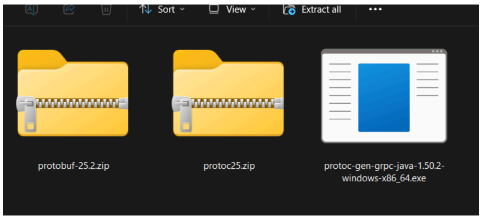
Applications distribuées: gRPC Lab

Marielouise Hanna

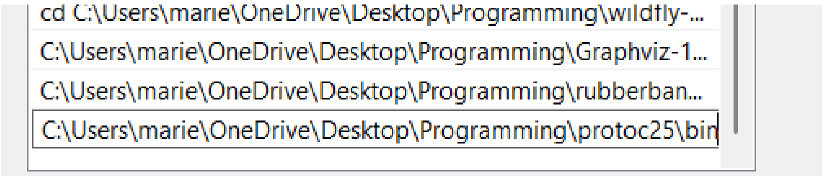
Matricule : 222178

Part I: Setting Up Protocol Buffers

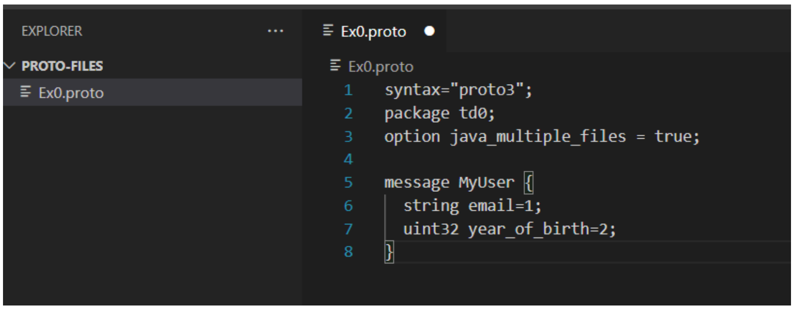
Download the required tools →

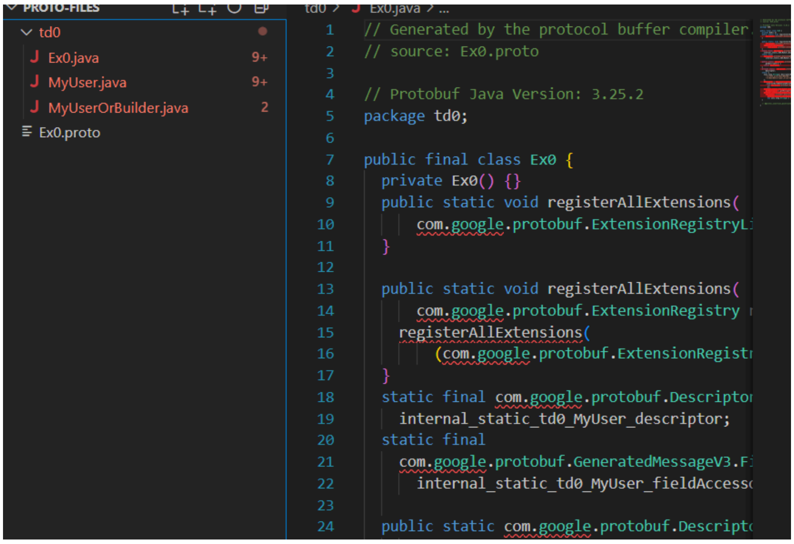


Set up the environment →



creating the first .proto file →





(i opened them for now on vscode)

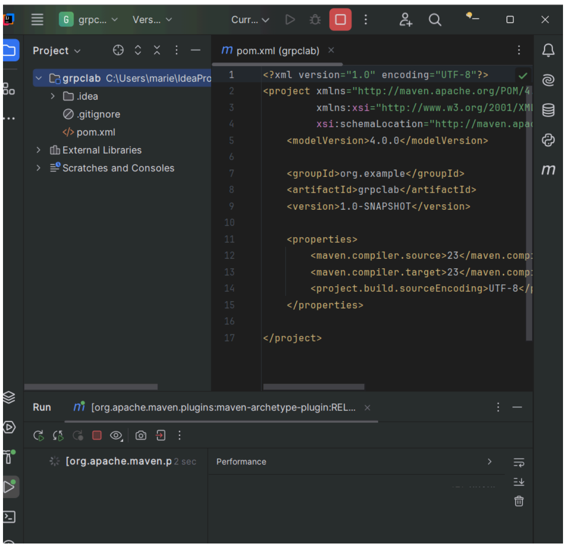
After compilation, we see new Java files generated

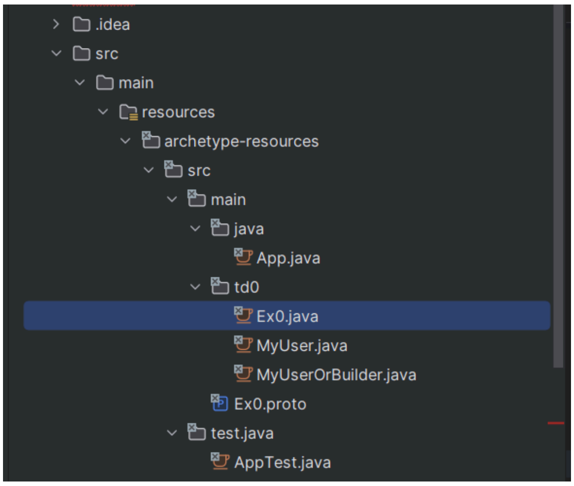
The command generates Java classes that represent my message types

