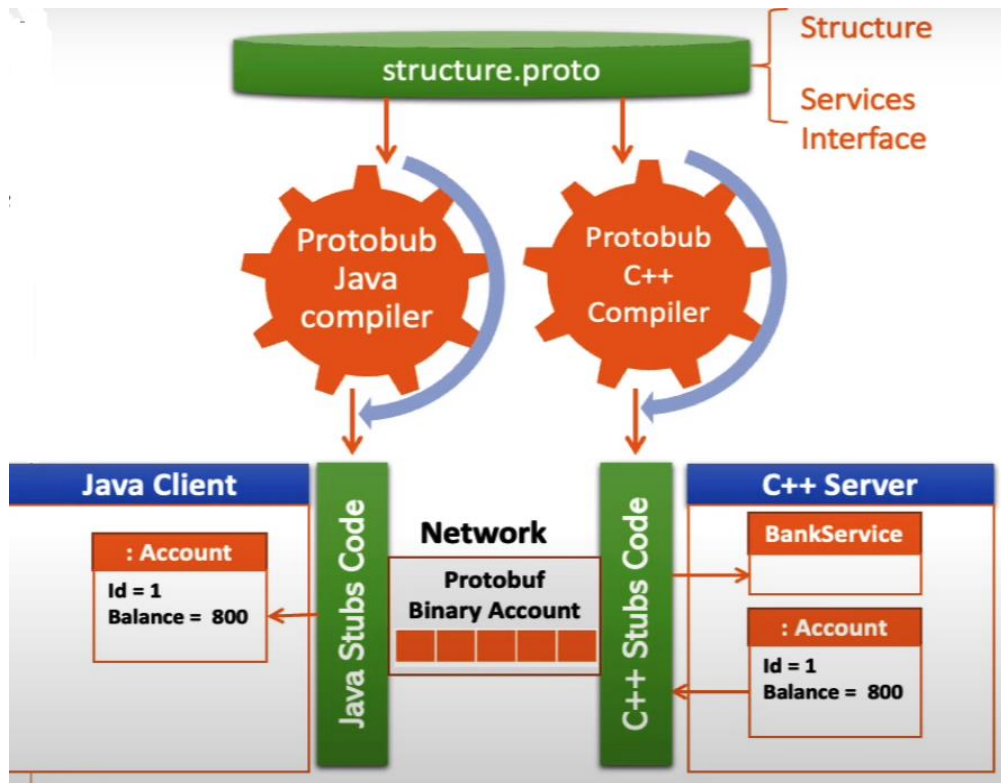
gRPC offre la possibilité de développer des services selon les 04 modèles suivants :



- Le modèle « **Unary** » c'est le modèle normal utilisé dans HTTP 1. En effet, le client envoie une seule requête et le serveur répond par une seule réponse.
- Le modèle « **Server Streaming** » offre la possibilité au client d'envoyer une seule requête et au serveur de répondre via plusieurs Stream.
- Le modèle « **Client Streaming** » offre la possibilité au client d'envoyer plusieurs requêtes et au serveur de répondre via une seule réponse.
- Le modèle « **Bi-Directional Streaming** » offre la possibilité au client d'envoyer plusieurs Stream et au serveur de répondre via plusieurs Stream.

e. Protocol Buffer

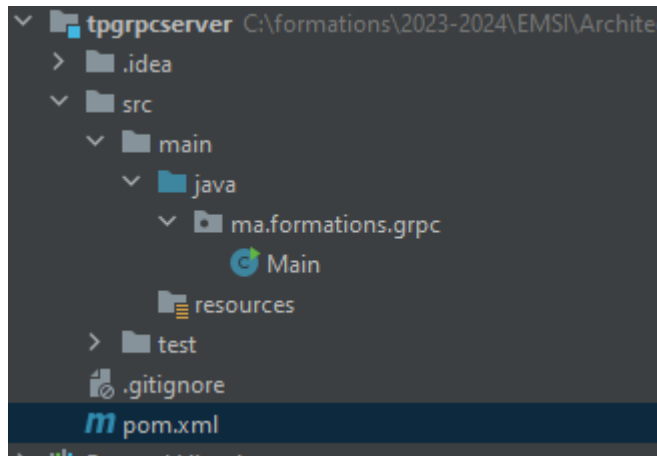
- **Protocol Buffers** ou **ProtoBuf** est un format de sérialisation avec un langage de description d'interface développé par Google.
- A l'interface de XML et JSON, le contenu d'un message **ProtoBuf** n'est pas lisible par l'être humain car le message est sérialisé au format binaire.
- La capacité d'un message sérialisé en **ProtoBuf** est 60% plus réduite que JSON.
- **ProtoBuf** est plus rapide à encoder et à décoder que JSON.
- La structure des messages et des services **ProtoBuf** sont déclarés dans un fichier séparé « .proto » au même titre que WSDL pour les Web Services SOAP qui utilise les schémas XML (XSD).
- Ce fichier « .proto » est compilé par la suite pour générer le code des stubs avec différents langages : C++, Java, Python, C#, Dart, Go, Ruby, Kotlin, Objective-C :



III. Développement du serveur gRPC fonctionnant en Unary Model

a. Création du projet Maven

- Créer un projet Maven, par exemple *tpgrpcserver* :



b. Création du fichier PROTO

- Créer le fichier *calculator.proto* suivant dans *src/main/java/resources* :

```
syntax = "proto3";
option java_package = "ma.formations.grpc.stubs";
```

```

service CalculatorService{
    //Unary Model :
    rpc sum(UnaryRequest) returns (UnaryResponse);

    //Server Streaming Model
    rpc getOperationStream(ServerStreamRequest) returns (stream ServerStreamResponse);

    //Client Streaming Model
    rpc performStream(stream ClientStreamRequest) returns (ClientStreamResponse);

    //Bidirectional Streaming Model
    rpc fullStream(stream BidirectionalStreamRequest) returns (stream
BidirectionalStreamResponse);
}

//Using for Unary Request and response example
message UnaryRequest {
    double a = 1;
    double b = 2;
}

message UnaryResponse {
    double a = 1;
    double b = 2;
    double result = 3;
}

//Using for server Streaming Request and response example
message ServerStreamRequest {
    double a = 1;
    double b = 2;
}

message ServerStreamResponse {
    double a = 1;
    double b = 2;
    double result = 3;
    string type = 4;
}

//Using for Client Streaming Request and response example
message ClientStreamRequest {
    double a = 1;
}

message ClientStreamResponse {
    double result = 1;
}

