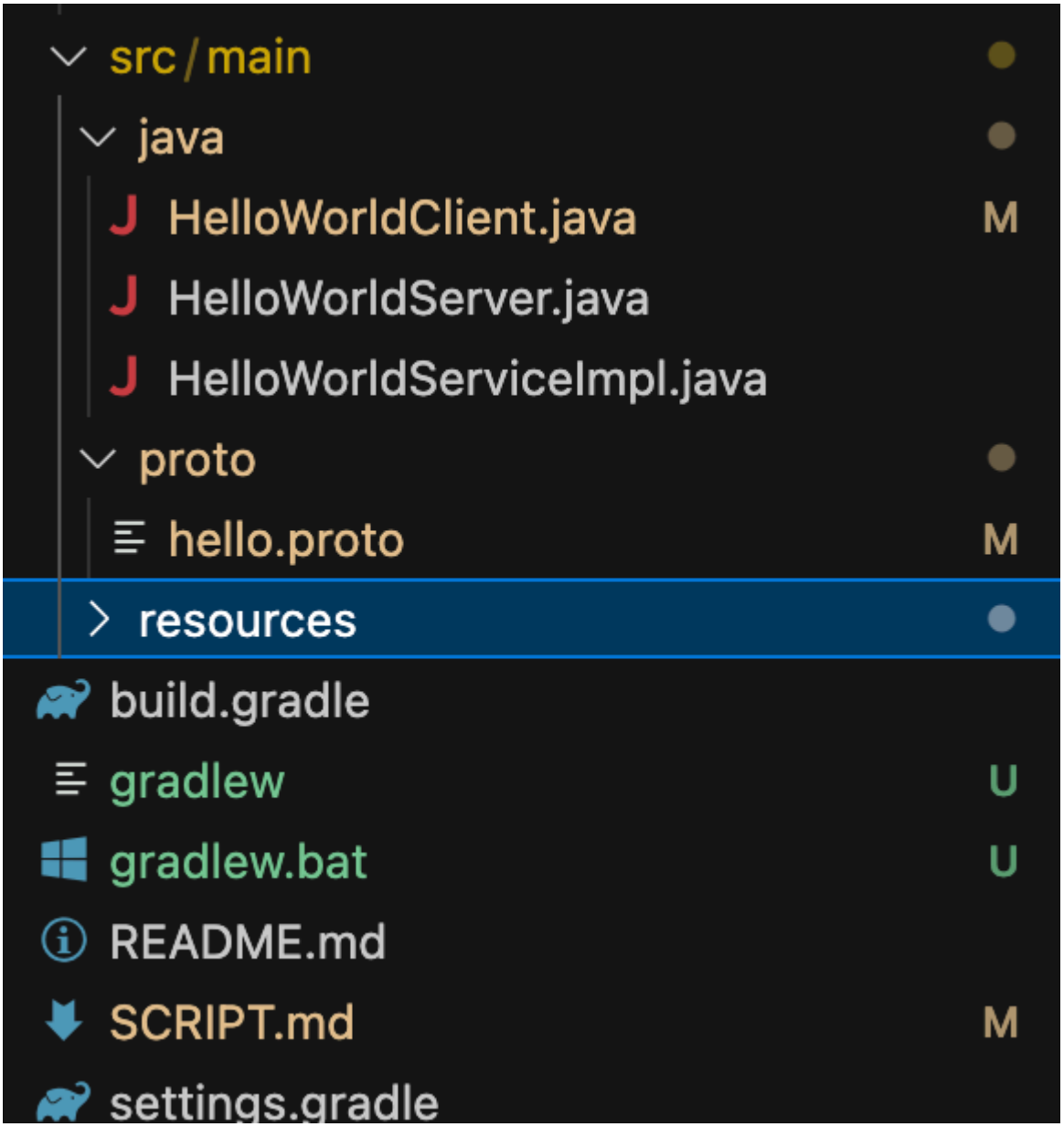
MidEng 7.2 gRPC Framework

1. Was ist gRPC und warum funktioniert es sprach- und plattformübergreifend?
   gRPC ist ein Framework, mit dem man Funktionen auf anderen Rechnern aufrufen kann („Remote Procedure Calls“).
   Es funktioniert über verschiedene Sprachen hinweg, weil alle Services mit **Protocol Buffers** beschrieben werden und der Compiler daraus automatisch passenden Code für Java, Python, Go usw. erzeugt.
2. Beschreibe den RPC-Lifecycle des Clients
   1. Client ruft eine Methode des Stubs auf.
   2. Stub verpackt die Daten in eine Protobuf-Nachricht.
   3. Nachricht wird über HTTP/2 an den Server gesendet.
   4. Server verarbeitet die Anfrage.
   5. Server sendet Antwort zurück.
   6. Client erhält die Antwort und nutzt sie weiter.
3. Workflow von Protocol Buffers
   1. Nachrichten & Services in einer `.proto` Datei definieren.
   2. Mit `protoc` Code generieren.
   3. Die generierten Klassen zum Senden/Empfangen von Daten verwenden.
4. Vorteile von Protocol Buffers
   - Sehr schnell
   - Sehr kleine Datenmengen
   - Klar definierte Schnittstellen
   - Funktioniert in vielen Sprachen
   - Gut geeignet für Microservices
5. Wann sollte man Protobuf nicht verwenden?
   - Wenn die Daten **menschenlesbar** sein müssen (z. B. JSON → Debugging).
   - Wenn man **direkt im Browser** Daten austauschen will.
   - Wenn das Datenformat **ständig wechselt**.
6. Drei Datentypen von Protobuf

