LAB 2 - gRPC

Another technology for practical application of middleware aka RPC
gRPC is equivalent to javaRMI, but it can allow us to better see the difference when developing with an Interface Definition Language or using the precompilation to generate the code stub and skeleton.
gRPC is a universal RPC framework developed initially open source by google. We will see a server in java and a client in both java and python, to underline the fact that the code is generated dynamically from the Interface Definition Language IDL which is not language dependant(which is extremely powerful because it allows for multi-language systems and distributed systems)

In gRPC the IDL (interface file) is called proto (txt .proto file extension). In our example we are building an helloService functionality

Server

- Proto
    - Proto files start by declaring which syntax aka version they use (proto3 is the latest). option is the keyword to tell the service extra directives (option java_multiple_files = true;)
    - Then it specifies the package to which it responds to (it needs to be the same on the server and the client proto)
    - Every proto specifies the messages and services that it handels.
    - a message has a name and a set of components/fields; each component/field has a type and an id, which specifies the order (to avoid potential conflicts with different languages). This also means that a response can return multiple results with multiple types.

- Each service has to be defined, and in each service there can be multiple functionalities, denoted by the keyword *rpc*. Each functionality has a name (*hello*) that uses one of the messages and returns another of the messages

```
syntax = "proto3";
option java_multiple_files = true;

package it.uniroma1.gRPCExample;

message HelloRequest {
    string firstName = 1;
    string lastName = 2;
}

message HelloResponse {
    string greeting = 1;
}

service HelloService {
    rpc hello(HelloRequest) returns (HelloResponse);
```

- Once we ahve defined our proto, we define our impleemtnation

- HelloServiceImpl
  - the class helloServiceImpl extends HelloServiceImplBase , which is the stub generated by the precompiler (in the generated sources).
  - In this case the class implemetns only one method (becasue we declarede only one service in the proto)
  - Typically the method takes as input an input message objecT (helloRequest) and a streamObserver object of the response mesasge (StreamObserver<HelloResponse> responseObserver). predetermined. The observer points to the observed object, so that when it changes, a callback method is called
  - The logic of the method is then implemetnd. in our case its very simple, it takes the two fields of the input message and produces a string. then a response is built through a new builder. Note that there is a set method for the fields of the response message.

- Finally the responseObserver is set so that the next response available is our messageE(onNext), and then onCompleted is called to tell we are finished

```
String greeting = new StringBuilder()
  .append("Hello, ")
  .append(request.getFirstName())
  .append(" ")
  .append(request.getLastName())
  .toString();

HelloResponse response = HelloResponse.newBuilder()
  .setGreeting(greeting)
  .build();

responseObserver.onNext(response);
responseObserver.onCompleted();
```

- MyServer
  - Completes thje server implementation and has only a main
  - it creates a server object and uses the classes serverbuuilder. The server is started
  - The server keeps going until it's interrupted

```
public class MyServer {
    public static void main(String[] args) throws IOException, InterruptedException {
        Server server;
        server = ServerBuilder
                .forPort(8080)
                .addService(new HelloServiceImpl()).build();

        server.start();
        server.awaitTermination();
    }
}
```

- POM File
  - A file for maven that specifies dependencies and build instructions. Needs to be the same in the client (at least java)

Client (Java)

- Proto File
  - Same as before because the stub as to be generated in the same way as the server can understand it

- Client
  - the main class needs to create a channel (ManagedChannel) with some java jargon.

- Then we create a stub object of type determined by precompilation.
- Then we call the methods offered by the interface on the stub
- at the end we shut down the channel

```java
public class Client {
    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 8080)
            .usePlaintext()
            .build();

        HelloServiceGrpc.HelloServiceBlockingStub stub
          = HelloServiceGrpc.newBlockingStub(channel);

        HelloResponse helloResponse = stub.hello(HelloRequest.newBuilder()
            .setFirstName("Massimo")
            .setLastName("Mecella")
            .build());

        System.out.println("Response received from server:\n" + helloResponse);

        channel.shutdown();
    }
}
```

Client (Python)

- Proto
  - Same file as before. Will automatically skip java options.
  - The stub is generated thorugh commands in the terminal (see txt instructions)

