DISTRIBUTED SYSTEMS

Lab 4

João Leitão, Sérgio Duarte, Pedro Camponês

GOALS

In the end of this lab you should be able to:

- Understand how gRPC operates and how to specify protobuf messages.
- Know how to develop gRPC services and expose them on Servers in Java.
- Know how to develop gRPC clients in Java.
- Understand how to avoid repeating code for the logic of the server.
- Understand how to generalize client code that can interact both with REST and gRPC servers.

Goals

In the end of this lab you should be able to:

- **Understand how gRPC operates and how to specify protobuf messages.**
- Know how to develop gRPC services and expose them on Servers in Java.
- Know how to develop gRPC clients in Java.
- Understand how to avoid repeating code for the logic of the server.
- Understand how to generalize client code that can interact both with REST and gRPC servers.

gRPC

gRPC is a modern open-source high performance Remote Procedure Call (RPC) framework that can run in any environment.

- It relies on the specification of RPCs in textual format (fully independent of programming language)
- Each RPC can receive one message as argument and one message as return.
- Messages are also defined in textual format in protobuf (Protocol Buffers) format.

gRPC: Example of Service Specification

```
syntax = "proto3";

option java_package = "lab4.impl.server.grpc.generated_java";
option java_outer_classname = "UsersProtoBuf";

service Users {
        rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}
        rpc getUser( GetUserArgs) returns (GetUserResult){}
        rpc updateUser( UpdateUserArgs) returns (UpdateUserResult){}
        rpc deleteUser( DeleteUserArgs) returns (DeleteUserResult){}
        rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
}
```

gRPC: Example of Service Specification

```
syntax = "proto3";
```

Version mandatory.

```
option java_package = "lab4.impl.server.grpc.generated_java";
option java_outer_classname = "UsersProtoBuf";


service Users {
		rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}
		rpc getUser( GetUserArgs) returns (GetUserResult){}
		rpc updateUser( UpdateUserArgs) returns (UpdateUserResult){}
		rpc deleteUser( DeleteUserArgs) returns (DeleteUserResult){}
		rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
}
```

gRPC: Example of Service Specification

```
syntax = "proto3";

option java_package = "lab4.impl.server.grpc.generated_java";
option java_outer_classname = "UsersProtoBuf";

service Users {
		rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}
		rpc getUser( GetUserArgs) returns (GetUserResult){}
		rpc updateUser( UpdateUserArgs) returns (UpdateUserResult){}
		rpc deleteUser( DeleteUserArgs) returns (DeleteUserResult){}
		rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
}
```

The options parameterize the generation of Java code.

java_package specifies the package in your source code where the generated java code will be placed
java_outer_classname specified the name of the outer Class that will be generated (more internal classes are generated)

gRPC: Example of Service Specification

```
syntax = "proto3";

option java_package = "lab4.impl.server.grpc.generated_java";
option java_outer_classname = "UsersProtoBuf";
```

RPCs are specified within the context of a service, in this example named Users.

```
service Users {
        rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}
        rpc getUser( GetUserArgs) returns (GetUserResult){}
        rpc updateUser( UpdateUserArgs) returns (UpdateUserResult){}
        rpc deleteUser( DeleteUserArgs) returns (DeleteUserResult){}
        rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
}
```

gRPC: Example of Service Specification

```
syntax = "proto3";


option java_package = "lab4.impl.server.grpc.generated_java";

option java_outer_classname = "UsersProtoBuf";


service Users {
        rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}
        rpc getUser( GetUserArgs) returns (GetUserResult){}
        rpc updateUser( UpdateUserArgs) returns (UpdateUserResult){}
        rpc deleteUser( DeleteUserArgs) returns (DeleteUserResult){}
        rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
}
```

An RPC is defined within a service by the keyword **rpc** followed by the *name* of the rpc, the *argument message* (at most one) the keyword **returns**, and the *return message* (at most one type)

gRPC: Example of Message Specification

rpc createUser(CreateUserArgs) returns (CreateUserResult) {}

```
message GrpcUser {
        optional string userId = 1;
        optional string email = 2;
        optional string fullName = 3;
        optional string password = 4;
        optional string avatar = 5;
}

message CreateUserArgs {
        GrpcUser user = 1;
}

message CreateUserResult {
        string userId = 1;
}
```

gRPC: Example of Message Specification

rpc createUser(CreateUserArgs) returns (CreateUserResult) {}

```
message GrpcUser {
        optional string userId = 1;
        optional string email = 2;
        optional string fullName = 3;
        optional string password = 4;
        optional string avatar = 5;
}
```

This defines one message named GrpcUser (that implicitly encodes the format of the java class User).

```
message CreateUserArgs {
        GrpcUser user = 1;
}

message CreateUserResult {
        string userId = 1;
}
```

gRPC: Example of Message Specification

rpc createUser(CreateUserArgs) returns (CreateUserResult) {}

```
message GrpcUser {
        optional string userId = 1;
        optional string email = 2;
        optional string fullName = 3;
        optional string password = 4;
        optional string avatar = 5;
}

message CreateUserArgs {
        GrpcUser user = 1;
}

message CreateUserResult {
        string userId = 1;
}
```

Messages in protobuf are defined as a set of elements (that can be optional) that have a type, a name, and a unique numerical identifier.

The numerical identifier must be unique within each message, and it is used to identify the filed in the serialized form of the message (which avoids to send the name of the field in the message)

gRPC: Example of Message Specification

rpc createUser(CreateUserArgs) returns (CreateUserResult) {}

```
message GrpcUser {
        optional string userId = 1;
        optional string email = 2;
        optional string fullName = 3;
        optional string password = 4;
        optional string avatar = 5;
}

message CreateUserArgs {
        GrpcUser user = 1;
}

message CreateUserResult {
        string userId = 1;
}
```

Messages in protobuf are defined as a set of elements (that can be optional) that have a **type**, a name, and a unique numerical identifier.

Supported types in protobuf include:
string
int32 – integer
int64 – float
bytes – byte[]
bool

And other numerical types:
https://protobuf.dev/programming-guides/proto3/

gRPC: Example of Message Specification

```
rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}

message GrpcUser {
        optional string userId = 1;
        optional string email = 2;
        optional string fullName = 3;
        optional string password = 4;
        optional string avatar = 5;
}

message CreateUserArgs {
        GrpcUser user = 1;
}

message CreateUserResult {
        string userId = 1;
}
```

Messages can include other Messages as one of the fields.

In this case the CreateUserArgs message is composed of a single field named user whose type is the GrpcUser message.

gRPC: Example of Message Specification

rpc createUser(CreateUserArgs) returns (CreateUserResult) }

```
message GrpcUser {
        optional string userId = 1;
        optional string email = 2;
        optional string fullName = 3;
        optional string password = 4;
        optional string avatar = 5;
}

message CreateUserArgs {
        GrpcUser user = 1;
}

message CreateUserResult {
        string userId = 1;
}
```

Each message defined as an argument or return of an RPC must be specified.

The same message can be used in more than one RPC.

For (java) code readability it's a good idea to have messages that are used as arguments for an RPC to have a distinctive name (e.g., terminated in Args) and the same for messages that are used as Responses (e.g., terminated with Result)

gRPC: Example of Service Specification

```
syntax = "proto3";

option java_package = "lab4.impl.server.grpc.generated_java";
option java_outer_classname = "UsersProtoBuf";

service Users {
        rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}
        rpc getUser( GetUserArgs) returns (GetUserResult){}
        rpc updateUser( UpdateUserArgs) returns (UpdateUserResult){}
        rpc deleteUser( DeleteUserArgs) returns (DeleteUserResult){}
        rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
}
```

An RPC argument or return can be parameterized with the stream keyword.

This allows respectively for an RPC to consume multiple instances of the argument message and to generate multiple instances of the return message

gRPC: Example of Service Specification

An RPC argument or return can be parameterized with the stream keyword.

This allows respectively for an RPC to consume multiple instances of the argument message and to generate multiple instances of the return message

```
syntax = "proto3";


option java_package = "lab4.impl.server.grpc.generated_java";

option java_outer_classname = "UsersProtoBuf";


service Users {
        rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}
        rpc getUser( GetUserArgs) returns (GetUserResult){}
        rpc updateUser( UpdateUserArgs) returns (UpdateUserResult){}
        rpc deleteUser( DeleteUserArgs) returns (DeleteUserResult){}
        rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
}
```

In this particular case this implies that the RPC searchUsers can generate zero or more instances of the GrpcUser messages as a reply to a single invocation.

gRPC: Example of Service Specification

```
syntax = "proto3";

option java_package = "lab4.impl.server.grpc.generated_java";
option java_outer_classname = "UsersProtoBuf";

service Users {
        rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}
        rpc getUser( GetUserArgs) returns (GetUserResult){}
        rpc updateUser( UpdateUserArgs) returns (UpdateUserResult){}
        rpc deleteUser( DeleteUserArgs) returns (DeleteUserResult){}
        rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
}
```

An RPC argument or return can be parameterized with the stream keyword.

This allows respectively for an RPC to consume multiple instances of the argument message and to generate multiple instances of the return message

In this particular case this implies that the RPC searchUsers can generate zero or more instances of the GrpcUser messages as a reply to a single invocation.

There is another way for a message to carry multiple values within a field

gRPC: Example of Message Specification

Example from the Project

```
message GetPostsResult {
        repeated string postId = 1;
}
```

gRPC: Example of Message Specification

Example from the Project

```
message GetPostsResult {
        repeated string postId = 1;
}
```

A field can be **repeated**.

This means that the message can store multiple entries of this element (or zero).

gRPC: GENERATING CODE FROM SPECIFICATION

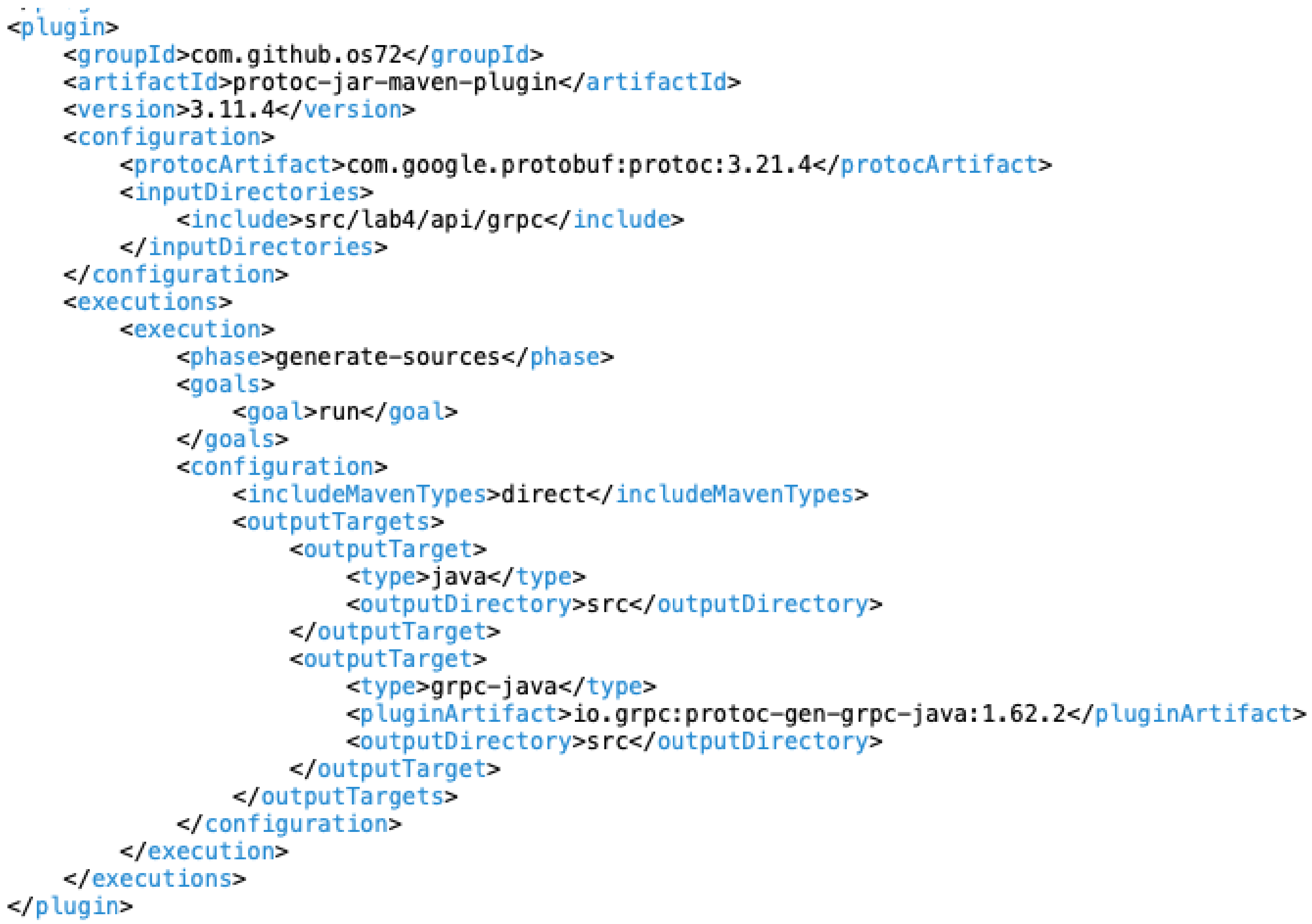
In general, the protoc tool can be used to generate code for a specific language.

Languages supported by gRPC include: C/C++, C#, Dart, Go, **Java**, Kotlin, Node.js, Objective-C, PHP, Python, Ruby

However, we can also use a maven plugin that will, as part of the compilation process, process *.proto files and generate Java code.

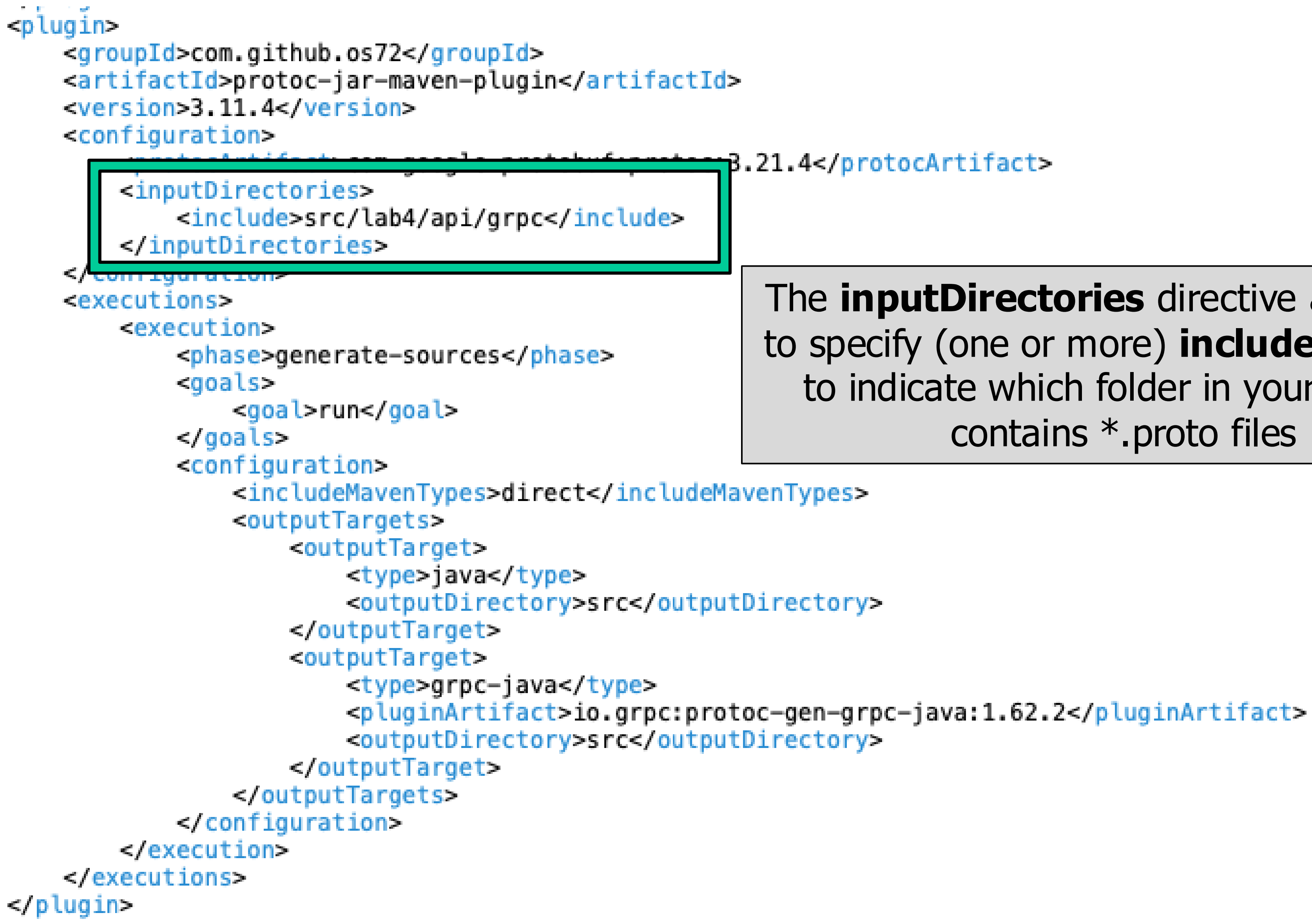
Generated code includes: client and server stubs, as well as classes for each message (argument or return) specified in the service.

gRPC: Maven plugin



```xml
<plugin>
    <groupId>com.github.os72</groupId>
    <artifactId>protoc-jar-maven-plugin</artifactId>
    <version>3.11.4</version>
    <configuration>
        <protocArtifact>com.google.protobuf:protoc:3.21.4</protocArtifact>
        <inputDirectories>
            <include>src/lab4/api/grpc</include>
        </inputDirectories>
    </configuration>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>run</goal>
            </goals>
            <configuration>
                <includeMavenTypes>direct</includeMavenTypes>
                <outputTargets>
                    <outputTarget>
                        <type>java</type>
                        <outputDirectory>src</outputDirectory>
                    </outputTarget>
                    <outputTarget>
                        <type>grpc-java</type>
                        <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.62.2</pluginArtifact>
                        <outputDirectory>src</outputDirectory>
                    </outputTarget>
                </outputTargets>
            </configuration>
        </execution>
    </executions>
</plugin>
```

gRPC: Maven plugin



```
<plugin>
    <groupId>com.github.os72</groupId>
    <artifactId>protoc-jar-maven-plugin</artifactId>
    <version>3.11.4</version>
    <configuration>
        <protocArtifact>com.google.protobuf:protoc:3.21.4</protocArtifact>
        <inputDirectories>
            <include>src/lab4/api/grpc</include>
        </inputDirectories>
    </configuration>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>run</goal>
            </goals>
            <configuration>
                <includeMavenTypes>direct</includeMavenTypes>
                <outputTargets>
                    <outputTarget>
                        <type>java</type>
                        <outputDirectory>src</outputDirectory>
                    </outputTarget>
                    <outputTarget>
                        <type>grpc-java</type>
                        <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.62.2</pluginArtifact>
                        <outputDirectory>src</outputDirectory>
                    </outputTarget>
                </outputTargets>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The **inputDirectories** directive allows you to specify (one or more) **include** directives to indicate which folder in your project contains *.proto files

gRPC: Maven plugin

```xml
<plugin>
    <groupId>com.github.os72</groupId>
    <artifactId>protoc-jar-maven-plugin</artifactId>
    <version>3.11.4</version>
    <configuration>
        <protocArtifact>com.google.protobuf:protoc:3.21.4</protocArtifact>
        <inputDirectories>
            <include>src/lab4/api/grpc</include>
        </inputDirectories>
    </configuration>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>run</goal>
            </goals>
            <configuration>
                <includeMavenTypes>direct</includeMavenTypes>
                <outputTargets>
                    <outputTarget>
                        <type>java</type>
                        <outputDirectory>src</outputDirectory>
                    </outputTarget>
                    <outputTarget>
                        <type>grpc-java</type>
                        <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.62.2</pluginArtifact>
                        <outputDirectory>src</outputDirectory>
                    </outputTarget>
                </outputTargets>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This defines the language that should be generated as well as the root of the source code in your project.

gRPC: GENERATING CODE FROM SPECIFICATION

The combination of the configuration in the pom.xml and the options in the proto file

```
option java_package = "lab4.impl.server.grpc.generated_java";
option java_outer_classname = "UsersProtoBuf";
```

Will generate code in the class:

```
lab4.impl.server.grpc.generated_java.UsersProto
```

Inner classes within this one will be created for all messages and the stubs.

GOALS

In the end of this lab you should be able to:

- Understand how gRPC operates and how to specify protobuf messages.
- **Know how to develop gRPC services and expose them on Servers in Java.**
- Know how to develop gRPC clients in Java.
- Understand how to avoid repeating code for the logic of the server.
- Understand how to generalize client code that can interact both with REST and gRPC servers.

FIRST A DETOUR

In the end of this lab you should be able to:

- Understand how gRPC operates and how to specify protobuf messages.
- **Know how to develop gRPC services and expose them on Servers in Java.**
- Know how to develop gRPC clients in Java.
- Understand how to avoid repeating code for the logic of the server.
- Understand how to generalize client code that can interact both with REST and gRPC servers.

Generalising the Logic of the Server

In the project (and in general) you might want to provide the same service through different (but equivalent) interfaces…

In this case we are considering the case of providing the functionality of the Users service both through REST and gRPC.

As you can imagine the logic for this service is extremely similar. We should avoid to have to repeat the logic of this service as to avoid imprecisions and different errors across different service implementations and to avoid repeating code that would be harder to maintain.

The lab4 project already shows to you how to achieve this.

GENERALISING THE LOGIC OF THE SERVER

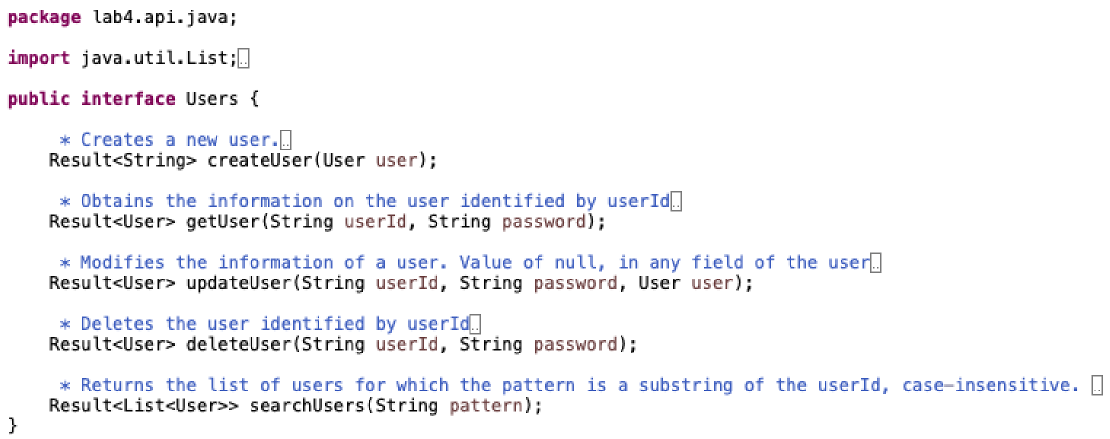
Due to this the project features a somewhat different structure than you can expect.

Interface
lab4.api.java.Users

GENERALISING THE LOGIC OF THE SERVER

Due to this the project features a somewhat different structure than you can expect.

Interface
lab4.api.java.Users

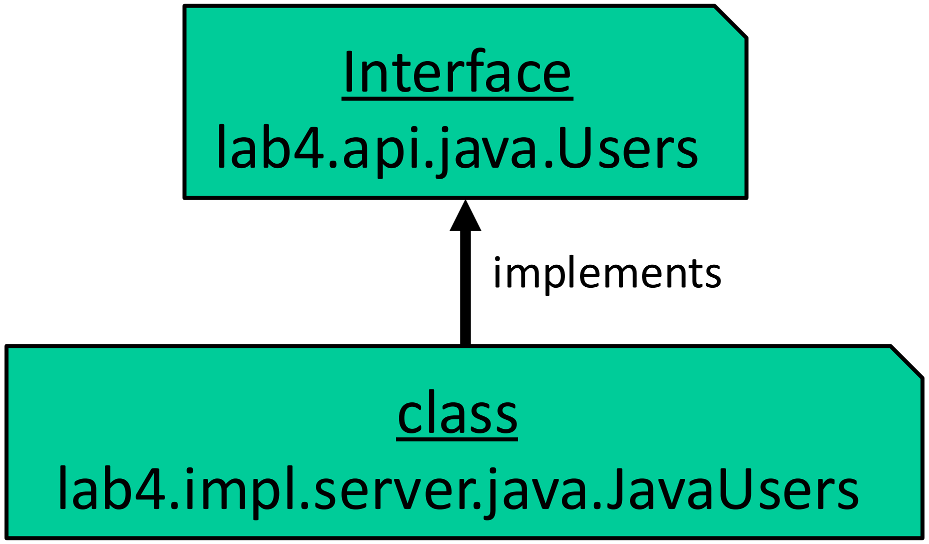


```
package lab4.api.java;

import java.util.List;

public interface Users {

     * Creates a new user.
    Result<String> createUser(User user);

     * Obtains the information on the user identified by userId
    Result<User> getUser(String userId, String password);

     * Modifies the information of a user. Value of null, in any field of the user
    Result<User> updateUser(String userId, String password, User user);

     * Deletes the user identified by userId
    Result<User> deleteUser(String userId, String password);

     * Returns the list of users for which the pattern is a substring of the userId, case-insensitive.
    Result<List<User>> searchUsers(String pattern);
}
```

Which have simple java methods (no REST annotations) that always return the Result class (parameterized with the type returned by the method in case of success). This allows to generalize error handling between REST implementations and gRPC implementations.

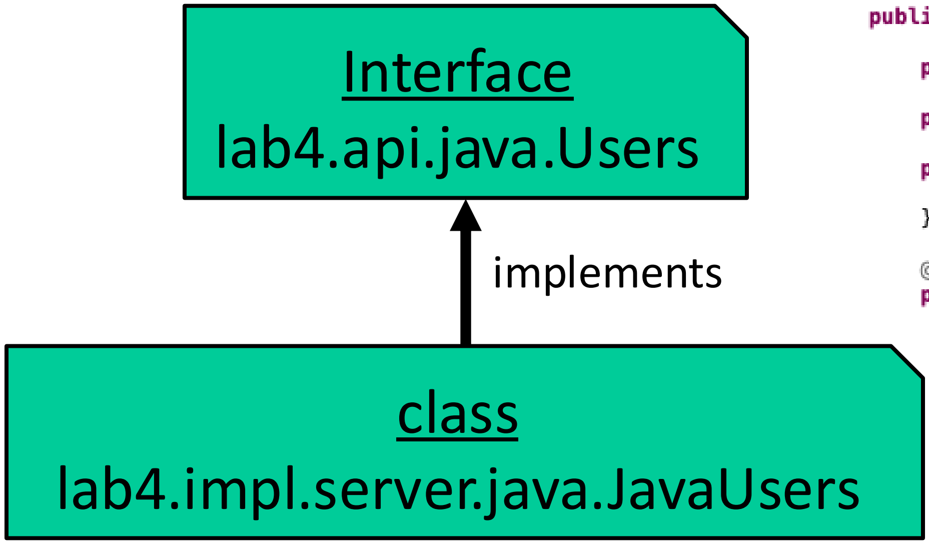
GENERALISING THE LOGIC OF THE SERVER

Due to this the project features a somewhat different structure than you can expect.

