gRPC Communication

Introducing gRPC with Spring Boot

Owerya RESOURCING

Objectives

- Introducing gRPC
- gRPC Overview & Protocol Buffers
- Example gRPC with Spring Boot

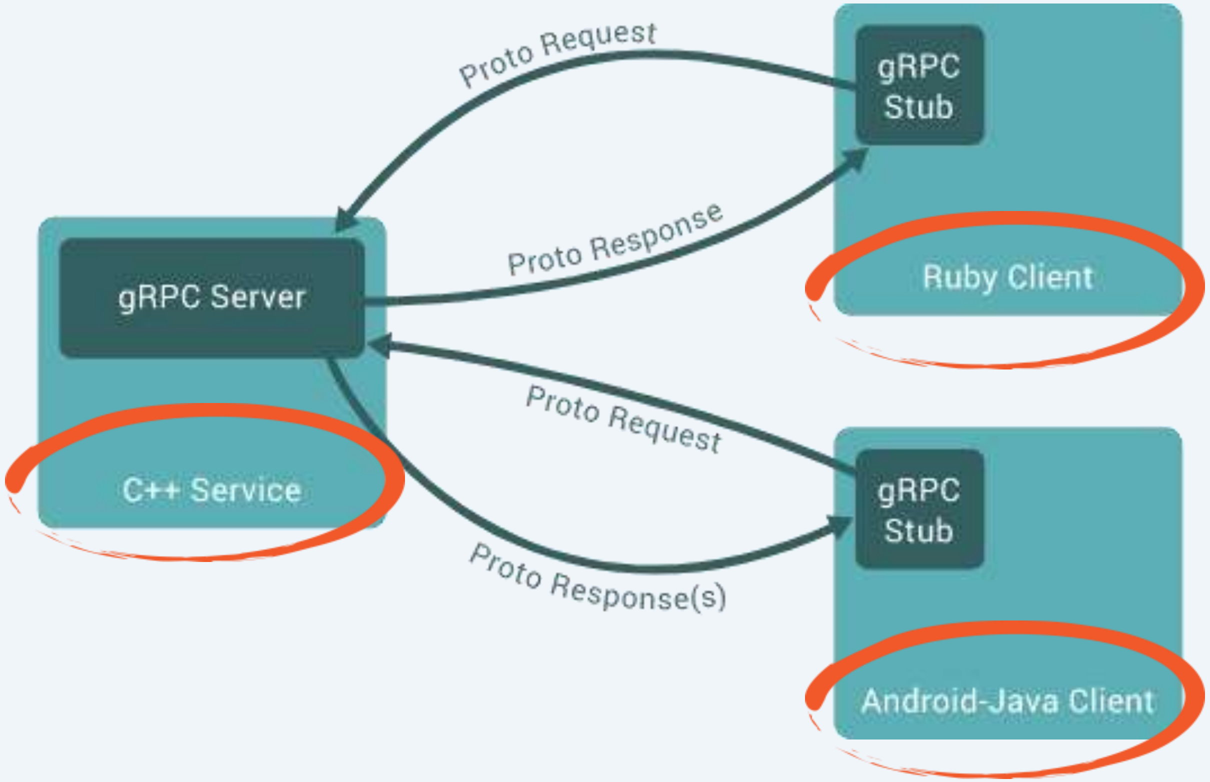
Introducing gRPC

- gRPC is an open-source Remote Procedure Call (RPC) framework
  - https://grpc.io
  - Originally created by Google (called Stubby), then open sourced in 2015
  - designed to run in any environment.
- Features:
  - Efficiently connect services in and across data centres
  - Pluggable support for load balancing, tracing, health checking and authentication
- Usage Scenarios:
  - Connecting polyglot services in microservices architecture
  - Connecting mobile devices and browser clients to backend services
  - Generating efficient client libraries
- Languages and Platforms:
  - Java, C++, Node, Python, C#, Go, PHP, Ruby, etc…

gRPC Overview

- A service interface is defined and implemented on the server, which runs a gRPC server to handle client calls.
- A client application can directly call a method on a server application on a *different* machine as if it were a local object.
  - This facilitates easier creation of microservices
- The clients each have a gRPC stub that provides the same methods as the server.
  - The client calls the stub method and the server generates the response
- If you want to use Google functionality, the latest Google APIs have gRPC versions of their interfaces.

