Author: Francesco Polichetti
Version: 2024-02-20

Questions

- **What is gRPC and why does it work accross languages and platforms?**

  gRPC is a high-performance, open-source universal RPC framework that uses HTTP/2 for transport, enabling seamless communication across languages and platforms due to its standardized protocol.
- **Describe the RPC life cycle starting with the RPC client?**

  It starts with the RPC client sending a request to the server. The server processes the request, executes the procedure, and sends a response back to the client.
- **Describe the workflow of Protocol Buffers?**

  Define data structures and services in `.proto` files, compile them into source code for the chosen language, then serialize and deserialize data using the generated code.
- **What are the benefits of using protocol buffers?**

  Efficient serialization, smaller message size, faster communication, backward compatibility, language-neutral, platform-independent, and easier API evolution.
- **When is the use of protocol not recommended?**

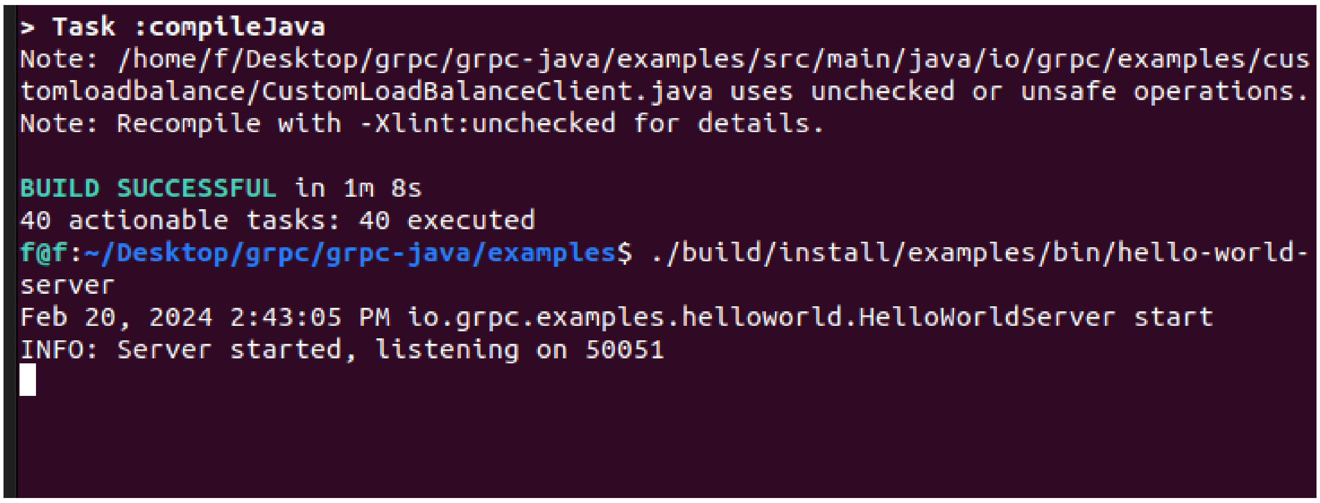
  For applications requiring dynamic data structures with high levels of flexibility or when human-readable format is a strict requirement.
- **List 3 different data types that can be used with protocol buffers?**

  `int32` for 32-bit integers, `string` for text, and `bool` for boolean values.

