Author: Francesco Polichetti
Version: 2024-03-19

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?**

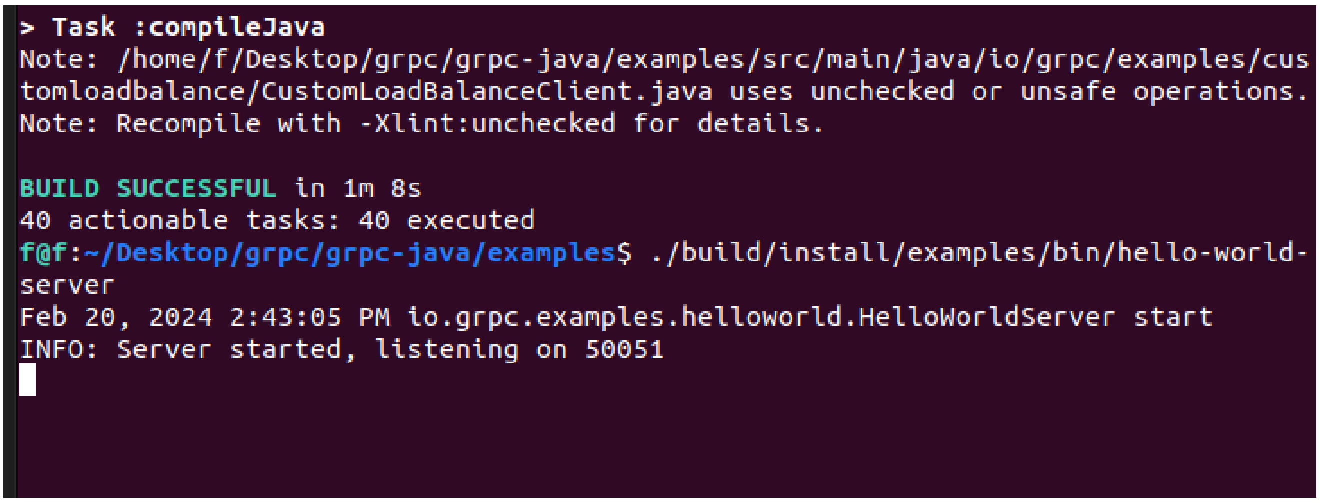
  For applications requiring dynamic data structures with high levels of flexibility or when human-readable format is a strict requirement.
- **List 3 different data types that can be used with protocol buffers?**

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

Basic Tasks

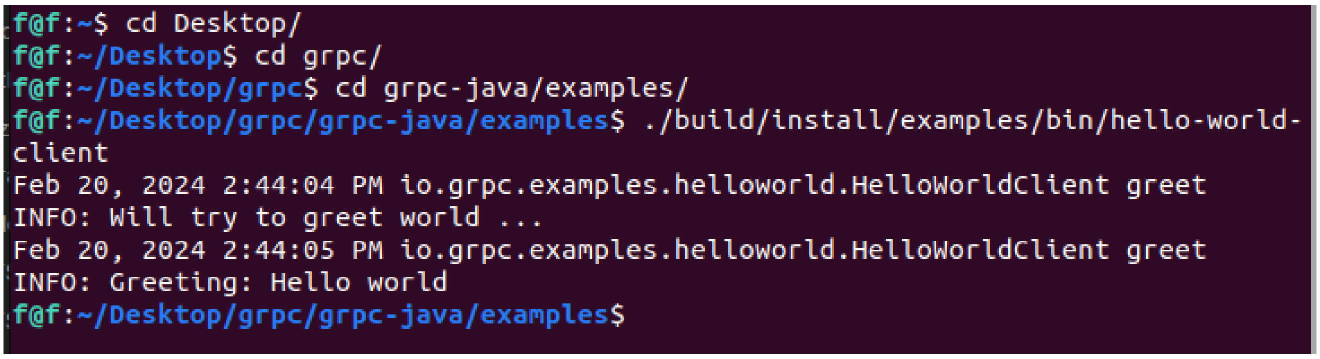
Step 1: Simple Installation

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:

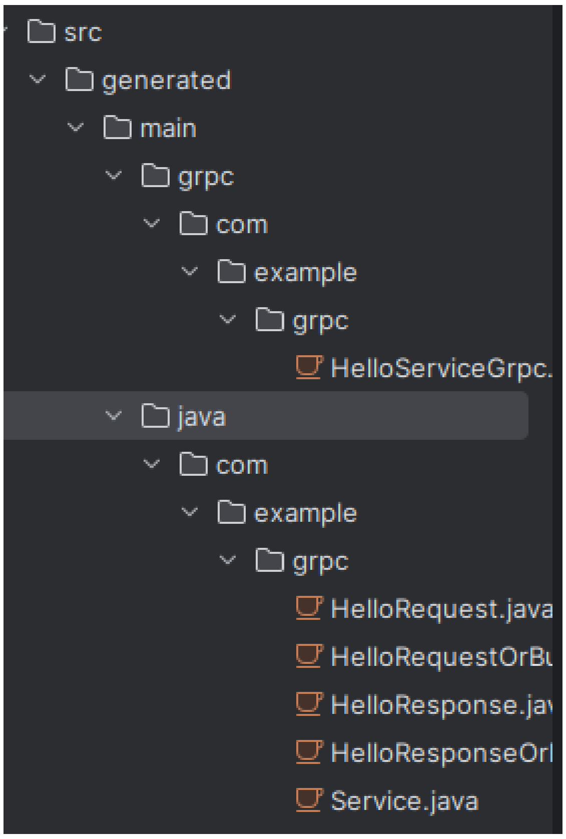
1. creating new Project using Java and Gradle
2. Create a .proto Configuration File under main > proto > service.proto that is used for shaping the service which is going to be needed, in my examplary case:

In this the **proto3** syntax is being used to shape the applications resources in this a Request consists of a firstname and a lastname, a response contains a single text.

```
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:

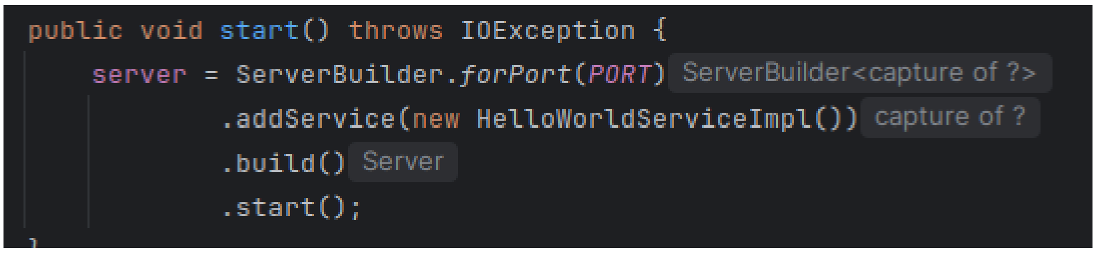
```
plugins {
    id 'java'
}

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

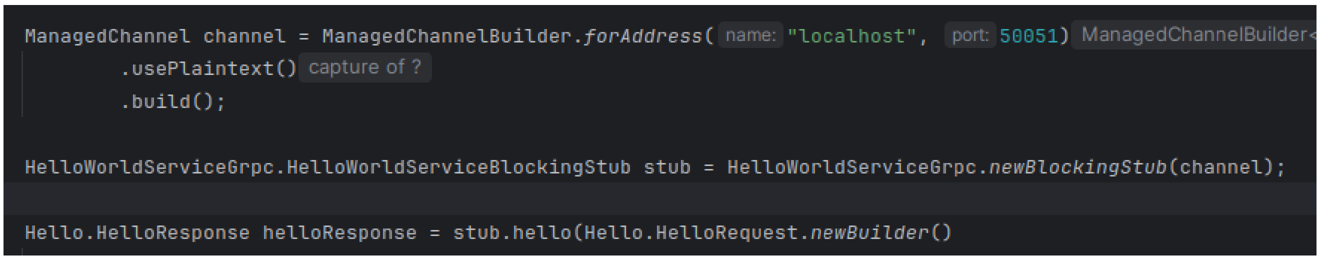
Implementation

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) ServerBuilder<capture of ?>
            .addService(new HelloWorldServiceImpl()) capture of ?
            .build() Server
            .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( name: "localhost",  port: 50051) ManagedChannelBuilder<
        .usePlaintext() capture of ?
        .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.