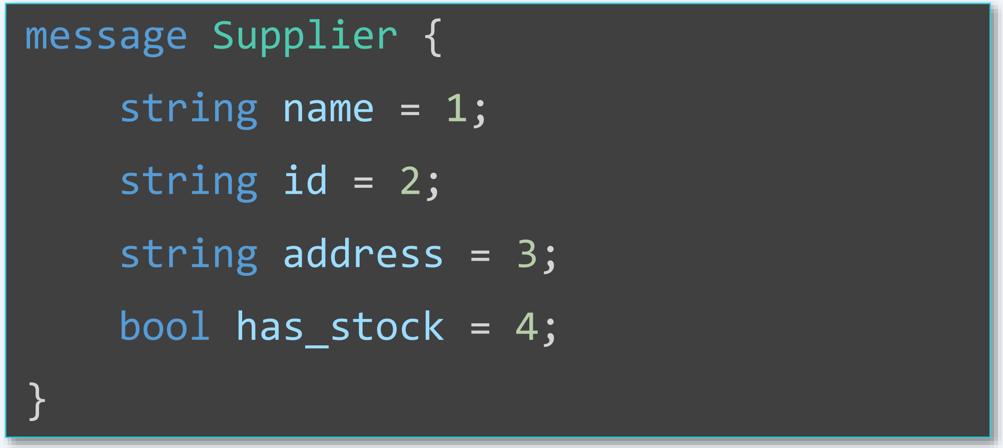
gRPC Overview



Clients and servers can communicate in a variety of environments and languages

Protocol Buffers: messages

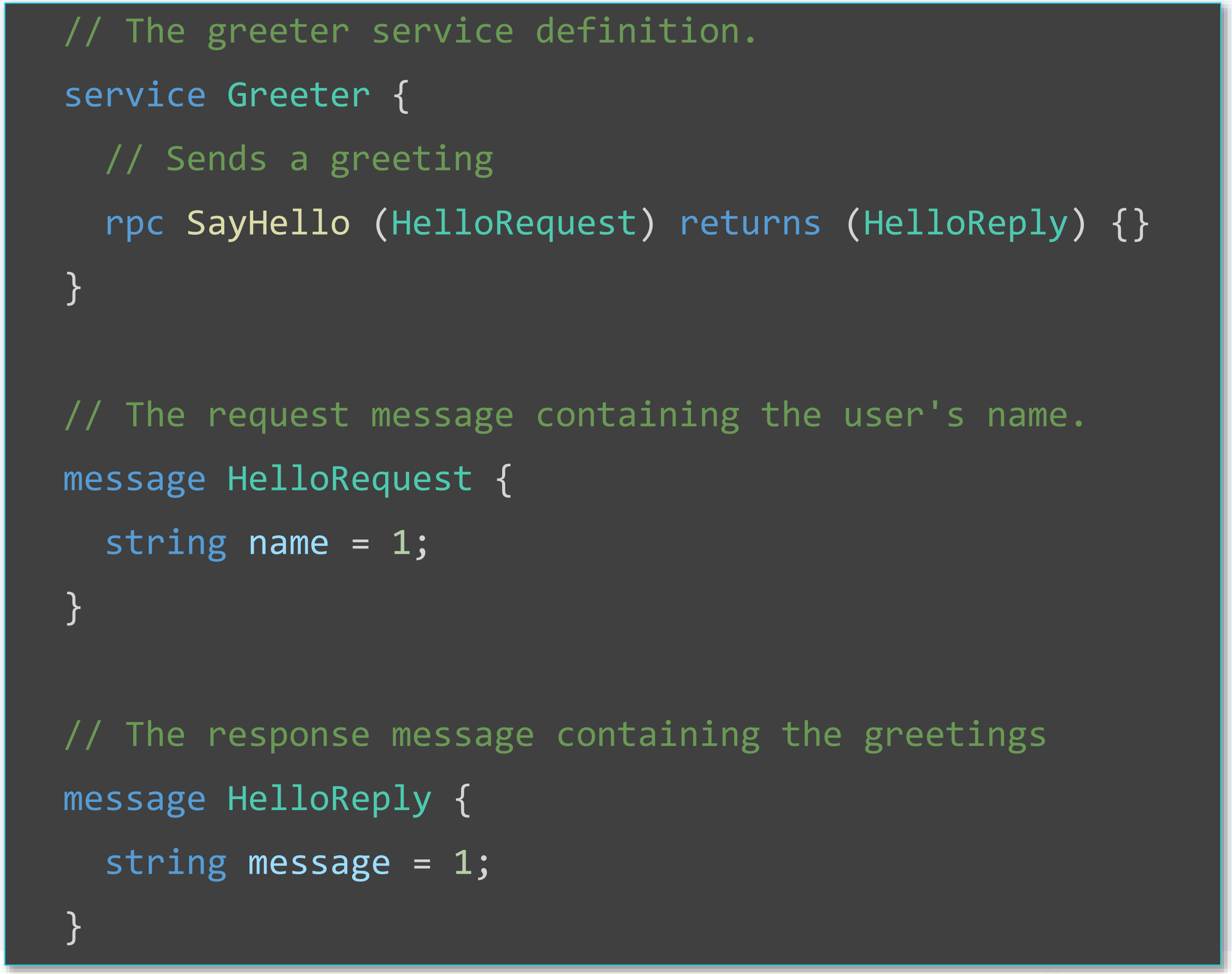
- gRPC uses Protocol Buffers to serialize the message data
  - From Google, open-source
  - You can use other data formats such as JSON.
- To use Protocol Buffers you define the data structure you want in a `.proto` file as messages, where each message is a logical record of info as name-value pairs:



```
message Supplier {
    string name = 1;
    string id = 2;
    string address = 3;
    bool has_stock = 4;
}
```