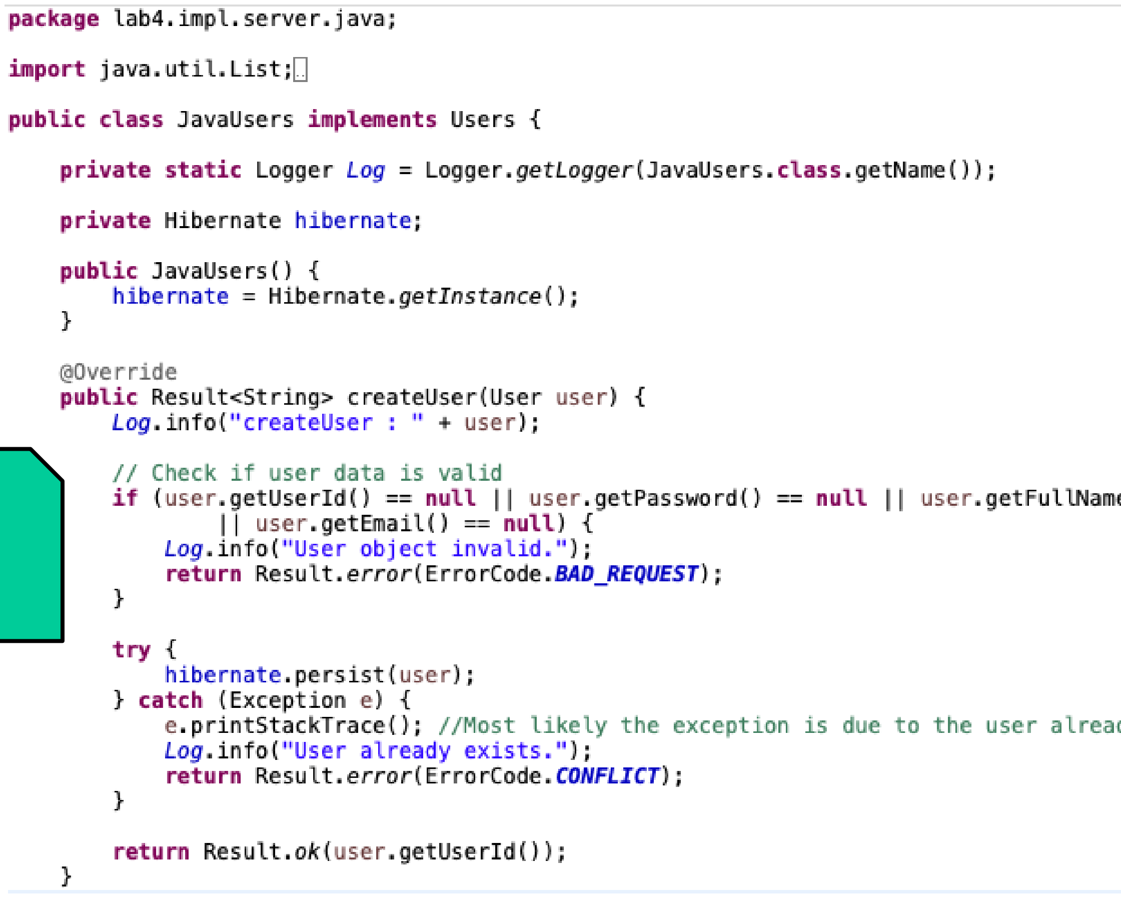


GENERALISING THE LOGIC OF THE SERVER

Due to this the project features a somewhat different structure than you can expect.



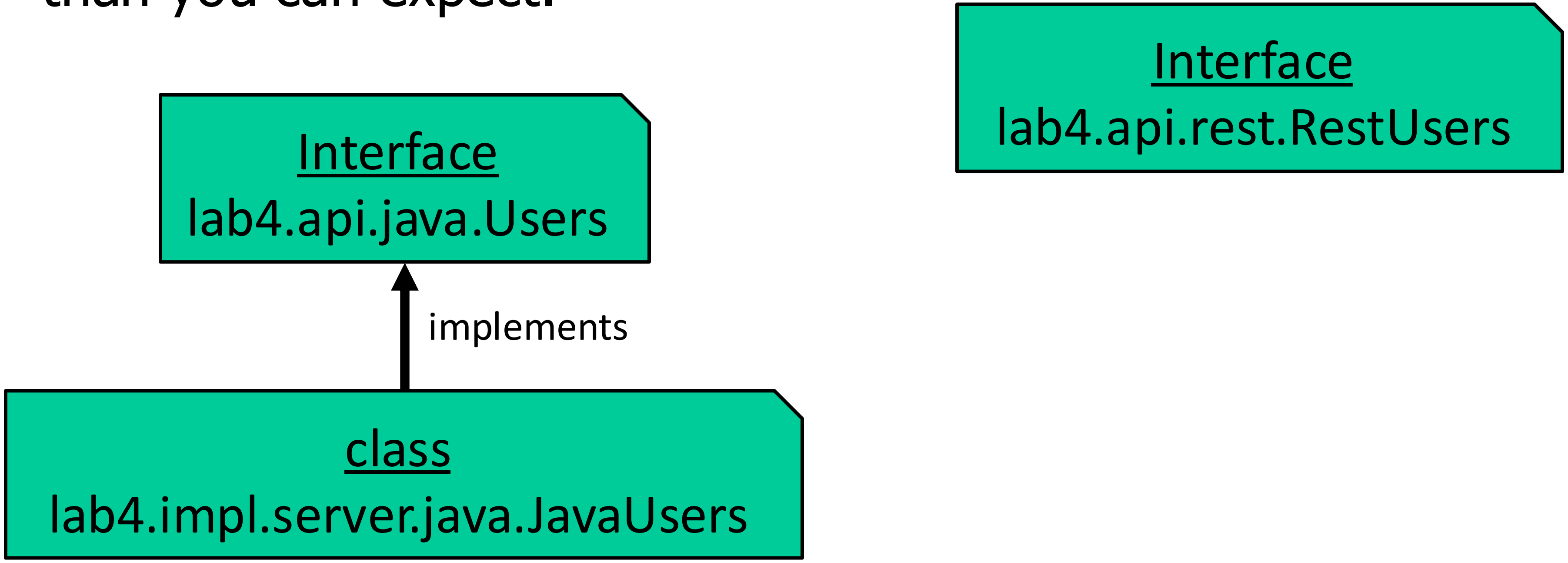
Which allows me to have an implementation that is agnostic to the way that I expose the service.



```
package lab4.impl.server.java;

import java.util.List;

public class JavaUsers implements Users {

    private static Logger Log = Logger.getLogger(JavaUsers.class.getName());

    private Hibernate hibernate;

    public JavaUsers() {
        hibernate = Hibernate.getInstance();
    }

    @Override
    public Result<String> createUser(User user) {
        Log.info("createUser : " + user);

        // Check if user data is valid
        if (user.getUserId() == null || user.getPassword() == null || user.getFullNam
                || user.getEmail() == null) {
            Log.info("User object invalid.");
            return Result.error(ErrorCode.BAD_REQUEST);
        }

        try {
            hibernate.persist(user);
        } catch (Exception e) {
            e.printStackTrace(); //Most likely the exception is due to the user alrea
            Log.info("User already exists.");
            return Result.error(ErrorCode.CONFLICT);
        }

        return Result.ok(user.getUserId());
    }
```

Generalising the Logic of the Server

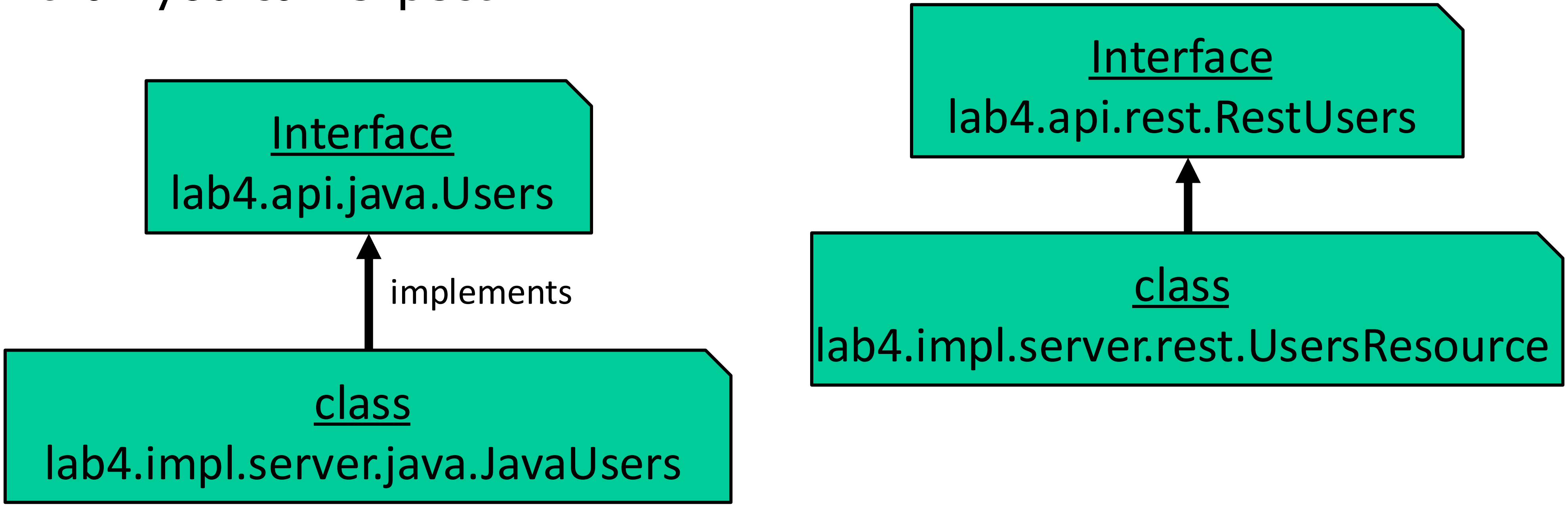
Due to this the project features a somewhat different structure than you can expect.



Then I can have the interface that specifies the REST service (does not return Result) that contains REST annotations.

GENERALISING THE LOGIC OF THE SERVER

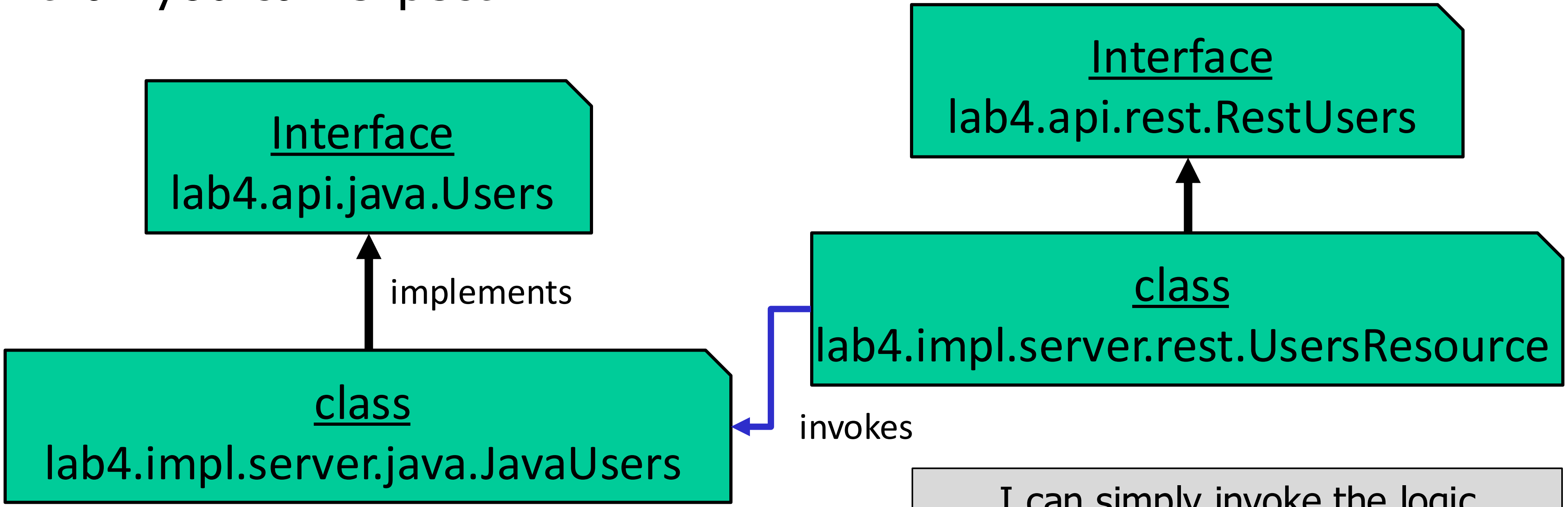
Due to this the project features a somewhat different structure than you can expect.



Instead of having the implementation of this class materializing the logic of the service in its own code.

GENERALISING THE LOGIC OF THE SERVER

Due to this the project features a somewhat different structure than you can expect.

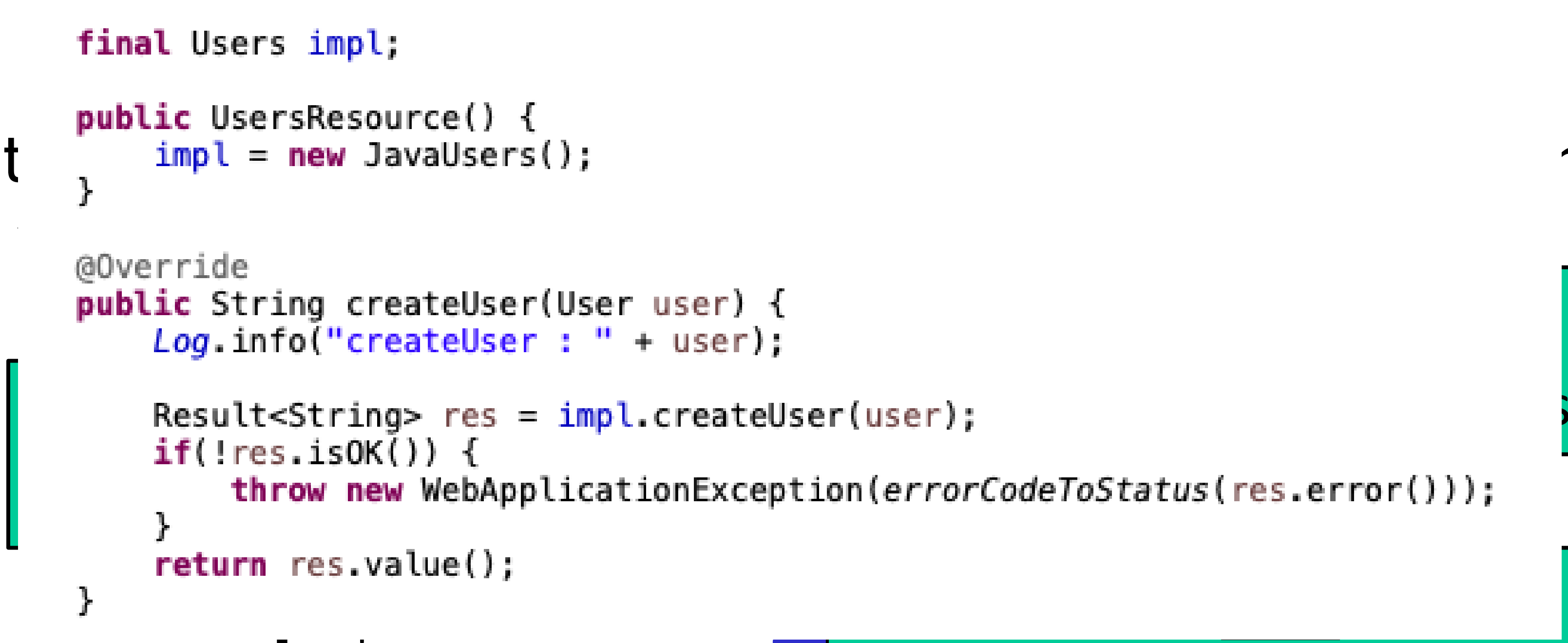


I can simply invoke the logic implemented in the generic implementation and translate error codes to corresponding REST error codes.

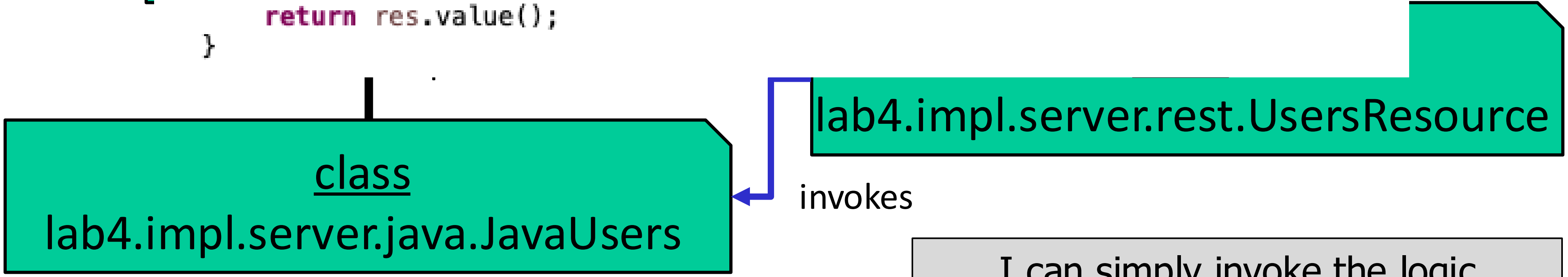
GENERALISING THE LOGIC OF THE SERVER

Due t … re

than



```java
final Users impl;

public UsersResource() {
    impl = new JavaUsers();
}

@Override
public String createUser(User user) {
    Log.info("createUser : " + user);

    Result<String> res = impl.createUser(user);
    if(!res.isOK()) {
        throw new WebApplicationException(errorCodeToStatus(res.error()));
    }
    return res.value();
}
```

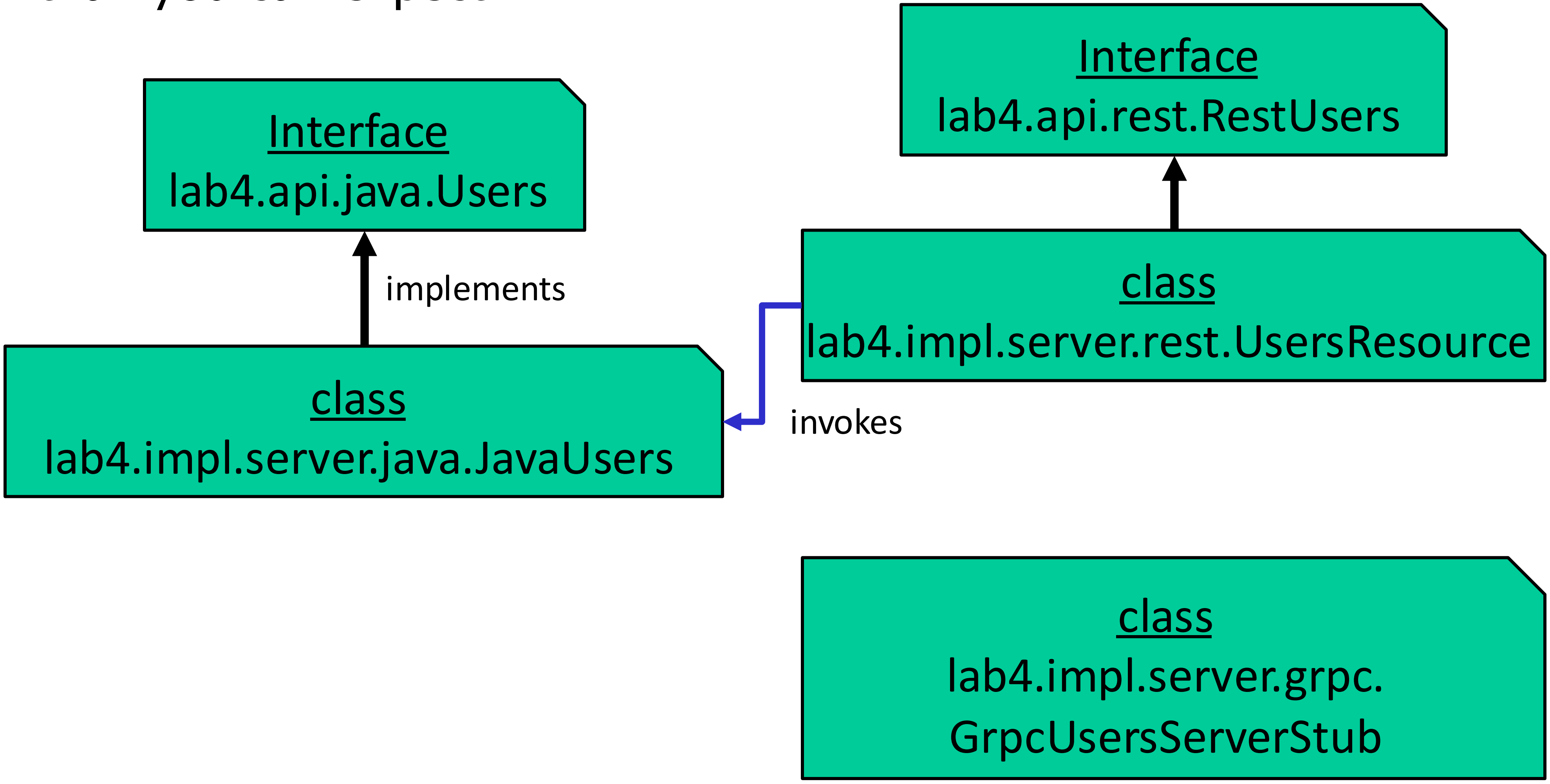


lab4.impl.server.rest.UsersResource
invokes
class
lab4.impl.server.java.JavaUsers

I can simply invoke the logic implemented in the generic implementation and translate error codes to corresponding REST error codes.

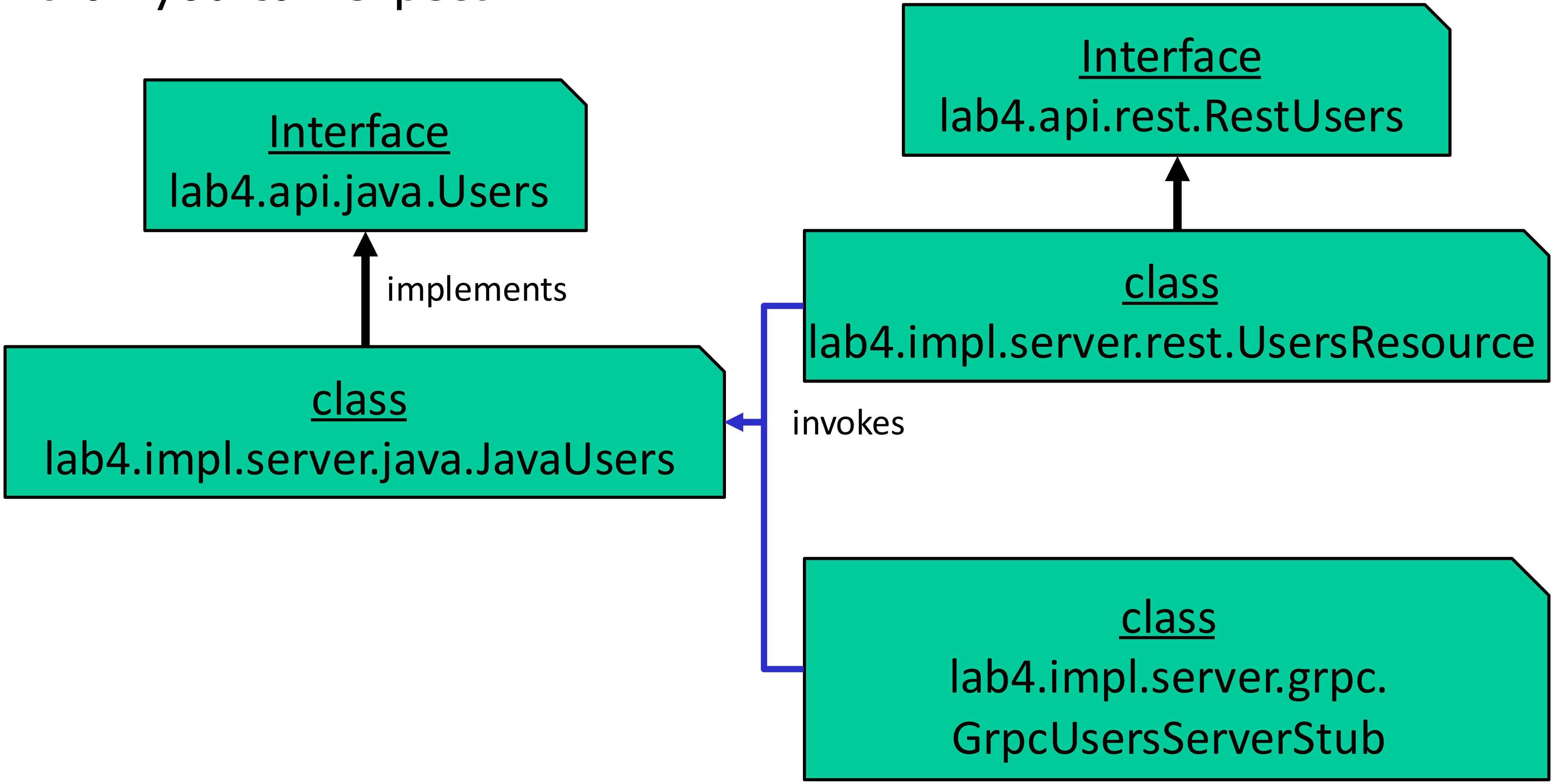
GENERALISING THE LOGIC OF THE SERVER

Due to this the project features a somewhat different structure than you can expect.



