gRPC Workshop for Java Developers

Self-link: bit.ly/grpc-java-lab-doc

Author: Ray Tsang @saturnism, Ryan Knight (@knight_cloud)

Prerequisites

- Git - https://git-scm.com/downloads
- OpenJDK 8 - http://openjdk.java.net/install/
- Maven 3+ - https://maven.apache.org/install.html
- A Good Java IDE - IntelliJ or Eclipse
- We'll be writing Java code!

Getting the Code

The code is on GitHub, and checkout the initial branch:

```
$ git clone https://github.com/saturnism/grpc-java-chatroom-workshop.git
$ cd grpc-java-chatroom-workshop
$ git checkout initial
```

Make sure everything compiles out of the box!

```
$ mvn package
$ mvn install
```

Introduction

In this lab, we'll be build a real-time chat client and server w/ gRPC. It's typically fairly easy to do in a matter of 10 minutes or so, for a single room server.

Exercise 1 - Basics

In this exercise, we'll use gRPC to create a simple Authentication service. Given a username, password pair, the authentication service will validate and if successful, return a JWT token.

Create a Proto File

First, define message payloads and RPC operations for the authentication service. We'll have two RPC operations:

Operation	Input	Output
authenticate	username, password	Check the username/password pair against the entry in the user repository. If it matches, return a signed JWT token.

		If not, return Status.UNAUTHENTICATED error.
authorization	token	If the token is valid, retrieve the user, and return a list of associated roles. If not, user was not found, return Status.NOT_FOUND error. If JWT was not verifiable, return Status.UNAUTHENTICATED error.

Create / open the proto file: `auth-service/src/main/proto/AuthService.proto`.

Initialize the Proto file

1. See that the Protobuffer syntax to `proto3` to ensure we are using Protobuffer 3 syntax.
2. Specify the `package`. The package corresponds to the Java packages.
3. Set protoc options with `option` keyword. By default, the generator will generate all of the message classes into a single giant class file. In this example, we'll generate each of the Protobuffer message into its own class by setting the `java_multiple_files` option.

```
// 1. Set the syntax, package, and options
syntax = "proto3";
option java_multiple_files = true;
package com.example.auth;
```

Define messages

After defining the syntax, package, and options, define the message payloads. Notice that every field is strongly typed. There is a variety of out of the box primitives:

- Primitives such as `string`, `int32`, `int64`, `bool`, etc.
- To represent an array, use `repeated`, e.g., `repeated string`
- Strongly typed map/hash, e.g., `map<string, int64>`
- Custom types, such as
    - Nested types
    - Re-use defined types
    - Enumerations, e.g. `enum Sentiment { HAPPY = 0; SAD = 1; }`

Every field must be assigned a tag number (e.g., `string username = 1`). The tag number is what uniquely identifies the field rather than the name of the field. When serialized, the tag number is stored in the serialized payload and transferred across the wire.

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

// 2. Add the message definitions
message AuthenticationRequest {
```

```
    string username = 1;
    string password = 2;
}

message AuthenticationResponse {
    string token = 1;
}

message AuthorizationRequest {
    string token = 1;
}

message AuthorizationResponse {
    string username = 1;
    repeated string roles = 2;
}
```

Define service and operations

After defining the message payload, define the service, and operations within the service. Every operation must have an input type and an output type.

For the Authentication service, we'll define an unary request, which is basically a simple request/response:

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

message AuthenticationRequest {
    ...
}
...
// 3. Add the service definitions
service AuthenticationService {
    rpc authenticate(AuthenticationRequest) returns (AuthenticationResponse);
    rpc authorization(AuthorizationRequest) returns (AuthorizationResponse);
}
```

Configure Maven

Add the Maven dependencies

In `auth-service/pom.xml`, there are the gRPC dependencies configured:

```
<project>
    <dependencies>
        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-netty</artifactId>
        </dependency>
        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-protobuf</artifactId>
```

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

Note: In this example, the gRPC versions are specified in the parent POM's `dependencyManagement` section.

Configure the plugins

There are 2 plugins you must configure in order to automatically translate the proto file into Java code. First is the `os-maven-plugin`. This plugin will automatically detect the operation system and architecture of where the build is taking place, and makes the value available in `${os.detected.classifier}` variable. This is needed for the second plugin, `protobuf-maven-plugin` to download the native `protoc` generator. Native protoc generator is needed to translate proto file into Java code (or any other supported languages).

```
<project>
    <dependencies>
        ...
    </dependencies>
    ...
    <build>
        <extensions>
            <extension>
                <groupId>kr.motd.maven</groupId>
                <artifactId>os-maven-plugin</artifactId>
            </extension>
        </extensions>
        <plugins>
            <plugin>
                <groupId>org.xolstice.maven.plugins</groupId>
                <artifactId>protobuf-maven-plugin</artifactId>
            </plugin>
            ...
        </plugin>
    </build>
</project>
```

Note: The plugins are also managed by the parent POM, in its `pluginManagement` section. There is much more to the configuration of `protobuf-maven-plugin`. In addition to the plugin configuration to download the right version of the `protoc` generator, it also ties the generation into Maven's `compile` and

`compile-custom` phases, so that the files are auto-generated whenever you recompile your code. See the parent POM for detail.