Protocol Buffers: services

- gRPC is based around the idea of defining a service
- A service specifies the methods that can be called remotely with their call parameters and return types.
- gRPC allows 4 kinds of service method:
  - Unary
  - Server Streaming
  - Client Streaming
  - Bidirectional Streaming



```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

gRPC Service Methods

- Unary RPCs
  - Client sends a single request to the server and gets a single response back
  - E.g. `rpc SayHello(HelloRequest) returns (HelloResponse);`
- Server Streaming RPCs
  - Client sends a request to the server and gets a stream to read a sequence of messages back.
  - The client reads the return stream until there are no more messages.
  - gRPC guarantees message ordering within an RPC call.
  - E.g. `rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);`

gRPC Service Methods

- Client Streaming RPCs
  - Client sends a sequence of messages to the server using a provided stream.
  - Once the client has finished sending, it waits for the server's response.
  - gRPC guarantees message ordering within an RPC call.
  - E.g. `rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);`
- Bidirectional Streaming RPCs
  - Client and server send a sequence of messages using a read-write stream.
  - The two streams operate independently, so clients and servers can read and write in whatever order they like.
  - gRPC guarantees message ordering within each stream.
  - E.g. `rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);`

Protocol Buffers

- When the data structures and services are specified the `protoc` compiler generates the following code in your specified language for populating, serializing and retrieving the specified message types:
  - gRPC client
  - server code
  - protocol buffer code

Synchronous vs Asynchronous gRPC

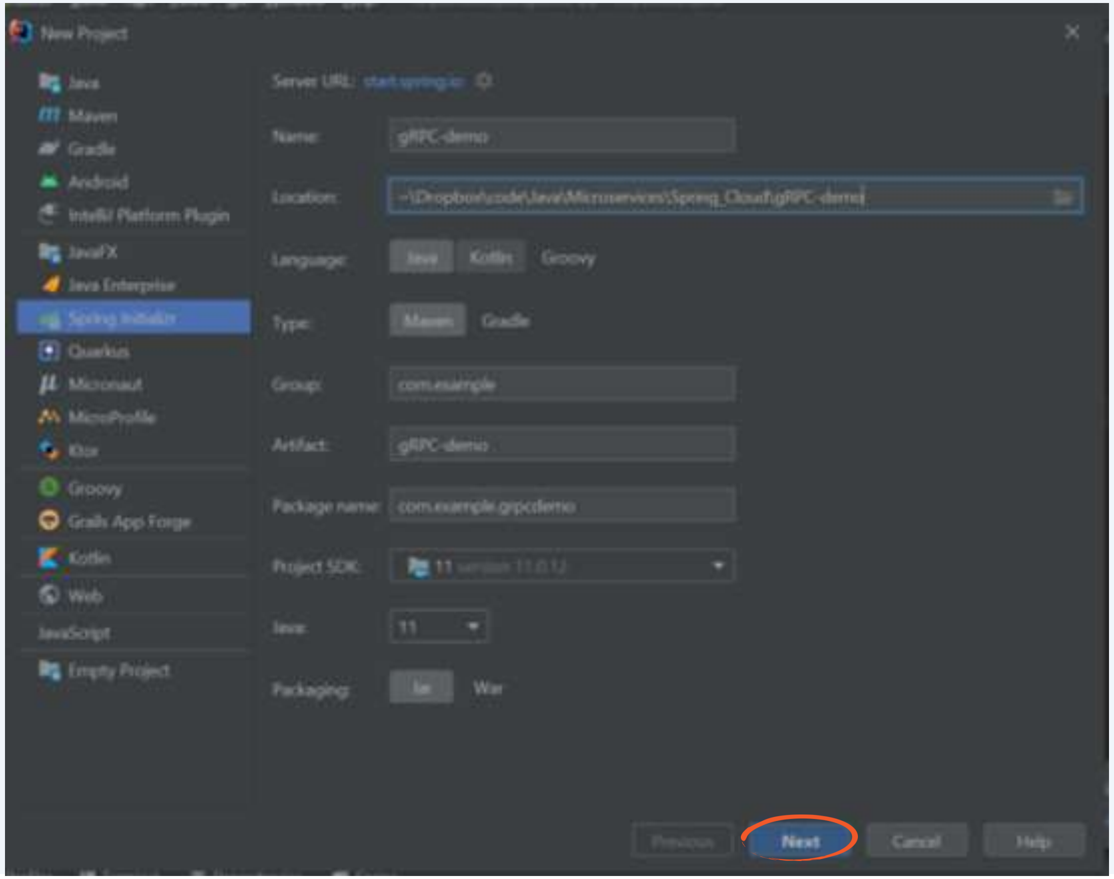
- We are considering gRPC as a synchronous (blocking) communication technology.
- The gRPC programming API in most languages also supplies asynchronous flavours so you can start RPCs without blocking the current thread.

RPC Life Cycle

- gRPC provides a connection between a client and a server using a **channel** on a specified host and port.
