gRPC Basics - Java

This tutorial provides a basic Java programmer's introduction to working with gRPC.

By walking through this example you'll learn how to:

- Define a service in a .proto file.
- Generate server and client code using the protocol buffer compiler.
- Use the Java gRPC API to write a simple client and server for your service.

It assumes that you have read the Overview and are familiar with protocol buffers. Note that the example in this tutorial uses the proto3 version of the protocol buffers language: you can find out more in the proto3 language guide and Java generated code guide.

Why use gRPC?

Our example is a simple route mapping application that lets clients get information about features on their route, create a summary of their route, and exchange route information such as traffic updates with the server and other clients.

With gRPC we can define our service once in a .proto file and implement clients and servers in any of gRPC's supported languages, which in turn can be run in environments ranging from servers inside Google to your own tablet - all the complexity of communication between different languages and environments is handled for you by gRPC. We also get all the advantages of working with protocol buffers, including efficient serialization, a simple IDL, and easy interface updating.

Example code and setup

The example code for our tutorial is in grpc/grpc-java/examples/src/main/java/io/grpc/examples. To download the example, clone the latest release in `grpc-java` repository by running the following command:

```
$ git clone -b v1.26.0 https://github.com/grpc/grpc-java.git
```

Then change your current directory to `grpc-java/examples`:

```
$ cd grpc-java/examples
```

Defining the service

Our first step (as you'll know from the Overview) is to define the gRPC *service* and the method *request* and *response* types using protocol buffers. You can see the complete .proto file in `grpc-java/examples/src/main/proto/route_guide.proto`.

As we're generating Java code in this example, we've specified a `java_package` file option in our .proto:

```
option java_package = "io.grpc.examples.routeguide";
```

This specifies the package we want to use for our generated Java classes. If no explicit `java_package` option is given in the .proto file, then by default the proto package (specified using the "package" keyword) will be used. However, proto packages generally do not make good Java packages since proto packages are not expected to start with reverse domain names. If we generate code in another language from this .proto, the `java_package` option has no effect.

To define a service, we specify a named `service` in the .proto file:

```
service RouteGuide {
   ...
}
```

Then we define `rpc` methods inside our service definition, specifying their request and response types. gRPC lets you define four kinds of service methods, all of which are used in the `RouteGuide` service:

- A *simple RPC* where the client sends a request to the server using the stub and waits for a response to come back, just like a normal function call.

```
// Obtains the feature at a given position.
rpc GetFeature(Point) returns (Feature) {}
```

- A *server-side streaming RPC* where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages. As you can see in our example, you specify a server-side streaming method by placing the `stream` keyword before the *response* type.

```
// Obtains the Features available within the given
Rectangle.  Results are
// streamed rather than returned at once (e.g. in a
response message with a
// repeated field), as the rectangle may cover a large area
and contain a
// huge number of features.
rpc ListFeatures(Rectangle) returns (stream Feature) {}
```

- A *client-side streaming RPC* where the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them all and return its response. You specify a

client-side streaming method by placing the `stream` keyword before the *request* type.

```
// Accepts a stream of Points on a route being traversed,
returning a
// RouteSummary when traversal is completed.
rpc RecordRoute(stream Point) returns (RouteSummary) {}
```

- A *bidirectional streaming RPC* where both sides 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: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved. You specify this type of method by placing the `stream` keyword before both the request and the response.

```
// Accepts a stream of RouteNotes sent while a route is
being traversed,
// while receiving other RouteNotes (e.g. from other
users).
rpc RouteChat(stream RouteNote) returns (stream RouteNote)
{}
```

Our .proto file also contains protocol buffer message type definitions for all the request and response types used in our service methods - for example, here's the `Point` message type:

```
// Points are represented as latitude-longitude pairs in
the E7 representation
// (degrees multiplied by 10**7 and rounded to the nearest
integer).
// Latitudes should be in the range +/- 90 degrees and
longitude should be in
// the range +/- 180 degrees (inclusive).
message Point {
  int32 latitude = 1;
  int32 longitude = 2;
}
```

Generating client and server code

Next we need to generate the gRPC client and server interfaces from our .proto service definition. We do this using the protocol buffer compiler `protoc` with a special gRPC Java plugin. You need to use the proto3 compiler (which supports both proto2 and proto3 syntax) in order to generate gRPC services.

