gRPC

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

gRPC – Table of Contents

tutorialspoint
SIMPLYEASYLEARNING

Overview

gRPC is a framework from Google. It provides an efficient and language-independent way to make Remote Procedure Calls. It supports remote procedure calls from languages like Java, Python, Go, Dart, etc. It is heavily used for creating efficient remote procedure calls across industries by various companies.

The major use-case for gRPC is the making of remote procedure calls performant while ensuring language independence. Remote procedure calls play an important role in microservices/distributed environment where a lot of data is transferred across services. That is why, it becomes a very useful framework in developing applications that require high scalability and performance.

Audience

This tutorial deep dives into various components that make gRPC a very useful library. It is directed towards software professionals who want to develop highly scalable and performant applications. Post this tutorial, you would have an intermediate knowledge of gRPC and its usage.

Prerequisite

To learn from this tutorial, you need to have a good hold over Java or Python and a basic knowledge of data structure is preferable.

gRPC – Introduction

Before we jump into gRPC, let us have a brief background about remote procedure calls which is what gRPC does.

What are Remote Procedure Calls?

Remote procedure calls are the function calls which look like general/local function calls but differ in the fact that the execution of remote functions calls typically take place on a different machine. However, for the developer writing the code, there is minimal difference between a function call and a remote call. The calls typically follow the client-server model, where the machine which executes the call acts as the server.

Why do we need remote procedure calls?

Remote procedure calls provide a way to execute codes on another machine. It becomes of utmost importance in big, bulky products where a single machine cannot host all the code which is required for the overall product to function.

In microservice architecture, the application is broken down into small services and these services communicate with each other via messaging queue and APIs. And all of this communication takes place over a network where different machines/nodes serve different functionality based on the service they host. So, creating remote procedure calls becomes a critical aspect when it comes to working in a distributed environment.

Why gRPC?

Google Remote Procedure Calls (gRPC) provides a framework to perform the remote procedure calls. But there are some other libraries and mechanisms to execute code on remote machine. So, what makes gRPC special? Let's find out.

- **Language independent**: gRPC uses Google Protocol Buffer internally. So, multiple languages can be used such as Java, Python, Go, Dart, etc. A Java client can make a procedure call and a server that uses Python can respond, effectively, achieving language independence.

- **Efficient Data Compaction**: In microservice environment, given that multiple communications take place over a network, it is critical that the data that we are sending is as succinct as possible. We need to avoid any superfluous data to ensure that the data is quickly transferred. Given that gRPC uses Google Protocol Buffer internally, it has this feature to its advantage.

- **Efficient serialization and deserialization**: In microservice environment, given that multiple communications take place over a network, it is critical that we serialize and deserialize the data as quickly as possible. Given that gRPC uses Google Protocol Buffer internally, it ensures quick serializing and deserializing of data.

- **Simple to use**: gRPC already has a library and plugins that auto-generate procedure code (as we will see in the upcoming chapters). For simple use-cases, it can be used as local function calls.

gRPC vs REST using JSON

Let's take a look at how other ways to transfer data over a network stack up against Protobuf.

Feature	gRPC	HTTP using JSON/XML
Language independent	Yes	Yes
HTTP version	HTTP/2	HTTP 1.1
Specifying Domain Schema	.proto files (Google Protocol Buffer)	None
Serialized data size	Least	High (higher for XML)
Human Readable	No, as it uses separate encoding schema	Yes, as it uses text based format
Serialization speed	Fastest	Slower (slowest for XML)
Data type support	Richer. Supports complex data types like Any, one of etc.	Supports basic data types
Support for evolving schema	Yes	No

gRPC – Setup

Protoc Setup

Note that the setup is required only for Python. For Java, all of this is handled by Maven file. Let us install the "proto" binary which we will use to auto-generate the code of our ".proto" files. The binaries can be found at https://github.com/protocolbuffers/protobuf/releases/. Choose the correct binary based on the OS. We will install the proto binary on Windows, but the steps are not very different for Linux.

Once installed, ensure that you are able to access it via command line:

```
protoc --version

libprotoc 3.15.6
```

It means that Protobuf is correctly installed. Now, let us move to the Project Structure.

We also need to setup the plugin required for gRPC code generation.

For Python, we need to execute the following commands:

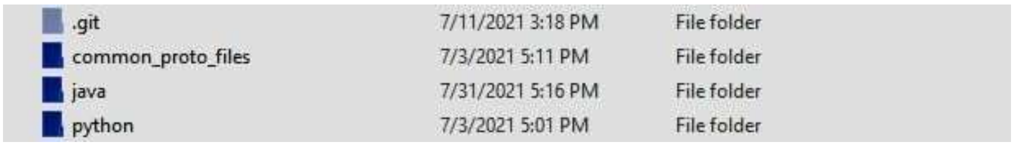
```
python -m pip install grpcio

python -m pip install grpcio-tools
```

It will install all the required binaries and add them to the path.

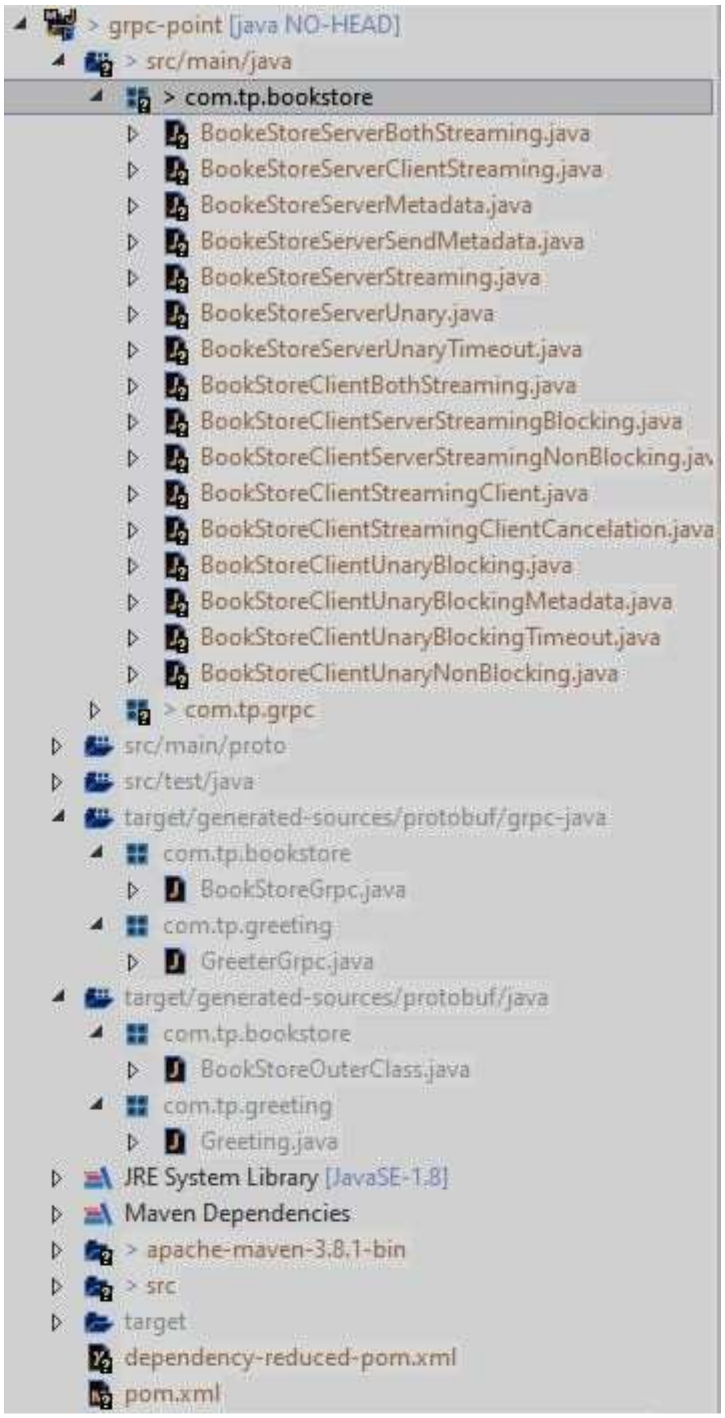
Project Structure

Here is the overall project structure that we would have:



```
.git                 7/11/2021 3:18 PM    File folder
common_proto_files   7/3/2021 5:11 PM     File folder
java                 7/31/2021 5:16 PM    File folder
python               7/3/2021 5:01 PM     File folder
```

The code related to individual languages go into their respective directories. We will have a separate directory to store our proto files. And, here is the project structure that we would be having for Java:



```
grpc-point [java NO-HEAD]
  src/main/java
    com.tp.bookstore
      BookeStoreServerBothStreaming.java
      BookeStoreServerClientStreaming.java
      BookeStoreServerMetadata.java
      BookeStoreServerSendMetadata.java
      BookeStoreServerStreaming.java
      BookeStoreServerUnary.java
      BookeStoreServerUnaryTimeout.java
      BookStoreClientBothStreaming.java
      BookStoreClientServerStreamingBlocking.java
      BookStoreClientServerStreamingNonBlocking.jav
      BookStoreClientStreamingClient.java
      BookStoreClientStreamingClientCancelation.java
      BookStoreClientUnaryBlocking.java
      BookStoreClientUnaryBlockingMetadata.java
      BookStoreClientUnaryBlockingTimeout.java
      BookStoreClientUnaryNonBlocking.java
    com.tp.grpc
  src/main/proto
  src/test/java
  target/generated-sources/protobuf/grpc-java
    com.tp.bookstore
      BookStoreGrpc.java
    com.tp.greeting
      GreeterGrpc.java
  target/generated-sources/protobuf/java
    com.tp.bookstore
      BookStoreOuterClass.java
    com.tp.greeting
      Greeting.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  apache-maven-3.8.1-bin
  src
  target
  dependency-reduced-pom.xml
  pom.xml
```

Project Dependency

Now that we have installed **protoc**, we can auto-generate the code from the proto files using **protoc**. Let us first create a Java project.

Following is the Maven configuration that we will use for our Java project. Note that it contains the required library for Protobuf as well.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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>com.grpc.point</groupId>
  <artifactId>grpc-point</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>

  <properties>
     <maven.compiler.source>1.8</maven.compiler.source>
     <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
     <dependency>
        <groupId>io.grpc</groupId>
        <artifactId>grpc-netty-shaded</artifactId>
        <version>1.38.0</version>
     </dependency>
     <dependency>
        <groupId>io.grpc</groupId>
```

```
      <artifactId>grpc-protobuf</artifactId>
      <version>1.38.0</version>
    </dependency>
    <dependency>
      <groupId>io.grpc</groupId>
      <artifactId>grpc-stub</artifactId>
      <version>1.38.0</version>
    </dependency>
    <dependency> <!-- necessary for Java 9+ -->
      <groupId>org.apache.tomcat</groupId>
      <artifactId>annotations-api</artifactId>
      <version>6.0.53</version>
      <scope>provided</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-
simple -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-simple</artifactId>
      <version>1.7.30</version>
    </dependency>
  </dependencies>

  <build>
    <extensions>
      <extension>
        <groupId>kr.motd.maven</groupId>
        <artifactId>os-maven-plugin</artifactId>
        <version>1.6.2</version>
      </extension>
    </extensions>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>build-helper-maven-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <id>test</id>
            <phase>generate-sources</phase>
            <goals>
              <goal>add-source</goal>
            </goals>
            <configuration>
              <sources>
                <source>${basedir}/target/generated-
sources</source>
              </sources>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.xolstice.maven.plugins</groupId>
        <artifactId>protobuf-maven-plugin</artifactId>
        <version>0.6.1</version>
        <configuration>
```

```
  <protocArtifact>com.google.protobuf:protoc:3.12.0:exe:${os.d
etected.classifier}</protocArtifact>
           <pluginId>grpc-java</pluginId>
           <pluginArtifact>io.grpc:protoc-gen-grpc-
java:1.38.0:exe:${os.detected.classifier}</pluginArtifact>

  <protoSourceRoot>../common_proto_files</protoSourceRoot>
         </configuration>
         <executions>
           <execution>
             <goals>
               <goal>compile</goal>
               <goal>compile-custom</goal>
             </goals>
           </execution>
         </executions>
       </plugin>
       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-shade-plugin</artifactId>
         <version>3.2.4</version>
         <configuration>
         </configuration>
         <executions>
           <execution>
             <phase>package</phase>
             <goals>
               <goal>shade</goal>
             </goals>
```

```
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

gRPC – Hello World App with Java

Let us now create a basic "Hello World" like app that will use gRPC along with Java.