Basic Tasks

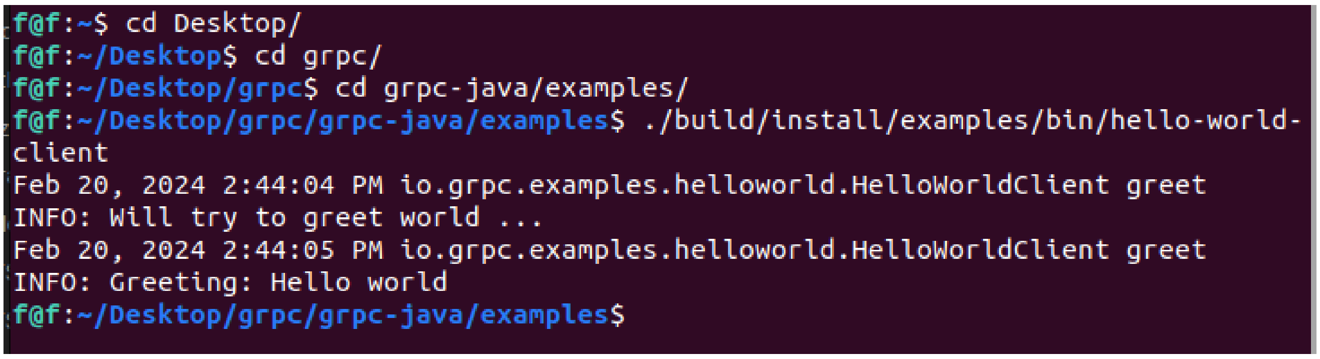
Step 1: Simple Installation

In order to install GRPC in a Java environment I headed to the official **grpc website**. On the webiste I found the installation instructions for Java. These are to clone the Java-Repo for GRPC from git and then to compile the example client and server using `./gradlew installDist` and the to run the server using `./build/install/examples/bin/hello-world-server` the result should look like the following:



```
> Task :compileJava
Note: /home/f/Desktop/grpc/grpc-java/examples/src/main/java/io/grpc/examples/cus
tomloadbalance/CustomLoadBalanceClient.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

BUILD SUCCESSFUL in 1m 8s
40 actionable tasks: 40 executed
f@f:~/Desktop/grpc/grpc-java/examples$ ./build/install/examples/bin/hello-world-
server
Feb 20, 2024 2:43:05 PM io.grpc.examples.helloworld.HelloWorldServer start
INFO: Server started, listening on 50051
```

Thats it for the server Part, running the client looks like the following in a new terminal window:



```
f@f:~$ cd Desktop/
f@f:~/Desktop$ cd grpc/
f@f:~/Desktop/grpc$ cd grpc-java/examples/
f@f:~/Desktop/grpc/grpc-java/examples$ ./build/install/examples/bin/hello-world-
client
Feb 20, 2024 2:44:04 PM io.grpc.examples.helloworld.HelloWorldClient greet
INFO: Will try to greet world ...
Feb 20, 2024 2:44:05 PM io.grpc.examples.helloworld.HelloWorldClient greet
INFO: Greeting: Hello world
f@f:~/Desktop/grpc/grpc-java/examples$
```

Step 2: Setup

But in order to fully extend our workspace using Gradle and Java in Combinatoin with gRPC we firstly have to in terms of setting up the gRPC Application do the following:

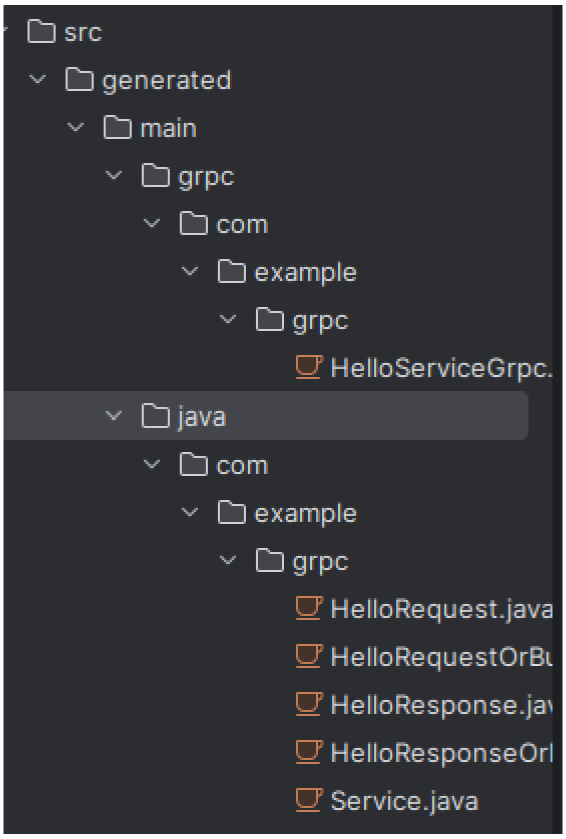
1. creating new Project using Java and Gradle
2. Create a .proto Configuration File under main > proto > service.proto that is used for shaping the service which is going to be needed, in my examplary case:

```
syntax = "proto3";

service HelloWorldService {
  rpc hello(HelloRequest) returns (HelloResponse) {}
}

message HelloRequest {
  string firstname = 1;
  string lastname = 2;

}
```

```
message HelloResponse {
  string text = 1;
}
```