When using Gradle or Maven, the protoc build plugin can generate the necessary code as part of the build. You can refer to the README for how to generate code from your own .proto files.

The following classes are generated from our service definition:

- `Feature.java`, `Point.java`, `Rectangle.java`, and others which contain all the protocol buffer code to populate, serialize, and

retrieve our request and response message types.
- `RouteGuideGrpc.java` which contains (along with some other useful code):
  - a base class for `RouteGuide` servers to implement, `RouteGuideGrpc.RouteGuideImplBase`, with all the methods defined in the `RouteGuide` service.
  - *stub* classes that clients can use to talk to a `RouteGuide` server.

Creating the server

First let's look at how we create a `RouteGuide` server. If you're only interested in creating gRPC clients, you can skip this section and go straight to Creating the client (though you might find it interesting anyway!).

There are two parts to making our `RouteGuide` service do its job:

- Overriding the service base class generated from our service definition: doing the actual "work" of our service.
- Running a gRPC server to listen for requests from clients and return the service responses.

You can find our example `RouteGuide` server in grpc-java/examples/src/main/java/io/grpc/examples/routeguide/RouteGuideServer.java. Let's take a closer look at how it works.

Implementing RouteGuide

As you can see, our server has a `RouteGuideService` class that extends the generated `RouteGuideGrpc.RouteGuideImplBase` abstract class:

```
private static class RouteGuideService extends
RouteGuideGrpc.RouteGuideImplBase {
...
}
```

Simple RPC

`RouteGuideService` implements all our service methods. Let's look at the simplest type first, `GetFeature`, which just gets a `Point` from the client and returns the corresponding feature information from its database in a `Feature`.

```
@Override
public void getFeature(Point request,
StreamObserver<Feature> responseObserver) {
  responseObserver.onNext(checkFeature(request));
  responseObserver.onCompleted();
}

...

private Feature checkFeature(Point location) {
  for (Feature feature : features) {
    if (feature.getLocation().getLatitude() ==
location.getLatitude()
        && feature.getLocation().getLongitude() ==
location.getLongitude()) {
      return feature;
    }
  }

  // No feature was found, return an unnamed feature.
  return
Feature.newBuilder().setName("").setLocation(location).build()

}
```

`getFeature()` takes two parameters:

- `Point`: the request
- `StreamObserver<Feature>`: a response observer, which is a special interface for the server to call with its response.

To return our response to the client and complete the call:

1. We construct and populate a `Feature` response object to return to the client, as specified in our service definition. In this example, we do this in a separate private `checkFeature()` method.
2. We use the response observer's `onNext()` method to return the `Feature`.
3. We use the response observer's `onCompleted()` method to specify that we've finished dealing with the RPC.

Server-side streaming RPC

Next let's look at one of our streaming RPCs. `ListFeatures` is a server-side streaming RPC, so we need to send back multiple `Feature`s to our client.

```
private final Collection<Feature> features;

...

@Override
public void listFeatures(Rectangle request,
StreamObserver<Feature> responseObserver) {
  int left = min(request.getLo().getLongitude(),
request.getHi().getLongitude());
  int right = max(request.getLo().getLongitude(),
request.getHi().getLongitude());
  int top = max(request.getLo().getLatitude(),
request.getHi().getLatitude());
  int bottom = min(request.getLo().getLatitude(),
request.getHi().getLatitude());

  for (Feature feature : features) {
    if (!RouteGuideUtil.exists(feature)) {
      continue;
    }

    int lat = feature.getLocation().getLatitude();
    int lon = feature.getLocation().getLongitude();
    if (lon >= left && lon <= right && lat >= bottom && lat
<= top) {
      responseObserver.onNext(feature);
    }
  }
  responseObserver.onCompleted();
}
```

Like the simple RPC, this method gets a request object (the `Rectangle` in which our client wants to find `Feature`s) and a `StreamObserver` response observer.