There is also a Gradle plugin!

Generate Stubs

Once you have the proto file and the Maven plugin configured, generate the Java class files from Maven

```
$ cd auth-service/
$ mvn install
```

Important: Because the service package is depended by other modules within the project, it's important to use `mvn install` to install the artifact so it can be referenced.

All Protobuffer 3 messages gRPC stubs should be generated under `target/generated-sources`:

```
$ find target/generated-sources
```

Implement Server Stub

Open `auth-service/src/main/java/com/example/auth/grpc/AuthServiceImpl.java`. In this file, we need to:

1. Extend from the gRPC generated `AuthServiceBaseImpl` class
2. Implement the `authenticate` and `authorization` methods

Extend the Base Implementation

```
...
// TODO Extend gRPC's AuthenticationServiceBaseImpl
public class AuthServiceImpl extends
AuthenticationServiceGrpc.AuthenticationServiceImplBase  {
  ...
}
```

Override and Implement the Methods

The base implementation bootstraps the underlying descriptors and pipes necessary for gRPC server to communicate with the actual service implementation, and makes the call to the actual service methods. The service methods have default implementations in the base class, but they will all throw `Status.UNIMPLEMENTED` error. You'll need to override these methods explicitly:

First, override the `authenticate()` base implementation method:

```
// TODO Override authenticate method
@Override
```

```
public void authenticate(AuthenticationRequest request,
StreamObserver<AuthenticationResponse> responseObserver) {
...
}
```

Notice that even though we defined a simple unary call (request/response), the server implementation is fully asynchronous. That means, to return the value or errors to the client, you must use `responseObserver`.

1. Use `UserRepository` to retrieve the user based on the username.

```
User user = repository.findUser(request.getUsername());
```

2. If user doesn't exist do the following steps to return an error:
    a. Return a `Status.UNAUTHENTICATED` error using `responseObserver.onError(...)`
    b. Wrap the status as a `StatusRuntimeException`. There is `Status.UNAUTHENTICATED.asRuntimeException()` convenience method to do that.
    c. Close the stream with `responseObserver.onError(...)` will close the stream.
    d. Immediately return and avoid calling any other `responseObserver` callbacks afterwards.
3. Similarly, if user exists, but the password doesn't match, also return `Status.UNAUTHENTICATED` error using `responseObserver.onError(...)`

```
if (user == null || !user.getPassword().equals(request.getPassword())) {
  responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());
  return;
}
```

4. Finally, if all things goes well, generate a JWT token by calling `generateToken(...)`, construct `AuthenticationResponse` with the token, then return that using `responseObserver.onNext(...)`. You must call `responseObserver.onCompleted()` to completed the call. Otherwise, the client will wait until that is called. If you never call it - then the client will wait forever.

```
String token = generateToken(request.getUsername());
responseObserver.onNext(AuthenticationResponse.newBuilder()
  .setToken(token)
  .build());
responseObserver.onCompleted();
```

Hint: Full implementation in solution.

Second, override the `authorize()` method:

```
// TODO Override authorization method
```

```
public void authorization(AuthorizationRequest request,
StreamObserver<AuthorizationResponse> responseObserver) {
  ...
}
```

1. Use `jwtFromToken(...)` to verify the token

```
DecodedJWT jwt = jwtFromToken(request.getToken());
```

2. Catch `JWTVerificationException`. If this exception was thrown, return Status.UNATHORIZED error using `responseObserver.onError(...)`

```
try {
  DecodedJWT jwt = jwtFromToken(request.getToken());
} catch (JWTVerificationException e) {
  responseObserver.onError(Status.UNAUTHENTICATED.asRuntimeException());
  return;
}
```

3. If JWT token is valid, a `DecodedJWT` object will be returned. From there, extract the username using getSubject(). If the user is not found return a `Status.NOT_FOUND` using `responseObserver.onError(...)`

```
String username = jwt.getSubject();
User user = repository.findUser(username);

if (user == null) {
  // send error
  return;
}
```

4. Use `UserRepository` to retrieve the user, construct `AuthorizationResponse` with the roles, then return that using `responseObserver.onNext(...)`

```
responseObserver.onNext(AuthorizationResponse.newBuilder()
  .addAllRoles(user.getRoles())
  .build());
responseObserver.onCompleted();
```

Hint: Full implementation in solution.