  - This is used when creating a client stub.
- Let’s consider the simplest RPC - where a client sends a single request and gets back a single response:
  - When the client calls a stub method the server is notified that the RPC has been invoked and is passed relevant client metadata for the call.
  - The server can then either respond with its own initial metadata, or wait for the client’s request and supply its metadata together with its response.
  - Once the server has received the client request, the response is populated. If this is successful the response is returned with a status code.
  - If the response status is OK, then the client gets the response which completes the call.

Example: gRPC with Spring Boot

- First we will just create a standard Spring Web project



Server URL: start.spring.io

Name: gRPC-demo

Location: ~\Dropbox\code\Java\Microservices\Spring_Cloud\gRPC-demo

Language: Java Kotlin Groovy

Type: Maven Gradle

Group: com.example

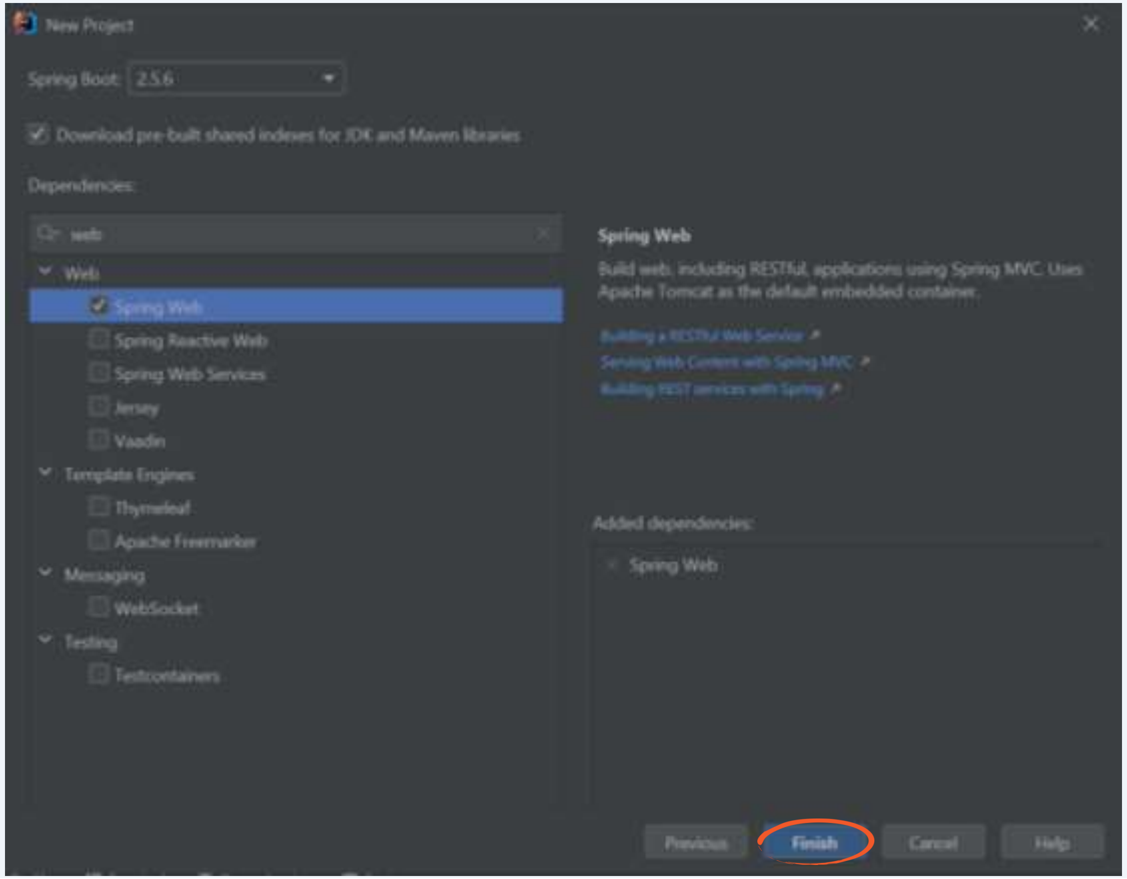
Artifact: gRPC-demo

Package name: com.example.grpcdemo

Project SDK: 11 version 11.0.12

Java: 11

Packaging: Jar War



Spring Boot: 2.5.6

☑ Download pre-built shared indexes for JDK and Maven libraries

Dependencies:

web

- Web
  - ☑ Spring Web
  - □ Spring Reactive Web
  - □ Spring Web Services
  - □ Jersey
  - □ Vaadin
- Template Engines
  - □ Thymeleaf
  - □ Apache Freemarker
- Messaging
  - □ WebSocket
- Testing
  - □ Testcontainers

Spring Web

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Building a RESTful Web Service ↗

Serving Web Content with Spring MVC ↗

Building REST services with Spring ↗

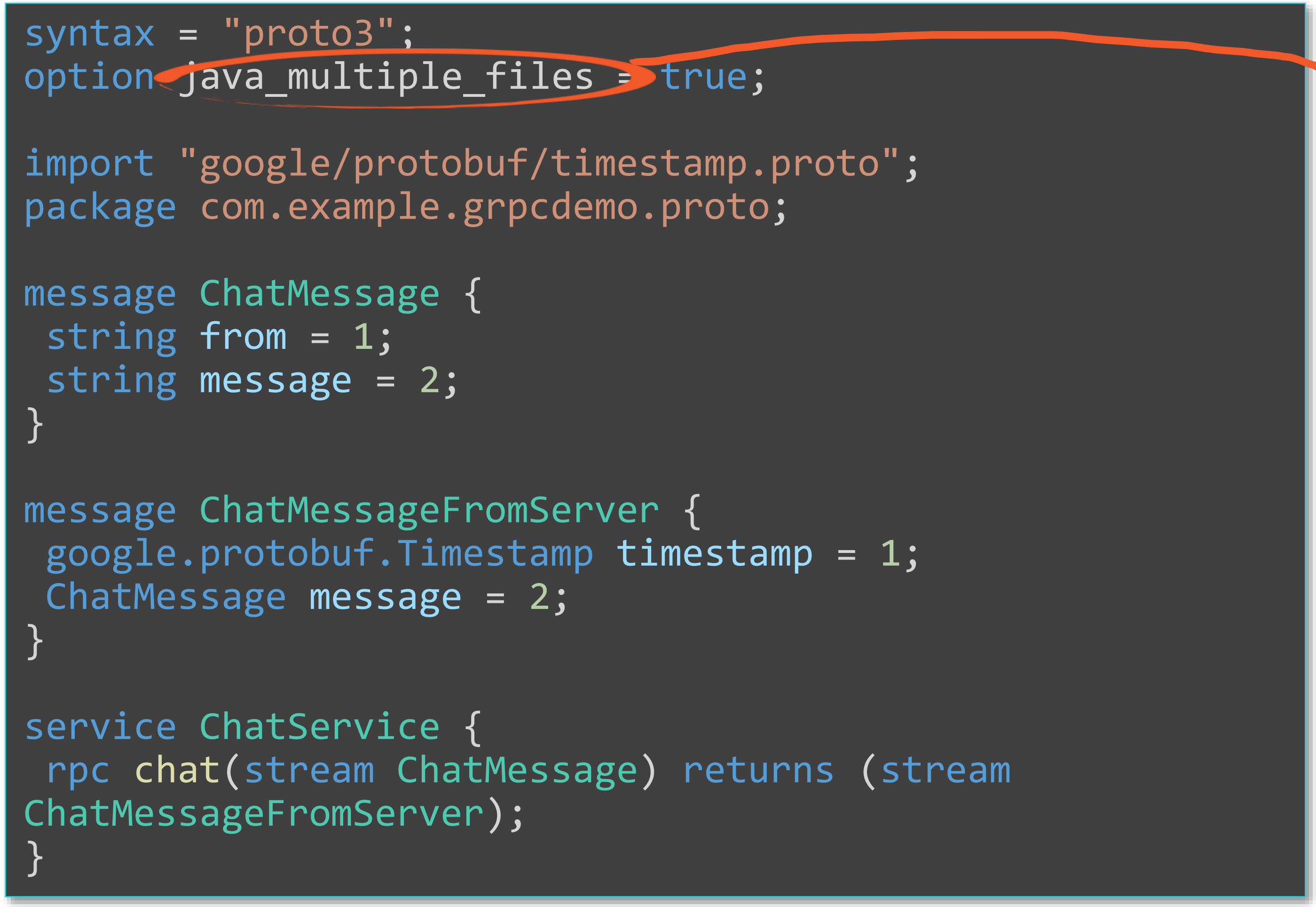
Added dependencies:

Spring Web

Example: gRPC with Spring Boot

- We will mostly follow Matt Penna’s code: https://mattpenna.dev/springboot-grpc/
- In `src/main` create a `proto` directory for the .proto files
  - This is the default location the `protobuf-maven-plugin` will look for .proto files.
- We are going to create a two way chat method
  - Both the server or the client can push data
  - To do this we will use streams as the request and response.

Example: proto file