This time, we get as many `Feature` objects as we need to return to the client (in this case, we select them from the service's feature collection based on whether they're inside our request `Rectangle`), and write them each in turn to the response observer using its `onNext()` method. Finally, as in our simple RPC, we use the response observer's `onCompleted()` method to tell gRPC that we've finished writing responses.

Client-side streaming RPC

Now let's look at something a little more complicated: the client-side streaming method `RecordRoute`, where we get a stream of `Point`s from the client and return a single `RouteSummary` with information about their trip.

```
@Override
public StreamObserver<Point> recordRoute(final
StreamObserver<RouteSummary> responseObserver) {
  return new StreamObserver<Point>() {
    int pointCount;
    int featureCount;
    int distance;
    Point previous;
    long startTime = System.nanoTime();

    @Override
    public void onNext(Point point) {
      pointCount++;
      if (RouteGuideUtil.exists(checkFeature(point))) {
        featureCount++;
      }
      // For each point after the first, add the
incremental distance from the previous point
      // to the total distance value.
      if (previous != null) {
        distance += calcDistance(previous, point);
      }
      previous = point;
    }

    @Override
    public void onError(Throwable t) {
      logger.log(Level.WARNING, "Encountered error in
recordRoute", t);
    }

    @Override
    public void onCompleted() {
      long seconds =
NANOSECONDS.toSeconds(System.nanoTime() - startTime);

responseObserver.onNext(RouteSummary.newBuilder().setPointCour


.setFeatureCount(featureCount).setDistance(distance)
          .setElapsedTime((int) seconds).build());
      responseObserver.onCompleted();
    }
  };
}
```

As you can see, like the previous method types our method gets a `StreamObserver` response observer parameter, but this time it returns a `StreamObserver` for the client to write its `Point`s.

In the method body we instantiate an anonymous `StreamObserver` to return, in which we:

- Override the `onNext()` method to get features and other information each time the client writes a `Point` to the message stream.
- Override the `onCompleted()` method (called when the *client* has finished writing messages) to populate and build our `RouteSummary`.

We then call our method's own response observer's `onNext()` with our `RouteSummary`, and then call its `onCompleted()` method to finish the call from the server side.

Bidirectional streaming RPC

Finally, let's look at our bidirectional streaming RPC `RouteChat()`.

```
@Override
public StreamObserver<RouteNote> routeChat(final
StreamObserver<RouteNote> responseObserver) {
  return new StreamObserver<RouteNote>() {
    @Override
    public void onNext(RouteNote note) {
      List<RouteNote> notes =
getOrCreateNotes(note.getLocation());

      // Respond with all previous notes at this location.
      for (RouteNote prevNote : notes.toArray(new
RouteNote[0])) {
        responseObserver.onNext(prevNote);
      }

      // Now add the new note to the list
      notes.add(note);
    }

    @Override
    public void onError(Throwable t) {
      logger.log(Level.WARNING, "Encountered error in
routeChat", t);
    }

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

As with our client-side streaming example, we both get and return a `StreamObserver` response observer, except this time we return values via our method's response observer while the client is still writing messages to *their* message stream. The syntax for reading and writing here is exactly the same as for our client-streaming and server-streaming methods. Although each side will always get the other's messages in the order they were written, both the client and server can read and write in any order — the streams operate completely independently.

Starting the server

Once we've implemented all our methods, we also need to start up a gRPC server so that clients can actually use our service. The following snippet shows how we do this for our `RouteGuide` service:

```
public RouteGuideServer(int port, URL featureFile) throws
IOException {
  this(ServerBuilder.forPort(port), port,
RouteGuideUtil.parseFeatures(featureFile));
}