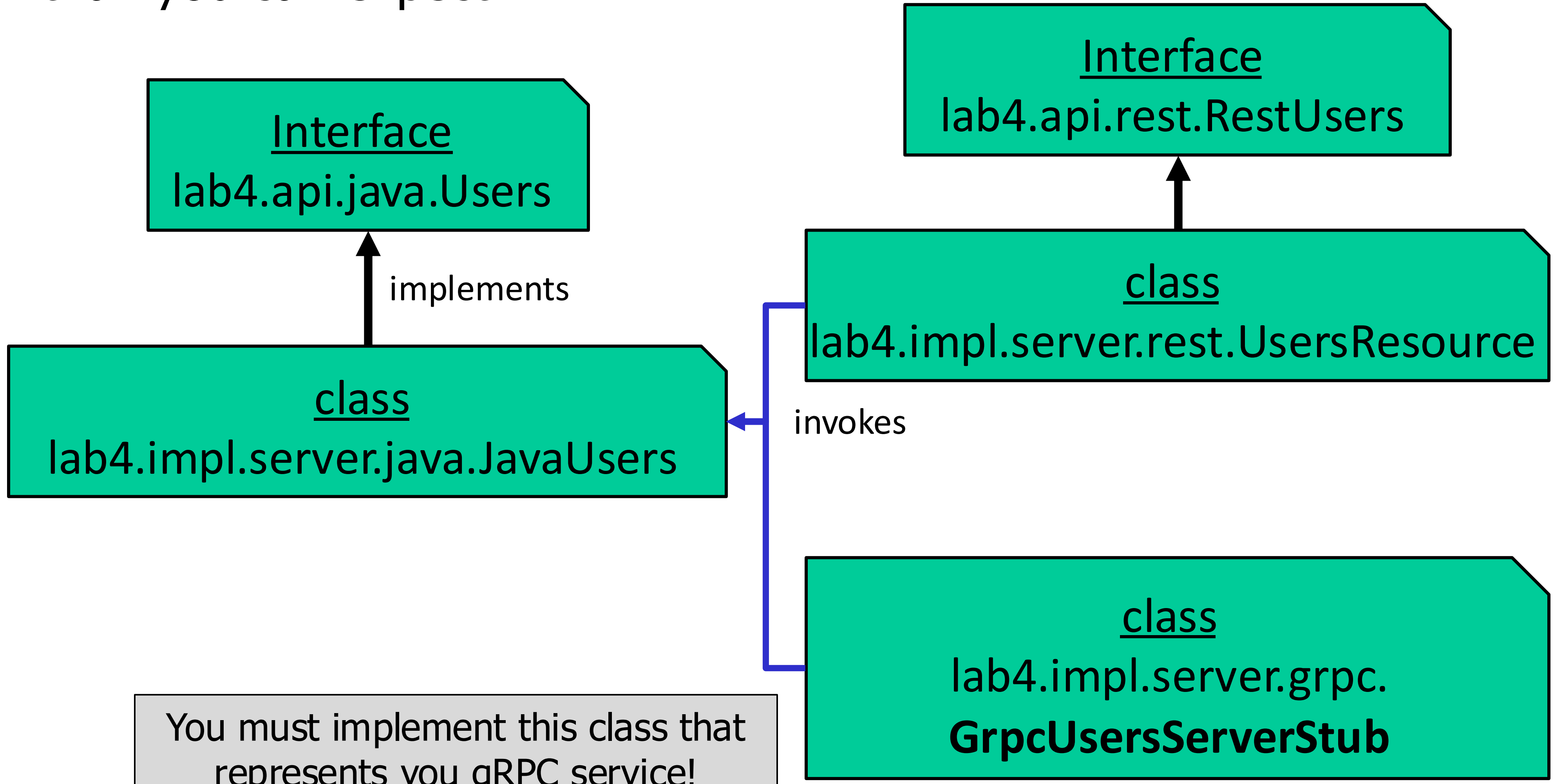
GENERALISING THE LOGIC OF THE SERVER

Due to this the project features a somewhat different structure than you can expect.



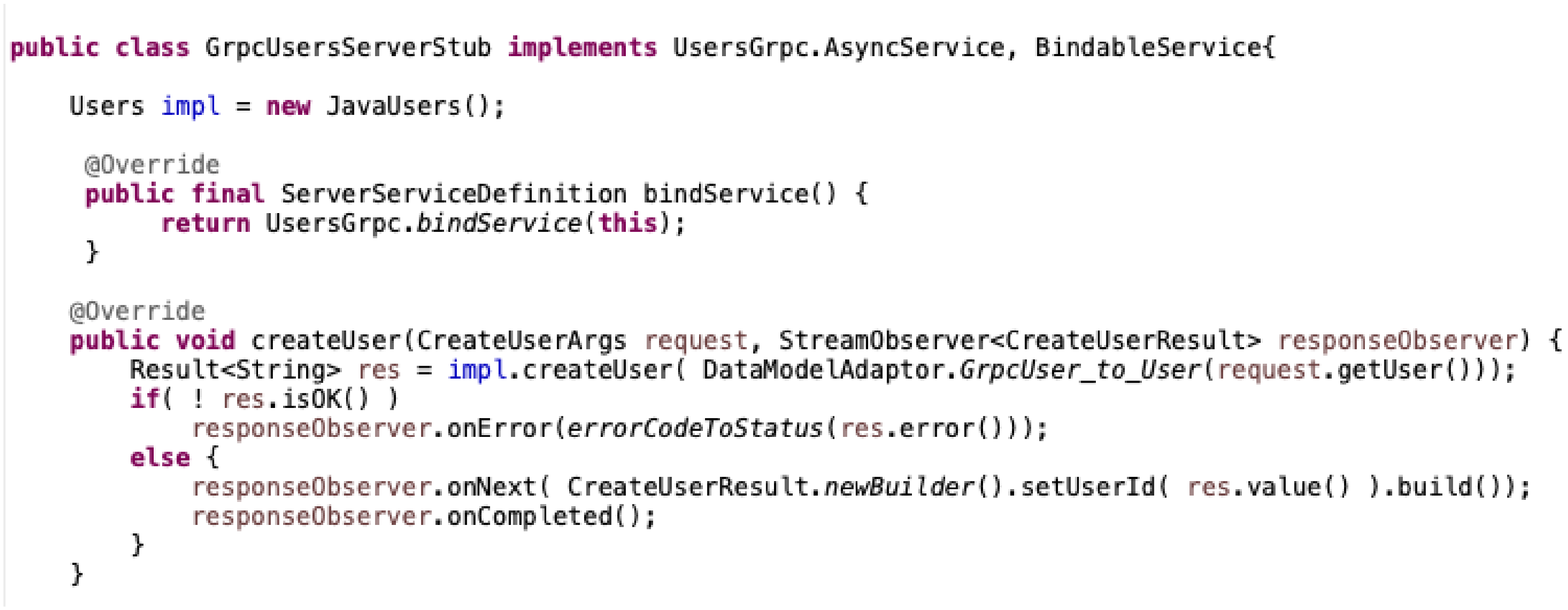
Generalising the logic of the Server

Due to this the project features a somewhat different structure than you can expect.



You must implement this class that represents you gRPC service!

The GrpcUsersServerStub



```java
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

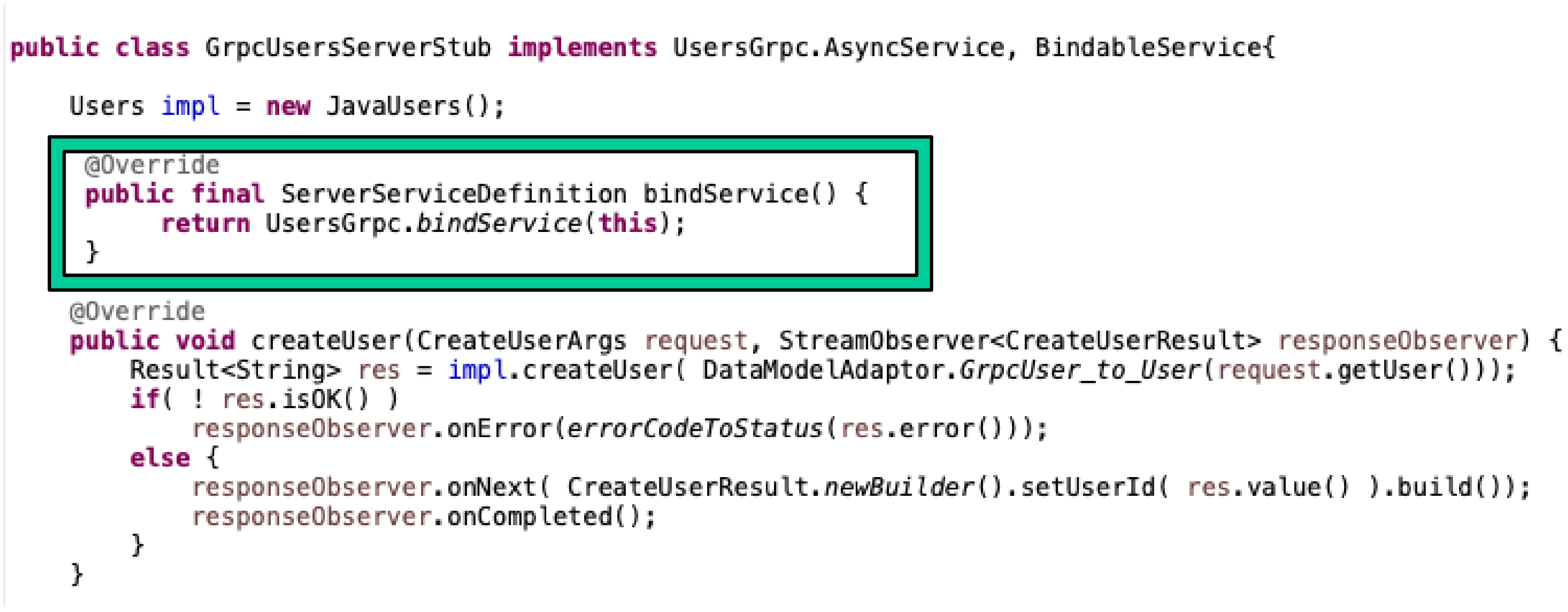
THE GRPCUSERSSERVERSTUB

You can name the class whatever you want, but it must implement the AsyncService inner interface of the Server Stub (UsersGrpc) and the BindableService interface.

```
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

THE GRPCUSERSSERVERSTUB

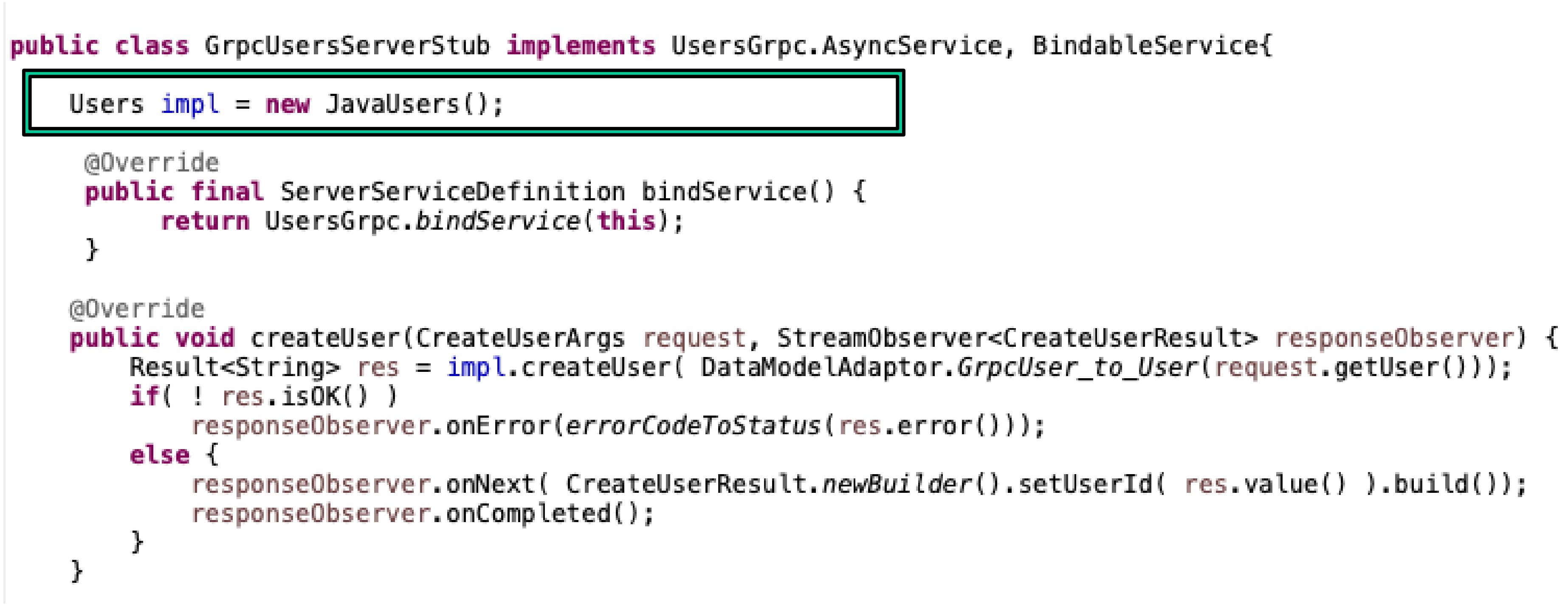
The constructor of this class must bind this instance in the Server Stub.



```java
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

The GrpcUsersServerStub

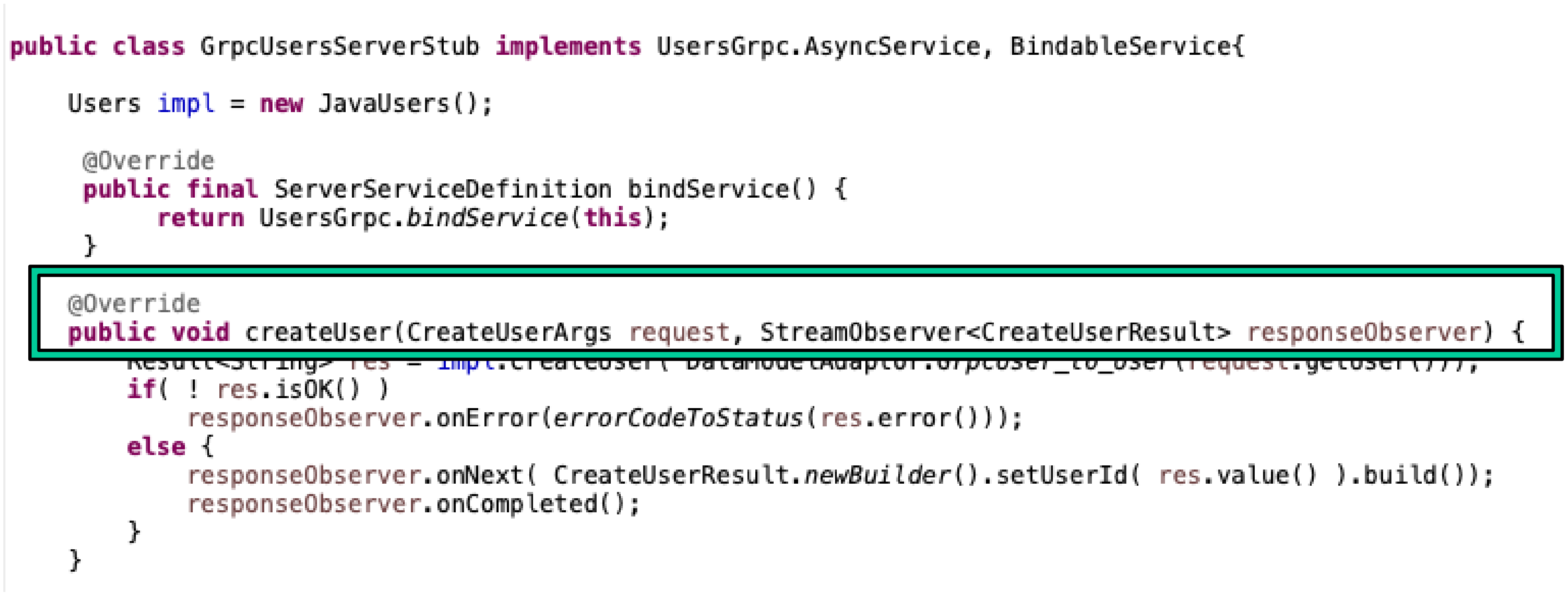
In this example we are using the generic implementation of the Users Service, that we will call to execute the logic of the service (including manipulating the database)



```
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

The GrpcUsersServerStub

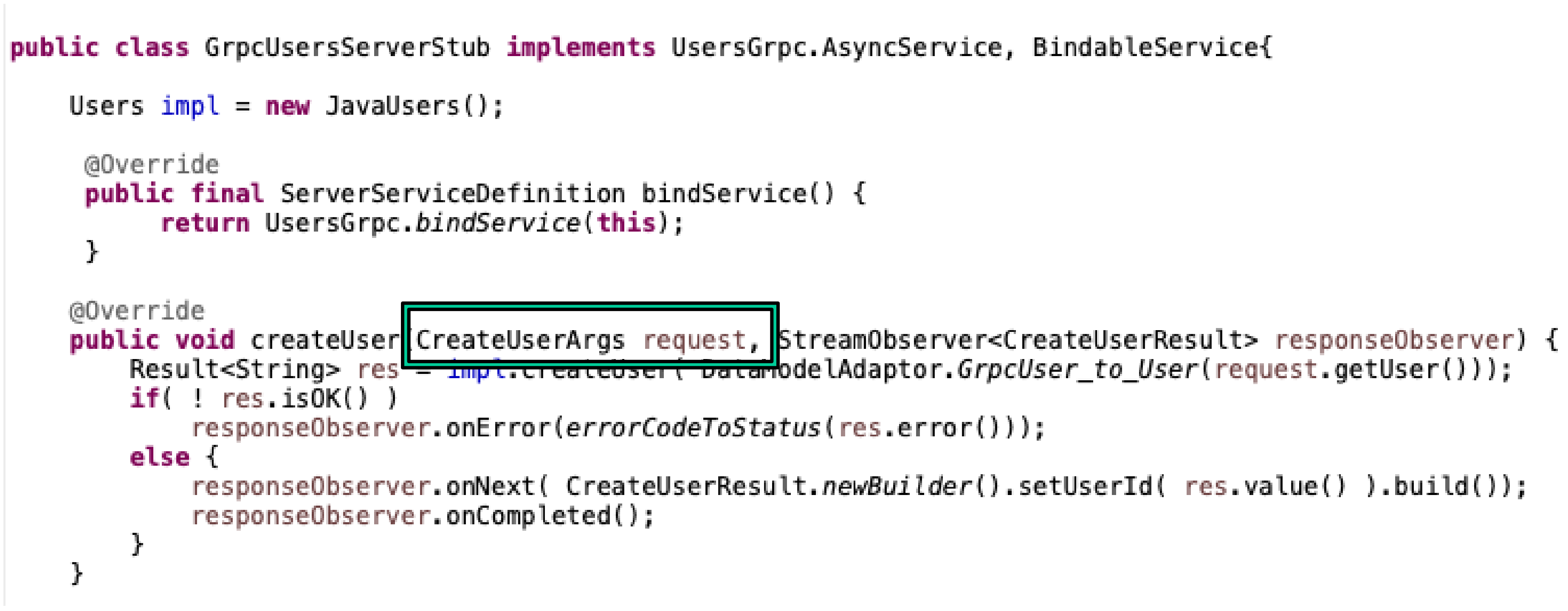
You must implement each of the functions that represents the each of the RPC made available by this service (here we have the example of the createUser RPC)



```
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

THE GRPCUSERSSERVERSTUB

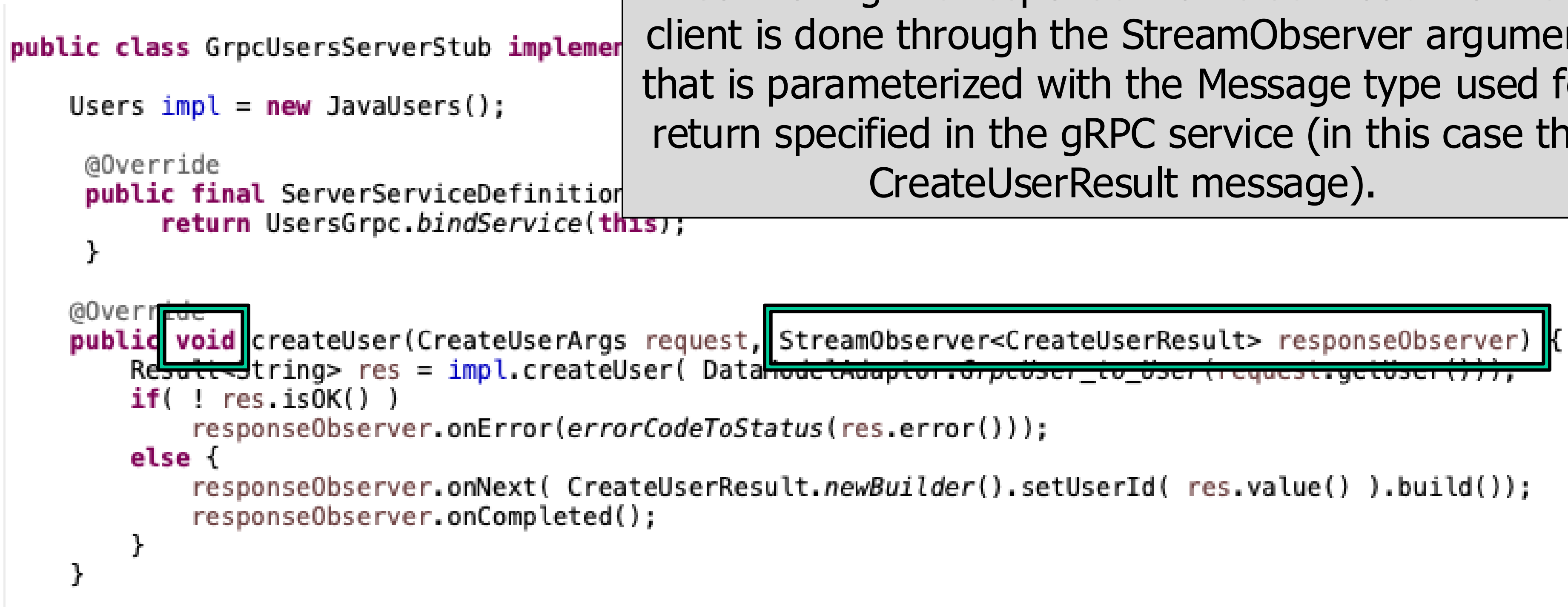
The first argument will be the generated java class that encodes the Message that contains the arguments for this RPC.



```java
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

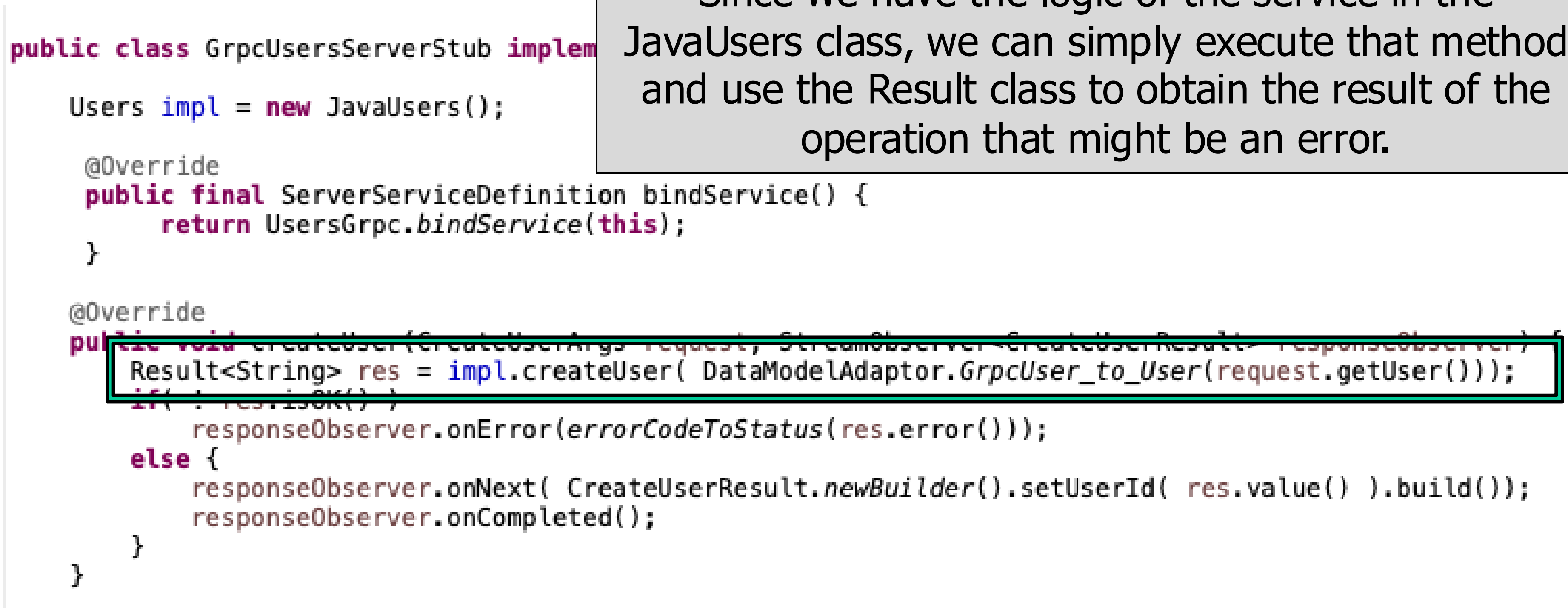
THE GRPCUSERSSERVERSTUB

The methods that represent the server logic of the RPC always return void.
Controlling the response that is sent back to the client is done through the StreamObserver argument that is parameterized with the Message type used for return specified in the gRPC service (in this case the CreateUserResult message).



```
public class GrpcUsersServerStub implemen

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinitior
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

THE GRPCUSERSSERVERSTUB

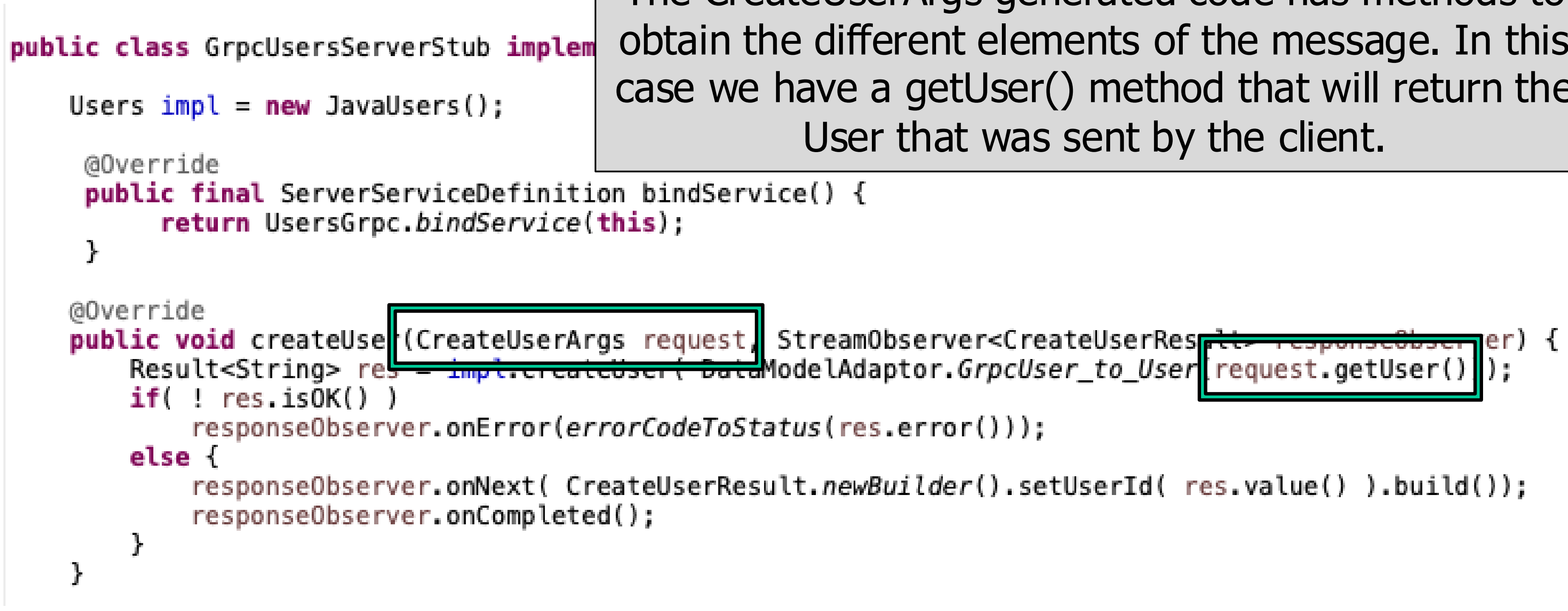


```
public class GrpcUsersServerStub implem

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
       Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
       if( ! res.isOK() )
           responseObserver.onError(errorCodeToStatus(res.error()));
       else {
           responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
           responseObserver.onCompleted();
       }
    }
```

Since we have the logic of the service in the JavaUsers class, we can simply execute that method and use the Result class to obtain the result of the operation that might be an error.

THE GRPCUSERSSERVERSTUB

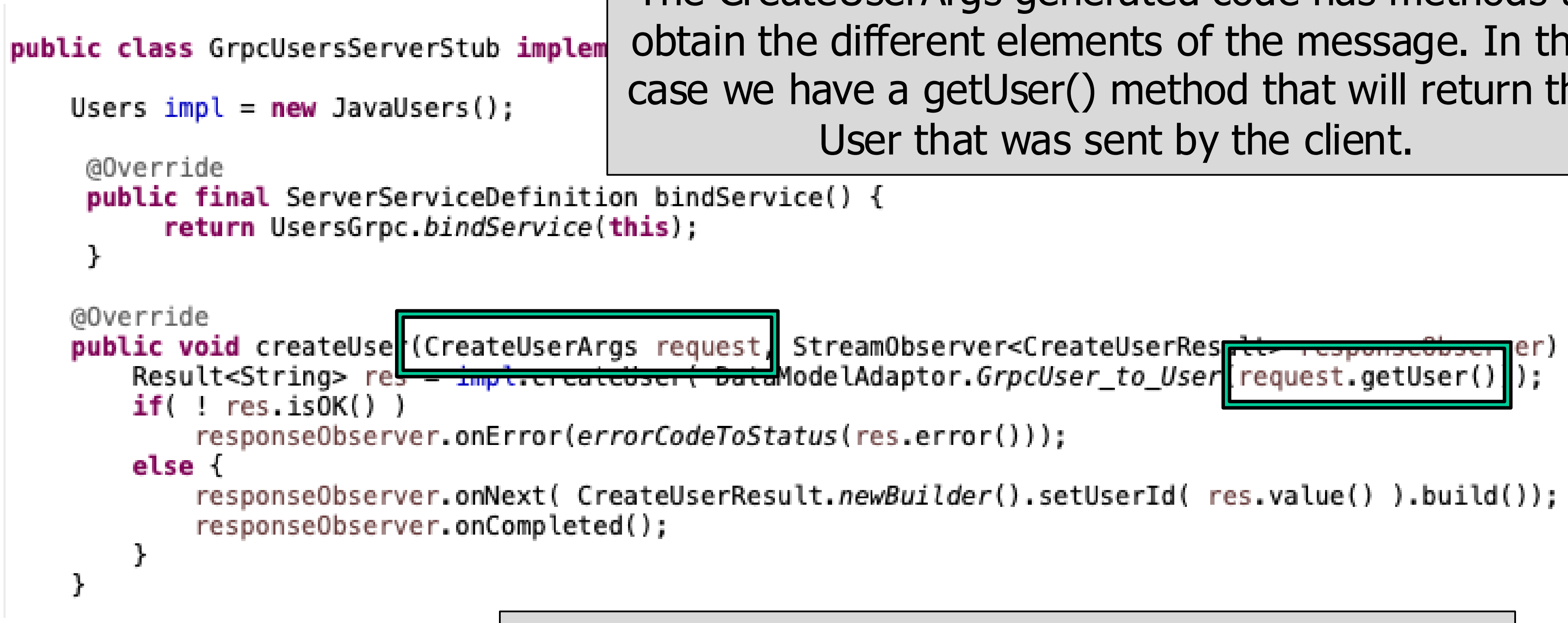
The CreateUserArgs generated code has methods to obtain the different elements of the message. In this case we have a getUser() method that will return the User that was sent by the client.