```
syntax = "proto3";
option java_multiple_files = true;

import "google/protobuf/timestamp.proto";
package com.example.grpcdemo.proto;

message ChatMessage {
 string from = 1;
 string message = 2;
}

message ChatMessageFromServer {
 google.protobuf.Timestamp timestamp = 1;
 ChatMessage message = 2;
}

service ChatService {
 rpc chat(stream ChatMessage) returns (stream
ChatMessageFromServer);
}
```

Create Java classes in multiple files instead of one large file

`ChatMessages.proto`

Example: Maven file

- In the `<plugins>` section of `<build>` in the pom.xml:

```
<plugin>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.6.1</version>
  <configuration>
    <protocArtifact>com.google.protobuf:protoc:3.3.0:exe:${os.detected.classifier}</protocArtifact>
    <pluginId>grpc-java</pluginId>
    <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.4.0:exe:${os.detected.classifier}</pluginArtifact>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>compile-custom</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

`POM.xml`

Example: Maven file

- Also in the `<build>` section of the pom.xml:

```
<extensions>
    <extension>
        <groupId>kr.motd.maven</groupId>
        <artifactId>os-maven-plugin</artifactId>
        <version>1.6.1</version>
    </extension>
</extensions>
```

POM.xml

Example: Maven File

- And add this dependency:

```xml
<dependency>
   <groupId>net.devh</groupId>
   <artifactId>grpc-spring-boot-starter</artifactId>
   <version>2.12.0.RELEASE</version>
</dependency>
```