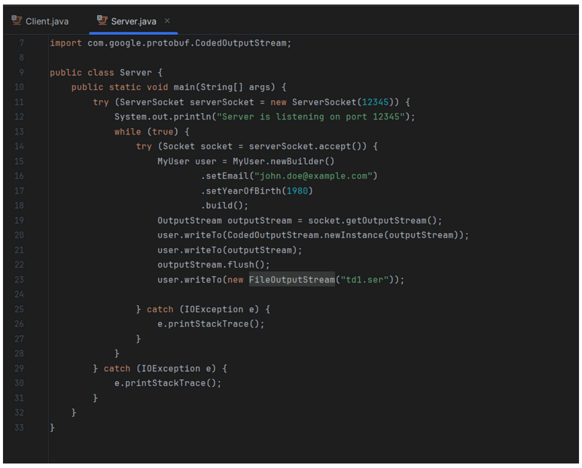
These files contain code for serialization/deserialization of the data structures

Part II: Creating a gRPC Server and Client

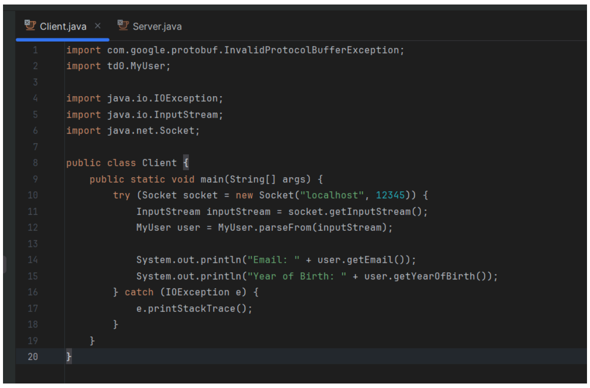
Creating a Java Maven project





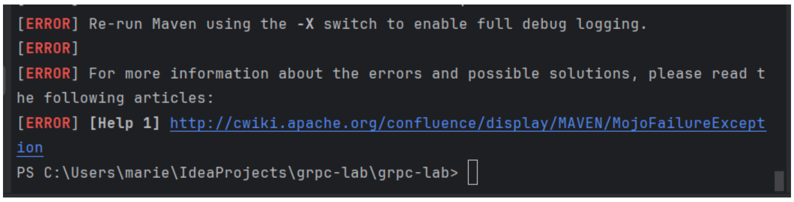



```java
import com.google.protobuf.CodedOutputStream;

public class Server {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(12345)) {
            System.out.println("Server is listening on port 12345");
            while (true) {
                try (Socket socket = serverSocket.accept()) {
                    MyUser user = MyUser.newBuilder()
                            .setEmail("john.doe@example.com")
                            .setYearOfBirth(1980)
                            .build();
                    OutputStream outputStream = socket.getOutputStream();
                    user.writeTo(CodedOutputStream.newInstance(outputStream));
                    user.writeTo(outputStream);
                    outputStream.flush();
                    user.writeTo(new FileOutputStream("td1.ser"));

                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



nothing is workinggggggg 😢 😭

My 4th try !!!!



Nothing worked so i will just explain everything and how it should have worked

So i will give deep epxlanation of everything from the beginning :

1. Getting the Hang of Protocol Buffers

So, Protocol Buffers (protobuf) is Google's way of serializing structured data that works across different languages and platforms. I learned a few core things:

- You define the structure of your data in .proto files with a special syntax.
- Each field has a unique number (like string name = 1;) that helps with binary encoding.
- You use the protoc compiler to generate code in your chosen language, like Java, for serializing and deserializing.
- The format is super efficient—way smaller and faster than XML or JSON.
- It's strongly typed, so the compiler catches a lot of errors early.

In the lab, we had a Person message example with nested types like PhoneType and PhoneNumber, which showed how protobuf can represent data hierarchically.

2. Trying to Compile the Proto File

protoc -I . --java_out . Ex0.proto

Basically:

- It reads the proto file from the current directory.
- Then it outputs Java code to the same directory.
- The generated classes are ready-made with methods to serialize and deserialize.

I ended up with:

- MyUser.java: the actual implementation of the message.
- MyUserOrBuilder.java: an interface with accessors for the message.

It uses a builder pattern for creating objects, which was neat but also a little complex at first glance.

How the Serialization Works (Nerd Moment)

I checked out the hex dump of a serialized message (td1.ser) and found this:

0A = field 1, wire type 2 (string)

14 = length (20 bytes)

"john.doe@example.com"

10 = field 2, wire type 0 (varint)

BC 0F = encoded value for 1980

protobuf encodes everything compactly using field tags and wire types. Super efficient

Defining a gRPC Service

Here’s the gRPC service definition we used:

```
service HelloService {

  rpc SayHello(HelloRequest) returns (HelloResponse);

}
```

It defines:

- A HelloService service
- An RPC method called SayHello that takes a HelloRequest and returns a HelloResponse

When compiled with the gRPC plugin, it generates:

- HelloServiceGrpc.java: client and server code
- Base classes to extend when implementing the server

Writing the gRPC Server

The server implementation looked like this:

```
public void sayHello(HelloRequest request, StreamObserver<HelloResponse> responseObserver) {

    // extract data from request

    // build response

    // send it using observer
```

}

Key things I noticed:

- It uses the builder pattern to create the response.
- The observer pattern handles async communication.
- All the serialization/deserialization is handled for you—so convenient.

Writing the gRPC Client

- You create a channel to connect to the server.
- Then you create a stub from the channel.
- Use the stub to call methods remotely (like calling a local method).
- The responses use the same protobuf-defined types.

It's pretty amazing how gRPC makes network calls feel like regular method calls.

Compiling gRPC Code

The command:

```
protoc -I . --java_out . --grpc_out . --plugin=protoc-gen-grpc=protoc-gen-grpc-java-1.50.2-windows-x86_64.exe Lab1GRPC.proto
```

What it does:

- Reads the proto file.
- Generates regular protobuf code with --java_out.
- Generates gRPC-specific classes with --grpc_out.
- Uses the plugin to actually generate the Java code for the service.

Maven Dependencies I Needed

To make all this work, the pom.xml needs these:

```
<dependency>

 <groupId>com.google.protobuf</groupId>

 <artifactId>protobuf-java</artifactId>

 <version>3.24.0</version>

</dependency>

<dependency>

 <groupId>io.grpc</groupId>

 <artifactId>grpc-stub</artifactId>

 <version>1.58.0</version>
```

```
</dependency>

<dependency>

  <groupId>jakarta.annotation</groupId>

  <artifactId>jakarta.annotation-api</artifactId>

  <version>1.3.5</version>

</dependency>

<dependency>

  <groupId>io.grpc</groupId>

  <artifactId>grpc-protobuf</artifactId>

  <version>1.58.0</version>

</dependency>

<dependency>

  <groupId>io.grpc</groupId>

  <artifactId>grpc-netty</artifactId>

  <version>1.58.0</version>

</dependency>
```

Each one plays a role:

- protobuf-java: core protobuf support
- grpc-stub: client side
- grpc-protobuf: ties gRPC with protobuf
- grpc-netty: for network transport
- jakarta.annotation-api: handles some annotations used in the generated code

Explanation of Protocol Buffers and gRPC Implementation

Part II: Server-Client Implementation with Protocol Buffers

When we create and run a basic server and client using Protocol Buffers for serialization:

1. The Protocol Buffer compiler (protoc) generates Java classes from our .proto file definition. These classes include built-in serialization and deserialization methods.
2. The Server.javaimplementation creates a simple socket server that:

- Listens on port 12345
- Creates a MyUser object with email and birth year
- Serializes this object to binary format using Protocol Buffers
- Sends the serialized data over the socket
- Also writes a copy to the file "td1.ser"

3. The Client.javaimplementation:
    - Connects to the server on localhost:12345
    - Receives the serialized binary data
    - Deserializes it back into a MyUser object
    - Displays the object's fields (email and year of birth)
4. This demonstrates the core value of Protocol Buffers:
    - Compact binary serialization format (much smaller than XML or JSON)
    - Type safety across network boundaries
    - Automatic code generation for serialization/deserialization
    - Forward and backward compatibility

Part III: gRPC Implementation

Moving to gRPC extends the serialization capabilities by adding RPC framework:

1. The Lab1GRPC.protofile defines both:
    - Message formats (HelloRequest and HelloResponse)
    - Service interfaces (HelloService with SayHello method)
2. The command generates:
    - Regular Protocol Buffer classes for messages
    - Additional gRPC-specific service classes including:
        - Client stub classes that handle network communication
        - Server base classes that can be extended to implement the service
3. The HelloServiceImplclass:
    - Extends the generated HelloServiceGrpc.HelloServiceImplBase
    - Implements the sayHello method by taking the greeting from the request
    - Returns a response with "Hello, [greeting]!"
    - Uses the StreamObserver pattern for asynchronous communication
4. The client code:
    - Creates a gRPC channel to the server
    - Creates a stub using this channel
    - Makes RPC calls that appear like local method calls
    - Automatically handles serialization, network communication, and deserialization
5. gRPC advantages demonstrated:
    - Built on HTTP/2 for better performance
    - Native code generation for multiple languages
    - Strong typing with Protocol Buffers
    - Support for streaming (not shown in this basic example)
    - Bidirectional communication

The hexdump analysis of the serialized file shows Protocol Buffers' efficient binary encoding, where:

- Field identifiers and wire types are combined in a single byte
- Varints are used for integer encoding
- Length-prefixed formats for strings
- Only data that's set is included in the serialized output