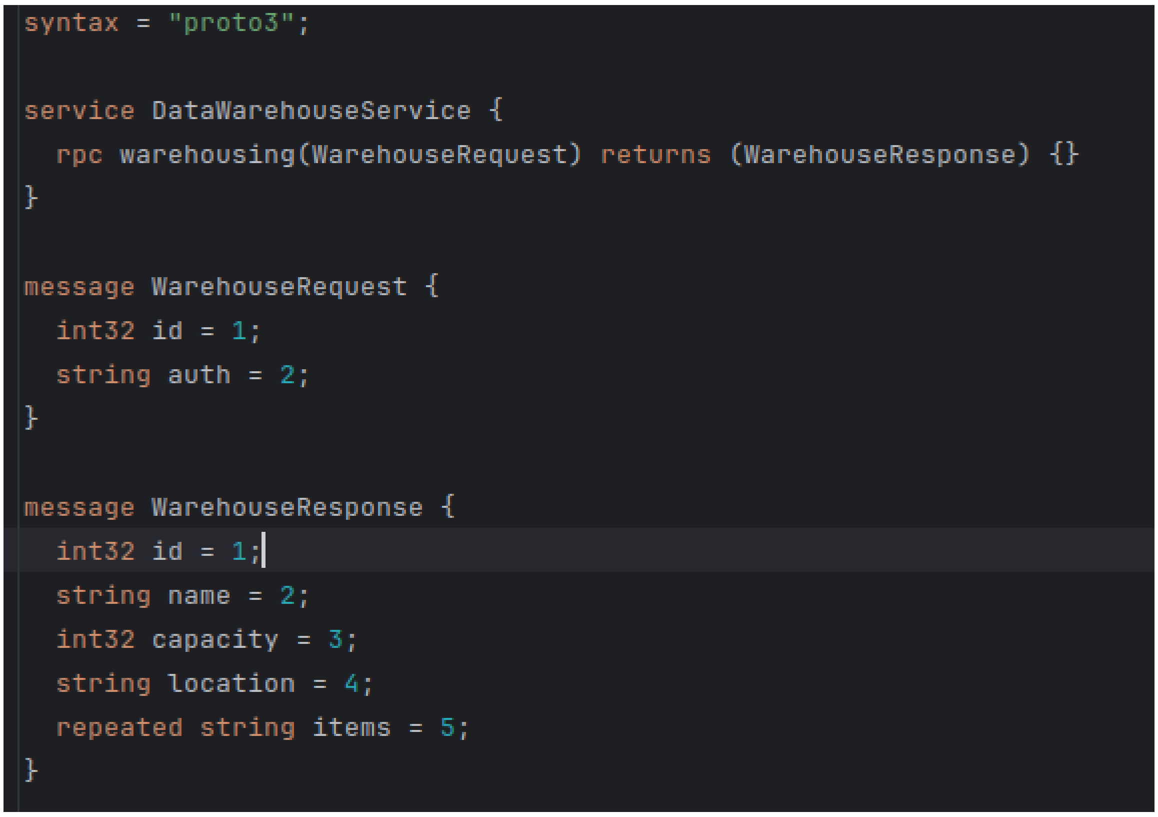
And now, onto the fun part:

Creating a datawarehouse

For this i took the schematic i just built and changed everything needed for it to now act as a datawarehouse instead of a simple helloWorld Service.

That's why I started by:

Changing the projects response and request parameters by adjusting the proto file, to now look like this:



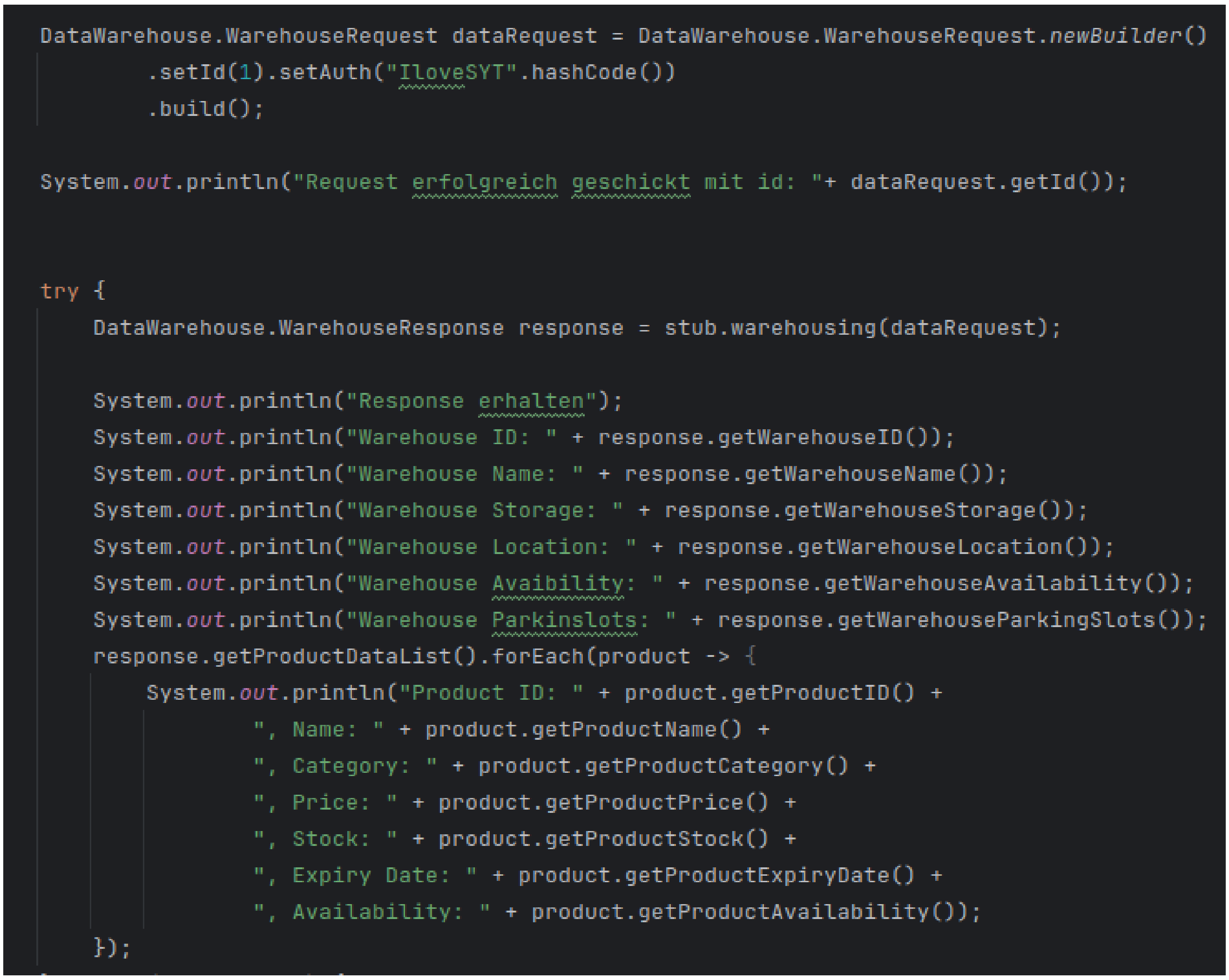
```
syntax = "proto3";

service DataWarehouseService {
  rpc warehousing(WarehouseRequest) returns (WarehouseResponse) {}
}

message WarehouseRequest {
  int32 id = 1;
  string auth = 2;
}

message WarehouseResponse {
  int32 id = 1;
  string name = 2;
  int32 capacity = 3;
  string location = 4;
  repeated string items = 5;
}
```

