GRPC

Author: Kevin Bauer

Its requested to use proto3 as the Language.

Working steps

First I looked at the Source from the Prof., but I noticed that it isn't working because of following few things:

```
implementation "io.grpc:grpc-netty:${grpcVersion}"
implementation "io.grpc:grpc-protobuf:${grpcVersion}"
implementation "io.grpc:grpc-stub:${grpcVersion}"
implementation group: 'com.google.protobuf', name: 'protobuf-java-util', version: '3.12.2'
```

The newer versions of gradle use implementation instead of compile.

Im not sure why to use another version here but Prof. recommended it.

```
dependencies {
    classpath 'com.google.protobuf:protobuf-gradle-plugin:0.9.4'
}
```

Definition of the protocol

```
syntax = "proto3";

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

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

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

Starting the server:

```
public void start() throws IOException {
    server = ServerBuilder.forPort(PORT) ServerBuilder<capture of ?>
            .addService(new HelloWorldServiceImpl()) capture of ?
            .build() Server
            .start();
}

1 usage  kbau3r
public void blockUntilShutdown() throws InterruptedException {
    if (server == null) {
        return;
    }
    server.awaitTermination();
}
```

Definition for the Datawarehouse:

```
syntax = "proto3";

package warehouse;

message Product {
  string productID = 1;
  string productName = 2;
  string productCategory = 3;
  int32 productQuantity = 4;
  string productUnit = 5;
}

message Warehouse {
  string warehouseID = 1;
  string warehouseName = 2;
  string timestamp = 3;
  string warehouseAddress = 4;
  repeated Product products = 5;
}

message WarehouseRequest {
  string warehouseID = 1;
}

message WarehouseResponse {
  repeated Warehouse warehouses = 1;
}

service WarehouseService {
  rpc GetWarehouse(WarehouseRequest) returns (WarehouseResponse);
}
```

Adding sample prodcuts:

```
Product product = Product.newBuilder()
        .setProductID("P001")
        .setProductName("Beispielprodukt")
        .setProductCategory("Kategorie")
        .setProductQuantity(100)
        .setProductUnit("Stück")
        .build();

Product product2 = Product.newBuilder()
        .setProductID("P002")
        .setProductName("Beispielprodukt")
        .setProductCategory("Kategorie")
        .setProductQuantity(100)
        .setProductUnit("Stück")
        .build();

Product product3 = Product.newBuilder()
        .setProductID("P003")
        .setProductName("Beispielprodukt")
        .setProductCategory("Kategorie")
        .setProductQuantity(100)
        .setProductUnit("Stück")
        .build();

List<Product> productList = new ArrayList<>();
productList.add(product);
productList.add(product2);
productList.add(product3);
```

Creating the warehouse and adding the products:

```
Warehouse warehouse = Warehouse.newBuilder()
        .setWarehouseID("W001")
        .setWarehouseName("Hauptlager")
        .setTimestamp("2023-01-01T00:00:00Z")
        .setWarehouseAddress("Lagerstraße 1, 12345 Lagerstadt")
        .addAllProducts(productList)
        .build();

WarehouseResponse response = WarehouseResponse.newBuilder()
        .addWarehouses(warehouse)
        .build();
```

Questions

- **What is gRPC and why does it work across languages and platforms?** gRPC is a high-performance, open-source universal RPC (Remote Procedure Call) framework that enables client and server applications to communicate transparently and efficiently. It works across languages and platforms by using Protocol Buffers as its interface definition language (IDL), which allows for the definition of simple and complex data structures in a language-neutral way. This ensures seamless interoperability among services written in different programming languages, as the protocol buffer compiler can generate source code for the service interface and message types for a variety of languages.
- **Describe the RPC life cycle starting with the RPC client?**
    1. **Call Invocation:** The RPC client starts by invoking a call to a procedure or method on the server as if it were a local function.
    2. **Request Serialization:** The client serializes the request data into a binary format (using Protocol Buffers, for example) and sends it over the network to the server.
    3. **Server Handling:** The server receives the request, deserializes it to understand which method is being called, and processes the request.
    4. **Response Serialization:** The server serializes the response data into a binary format and sends it back to the client.
    5. **Client Deserialization:** The client deserializes the response data from the binary format back into a usable object or data structure.
    6. **Completion:** The client receives the response, and the RPC call is completed.
- **Describe the workflow of Protocol Buffers?**
    1. **Definition:** First, define the structure of the data (messages) and the service interfaces in a `.proto` file using the Protocol Buffers language.
    2. **Compilation:** Use the Protocol Buffers compiler (`protoc`) to generate source code from the `.proto` file for the desired target languages. This source code includes data access classes for each message and client and server code for each service.
    3. **Integration:** Integrate the generated source code into your client and server applications. This code is used to serialize and deserialize the Protocol Buffers messages.
    4. **Communication:** Client and server applications communicate by serializing message data to a binary format that can be sent over the network and deserializing the received message back into the application-specific data structures.
- **What are the benefits of using protocol buffers?**
    - **Efficiency:** Protocol Buffers serialize data into a smaller, faster format compared to JSON or XML, reducing bandwidth and storage requirements.
    - **Interoperability:** They enable seamless communication between services written in different languages.
    - **Scalability:** Protocol Buffers are designed to support backward and forward compatibility, making it easier to evolve your data model.
    - **Strong Typing:** They enforce a schema for your data, reducing the chances of errors.
- **When is the use of protocol not recommended?**

    - **Human-Readable Formats Needed:** When the data needs to be easily readable and editable by humans without additional tools, as Protocol Buffers are binary.
    - **Simple Projects:** For very simple projects or when the overhead of adding a serialization/deserialization layer is not justified.
    - **Limited Support Environments:** In environments or platforms with limited support for Protocol Buffers or where integrating the `protoc` compiler is challenging.
- **List 3 different data types that can be used with protocol buffers?**
    1. **Scalars:** Basic data types like `int32`, `float`, `bool`, and `string`.
    2. **Enums:** Enumerated types, which are a list of possible values for a field.
    3. **Messages:** Complex types, which are essentially custom structures composed of other data types, including other messages, scalars, and enums.

Quellen

Recommended by the teacher:
https://intuting.medium.com/implement-grpc-service-using-java-gradle-7a54258b60b8