```



```

repeated double receivedData = 2;
}

//Using for Bidirectional Streaming Request and response example
message BidirectionalStreamRequest {
    double a = 1;
}

message BidirectionalStreamResponse {
    double result = 1;
}

```

Explications :

- Le fichier doit respecter le protocole Buffers.
- Vous précisez la valeur "proto3" dans syntax.
- Le compilateur gRPC générera les classes de STUB dans le package ***ma.formations.grpc.stubs***.
- Le nom du service est CalculatorService qui offre 04 méthodes suivantes :
 - sum() implémentant le modèle : "Unary Model"
 - getOperationStream() implémentant le modèle "Server Streaming Model"
 - performStream() implémentant le modèle "Client Streaming Model"
 - fullStream() implémentant le modèle "Bidirectional Streaming Model"

c. Ajout des dépendances et plugin PROTOBUF

- Ajouter dans le fichier pom.xml les dépendances et le plugin suivants pour pouvoir compiler le fichier proto:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://maven.apache.org/POM/4.0.0"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>ma.formations.grpc</groupId>
    <artifactId>tpgrpcserver</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

```

```

<dependencies>
  <!-- https://mvnrepository.com/artifact/com.google.protobuf/protobuf-java -->
  <dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.24.4</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/io.grpc/grpc-netty-shaded -->
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty-shaded</artifactId>
    <version>1.58.0</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/io.grpc/grpc-protobuf -->
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-protobuf</artifactId>
    <version>1.58.0</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/io.grpc/grpc-stub -->
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-stub</artifactId>
    <version>1.58.0</version>
  </dependency>
  <dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>com.github.os72</groupId>
      <artifactId>protoc-jar-maven-plugin</artifactId>
      <version>3.11.4</version>
      <executions>
        <execution>
          <phase>generate-sources</phase>
          <goals>
            <goal>run</goal>
          </goals>
          <configuration>
            <includeMavenTypes>direct</includeMavenTypes>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

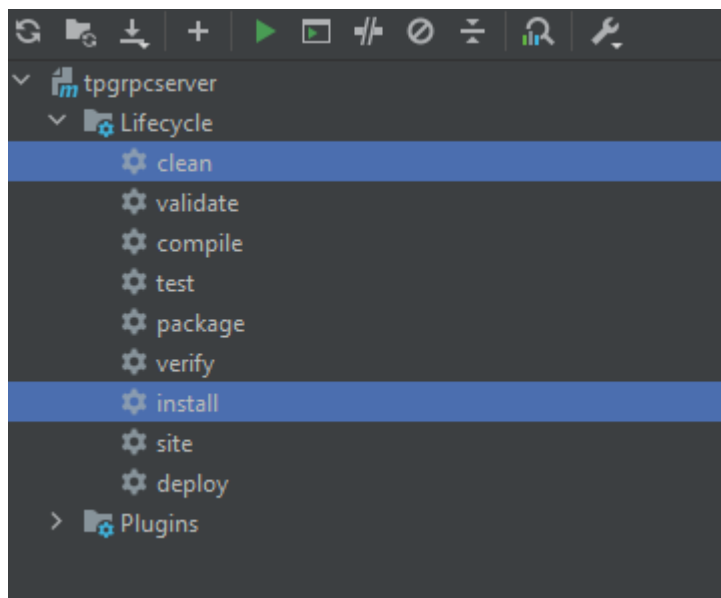
```

        <inputDirectories>
          <include>src/main/resources</include>
        </inputDirectories>
        <outputTargets>
          <outputTarget>
            <type>java</type>
            <outputDirectory>src/main/java</outputDirectory>
          </outputTarget>
          <outputTarget>
            <type>grpc-java</type>
            <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.15.0</pluginArtifact>
            <outputDirectory>src/main/java</outputDirectory>
          </outputTarget>
        </outputTargets>
      </configuration>
    </execution>
  </executions>
</plugin>
</plugins>
</build>
</project>

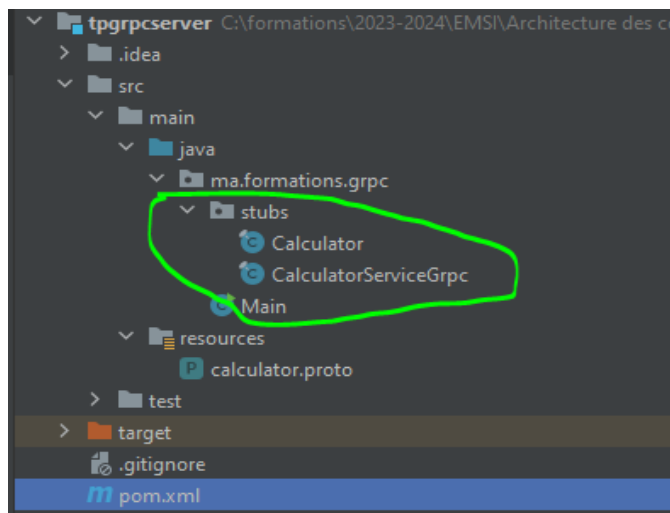
```

d. Génération du Stub

- Lancer la commande *clean install* comme illustré ci-dessous afin de générer le STUB :