```java
public class GrpcUsersServerStub implem

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

THE GRPCUSERSSERVERSTUB

The CreateUserArgs generated code has methods to obtain the different elements of the message. In this case we have a getUser() method that will return the User that was sent by the client.



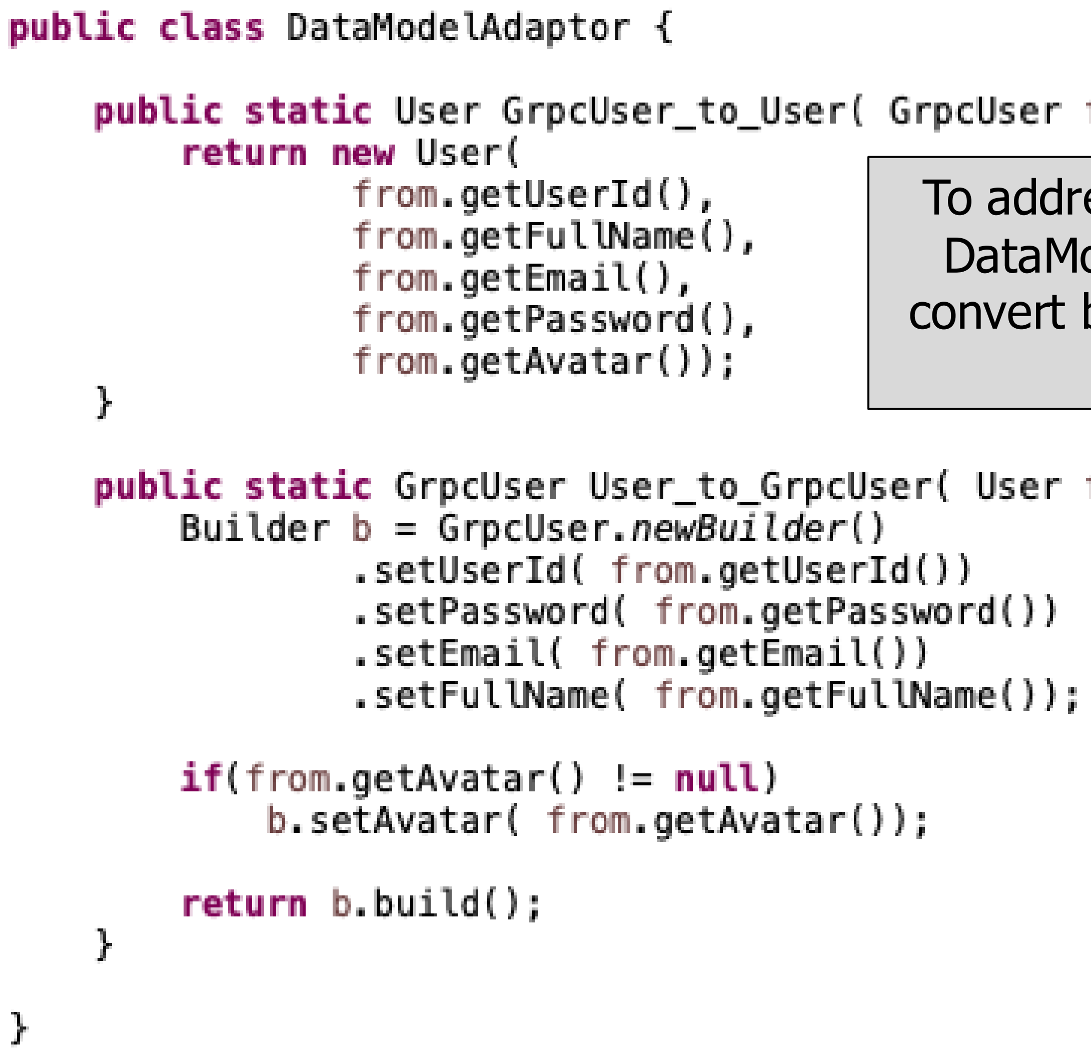
```
public class GrpcUsersServerStub implem

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

But the user will be encoded as a GrpcUser (the message we defined to encode a User) which is not an instance of the User class that is expected by our generic JavaUsers implementation…

THIS WILL REQUIRE THAT WE CONVERT BETWEEN REPRESENTATIONS OF GRPCUSER AND USER.

```java
public class DataModelAdaptor {

    public static User GrpcUser_to_User( GrpcUser from )  {
        return new User(
                from.getUserId(),
                from.getFullName(),
                from.getEmail(),
                from.getPassword(),
                from.getAvatar());
    }

    public static GrpcUser User_to_GrpcUser( User from )  {
        Builder b = GrpcUser.newBuilder()
                .setUserId( from.getUserId())
                .setPassword( from.getPassword())
                .setEmail( from.getEmail())
                .setFullName( from.getFullName());

        if(from.getAvatar() != null)
            b.setAvatar( from.getAvatar());

        return b.build();
    }

}
```

THIS WILL REQUIRE THAT WE CONVERT BETWEEN REPRESENTATIONS OF GRPCUSER AND USER.



```java
public class DataModelAdaptor {

    public static User GrpcUser_to_User( GrpcUser from )  {
        return new User(
                from.getUserId(),
                from.getFullName(),
                from.getEmail(),
                from.getPassword(),
                from.getAvatar());
    }

    public static GrpcUser User_to_GrpcUser( User from )  {
        Builder b = GrpcUser.newBuilder()
                .setUserId( from.getUserId())
                .setPassword( from.getPassword())
                .setEmail( from.getEmail())
                .setFullName( from.getFullName());

        if(from.getAvatar() != null)
            b.setAvatar( from.getAvatar());

        return b.build();
    }

}
```

To address this, we have created a DataModelAdaptor class that can convert between GrpcUser and User and vice-versa.

THIS WILL REQUIRE THAT WE CONVERT BETWEEN REPRESENTATIONS OF GRPCUSER AND USER.

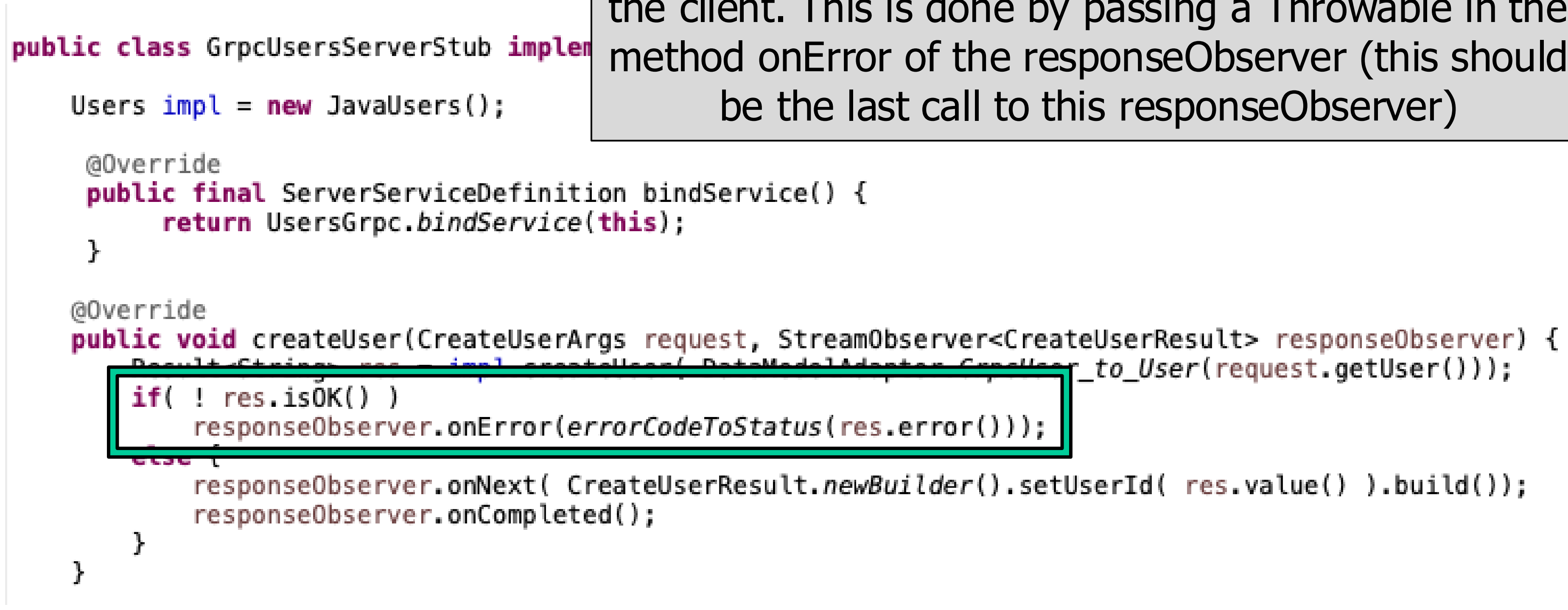
```
public class DataModelAdaptor {

    public static User GrpcUser_to_User( GrpcUser from )  {
        return new User(
                from.getUserId(),
                from.getFullName(),
                from.getEmail(),
                from.getPassword(),
                from.getAvatar());
    }

    public static GrpcUser User_to_GrpcUser( User from )  {
        Builder b = GrpcUser.newBuilder()
                .setUserId( from.getUserId())
                .setPassword( from.getPassword())
                .setEmail( from.getEmail())
                .setFullName( from.getFullName());

        if(from.getAvatar() != null)
            b.setAvatar( from.getAvatar());

        return b.build();
    }

}
```

While all elements of the GrpcUser message are optional, you cannot set a gRPC message element to null!

THE GRPCUSERSSERVERSTUB

After executing the (generic) service logic we can test the success of the operation.

In case of an error, we have to report that error to the client. This is done by passing a Throwable in the method onError of the responseObserver (this should be the last call to this responseObserver)

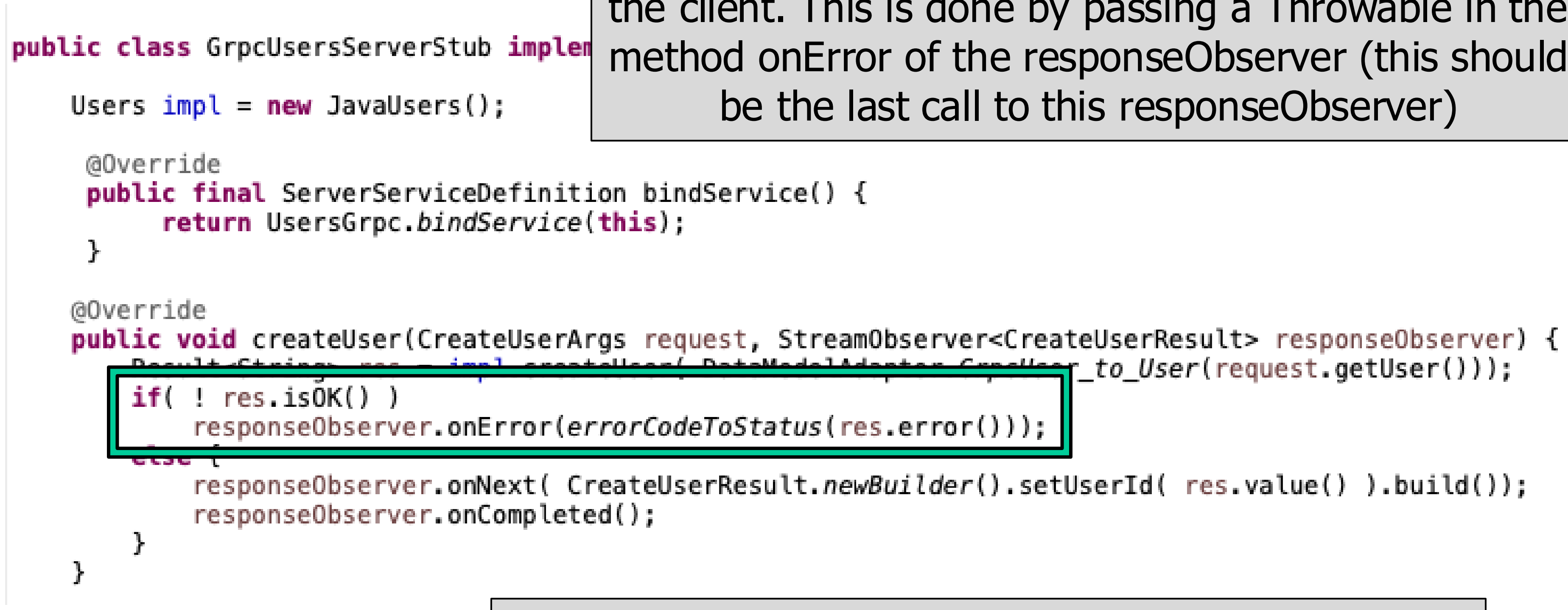


```
public class GrpcUsersServerStub implem

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdapter.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

THE GRPCUSERSSERVERSTUB

After executing the (generic) service logic we can test the success of the operation.

In case of an error, we have to report that error to the client. This is done by passing a Throwable in the method onError of the responseObserver (this should be the last call to this responseObserver)

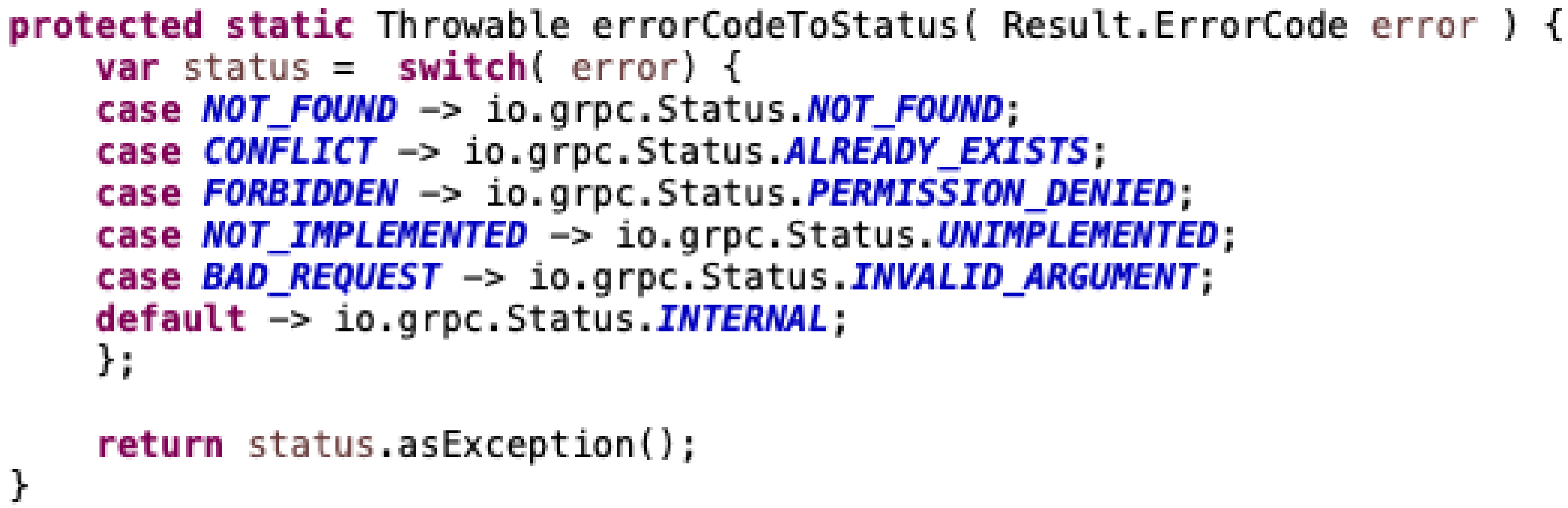


```
public class GrpcUsersServerStub implem

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdapter.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

We use a set of generic codes in the Result message that now must be converted to the appropriate Throwable. To do this we use an auxiliary function.

TRANSLATING ERROR CODES TO THROWABLE

We use a set of generic codes in the Result message that now must be converted to the appropriate Throwable. To do this we use an auxiliary function.



```java
protected static Throwable errorCodeToStatus( Result.ErrorCode error ) {
    var status =  switch( error) {
    case NOT_FOUND -> io.grpc.Status.NOT_FOUND;
    case CONFLICT -> io.grpc.Status.ALREADY_EXISTS;
    case FORBIDDEN -> io.grpc.Status.PERMISSION_DENIED;
    case NOT_IMPLEMENTED -> io.grpc.Status.UNIMPLEMENTED;
    case BAD_REQUEST -> io.grpc.Status.INVALID_ARGUMENT;
    default -> io.grpc.Status.INTERNAL;
    };

    return status.asException();
}
```

TRANSLATING ERROR CODES TO THROWABLE

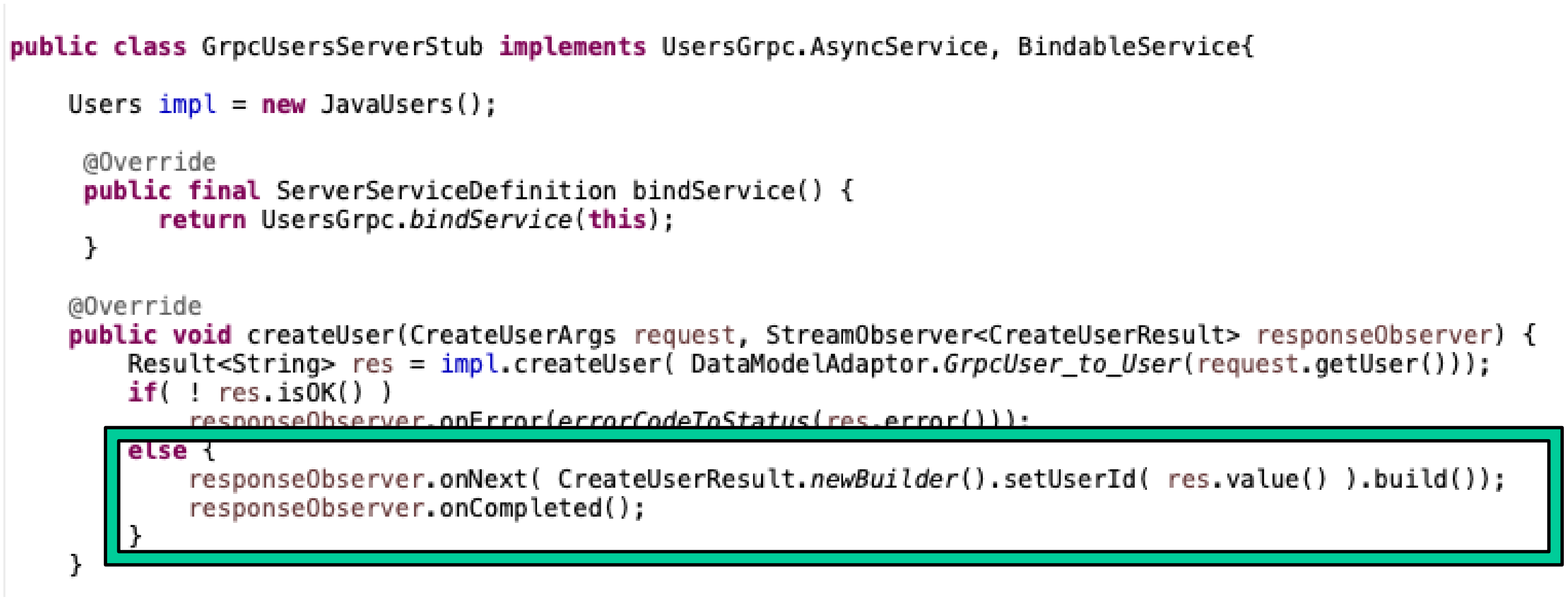
We use a set of generic codes in the Result message that now must be converted to the appropriate Throwable. To do this we use an auxiliary function.

```
protected static Throwable errorCodeToStatus( Result.ErrorCode error ) {
    var status =  switch( error) {
    case NOT_FOUND -> io.grpc.Status.NOT_FOUND;
    case CONFLICT -> io.grpc.Status.ALREADY_EXISTS;
    case FORBIDDEN -> io.grpc.Status.PERMISSION_DENIED;
    case NOT_IMPLEMENTED -> io.grpc.Status.UNIMPLEMENTED;
    case BAD_REQUEST -> io.grpc.Status.INVALID_ARGUMENT;
    default -> io.grpc.Status.INTERNAL;
    };

    return status.asException();
}
```

REST services implementations must throw a WebApplicationException with a different error code (HTTP error codes). We can have a similar function to this one but that returns a Status in the REST implementation.

THE GRPCUSERSSERVERSTUB

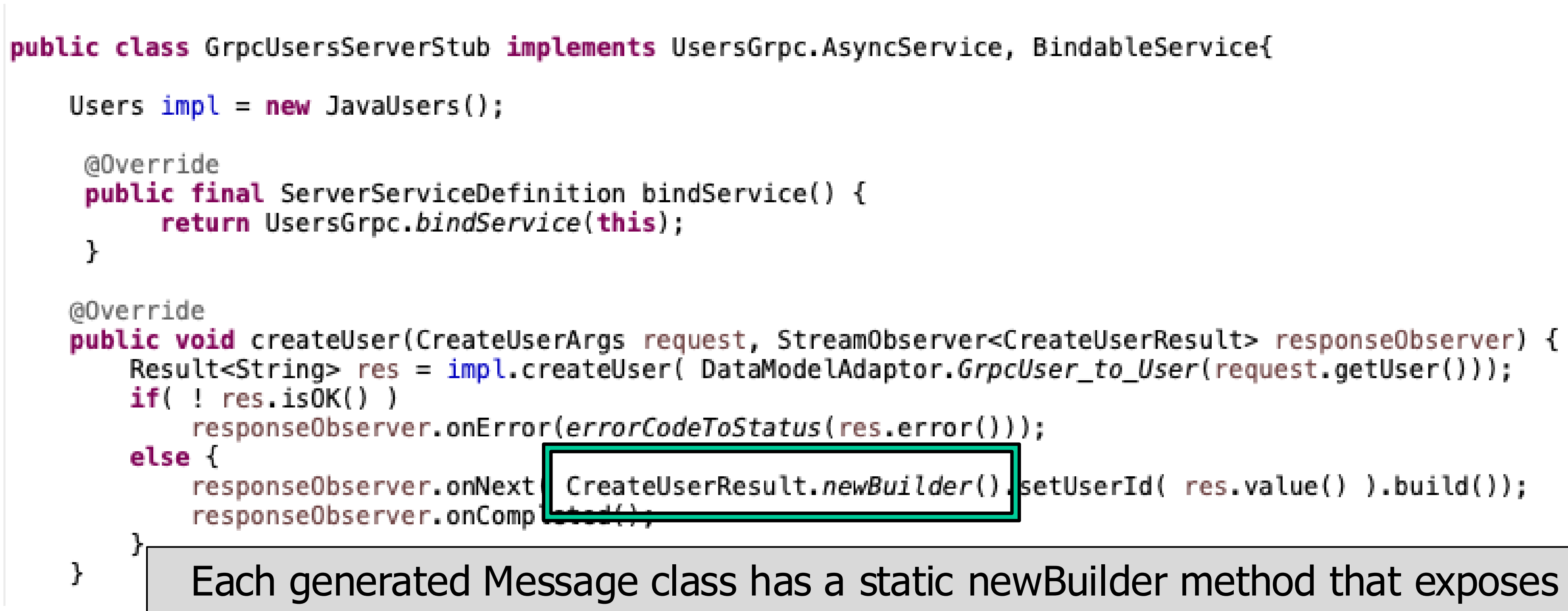
If the operation is successful, we must generate and send an Appropriate Response Message as specified in the RPC specification (in this case a CreateUserResult)



```
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

THE GRPCUSERSSERVERSTUB

If the operation is successful, we must generate and send an Appropriate Response Message as specified in the RPC specification (in this case a CreateUserResult)