   - string
   - int32
   - bool

erstmal meine Struktur:




```java
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;

public class HelloWorldClient {

    Run | Debug
    public static void main(String[] args) {

        String firstname = args.length > 0 ? args[0] : "Max";
        String lastname  = args.length > 1 ? args[1] : "Mustermann";

        ManagedChannel channel = ManagedChannelBuilder.forAddress(name: "localhost", port: 50051)
                .usePlaintext()
                .build();

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

        Hello.HelloResponse helloResponse = stub.hello(Hello.HelloRequest.newBuilder()
                .setFirstname(firstname)
                .setLastname(lastname)
                .build());
        System.out.println( helloResponse.getText() );
        channel.shutdown();

    }

}
```

Der Client baut eine Verbindung zum Server auf, erstellt eine Anfrage mit Vor- und

Nachname und ruft die Methode `hello()` des gRPC-Services auf. Die Antwort („Hello World …“) wird anschließend im Terminal ausgegeben.

```java
import io.grpc.Server;
import io.grpc.ServerBuilder;

import java.io.IOException;

public class HelloWorldServer {

    private static final int PORT = 50051;
    private Server server;

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

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

    Run | Debug
    public static void main(String[] args) throws InterruptedException, IOException {
        HelloWorldServer server = new HelloWorldServer();
        System.out.println(x: "\n\n");
        System.out.println( x: "HelloWorld Service is running!");
        System.out.println(x: "\n\n");
        server.start();
        server.blockUntilShutdown();
    }

}
```

Der Server startet auf Port 50051, registriert den gRPC-Service (`HelloWorldServiceImpl`) und bleibt aktiv, bis er beendet wird. Er wartet auf eingehende Client-Anfragen.

```java
import io.grpc.stub.StreamObserver;

public class HelloWorldServiceImpl extends HelloWorldServiceGrpc.HelloWorldServiceImplBase {

    @Override
    public void hello( Hello.HelloRequest request, StreamObserver<Hello.HelloResponse> responseObserver) {

        System.out.println("Handling hello endpoint: " + request.toString());

        String text = "Hello World, " + request.getFirstname() + " " + request.getLastname();
        Hello.HelloResponse response = Hello.HelloResponse.newBuilder().setText(text).build();

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

    }
}
```

Die konkrete Implementierung des gRPC-Services.

Hier wird definiert, was passieren soll, wenn der Client die Methode `hello()` aufruft.
Der Server erzeugt aus den Clientdaten eine Antwortnachricht und sendet sie zurück.

```
syntax = "proto3";

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

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

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

Definiert die Datenstrukturen (`HelloRequest` , `HelloResponse`) und den gRPC-Service `HelloWorldService` mit der Methode `hello()` .
Diese Datei ist die Grundlage für die automatische Codegenerierung in Java und anderen Sprachen.

Als nächstes habe ich HelloWorldServer gestartet auf den Run Button oben rechts.

```
HelloWorld Service is running!


WARNING: A terminally deprecated method in sun.misc.Unsafe has been called
```

Als nächstes habe ich Den Client gestartet und verbunden und das kam als Ausgabe:

```
WARNING: sun.misc.Unsafe::objectFieldOffset will be removed in a future release
Hello World, Max Mustermann
luca_@MacBook-Air-4 DEZSYS_GK_HELLOWORLD_GRPC %
```

Für die erweiterten Grundlagen musste ich Records erstellen diese in die Proto Datei hinzufügen, den Client angepasst und den ServiceImpl angepasst.
Nach dem starten kommt diese Ausgabe.

```
message DataRecord {
  int32 id = 1;
  string name = 2;
  double value = 3;
}


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

```
Hello.DataRecord data = Hello.DataRecord.newBuilder()
        .setId(value: 1)
        .setName(value: "Temperature Sensor A")
        .setValue(value: 21.7)
        .build();

Hello.DataResponse dataResponse = stub.sendData(data);
System.out.println( dataResponse.getMessage() );

channel.shutdown();
```

```
@Override
public void sendData(Hello.DataRecord request, StreamObserver<Hello.DataResponse> responseObserver) {

System.out.println("Received DataRecord: " + request);

String msg = "Received DataRecord with ID=" + request.getId() +
        ", Name=" + request.getName() +
        ", Value=" + request.getValue();

Hello.DataResponse response = Hello.DataResponse.newBuilder()
        .setMessage(msg)
        .build();

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

```
WARNING: sun.misc.Unsafe::objectFieldOffset will be removed in a future release
Hello World, Max Mustermann
Received DataRecord with ID=1, Name=Temperature Sensor A, Value=21.7
luca_@MacBook-Air-4 DEZSYS_GK_HELLOWORLD_GRPC %
```

Für die Vertiefung habe ich einen neuen Client erstellt in python und musste diesen mit dem Server verbinden, einen DataRecord erstellen und RPC aufrufen. Dann habe ich den Server

gestartet und mit diesem Befehl den Client ausgeführt und den DataRecord zurückbekommen.

```python
import grpc
import hello_pb2
import hello_pb2_grpc

def run():
    # Verbindung zum Java-Server
    channel = grpc.insecure_channel('localhost:50051')
    stub = hello_pb2_grpc.HelloWorldServiceStub(channel)

    # DataRecord erstellen
    record = hello_pb2.DataRecord(
        id=42,
        name="Weather Station B",
        value=18.9
    )

    # RPC aufrufen
    response = stub.sendData(record)

    print("Server response:", response.message)

if __name__ == "__main__":
    run()
```

```
luca_@MacBook-Air-4 DEZSYS_GK_HELLOWORLD_GRPC % python3 src/main/resources/data_client.py

Server response: Received DataRecord with ID=42, Name=Weather Station B, Value=18.9
luca_@MacBook-Air-4 DEZSYS_GK_HELLOWORLD_GRPC %
```