This defines a Request that goes from the user containing an ID and a way of authentication (more on that later) and a response which returns various data from a warehouse.

After that I cleaned and rebuilt the project

I also modified the Client to be able to send a request to the Server containing an ID and a Authentification which is just the Hash-value of the String "IloveSYT".

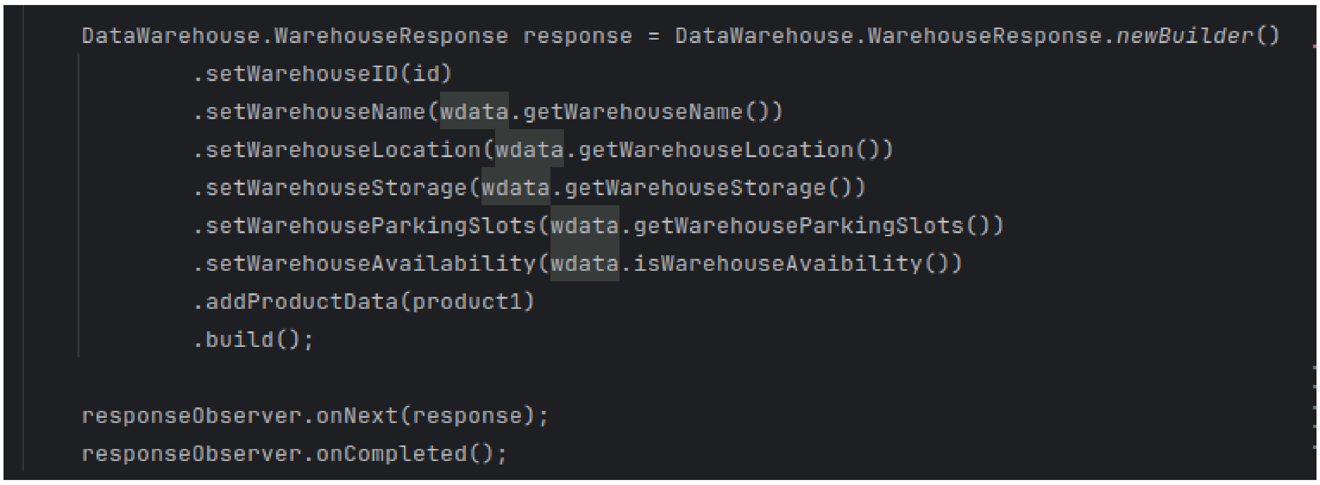
This then looks like the following after implementing the functionality to display the contents of a response:



```java
DataWarehouse.WarehouseRequest dataRequest = DataWarehouse.WarehouseRequest.newBuilder()
        .setId(1).setAuth("IloveSYT".hashCode())
        .build();

System.out.println("Request erfolgreich geschickt mit id: "+ dataRequest.getId());


try {
    DataWarehouse.WarehouseResponse response = stub.warehousing(dataRequest);

    System.out.println("Response erhalten");
    System.out.println("Warehouse ID: " + response.getWarehouseID());
    System.out.println("Warehouse Name: " + response.getWarehouseName());
    System.out.println("Warehouse Storage: " + response.getWarehouseStorage());
    System.out.println("Warehouse Location: " + response.getWarehouseLocation());
    System.out.println("Warehouse Avaibility: " + response.getWarehouseAvailability());
    System.out.println("Warehouse Parkinslots: " + response.getWarehouseParkingSlots());
    response.getProductDataList().forEach(product -> {
        System.out.println("Product ID: " + product.getProductID() +
                ", Name: " + product.getProductName() +
                ", Category: " + product.getProductCategory() +
                ", Price: " + product.getProductPrice() +
                ", Stock: " + product.getProductStock() +
                ", Expiry Date: " + product.getProductExpiryDate() +
                ", Availability: " + product.getProductAvailability());
    });
```