.proto file

First let us define the "**`greeting.proto`**" file in **`common_proto_files`**:

```
syntax = "proto3";


option java_package = "com.tp.greeting";


service Greeter {
  rpc greet (ClientInput) returns (ServerOutput) {}
}


message ClientInput {
  string greeting = 1;
  string name = 2;
}


message ServerOutput {
  string message = 1;
}
```

Let us now take a closer look at this code block and see the role of each line:

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

The "syntax" here represents the version of Protobuf that we are using. So, we are using the latest version 3 and the schema thus can use all the syntax which is valid for version 3.

```
package tutorial;
```

The package here is used for conflict resolution if, say, we have multiple classes/members with the same name.

```
option java_package = "com.tp.greeting";
```

This argument is specific to Java, i.e., the package where to auto-generate the code from the ".proto" file.

```
service Greeter {
  rpc greet(ClientInput) returns (ServerOutput) {}
}
```

This block represents the name of the service "**`Greeter`**" and the function name "**`greet`**" which can be called. The "**`greet`**" function takes in the input of type "**`ClientInput`**" and returns the output of type "**`ServerOutput`**". Now let us look at these types.

```
message ClientInput {
  string greeting = 1;
  string name = 2;
}
```

In the above block, we have defined the **`ClientInput`** which contains two attributes, "**`greeting`**" and the "**`name`**", both of them being **`strings`**. The client is supposed to send the object of type "**`ClientInput`**" to the server.

```
message ServerOutput {
  string message = 1;
}
```

In the above block, we have defined that, given a "**`ClientInput`**", the server would return the "**`ServerOutput`**" which will contain a single attribute "**`message`**". The server is supposed to send the object of type "**`ServerOutput`**" to the client.

Note that we already had the Maven setup done to auto-generating our class files as well as our RPC code. So, now we can simply compile our project:

```
mvn clean install
```

This should auto-generate the source code required for us to use gRPC. The source code would be placed under:

```
Protobuf class code: target/generated-
sources/protobuf/java/com.tp.greeting

Protobuf gRPC code: target/generated-sources/protobuf/grpc-
java/com.tp.greeting
```

. Setting up gRPC server

Now that we have defined the proto file which contains the function definition, let us setup a server which can serve to call these functions.

Let us write our server code to serve the above function and save it in **`com.tp.grpc.GreetServer.java`**:

```
package com.tp.grpc;


import io.grpc.Server;

import io.grpc.ServerBuilder;

import io.grpc.stub.StreamObserver;

import java.io.IOException;

import java.util.concurrent.TimeUnit;

import java.util.logging.Logger;



import com.tp.greeting.GreeterGrpc;

import com.tp.greeting.Greeting.ClientInput;

import com.tp.greeting.Greeting.ServerOutput;


public class GreetServer {

  private static final Logger logger =
Logger.getLogger(GreetServer.class.getName());


  private Server server;
  private void start() throws IOException {
    int port = 50051;
    server = ServerBuilder.forPort(port)
        .addService(new GreeterImpl()).build().start();

    logger.info("Server started, listening on " + port);

    Runtime.getRuntime().addShutdownHook(new Thread() {
      @Override
      public void run() {
        System.err.println("Shutting down gRPC server");
        try {
            server.shutdown().awaitTermination(30,
TimeUnit.SECONDS);
        } catch (InterruptedException e) {
          e.printStackTrace(System.err);
        }
      }
    });
  }

  static class GreeterImpl extends
GreeterGrpc.GreeterImplBase {

    @Override
    public void greet(ClientInput req,
StreamObserver<ServerOutput> responseObserver) {
      logger.info("Got request from client: " + req);
      ServerOutput reply =
ServerOutput.newBuilder().setMessage("Server says " +
              "\"" + req.getGreeting() + " " + req.getName()
+ "\"").build();
      responseObserver.onNext(reply);
```

```
        responseObserver.onCompleted();
      }
    }

    public static void main(String[] args) throws IOException,
InterruptedException {
      final GreetServer greetServer = new GreetServer();
      greetServer.start();
      greetServer.server.awaitTermination();
    }
}
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our proto file. Let us walk through the above code:

1. Starting from the **`main`** method, we create a gRPC server at a specified port.

2. But before starting the server, we assign the server the service which we want to run, i.e., in our case, the **`Greeter`** service.

3. For this purpose, we need to pass the service instance to the server, so we go ahead and create a service instance, i.e., in our case, the **`GreeterImpl`**.

4. The service instance needs to provide an implementation of the method/function which is present in the "**`.proto`**" file, i.e., in our case, the **`greet`** method.

5. The method expects an object of type as defined in the "**`.proto`**" file, i.e., in our case, the **`ClientInput`**.

6. The method works on the above input, does the computation, and then is supposed to return the mentioned output in the "**`.proto`**" file, i.e., in our case, the **`ServerOutput`**.

7. Finally, we also have a **`shutdown hook`** to ensure clean shutting down of the server when we are done executing our code.

. Setting up gRPC client

Now that we have written the code for the server, let us setup a client which can call these functions.

Let us write our client code to call the above function and save it in **`com.tp.grpc.GreetClient.java`**:

```
package com.tp.grpc;


import io.grpc.Channel;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;


import com.tp.greeting.GreeterGrpc;
import com.tp.greeting.Greeting.ServerOutput;
import com.tp.greeting.Greeting.ClientInput;


public class GreetClient {
  private static final Logger logger =
Logger.getLogger(GreetClient.class.getName());


  private final GreeterGrpc.GreeterBlockingStub blockingStub;


  public GreetClient(Channel channel) {
      blockingStub = GreeterGrpc.newBlockingStub(channel);
  }


  public void makeGreeting(String greeting, String username) {
    logger.info("Sending greeting to server: " + greeting + "
for name: " + username);
```

```
    ClientInput request =
ClientInput.newBuilder().setName(username).setGreeting(greeti
ng).build();

    logger.info("Sending to server: " + request);

    ServerOutput response;
    try {
      response = blockingStub.greet(request);
    } catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
      return;
    }
    logger.info("Got following from the server: " +
response.getMessage());
  }


  public static void main(String[] args) throws Exception {
    String greeting = args[0];
    String username = args[1];
    String serverAddress = "localhost:50051";

    ManagedChannel channel =
ManagedChannelBuilder.forTarget(serverAddress)
        .usePlaintext()
        .build();

    try {
      GreetClient client = new GreetClient(channel);
      client.makeGreeting(greeting, username);
    } finally {
      channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
```

```
        }
    }
}
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our **proto** file. Let us walk through the above code:

1. Starting from **main** method, we accept two arguments, i.e., **name** and the **greeting**.

2. We setup a Channel for gRPC communication to our server.

3. Next, we create a blocking stub using the channel we created. This is where we have the service "**Greeter**" whose functions we plan to call. A **stub** is nothing but a wrapper that hides the remote call complexity from the caller.

4. Then, we simply create the expected input defined in the "**.proto**" file, i.e., in our case, the **ClientInput**.

5. We ultimately make the call and await result from the server.

6. Finally, we close the channel to avoid any resource leak.

So, that is our client code.

. Client server call

Now, that we have defined our **proto** file, written our server, and the client code, let us proceed and execute this code and see things in actions.

For running the code, fire up two shells. Start the server on the first shell by executing the following command:

```
java -cp .\target\grpc-point-1.0.jar com.tp.grpc.GreetServer
```

We would see the following output:

```
Jul 03, 2021 1:04:23 PM com.tp.grpc.GreetServer start

INFO: Server started, listening on 50051
```

The above output implies that the server has started.

Now, let us start the client.

```
java -cp .\target\grpc-point-1.0.jar com.tp.grpc.GreetClient
Hello Jane
```

We would see the following output:

```
Jul 03, 2021 1:05:59 PM com.tp.grpc.GreetClient greet

INFO: Sending greeting to server: Hello for name: Jane

Jul 03, 2021 1:05:59 PM com.tp.grpc.GreetClient greet

INFO: Sending to server: greeting: "Hello"

name: "Jane"


Jul 03, 2021 1:06:00 PM com.tp.grpc.GreetClient greet

INFO: Got following from the server: Server says "Hello Jane"
```

And now, if we open the server logs, we will get to see the following:

```
Jul 03, 2021 1:04:23 PM com.tp.grpc.GreetServer start

INFO: Server started, listening on 50051

Jul 03, 2021 1:06:00 PM com.tp.grpc.GreetServer$GreeterImpl
greet

INFO: Got request from client: greeting: "Hello"

name: "Jane"
```

So, the client was able to call the server as expected and the server responded with greeting the client back.

gRPC – Hello World App with Python

Let us now create a basic "Hello World" like app that will use gRPC along with Python.

.proto file

First let us define the **greeting.proto** file in **common_proto_files**:

```
syntax = "proto3";


service Greeter {
  rpc greet (ClientInput) returns (ServerOutput) {}
}


message ClientInput {
  string greeting = 1;
  string name = 2;
}


message ServerOutput {
  string message = 1;
}
```

Let us now take a closer look at each of the lines in the above block:

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

The "**syntax**" here represents the version of Protobuf we are using. So, we are using the latest version 3 and the schema thus can use all the syntax which is valid for version 3.

```
package tutorial;
```

The **package** here is used for conflict resolution if, say, we have multiple classes/members with the same name.

```
service Greeter {
  rpc greet(ClientInput) returns (ServerOutput) {}
}
```

This block represents the name of the service "**Greeter**" and the function name "**greet**" which can be called. The "**greet**" function takes in the input of type "**ClientInput**" and returns the output of type "**ServerOutput**". Now let us look at these types.

```
message ClientInput {
  string greeting = 1;
  string name = 2;
}
```

In the above block, we have defined the **ClientInput** which contains two attributes, "**greeting**" and the "**name**" both of them being strings. The client is supposed to send the object of type of "**ClientInput**" to the server.

```
message ServerOutput {
  string message = 1;
}
```

Here, we have also defined that, given a "**ClientInput**", the server would return the "**ServerOutput**" with a single attribute "**message**". The server is supposed to send the object of type "**ServerOutput**" to the client.

Now, let us generate the underlying code for the Protobuf classes and the gRPC classes. For doing that, we need to execute the following command:

```
python -m grpc_tools.protoc -I ..\common_proto_files\ --
python_out=../python --grpc_python_out=. greeting.proto
```

However, note that to execute the command, we need to install the correct dependency as mentioned in the **setup** section of the tutorial.

This should auto-generate the source code required for us to use gRPC. The source code would be placed under:

```
Protobuf class code: python/greeting_pb2.py

Protobuf gRPC code: python/greeting_pb2_grpcpb2.py
```

. Setting up gRPC server

Now that we have defined the **`proto`** file which contains the function definition, let us setup a server which can call these functions.

Let us write our server code to serve the above function and save it in **`server.py`**:

```
from concurrent import futures


import grpc


import greeting_pb2

import greeting_pb2_grpc


class Greeter(greeting_pb2_grpc.GreeterServicer):


    def greet(self, request, context):

        print("Got request " + str(request))

        return greeting_pb2.ServerOutput(message='{0}
{1}!'.format(request.greeting, request.name))


def server():

    server = grpc.server(futures.ThreadPoolExecutor(max_workers=2))

    greeting_pb2_grpc.add_GreeterServicer_to_server(Greeter(), server)

    server.add_insecure_port('[::]:50051')

    print("gRPC starting")

    server.start()
    server.wait_for_termination()


server()
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our **proto** file. Let us walk through the above code:

1. Starting from **main** method, we create a gRPC server at a specified port.

2. But before starting the server, we assign the server the service which we want to run, i.e., in our case, the **Greeter** service.

3. For this purpose, we need to pass the service instance to the server, so we go ahead and create a service instance, i.e., in our case, the **Greeter**.

4. The service instance need to provide an implementation of the method/function which is present in the **.proto** file, i.e., in our case, the **greet** method.

5. The method expects the an object of type as defined in the **.proto** file, i.e., for us, the **request**.

6. The method works on the above input, does the computation, and then is supposed to return the mentioned output in the **.proto** file, i.e., in our case, the **ServerOutput**.

. Setting up gRPC client

Now that we have written the code for the server, let us setup a client which can call these functions.

Let us write our client code to call the above function and save it in **client.py**:

```
import grpc


import greeting_pb2
import greeting_pb2_grpc

```

```
def run():
    with grpc.insecure_channel('localhost:50051') as channel:
        stub = greeting_pb2_grpc.GreeterStub(channel)
        response = stub.greet(greeting_pb2.ClientInput(name='John',
greeting = "Yo"))
    print("Greeter client received following from server: " +
response.message)


run()
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our **proto** file. Let us walk through the above code:

1. Starting from the **main** method, we have setup a Channel for gRPC communication with our server.

2. And then, we create a **stub** using the channel. This is where we use the service "**Greeter**" whose functions we plan to call. A stub is nothing but a wrapper which hides the complexity of the remote call from the caller.