Furthermore we need to update the project's Dependencies following this tutorial: https://blog.shiftasia.com/introduction-grpc-and-implement-with-spring-boot/

After, that I will create the remaining Project strucutre, which is possible by simply building the projects after successfully implementing the dependencies, the outcome looks like the following:



```
src
  generated
    main
      grpc
        com
          example
            grpc
              HelloServiceGrpc.
      java
        com
          example
            grpc
              HelloRequest.java
              HelloRequestOrBu
              HelloResponse.jav
              HelloResponseOrl
              Service.java
```

And then finally the last setup-step is to simply create a server application by doing the following:

1. create a new module named "server-side" inside the project
2. update the gradle file to the following:

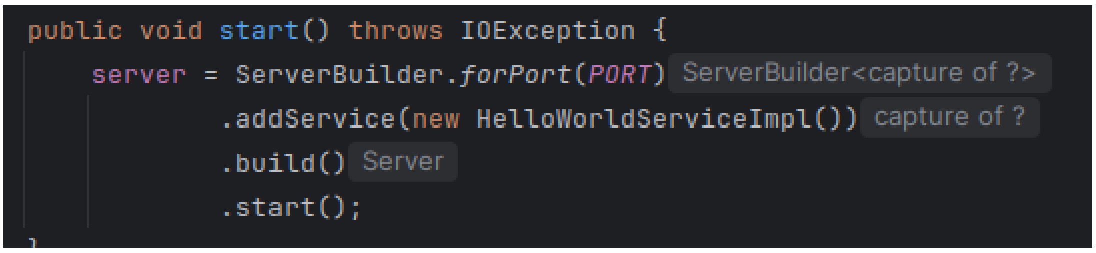
```
plugins {
    id 'java'
}

group = 'org.example'
version = '1.0-SNAPSHOT'
```

```
repositories {
    mavenCentral()
}

dependencies {
    implementation project(path: ':') // Reference to parent project
    implementation 'net.devh:grpc-server-spring-boot-
starter:2.14.0.RELEASE' // gRPC server dependence
}

test {
    useJUnitPlatform()
}
```

and that's it

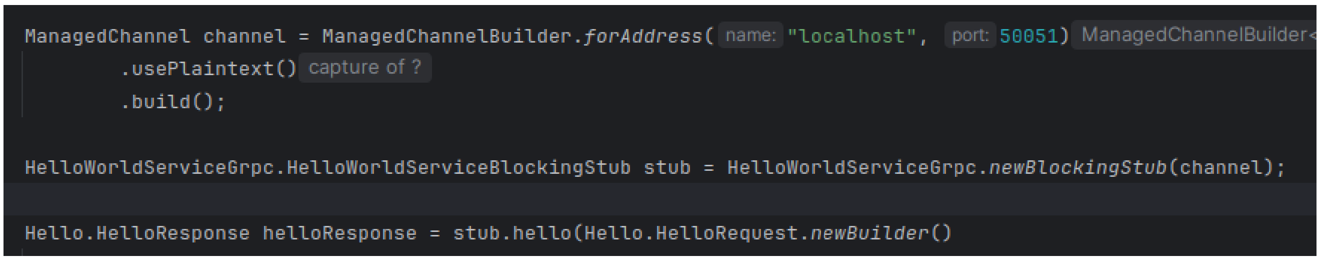
Implementation

To implement this I used the 3 Example Files which are used for the following reasons: 1) HelloWorld Client as an Client application, 2) HelloWorldServer as a Server Application and 3) HelloWorldServiceImpl a Service Implementation Application. Useful Codes:



```
public void start() throws IOException {
    server = ServerBuilder.forPort(PORT)
            .addService(new HelloWorldServiceImpl())
            .build()
            .start();
```

Start Method which contains a ServerBuilder which adds the Service that should be run, the port and a build call.



```
ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 50051)
        .usePlaintext()
        .build();

HelloWorldServiceGrpc.HelloWorldServiceBlockingStub stub = HelloWorldServiceGrpc.newBlockingStub(channel);

Hello.HelloResponse helloResponse = stub.hello(Hello.HelloRequest.newBuilder()
```

Here is a snippet from the Client and how Messages and Responses are being built.