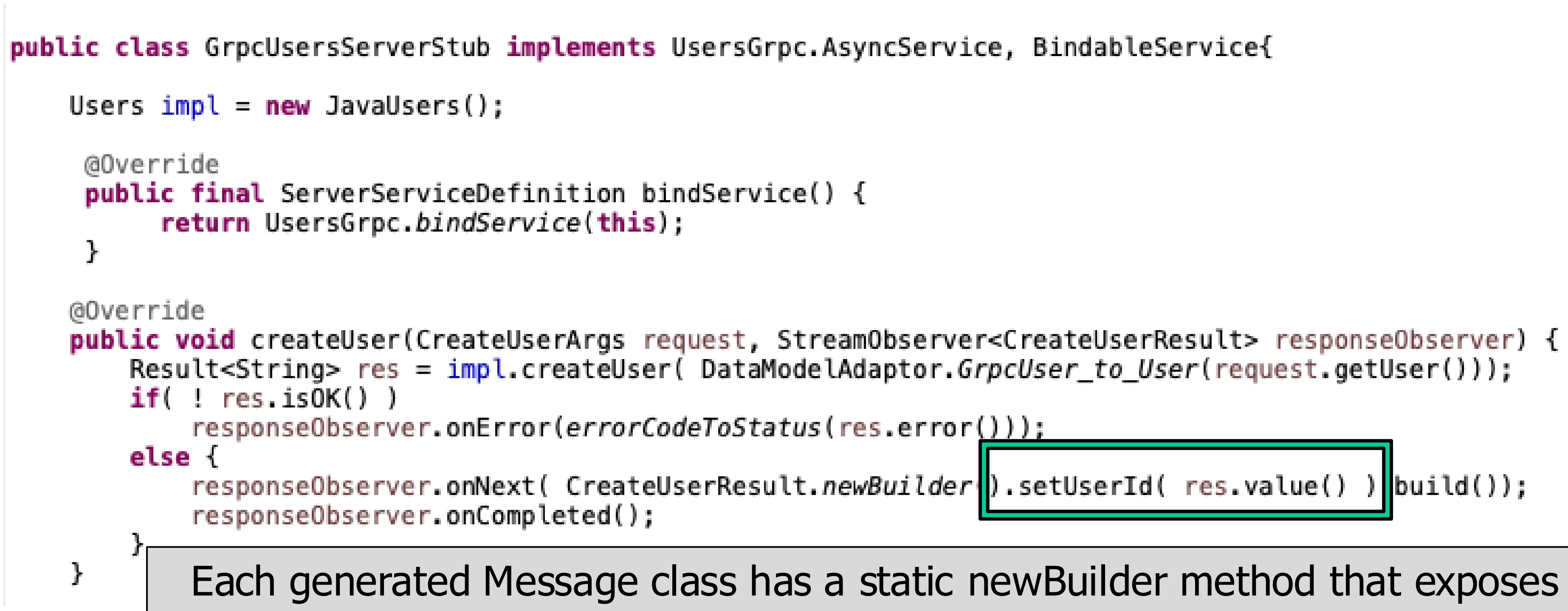


```java
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

Each generated Message class has a static newBuilder method that exposes a Builder for the message…

THE GRPCUSERSSERVERSTUB

If the operation is successful, we must generate and send an Appropriate Response Message as specified in the RPC specification (in this case a CreateUserResult)



```
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

Each generated Message class has a static newBuilder method that exposes a Builder for the message…
…that allows you to set all fields (its chainable, but you cannot set a field to null, even if it is optional)

THE GRPCUSERSSERVERSTUB

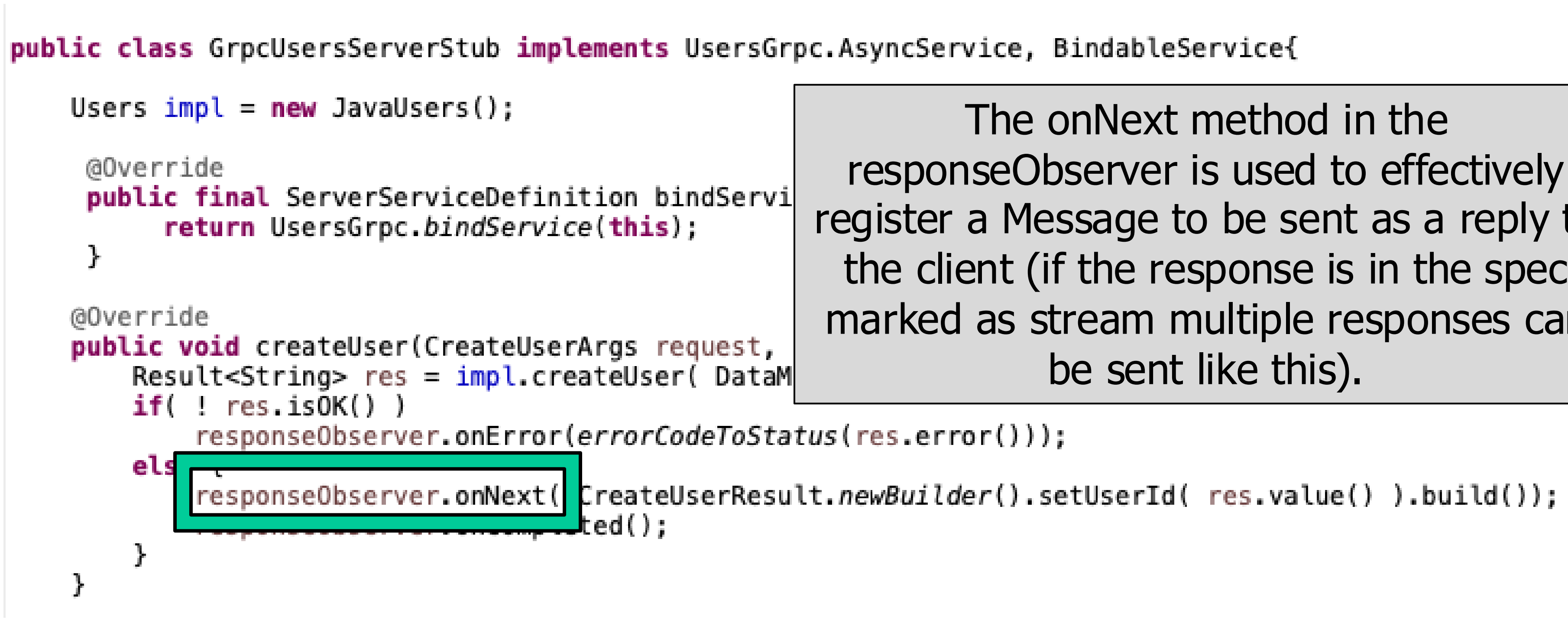
If the operation is successful, we must generate and send an Appropriate Response Message as specified in the RPC specification (in this case a CreateUserResult)

```java
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindService() {
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request, StreamObserver<CreateUserResult> responseObserver) {
        Result<String> res = impl.createUser( DataModelAdaptor.GrpcUser_to_User(request.getUser()));
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

Each generated Message class has a static newBuilder method that exposes a Builder for the message…
…that allows you to set all fields (its chainable, but you cannot set a field to null, even if it is optional)
Finally, a method build allows to create the instance of the Message.

THE GRPCUSERSSERVERSTUB

If the operation is successful, we must generate and send an Appropriate Response Message as specified in the RPC specification (in this case a CreateUserResult)

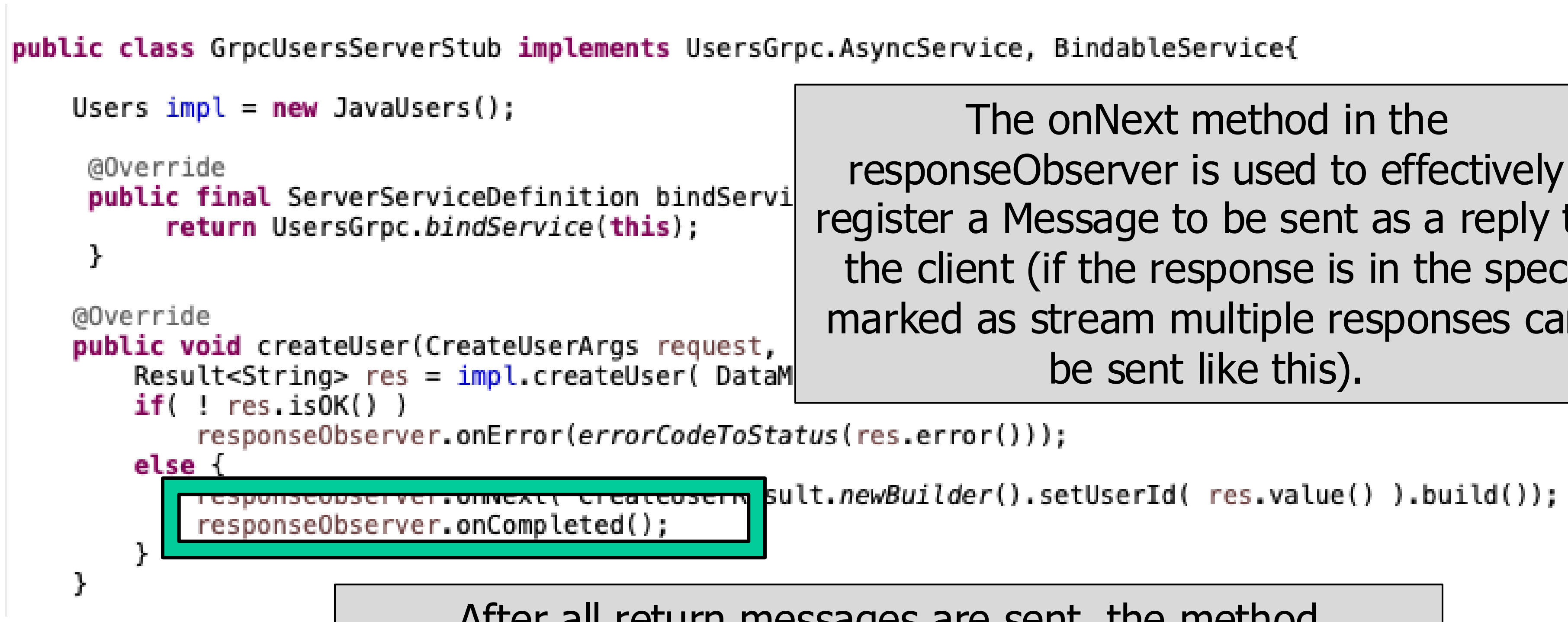


```
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindServi
         return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request,
        Result<String> res = impl.createUser( DataM
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        els
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
                                    ted();
        }
    }
```

The onNext method in the responseObserver is used to effectively register a Message to be sent as a reply to the client (if the response is in the spec marked as stream multiple responses can be sent like this).

THE GRPCUSERSSERVERSTUB

If the operation is successful, we must generate and send an Appropriate Response Message as specified in the RPC specification (in this case a CreateUserResult)



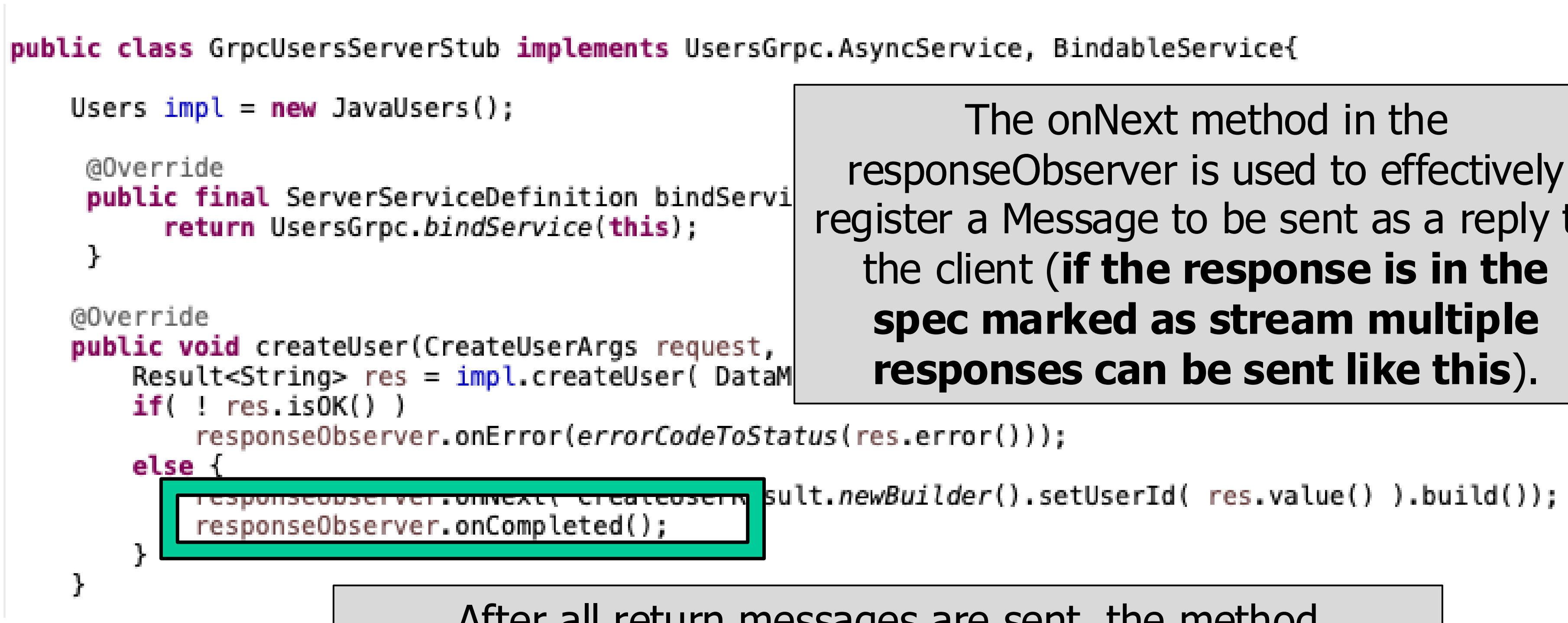
```
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindServi
         return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request,
        Result<String> res = impl.createUser( DataM
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

The onNext method in the responseObserver is used to effectively register a Message to be sent as a reply to the client (if the response is in the spec marked as stream multiple responses can be sent like this).

After all return messages are sent, the method onCompleted must be called! After this no more messages can be sent.

THE GRPCUSERSSERVERSTUB

If the operation is successful, we must generate and send an Appropriate Response Message as specified in the RPC specification (in this case a CreateUserResult)



```
public class GrpcUsersServerStub implements UsersGrpc.AsyncService, BindableService{

    Users impl = new JavaUsers();

     @Override
     public final ServerServiceDefinition bindServi
          return UsersGrpc.bindService(this);
     }

    @Override
    public void createUser(CreateUserArgs request,
        Result<String> res = impl.createUser( DataM
        if( ! res.isOK() )
            responseObserver.onError(errorCodeToStatus(res.error()));
        else {
            responseObserver.onNext( CreateUserResult.newBuilder().setUserId( res.value() ).build());
            responseObserver.onCompleted();
        }
    }
```

The onNext method in the responseObserver is used to effectively register a Message to be sent as a reply to the client (**if the response is in the spec marked as stream multiple responses can be sent like this**).

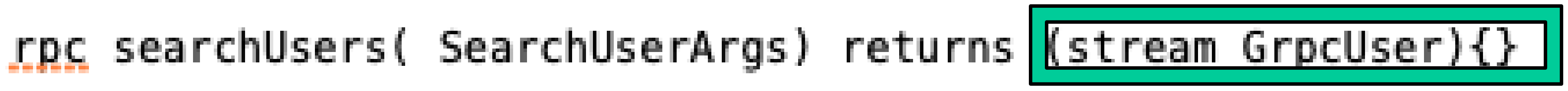
After all return messages are sent, the method onCompleted must be called! After this no more messages can be sent.

THE GRPCUSERSSERVERSTUB

```
rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
```

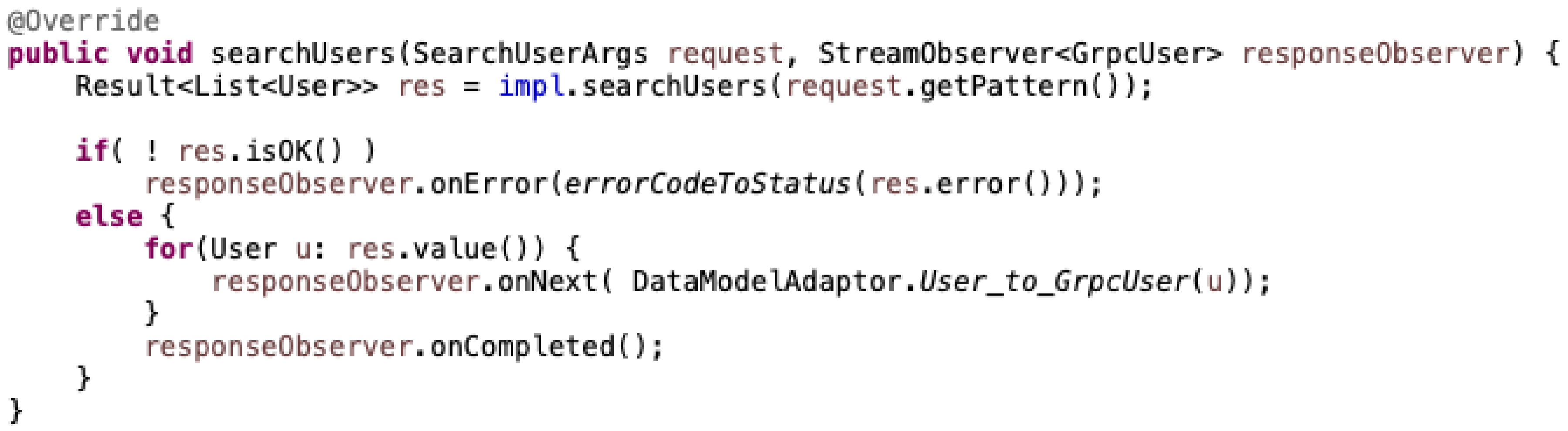
```java
@Override
public void searchUsers(SearchUserArgs request, StreamObserver<GrpcUser> responseObserver) {
    Result<List<User>> res = impl.searchUsers(request.getPattern());

    if( ! res.isOK() )
        responseObserver.onError(errorCodeToStatus(res.error()));
    else {
        for(User u: res.value()) {
            responseObserver.onNext( DataModelAdaptor.User_to_GrpcUser(u));
        }
        responseObserver.onCompleted();
    }
}
```

THE GRPCUSERSSERVERSTUB



```
rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
```

For the searchUsers operation the return is marked as stream, which means that multiple GrpcUser instances can be sent to the client.



```
@Override
public void searchUsers(SearchUserArgs request, StreamObserver<GrpcUser> responseObserver) {
    Result<List<User>> res = impl.searchUsers(request.getPattern());

    if( ! res.isOK() )
        responseObserver.onError(errorCodeToStatus(res.error()));
    else {
        for(User u: res.value()) {
            responseObserver.onNext( DataModelAdaptor.User_to_GrpcUser(u));
        }
        responseObserver.onCompleted();
    }
}
```

THE GRPCUSERSSERVERSTUB

```
rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
```

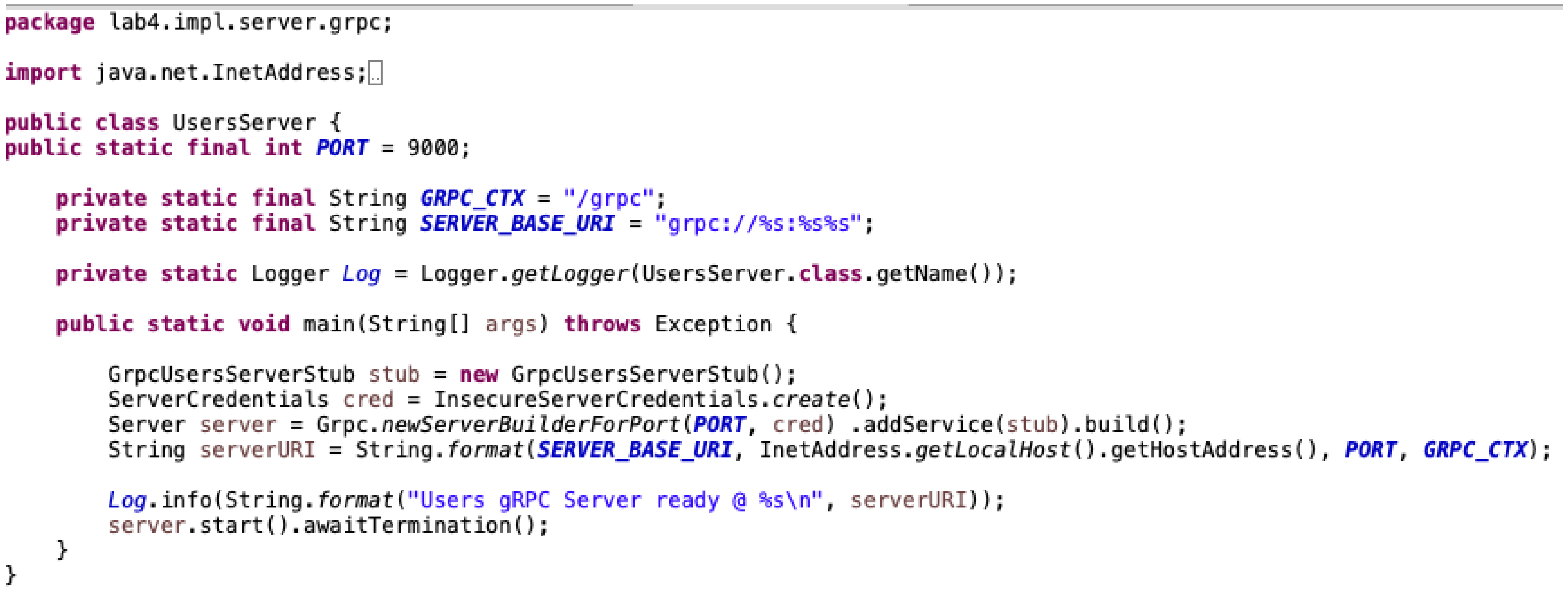
For the searchUsers operation the return is marked as stream, which means that multiple GrpcUser instances can be sent to the client.

```
@Override
public void searchUsers(SearchUserArgs request, StreamObserver<GrpcUser> responseObserver) {
    Result<List<User>> res = impl.searchUsers(request.getPattern());

    if( ! res.isOK() )
        responseObserver.onError(errorCodeToStatus(res.error()));
    else {
        for(User u: res.value()) {
            responseObserver.onNext( DataModelAdaptor.User_to_GrpcUser(u));
        }
        responseObserver.onCompleted();
    }
}
```

This is achieved by multiple invocations of the onNext method on the responseObserver.

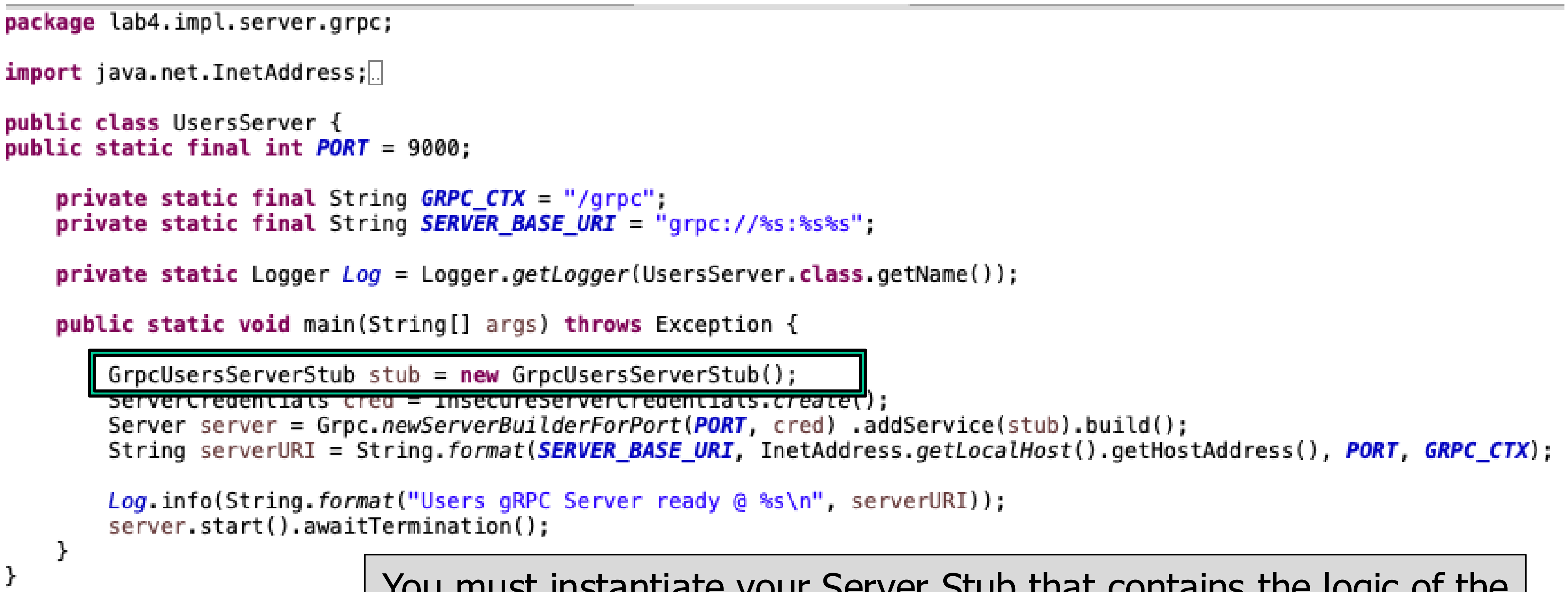
After generating all responses, the onCompleted methods must be called (a single time).