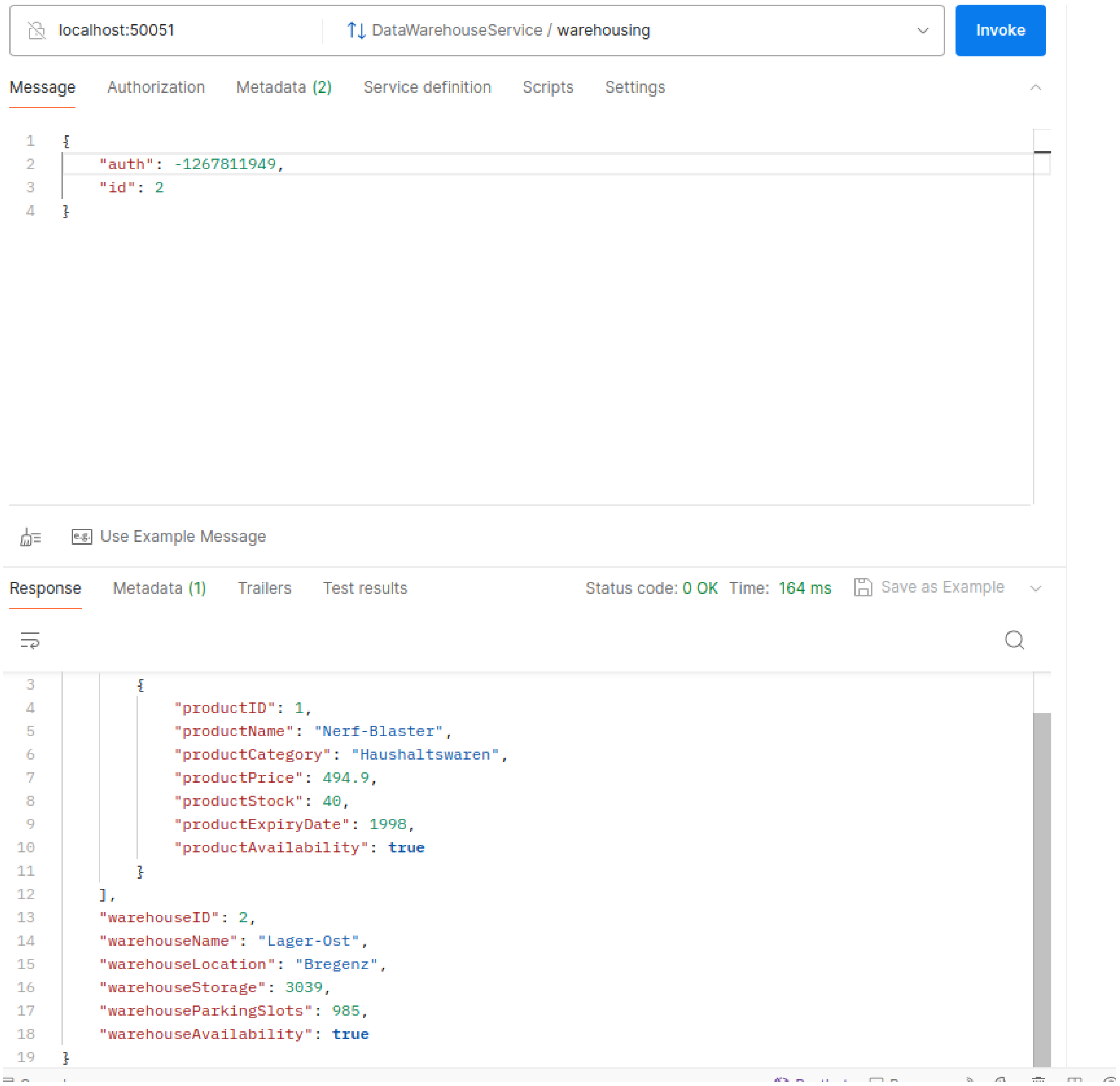
After that, I altered the Server part.