Write the Server main entry

In `auth-service/src/main/java/AuthServer.java`'s main() method:

1. Use `ServerBuilder` to build a new server that listens on port 9091: `ServerBuilder.forPort(9091)`
2. Add an instance of `AuthServiceImpl` as a service: `addService(...).build;`

```
// TODO Use ServerBuilder to create a new Server instance. Start it, and await termination.
// Algorithm and "auth-issuer" are used for JWT signing and verification.
Algorithm algorithm = Algorithm.HMAC256("secret");
Server server = ServerBuilder.forPort(9091)
  .addService(new AuthServiceImpl(repository, "auth-issuer", algorithm))
  .build();
```

3. Add a shutdown hook to shutdown the server in case the process receives shutdown signal, and call `server.shutdownNow()`. This will try to shutdown the server gracefully, if shutdown hook is called.

```
Runtime.getRuntime().addShutdownHook(new Thread() {
  @Override
  public void run() {
    server.shutdownNow();
  }
});
```

4. Start the server: `server.start()`

```
server.start();
logger.info("Server started on port 9091");
```

5. Server will be running in a background thread. Call `server.awaitTermination()` to block the main thread

```
server.awaitTermination();
```

Hint: Full implementation in solution.

Run the Server

The project has exec-maven-plugin configured to run the main class. To run the server:

```
$ mvn install exec:java
...
INFO: Server started on port 9091
```

Not bad for the first gRPC service!

Testing from the Client

Let's test it from a client.

Open `chat-cli-client/src/main/java/com/example/chat/ChatClient.java`
First, implement `initAuthService()` method in `ChatClient`:

1. Use a ManagedChannelBuilder to create a new ManagedChannel and assign it to `authChannel`:

```
// TODO Build a new ManagedChannel
authChannel = ManagedChannelBuilder.forTarget("localhost:9091")
  .usePlaintext(true)
  .build();
```

2. Instantiate a new blocking stub and assign it to authService:

```
// TODO Get a new Blocking Stub
authService = AuthenticationServiceGrpc.newBlockingStub(authChannel);
```

Next implement the `authenticate(...)` method in `ChatClient`. This method will be called when you start the client and initiate the login process:

1. Call `authService.authenticate(...)` to retrieve the token

```
// TODO Call authService.authenticate(...) and retreive the token
AuthenticationResponse authenticationReponse =
authService.authenticate(AuthenticationRequest.newBuilder()
  .setUsername(username)
  .setPassword(password)
  .build());

String token = authenticationReponse.getToken();
```

2. Once retrieved the token, call `authService.authorization(...)` to retrieve all roles, and print them out

```
// TODO Retrieve all the roles with authService.authorization(...) and print
out all the roles
AuthorizationResponse authorizationResponse =
authService.authorization(AuthorizationRequest.newBuilder()
  .setToken(token)
  .build());
logger.info("user has these roles: " + authorizationResponse.getRolesList());
```

3. Finally, return the token

```
// TODO Return the token
```

```
return token;
```

4. If any `StatusRuntimeException` is caught, handle it, print the error, and then return `null`. Do this by wrapping the call to authenticate and token processing in a try catch block:

```
try {
  ...
} catch (StatusRuntimeException e) {
  // TODO Catch StatusRuntimeException, because there could be Unauthenticated
errors.
  if (e.getStatus().getCode() == Status.Code.UNAUTHENTICATED) {
    logger.log(Level.SEVERE, "user not authenticated: " + username, e);
  } else {
    logger.log(Level.SEVERE, "caught a gRPC exception", e);
  }
  // TODO If there are errors, return null
  return null;
}
```

Try it out

Leave the authentication service running in the first terminal window and open a separate terminal window to run the client. Login as the `admin` user, and you should see the admin has roles `admin` and `user`. Be sure your authentication

```
$ mvn install exec:java
...
-> /login admin
Jun 28, 2017 3:28:31 PM com.example.chat.ChatClient readLogin
INFO: processing login user
password> qwerty
Jun 28, 2017 3:28:32 PM com.example.chat.ChatClient authenticate
INFO: authenticating user: admin
Jun 28, 2017 3:28:33 PM com.example.chat.ChatClient authenticate
INFO: user has these roles: [admin, user]
Jun 28, 2017 3:28:33 PM com.example.chat.ChatClient initChatServices
INFO: initializing chat services with token:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiIsImlzcyI6ImF1dGgtaXNzd
WVyIn0.oQebRc48QIig1R5-JajeTCP3TWP1fThBilNpoOtKavA
[chat message] | /join [room] | /leave [room] | /create [room] | /list | /quit
-> /quit
```

Login with an invalid credential, and you should see a `StatusRuntimeException` with `Status.UNAUTHENTICATED` error:

```
$ cd chat-cli-client
$ mvn install exec:java
...
-> /login kevin
```

```
Jun 28, 2017 3:30:46 PM com.example.chat.ChatClient readLogin
INFO: processing login user
password> asdf
Jun 28, 2017 3:30:48 PM com.example.chat.ChatClient authenticate
INFO: authenticating user: kevin
Jun 28, 2017 3:30:48 PM com.example.chat.ChatClient authenticate
SEVERE: user not authenticated: kevin
io.grpc.StatusRuntimeException: UNAUTHENTICATED
     at
io.grpc.stub.ClientCalls.toStatusRuntimeException(ClientCalls.java:227)
-> /quit
```

Important: Once logged into the chat server all chat commands are started with / followed by the command name. Enter /? To see all possible chat commands.