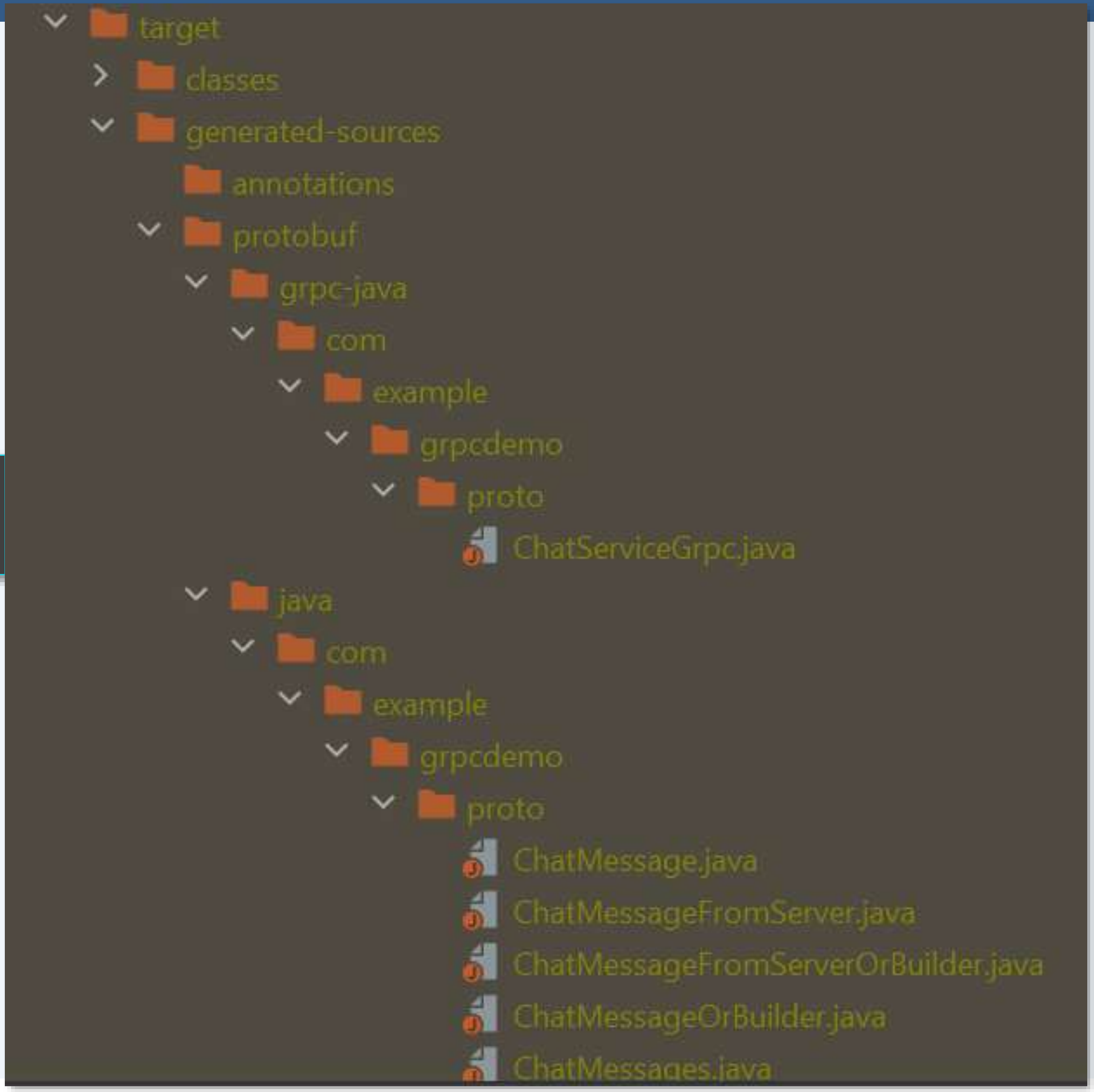
POM.xml

Example: Generating the gRPC classes

- Now we need to build the gRPC java classes.
- Open a terminal in your root directory and run
  - The –X option is only necessary if you get a build failure.

```
C:\code\Java\gRPC-demo> ./mvnw clean -X compile
```

- You should see something like → in your target directory



```
target
  classes
  generated-sources
    annotations
    protobuf
      grpc-java
        com
          example
            grpcdemo
              proto
                ChatServiceGrpc.java
      java
        com
          example
            grpcdemo
              proto
                ChatMessage.java
                ChatMessageFromServer.java
                ChatMessageFromServerOrBuilder.java
                ChatMessageOrBuilder.java
                ChatMessages.java
```

Example: Generating the gRPC classes

- These classes are auto-generated by the proto compiler
  - We will use them to implement our chat service
- But first we need to add the generated-sources to our build

Example: Including generated sources

- In the `<executions>` section of the protobuf plugin in pom.xml:

```
<execution>
  <id>add-source</id>
  <phase>generate-sources</phase>
  <configuration>
    <sources>
      <source>${project.build.directory}/target/generated-sources/protobuf</source>
      <source>${project.build.directory}/target/generated-sources/protobuf/grpc-java</source>
    </sources>
  </configuration>
</execution>
```

`POM.xml`

Example: gRPC with Spring Boot

- Now create a new package `com.example.grpcdemo.controllers`

- Then create a new class `ChatServiceImpl` that extends `ChatServiceGrpc.ChatServiceImplBase`
  - In this class we will override the `chat` method

- We will keep a list of all the clients that connect to our chat subscription in the static variable `chatClient`