3. Then, we simply create the expected input defined in the **proto** file, i.e., in our case, the **ClientInput**. We have hard-coded two arguments, i.e., **name** and the **greeting**.

4. We ultimately make the call and await the result from the server.

So, that is our client code.

. Client server call

Now, that we have defined our **proto** file, written our server, and the client code, let us proceed and execute this code and see things in action.

For running the code, fire up two shells. Start the server on the first shell by executing the following command:

```
python .\server.py
```

We would get the following output:

```
gRPC starting
```

The above output means that the server has started.

Now, let us start the client.

```
python .\client.py
```

We would see the following output:

```
Greeter client received following from server: Yo John!
```

And now, if we open the server logs, we will get to see the following data:

```
gRPC starting

Got request greeting: "Yo"

name: "John"
```

So, as we see, the client was able to call the server as expected and the server responded with greeting the client back.

gRPC – Unary gRPC

We will now look at various types of communication that the gRPC framework supports. We will use an example of **`Bookstore`** where the client can search and place an order for book delivery.

Let us see **`unary gRPC communication`** where we let the client search for a title and return randomly one of the book matching the title queried for.

.proto file

First let us define the **`bookstore.proto`** file in **`common_proto_files`**:

```
syntax = "proto3";

option java_package = "com.tp.bookstore";

service BookStore {
  rpc first (BookSearch) returns (Book) {}
}

message BookSearch {
  string name = 1;
  string author = 2;
  string genre = 3;
}

message Book {
  string name = 1;
  string author = 2;
  int32 price = 3;
}
```

Let us now take a closer look at each of the lines in the above block.

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

The "syntax" here represents the version of Protobuf we are using. We are using the latest version 3 and the schema thus can use all the syntax which is valid for version 3.

```
package tutorial;
```

The **package** here is used for conflict resolution if, say, we have multiple classes/members with the same name.

```
option java_package = "com.tp.bookstore";
```

This argument is specific to Java, i.e., the package where to auto-generate the code from the **.proto** file.

```
service BookStore {
  rpc first (BookSearch) returns (Book) {}
}
```

This represents the name of the service "**BookStore**" and the function name "**first**" which can be called. The "**first**" function takes in the input of type "**BookSearch**" and returns the output of type "**Book**". So, effectively, we let the client search for a title and return one of the book matching the title queried for.

Now let us look at these types.

```
message BookSearch {
  string name = 1;
  string author = 2;
  string genre = 3;
}
```

In the above block, we have defined the **BookSearch** which contains the attributes like **name**, **author**, and **genre**. The client is supposed to send the object of type of "**BookSearch**" to the server.

```
message Book {
  string name = 1;
  string author = 2;
  int32 price = 3;
}
```

Here, we have also defined that, given a "**BookSearch**", the server would return the "**Book**" which contains **book attributes** along with the price of the book. The server is supposed to send the object of type of "**Book**" to the client.

Note that we already had the Maven setup done for auto-generating our class files as well as our RPC code. So, now we can simply compile our project:

```
mvn clean install
```

This should auto-generate the source code required for us to use gRPC. The source code would be placed under:

```
Protobuf class code: target/generated-sources/protobuf/java/com.tp.bookstore

Protobuf gRPC code: target/generated-sources/protobuf/grpc-java/com.tp.bookstore
```

. Setting up gRPC server

Now that we have defined the **proto** file which contains the function definition, let us setup a server which can call these functions.

Let us write our server code to serve the above function and save it in **com.tp.bookstore.BookeStoreServerUnary.java**:

```
package com.tp.bookstore;


import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.stub.StreamObserver;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;
import java.util.stream.Collectors;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;


public class BookeStoreServerUnary {
  private static final Logger logger =
Logger.getLogger(BookeStoreServerUnary.class.getName());


  static Map<String, Book> bookMap = new HashMap<>();
  static {
      bookMap.put("Great Gatsby",
Book.newBuilder().setName("Great Gatsby")
              .setAuthor("Scott Fitzgerald")
              .setPrice(300).build());
      bookMap.put("To Kill MockingBird",
Book.newBuilder().setName("To Kill MockingBird")
              .setAuthor("Harper Lee")
              .setPrice(400).build());
      bookMap.put("Passage to India",
Book.newBuilder().setName("Passage to India")
              .setAuthor("E.M.Forster")
              .setPrice(500).build());
      bookMap.put("The Side of Paradise",
Book.newBuilder().setName("The Side of Paradise")
              .setAuthor("Scott Fitzgerald")
              .setPrice(600).build());
      bookMap.put("Go Set a Watchman",
Book.newBuilder().setName("Go Set a Watchman")
              .setAuthor("Harper Lee")
              .setPrice(700).build());
  }
  private Server server;

  private void start() throws IOException {
    int port = 50051;
    server = ServerBuilder.forPort(port)
        .addService(new BookStoreImpl()).build().start();

    logger.info("Server started, listening on " + port);

    Runtime.getRuntime().addShutdownHook(new Thread() {
      @Override
      public void run() {
        System.err.println("Shutting down gRPC server");
        try {
            server.shutdown().awaitTermination(30,
TimeUnit.SECONDS);
        } catch (InterruptedException e) {
          e.printStackTrace(System.err);
        }
      }
    });
  }

  public static void main(String[] args) throws IOException,
InterruptedException {
    final BookeStoreServerUnary greetServer = new
BookeStoreServerUnary();
    greetServer.start();
    greetServer.server.awaitTermination();
  }

  static class BookStoreImpl extends
BookStoreGrpc.BookStoreImplBase {
      @Override
    public void first(BookSearch searchQuery,
StreamObserver<Book> responseObserver) {
          logger.info("Searching for book with title: " +
searchQuery.getName());
          List<String> matchingBookTitles =
bookMap.keySet().stream()
                  .filter(title ->
title.startsWith(searchQuery.getName().trim()))
                  .collect(Collectors.toList());
          Book foundBook = null;
          if(matchingBookTitles.size() > 0) {
              foundBook =
bookMap.get(matchingBookTitles.get(0));
          }


          responseObserver.onNext(foundBook);
          responseObserver.onCompleted();
      }
  }
}
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our **proto** file. Let us walk through the above code:

1. Starting from the **main** method, we create a gRPC server at a specified port.

2. But before starting the server, we assign the server the service which we want to run, i.e., in our case, the **BookStore** service.

3. For this purpose, we need to pass the service instance to the server, so we go ahead and create a service instance, i.e., in our case, the **BookStoreImpl**

4. The service instance need to provide an implementation of the method/function which is present in the **`.proto`** file, i.e., in our case, the **`first`** method.

5. The method expects an object of type as defined in the **`.proto`** file, i.e., for us the **`BookSearch`**

6. The method searches for the book in the available **`bookMap`** and then returns the **`Book`** by calling the **`onNext()`** method. Once done, the server announces that it is done with the output by calling **`onCompleted()`**

7. Finally, we also have a shutdown hook to ensure clean shutting down of the server when we are done executing our code.

. Setting up gRPC client

Now that we have written the code for the server, let us setup a client which can call these functions.

Let us write our client code to call the above function and save it in **`com.tp.bookstore.BookStoreClientUnaryBlocking.java`**:

```
package com.tp.bookstore;


import io.grpc.Channel;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.greeting.GreeterGrpc;
import com.tp.greeting.Greeting.ServerOutput;
import com.tp.greeting.Greeting.ClientInput;


public class BookStoreClientUnaryBlocking {
  private static final Logger logger =
Logger.getLogger(BookStoreClientUnaryBlocking.class.getName());


  private final BookStoreGrpc.BookStoreBlockingStub
blockingStub;


  public BookStoreClientUnaryBlocking(Channel channel) {
      blockingStub = BookStoreGrpc.newBlockingStub(channel);
  }


  public void getBook(String bookName) {
    logger.info("Querying for book with title: " + bookName);
    BookSearch request =
BookSearch.newBuilder().setName(bookName).build();


    Book response;
    try {
      response = blockingStub.first(request);
    } catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}",
e.getStatus());
      return;
    }
    logger.info("Got following book from server: " +
response);
  }



  public static void main(String[] args) throws Exception {
    String bookName = args[0];
    String serverAddress = "localhost:50051";


    ManagedChannel channel =
ManagedChannelBuilder.forTarget(serverAddress)
```

```
        .usePlaintext()
        .build();


    try {
      BookStoreClientUnaryBlocking client = new
BookStoreClientUnaryBlocking(channel);
      client.getBook(bookName);
    } finally {
      channel.shutdownNow().awaitTermination(5,
TimeUnit.SECONDS);
    }
  }
}
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our **`proto`** file. Let us walk through the above code:

1. Starting from the **`main`** method, we accept one argument, i.e., the title of the book we want to search for.

2. We setup a Channel for gRPC communication with our server.

3. And then, we create a blocking stub using the channel. This is where we choose the service "**`BookStore`**" whose functions we plan to call. A "stub" is nothing but a wrapper which hides the complexity of the remote call from the caller.

4. Then, we simply create the expected input defined in the **`.proto`** file, i.e., in our case **`BookSearch`** and we add the title name we want the server to search for.

5. We ultimately make the call and await the result from the server.

6. Finally, we close the channel to avoid any resource leak.

So, that is our client code.

. Client server call

To sum up, what we want to do is the following:

- Start the gRPC server.
- The Client queries the Server for a book with a given name/title.
- The Server searches the book in its store.
- The Server then responds with the book and its other attributes.

Now, that we have defined our **`proto`** file, written our server and the client code, let us proceed to execute this code and see things in action.

For running the code, fire up two shells. Start the server on the first shell by executing the following command:

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookeStoreServerUnary
```

We would see the following output:

```
Jul 03, 2021 7:21:58 PM
com.tp.bookstore.BookeStoreServerUnary start

INFO: Server started, listening on 50051
```

The above output means the server has started.

Now, let us start the client.

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookStoreClientUnaryBlocking "To Kill"
```

We would see the following output:

```
Jul 03, 2021 7:22:03 PM
com.tp.bookstore.BookStoreClientUnaryBlocking getBook

INFO: Querying for book with title: To Kill

Jul 03, 2021 7:22:04 PM
com.tp.bookstore.BookStoreClientUnaryBlocking getBook

INFO: Got following book from server: name: "To Kill
MockingBird"

author: "Harper Lee"

price: 400
```

So, as we see, the client was able to get the book details by querying the server with the name of the book.

gRPC – Server Streaming RPC

Let us now discuss how server streaming works while using gRPC communication. In this case, the client will search for books with a given author. Assume the server requires some time to go through all the books. Instead of waiting to give all the books after going through all the books, the server instead would provide books in a streaming fashion, i.e., as soon as it finds one.

.proto file

First let us define the **bookstore.proto** file in **common_proto_files**:

```
syntax = "proto3";

option java_package = "com.tp.bookstore";

service BookStore {
  rpc searchByAuthor (BookSearch) returns (stream Book) {}
}

message BookSearch {
  string name = 1;
  string author = 2;
  string genre = 3;
}

message Book {
  string name = 1;
  string author = 2;
  int32 price = 3;
}
```

The following block represents the name of the service "**BookStore**" and the function name "**searchByAuthor**" which can be called. The "**searchByAuthor**" function takes in the input of type "**BookSearch**" and returns the stream of type "**Book**". So, effectively, we let the client search for a title and return one of the book matching the author queried for.

```
service BookStore {

  rpc searchByAuthor (BookSearch) returns (stream Book) {}

}
```

Now let us look at these types.

```
message BookSearch {

  string name = 1;

  string author = 2;

  string genre = 3;

}
```

Here, we have defined **BookSearch** which contains a few attributes like **name**, **author** and **genre**. The client is supposed to send the object of type "**BookSearch**" to the server.

```
message Book {

  string name = 1;

  string author = 2;

  int32 price = 3;

}
```

We have also defined that, given a "**BookSearch**", the server would return a stream of "**Book**" which contains the book attributes along with the price of the book. The server is supposed to send a stream of "Book".

Note that we already had the Maven setup done for auto-generating our class files as well as our RPC code. So, now we can simply compile our project:

```
mvn clean install
```

This should auto-generate the source code required for us to use gRPC. The source code would be placed under:

```
Protobuf class code: target/generated-
sources/protobuf/java/com.tp.bookstore
```

```
Protobuf gRPC code: target/generated-sources/protobuf/grpc-
java/com.tp.bookstore
```

. Setting up gRPC server

Now that we have defined the **proto** file which contains the function definition, let us setup a server which can serve call these functions.