Deadlines

A frontend often need to make multiple backends calls, or, making backend calls that calls other backends. If you know the performance characteristics of your services, or if you are able to drop requests that takes too long to finish, and return to user something else, then you should explore deadlines.

In gRPC, you can set a deadline to a call, e.g., no more than 1 second in total. When the deadline has exceeded, it'll throw a `StatusRuntimeException` with `Status.DEADLINE_EXCEEDED`. The deadline will propagate across all subsequent nested calls too. E.g., if you set a deadline to 1 second, and the first call took 0.7s, and if that call makes another nested call, then the nested call would have 0.3s to complete.

We are not going to be using deadlines in this sample but the following is an example of how this would be done. To set a deadline is simple, for example, you can decorate a stub with deadline:

```
authService.withDeadlineAfter(1, TimeUnit.SECONDS);
  .authenticate(AuthenticationRequest.newBuilder()
  .setUsername(username)
  .setPassword(password)
  .build());
```

On the server side, if the deadline was exceeded, then the server will also receive a notification. You can listen to call cancellations by attaching a listener to the Context:

```
Context.current().addListener(new Context.CancellationListener() {
  @Override
  public void cancelled(Context context) {
    System.out.println("Call was cancelled!");
  }
}, Executors.newCachedThreadPool());
```

Exercise 2 - Metadata, Context, and Interceptors

Metadata

You can use Metadata to pass additional request information, such as a authentication token, or a trace ID, across network boundary to another service. Effectively, all metadata key/value pairs are transmitted via HTTP/2 headers. There is no direct access to HTTP/2. Metadata is the abstraction for referring to the header.

Define a metadata
All Metadata are strongly typed and keyed. That means, rather than using a string as a key you pass into a Map, you have to create a Metadata Key.

Open `chat-service/src/main/java/com/example/chat/grpc/Constant.java` to define a new metadata for transferring JWT token between services:

```
// TODO Add a JWT_METADATA_KEY
public static final Metadata.Key<String> JWT_METADATA_KEY =
  Metadata.Key.of("jwt", ASCII_STRING_MARSHALLER);
```

Interceptor

Capture Metadata from Server Interceptor
On the server side, metadata can only be captured from a server interceptor. Open `chat-service/src/main/java/com/example/chat/grpc/JwtServerInterceptor.java`.

Capture the JWT token and just print it out for now. We will implement the token processing in later exercises:

```
public class JwtServerInterceptor implements ServerInterceptor {
  ...

  @Override
  public <ReqT, RespT> ServerCall.Listener<ReqT>
interceptCall(ServerCall<ReqT, RespT> serverCall, Metadata metadata,
ServerCallHandler<ReqT, RespT> serverCallHandler) {
    // TODO Get token from Metadata
    String token = metadata.get(Constant.JWT_METADATA_KEY);
    System.out.println("Token: " + token);

    return serverCallHandler.startCall(serverCall, metadata);
  }
}
```

Attach Server Interceptor
Open `chat-service/src/main/java/com/example/chat/grpc/ChatServer.java`. The `JwtServerInterceptor` is already instantiated. However, we need to add it into the interceptor chain:

In the `ServerBuilder`, where we register existing service instances, we can modify it to apply interceptors for each of the service:

```
// TODO Add JWT Server Interceptor, then later, trace interceptor
final Server server = ServerBuilder.forPort(9092)
  .addService(ServerInterceptors
    .intercept(chatRoomService, jwtServerInterceptor))
  .addService(ServerInterceptors
    .intercept(chatStreamService, jwtServerInterceptor))
  .build();
```

Now, every request sent to these services will first be intercepted by the `JWTServerInterceptor`.

In a separate terminal window now run the server. You should now have 3 terminal windows with the authentication service, the chat service and the client.

```
$ cd chat-service
$ mvn install exec:java
```

Passing Metadata from a Client

There are several ways to attach metadata to a call from the client. We'll illustrate the easiest ways first, and move to more sophisticated ways in later parts of the lab.

Decorate with MetadataUtils

In `ChatClient.initChatServices()`, we instantiated the `chatRoomService` stub. We can decorate the stub using `MetadataUtils` to attach additional headers:

```
public void initChatService() {
  …
  Metadata metadata = new Metadata();
  metadata.put(Constant.JWT_METADATA_KEY, token);

  chatRoomService = MetadataUtils.attachHeaders(
    ChatRoomServiceGrpc.newBlockingStub(chatChannel), metadata);
  chatStreamService = ChatStreamServiceGrpc.newStub(chatChannel);
}
```

Now, every outgoing call from `chatRoomService` will have the JWT metadata attached. Let's try it by running the client, and login as `admin` user, and then create a new room:

```
$ cd chat-cli-client
$ mvn install exec:java
-> /login admin
Jun 28, 2017 4:45:47 PM com.example.chat.ChatClient readLogin
INFO: processing login user
password> qwerty
...
INFO: initializing chat services with token:
```