- Vérifier que les classes du STUB ont été bien créées au niveau du package ***ma.formations.grpc.stubs*** :



e. Développement du service

- Créer la classe **CalculatorService** et redéfinir la méthode *sum()* suivante :

```
package ma.formations.grpc.service;

import io.grpc.stub.StreamObserver;
import ma.formations.grpc.stubs.Calculator;
import ma.formations.grpc.stubs.CalculatorServiceGrpc;

import java.util.ArrayList;
import java.util.List;
import java.util.Timer;
import java.util.TimerTask;

public class CalculatorService extends CalculatorServiceGrpc.CalculatorServiceImplBase {

    @Override
    public void sum(Calculator.UnaryRequest request,
StreamObserver<Calculator.UnaryResponse> responseObserver) {
        double a = request.getA();
        double b = request.getB();
        double result = a + b;
        Calculator.UnaryResponse response = Calculator.UnaryResponse.newBuilder().
            setA(a).
            setB(b).
            setResult(result).
            build();
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
}
```

f. Création du serveur gRPC

- Créer la classe **GrpcServer** suivante :

```
package ma.formationen.grpc.server;

import io.grpc.Server;
import io.grpc.ServerBuilder;
import ma.formationen.grpc.service.CalculatorService;

import java.io.IOException;

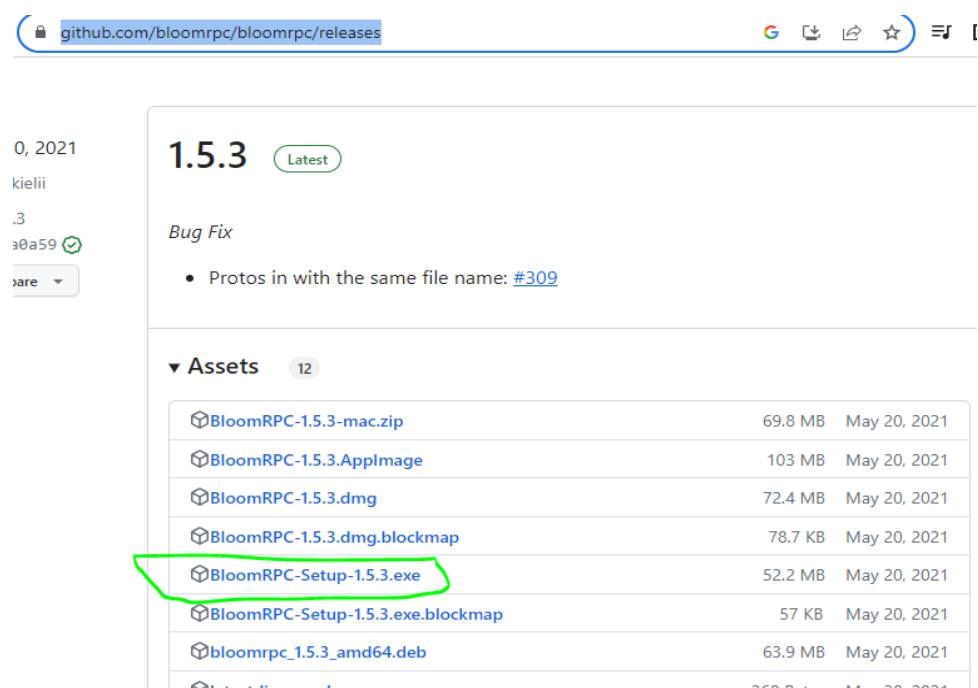
public class GrpcServer {
    public static void main(String[] args) throws IOException, InterruptedException {
        Server grpcServer = ServerBuilder.forPort(9999)
            .addService(new CalculatorService()).build();
        grpcServer.start();
        System.out.println("serveur gRPC démarré : http://localhost:9999");
        grpcServer.awaitTermination();
    }
}
```

- Lancer la méthode main() ci-dessus pour démarrer le serveur gRPC.

IV. Tester avec le client BloomRPC

- Le client BloomRPC est téléchargeable via le lien suivant :

<https://github.com/bloomrpc/bloomrpc/releases>



0, 2021
kielii
.3
30a59
bare

1.5.3 Latest

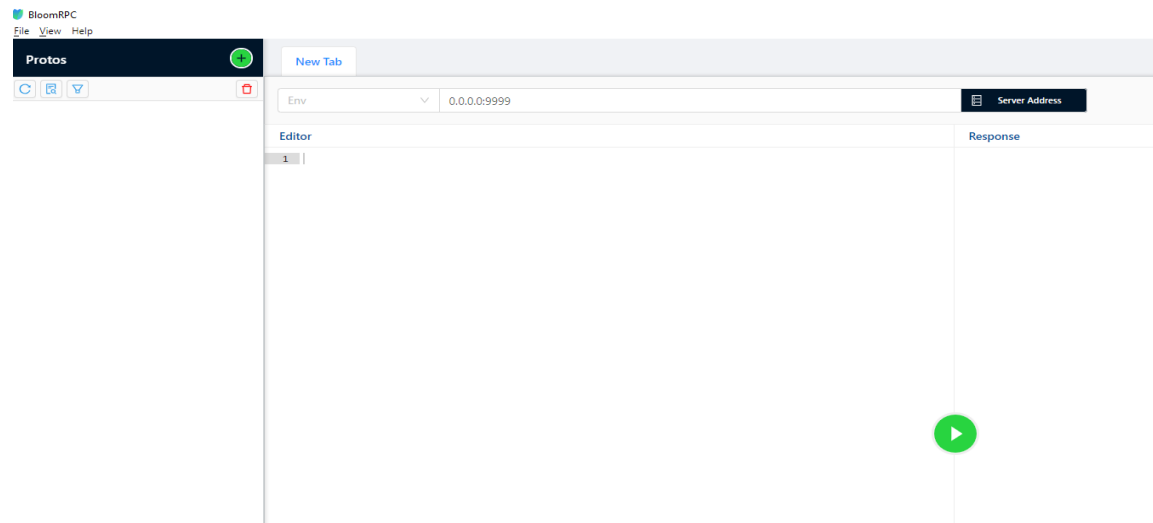
Bug Fix

- Protos in with the same file name: [#309](#)