LAUNCHING THE gRPC SERVER



```java
package lab4.impl.server.grpc;

import java.net.InetAddress;

public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

    private static Logger Log = Logger.getLogger(UsersServer.class.getName());

    public static void main(String[] args) throws Exception {

        GrpcUsersServerStub stub = new GrpcUsersServerStub();
        ServerCredentials cred = InsecureServerCredentials.create();
        Server server = Grpc.newServerBuilderForPort(PORT, cred) .addService(stub).build();
        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostAddress(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

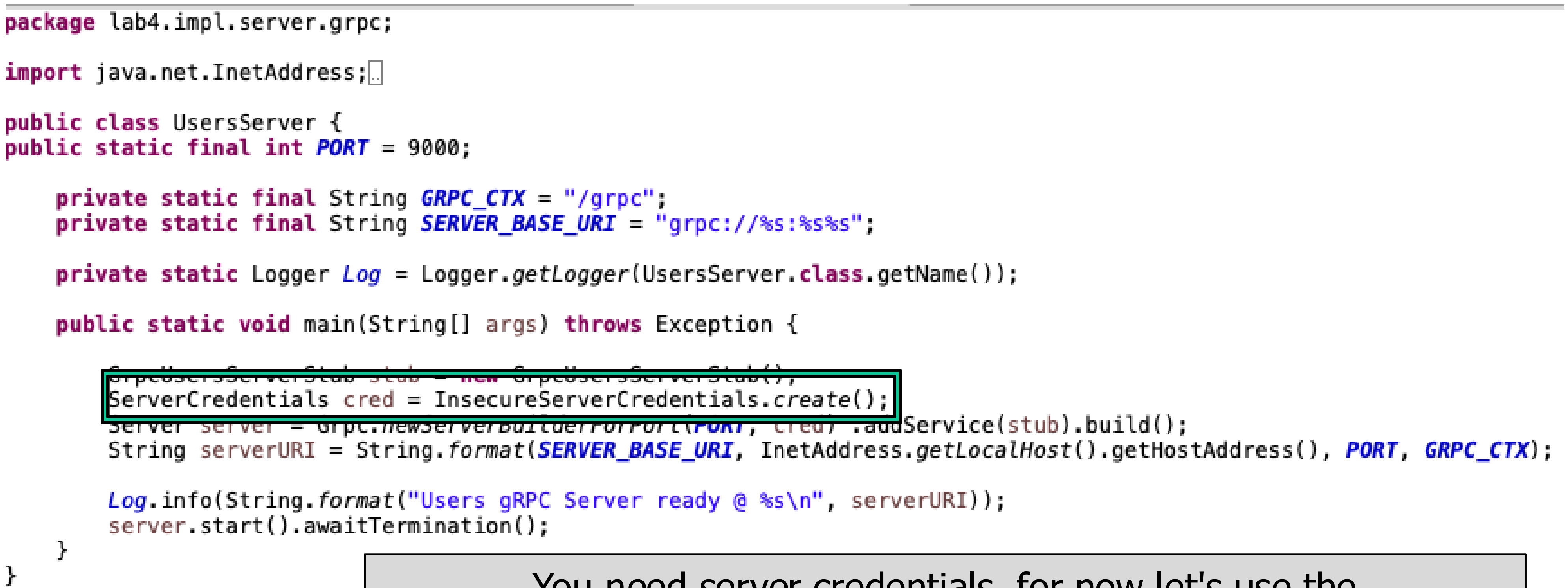
LAUNCHING THE gRPC SERVER



```
package lab4.impl.server.grpc;

import java.net.InetAddress;

public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

    private static Logger Log = Logger.getLogger(UsersServer.class.getName());

    public static void main(String[] args) throws Exception {

        GrpcUsersServerStub stub = new GrpcUsersServerStub();
        ServerCredentials cred = InsecureServerCredentials.create();
        Server server = Grpc.newServerBuilderForPort(PORT, cred) .addService(stub).build();
        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostAddress(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

You must instantiate your Server Stub that contains the logic of the service and the handlers for the different RPCs.

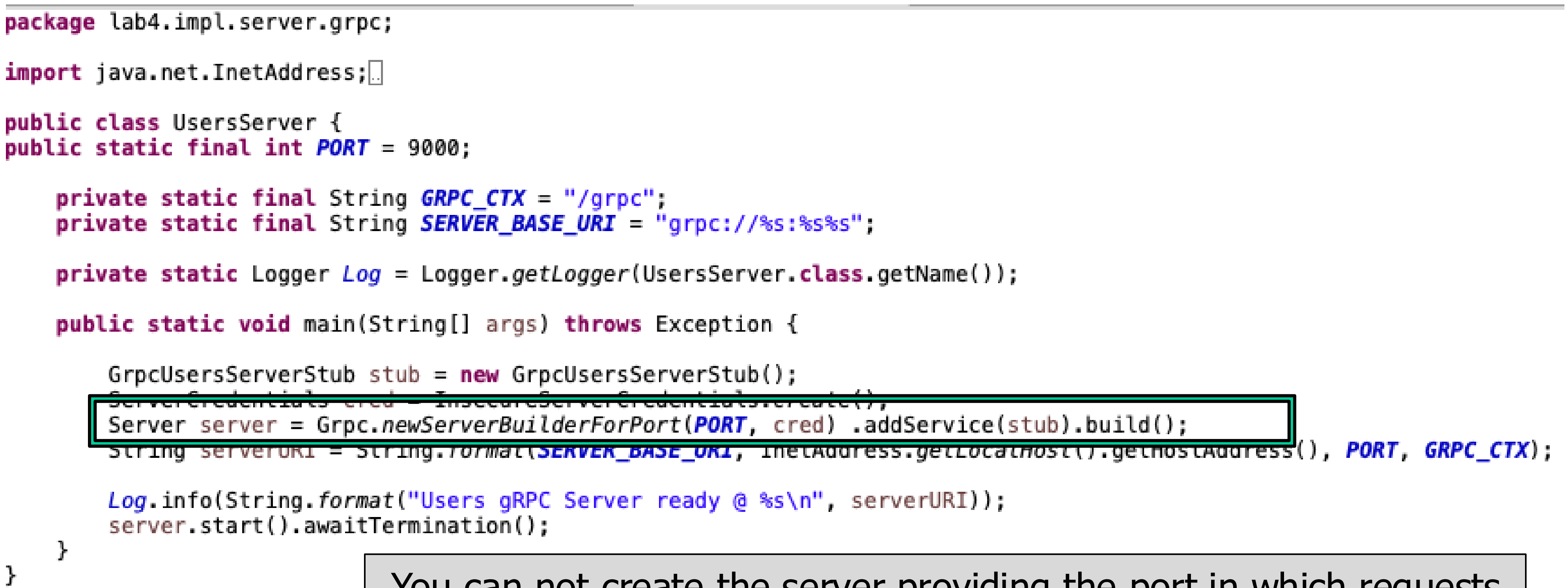
LAUNCHING THE gRPC SERVER



```
package lab4.impl.server.grpc;

import java.net.InetAddress;

public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

    private static Logger Log = Logger.getLogger(UsersServer.class.getName());

    public static void main(String[] args) throws Exception {

        GrpcUsersServerStub stub = new GrpcUsersServerStub();
        ServerCredentials cred = InsecureServerCredentials.create();
        Server server = Grpc.newServerBuilderForPort(PORT, cred).addService(stub).build();
        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostAddress(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

You need server credentials, for now let's use the InsecureServerCredentials (we will revisit this in the second project)

LAUNCHING THE gRPC SERVER

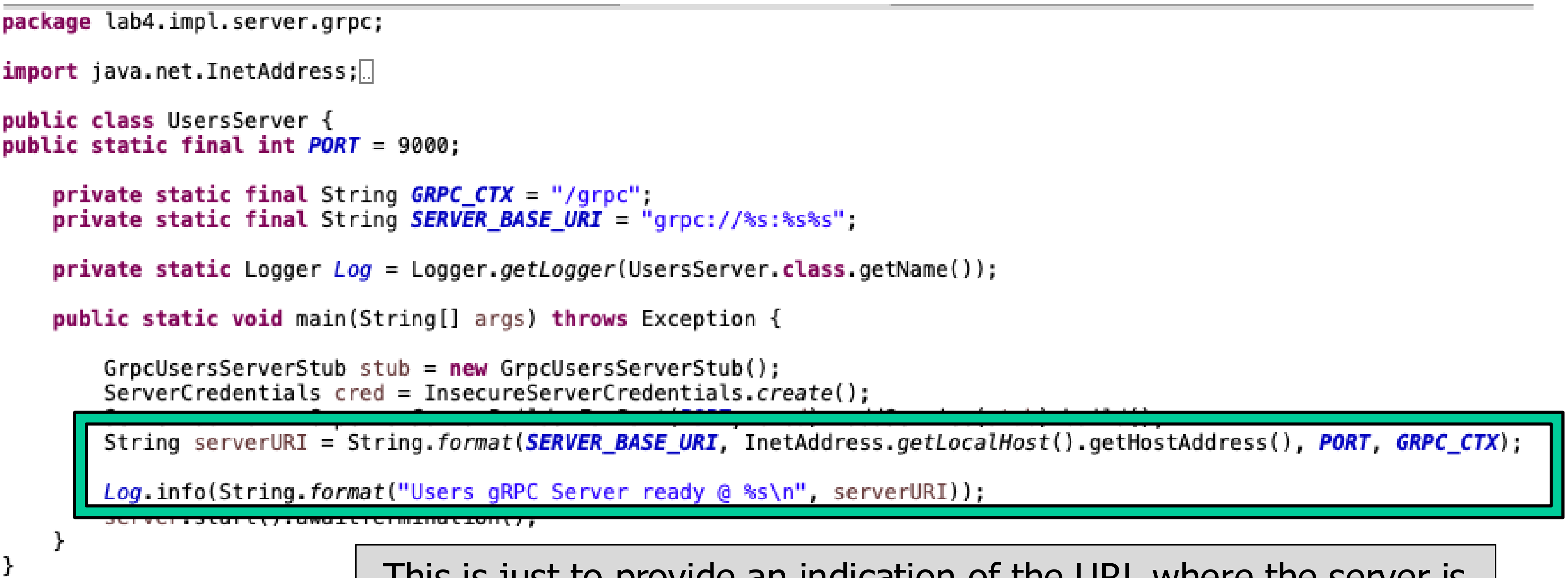


```
package lab4.impl.server.grpc;

import java.net.InetAddress;

public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

    private static Logger Log = Logger.getLogger(UsersServer.class.getName());

    public static void main(String[] args) throws Exception {

        GrpcUsersServerStub stub = new GrpcUsersServerStub();
        ServerCredentials cred = InsecureServerCredentials.create();
        Server server = Grpc.newServerBuilderForPort(PORT, cred) .addService(stub).build();
        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostAddress(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

You can not create the server providing the port in which requests are going to be received, and the ServerCredentials instance.

Over this we can register our ServerStub and build the server.

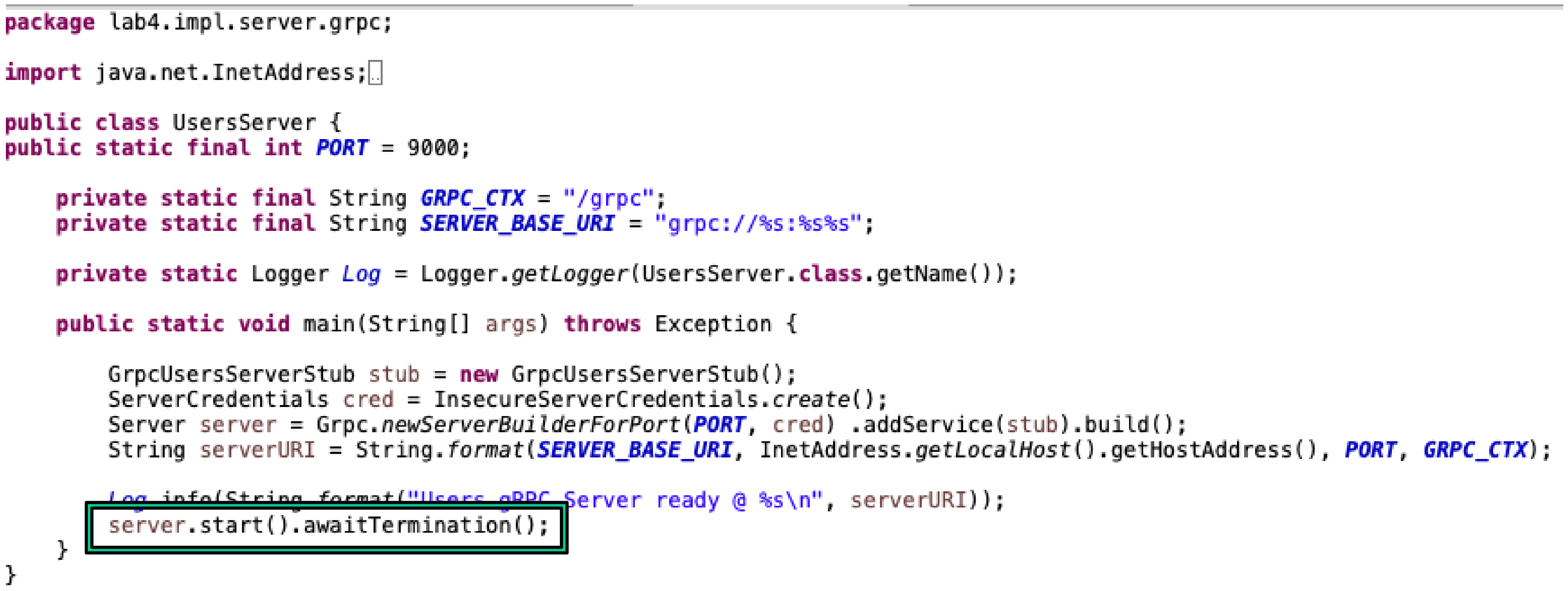
LAUNCHING THE gRPC SERVER



```
package lab4.impl.server.grpc;

import java.net.InetAddress;

public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

    private static Logger Log = Logger.getLogger(UsersServer.class.getName());

    public static void main(String[] args) throws Exception {

        GrpcUsersServerStub stub = new GrpcUsersServerStub();
        ServerCredentials cred = InsecureServerCredentials.create();

        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostAddress(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));

    }
}
```

This is just to provide an indication of the URL where the server is expecting to receive requests. Notice that this url starts with grpc:// and end with /grpc

Launching the gRPC Server



```
package lab4.impl.server.grpc;

import java.net.InetAddress;

public class UsersServer {
public static final int PORT = 9000;

    private static final String GRPC_CTX = "/grpc";
    private static final String SERVER_BASE_URI = "grpc://%s:%s%s";

    private static Logger Log = Logger.getLogger(UsersServer.class.getName());

    public static void main(String[] args) throws Exception {

        GrpcUsersServerStub stub = new GrpcUsersServerStub();
        ServerCredentials cred = InsecureServerCredentials.create();
        Server server = Grpc.newServerBuilderForPort(PORT, cred) .addService(stub).build();
        String serverURI = String.format(SERVER_BASE_URI, InetAddress.getLocalHost().getHostAddress(), PORT, GRPC_CTX);

        Log.info(String.format("Users gRPC Server ready @ %s\n", serverURI));
        server.start().awaitTermination();
    }
}
```

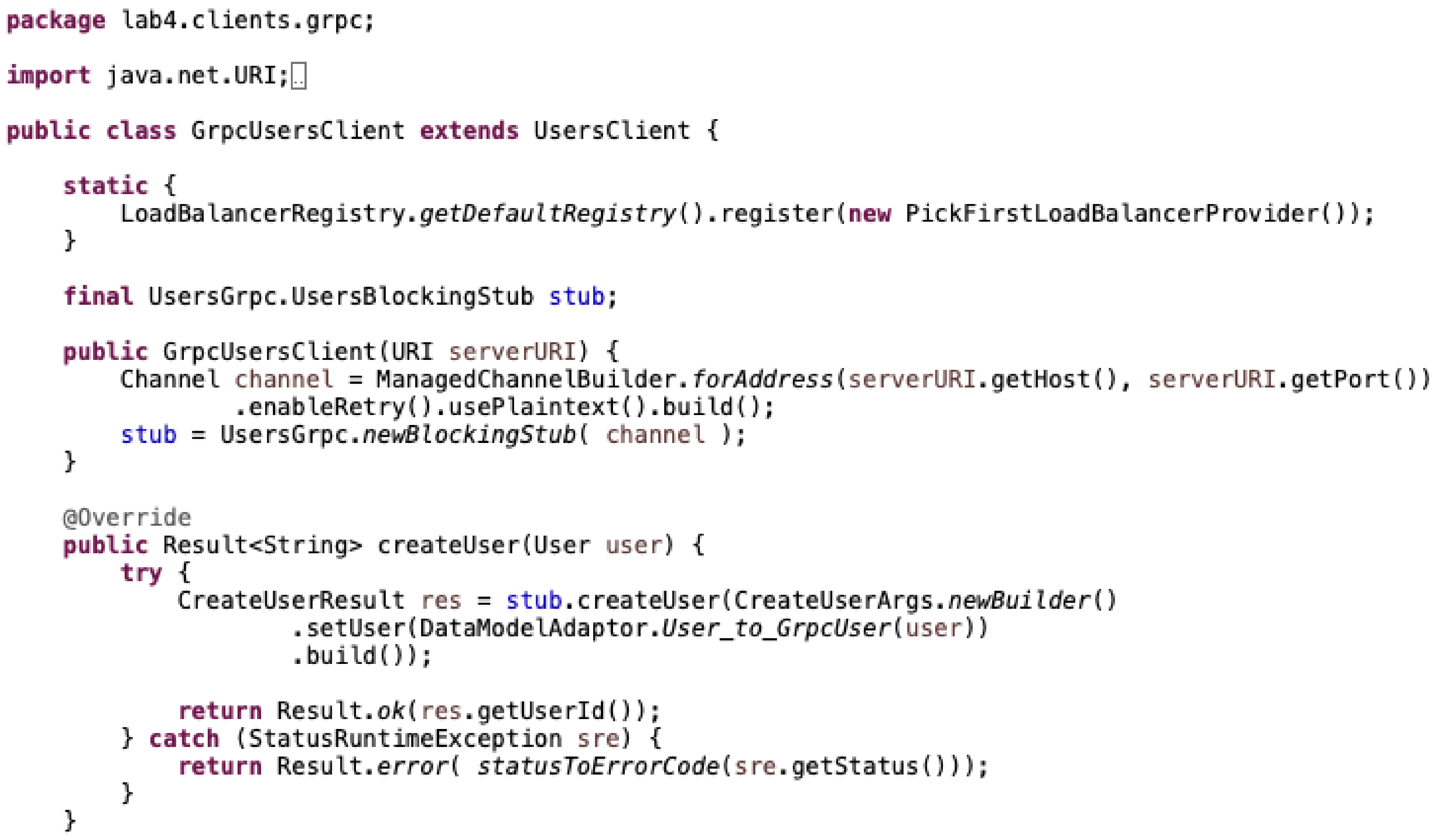
When the .start() is executed on the server the server starts. The .awaitTermination() method of the Server class waits for the server to terminate.

GOALS

In the end of this lab you should be able to:

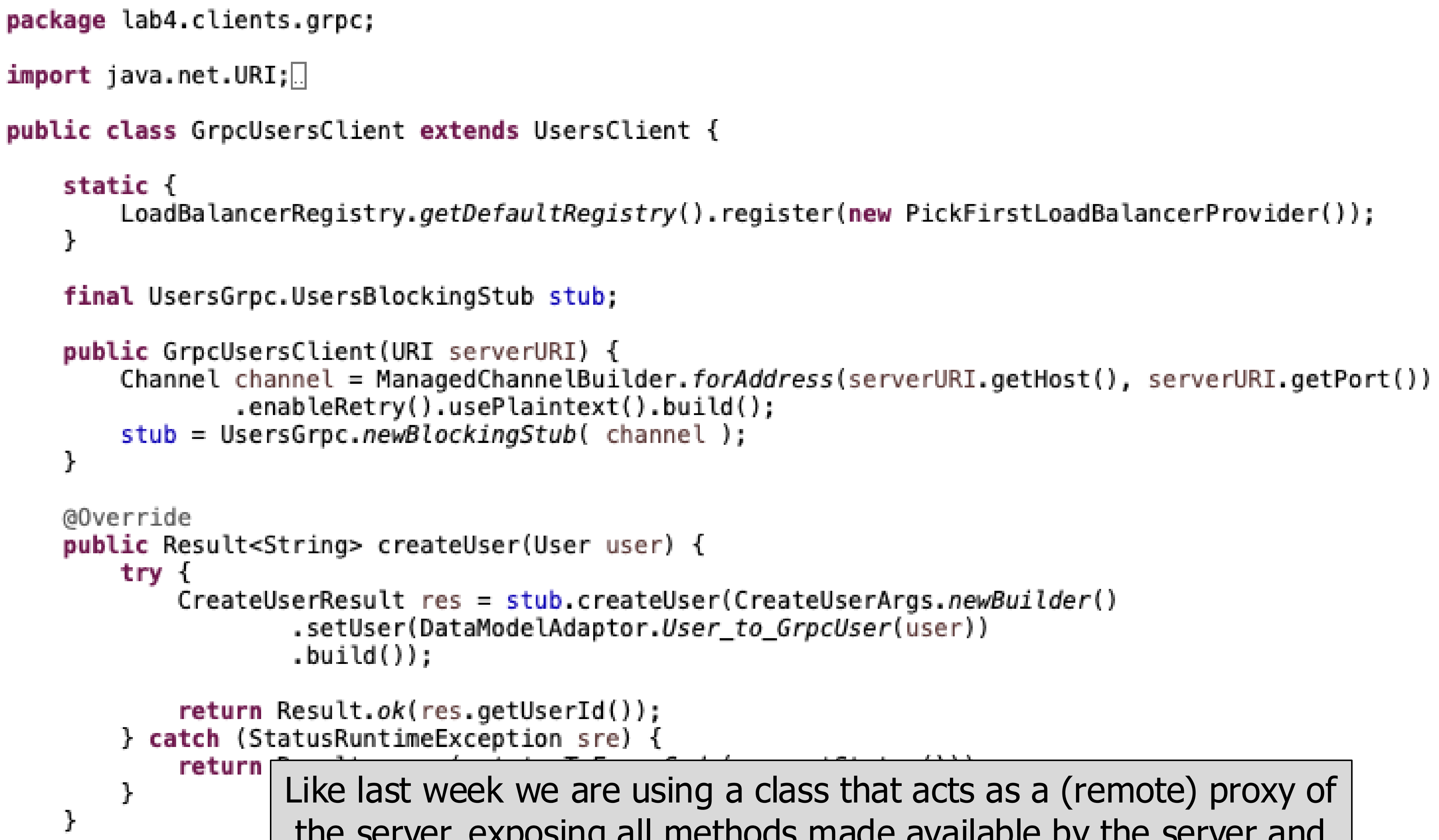
- Understand how gRPC operates and how to specify protobuf messages.
- Know how to develop gRPC services and expose them on Servers in Java.
- **Know how to develop gRPC clients in Java.**
- Understand how to avoid repeating code for the logic of the server.
- Understand how to generalize client code that can interact both with REST and gRPC servers.

gRPC clients in Java



```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (StatusRuntimeException sre) {
            return Result.error( statusToErrorCode(sre.getStatus()));
        }
    }
```

gRPC clients in Java



```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (StatusRuntimeException sre) {
            return
        }
    }
```

Like last week we are using a class that acts as a (remote) proxy of the server, exposing all methods made available by the server and that can be used to perform the remove invocations.

gRPC clients in Java

```
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (Stat
            return Re
        }
    }
```

We have however made this class extends and Abstract class called UserClient, this is to simplify the interaction with both REST and gRPC servers that expose the same service.

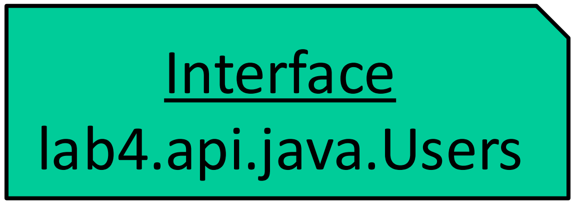
SECOND DETOUR

In the end of this lab you should be able to:

- Understand how gRPC operates and how to specify protobuf messages.
- **Know how to develop gRPC services and expose them on Servers in Java.**
- Know how to develop gRPC clients in Java.
- Understand how to avoid repeating code for the logic of the server.
- Understand how to generalize client code that can interact both with REST and gRPC servers.

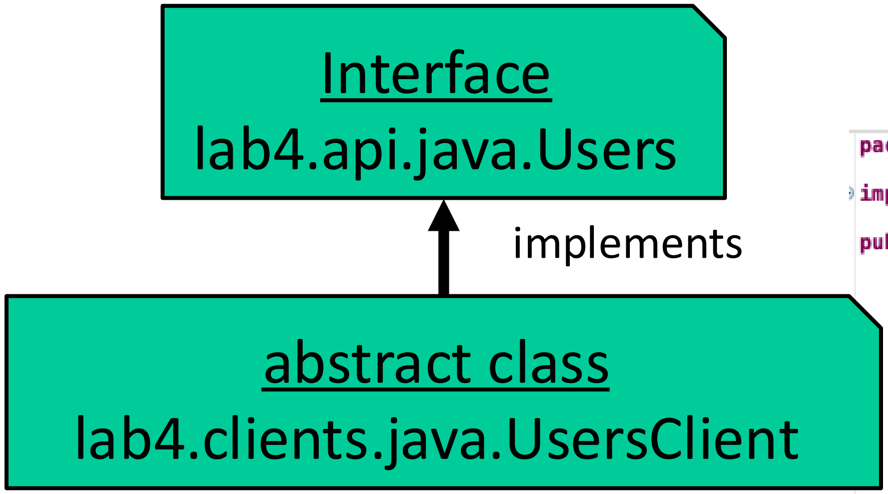
GENERALISING THE TYPE OF THE CLIENT

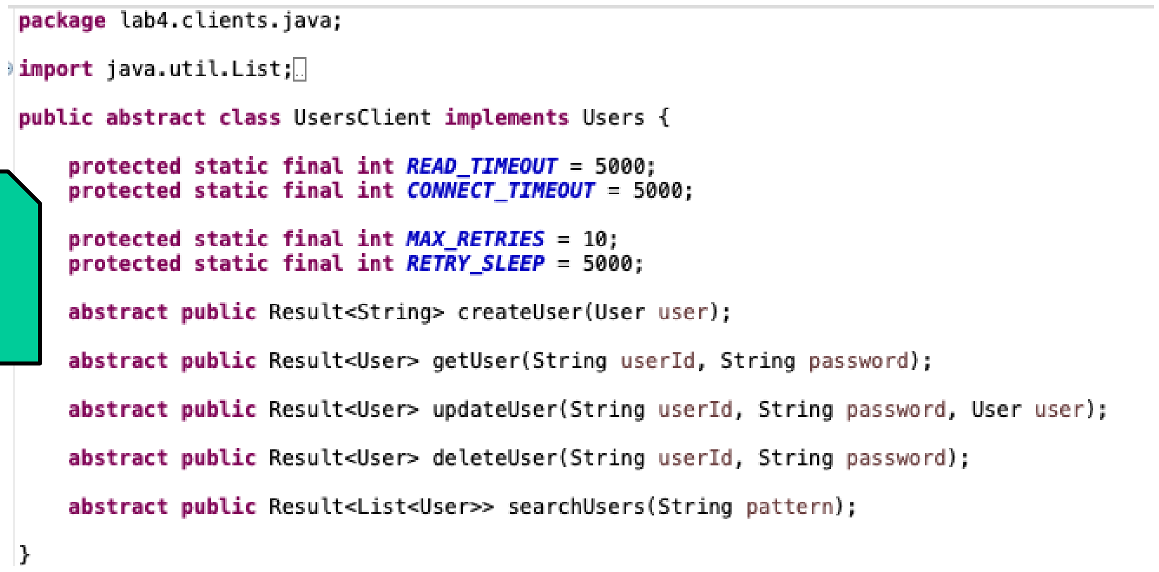
As you recall we have a (generic) api in Java for the Users functionality (the one that returns Result on all call).



GENERALISING THE TYPE OF THE CLIENT

As you recall we have a (generic) api in Java for the Users functionality (the one that returns Result on all call).