/** Create a RouteGuide server using serverBuilder as a
base and features as data. */
public RouteGuideServer(ServerBuilder<?> serverBuilder, int
port, Collection<Feature> features) {
  this.port = port;
  server = serverBuilder.addService(new
RouteGuideService(features))
      .build();
}
...
public void start() throws IOException {
  server.start();
  logger.info("Server started, listening on " + port);
 ...
}
```

As you can see, we build and start our server using a `ServerBuilder`.

To do this, we:

1. Specify the address and port we want to use to listen for client requests using the builder's `forPort()` method.
2. Create an instance of our service implementation class `RouteGuideService` and pass it to the builder's `addService()` method.
3. Call `build()` and `start()` on the builder to create and start an RPC server for our service.

Creating the client

In this section, we'll look at creating a Java client for our `RouteGuide` service. You can see our complete example client code in grpc-java/examples/src/main/java/io/grpc/examples/routeguide/RouteGuideClient.java.

Creating a stub

To call service methods, we first need to create a *stub*, or rather, two stubs:

- a *blocking/synchronous* stub: this means that the RPC call waits for the server to respond, and will either return a response or raise an exception.
- a *non-blocking/asynchronous* stub that makes non-blocking calls to the server, where the response is returned asynchronously. You can make certain types of streaming call only using the asynchronous stub.

First we need to create a gRPC *channel* for our stub, specifying the server address and port we want to connect to:

```
public RouteGuideClient(String host, int port) {
  this(ManagedChannelBuilder.forAddress(host,
port).usePlaintext());
}

/** Construct client for accessing RouteGuide server using
the existing channel. */
public RouteGuideClient(ManagedChannelBuilder<?>
channelBuilder) {
  channel = channelBuilder.build();
  blockingStub = RouteGuideGrpc.newBlockingStub(channel);
  asyncStub = RouteGuideGrpc.newStub(channel);
}
```

We use a `ManagedChannelBuilder` to create the channel.

Now we can use the channel to create our stubs using the `newStub` and `newBlockingStub` methods provided in the `RouteGuideGrpc` class we generated from our .proto.

```
blockingStub = RouteGuideGrpc.newBlockingStub(channel);
asyncStub = RouteGuideGrpc.newStub(channel);
```

Calling service methods

Now let's look at how we call our service methods.

Simple RPC

Calling the simple RPC `GetFeature` on the blocking stub is as straightforward as calling a local method.

```
Point request =
Point.newBuilder().setLatitude(lat).setLongitude(lon).build();

Feature feature;
try {
  feature = blockingStub.getFeature(request);
} catch (StatusRuntimeException e) {
  logger.log(Level.WARNING, "RPC failed: {0}",
e.getStatus());
  return;
}
```

We create and populate a request protocol buffer object (in our case `Point`), pass it to the `getFeature()` method on our blocking stub, and get back a `Feature`.

If an error occurs, it is encoded as a `Status`, which we can obtain from the `StatusRuntimeException`.

Server-side streaming RPC

Next, let's look at a server-side streaming call to `ListFeatures`, which returns a stream of geographical `Feature`s:

```
Rectangle request =
    Rectangle.newBuilder()

.setLo(Point.newBuilder().setLatitude(lowLat).setLongitude(low


.setHi(Point.newBuilder().setLatitude(hiLat).setLongitude(hiLo

Iterator<Feature> features;
try {
  features = blockingStub.listFeatures(request);
} catch (StatusRuntimeException ex) {
  logger.log(Level.WARNING, "RPC failed: {0}",
e.getStatus());
  return;
}
```

As you can see, it's very similar to the simple RPC we just looked at, except instead of returning a single `Feature`, the method returns an `Iterator` that the client can use to read all the returned `Feature`s.

Client-side streaming RPC

Now for something a little more complicated: the client-side streaming method `RecordRoute`, where we send a stream of `Point`s to the server and get back a single `RouteSummary`. For this method we need to use the asynchronous stub. If you've already read Creating the server some of this may look very familiar - asynchronous streaming RPCs are implemented in a similar way on both sides.