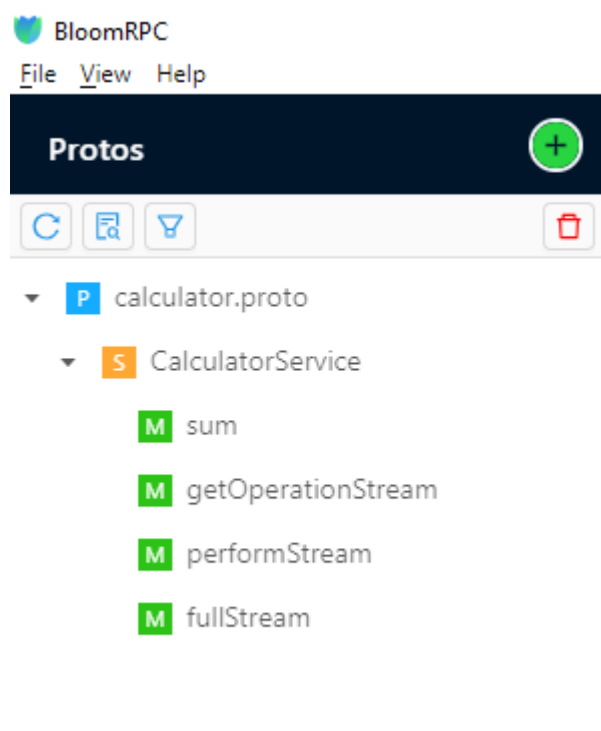
▼ Assets 12

BloomRPC-1.5.3-mac.zip	69.8 MB	May 20, 2021
BloomRPC-1.5.3.Applimage	103 MB	May 20, 2021
BloomRPC-1.5.3.dmg	72.4 MB	May 20, 2021
BloomRPC-1.5.3.dmg.blockmap	78.7 KB	May 20, 2021
BloomRPC-Setup-1.5.3.exe	52.2 MB	May 20, 2021
BloomRPC-Setup-1.5.3.exe.blockmap	57 KB	May 20, 2021
bloomrpc_1.5.3_amd64.deb	63.9 MB	May 20, 2021
Latest Release	269 Downloads	May 20, 2021

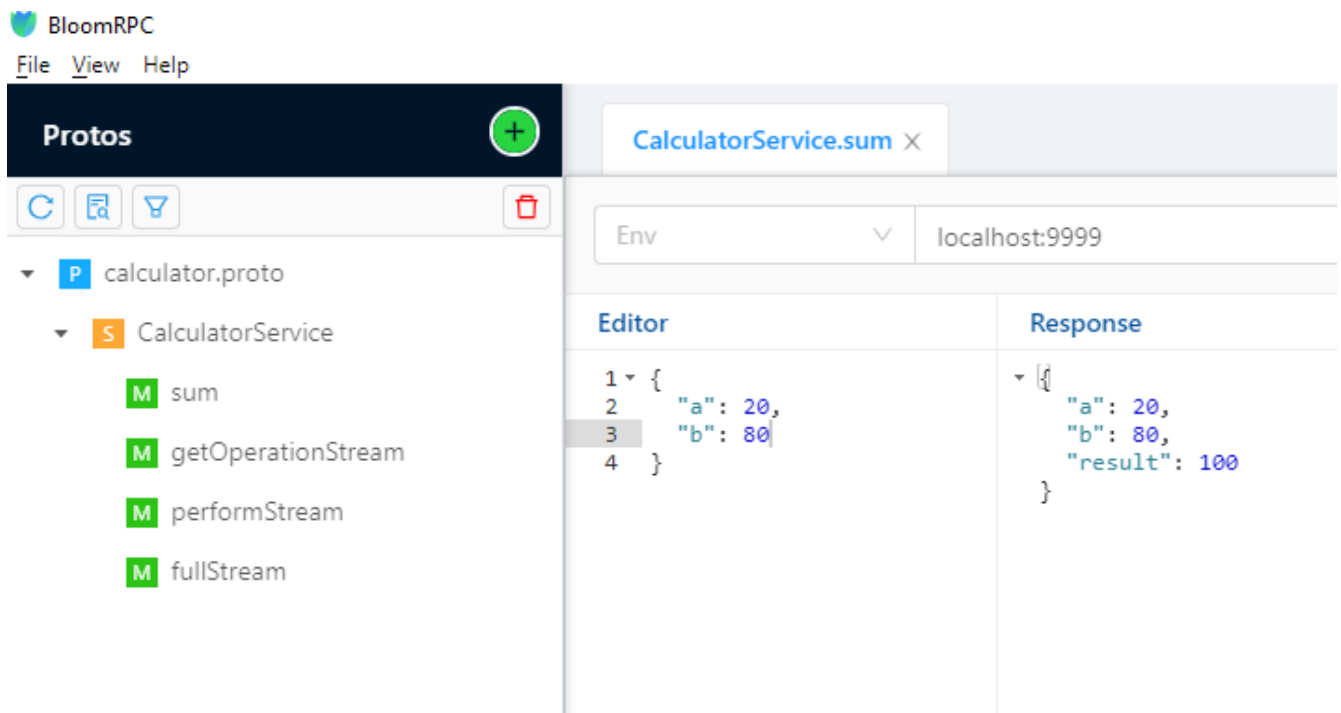
- Installer l'outil en cliquant sur l'exécutable téléchargé. L'interface de l'outil est la suivante :



- Cliquer sur (+) pour ajouter le contrat *calculator.proto*. Le service CalculatorService ainsi que ces méthodes exposées seront affichés comme suit :



- Préciser le serveur dans l'URL (**localhost :9999** ou bien **0.0.0.0 :9999**)
- Pour tester la méthode *sum*, cliquer sur cette dernière, entrer les valeurs de a et b et cliquer sur le bouton en bas comme illustré ci-dessous :



V. Développement du client gRPC

- Créer un nouveau projet Maven, par exemple *tpgrpcclient*.
- Copier les mêmes dépendances gRPC ainsi que le plugin gRPC de la partie serveur ci-dessus.
- Créer le fichier *calculator.proto* dans *src/main/java/resources* (le même contenu que celui de la partie serveur ci-dessus).
- Faire un *clean install* pour générer le STUB.
- Créer la classe **UnaryModelClient** suivante :

```
package ma.formationen.grpc;

import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import ma.formationen.grpc.stubs.Calculator;
import ma.formationen.grpc.stubs.CalculatorServiceGrpc;

public class UnaryModelClient {

    public static void main(String[] args) {
        ManagedChannel channel = ManagedChannelBuilder.
            forAddress("localhost", 9999).
            usePlaintext().
            build();
```

```

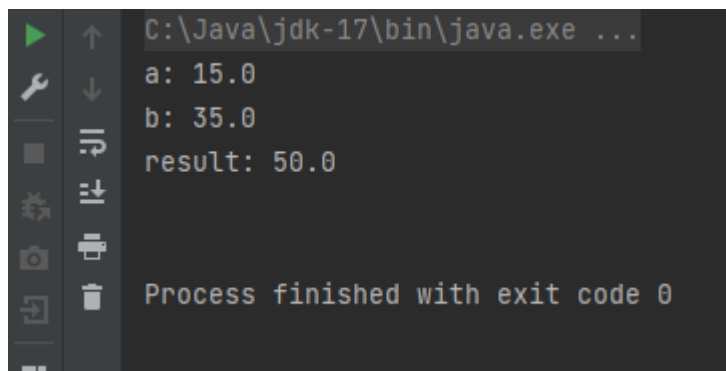
double a = 15;
double b = 35;

    CalculatorServiceGrpc.CalculatorServiceBlockingStub blockingStub =
CalculatorServiceGrpc.newBlockingStub(channel);
    Calculator.UnaryRequest request = Calculator.UnaryRequest.newBuilder()
        .setA(a)
        .setB(b)
        .build();
    Calculator.UnaryResponse response = blockingStub.sum(request);
    System.out.println(response);

}
}