For this I added my previous work that implemented a random data generator for WarehouseData and ProductData. This is being used to build up a message that is then being forged into a response and added to the response observer as seen here:



```java
DataWarehouse.WarehouseResponse response = DataWarehouse.WarehouseResponse.newBuilder()
        .setWarehouseID(id)
        .setWarehouseName(wdata.getWarehouseName())
        .setWarehouseLocation(wdata.getWarehouseLocation())
        .setWarehouseStorage(wdata.getWarehouseStorage())
        .setWarehouseParkingSlots(wdata.getWarehouseParkingSlots())
        .setWarehouseAvailability(wdata.isWarehouseAvaibility())
        .addProductData(product1)
        .build();

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

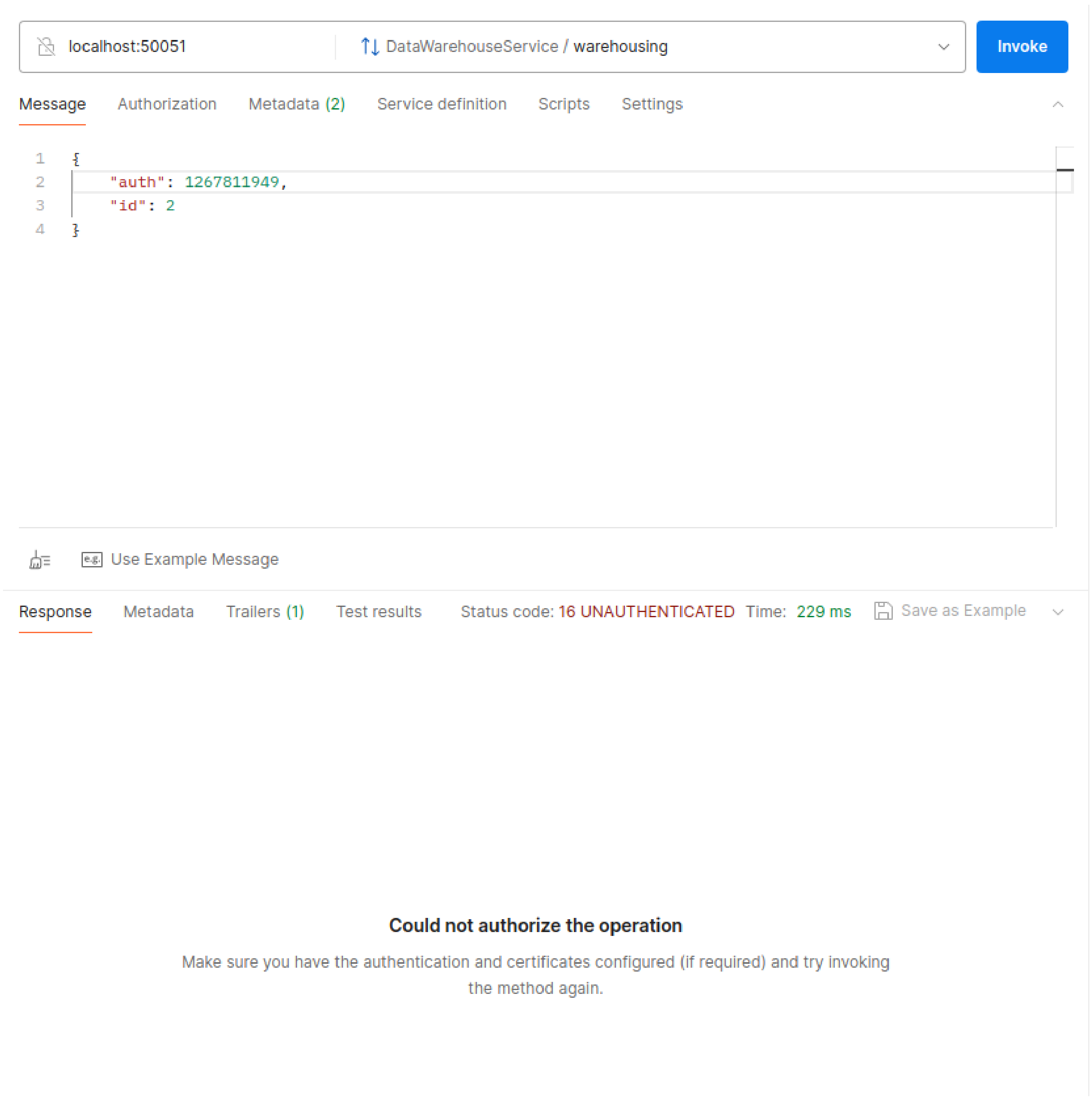
All of this can be tested by using Postman for example. When setting the auth to the hashvalue of "IloveSYT" and a valid ID this is the result:



```
{
    "auth": -1267811949,
    "id": 2
}
```

```
        {
            "productID": 1,
            "productName": "Nerf-Blaster",
            "productCategory": "Haushaltswaren",
            "productPrice": 494.9,
            "productStock": 40,
            "productExpiryDate": 1998,
            "productAvailability": true
        }
    ],
    "warehouseID": 2,
    "warehouseName": "Lager-Ost",
    "warehouseLocation": "Bregenz",
    "warehouseStorage": 3039,
    "warehouseParkingSlots": 985,
    "warehouseAvailability": true
}
```

If the auth is wrong, there is no response being generated:



```
{
    "auth": 1267811949,
    "id": 2
}
```

Status code: 16 UNAUTHENTICATED Time: 229 ms

Could not authorize the operation

Make sure you have the authentication and certificates configured (if required) and try invoking the method again.

That's it for the Basic Tasks, now onto the fun part:

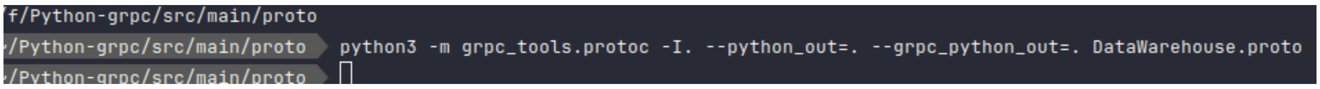
Extended Tasks

1. Implementation of a client using another programming language (Python)

For this, I firstly cloned my previous project to now add the python part to this new project in my IDE. Then, I installed the needed packages using:

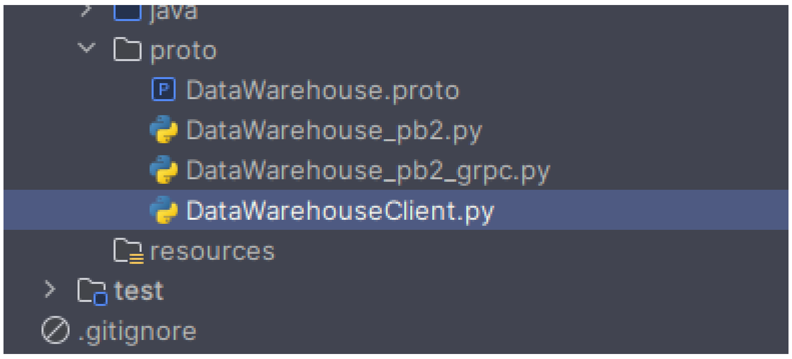
```
pip install grpcio grpcio-tools
```

Then, I entered the following into the command prompt so that the GRPC Projects gets built correctly using the python compiler:



```
f/Python-grpc/src/main/proto
/Python-grpc/src/main/proto  python3 -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. DataWarehouse.proto
/Python-grpc/src/main/proto
```

This then lead to the creation of the following resource files for python implementation:



Now onto the implementation using Python:

The implementation in Python mostly works just like in Java for example like here:

```
print("Warehouse Parking Slots:", response.warehouseParkingSlots)

for product in response.productData:

print("Product ID:", product.productID,

"Name:", product.productName,

"Category:", product.productCategory,

"Price:", product.productPrice,

"Stock:", product.productStock,

"Expiry Date:", product.productExpiryDate,

"Availability:", product.productAvailability)
```

And now onto the final part, health checking:

Health Checking

I designed a health checking system in which the server pings its client(s) every second. When there is a valid answer to the ping the status is healthy, if not its "not responding". Crucial codes are:

Changing the proto File:

adding `rpc ping(PingRequest) returns (PingResponse) {}`

and