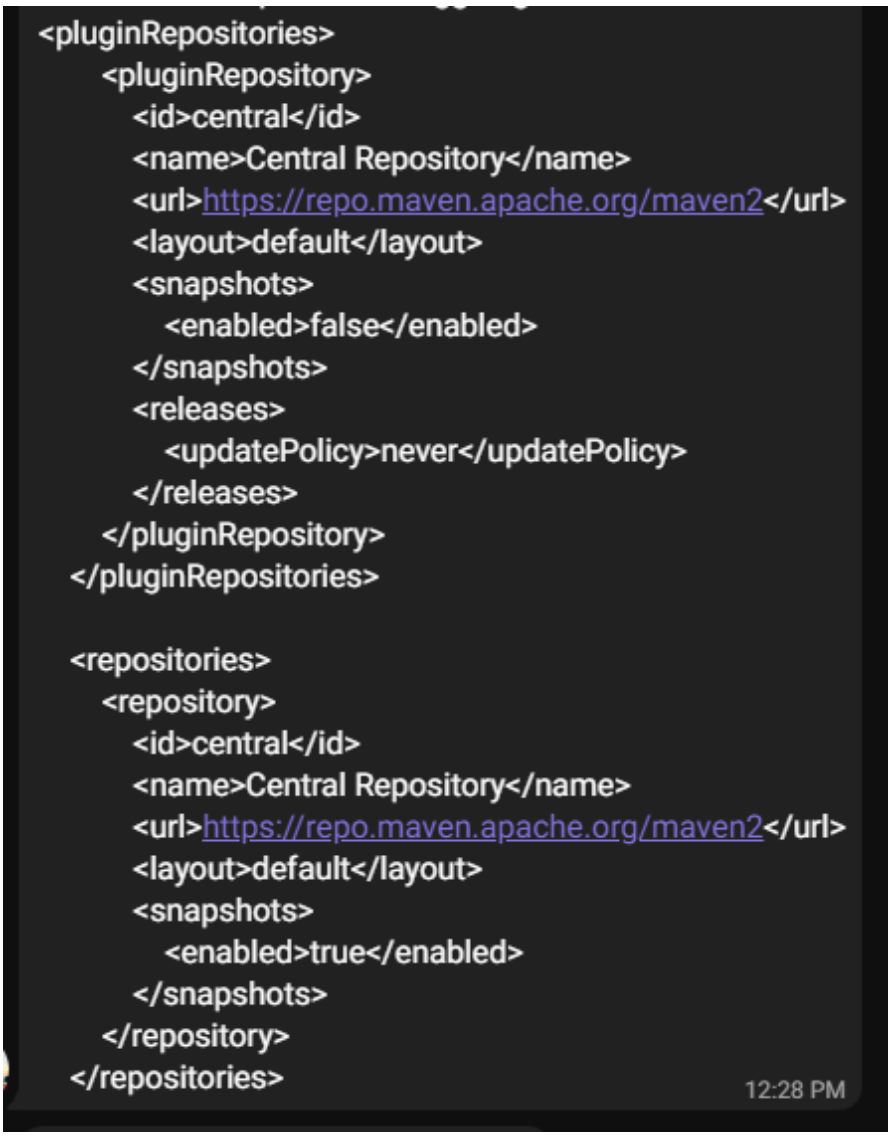
- Client
  - Same logic as before but with python syntax
  - Create a channel, build stub on the channel, create message using stub and service.

```python
import logging
import grpc
import HelloService_pb2
import HelloService_pb2_grpc

def run():
    with grpc.insecure_channel('localhost:8080') as channel:
        stub = HelloService_pb2_grpc.HelloServiceStub(channel)
        response = stub.hello(HelloService_pb2.HelloRequest(firstName='Giulia',lastName='Rossi'))
        print("The client received: " + response.greeting)

if __name__ == '__main__':
    logging.basicConfig()
    run()
```

  -

can't do the projects in netbeans btw fkn cringe jesuschrist

general pom code to help download dependencies


<pluginRepositories>
<pluginRepository>
<id>central</id>
<name>Central Repository</name>
<url>https://repo.maven.apache.org/maven2</url>
<layout>default</layout>
<snapshots>
<enabled>false</enabled>
</snapshots>
<releases>
<updatePolicy>never</updatePolicy>
</releases>
</pluginRepository>
</pluginRepositories>
<repositories>
<repository>
<id>central</id>
<name>Central Repository</name>
<url>https://repo.maven.apache.org/maven2</url>
<layout>default</layout>
<snapshots>
<enabled>true</enabled>
</snapshots>
</repository>
</repositories>
12:28 PM