```

- Exécuter la méthode main() ci-dessus et vérifier le résultat :



```

C:\Java\jdk-17\bin\java.exe ...
a: 15.0
b: 35.0
result: 50.0

Process finished with exit code 0

```

VI. Développement du serveur gRPC fonctionnant en Server Streaming Model

a. Implémenter le service

- Redéfinir la méthode la méthode **getOperationStream** au niveau de la classe

CalculatorService comme suit :

```

@Override
public void getOperationStream(Calculator.ServerStreamRequest request,
StreamObserver<Calculator.ServerStreamResponse> responseObserver) {
    double a = request.getA();
    double b = request.getB();

    Timer timer = new Timer();
    timer.schedule(new TimerTask() {
        String type = "No operation is performed by the server";
        int counter = 0;
        double result = -1;
    }, 1000);
}

```



```

@Override
public void run() {
    if (counter == 0) {
        result = a + b;
        type = "a + b";
    }

    if (counter == 1) {
        result = a - b;
        type = "a - b";
    }

    if (counter == 2) {
        result = a * b;
        type = "a * b";
    }

    if (counter == 3) {
        result = a / b;
        type = "a / b";
    }

    if (counter == 4) {
        result = (a + b) * (a + b);
        type = "(a + b)*(a + b)";
    }

    if (counter >= 5) {
        result = -1;
        type = "No operation is performed by the server";
    }
}

```

```

Calculator.ServerStreamResponse response = Calculator.
    ServerStreamResponse.
        newBuilder().
            setA(a).
            setB(b).
            setResult(result).
            setType(type).
            build();
responseObserver.onNext(response);
++counter;

if (counter == 10) {
    responseObserver.onCompleted();
    timer.cancel();
}

```

```

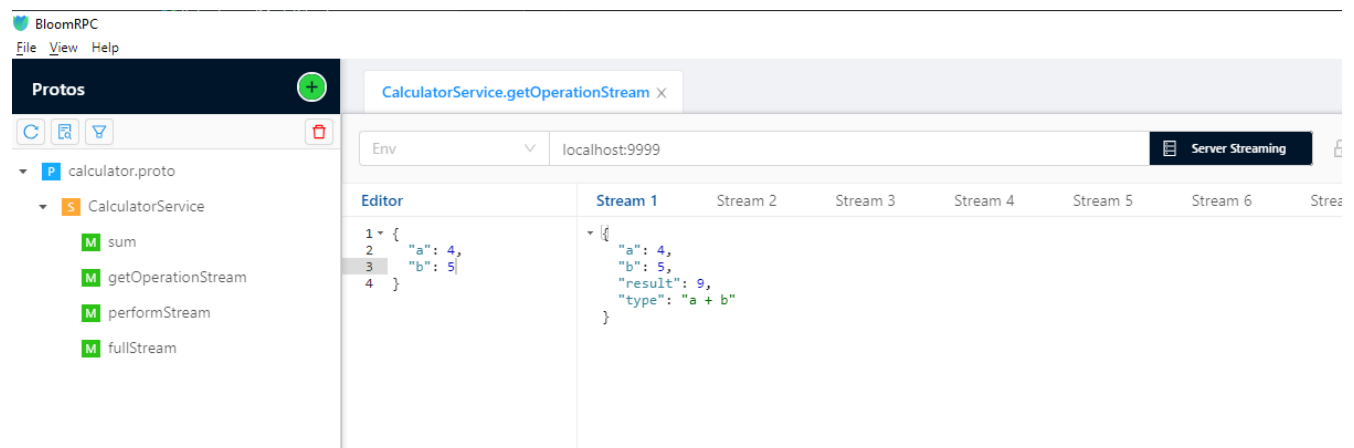
    }

    }, 1000, 1000);
}

```

Explications :

- La méthode *getOperationStream* exécutera en mode streaming chaque seconde les opérations suivantes dans l'ordre : l'addition, la soustraction, la multiplication, la division et le carré de la somme des deux nombres. A partir de la 6^{ième} seconde, la méthode n'exécutera aucune opération.
- Tester le service en utilisant le client BloomRPC et remarquer que le serveur fonctionne en mode Server Streaming :



b. Développer le client

- Au niveau du projet *tpgrpcclient*, créer la classe *ServerStreamingClient* suivante :

```

package ma. formations.grpc;

import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.stub.StreamObserver;
import ma. formations.grpc.stubs.Calculator;
import ma. formations.grpc.stubs.CalculatorServiceGrpc;

import java.io.IOException;

public class ServerStreamingClient {