```
message PingRequest {
  string clientID = 1;
}

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

so that there are services and messages used by the pinging principle

adding a HealthCheckManager which covers all the needed functions to monitor the status and write it into a logfile:

```
for (Map.Entry<String, Long> entry :
DataWarehouseServiceImpl.activeClients.entrySet()) {
    long lastSeen = System.currentTimeMillis() - entry.getValue();
    String status = lastSeen < 2000 ? "healthy" : "no response";
    try (FileWriter fw = new FileWriter("/home/f/Java-grpc/logs.txt",
true)) {
        fw.write(entry.getKey() +", " +LocalTime.now() + ",
"+status+"\n");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Here a FileWriter is used to write down the id of the currently selected Client, the local time and the current status so healthy/not responding.

Other than that I also added the following:

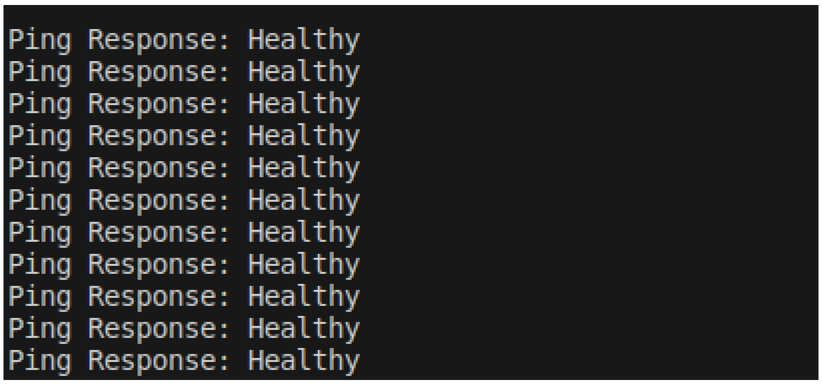
```
@Override
public void ping(DataWarehouse.PingRequest request,
StreamObserver<DataWarehouse.PingResponse> responseObserver) {
    String clientId = request.getClientID();
    if (activeClients.containsKey(clientId)) {
        activeClients.put(clientId, System.currentTimeMillis());

responseObserver.onNext(DataWarehouse.PingResponse.newBuilder().setMessage
("Healthy").build());
    } else {

responseObserver.onNext(DataWarehouse.PingResponse.newBuilder().setMessage
("Unknown client").build());
    }
    responseObserver.onCompleted();
}
```

A simple ping function which is used to transfer requests and responses to get the status of the client to the server

and finally adding the pinging function to the python client:

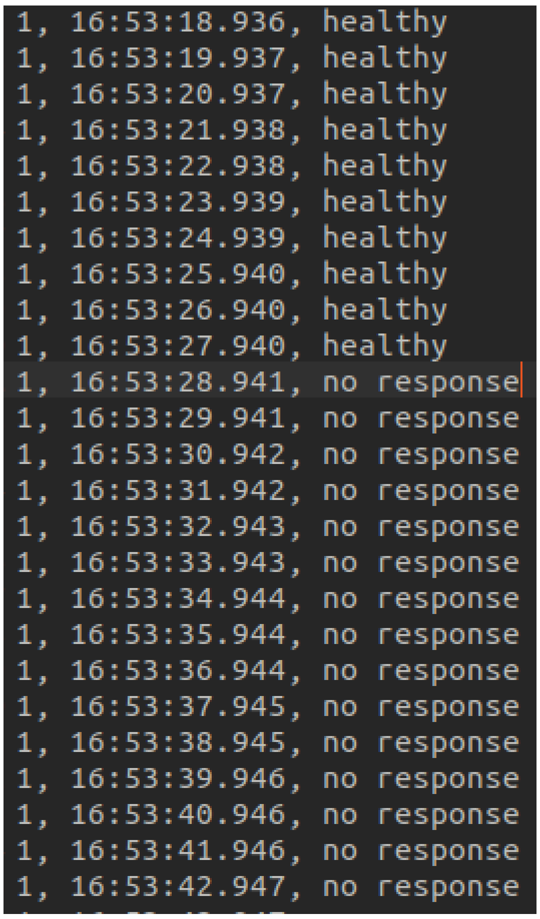
```
while True:

try:

ping_response = stub.ping(PingRequest(clientID="1"))

print(f"Ping Response: {ping_response.message}")

except grpc.RpcError as e:

print(f"Fehler beim Pingen: {e.details()}")

time.sleep(1)
```

And that's pretty much it the output looks like the following:



```
Ping Response: Healthy
Ping Response: Healthy
Ping Response: Healthy
Ping Response: Healthy
Ping Response: Healthy
Ping Response: Healthy
Ping Response: Healthy
Ping Response: Healthy
Ping Response: Healthy
Ping Response: Healthy
Ping Response: Healthy
```

and



```
1, 16:53:18.936, healthy
1, 16:53:19.937, healthy
1, 16:53:20.937, healthy
1, 16:53:21.938, healthy
1, 16:53:22.938, healthy
1, 16:53:23.939, healthy
1, 16:53:24.939, healthy
1, 16:53:25.940, healthy
1, 16:53:26.940, healthy
1, 16:53:27.940, healthy
1, 16:53:28.941, no response
1, 16:53:29.941, no response
1, 16:53:30.942, no response
1, 16:53:31.942, no response
1, 16:53:32.943, no response
1, 16:53:33.943, no response
1, 16:53:34.944, no response
1, 16:53:35.944, no response
1, 16:53:36.944, no response
1, 16:53:37.945, no response
1, 16:53:38.945, no response
1, 16:53:39.946, no response
1, 16:53:40.946, no response
1, 16:53:41.946, no response
1, 16:53:42.947, no response
```