Let us write our server code to serve the above function and save it in **com.tp.bookstore.BookeStoreServerStreaming.java**:

```
package com.tp.bookstore;


import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.stub.StreamObserver;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;
import java.util.stream.Collectors;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;


public class BookeStoreServerStreaming {
  private static final Logger logger =
Logger.getLogger(BookeStoreServerStreaming.class.getName());


  static Map<String, Book> bookMap = new HashMap<>();
  static {
      bookMap.put("Great Gatsby",
Book.newBuilder().setName("Great Gatsby")

              .setAuthor("Scott Fitzgerald")

              .setPrice(300).build());

      bookMap.put("To Kill MockingBird",
Book.newBuilder().setName("To Kill MockingBird")

              .setAuthor("Harper Lee")

              .setPrice(400).build());

      bookMap.put("Passage to India",
Book.newBuilder().setName("Passage to India")

              .setAuthor("E.M.Forster")

              .setPrice(500).build());

      bookMap.put("The Side of Paradise",
Book.newBuilder().setName("The Side of Paradise")

              .setAuthor("Scott Fitzgerald")

              .setPrice(600).build());

      bookMap.put("Go Set a Watchman",
Book.newBuilder().setName("Go Set a Watchman")

              .setAuthor("Harper Lee")

              .setPrice(700).build());

  }


  private Server server;


  private void start() throws IOException {

    int port = 50051;

    server = ServerBuilder.forPort(port)

        .addService(new BookStoreImpl()).build().start();


    logger.info("Server started, listening on " + port);


    Runtime.getRuntime().addShutdownHook(new Thread() {

      @Override
      public void run() {
        System.err.println("Shutting down gRPC server");
        try {
            server.shutdown().awaitTermination(30,
TimeUnit.SECONDS);
        } catch (InterruptedException e) {
          e.printStackTrace(System.err);
        }
      }
    });
  }

  public static void main(String[] args) throws IOException,
InterruptedException {
    final BookeStoreServerStreaming greetServer = new
BookeStoreServerStreaming();
    greetServer.start();
    greetServer.server.awaitTermination();
  }

  static class BookStoreImpl extends
BookStoreGrpc.BookStoreImplBase {

    @Override
    public void searchByAuthor(BookSearch searchQuery,
StreamObserver<Book> responseObserver) {
          logger.info("Searching for book with author: " +
searchQuery.getAuthor());
          for (Entry<String, Book> bookEntry :
bookMap.entrySet()) {
              try {
                  logger.info("Going through more
books....");
                  Thread.sleep(5000);
              } catch (InterruptedException e) {
                  e.printStackTrace();
              }

if(bookEntry.getValue().getAuthor().startsWith(searchQuery.ge
tAuthor())){
                  logger.info("Found book with required
author: " + bookEntry.getValue().getName()
                          + ". Sending....");

responseObserver.onNext(bookEntry.getValue());
                  }
              }
          responseObserver.onCompleted();
          }
      }
}
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our **`proto`** file. Let us walk through the above code:

1. Starting from the **`main`** method, we create a gRPC server at a specified port.

2. But before starting the server, we assign the server the service which we want to run, i.e., in our case, the **`BookStore`** service.

3. For this purpose, we need to pass the service instance to the server, so we go ahead and create a service instance, i.e., in our case, the **`BookStoreImpl`**

4. The service instance need to provide an implementation of the method/function which is present in the **`.proto`** file, i.e., in our case, the **`searchByAuthor`** method.

5. The method expects an object of type as defined in the **`.proto`** file, i.e., the **`BookSearch`**.

6. The method searches for the book in the available **`bookMap`** and then returns a stream of **`Books`**.

7. Note that we have added a **`sleep`** to mimic the operation of searching through all the books. In case of streaming, the server does not wait for all the searched books to be available. It returns the book as soon it is available by using the **`onNext()`** call.

8. When the server is done with the request, it shuts down the channel by calling **`onCompleted()`**.

9. Finally, we also have a shutdown hook to ensure clean shutting down of the server when we are done executing our code.

. Setting up gRPC client

Now that we have written the code for the server, let us setup a client which can call these functions.

Let us write our client code to call the above function and save it in **`com.tp.bookstore.BookStoreClientServerStreamingBlocking.java`**:

```
package com.tp.bookstore;


import io.grpc.Channel;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;


import java.util.Iterator;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.greeting.GreeterGrpc;
import com.tp.greeting.Greeting.ServerOutput;
import com.tp.greeting.Greeting.ClientInput;
```

```
public class BookStoreClientServerStreamingBlocking {
  private static final Logger logger =
Logger.getLogger(BookStoreClientServerStreamingBlocking.class
.getName());


  private final BookStoreGrpc.BookStoreBlockingStub
blockingStub;


  public BookStoreClientServerStreamingBlocking(Channel
channel) {
      blockingStub = BookStoreGrpc.newBlockingStub(channel);
  }


  public void getBook(String author) {
    logger.info("Querying for book with author: " + author);
    BookSearch request =
BookSearch.newBuilder().setAuthor(author).build();


    Iterator<Book> response;
    try {
        response = blockingStub.searchByAuthor(request);
        while(response.hasNext()) {
            logger.info("Found book: " + response.next());
        }
    } catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}",
e.getStatus());
      return;
    }
  }
  public static void main(String[] args) throws Exception {
    String authorName = args[0];
    String serverAddress = "localhost:50051";


    ManagedChannel channel =
ManagedChannelBuilder.forTarget(serverAddress)
        .usePlaintext()
        .build();


    try {
      BookStoreClientServerStreamingBlocking client = new
BookStoreClientServerStreamingBlocking(channel);
      client.getBook(authorName);
    } finally {
      channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
    }
  }
}
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our **proto** file. Let us walk through the above code:

1. Starting from the **main** method, we accept one argument, i.e., the **title** of the book we want to search for.

2. We setup a Channel for gRPC communication with our server.

3. And then, we create a **blocking stub** using the channel. This is where we choose the service "**BookStore**" whose functions we plan to call.

4. Then, we simply create the expected input defined in the **.proto** file, i.e., in our case, **BookSearch**, and we add the title that we want the server to search for.

5. We ultimately make the call and get an iterator on valid Books. When we iterate, we get the corresponding Books made available by the Server.

6. Finally, we close the channel to avoid any resource leak.

So, that is our client code.

. Client server call

To sum up, what we want to do the following:

- Start the gRPC server.
- The Client queries the Server for a book with a given author.
- The Server searches the book in its store which is a time-consuming process.
- The Server responds whenever it finds a book with the given criteria. The Server does not wait for all the valid books to be available. It sends the output as soon as it finds one. And then repeats the process.

Now, that we have defined our **proto** file, written our server and the client code, let us proceed and execute this code and see things in action.

For running the code, fire up two shells. Start the server on the first shell by executing the following command:

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookeStoreServerStreaming
```

We would see the following output:

```
Jul 03, 2021 10:37:21 PM
com.tp.bookstore.BookeStoreServerStreaming start

INFO: Server started, listening on 50051
```

The above output means the server has started.

Now, let us start the client.

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookStoreClientServerStreamingBlocking "Har"
```

We would see the following output.

```
Jul 03, 2021 10:40:31 PM
com.tp.bookstore.BookStoreClientServerStreamingBlocking
getBook

INFO: Querying for book with author: Har


Jul 03, 2021 10:40:37 PM
com.tp.bookstore.BookStoreClientServerStreamingBlocking
getBook

INFO: Found book: name: "Go Set a Watchman"

author: "Harper Lee"

price: 700


Jul 03, 2021 10:40:42 PM
com.tp.bookstore.BookStoreClientServerStreamingBlocking
getBook

INFO: Found book: name: "To Kill MockingBird"

author: "Harper Lee"

price: 400
```

So, as we see, the client was able to get the book details by querying the server with the name of the book. But more importantly, the client got the 1st book and the 2nd book at different timestamps, i.e., a gap of almost 5 seconds.

gRPC – Client Streaming RPC

Let us see now see how client streaming works while using gRPC communication. In this case, the client will search and add books to the cart. Once the client is done adding all the books, the server would provide the checkout cart value to the client.

.proto file

First let us define the **`bookstore.proto`** file in **`common_proto_files`**:

```
syntax = "proto3";

option java_package = "com.tp.bookstore";

service BookStore {
  rpc totalCartValue (stream Book) returns (Cart) {}
}

message Book {
  string name = 1;
  string author = 2;
  int32 price = 3;
}

message Cart {
  int32 books = 1;
  int32 price = 2;
}
```

Here, the following block represents the name of the service "**`BookStore`**" and the function name "**`totalCartValue`**" which can be called. The "**`totalCartValue`**" function takes in the input of type "**`Book`**" which is a stream. And the function returns an object of type "**`Cart`**". So, effectively, we let the client add books in a streaming fashion and once the client is done, the server provides the total cart value to the client.

```
service BookStore {
  rpc totalCartValue (stream Book) returns (Cart) {}
}
```

Now let us look at these types.

```
message Book {
  string name = 1;
  string author = 2;
  int32 price = 3;
}
```

The client would send in the "**Book**" it wants to buy. It may not be the complete book info; it can simply be the title of the book.

```
message Cart {
  int32 books = 1;
  int32 price = 2;
}
```

The server, on getting the list of books, would return the "**Cart**" object which is nothing but the total number of books the client has purchased and the total price.

Note that we already had the Maven setup done for auto-generating our class files as well as our RPC code. So, now we can simply compile our project:

```
mvn clean install
```

This should auto-generate the source code required for us to use gRPC. The source code would be placed under:

```
Protobuf class code: target/generated-sources/protobuf/java/com.tp.bookstore

Protobuf gRPC code: target/generated-sources/protobuf/grpc-java/com.tp.bookstore
```

. Setting up gRPC server

Now that we have defined the **proto** file which contains the function definition, let us setup a server which can serve call these functions.

Let us write our server code to serve the above function and save it in **com.tp.bookstore.BookeStoreServerClientStreaming.java**:

```
package com.tp.bookstore;


import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.stub.StreamObserver;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;
import java.util.stream.Collectors;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.bookstore.BookStoreOuterClass.Cart;


public class BookeStoreServerClientStreaming {
  private static final Logger logger =
Logger.getLogger(BookeStoreServerClientStreaming.class.getNam
e());


  static Map<String, Book> bookMap = new HashMap<>();
  static {
      bookMap.put("Great Gatsby",
Book.newBuilder().setName("Great Gatsby")
              .setAuthor("Scott Fitzgerald")
              .setPrice(300).build());
      bookMap.put("To Kill MockingBird",
Book.newBuilder().setName("To Kill MockingBird")
              .setAuthor("Harper Lee")
              .setPrice(400).build());
      bookMap.put("Passage to India",
Book.newBuilder().setName("Passage to India")
              .setAuthor("E.M.Forster")
              .setPrice(500).build());
      bookMap.put("The Side of Paradise",
Book.newBuilder().setName("The Side of Paradise")
              .setAuthor("Scott Fitzgerald")
              .setPrice(600).build());
      bookMap.put("Go Set a Watchman",
Book.newBuilder().setName("Go Set a Watchman")
              .setAuthor("Harper Lee")
              .setPrice(700).build());
  }

  private Server server;

  private void start() throws IOException {
    int port = 50051;
    server = ServerBuilder.forPort(port)
        .addService(new BookStoreImpl()).build().start();

    logger.info("Server started, listening on " + port);

    Runtime.getRuntime().addShutdownHook(new Thread() {
      @Override
      public void run() {
        System.err.println("Shutting down gRPC server");
        try {
            server.shutdown().awaitTermination(30,
TimeUnit.SECONDS);
        } catch (InterruptedException e) {
          e.printStackTrace(System.err);
        }
      }
    });
  }

  public static void main(String[] args) throws IOException,
InterruptedException {
    final BookeStoreServerClientStreaming greetServer = new
BookeStoreServerClientStreaming();
    greetServer.start();
    greetServer.server.awaitTermination();
  }

  static class BookStoreImpl extends
BookStoreGrpc.BookStoreImplBase {

    @Override
    public StreamObserver<Book>
totalCartValue(StreamObserver<Cart> responseObserver) {

        return new StreamObserver<Book>() {

            ArrayList<Book> bookCart = new ArrayList<Book>();

            @Override
            public void onNext(Book book) {
                logger.info("Searching for book with title
starting with: " + book.getName());


                for (Entry<String, Book> bookEntry :
bookMap.entrySet()) {



if(bookEntry.getValue().getName().startsWith(book.getName()))
{
                        logger.info("Found book, adding to
cart:....");
                        bookCart.add(bookEntry.getValue());
                        }
                    }
            }


            @Override
            public void onError(Throwable t) {
                logger.info("Error while reading book stream:
" + t);
            }


            @Override
            public void onCompleted() {
                int cartValue = 0;


                for (Book book : bookCart) {
                    cartValue += book.getPrice();
                }


                responseObserver.onNext(Cart.newBuilder()
                        .setPrice(cartValue)
                        .setBooks(bookCart.size()).build());
                responseObserver.onCompleted();


            }
        };


    }
  }
}
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our **`proto`** file. Let us walk through the above code:

1. Starting from the **`main`** method, we create a gRPC server at a specified port.

2. But before starting the server, we assign the server the service which we want to run, i.e., in our case, the **`BookStore`** service.

3. For this purpose, we need to pass in the service instance to the server, so we go ahead and create a service instance, i.e., in our case, the **`BookStoreImpl`**

4. The service instance needs to provide an implementation of the method/function which is present in the **`proto`** file, i.e., in our case the **`totalCartValue`** method.

5. Now, given that this is the case of client streaming, the server will get a list of Book (defined in the **`proto`** file) as the client adds them. The server thus returns a **`custom stream observer`**. This stream observer implements what happens when a new Book is found and what happens when the stream is closed.

6. The **`onNext()`** method would be called by the gRPC framework when the client adds a Book. At this point, the server adds that to the cart. In case of streaming, the server does not wait for all the books available.

7. When the client is done with the addition of Books, the stream observer's **`onCompleted()`** method is called. This method implements what the server wants to send when the client is done adding **`Book`**, i.e., it returns the **`Cart`** object to the client.

8. Finally, we also have a shutdown hook to ensure clean shutting down of the server when we are done executing our code.

. Setting up gRPC client

Now that we have written code for the server, let us setup a client which can call these functions.

Let us write our client code to call the above function and save it in **`com.tp.bookstore.BookStoreClientStreamingClient.java`**:

```
package com.tp.bookstore;


import io.grpc.Channel;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;
import io.grpc.stub.StreamObserver;


import java.util.Iterator;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;


import com.tp.bookstore.BookStoreGrpc.BookStoreFutureStub;
import com.tp.bookstore.BookStoreGrpc.BookStoreStub;
import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.bookstore.BookStoreOuterClass.Cart;
import com.tp.greeting.GreeterGrpc;
import com.tp.greeting.Greeting.ServerOutput;
import com.tp.greeting.Greeting.ClientInput;


public class BookStoreClientStreamingClient {
  private static final Logger logger =
Logger.getLogger(BookStoreClientStreamingClient.class.getName
());


  private final BookStoreStub stub;


  private boolean serverResponseCompleted = false;


  StreamObserver<Book> streamClientSender;


  public BookStoreClientStreamingClient(Channel channel) {
      stub = BookStoreGrpc.newStub(channel);
  }


  public StreamObserver<Cart> getServerResponseObserver(){
      StreamObserver<Cart> observer = new StreamObserver<Cart>(){


        @Override
        public void onNext(Cart cart) {
            logger.info("Order summary:" +
                    "\nTotal number of Books:" + cart.getBooks()+
                    "\nTotal Order Value:" + cart.getPrice());
        }


        @Override
        public void onError(Throwable t) {
            logger.info("Error while reading response from
Server: " + t);
        }


        @Override
        public void onCompleted() {
            //logger.info("Server: Done reading orderreading cart");
            serverResponseCompleted = true;
        }
      };
      return observer;
  }

  public void addBook(String book) {
    logger.info("Adding book with title starting with: " + book);
    Book request = Book.newBuilder().setName(book).build();

    if(streamClientSender == null) {
        streamClientSender =
stub.totalCartValue(getServerResponseObserver());
    }

    try {
           streamClientSender.onNext(request);
    }
    catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
    }
  }

    public void completeOrder() {
        logger.info("Done, waiting for server to create order
summary...");
        if(streamClientSender != null);
            streamClientSender.onCompleted();
  }



  public static void main(String[] args) throws Exception {
      String serverAddress = "localhost:50051";


      ManagedChannel channel = 
ManagedChannelBuilder.forTarget(serverAddress)
          .usePlaintext()
          .build();


      try {
        BookStoreClientStreamingClient client = new 
BookStoreClientStreamingClient(channel);
        String bookName = "";


        while(true) {
            System.out.println("Type book name to be added to 
the cart....");
            bookName = System.console().readLine();
            if(bookName.equals("EXIT")) {
                client.completeOrder();
                break;
            }
            client.addBook(bookName);
        }


        while(client.serverResponseCompleted == false) {
            Thread.sleep(2000);
        }


      } finally {
        channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
      }
    }
}
```

The above code starts a gRPC client and connects to a server at a specified port and call the service and functions which we had written in our **`proto`** file. Let us walk through the above code:

1. Starting from the **`main`** method, we accept the name of the books to be added to the cart. Once all the books are to added, the user is expected to print "EXIT".

2. We setup a Channel for gRPC communication with our server.

3. Next, we create a **`non-blocking stub`** using the channel we created. This is where we are choosing the service "**`BookStore`**" whose functions we plan to call.

4. Then, we simply create the expected input defined in the **`proto`** file, i.e., in our case, **`Book`**, and we add the title name we want the server to add.

5. But given this is the case of client streaming, we first create a stream observer for the server. This server stream observer lists the behavior on what needs to be done when the server responds, i.e., **`onNext()`** and **`onCompleted()`**

6. And using the stub, we also get the client stream observer. We use this stream observer for sending the data, i.e., **`Book`**, to be added to the cart. We ultimately, make the call and get an iterator on valid Books. When we iterate, we get the corresponding Books made available by the Server.

7. And once our order is complete, we ensure that the client stream observer is closed. It tells the server to calculate the Cart Value and provide that as an output

8. Finally, we close the channel to avoid any resource leak.

So, that is our client code.

. Client server call

To sum up, what we want to do is the following:

- Start the gRPC server.
- The Client adds a stream of books by notifying them to the server.
- The Server searches the book in its store and adds them to the cart.

- When the client is done ordering, the Server responds the total cart value of the client.

Now that we have defined our **`proto`** file, written our server and the client code, let us proceed to execute this code and see things in action.

For running the code, fire up two shells. Start the server on the first shell by executing the following command:

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookeStoreServerClientStreaming
```

We would see the following output.

```
Jul 03, 2021 10:37:21 PM
com.tp.bookstore.BookeStoreServerStreaming start

INFO: Server started, listening on 50051
```

The above output means server has started.

Now, let us start the client.

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookStoreClientStreamingClient
```

Let us add a few books to our client.

```
Type book name to be added to the cart....

Gr

Jul 24, 2021 5:53:07 PM
com.tp.bookstore.BookStoreClientStreamingClient addBook

INFO: Adding book with title starting with: Great


Type book name to be added to the cart....

Pa

Jul 24, 2021 5:53:20 PM
com.tp.bookstore.BookStoreClientStreamingClient addBook

INFO: Adding book with title starting with: Passage


Type book name to be added to the cart....
```

Once we have added the books and we input "EXIT", the server then calculates the cart value and here is the output we get:

```
EXIT

Jul 24, 2021 5:53:33 PM
com.tp.bookstore.BookStoreClientStreamingClient completeOrder

INFO: Done, waiting for server to create order summary...

Jul 24, 2021 5:53:33 PM
com.tp.bookstore.BookStoreClientStreamingClient$1 onNext

INFO: Order summary:

Total number of Books: 2

Total Order Value: 800
```

So, as we can see, the client was able to add books. And once all the books were added, the server responds with the total number of books and the total price.

Let us see now see how the client-server streaming works while using gRPC communication. In this case, the client will search and add books to the cart. The server would respond with live cart value every time a book is added.

.proto file

First let us define the **`bookstore.proto`** file in **`common_proto_files`**:

```
syntax = "proto3";

option java_package = "com.tp.bookstore";

service BookStore {
  rpc liveCartValue (stream Book) returns (stream Cart) {}
}

message Book {
  string name = 1;
  string author = 2;
  int32 price = 3;
}

message Cart {
  int32 books = 1;
  int32 price = 2;
}
```

The following block represents the name of the service "**`BookStore`**" and the function name "**`liveCartValue`**" which can be called. The "**`liveCartValue`**" function takes in the input of type "**`Book`**" which is a stream. And the function returns a stream of object of type "**`Cart`**". So, effectively, we let the client add books in a streaming fashion and whenever a new book is added, the server responds the current cart value to the client.

```
service BookStore {
  rpc liveCartValue (stream Book) returns (stream Cart) {}
}
```

Now let us look at these types.

```
message Book {
  string name = 1;
  string author = 2;
  int32 price = 3;
}
```

The client would send in the "**Book**" it wants to buy. It does not have to be the complete book info; it can simply be title of the book.

```
message Cart {
  int32 books = 1;
  int32 price = 2;
}
```

The server, on getting the list of books, would return the "**Cart**" object which is nothing but the total number of books the client has purchased and the total price.

Note that we already had the Maven setup done for auto-generating our class files as well as our RPC code. So, now we can simply compile our project:

```
mvn clean install
```

This should auto-generate the source code required for us to use gRPC. The source code would be placed under:

```
Protobuf class code: target/generated-
sources/protobuf/java/com.tp.bookstore

Protobuf gRPC code: target/generated-sources/protobuf/grpc-
java/com.tp.bookstore
```

. Setting up gRPC server

Now that we have defined the **proto** file which contains the function definition, let us setup a server which can call these functions.

Let us write our server code to serve the above function and save it in **com.tp.bookstore.BookeStoreServerBothStreaming.java**:

```
package com.tp.bookstore;


import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.stub.StreamObserver;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;
import java.util.stream.Collectors;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.bookstore.BookStoreOuterClass.Cart;


public class BookeStoreServerBothStreaming {
  private static final Logger logger =
Logger.getLogger(BookeStoreServerBothStreaming.class.getName(
));


  static Map<String, Book> bookMap = new HashMap<>();
  static {
      bookMap.put("Great Gatsby",
Book.newBuilder().setName("Great Gatsby")
              .setAuthor("Scott Fitzgerald")
              .setPrice(300).build());
      bookMap.put("To Kill MockingBird",
Book.newBuilder().setName("To Kill MockingBird")
              .setAuthor("Harper Lee")
              .setPrice(400).build());
      bookMap.put("Passage to India",
Book.newBuilder().setName("Passage to India")
              .setAuthor("E.M.Forster")
              .setPrice(500).build());
      bookMap.put("The Side of Paradise",
Book.newBuilder().setName("The Side of Paradise")
              .setAuthor("Scott Fitzgerald")
              .setPrice(600).build());
      bookMap.put("Go Set a Watchman",
Book.newBuilder().setName("Go Set a Watchman")
              .setAuthor("Harper Lee")
              .setPrice(700).build());
  }

  private Server server;

  private void start() throws IOException {
    int port = 50051;
    server = ServerBuilder.forPort(port)
        .addService(new BookStoreImpl()).build().start();

    logger.info("Server started, listening on " + port);

    Runtime.getRuntime().addShutdownHook(new Thread() {
      @Override
      public void run() {
        System.err.println("Shutting down gRPC server");
        try {
            server.shutdown().awaitTermination(30, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
          e.printStackTrace(System.err);
        }
      }
    });
  }

  public static void main(String[] args) throws IOException,
InterruptedException {
    final BookeStoreServerBothStreaming greetServer = new
BookeStoreServerBothStreaming();
    greetServer.start();
    greetServer.server.awaitTermination();
  }

  static class BookStoreImpl extends
BookStoreGrpc.BookStoreImplBase {

    @Override
    public StreamObserver<Book>
liveCartValue(StreamObserver<Cart> responseObserver) {

        return new StreamObserver<Book>() {

            ArrayList<Book> bookCart = new ArrayList<Book>();
            int cartValue = 0;

            @Override
            public void onNext(Book book) {
                logger.info("Searching for book with title 
starting with: " + book.getName());


                for (Entry<String, Book> bookEntry : 
bookMap.entrySet()) {



if(bookEntry.getValue().getName().startsWith(book.getName())) 
{
                        logger.info("Found book, adding to 
cart:....");
                        bookCart.add(bookEntry.getValue());
                        cartValue += 
bookEntry.getValue().getPrice();
                        }
                    }


                logger.info("Updating cart value...");


                responseObserver.onNext(Cart.newBuilder()
                        .setPrice(cartValue)
                        .setBooks(bookCart.size()).build());
            }