```

```

public static void main(String[] args) throws IOException {
    ManagedChannel channel = ManagedChannelBuilder.
        forAddress("localhost", 9999).
        usePlaintext().
        build();
    double a = 10d;
    double b = 20d;
    Calculator.ServerStreamRequest request = Calculator.ServerStreamRequest.newBuilder()
        .setA(a)
        .setB(b).
        build();
    CalculatorServiceGrpc.CalculatorServiceStub asynStub =
    CalculatorServiceGrpc.newStub(channel);

    asynStub.getOperationStream(request, new
    StreamObserver<Calculator.ServerStreamResponse>() {
        @Override
        public void onNext(Calculator.ServerStreamResponse serverStreamResponse) {
            System.out.println("#####");
            System.out.println(serverStreamResponse);
            System.out.println("#####");
        }

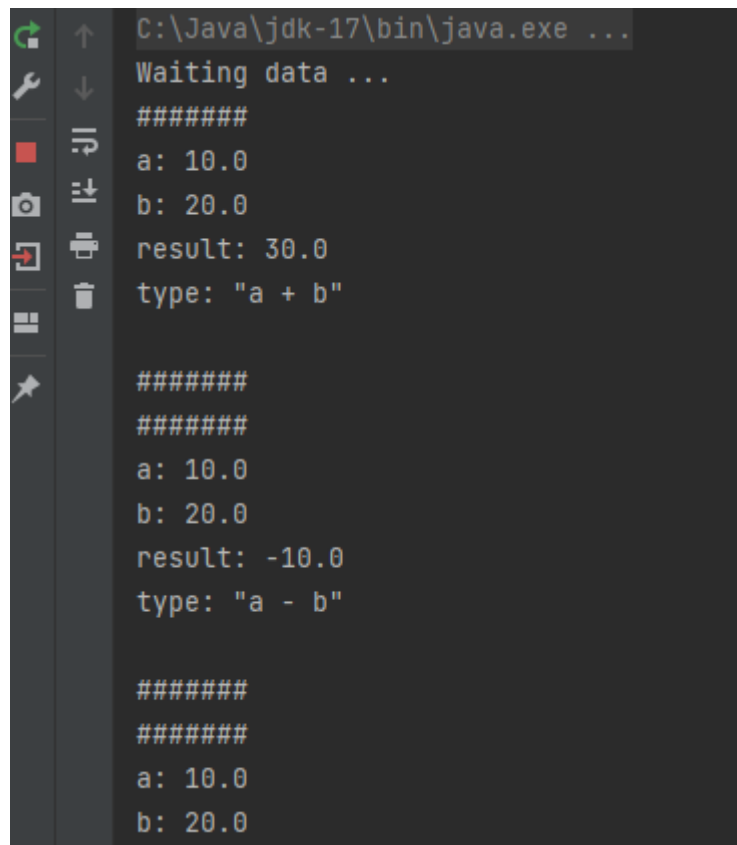
        @Override
        public void onError(Throwable throwable) {
            System.out.println(throwable.getMessage());
        }

        @Override
        public void onCompleted() {
            System.out.println("FIN...");
        }
    });

    System.out.println("Waiting data ...");
    System.in.read();
}
}

```

- Exécuter la méthode main ci-dessus et observer le résultat :



```
C:\Java\jdk-17\bin\java.exe ...  
Waiting data ...  
#####  
a: 10.0  
b: 20.0  
result: 30.0  
type: "a + b"  
  
#####  
#####  
a: 10.0  
b: 20.0  
result: -10.0  
type: "a - b"  
  
#####  
#####  
a: 10.0  
b: 20.0
```

VII. Développement du serveur gRPC fonctionnant en Client Streaming Model

a. Développer le service

- Redéfinir la méthode **performStream** au niveau de la classe **CalculatorService** comme illustré ci-dessous :

```
@Override  
public StreamObserver<Calculator.ClientStreamRequest>  
performStream(StreamObserver<Calculator.ClientStreamResponse> responseObserver) {  
    return new StreamObserver<Calculator.ClientStreamRequest>() {  
        final List<Double> receivedData = new ArrayList<>();  
        double result = 0;  
  
        @Override  
        public void onNext(Calculator.ClientStreamRequest operationRequest) {  
  
            result += operationRequest.getA();  
            receivedData.add(operationRequest.getA());  
        }  
  
        @Override  
        public void onError(Throwable throwable) {  
            throwable.printStackTrace();  
        }  
    };  
}
```

```

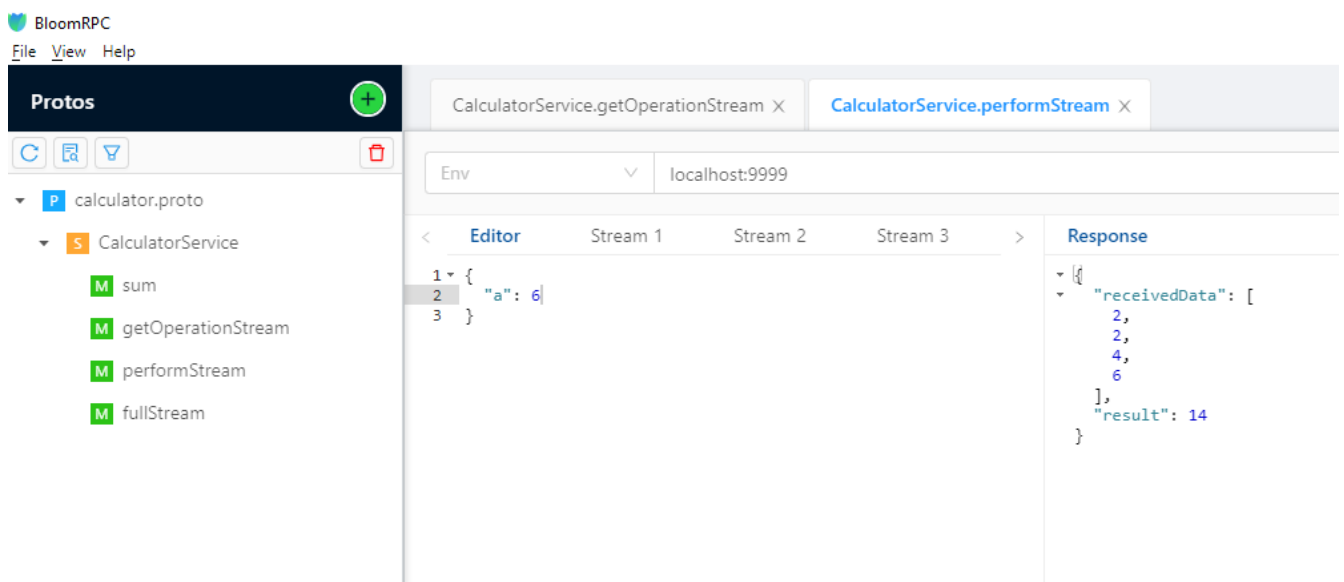
    }

    @Override
    public void onComplete() {
        Calculator.ClientStreamResponse.Builder response = Calculator.ClientStreamResponse.
            newBuilder();
        response.setResult(result);
        receivedData.forEach(data -> response.addReceivedData(data));
        responseObserver.onNext(response.build());
        responseObserver.onCompleted();
    }
};
}