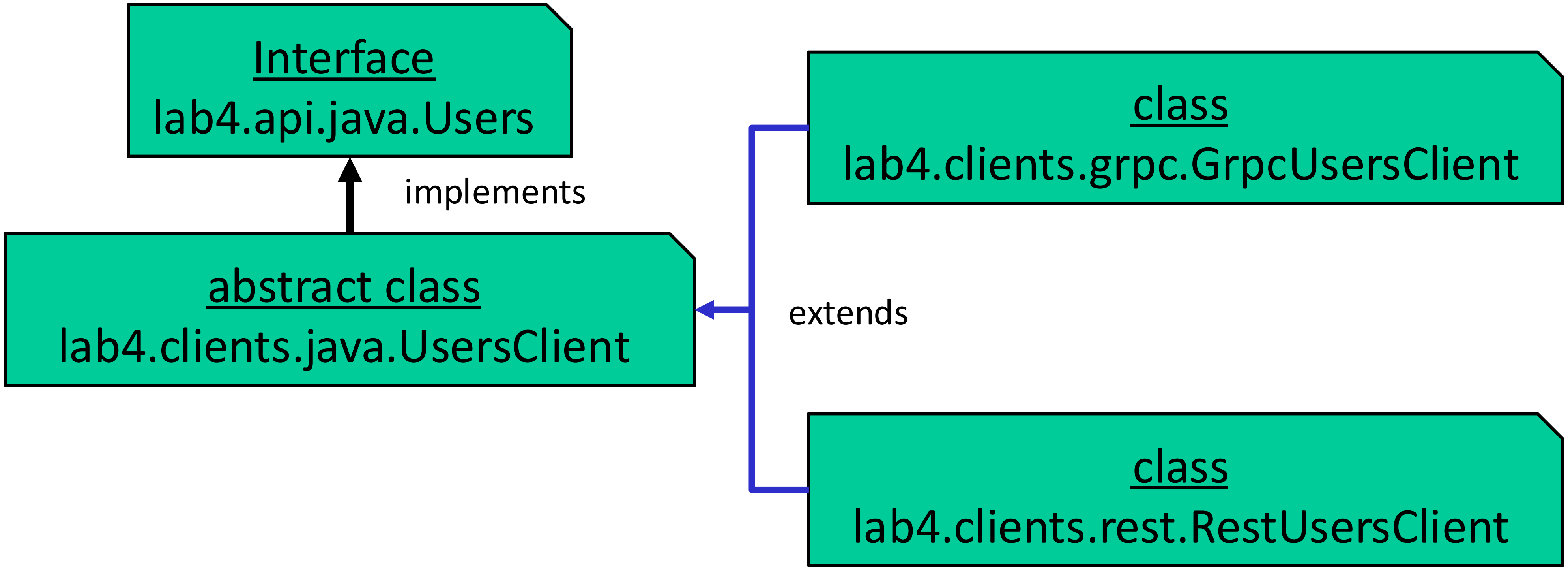


```
package lab4.clients.java;

import java.util.List;

public abstract class UsersClient implements Users {

    protected static final int READ_TIMEOUT = 5000;
    protected static final int CONNECT_TIMEOUT = 5000;

    protected static final int MAX_RETRIES = 10;
    protected static final int RETRY_SLEEP = 5000;

    abstract public Result<String> createUser(User user);

    abstract public Result<User> getUser(String userId, String password);

    abstract public Result<User> updateUser(String userId, String password, User user);

    abstract public Result<User> deleteUser(String userId, String password);

    abstract public Result<List<User>> searchUsers(String pattern);

}
```

We have created this abstract class that implements this interface and has constants to control the way clients (independently of their type) interact with the server.

Generalising The Type of the Client

As you recall we have a (generic) api in Java for the Users functionality (the one that returns Result on all call).

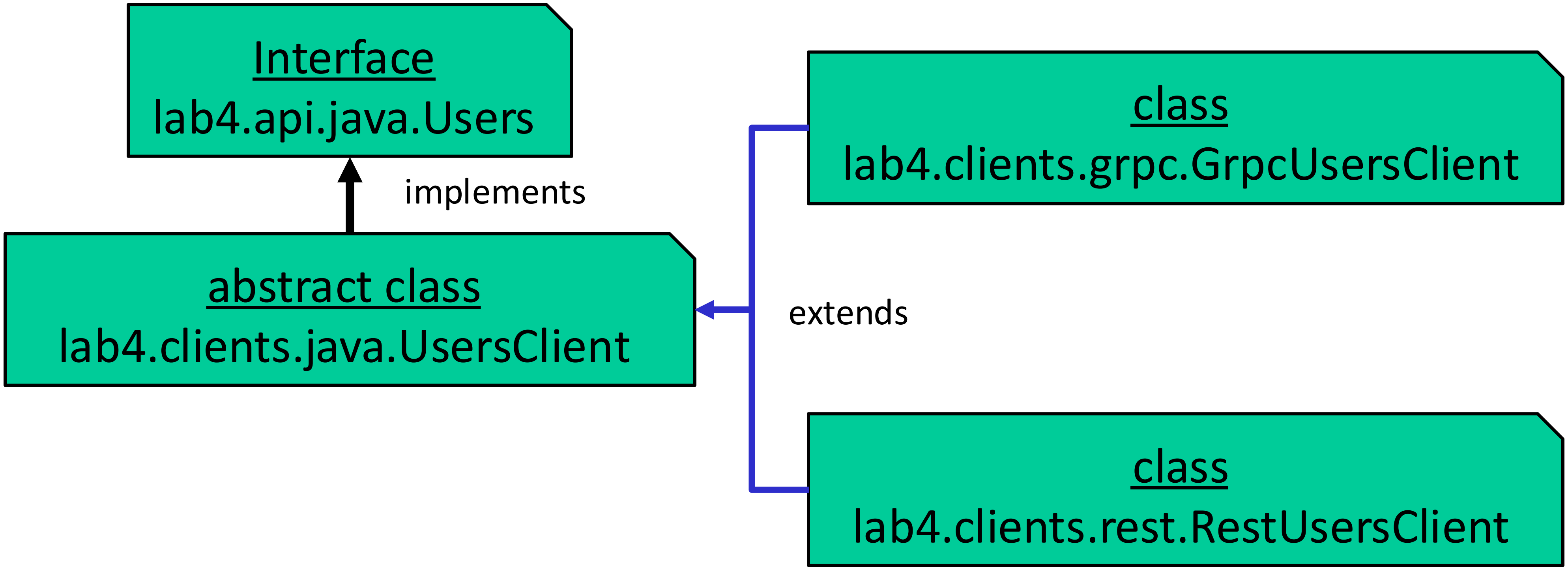

Interface
lab4.api.java.Users
implements
abstract class
lab4.clients.java.UsersClient
extends
class
lab4.clients.grpc.GrpcUsersClient
class
lab4.clients.rest.RestUsersClient

This allows to have two classes that extend this abstract class to make the remote executions using gRPC or REST respectively.

How can I decide which class to instanciate?

Generalising The Type of the Client

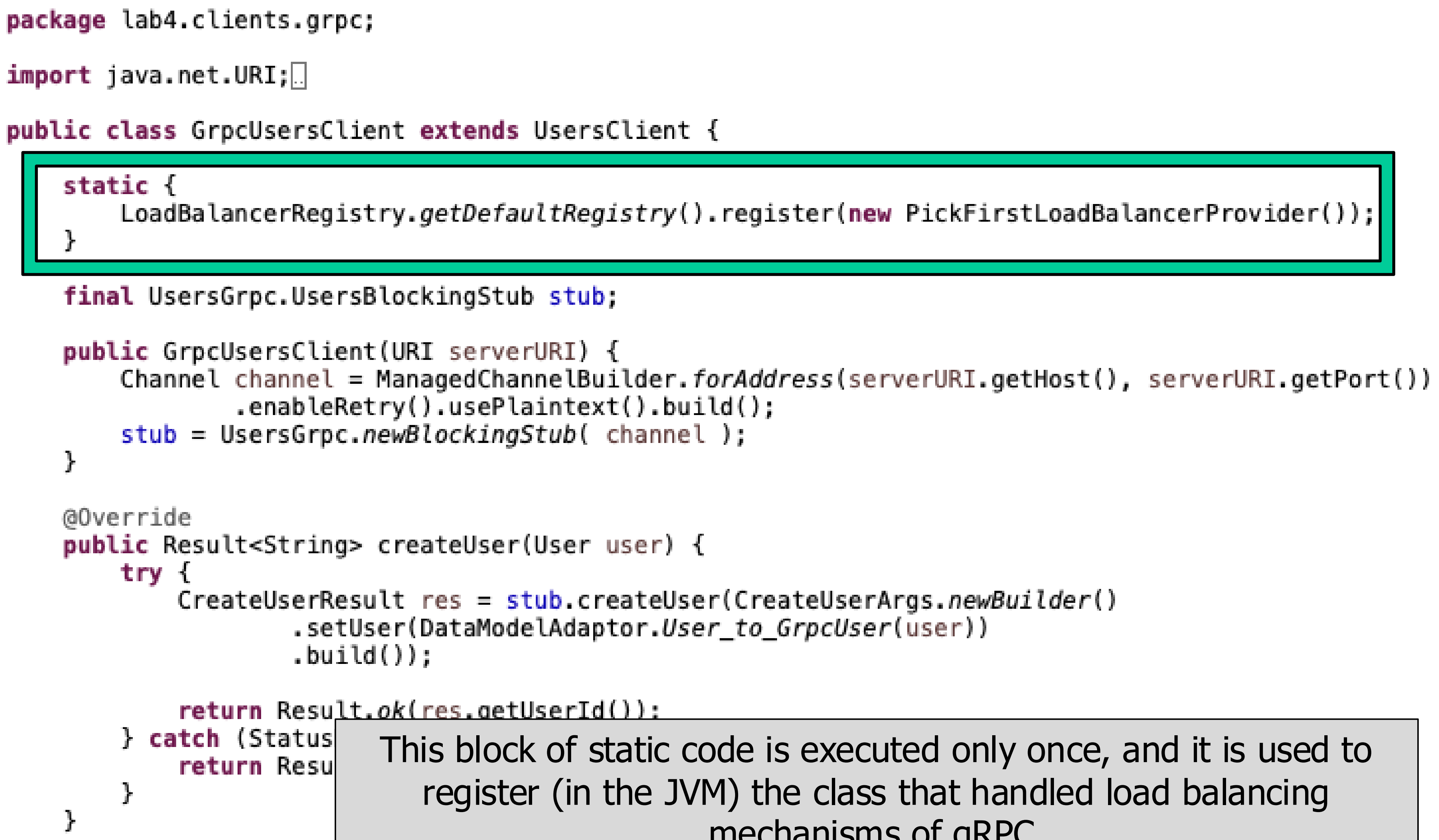
As you recall we have a (generic) api in Java for the Users functionality (the one that returns Result on all call).


Interface
lab4.api.java.Users
implements
abstract class
lab4.clients.java.UsersClient
extends
class
lab4.clients.grpc.GrpcUsersClient
class
lab4.clients.rest.RestUsersClient

I can look at the end of the server URL, and if it ends with /rest using the REST class, and if it ends with /grpc the gRPC class.

Evidently this would be betted done with a Factory that can abstract the existence of multiple classes from code.

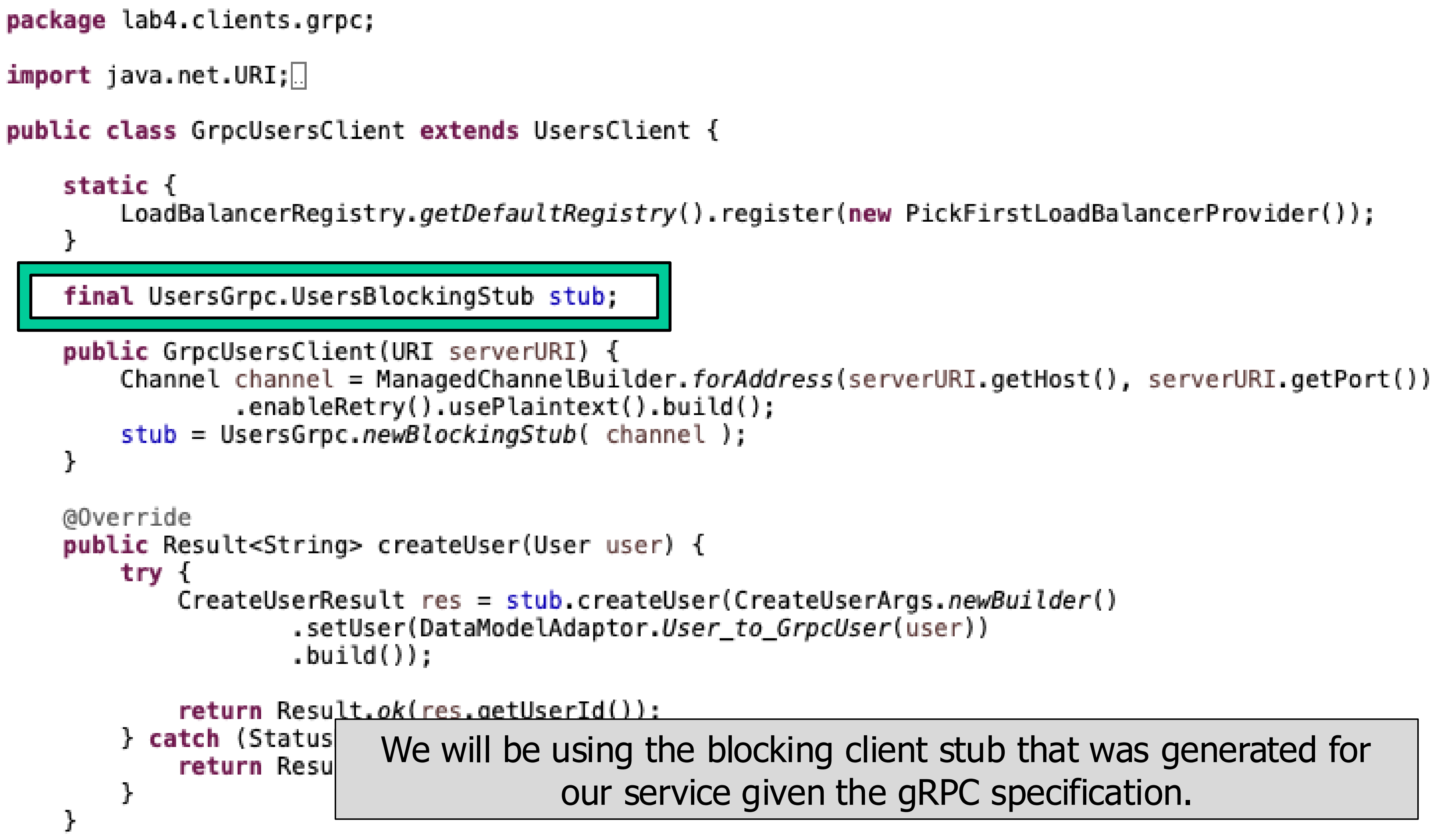
gRPC CLIENTS IN JAVA



```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (Status
            return Resu
        }
    }
```

This block of static code is executed only once, and it is used to register (in the JVM) the class that handled load balancing mechanisms of gRPC.

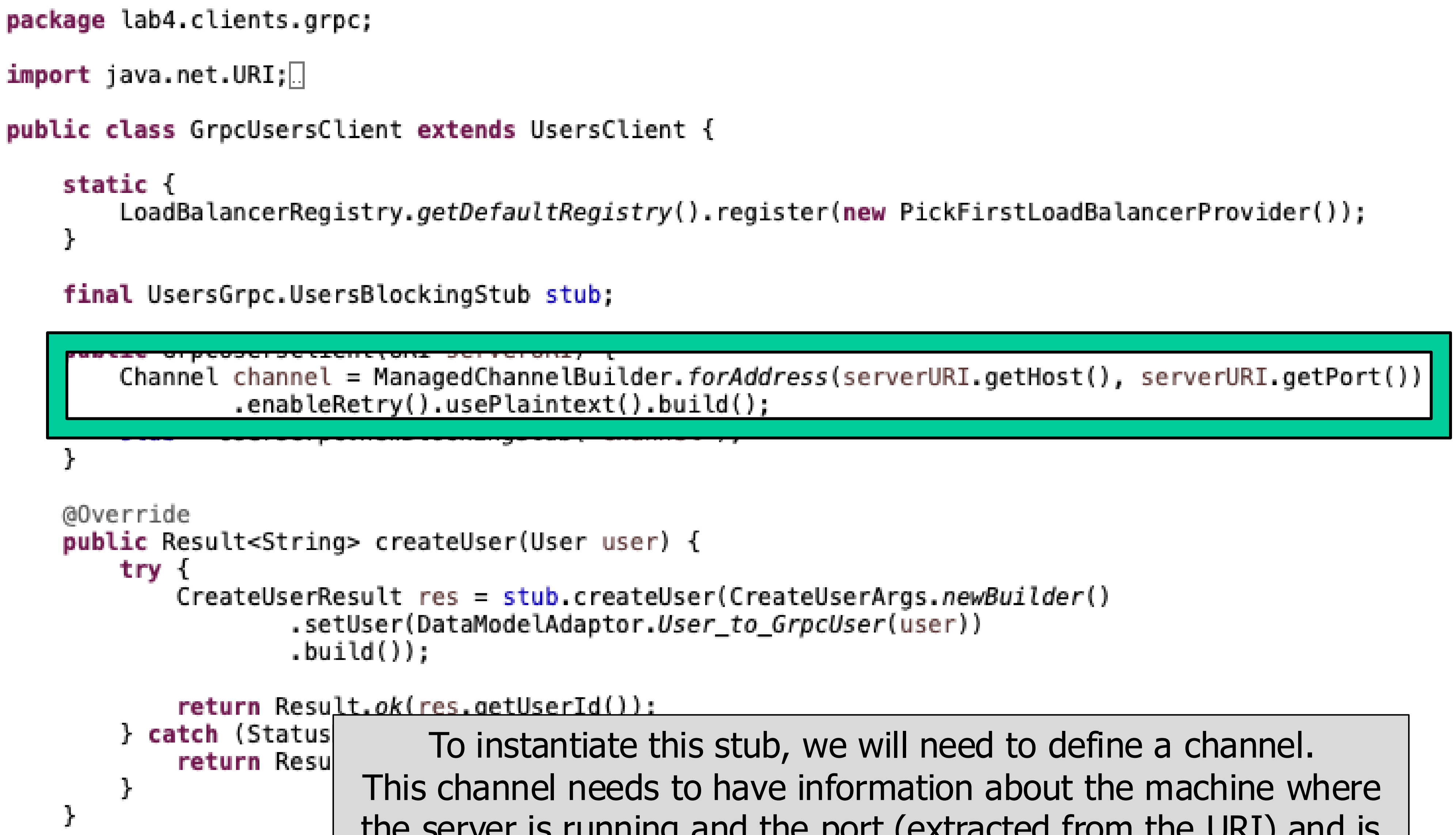
gRPC clients in Java



```
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (Status
            return Resu
        }
    }
```

We will be using the blocking client stub that was generated for our service given the gRPC specification.

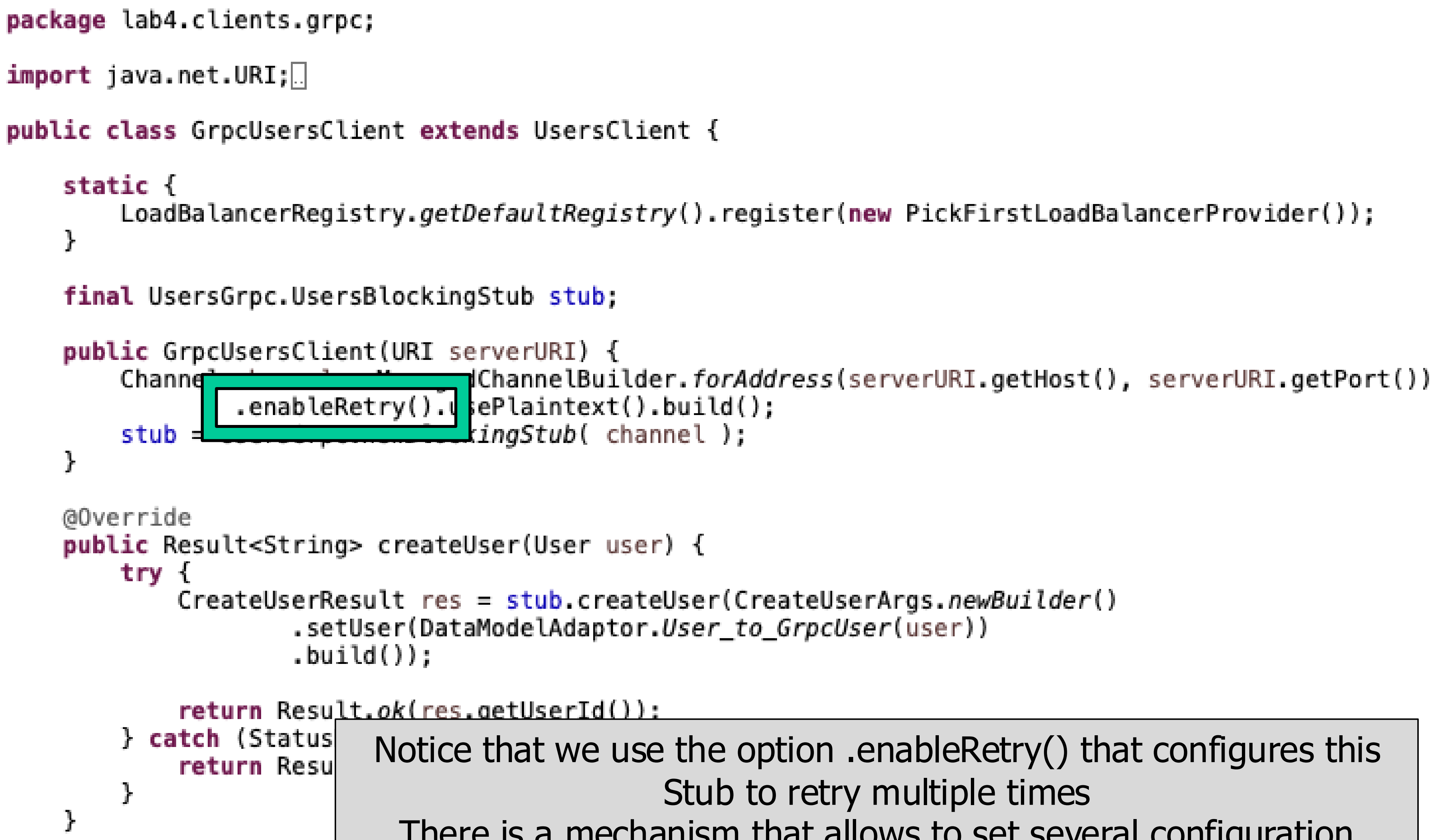
gRPC clients in Java



```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();

    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (Status
            return Resu
        }
    }
```

To instantiate this stub, we will need to define a channel.
This channel needs to have information about the machine where the server is running and the port (extracted from the URI) and is configured to use plaintext (not encrypted).

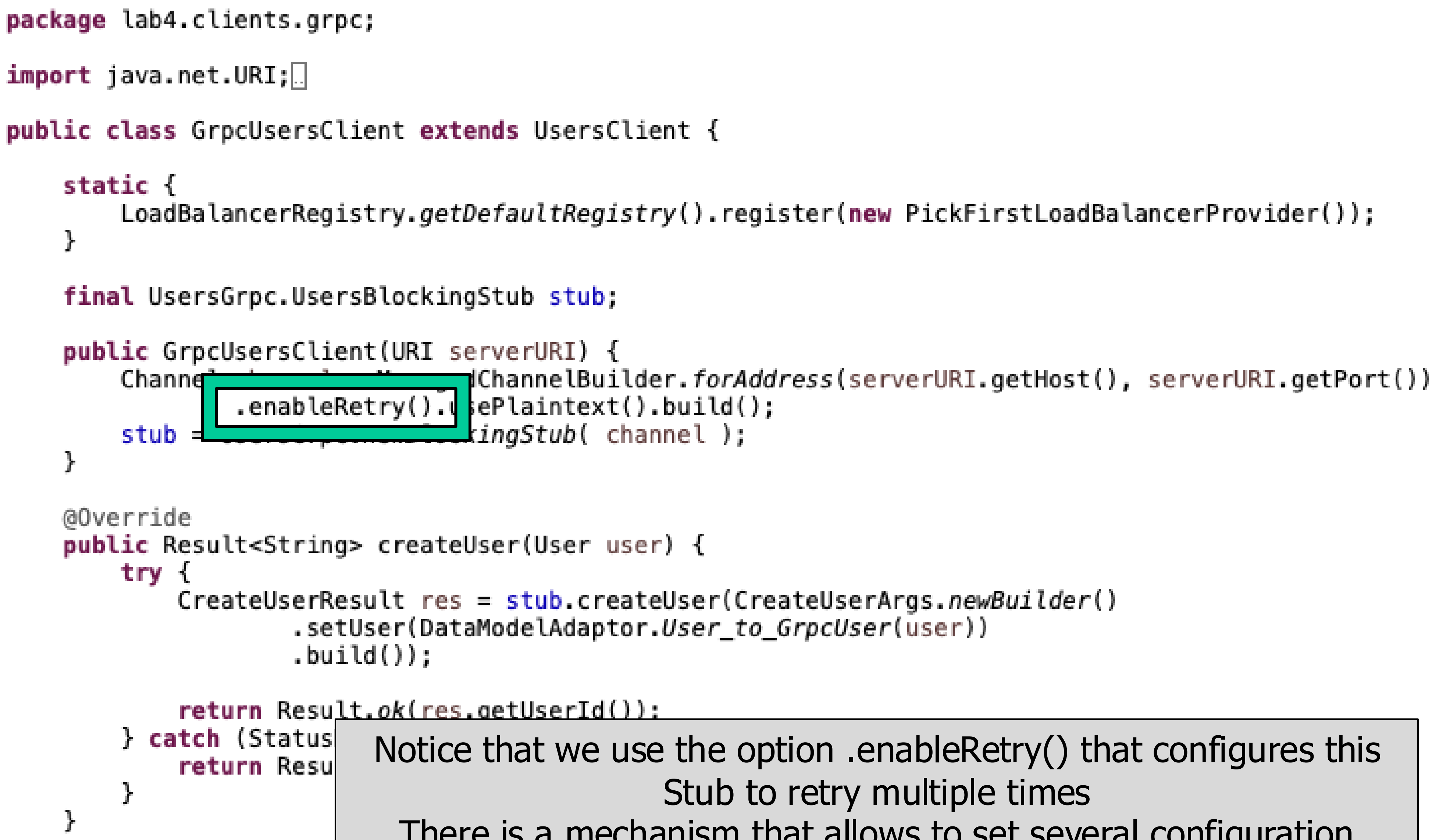
gRPC clients in Java



```
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channe                         dChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub =                          ingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (Status
            return Resu
        }
    }
```

Notice that we use the option .enableRetry() that configures this Stub to retry multiple times
There is a mechanism that allows to set several configuration variables that deal with the retry mechanism.