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiIsImlzcyI6ImF1dGgtaXNzd
WVyIn0.oQebRc48QIig1R5-JajeTCP3TWP1fThBilNpoOtKavA
[chat message] | /join [room] | /leave [room] | /create [room] | /list | /quit
admin-> /create test
```

On the server side, you should see the JWT token printed in the console:

```
INFO: Server Started on port 9092
Token: eyJ0eXAiOiJKV1QiLCJhbGci...
```

Metadata Client Interceptor

Alternatively, you can use a client interceptor to attach headers too. There is an out of the box interceptor to do just that. In `ChatServer.initChatServices()`, use the stub normally. But you can add the client interceptor to the `Channel`:

```
public void initChatService() {
  ...
  Metadata metadata = new Metadata();
  metadata.put(Constant.JWT_METADATA_KEY, token);

  // TODO Add JWT Token via a Call Credential
  chatChannel = ManagedChannelBuilder.forTarget("localhost:9092")
      .intercept(MetadataUtils.newAttachHeadersInterceptor(metadata))
      .usePlaintext(true)
      .build();

  chatRoomService = ChatRoomServiceGrpc.newBlockingStub(chatChannel);
  chatStreamService = ChatStreamServiceGrpc.newStub(chatChannel);
}
```

Context

Metadata is great to propagate metadata information across network boundary. However it is limited because it only text or something that be serialized into text. As well it is not directly accessible from server stubs. This is a problem for example if the server stub needs to get the JWT token for validation. It also doesn't work if you need to propagate certain values downstream to nested calls within the same JVM. More documentation on Context can be found at http://www.grpc.io/grpc-java/javadoc/io/grpc/Context.html

For those use cases, we need to use Context. Similar to Metadata, to use Context, we must declare the Context keys. Do this in `chat-service/src/main/java/com/example/chat/grpc/Constant.java`:

```
// TODO Add a JWT Context Key
public static final Context.Key<DecodedJWT> JWT_CTX_KEY = Context.key("jwt");

public static final Context.Key<String> USER_ID_CTX_KEY =
```

```
Context.key("userId");
```

Metadata/Context Propagation

Server Interceptor - Metadata to Context
Since the server interceptor can capture the Metadata, we can also use it to propagate the information into a Context variable.

Let's implement the full on JWT Interceptor so it will:

1. Capture the JWT token from Metadata
2. Verify that the token is valid
3. Converting the token into a DecodedJWT object, and store both the DecodedJWT and the User ID values into respective contexts.

Open `chat-service/src/main/java/com/example/chat/grpc/JwtServerInterceptor.java` and finish the implementation of the JWT Token processing:

```
public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCall(
  ServerCall<ReqT, RespT> serverCall, Metadata metadata,
  ServerCallHandler<ReqT, RespT> serverCallHandler) {

  // TODO Get token from Metadata
  String token = metadata.get(Constant.JWT_METADATA_KEY);

    // TODO If token is null, or is invalid, close the call with
Status.UNAUTHENTICATED
  if (token == null) {
    serverCall.close(Status.UNAUTHENTICATED
      .withDescription("JWT Token is missing from Metadata"), metadata);
    return NOOP_LISTENER;
  }

  try {
    DecodedJWT jwt = verifier.verify(token);
    Context ctx = Context.current()
        .withValue(Constant.USER_ID_CTX_KEY, jwt.getSubject())
        .withValue(Constant.JWT_CTX_KEY, jwt);
    return Contexts.interceptCall(ctx, serverCall, metadata, serverCallHandler);
  } catch (Exception e) {
    System.out.println("Verification failed - Unauthenticated!");
    serverCall.close(Status.UNAUTHENTICATED
      .withDescription(e.getMessage()).withCause(e), metadata);
    return NOOP_LISTENER;
  }
}
```

Note: The magic here is `Context ctx = Context.current().withValue(...)` to capture the context value, and subsequently, using `Contexts.interceptCall(...)` to propagate the context to the service implementation.

Client Interceptor - Context to Metadata

Similarly, you can propagate context value to another service over network boundary by converting the context value into a Metadata. You can do this in a Client Interceptor.

Open `chat-service/src/main/java/com/example/chat/grpc/JwtClientInterceptor.java`, and implement the `SimpleForwardingCall.start(...)` method.

```
public class JwtClientInterceptor implements ClientInterceptor {
  @Override
  public <ReqT, RespT> ClientCall<ReqT, RespT> interceptCall(...) {

    return new ForwardingClientCall
      .SimpleForwardingClientCall<ReqT, RespT>(...) {
      @Override
      public void start(...) {
        // TODO Convert JWT Context to Metadata header
        DecodedJWT jwt = Constant.JWT_CTX_KEY.get();
        if (jwt != null) {
          headers.put(Constant.JWT_METADATA_KEY, jwt.getToken());
        }
        super.start(responseListener, headers);
      }
    };
  }
}
```

Register client interceptor

You can attach a client interceptor to a channel similarly to how we attached the metadata interceptor. For example, in `ChatServer.initAuthService(...)`, we can attach this client interceptor to `authChannel`:

```
final ManagedChannel authChannel = ManagedChannelBuilder
  .forTarget("localhost:9091")
  .intercept(new JwtClientInterceptor())
  .usePlaintext(true)
  .build();
```

Enforce Role-based Access

Now that we have the User ID and JWT Token in Context, how can we access it from the server stubs? For example, to get the User ID, and check the roles?

Currently, anyone can create a new Chat Room. Let's lock it down so that only users with the `admin` role can perform those tasks. In `chat-service/src/main/java/com/example/chat/grpc/ChatRoomServiceImpl.java` implement the `ChatRoomServiceImpl.failBecauseNoAdminRole(...)`:

1. Get the Decoded JWT token from context

```
protected <T> boolean failBecauseNoAdminRole(StreamObserver<T>
responseObserver) {
    // TODO Retrieve JWT from Constant.JWT_CTX_KEY
    DecodedJWT jwt = Constant.JWT_CTX_KEY.get();
}
```

2. Use it to fetch the associated roles, and then validate if the user has the `admin` role

```
protected <T> boolean failBecauseNoAdminRole(StreamObserver<T>
responseObserver) {
    // TODO Retrieve JWT from Constant.JWT_CTX_KEY
    DecodedJWT jwt = Constant.JWT_CTX_KEY.get();

    // TODO Retrieve the roles
    AuthorizationResponse authorization = authService.authorization(
        AuthorizationRequest.newBuilder()
            .setToken(jwt.getToken())
            .build());

    // TODO If not in the admin role, return Status.PERMISSION_DENIED
    if (!authorization.getRolesList().contains("admin")) {
      responseObserver.onError(Status.PERMISSION_DENIED.
          withDescription("You don't have admin role").asRuntimeException());
      return true;
    }

    return false;
  }
```

Restart the server:

```
$ cd chat-service
$ mvn install exec:java
```

Then, try in client with a non-admin user:

```
$ cd chat-cli-client
$ mvn install exec:java
/login [username] | /quit
-> /login rayt
Jun 28, 2017 5:24:35 PM com.example.chat.ChatClient readLogin
```

```
INFO: processing login user
password> hello
...
[chat message] | /join [room] | /leave [room] | /create [room] | /list | /quit
rayt-> /create ray-room
[WARNING]
io.grpc.StatusRuntimeException: PERMISSION_DENIED: You don't have admin role
     at
io.grpc.stub.ClientCalls.toStatusRuntimeException(ClientCalls.java:227)
```

But, if you try it with the `admin` user:

```
$ cd chat-cli-client
$ mvn install exec:java
/login [username] | /quit
-> /login admin
Jun 28, 2017 5:24:35 PM com.example.chat.ChatClient readLogin
INFO: processing login user
password> qwerty
...
[chat message] | /join [room] | /leave [room] | /create [room] | /list | /quit
admin-> /create ray-room
Jun 28, 2017 5:26:42 PM com.example.chat.ChatClient createRoom
INFO: create room: ray-room
Jun 28, 2017 5:26:42 PM com.example.chat.ChatClient createRoom
INFO: created room: ray-room
```

Client Call Credential

Last but not least, there is a special way to apply metadata in the client to outgoing calls, that's designed specifically to append credentials information. For example, OAuth token might expire, and for each call, if the token expired, you can proactively refresh it, rather than being stuck to a static value.

A Call Credential example is already in the repository. Open `chat-cli-client/src/main/java/com/example/chat/grpc/JWTCallCredential.java`.

Every time a call takes place, this handler will execute, and you can use MetadataApplier to apply the headers. This method also should not block. If you need to refresh to token, or fetch the token from the network, it should be done asynchronously.

To use the Call Credential, open `ChatClient.initChatService(...)`:

1. Instantiate a `JWTCallCredential` object with the token
2. Use `stub.withCallCredential(...)` to decorate the stub

```
public void initChatServices(String token) {
    logger.info("initializing chat services with token: " + token);
```

```
    // TODO Add JWT Token via a Call Credential
    chatChannel = ManagedChannelBuilder.forTarget("localhost:9092")
        .usePlaintext(true)
        .build();

    JwtCallCredential callCredential = new JwtCallCredential(token);
    chatRoomService = ChatRoomServiceGrpc.newBlockingStub(chatChannel)
        .withCallCredentials(callCredential);
    chatStreamService = ChatStreamServiceGrpc.newStub(chatChannel)
        .withCallCredentials(callCredential);
}
```

Exercise 3 - Bidirectional Streaming

The true power of gRPC lies in its use of HTTP/2 as the underlying protocol as oppose to HTTP/1. In HTTP/2, streaming is a native concept, and you can stream from server to client, from client to server, and also in both directions. Moreover, multiple streams are multiplexed over the same connection to reduce TCP connection handshake overhead.

In this lab, we'll use bidirectional streaming to establish a chat session with the server.

Implement Chat Stream Service

Open `chat-service/src/main/java/com/example/chat/grpc/ChatStreamServiceImpl.java`. `ChatStreamServiceImpl.chat(...)` is the bidirectional stream stub we need to implement.

On the server side, we'll need to listen to incoming streamed messages. To do this, the server needs to return a StreamObserver, to listen to incoming messages:

```
@Override
public StreamObserver<ChatMessage> chat(
  StreamObserver<ChatMessageFromServer> responseObserver) {
  final String username = Constant.USER_ID_CTX_KEY.get();

  return new StreamObserver<ChatMessage>() {
    @Override
    public void onNext(ChatMessage chatMessage) {

    }

    @Override
    public void onError(Throwable throwable) {

    }

    @Override
    public void onCompleted() {
```

```
    }
  };
}
```

The `responseObserver` in the method parameter is what the server needs to use to stream data to the client.

The ChatMessage being streamed to the server may have different types, such as `JOIN` a room, `LEAVE` a room, or simply a `TEXT` message for a room:

1. When joining a chat room, add the `responseObserver` to the room's set of all currently connected observers.

```
@Override
public void onNext(ChatMessage chatMessage) {
  Set<StreamObserver<ChatMessageFromServer>> observers =
      getRoomObservers(chatMessage.getRoomName());
  switch (chatMessage.getType()) {
    case JOIN:
      observers.add(responseObserver);
      break;
  }
}
```

2. When leaving a chat room, remove the `responseObserver` from the room's observers set.

```
@Override
public void onNext(ChatMessage chatMessage) {
  Set<StreamObserver<ChatMessageFromServer>> observers =
      getRoomObservers(chatMessage.getRoomName());
  switch (chatMessage.getType()) {
    case JOIN:
      observers.add(responseObserver);
      break;
    case LEAVE:
      observers.remove(responseObserver);
      break;
  }
}
```

3. When sending a message, first make sure user is in the room, and then send to all the connected observers in the room:

```
@Override
public void onNext(ChatMessage chatMessage) {
  Set<StreamObserver<ChatMessageFromServer>> observers =
      getRoomObservers(chatMessage.getRoomName());
  switch (chatMessage.getType()) {
    case JOIN:
```

```
      observers.add(responseObserver);
      break;
    case LEAVE:
      observers.remove(responseObserver);
      break;
    case TEXT:
      if (!observers.contains(responseObserver)) {
        responseObserver.onError(
          Status.PERMISSION_DENIED.withDescription("You are not in the room " +
            chatMessage.getRoomName()).asRuntimeException());
        return;
      }
      Timestamp now = Timestamp.newBuilder()
          .setSeconds(new Date().getTime()).build();
      ChatMessageFromServer messageFromServer =
          ChatMessageFromServer.newBuilder()
              .setType(chatMessage.getType())
              .setTimestamp(now)
              .setFrom(username)
              .setMessage(chatMessage.getMessage())
              .setRoomName(chatMessage.getRoomName())
          .build();
      observers.stream().forEach(o -> o.onNext(messageFromServer));
      break;
  }
}
```

4. If there is an error, or when the client closes connection, remove the responseObserver from all rooms

```
@Override
public void onError(Throwable throwable) {
    logger.log(Level.SEVERE, "gRPC error", throwable);
    removeObserverFromAllRooms(responseObserver);
}

@Override
public void onCompleted() {
  removeObserverFromAllRooms(responseObserver);
}
```

Implement Chat CLI Client

In the `ChatClient.initChatStream()`, call `chatStreamService.chat(...)`: and assign it to `this.toServer` variable.

```
public void initChatStream() {
  this.toServer = chatStreamService.chat(
      new StreamObserver<ChatMessageFromServer>() {
    @Override
    public void onNext(ChatMessageFromServer chatMessageFromServer) {
```

```
    }

    @Override
    public void onError(Throwable throwable) {
    }

    @Override
    public void onCompleted() {
    }
  });
}
```

When the chat messages arrive from the server, `onNext` will be called. Print out the message.

```
@Override
public void onNext(ChatMessageFromServer chatMessageFromServer) {
  try {
    printLine(console, String.format("%tr %s> %s", chatMessageFromServer
        .getTimestamp().getSeconds(),
    chatMessageFromServer.getFrom(),
    chatMessageFromServer.getMessage()));
  } catch (IOException e) {
    logger.log(Level.SEVERE, "error printing to console", e);
  }
}
```

When the server throws an error or close the connection, shutdown the client:

```
@Override
public void onError(Throwable throwable) {
  logger.log(Level.SEVERE, "gRPC error", throwable);
  shutdown();
}

@Override
public void onCompleted() {
  logger.severe("server closed connection, shutting down...");
  shutdown();
}
```

Finally, implement `ChatClient.sendMessage(...)` method. Every time a user presses enter, it'll call this method to send the message out to the server:

1. Check that `toServer` observer is not null
2. Then, call `toServer.onNext(...)` to send the message to the server to be broadcasted to the room

```
private void sendMessage(String room, String message) {
  logger.info("sending chat message");
```