```

Explications :

- La méthode *performStream* exécutera en mode *Client streaming* chaque seconde la somme des nombres envoyés par le client et enverra à la fin (après 10 secondes) la somme de l'ensemble des nombres reçus.
- Tester le service en utilisant le client BloomRPC et remarquer que le serveur fonctionne en mode *Client Streaming* :



b. Développer le client

- Dans le projet `tpgrpcclient`, créer la classe **ClientStreamingClient** suivante :

```

package ma.formation.grpc;

import io.grpc.ManagedChannel;

```

```

import io.grpc.ManagedChannelBuilder;
import io.grpc.stub.StreamObserver;
import ma.formations.grpc.stubs.Calculator;
import ma.formations.grpc.stubs.CalculatorServiceGrpc;

import java.io.IOException;
import java.util.Timer;
import java.util.TimerTask;

public class ClientStreamingClient {

    public static void main(String[] args) throws IOException {
        ManagedChannel channel = ManagedChannelBuilder.
            forAddress("localhost", 9999).
            usePlaintext().
            build();

        CalculatorServiceGrpc.CalculatorServiceStub asyncStub =
        CalculatorServiceGrpc.newStub(channel);

        StreamObserver<Calculator.ClientStreamRequest> performStream =
        asyncStub.performStream(new StreamObserver<Calculator.ClientStreamResponse>() {
            @Override
            public void onNext(Calculator.ClientStreamResponse clientStreamResponse) {
                System.out.println(clientStreamResponse);
            }

            @Override
            public void onError(Throwable throwable) {
            }

            @Override
            public void onCompleted() {
                System.out.println("END");
            }
        });

        Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            int counter = 0;

            @Override
            public void run() {
                Calculator.ClientStreamRequest clientStreamRequest = Calculator.
                ClientStreamRequest.
                newBuilder().
                setA(Math.random() * 10).

```

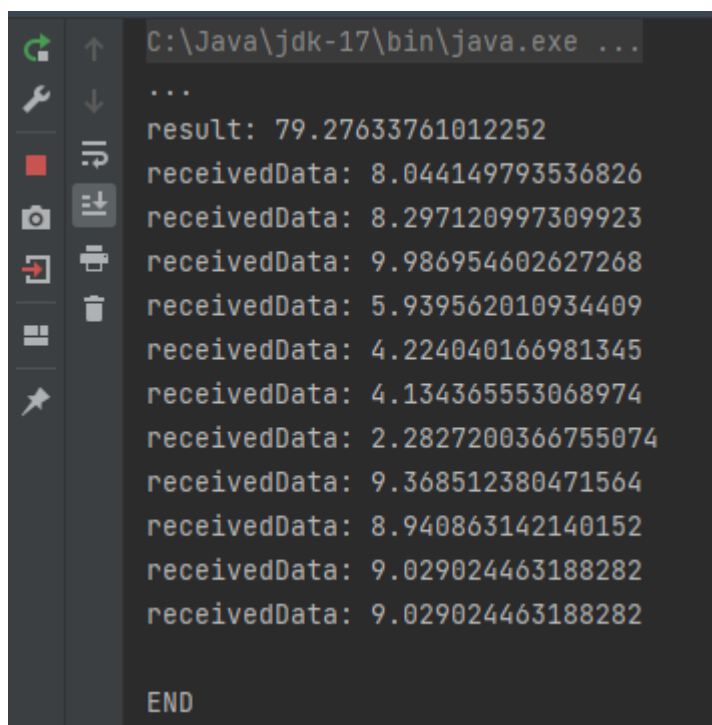
```

        build();
        performStream.onNext(clientStreamRequest);
        counter++;
        if (counter == 10) {
            performStream.onNext(clientStreamRequest);
            performStream.onCompleted();
            timer.cancel();
        }
    }

    }, 1000, 1000);
    System.out.println("...");
    System.in.read();
}
}

```

- Exécuter la méthode main ci-dessus et observer le résultat :



```

C:\Java\jdk-17\bin\java.exe ...
...
result: 79.27633761012252
receivedData: 8.044149793536826
receivedData: 8.297120997309923
receivedData: 9.986954602627268
receivedData: 5.939562010934409
receivedData: 4.224040166981345
receivedData: 4.134365553068974
receivedData: 2.2827200366755074
receivedData: 9.368512380471564
receivedData: 8.940863142140152
receivedData: 9.029024463188282
receivedData: 9.029024463188282
END

```

VIII. Développement du serveur gRPC fonctionnant en Bidirectional Streaming Model

a. Développer le service

- Redéfinir la méthode **fullStream** au niveau de la classe **CalculatorService** comme illustré ci-dessous :

```

@Override
public StreamObserver<Calculator.BidirectionalStreamRequest>
fullStream(StreamObserver<Calculator.BidirectionalStreamResponse> responseObserver) {
    return new StreamObserver<Calculator.BidirectionalStreamRequest>() {
        @Override
        public void onNext(Calculator.BidirectionalStreamRequest operationRequest) {

            Calculator.BidirectionalStreamResponse response = Calculator.
                BidirectionalStreamResponse.
                newBuilder().
                setResult(Math.pow(operationRequest.getA(), 2)).
                build();
            responseObserver.onNext(response);
        }

        @Override
        public void onError(Throwable throwable) {

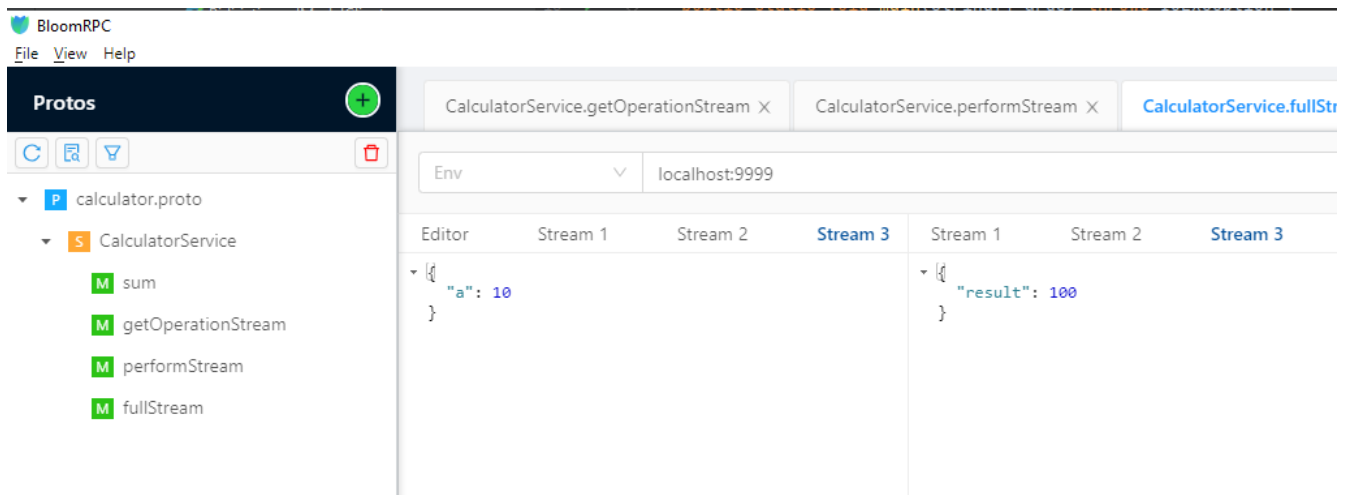
        }

        @Override
        public void onCompleted() {
            responseObserver.onCompleted();
        }
    };
}