ChatServiceImpl.java

```java
// package and imports up here
public class ChatServiceImpl extends ChatServiceGrpc.ChatServiceImplBase {

    private static LinkedHashSet<StreamObserver<ChatMessageFromServer>> chatClients = new LinkedHashSet<>();

    @Override
    public StreamObserver<ChatMessage> chat(StreamObserver<ChatMessageFromServer> responseObserver)
    {
        chatClients.add(responseObserver);

        return new StreamObserver<>() {
            @Override
            public void onNext(ChatMessage chatMessage) {
                for (StreamObserver<ChatMessageFromServer> chatClient : chatClients) {
                    chatClient.onNext(ChatMessageFromServer.newBuilder().setMessage(chatMessage).build());
                }
            }
                // trivial overrides of onError and onCompleted here
        };
    }
}
```

Everytime a client sends a chat, it gets put into the list of clients

ChatServiceImpl.java

```java
// package and imports up here
public class ChatServiceImpl extends ChatServiceGrpc.ChatServiceImplBase {

  private static LinkedHashSet<StreamObserver<ChatMessageFromServer>> chatClients = new LinkedHashSet<>();

  @Override
  public StreamObserver<ChatMessage> chat(StreamObserver<ChatMessageFromServer> responseObserver)
  {
    chatClients.add(responseObserver);

    return new StreamObserver<>() {
      @Override
      public void onNext(ChatMessage chatMessage) {
        for (StreamObserver<ChatMessageFromServer> chatClient : chatClients) {
          chatClient.onNext(ChatMessageFromServer.newBuilder().setMessage(chatMessage).build());
        }
      }
        // trivial overrides of onError and onCompleted here
    };
  }
}
```

Chat returns a StreamObserver which must override these methods

ChatServiceImpl.java

```java
// package and imports up here
public class ChatServiceImpl extends ChatServiceGrpc.ChatServiceImplBase {

   private static LinkedHashSet<StreamObserver<ChatMessageFromServer>> chatClients = new LinkedHashSet<>();

   @Override
   public StreamObserver<ChatMessage> chat(StreamObserver<ChatMessageFromServer> responseObserver)
   {
      chatClients.add(responseObserver);

      return new StreamObserver<>() {
         @Override
         public void onNext(ChatMessage chatMessage) {
            for (StreamObserver<ChatMessageFromServer> chatClient : chatClients) {
               chatClient.onNext(ChatMessageFromServer.newBuilder().setMessage(chatMessage).build());
            }
         }
            // trivial overrides of onError and onCompleted here
      };
   }
}
```

onNext iterates through the client list and posts the chat message to everyone

Example: gRPC with Spring Boot

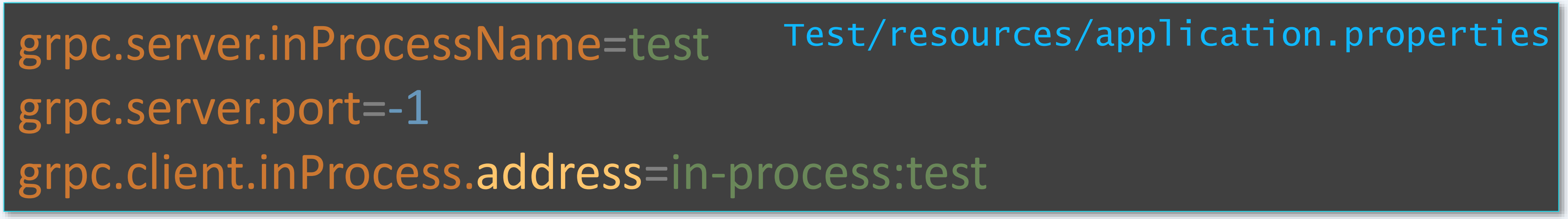
- When this class is in place, we have a gRPC server that can accept and send chat messages to clients!

- This is almost entirely configuration
  - The only coding was in the protobuffer file, and the implementation of the server class.

- It will work with gRPC clients from any platform
  - We will test it with a Spring Boot integration test.

Testing the gRPC server

- First create a resources directory under src/test
- Then create a new application.properties file in that directory:



Test/resources/application.properties

```
grpc.server.inProcessName=test
grpc.server.port=-1
grpc.client.inProcess.address=in-process:test
```

- Setting the grpc.server.port to -1 forces the server to work locally.

Testing the gRPC server

- Open the auto-generated `GRpcDemoApplicationTests.java` file in the `test/java/com/example/grpcdemo` directory.

- We will use the `@GrpcClient` annotation to create two clients, and use “`inProcess`” so it knows to use the `inprocess` connection.
  - The address value comes from the `grpc.client.inProcess.address` we set in our `test/application.properties` file

Testing: creating clients

```
@SpringBootTest
class GRpcDemoApplicationTests {
   @GrpcClient("inProcess")
   private ChatServiceGrpc.ChatServiceStub chatClient1;

   @GrpcClient("inProcess")
   private ChatServiceGrpc.ChatServiceStub chatClient2;

   @Test
   void contextLoads() {
   }
}
```

GRpcDemoApplicationTests.java

Testing: Client Stream Observers

- Now we can write the test – called testChat

- We will need to create 2 stream observers (1 for each chat client).
  - Just add received messages to a list in onNext so we can validate our test.
  - Leave the other methods empty

