#CS #school

Author: Oliwier Przewlocki

preparation & structure

This assignment requires 4 projects.

1. The library that contains the proto files
2. The voting project that has been modified to be a server and a client at once
3. The server that acts as a middle man between the client and the voting center
4. The client that requests the data

The base dependencies that allow grpc to work are:

```
<dependency>
    <groupId>net.devh</groupId>
    <artifactId>grpc-server-spring-boot-starter</artifactId>
    <version>2.15.0.RELEASE</version>
</dependency>

<dependency>
    <groupId>net.devh</groupId>
    <artifactId>grpc-client-spring-boot-starter</artifactId>
    <version>2.15.0.RELEASE</version>
</dependency>

<dependency>
    <groupId>com.oliwier</groupId>
    <artifactId>proto-library</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>

<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.21.12</version>
</dependency>

<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-protobuf</artifactId>
    <version>1.56.0</version>
</dependency>

<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-stub</artifactId>
```

```
    <version>1.56.0</version>
</dependency>

<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty-shaded</artifactId>
    <version>1.56.0</version>
</dependency>
```

gkü/hello world

The gkü task is to create a connection between a client and a server, where the clients sends a request and passes on a name parameter and the server sends a hello message back.

proto library

The first step is to define the proto file that will be used as a blueprint for communication between client and server, regardless of the language used.

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "com.oliwier.helloworld";
option java_outer_classname = "HelloWorldProto";

package helloworld;

service HelloWorldService {
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

message HelloRequest {
  string name = 1;
}

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

> Proto files are generally used to define the *service* and *message* types in a “platform-agnostic” way.

- The **service** block defines the RPC methods that the server will provie. Each method has a request and response.
- The message block defines the structure of data that are exchanged. They’re serialized using protocol buffers, which is binary.
- Regarding the metadata, there are two syntaxes, `proto3` and `proto2`, it’s recommended to use proto3. the `java_multiple_files` set to true generates multiple files, by default it’s

all put into one single file, the `java_package` specifies the namespacing, ergo the package where the proto will be placed, the `java_outer_classname` defines the outer java class. Normally protobufs compiler wraps all messages and services into an outer class if multiple files aren't used. Here, it will generate a "HelloWorldProto" class that will contain inner classes/references to the generated code.

> ⚠ **Attention**
>
> The proto files need to be created in a `src/main/proto/` directory if the default path isn't changed.

For the correct class/file generation, the following needs to be written into the `pom.xml` file:

```
<build>
    <extensions>
        <extension>
            <groupId>kr.motd.maven</groupId>
            <artifactId>os-maven-plugin</artifactId>
            <version>1.6.2</version>
        </extension>
    </extensions>
    <plugins>
        <!-- Protobuf plugin -->
        <plugin>
            <groupId>org.xolstice.maven.plugins</groupId>
            <artifactId>protobuf-maven-plugin</artifactId>
            <version>0.6.1</version>
            <configuration>
                <protocArtifact>com.google.protobuf:protoc:3.21.12:exe:${os.detected.classifie
                <pluginId>grpc-java</pluginId>
                <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.56.0:exe:${os.detected.classifi
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                        <goal>compile-custom</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Now you just need to run `mvn clean install` and it should compile.

The library can be added via:

```
<dependency>
    <groupId>com.oliwier</groupId>
    <artifactId>proto-library</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

server

For the server, we just need to write a service class:

```
@GrpcService
public class HelloWorldServiceImpl extends HelloWorldServiceGrpc.HelloWorldServiceImplBase {

    @Override
    public void sayHello(HelloRequest request, StreamObserver<HelloReply> responseObserver) {
        String message = "Hello, " + request.getName();
        HelloReply reply = HelloReply.newBuilder().setMessage(message).build();
        responseObserver.onNext(reply);
        responseObserver.onCompleted();
    }
}
```

Because I defined a `HelloWorldService` in the proto file, it generated me a base class called `HelloWorldServiceGrpc.HelloWorldServiceImplBase` that I can extend to then be able to override the sayHello method accepting a `HelloRequest`.