```

Explications :

- La méthode **fullStream** exécutera en mode *Bidirectional streaming* : la méthode calculera le carré de chaque nombre reçu.
- C'est le client qui arrête le Stream.
- Tester le service en utilisant le client BloomRPC et remarquer que le serveur fonctionne en mode *Bidirectional Streaming* :



b. Développer le client

- Dans le projet tpgrpcclient, créer la classe **BidirectionnalModelClient** suivante :

```
package ma. formations. grpc;

import io. grpc. ManagedChannel;
import io. grpc. ManagedChannelBuilder;
import io. grpc. stub. StreamObserver;
import ma. formations. grpc. stubs. Calculator;
import ma. formations. grpc. stubs. CalculatorServiceGrpc;

import java. util. Timer;
import java. util. TimerTask;

public class BidirectionnalModelClient {

    public static void main(String[] args) {
        ManagedChannel channel = ManagedChannelBuilder.
            forAddress("localhost", 9999)
            .usePlaintext().
            build();

        CalculatorServiceGrpc. CalculatorServiceStub aysncStub =
        CalculatorServiceGrpc. newStub(channel);
        StreamObserver fullStream = aysncStub. fullStream(new
        StreamObserver< Calculator. BidirectionalStreamResponse>() {
            @Override
            public void onNext(Calculator. BidirectionalStreamResponse operationResponse) {
                System. out. println(operationResponse);
                System. out. println("#####");
            }
        });

        @Override
```

```

    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

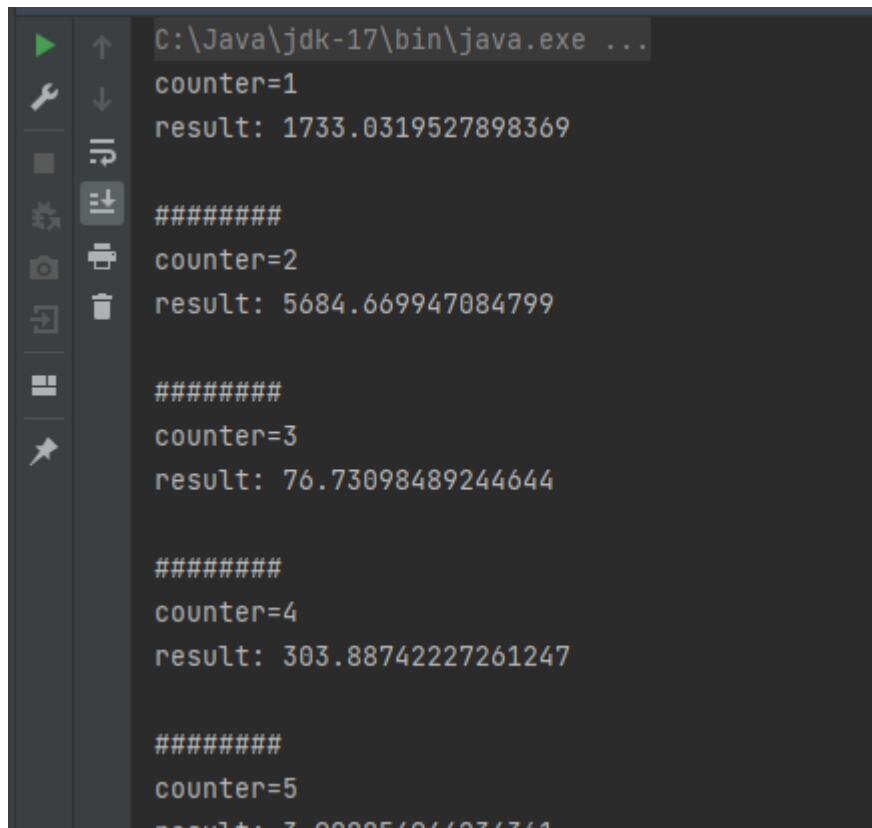
    @Override
    public void onCompleted() {
        System.out.println("FIN");
    }
});

Timer timer = new Timer();
timer.schedule(new TimerTask() {
    int counter = 0;

    @Override
    public void run() {
        Calculator.BidirectionalStreamRequest bidirectionalStreamRequest = Calculator.
            BidirectionalStreamRequest.
            newBuilder().
            setA(Math.random() * 100).
            build();
        fullStream.onNext(bidirectionalStreamRequest);
        counter++;
        System.out.println("counter=" + counter);
        if (counter == 10) {
            fullStream.onCompleted();
            timer.cancel();
        }
    }
}, 1000, 1000);
}
}

```

- Exécuter la méthode main ci-dessus et observer le résultat :



```
C:\Java\jdk-17\bin\java.exe ...  
counter=1  
result: 1733.0319527898369  
  
#####  
counter=2  
result: 5684.669947084799  
  
#####  
counter=3  
result: 76.73098489244644  
  
#####  
counter=4  
result: 303.88742227261247  
  
#####  
counter=5  
result: 7.088851966276761
```