            @Override
            public void onError(Throwable t) {
                logger.info("Error while reading book stream: " + t);
            }


            @Override
            public void onCompleted() {
                logger.info("Order completed");
                responseObserver.onCompleted();
                }
            };


        }
    }
}
```

The above code starts a gRPC server at a specified port and serves the functions and services which we had written in our `proto` file. Let us walk through the above code:

1. Starting from the `main` method, we create a gRPC server at a specified port.

2. But before starting the server, we assign the server the service which we want to run, i.e., in our case, the `BookStore` service.

3. For this purpose, we need to pass in the service instance to the server, so we go ahead and create service instance i.e. in our case the `BookStoreImpl`

4. The service instance need to provide an implementation of the method/function which is present in the `proto` file, i.e., in our case, the `totalCartValue` method.

5. Now, given that this is the case of server and client streaming, the server will get the list of `Books` (defined in the `proto` file) as the client adds them. The server thus returns a custom stream observer. This stream observer implements what happens when a new Book is found and what happens when the stream is closed.

6. The `onNext()` method would be called by the gRPC framework when the client adds a Book. At this point, the server adds that to the cart and uses the response observer to return the Cart Value. In case of streaming, the server does not wait for all the valid books to be available.

7. When the client is done with the addition of Books, the stream observer's `onCompleted()` method is called. This method implements what the server wants to do when the client is done adding the `Books`, i.e., claim it is done with taking the client order.

8. Finally, we also have a shutdown hook to ensure clean shutting down of the server when we are done executing our code.

. Setting up gRPC client

Now that we have written the code for the server, let us setup a client which can call these functions.

Let us write our client code to call the above function and save it in **com.tp.bookstore.BookStoreClientBothStreaming.java**:

```
package com.tp.bookstore;


import io.grpc.Channel;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;
import io.grpc.stub.StreamObserver;


import java.util.Iterator;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;


import com.tp.bookstore.BookStoreGrpc.BookStoreFutureStub;
import com.tp.bookstore.BookStoreGrpc.BookStoreStub;
import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.bookstore.BookStoreOuterClass.Cart;
import com.tp.greeting.GreeterGrpc;
import com.tp.greeting.Greeting.ServerOutput;
import com.tp.greeting.Greeting.ClientInput;


public class BookStoreClientBothStreaming {
  private static final Logger logger =
Logger.getLogger(BookStoreClientBothStreaming.class.getName()
);
```

```
  private final BookStoreStub stub;

  private boolean serverIntermediateResponseCompleted = true;
  private boolean serverResponseCompleted = false;

  StreamObserver<Book> streamClientSender;

  public BookStoreClientBothStreaming(Channel channel) {
      stub = BookStoreGrpc.newStub(channel);
  }

  public StreamObserver<Cart> getServerResponseObserver(){
      StreamObserver<Cart> observer = new StreamObserver<Cart>(){
        @Override
        public void onNext(Cart cart) {
            logger.info("Order summary:" +
                  "\nTotal number of Books:" + cart.getBooks()+
                  "\nTotal Order Value: " + cart.getPrice());

            serverIntermediateResponseCompleted = true;
        }

        @Override
        public void onError(Throwable t) {
            logger.info("Error while reading response from
Server: " + t);
        }

        @Override
        public void onCompleted() {
            //logger.info("Server: Done reading orderreading
cart");
```

```
            serverResponseCompleted = true;
        }
      };

      return observer;
  }

  public void addBook(String book) {
    logger.info("Adding book with title starting with: " + book);
    Book request = Book.newBuilder().setName(book).build();

    if(streamClientSender == null) {
        streamClientSender = 
stub.liveCartValue(getServerResponseObserver());
    }

    try {
           streamClientSender.onNext(request);
    }
    catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
    }
  }

    public void completeOrder() {
        logger.info("Done, waiting for server to create order 
summary...");
        if(streamClientSender != null);
            streamClientSender.onCompleted();
  }
  public static void main(String[] args) throws Exception {
      String serverAddress = "localhost:50051";

      ManagedChannel channel =
ManagedChannelBuilder.forTarget(serverAddress)
          .usePlaintext()
          .build();

      try {
        BookStoreClientBothStreaming client = new
BookStoreClientBothStreaming(channel);
        String bookName = "";

        while(true) {
            if(client.serverIntermediateResponseCompleted ==
true) {
                System.out.println("Type book name to be
added to the cart....");
                bookName = System.console().readLine();
                if(bookName.equals("EXIT")) {
                    client.completeOrder();
                    break;
                }

                client.serverIntermediateResponseCompleted = false;
                client.addBook(bookName);
                Thread.sleep(500);
            }
        }

        while(client.serverResponseCompleted == false) {
            Thread.sleep(2000);
        }
      } finally {
        channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
      }
    }
}
```

The above code starts a gRPC client and connects to a server at a specified port and call the functions and services which we had written in our **`proto`** file. Let us walk through the above code:

1. Starting from the **`main`** method, we accept the name of the books to be added to the cart. Once all the books are to be added, the user is expected to print "EXIT".

2. We setup a Channel for gRPC communication with our server.

3. Next, we create a non-blocking stub using the channel. This is where we are choosing the service "**`BookStore`**" whose functions we plan to call.

4. Then, we simply create the expected input defined in the **`proto`** file, i.e., in our case, **`Book`**, and we add the title we want the server to add.

5. But given that this is the case of both server and client streaming, we first create a **`stream observer`** for the server. This server stream observer lists the behavior on what needs to be done when the server responds, i.e., **`onNext()`** and **`onCompleted()`**.

6. And using the stub, we also get the client stream observer. We use this stream observer for sending the data, i.e., the **`Book`** to be added to the cart.

7. And once our order is complete, we ensure that the client stream observer is closed. This tells the server to close the stream and perform the cleanup.

8. Finally, we close the channel to avoid any resource leak.

So, that is our client code.

. Client server call

To sum up, what we want to do is the following:

- Start the gRPC server.
- The Client adds a stream of books by notifying them to the server.
- The Server searches the book in its store and adds them to the cart.
- With each book addition, the server tells the client about the cart value.
- When the client is done ordering, both the server and the client close the stream.

Now that we have defined our **`proto`** file, written our server and the client code, let us now execute this code and see things in action.

For running the code, fire up two shells. Start the server on the first shell by executing the following command:

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookeStoreServerClientStreaming
```

We would see the following output:

```
Jul 03, 2021 10:37:21 PM
com.tp.bookstore.BookeStoreServerStreaming start

INFO: Server started, listening on 50051
```

This output implies that the server has started.

Now, let us start the client.

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookStoreClientBothStreaming
```

Let us add a book to our client.

```
Jul 24, 2021 7:21:45 PM
com.tp.bookstore.BookStoreClientBothStreaming main

Type book name to be added to the cart....

Great


Jul 24, 2021 7:21:48 PM
com.tp.bookstore.BookStoreClientBothStreaming addBook

INFO: Adding book with title starting with: Gr
```

```

Jul 24, 2021 7:21:48 PM
com.tp.bookstore.BookStoreClientBothStreaming$1 onNext

INFO: Order summary:

Total number of Books: 1

Total Order Value: 300
```

So, as we can see, we get the current cart value of the order. Let us now add one more book to our client.

```
Type book name to be added to the cart....

Passage



Jul 24, 2021 7:21:51 PM
com.tp.bookstore.BookStoreClientBothStreaming addBook

INFO: Adding book with title starting with: Pa



Jul 24, 2021 7:21:51 PM
com.tp.bookstore.BookStoreClientBothStreaming$1 onNext

INFO: Order summary:

Total number of Books: 2

Total Order Value: 800
```

Once we have added the books and we input "EXIT", the client shuts down.

```
Type book name to be added to the cart....

EXIT

Jul 24, 2021 7:21:59 PM
com.tp.bookstore.BookStoreClientBothStreaming completeOrder

INFO: Done, waiting for server to create order summary...
```

So, as we can see the client was able to add books. And as the books are being added, the server responds with the current cart value.

gRPC – Client Calls

gRPC client supports two types of client calls, i.e., how the client calls the server. Following are the two ways:

- Blocking client call
- Async client call

In this chapter, we will look at both of them one by one.

Blocking Client Calls

gRPC supports blocking client call. What this means is that once the client makes the call to the service, the client would not proceed with rest of the code execution until it gets the response back from the server.

Note that a blocking client call is possible for unary calls and server streaming calls.

Here is an example of a unary blocking client call.

```
package com.tp.bookstore;


import io.grpc.Channel;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.greeting.GreeterGrpc;
import com.tp.greeting.Greeting.ServerOutput;
import com.tp.greeting.Greeting.ClientInput;


public class BookStoreClientUnaryBlocking {
  private static final Logger logger =
Logger.getLogger(BookStoreClientUnaryBlocking.class.getName()
);


  private final BookStoreGrpc.BookStoreBlockingStub
blockingStub;


  public BookStoreClientUnaryBlocking(Channel channel) {
      blockingStub = BookStoreGrpc.newBlockingStub(channel);
  }


  public void getBook(String bookName) {
    logger.info("Querying for book with title: " + bookName);
    BookSearch request =
BookSearch.newBuilder().setName(bookName).build();


    Book response;
    try {
      response = blockingStub.first(request);
    } catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
      return;
    }
    logger.info("Got following book from server: " + response);
  }



  public static void main(String[] args) throws Exception {
    String bookName = args[0];
    String serverAddress = "localhost:50051";


    ManagedChannel channel = 
ManagedChannelBuilder.forTarget(serverAddress)
        .usePlaintext()
        .build();


    try {
      BookStoreClientUnaryBlocking client = new 
BookStoreClientUnaryBlocking(channel);
      client.getBook(bookName);
    } finally {
      channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
    }
  }
}
```

In the above example, we have,

```
public BookStoreClientUnaryBlocking(Channel channel) {
      blockingStub = BookStoreGrpc.newBlockingStub(channel);
}
```

which means we will be using a blocking RPC call.

And then, we have,

```
BookSearch request = 
BookSearch.newBuilder().setName(bookName).build();
Book response;
response = blockingStub.first(request);
```

This is where we use the **`blockingStub`** to call the RPC method **`first()`** to get the book details.

Similarly, for server streaming, we can use the blocking stub:

```
logger.info("Querying for book with author: " + author);
BookSearch request =
BookSearch.newBuilder().setAuthor(author).build();


Iterator<Book> response;
try {
    response = blockingStub.searchByAuthor(request);
    while(response.hasNext()) {
        logger.info("Found book: " + response.next());
    }
```

Where we call the RPC method **`searchByAuthor`** method and iterate over the response till the server stream has not ended.

Non-Blocking Client Calls

gRPC supports non-blocking client calls. What this means is that when the client makes a call to the service, it does not need to wait for the server response. To handle the server response, the client can simply pass in the observer which dictates what to do when the response is received.

Note that a non-blocking client call is possible for unary calls as well as streaming calls. However, we would specifically look at the case of server streaming call to compare it against a blocking call.

Here is an example of a server streaming non-blocking client call.

```
package com.tp.bookstore;


import io.grpc.Channel;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;
import io.grpc.stub.StreamObserver;
import java.util.Iterator;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.greeting.GreeterGrpc;
import com.tp.greeting.Greeting.ServerOutput;
import com.tp.greeting.Greeting.ClientInput;

public class BookStoreClientServerStreamingNonBlocking {
  private static final Logger logger =
Logger.getLogger(BookStoreClientServerStreamingNonBlocking.cl
ass.getName());

  private final BookStoreGrpc.BookStoreStub nonBlockingStub;

  public BookStoreClientServerStreamingNonBlocking(Channel
channel) {
      nonBlockingStub = BookStoreGrpc.newStub(channel);
  }

  public StreamObserver<Book> getServerResponseObserver(){

      StreamObserver<Book> observer = new
StreamObserver<Book>(){

        @Override
        public void onNext(Book book) {
            logger.info("Server returned following book: " +
book);
        }
        @Override
        public void onError(Throwable t) {
            logger.info("Error while reading response from
Server: " + t);
        }

        @Override
        public void onCompleted() {
        }
      };

      return observer;
  }

  public void getBook(String author) {
    logger.info("Querying for book with author: " + author);
    BookSearch request =
BookSearch.newBuilder().setAuthor(author).build();


    try {
        nonBlockingStub.searchByAuthor(request,
getServerResponseObserver());
    } catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}",
e.getStatus());
      return;
    }
  }

  public static void main(String[] args) throws Exception {
    String authorName = args[0];
    String serverAddress = "localhost:50051";
    ManagedChannel channel = 
ManagedChannelBuilder.forTarget(serverAddress)
        .usePlaintext()
        .build();

    try {
      BookStoreClientServerStreamingNonBlocking client = new 
BookStoreClientServerStreamingNonBlocking(channel);
      client.getBook(authorName);
    } finally {
      channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
    }
  }
}
```

As we see in the above example,

```
public BookStoreClientUnaryNonBlocking(Channel channel) {
    nonBlockingStub = BookStoreGrpc.newStub(channel);
}
```

It defines that the stub is a non-blocking one. Similarly, the following code is used to handle the response that we get from the server. Once the server sends the response, we log the output.