Furthermore,

- `@GrpcService` marks the class as a grpc service in the context of spring boot
- Because HelloRequest has a name field and protobuf generates getters, you can use the `getName()` method.
- Message objects are immutable, therefore you need to use the mutable builder method before setting the message (a field in the HelloReply message) and then building it as a immutable message object
- The `onNext()` and `onCompleted()` methods finalize the request

Because it is a server, you need to configure a grpc server port in `application.properties`:

```
grpc.server.port=6969
```

client

For the client, you need a service that sends a request via grcp and a controller that opens an endpoint to trigger the method in the service. We'll start with the service:

```
@Service
public class GrpcClientService {

    @GrpcClient("grpc-server")
    private HelloWorldServiceGrpc.HelloWorldServiceBlockingStub helloWorldServiceBlockingStub;

    public String sendMessage(String name) {
        HelloRequest request = HelloRequest.newBuilder().setName(name).build();
        HelloReply reply = helloWorldServiceBlockingStub.sayHello(request);
        return reply.getMessage();
    }
}
```

Here we build a request using the builder like in the server. It also uses the `@GrpcClient` annotation that injects the client, here it specifies that the client will connect to a server called `grpc-server`. That name depends on your `application.properties` configuration, which is mandatory to make proper use of that name:

```
grpc.client.grpc-server.address=static://localhost:6969
grpc.client.grpc-server.negotiationType=plaintext
```

> **Note**
>
> Notice how the port on the client for the server is the same port defined for the `grpc.server.port` field on the server.

The `helloWorldServiceBlockingStub` is essentially a connection to the server, so that you can call the `sayHello(request)` method which sends a request to the server via the address specified in the properties.

> **Meaning of "stub"**
>
> The term "stub" (e.g. in `helloWorldServiceBlockingStub`) means a **client-side proxy**, it allows for transparent communication with the server, so that the client doesn't need to handle the low-level stuff like network communication, serialization or deserialization

The last piece of logic is the rest controller.

```
@RestController
@RequestMapping("/api/v1/grpc")
@AllArgsConstructor
public class GrpcClientController {
```

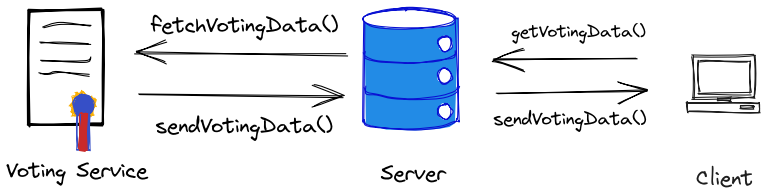
```
    private GrpcClientService grpcClientService;

    @GetMapping("/hello/{name}")
    public String sayHello(@PathVariable String name) {
        return grpcClientService.sendMessage(name);
    }}
```

This just triggers the sendMessage method in the client along with the specified **name**-parameter.

gkv/nationalwahlen

For this, the client needs to be able to request voting data from another project. I have set it up so that the voting project acts as a server to be able to send data to the actual server that acts as a middle man for the client.



First, we need to define the necessary proto file and build it

proto library

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "com.oliwier.voting";
option java_outer_classname = "VotingProto";

package voting;

service VotingService {
  rpc SendVotingData (EmptyRequest) returns (VotingReply) {}
}

message EmptyRequest {}

message VotingReply {
  string regionID = 1;
  string regionName = 2;
  string regionAddress = 3;
  string regionPostalCode = 4;
  string federalState = 5;
  repeated PartyData parties = 6;
}

message PartyData {
  string partyID = 1;
  string amountVotes = 2;
```

```
  repeated PreferredCandidate preferredCandidates = 3;
}

message PreferredCandidate {
  string candidateName = 1;
  string candidateVotes = 2;
}
```

We create a `EmptyRequest` because you don't need to specify anything as it is only required to exchange basic data.

voting project

Let us start with the voting project. Here, you will first need to specify the server grpc field in the `application.properties` file, just like in the server project

```
grpc.server.port=7070
```

Now there only needs to be one extra file created (along with the grpc spring dependencies that need to be added in `pom.xml`), that is a grpc service which overrides the `sendVotingData()` method by extending the `VotingServiceGrpc.VotingServiceImplBase` class:

```
@GrpcService
@AllArgsConstructor
public class VotingDummyDataSenderService extends VotingServiceGrpc.VotingServiceImplBase {

    private final WarehouseService warehouseService;

    @Override
    public void sendVotingData(EmptyRequest request, StreamObserver<VotingReply> responseObser
        WarehouseData warehouseData = warehouseService.getWarehouseData("001");

        VotingReply.Builder votingReplyBuilder = VotingReply.newBuilder()
                .setRegionID(warehouseData.getRegionID())
                .setRegionName(warehouseData.getRegionName())
                .setRegionAddress(warehouseData.getRegionAddress())
                .setRegionPostalCode(warehouseData.getRegionPostalCode())
                .setFederalState(warehouseData.getFederalState());

        //add parties...

        votingReplyBuilder.addParties(party1.build());
        votingReplyBuilder.addParties(party2.build());

        responseObserver.onNext(votingReplyBuilder.build());
        responseObserver.onCompleted();
    }
}
```

Those are dummy data and it just puts all the data from the service into the `VotingReply`.

server

Here we need two files, one to fetch the data from the voting project and one to send it back to the client. We'll start with the fetch

```
@Service
public class VotingDataFetchService {

    @GrpcClient("nationalwahlen-server")
    private VotingServiceGrpc.VotingServiceBlockingStub nationalwahlenStub;

    public VotingReply fetchVotingDataFromNationalwahlen() {
        EmptyRequest request = EmptyRequest.newBuilder().build();
        return nationalwahlenStub.sendVotingData(request);
    }
}
```

As you may notice, we also need to define the proper address for the `nationalwahlen-server` in the `application.properties`:

```
grpc.client.nationalwahlen-server.address=static://localhost:7070
grpc.client.nationalwahlen-server.negotiationType=plaintext
```

This works in the same way as the client that requests from the server in the gkü part of this assignment, so refer to it for further explaination.

Now we will construct the file that sends the data to the client:

```
@GrpcService
@AllArgsConstructor
public class VotingServiceImpl extends VotingServiceGrpc.VotingServiceImplBase {

    private final VotingDataFetchService votingDataSenderService;

    @Override
    public void sendVotingData(EmptyRequest request,
        StreamObserver<VotingReply> responseObserver) {

        VotingReply votingData = votingDataSenderService
            .fetchVotingDataFromNationalwahlen();

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

Here we use the service in the earlier file to fetch the data to the server and use it as our reply. That's pretty much all for the server.

client

In the GrcpClientService, we need to add another stub to set up a proxy with the server and get the voting data:

```
@Service
public class GrpcClientService {

    //the hello world method...

    @GrpcClient("grpc-server")
    private VotingServiceGrpc.VotingServiceBlockingStub votingServiceBlockingStub;

    public String getVotingData() throws Exception {
        EmptyRequest request = EmptyRequest.newBuilder().build();

        VotingReply votingReply =
            votingServiceBlockingStub.sendVotingData(request);

        return JsonFormat.printer().includingDefaultValueFields()
            .print(votingReply);
    }
}
```

We need to setup an extra stub with the server, because the hello world stub is of a different data type. We also use the JsonFormat function that is included in `import com.google.protobuf.util.JsonFormat;`. I had some problems importing this so be careful, this is the dependency I've used:

```
<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java-util</artifactId>
    <version>3.24.0</version>
</dependency>
```

This is required because the jackson json formatter doesnt seem to interpret the emptyMessage correctly and throws errors, but it works with the protobuf JsonFormat.

The last step is to set up a rest endpoint for the client to be able to send a get request:

```
@RestController
@AllArgsConstructor
public class VotingClientController {

    private final GrpcClientService grpcClientService;
```

```
    @GetMapping("/get-voting-data")
    public String getVotingData() {
        try {
            return grpcClientService.getVotingData();
        } catch (Exception e) {
            return "Error fetching voting data: " + e.getMessage();
        }
    }
}
```

With that, our project is complete.