```
  if (toServer == null) {
    logger.severe("Not connected");
  }

  toServer.onNext(ChatMessage.newBuilder()
      .setType(MessageType.TEXT)
      .setRoomName(room)
      .setMessage(message)
      .build());
}
```

Exercise 4 - Distributed Tracing

There are a number of OpenTracing tracers for gRPC. In this lab, we'll use Brave. Brave has a number of Java-based trace interceptors that can propagate trace headers across network boundaries. Similarly, there is one for gRPC!

In auth-service/pom.xml, you can see the dependencies:

1. Brave instrumentation library provides the client side and server side interceptors to propagate trace ID and Spans.
2. URL Connection Reporter can post trace data to a Zipkin server

```
 <!-- Tracing -->
<dependency>
  <groupId>io.zipkin.brave</groupId>
  <artifactId>brave-instrumentation-grpc</artifactId>
  <version>4.4.0</version>
</dependency>
<dependency>
  <groupId>io.zipkin.reporter</groupId>
  <artifactId>zipkin-sender-urlconnection</artifactId>
  <version>0.10.0</version>
</dependency>
```

Define a Reporter and Tracer

To use a tracer, first define a reporter so that trace data can be forwarded to a Zipkin endpoint. We'll use the `URLConnectionSender`, and post it to a local Zipkin instance.

You'll need to add this to every service and client:

- `AuthServer.main(...)`
- `ChatServer.main(...)`
- `ChatClient`

```
AsyncReporter<Span> reporter = AsyncReporter.create(
    URLConnectionSender.create("http://localhost:9411/api/v1/spans"));
```

```
GrpcTracing tracing = GrpcTracing.create(Tracing.newBuilder()
    .localServiceName("my-service") // MAKE SURE YOU CHANGE THE NAME
    .reporter(reporter)
    .build());
```

Add Server Interceptors

We looked at server interceptors in the previous section. Add the `tracing.newServerInterceptor()` to every service:

- `AuthServer.main(...)`
- `ChatServer.main(...)`

```
final Server server = ServerBuilder.forPort(9091)
        .addService(ServerInterceptors.intercept(
            someServiceImpl, tracing.newServerInterceptor()))
        .build();
```

Add Client Interceptor

Also, add `tracing.newClientInterceptor()` to every outbound channels:

- `ChatServer.main(...)`
- `ChatClient.initAuthService()`
- `ChatClient.initChatServices()`

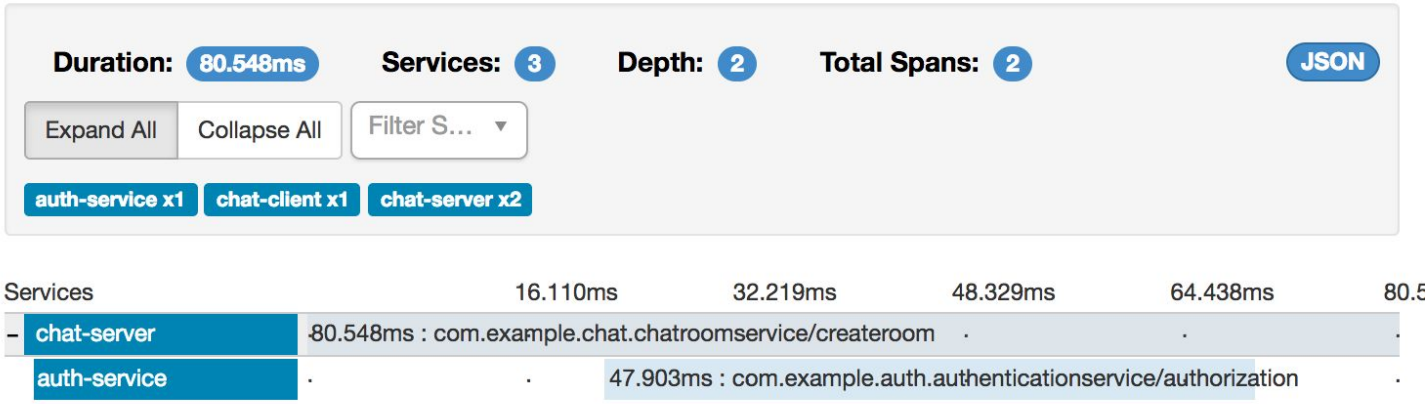
```
channel = ManagedChannelBuilder…
    .intercept(tracing.newClientInterceptor())
    .build()
```

Start Zipkin Server

You can download the Zipkin server from Zipkin's Quickstart: https://github.com/openzipkin/zipkin#quick-start

```
$ java -jar zipkin.jar
```

If you covered everything, restart all the servers and client, login and send a few more messages. Then login into the Zipkin server to see the traces! (Access Zipkin console on http://localhost:9411)


Duration: 80.548ms
Services: 3
Depth: 2
Total Spans: 2
JSON
Expand All
Collapse All
Filter S...
auth-service x1
chat-client x1
chat-server x2
Services
16.110ms
32.219ms
48.329ms
64.438ms
80.5
chat-server
80.548ms : com.example.chat.chatroomservice/createroom
auth-service
47.903ms : com.example.auth.authenticationservice/authorization