Testing: Client Stream Observers

```java
private StreamObserver<ChatMessage> generateObserver(List messages)
{
    return new StreamObserver<>() {
        @Override
        public void onNext(ChatMessage chatMessage) {
            messages.add(chatMessage);
            System.out.println(chatMessage.getMessage());
        }

        @Override
        public void onError(Throwable throwable) { }

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

GRpcDemoApplicationTests.java

Testing: Register clients with the server

- Then all we need to do is register the client observers with the server, and start sending messages.
- The full code is in the code snippets.

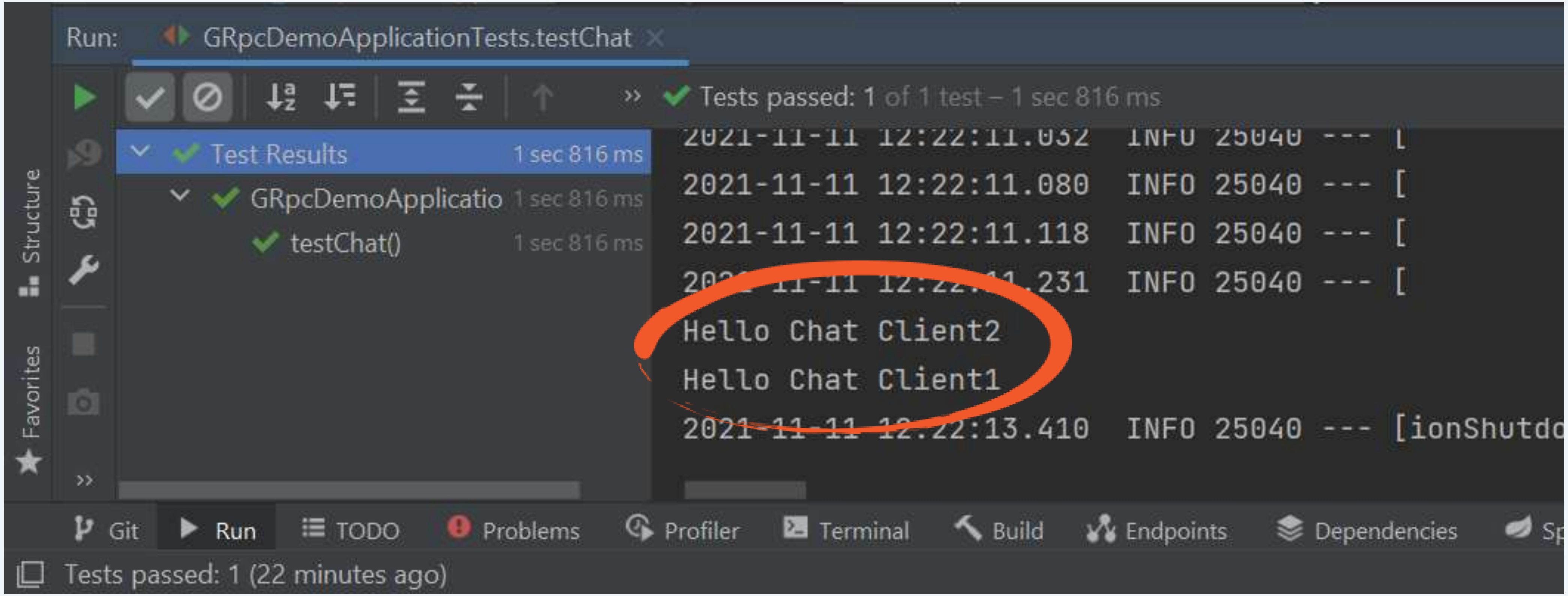
Testing: Register clients with the server

```java
@Test
public void testChat()
{
   List messages = new ArrayList<ChatMessageFromServer>();

   StreamObserver chatClient1Observer = generateObserver(messages);
   StreamObserver chatClient2Observer = generateObserver(messages);

   chatClient1.chat(chatClient1Observer);
   chatClient2.chat(chatClient2Observer);

   chatClient1Observer.onNext(generateChatMessage("Hello Chat Client2", "ChatClient1"));
   chatClient2Observer.onNext(generateChatMessage("Hello Chat Client1", "ChatClient2"));

   Assert.notEmpty(messages, "Validate Messages are populated");
}
```

GRpcDemoApplicationTests.java

Testing complete

- Right click on the testChat() method and select run.
- The test should pass, and you will see the chat messages printed to the terminal.



Run: GRpcDemoApplicationTests.testChat
Tests passed: 1 of 1 test – 1 sec 816 ms
Test Results 1 sec 816 ms
GRpcDemoApplicatio 1 sec 816 ms
testChat() 1 sec 816 ms
Structure
Favorites
Git
Run
TODO
Problems
Profiler
Terminal
Build
Endpoints
Dependencies
Sp
Tests passed: 1 (22 minutes ago)

```
2021-11-11 12:22:11.032  INFO 25040 --- [
2021-11-11 12:22:11.080  INFO 25040 --- [
2021-11-11 12:22:11.118  INFO 25040 --- [
2021-11-11 12:22:11.231  INFO 25040 --- [
Hello Chat Client2
Hello Chat Client1
2021-11-11 12:22:13.410  INFO 25040 --- [ionShutdo
```

Summary

- Introducing gRPC
- gRPC Overview & Protocol Buffers
- Example gRPC with Spring Boot

Questions or Comments?