```
public StreamObserver<Book> getServerResponseObserver(){
    StreamObserver<Book> observer = new 
StreamObserver<Book>(){
    ....
    ....
    return observer;
}
```

The following gRPC call is a non-blocking call.

```
logger.info("Querying for book with author: " + author);

BookSearch request =
BookSearch.newBuilder().setAuthor(author).build();


try {

    nonBlockingStub.searchByAuthor(request,
getServerResponseObserver());

}
```

This is how we ensure that our client does not need to wait till we have the server complete the execution of **searchByAuthor**. That would be handled directly by the stream observer object as and when the server returns the **Book** objects.

gRPC – Timeouts and Cancellation

gRPC supports assigning timeouts to the requests. It is a way to perform cancellation of requests. It helps to avoid using the resources for both the client and the server for a request whose result would not be useful to the client.

Request Timeout

gRPC supports specifying a timeout for both client as well as the server.

- The Client can specify during runtime the amount of time it wants to wait before cancelling the request.
- The Server can also check on its end whether the requests need to be catered to or has the Client already given up on the request.

Let us take an example where the client expects a response in 2 seconds, but the server takes a longer time. So, here is our **server code**:

```
package com.tp.bookstore;


import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.stub.StreamObserver;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;
import java.util.stream.Collectors;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
```

```
public class BookeStoreServerUnaryTimeout {
  private static final Logger logger =
Logger.getLogger(BookeStoreServerUnaryTimeout.class.getName()
);


  static Map<String, Book> bookMap = new HashMap<>();
  static {
      bookMap.put("Great Gatsby",
Book.newBuilder().setName("Great Gatsby")
              .setAuthor("Scott Fitzgerald")
              .setPrice(300).build());
      bookMap.put("To Kill MockingBird",
Book.newBuilder().setName("To Kill MockingBird")
              .setAuthor("Harper Lee")
              .setPrice(400).build());
      bookMap.put("Passage to India",
Book.newBuilder().setName("Passage to India")
              .setAuthor("E.M.Forster")
              .setPrice(500).build());
      bookMap.put("The Side of Paradise",
Book.newBuilder().setName("The Side of Paradise")
              .setAuthor("Scott Fitzgerald")
              .setPrice(600).build());
      bookMap.put("Go Set a Watchman",
Book.newBuilder().setName("Go Set a Watchman")
              .setAuthor("Harper Lee")
              .setPrice(700).build());
  }


  private Server server;


  private void start() throws IOException {
    int port = 50051;
    server = ServerBuilder.forPort(port)
```

```
        .addService(new BookStoreImpl()).build().start();

    logger.info("Server started, listening on " + port);

    Runtime.getRuntime().addShutdownHook(new Thread() {
      @Override
      public void run() {
        System.err.println("Shutting down gRPC server");
        try {
            server.shutdown().awaitTermination(30,
TimeUnit.SECONDS);
        } catch (InterruptedException e) {
          e.printStackTrace(System.err);
        }
      }
    });
  }

  public static void main(String[] args) throws IOException,
InterruptedException {
    final BookeStoreServerUnaryTimeout greetServer = new
BookeStoreServerUnaryTimeout();
    greetServer.start();
    greetServer.server.awaitTermination();
  }

  static class BookStoreImpl extends
BookStoreGrpc.BookStoreImplBase {

    @Override
    public void first(BookSearch searchQuery,
StreamObserver<Book> responseObserver) {
```

```
          logger.info("Searching for book with title: " +
searchQuery.getName());
          logger.info("This may take more time...");
          try {
              Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
          List<String> matchingBookTitles =
bookMap.keySet().stream()
                  .filter(title ->
title.startsWith(searchQuery.getName().trim()))
                  .collect(Collectors.toList());
          Book foundBook = null;
          if(matchingBookTitles.size() > 0) {
              foundBook = bookMap.get(matchingBookTitles.get(0));
          }

          responseObserver.onNext(foundBook);
          responseObserver.onCompleted();
      }
  }
}
```

In the above code, the server searches for the **book** the title of which the client has provided. We added a **`dummy sleep`** so that we can see the request getting timed out.

And here is our **client code**:

```
package com.tp.bookstore;


import io.grpc.Channel;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.greeting.GreeterGrpc;
import com.tp.greeting.Greeting.ServerOutput;
import com.tp.greeting.Greeting.ClientInput;

public class BookStoreClientUnaryBlockingTimeout {
  private static final Logger logger =
Logger.getLogger(BookStoreClientUnaryBlockingTimeout.class.ge
tName());

  private final BookStoreGrpc.BookStoreBlockingStub
blockingStub;

  public BookStoreClientUnaryBlockingTimeout(Channel channel)
{
      blockingStub = BookStoreGrpc.newBlockingStub(channel);
  }

  public void getBook(String bookName) {
    logger.info("Querying for book with title: " + bookName);
    BookSearch request =
BookSearch.newBuilder().setName(bookName).build();

    Book response;
    try {
      response = blockingStub.withDeadlineAfter(2,
TimeUnit.SECONDS).first(request);
```

```
    } catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}",
e.getStatus());
      return;
    }
    logger.info("Got following book from server: " +
response);
  }



  public static void main(String[] args) throws Exception {
    String bookName = args[0];
    String serverAddress = "localhost:50051";

    ManagedChannel channel =
ManagedChannelBuilder.forTarget(serverAddress)
        .usePlaintext()
        .build();

    try {
      BookStoreClientUnaryBlockingTimeout client = new
BookStoreClientUnaryBlockingTimeout(channel);
      client.getBook(bookName);
    } finally {
      channel.shutdownNow().awaitTermination(5,
TimeUnit.SECONDS);
    }
  }
}
```

The above code calls the server with a **`title`** to search for. But more importantly, it provides a **`timeout of 2 seconds`** to the gRPC call.

Let us now see this in action. For running the code, fire up two shells. Start the server on the first shell by executing the following command:

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookeStoreServerUnaryTimeout
```

We would see the following output:

```
Jul 31, 2021 12:29:31 PM
com.tp.bookstore.BookeStoreServerUnaryTimeout start

INFO: Server started, listening on 50051
```

The above output shows that the server has started.

```
Jul 31, 2021 12:29:35 PM
com.tp.bookstore.BookeStoreServerUnaryTimeout$BookStoreImpl
first

INFO: Searching for book with title: Great

Jul 31, 2021 12:29:35 PM
com.tp.bookstore.BookeStoreServerUnaryTimeout$BookStoreImpl
first

INFO: This may take more time...
```

Now, let us start the client.

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookStoreClientUnaryBlockingTimeout Great
```

We would get the following output:

```
Jul 31, 2021 12:29:34 PM
com.tp.bookstore.BookStoreClientUnaryBlockingTimeout getBook

INFO: Querying for book with title: Great


Jul 31, 2021 12:29:36 PM
com.tp.bookstore.BookStoreClientUnaryBlockingTimeout getBook

WARNING: RPC failed: Status{code=DEADLINE_EXCEEDED,
description=deadline exceeded after 1.970455800s.
[buffered_nanos=816522700,
remote_addr=localhost/127.0.0.1:50051], cause=null}

P
```

So, as we can see, the client did not get the response in 2 seconds, hence it cancelled the request and termed it as a timeout, i.e., **`DEADLINE_EXCEEDED`**.

Request Cancellation

gRPC supports canceling of requests both from the client as well as the server side. The Client can specify during runtime the amount of time it wants to wait before cancelling the request. The Server can also check on its end whether the requests need to be catered to or has the client already given up on the request.

Let us look at an example of client streaming, where the client invokes cancellation. So, here is our server code:

```
package com.tp.bookstore;


import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.stub.StreamObserver;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;
import java.util.stream.Collectors;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;
import com.tp.bookstore.BookStoreOuterClass.Cart;


public class BookeStoreServerClientStreaming {
  private static final Logger logger =
Logger.getLogger(BookeStoreServerClientStreaming.class.getNam
e());
```

```
  static Map<String, Book> bookMap = new HashMap<>();
  static {
      bookMap.put("Great Gatsby",
Book.newBuilder().setName("Great Gatsby")
              .setAuthor("Scott Fitzgerald")
              .setPrice(300).build());
      bookMap.put("To Kill MockingBird",
Book.newBuilder().setName("To Kill MockingBird")
              .setAuthor("Harper Lee")
              .setPrice(400).build());
      bookMap.put("Passage to India",
Book.newBuilder().setName("Passage to India")
              .setAuthor("E.M.Forster")
              .setPrice(500).build());
      bookMap.put("The Side of Paradise",
Book.newBuilder().setName("The Side of Paradise")
              .setAuthor("Scott Fitzgerald")
              .setPrice(600).build());
      bookMap.put("Go Set a Watchman",
Book.newBuilder().setName("Go Set a Watchman")
              .setAuthor("Harper Lee")
              .setPrice(700).build());
  }

  private Server server;

  private void start() throws IOException {
    int port = 50051;
    server = ServerBuilder.forPort(port)
        .addService(new BookStoreImpl()).build().start();

    logger.info("Server started, listening on " + port);
    Runtime.getRuntime().addShutdownHook(new Thread() {
      @Override
      public void run() {
        System.err.println("Shutting down gRPC server");
        try {
            server.shutdown().awaitTermination(30,
TimeUnit.SECONDS);
        } catch (InterruptedException e) {
          e.printStackTrace(System.err);
        }
      }
    });
  }

  public static void main(String[] args) throws IOException,
InterruptedException {
    final BookeStoreServerClientStreaming greetServer = new
BookeStoreServerClientStreaming();
    greetServer.start();
    greetServer.server.awaitTermination();
  }

  static class BookStoreImpl extends
BookStoreGrpc.BookStoreImplBase {

    @Override
    public StreamObserver<Book>
totalCartValue(StreamObserver<Cart> responseObserver) {

        return new StreamObserver<Book>() {

            ArrayList<Book> bookCart = new ArrayList<Book>();
            @Override
            public void onNext(Book book) {
                logger.info("Searching for book with title
starting with: " + book.getName());


                for (Entry<String, Book> bookEntry :
bookMap.entrySet()) {



if(bookEntry.getValue().getName().startsWith(book.getName()))
{
                        logger.info("Found book, adding to
cart:....");
                        bookCart.add(bookEntry.getValue());
                        }
                    }
            }


            @Override
            public void onError(Throwable t) {
                logger.info("Error while reading book stream:
" + t);
            }


            @Override
            public void onCompleted() {
                int cartValue = 0;


                for (Book book : bookCart) {
                    cartValue += book.getPrice();
                }


                responseObserver.onNext(Cart.newBuilder()
                        .setPrice(cartValue)
                        .setBooks(bookCart.size()).build());

                responseObserver.onCompleted();

            }
        };

     }
  }
}
```

This server code is a simple example of client side streaming. The server simply tracks the books which the client wants and at the end, it provides the total Cart Value of the order.

But there is nothing special here with respect to cancelation of request as that is something the client would invoke. So, let us look at the client code.