gRPC clients in Java

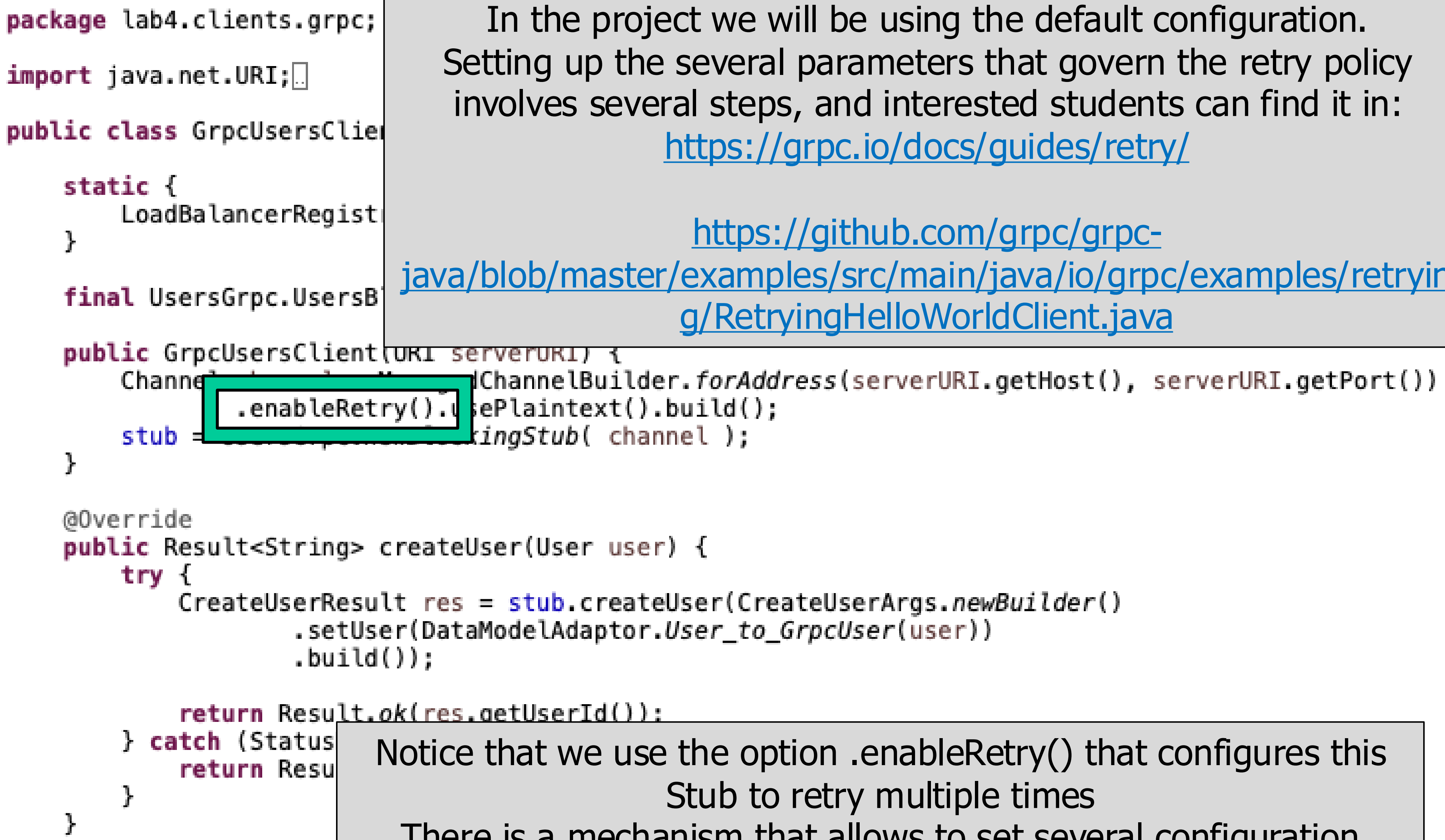


```
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channe                 dChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
               .enableRetry().u ePlaintext().build();
        stub =                 ingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (Status
            return Resu
        }
    }
```

Notice that we use the option .enableRetry() that configures this Stub to retry multiple times

There is a mechanism that allows to set several configuration variables that deal with the retry mechanism.

gRPC CLIENTS IN JAVA



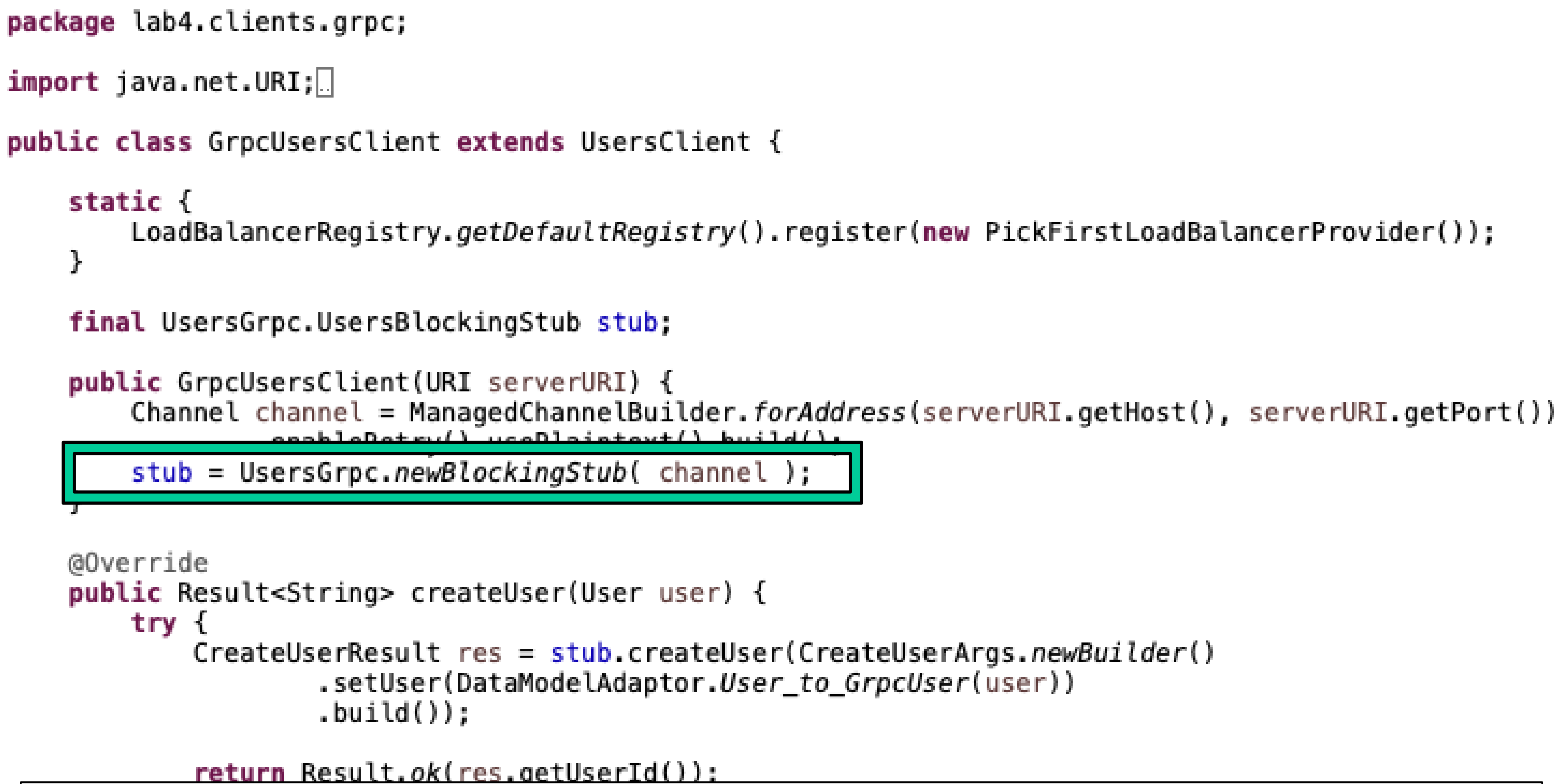
```
package lab4.clients.grpc;

import java.net.URI;□

public class GrpcUsersClie

    static {
        LoadBalancerRegist
    }

    final UsersGrpc.UsersB

    public GrpcUsersClient(URI serverURI) {
        Channe                        dChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
               .enableRetry().usePlaintext().build();
        stub =                        ingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (Status
            return Resu
        }
    }
```

In the project we will be using the default configuration.
Setting up the several parameters that govern the retry policy involves several steps, and interested students can find it in:
https://grpc.io/docs/guides/retry/

https://github.com/grpc/grpc-java/blob/master/examples/src/main/java/io/grpc/examples/retrying/RetryingHelloWorldClient.java

Notice that we use the option .enableRetry() that configures this Stub to retry multiple times
There is a mechanism that allows to set several configuration variables that deal with the retry mechanism.

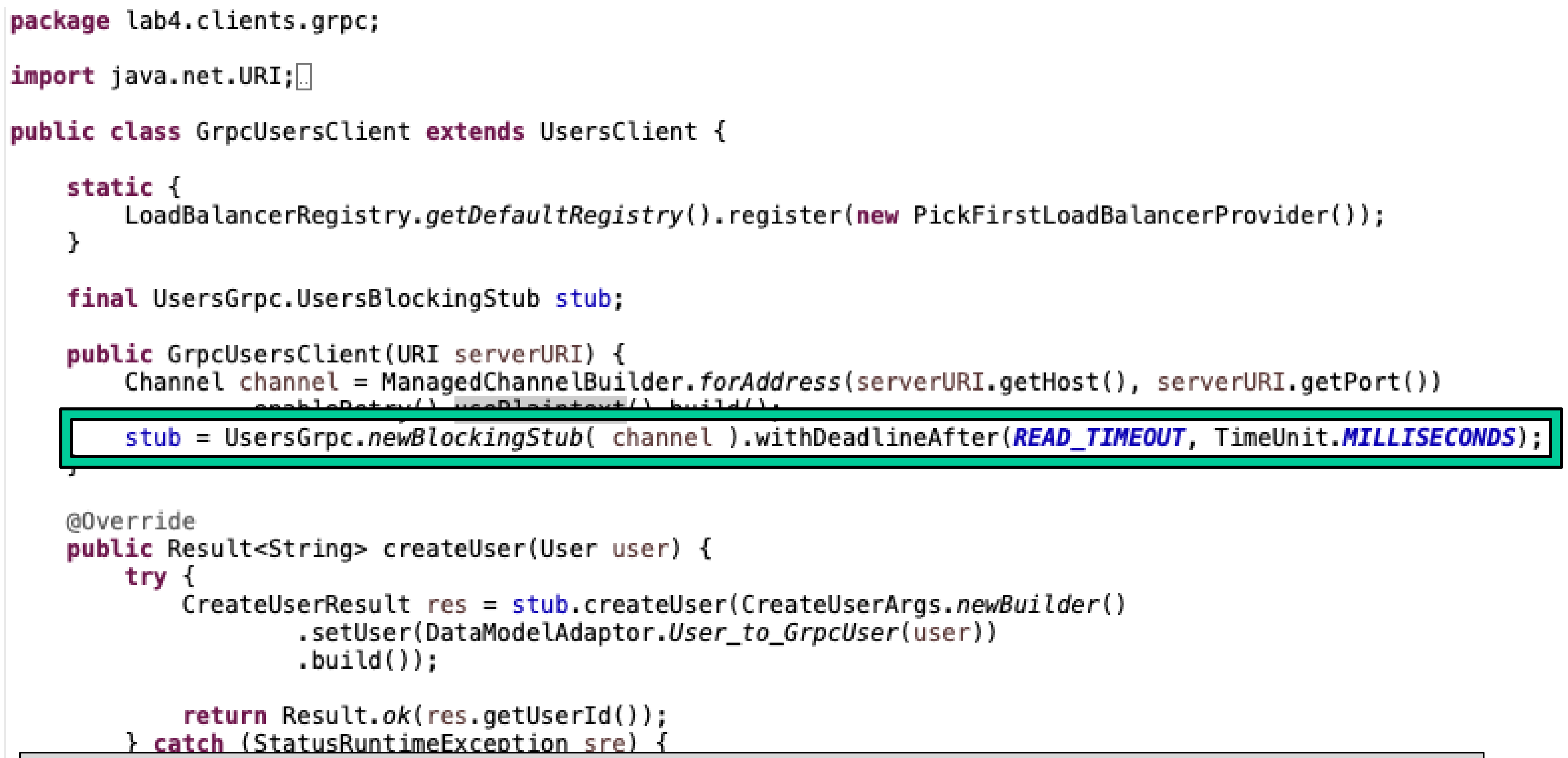
gRPC clients in Java



```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
```

With the configuration explained before gRPC stubs already handle the execution of requests multiple times when a transient failure makes it impossible to obtain a response within a configurable timeout, with a backoff time to wait between attempts.

gRPC clients in Java



```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel ).withDeadlineAfter(READ_TIMEOUT, TimeUnit.MILLISECONDS);
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (StatusRuntimeException sre) {
```

The configuration **withDeadlineAfter** shown here allows to have a somewhat similar behavior with a big caveat: the stub that is created will only retry operations until a deadline that is define to be READ_TIMEOUT milliseconds after the creation of the stub. Using this the stub must be recreated before executing each remote operation. **Hence, we will not be using this in the project!!!**

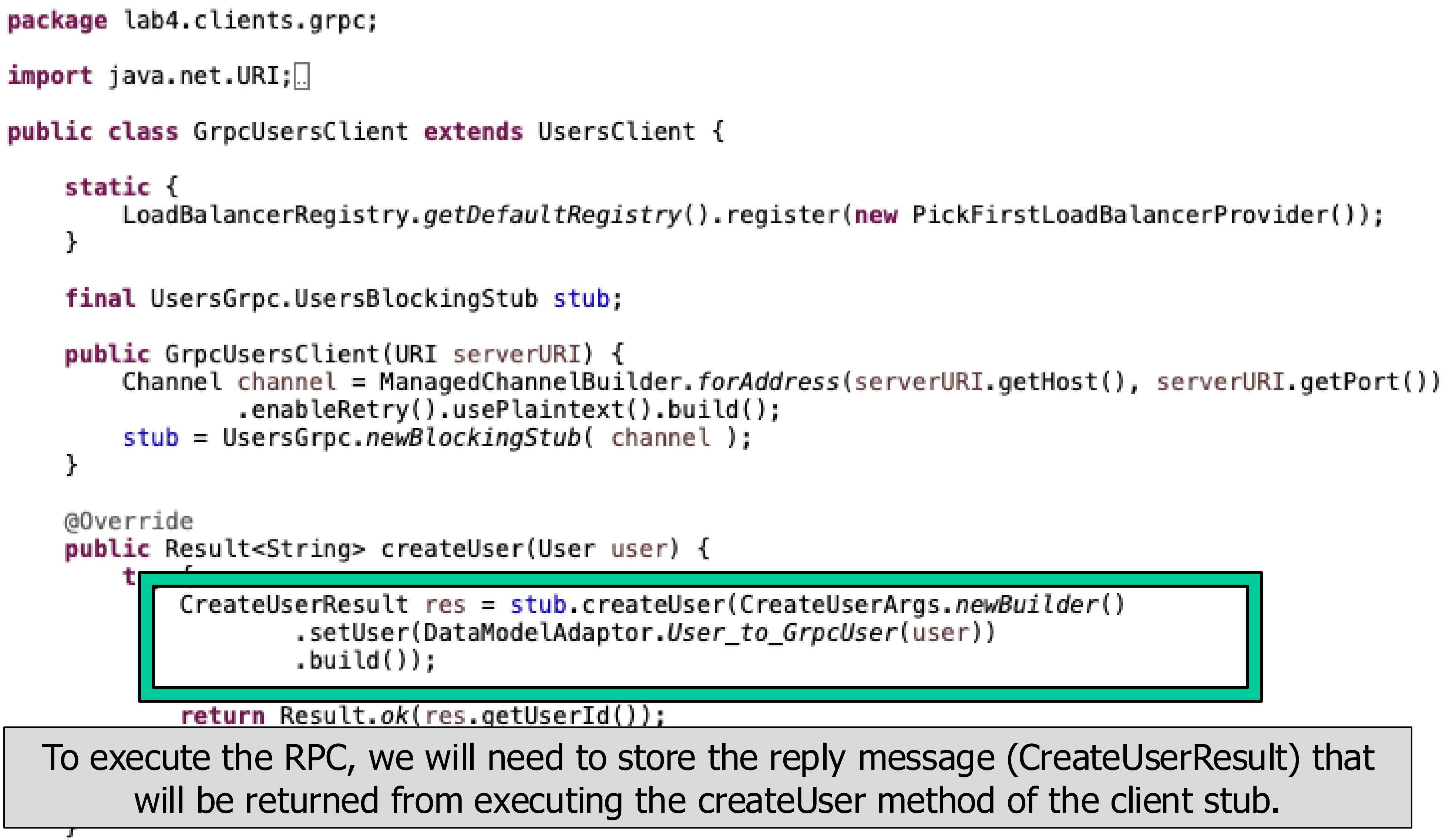
gRPC clients in Java

```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    fi
```

This method shows how to execute the remote RCP to create a user (receiving a User as an argument).

```java
    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (StatusRuntimeException sre) {
            return Result.error( statusToErrorCode(sre.getStatus()));
        }
```

gRPC CLIENTS IN JAVA



```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        t
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
```

To execute the RPC, we will need to store the reply message (CreateUserResult) that will be returned from executing the createUser method of the client stub.

gRPC CLIENTS IN JAVA

```
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends Users

    static {
        LoadBalancerRegistry.getDefaultReg
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        t
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
```

We must create the appropriate message (CreateUserArgs) using its builder and setting up the relevant elements of the message.

Notice that we again must convert from the User representation to GrpcUser, and we again use the DataModelAdaptor class.

To execute the RPC, we will need to store the reply message (CreateUserResult) that will be returned from executing the createUser method of the client stub.

gRPC clients in Java

```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
```

If the remote execution is successful, we just need to encapsulate the answer of the server (within the CreateUserResult) into a Result instance and return it.

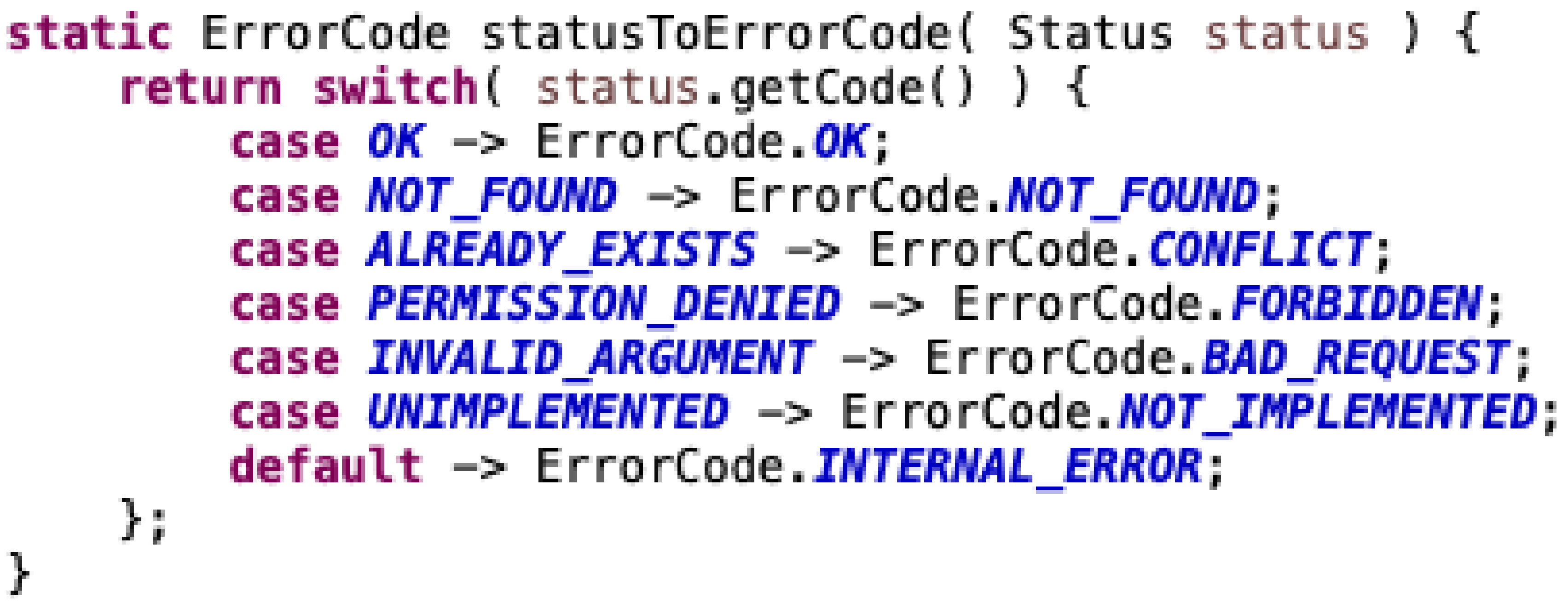
```java
    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (StatusRuntimeException sre) {
            return Result.error( statusToErrorCode(sre.getStatus()));
        }
    }
```

gRPC clients in Java

```
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final User

    public Grp
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        try {
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            return Result.ok(res.getUserId());
        } catch (StatusRuntimeException sre) {
            return Result.error( statusToErrorCode(sre.getStatus()));
        }
    }
```

If an error happens, a StatusRuntimeException is thrown, whose status will encode the nature of the error… we are using an auxiliary function to translate this error to a set of generic errors encoded in the Result message.

gRPC clients in Java: Converting Status Errors



```java
static ErrorCode statusToErrorCode( Status status ) {
    return switch( status.getCode() ) {
        case OK -> ErrorCode.OK;
        case NOT_FOUND -> ErrorCode.NOT_FOUND;
        case ALREADY_EXISTS -> ErrorCode.CONFLICT;
        case PERMISSION_DENIED -> ErrorCode.FORBIDDEN;
        case INVALID_ARGUMENT -> ErrorCode.BAD_REQUEST;
        case UNIMPLEMENTED -> ErrorCode.NOT_IMPLEMENTED;
        default -> ErrorCode.INTERNAL_ERROR;
    };
}
```

gRPC clients in Java

```java
package lab4.clients.grpc;

import java.net.URI;

public class GrpcUsersClient extends UsersClient {

    static {
        LoadBalancerRegistry.getDefaultRegistry().register(new PickFirstLoadBalancerProvider());
    }

    final UsersGrpc.UsersBlockingStub stub;

    public GrpcUsersClient(URI serverURI) {
        Channel channel = ManagedChannelBuilder.forAddress(serverURI.getHost(), serverURI.getPort())
                .enableRetry().usePlaintext().build();
        stub = UsersGrpc.newBlockingStub( channel );
    }

    @Override
    public Result<String> createUser(User user) {
        t
            CreateUserResult res = stub.createUser(CreateUserArgs.newBuilder()
                    .setUser(DataModelAdaptor.User_to_GrpcUser(user))
                    .build());

            re
        } catc
            re
        }
    }
```

In the case of this RPC we are sure that at most we will receive a single response in the form of a CreateUserResult because that is what we specified in the gRPC Service.

```
rpc createUser( CreateUserArgs ) returns (CreateUserResult) {}
```

gRPC clients in Java

```
rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
```

What about the case where the response is a stream of messages?

gRPC clients in Java

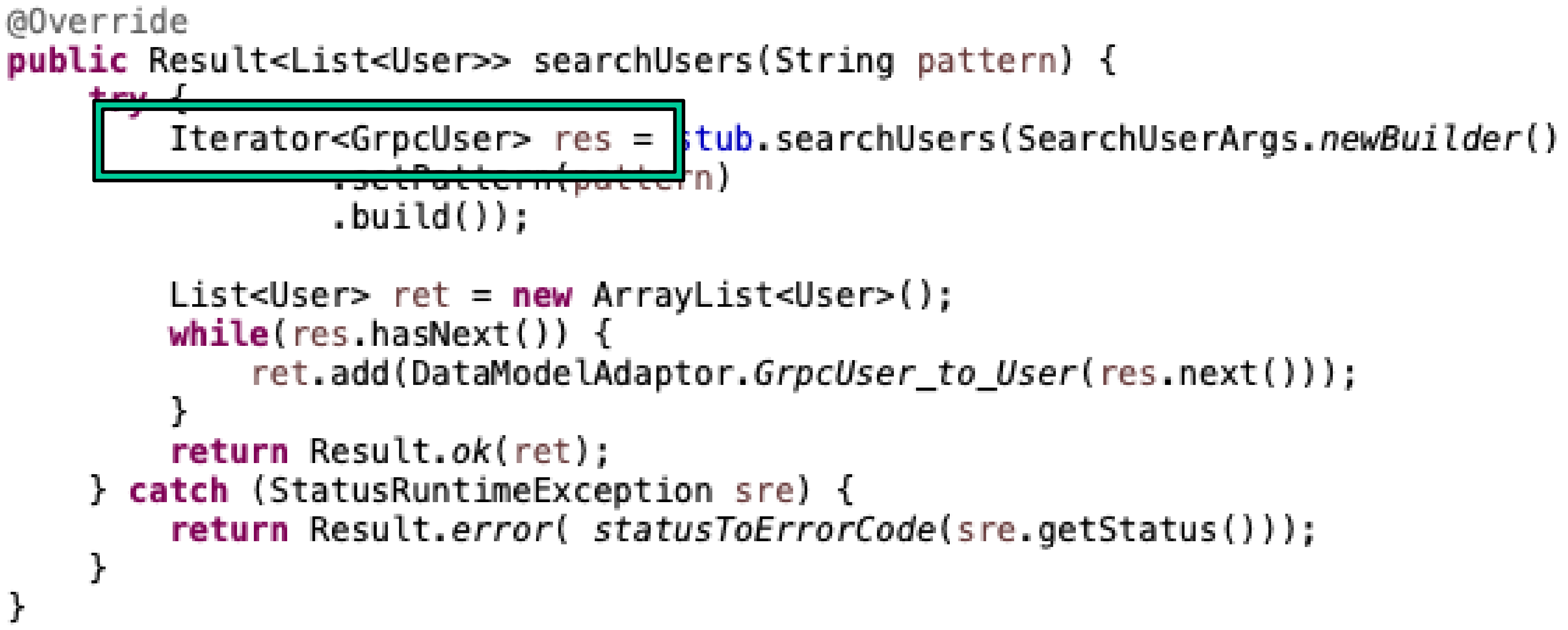
```
rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
```

```
@Override
public Result<List<User>> searchUsers(String pattern) {
    try {
        Iterator<GrpcUser> res = stub.searchUsers(SearchUserArgs.newBuilder()
                .setPattern(pattern)
                .build());

        List<User> ret = new ArrayList<User>();
        while(res.hasNext()) {
            ret.add(DataModelAdaptor.GrpcUser_to_User(res.next()));
        }
        return Result.ok(ret);
    } catch (StatusRuntimeException sre) {
        return Result.error( statusToErrorCode(sre.getStatus()));
    }
}
```

What about the case where the response is a stream of messages?

gRPC clients in Java

```
rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
```



```
@Override
public Result<List<User>> searchUsers(String pattern) {
    try {
        Iterator<GrpcUser> res = stub.searchUsers(SearchUserArgs.newBuilder()
                .setPattern(pattern)
                .build());

        List<User> ret = new ArrayList<User>();
        while(res.hasNext()) {
            ret.add(DataModelAdaptor.GrpcUser_to_User(res.next()));
        }
        return Result.ok(ret);
    } catch (StatusRuntimeException sre) {
        return Result.error( statusToErrorCode(sre.getStatus()));
    }
}
```

In that case the client stub that is generated will return an iterator for the type of Message that can be returned by the server.

gRPC clients in Java

```
rpc searchUsers( SearchUserArgs) returns (stream GrpcUser){}
```

```
@Override
public Result<List<User>> searchUsers(String pattern) {
    try {
        Iterator<GrpcUser> res = stub.searchUsers(SearchUserArgs.newBuilder()
                .setPattern(pattern)
                .build());
        List<User> ret = new ArrayList<User>();
        while(res.hasNext()) {
            ret.add(DataModelAdaptor.GrpcUser_to_User(res.next()));
        }
        return Result.ok(ret);
    }
        return Result.error( statusToErrorCode(sre.getStatus()));
    }
}
```

In this case handling this response requires iterating over the several elements of the response and doing something for each one util there are no more values accessible in the interator.
(in this case store each returned User in a List)

EXERCISE

1. Test the provided code to make sure that you can execute gRPC servers and clients.
2. Complete the logic of the gRPC server for the UsersService and complete create the clients that are missing to test the code (you can rely on code from previous weeks with relevant adaptions).
3. Based on the provided example and this explanation, create the gRPC server for the Image service (if you want, instead of the simplified specification in the lab4 materials, you can use the specification from the first project).
4. Start the first project (you can freely adapt code that was provided, and that you did in this and previous labs)