```
package com.tp.bookstore;

import io.grpc.Channel;
import io.grpc.Context;
import io.grpc.Context.CancellableContext;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.StatusRuntimeException;
import io.grpc.stub.StreamObserver;

import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;

import com.tp.bookstore.BookStoreGrpc.BookStoreStub;
import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.Cart;

public class BookStoreClientStreamingClientCancelation {
  private static final Logger logger =
Logger.getLogger(BookStoreClientStreamingClientCancelation.cl
ass.getName());
  private final BookStoreStub stub;
  StreamObserver<Book> streamClientSender;

private CancellableContext withCancellation;

  public BookStoreClientStreamingClientCancelation(Channel channel) {
      stub = BookStoreGrpc.newStub(channel);
  }

  public StreamObserver<Cart> getServerResponseObserver(){
      StreamObserver<Cart> observer = new StreamObserver<Cart>(){

        @Override
        public void onNext(Cart cart) {
            logger.info("Order summary:" +
                  "\nTotal number of Books: " + cart.getBooks() +
                  "\nTotal Order Value: " + cart.getPrice());
        }

        @Override
        public void onError(Throwable t) {
            logger.info("Error while reading response from
Server: " + t);
        }

        @Override
        public void onCompleted() {
        }
      };
      return observer;
  }

  public void addBook(String book) {
    logger.info("Adding book with title starting with: " + book);
    Book request = Book.newBuilder().setName(book).build();

    if(streamClientSender == null) {
        withCancellation = Context.current().withCancellation();
        streamClientSender =
stub.totalCartValue(getServerResponseObserver());
    }

    try {
       streamClientSender.onNext(request);


    }
    catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
    }
  }

    public void completeOrder() {
        logger.info("Done, waiting for server to create order
summary...");
        if(streamClientSender != null);
            streamClientSender.onCompleted();
  }
```

```
    public void cancelOrder() {
        withCancellation.cancel(null);
  }



  public static void main(String[] args) throws Exception {
      String serverAddress = "localhost:50051";
      ManagedChannel channel = 
ManagedChannelBuilder.forTarget(serverAddress)
          .usePlaintext()
          .build();

      try {
        BookStoreClientStreamingClientCancelation client = 
new BookStoreClientStreamingClientCancelation(channel);
        String bookName = "";

        while(true) {
            System.out.println("Type book name to be added to 
the cart....");
            bookName = System.console().readLine();
            if(bookName.equals("EXIT")) {
                client.completeOrder();
                break;
            }

            if(bookName.equals("CANCEL")) {
                client.cancelOrder();
                break;
            }

            client.addBook(bookName);
        }
      } finally {
        channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
      }
    }
}
```

So, if we see the above code, the following line defines a context which has cancellation enabled.

```
withCancellation = Context.current().withCancellation();
```

And here is the method which would be called when the user types CANCEL. This would cancel the order and also let the server know about it.

```
public void cancelOrder() {
    withCancellation.cancel(null);
}
```

Let us now see this in action. For running the code, fire up two shells. Start the server on the first shell by executing the following command:

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookeStoreServerClientStreaming
```

We would get to see the following output:

```
Jul 31, 2021 3:29:58 PM
com.tp.bookstore.BookeStoreServerClientStreaming start
INFO: Server started, listening on 50051
```

The above output implies that the server has started.

Now, let us start the client.

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookStoreClientStreamingClientCancelation
```

We would get the following output:

```
Type book name to be added to the cart....

Great

Jul 31, 2021 3:30:55 PM
com.tp.bookstore.BookStoreClientStreamingClientCancelation
addBook

INFO: Adding book with title starting with: Great

Type book name to be added to the cart....

CANCEL

Jul 31, 2021 3:30:58 PM
com.tp.bookstore.BookStoreClientStreamingClientCancelation$1
onError

INFO: Error while reading response from Server:
io.grpc.StatusRuntimeException: UNAVAILABLE: Channel
shutdownNow invoked
```

And we would get the following data in the server logs:

```
INFO: Searching for book with title starting with: Great

Jul 31, 2021 3:30:56 PM
com.tp.bookstore.BookeStoreServerClientStreaming$BookStoreImp
l$1 onNext

INFO: Found book, adding to cart:....

Jul 31, 2021 3:30:58 PM
com.tp.bookstore.BookeStoreServerClientStreaming$BookStoreImp
l$1 onError

INFO: Error while reading book stream:
io.grpc.StatusRuntimeException: CANCELLED: client cancelled
```

So, as we can see, the client initiated a cancellation of the request it made to the server. The server was also notified about the cancellation.

gRPC – Send/Receive Metadata

gRPC supports sending metadata. The metadata is basically a set of data we want to send that is not part of the business logic, while making gRPC calls. Let us look at the following two cases:

- The Client sends Metadata and the Server reads it.
- The Server sends Metadata and Client reads it.

We will go through both these cases one by one.

Client Sends Metadata

As mentioned, gRPC supports the client sending the metadata which the server can read. gRPC supports extending the client and server interceptor which can be used to write and read metadata, respectively.

Let us take an example to understand it better. Here is our client code which sends the hostname as metadata:

```
package com.tp.bookstore;


import io.grpc.CallOptions;
import io.grpc.Channel;
import io.grpc.ClientCall;
import io.grpc.ClientInterceptor;
import io.grpc.ForwardingClientCall;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import io.grpc.Metadata;
import io.grpc.MethodDescriptor;
import io.grpc.StatusRuntimeException;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;
import static io.grpc.Metadata.ASCII_STRING_MARSHALLER;

import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;

public class BookStoreClientUnaryBlockingMetadata {
  private static final Logger logger =
Logger.getLogger(BookStoreClientUnaryBlockingMetadata.class.g
etName());

  private final BookStoreGrpc.BookStoreBlockingStub
blockingStub;

  public BookStoreClientUnaryBlockingMetadata(Channel channel) {
      blockingStub = BookStoreGrpc.newBlockingStub(channel);
  }

  static class BookClientInterceptor implements
ClientInterceptor{
        @Override
        public <ReqT, RespT> ClientCall<ReqT, RespT>
interceptCall(MethodDescriptor<ReqT, RespT> method,
                CallOptions callOptions, Channel next) {

            return new
ForwardingClientCall.SimpleForwardingClientCall<ReqT,
RespT>(next.newCall(method, callOptions)) {
                @Override
                public void start(Listener<RespT>
responseListener, Metadata headers) {
                    logger.info("Added metadata");
                    headers.put(Metadata.Key.of("HOSTNAME",
ASCII_STRING_MARSHALLER), "MY_HOST");
```

```
                    super.start(responseListener, headers);
                }
            };
        }
    }

  public void getBook(String bookName) {
    logger.info("Querying for book with title: " + bookName);
    BookSearch request =
BookSearch.newBuilder().setName(bookName).build();
    Book response;
    CallOptions.Key<String> metaDataKey =
CallOptions.Key.create("my_key");
    try {
      response = blockingStub.withOption(metaDataKey,
"bar").first(request);
    } catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
      return;
    }
    logger.info("Got following book from server: " + response);
  }

  public static void main(String[] args) throws Exception {
    String bookName = args[0];
    String serverAddress = "localhost:50051";

    ManagedChannel channel =
ManagedChannelBuilder.forTarget(serverAddress)
        .usePlaintext()
        .intercept(new BookClientInterceptor())
        .build();
    try {
      BookStoreClientUnaryBlockingMetadata client = new
BookStoreClientUnaryBlockingMetadata(channel);
      client.getBook(bookName);
    } finally {
      channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
    }
  }
}
```

The interesting bit here is the **interceptor**.

```
static class BookClientInterceptor implements ClientInterceptor{


        @Override
        public <ReqT, RespT> ClientCall<ReqT, RespT>
interceptCall(MethodDescriptor<ReqT, RespT> method,
                CallOptions callOptions, Channel next) {


            return new
ForwardingClientCall.SimpleForwardingClientCall<ReqT,
RespT>(next.newCall(method, callOptions)) {
                @Override
                public void start(Listener<RespT>
responseListener, Metadata headers) {
                    logger.info("Added metadata");
                    headers.put(Metadata.Key.of("HOSTNAME",
ASCII_STRING_MARSHALLER), "MY_HOST");
                    super.start(responseListener, headers);
                }
            };
        }
    }
```

We intercept any call which is being made by the client and then add hostname metadata to it before it is called further.

Server Reads Metadata

Now, let us look at the server code which reads this metadata:

```
package com.tp.bookstore;


import io.grpc.CallOptions;
import io.grpc.Context;
import io.grpc.Metadata;
import io.grpc.Server;
import io.grpc.ServerBuilder;
import io.grpc.ServerCall;
import io.grpc.ServerCall.Listener;
import io.grpc.ServerCallHandler;
import io.grpc.ServerInterceptor;
import io.grpc.stub.StreamObserver;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;
import java.util.stream.Collectors;


import com.tp.bookstore.BookStoreOuterClass.Book;
import com.tp.bookstore.BookStoreOuterClass.BookSearch;


public class BookeStoreServerMetadata {
  private static final Logger logger =
Logger.getLogger(BookeStoreServerMetadata.class.getName());


  static Map<String, Book> bookMap = new HashMap<>();
  static {
      bookMap.put("Great Gatsby",
Book.newBuilder().setName("Great Gatsby")
              .setAuthor("Scott Fitzgerald")
              .setPrice(300).build());
  }

  private Server server;

  class BookServerInterceptor implements ServerInterceptor{

    @Override
    public <ReqT, RespT> Listener<ReqT>
interceptCall(ServerCall<ReqT, RespT> call,
            Metadata headers,
            ServerCallHandler<ReqT, RespT> next) {

        logger.info("Recieved following metadata: " + headers);
        return next.startCall(call, headers);
    }
  }

  private void start() throws IOException {
    int port = 50051;
    server = ServerBuilder.forPort(port)
        .addService(new BookStoreImpl())
        .intercept(new BookServerInterceptor())
        .build().start();

    logger.info("Server started, listening on " + port);

    Runtime.getRuntime().addShutdownHook(new Thread() {
      @Override
      public void run() {
        System.err.println("Shutting down gRPC server");
        try {
            server.shutdown().awaitTermination(30, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
          e.printStackTrace(System.err);
        }
      }
    });
  }

  public static void main(String[] args) throws IOException,
InterruptedException {
    final BookeStoreServerMetadata greetServer = new
BookeStoreServerMetadata();
    greetServer.start();
    greetServer.server.awaitTermination();
  }

  static class BookStoreImpl extends
BookStoreGrpc.BookStoreImplBase {

      @Override
    public void first(BookSearch searchQuery,
StreamObserver<Book> responseObserver) {
          logger.info("Searching for book with title: " +
searchQuery.getName());
          List<String> matchingBookTitles =
bookMap.keySet().stream()
                  .filter(title ->
title.startsWith(searchQuery.getName().trim()))
                  .collect(Collectors.toList());
          Book foundBook = null;
          if(matchingBookTitles.size() > 0) {
              foundBook = bookMap.get(matchingBookTitles.get(0));
          }
          responseObserver.onNext(foundBook);
          responseObserver.onCompleted();
      }
  }
}
```

Again, the interesting bit here is the **interceptor**.

```
class BookServerInterceptor implements ServerInterceptor{
    @Override
    public <ReqT, RespT> Listener<ReqT>
interceptCall(ServerCall<ReqT, RespT> call,
            Metadata headers,
            ServerCallHandler<ReqT, RespT> next) {

        logger.info("Recieved following metadata: " + headers);
        return next.startCall(call, headers);
    }
}
```

We intercept any call which is incoming to the server and then log the metadata before the call can be handled by the actual method.

Client-Server Call

Let us now see this in action. For running the code, fire up two shells. Start the server on the first shell by executing the following command:

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookeStoreServerMetadata
```

We would see the following output:

```
Jul 31, 2021 5:29:14 PM
com.tp.bookstore.BookeStoreServerMetadata start
INFO: Server started, listening on 50051
```

The above output implies that the server has started.

Now, let us start the client.

```
java -cp .\target\grpc-point-1.0.jar
com.tp.bookstore.BookStoreClientUnaryBlockingMetadata Great
```

We would see the following output:

```
Jul 31, 2021 5:29:39 PM
com.tp.bookstore.BookStoreClientUnaryBlockingMetadata getBook

INFO: Querying for book with title: Great

Jul 31, 2021 5:29:39 PM
com.tp.bookstore.BookStoreClientUnaryBlockingMetadata$BookCli
entInterceptor$1 start

INFO: Added metadata

Jul 31, 2021 5:29:40 PM
com.tp.bookstore.BookStoreClientUnaryBlockingMetadata getBook

INFO: Got following book from server: name: "Great Gatsby"

author: "Scott Fitzgerald"

price: 300
```

And we will have the following data in the server logs:

```
Jul 31, 2021 5:29:40 PM
com.tp.bookstore.BookeStoreServerMetadata$BookServerIntercept
or interceptCall

INFO: Recieved following metadata: Metadata(content-
type=application/grpc,user-agent=grpc-java-
netty/1.38.0,hostname=MY_HOST,grpc-accept-encoding=gzip)

Jul 31, 2021 5:29:40 PM
com.tp.bookstore.BookeStoreServerMetadata$BookStoreImpl first

INFO: Searching for book with title: Great
```

As we can see, the server is able to read the metadata: **`hostname=MY_HOST`** which was added by the client.