```
public void recordRoute(List<Feature> features, int
numPoints) throws InterruptedException {
  info("*** RecordRoute");
  final CountDownLatch finishLatch = new CountDownLatch(1);
  StreamObserver<RouteSummary> responseObserver = new
StreamObserver<RouteSummary>() {
    @Override
    public void onNext(RouteSummary summary) {
      info("Finished trip with {0} points. Passed {1}
features. "
          + "Travelled {2} meters. It took {3} seconds.",
summary.getPointCount(),
          summary.getFeatureCount(), summary.getDistance(),
summary.getElapsedTime());
    }

    @Override
    public void onError(Throwable t) {
      Status status = Status.fromThrowable(t);
      logger.log(Level.WARNING, "RecordRoute Failed: {0}",
status);
      finishLatch.countDown();
    }

    @Override
    public void onCompleted() {
      info("Finished RecordRoute");
      finishLatch.countDown();
    }
  };

  StreamObserver<Point> requestObserver =
asyncStub.recordRoute(responseObserver);
  try {
    // Send numPoints points randomly selected from the
features list.
    Random rand = new Random();
    for (int i = 0; i < numPoints; ++i) {
      int index = rand.nextInt(features.size());
      Point point = features.get(index).getLocation();
      info("Visiting point {0}, {1}",
RouteGuideUtil.getLatitude(point),
          RouteGuideUtil.getLongitude(point));
      requestObserver.onNext(point);
      // Sleep for a bit before sending the next one.
      Thread.sleep(rand.nextInt(1000) + 500);
      if (finishLatch.getCount() == 0) {
        // RPC completed or errored before we finished
sending.
        // Sending further requests won't error, but they
will just be thrown away.
        return;
      }
    }
  } catch (RuntimeException e) {
    // Cancel RPC
    requestObserver.onError(e);
    throw e;
```

```
    }
    // Mark the end of requests
    requestObserver.onCompleted();

    // Receiving happens asynchronously
    finishLatch.await(1, TimeUnit.MINUTES);
  }
```

As you can see, to call this method we need to create a `StreamObserver`, which implements a special interface for the server to call with its `RouteSummary` response. In our `StreamObserver` we:

- Override the `onNext()` method to print out the returned information when the server writes a `RouteSummary` to the message stream.
- Override the `onCompleted()` method (called when the *server* has completed the call on its side) to reduce a `CountDownLatch` that we can check to see if the server has finished writing.

We then pass the `StreamObserver` to the asynchronous stub's `recordRoute()` method and get back our own `StreamObserver` request observer to write our `Point`s to send to the server. Once we've finished writing points, we use the request observer's `onCompleted()` method to tell gRPC that we've finished writing on the client side. Once we're done, we check our `CountDownLatch` to check that the server has completed on its side.

Bidirectional streaming RPC

Finally, let's look at our bidirectional streaming RPC `RouteChat()`.

```
public void routeChat() throws Exception {
  info("*** RoutChat");
  final CountDownLatch finishLatch = new CountDownLatch(1);
  StreamObserver<RouteNote> requestObserver =
      asyncStub.routeChat(new StreamObserver<RouteNote>() {
        @Override
        public void onNext(RouteNote note) {
          info("Got message \"{0}\" at {1}, {2}",
note.getMessage(), note.getLocation()
              .getLatitude(),
note.getLocation().getLongitude());
        }

        @Override
        public void onError(Throwable t) {
          Status status = Status.fromThrowable(t);
          logger.log(Level.WARNING, "RouteChat Failed:
{0}", status);
          finishLatch.countDown();
        }

        @Override
        public void onCompleted() {
          info("Finished RouteChat");
          finishLatch.countDown();
        }
      });

  try {
    RouteNote[] requests =
        {newNote("First message", 0, 0), newNote("Second
message", 0, 1),
            newNote("Third message", 1, 0), newNote("Fourth
message", 1, 1)};

    for (RouteNote request : requests) {
      info("Sending message \"{0}\" at {1}, {2}",
request.getMessage(), request.getLocation()
          .getLatitude(),
request.getLocation().getLongitude());
      requestObserver.onNext(request);
    }
  } catch (RuntimeException e) {
    // Cancel RPC
    requestObserver.onError(e);
    throw e;
  }
  // Mark the end of requests
  requestObserver.onCompleted();

  // Receiving happens asynchronously
  finishLatch.await(1, TimeUnit.MINUTES);
}
```

As with our client-side streaming example, we both get and return a `StreamObserver` response observer, except this time we send values via our method's response observer while the server is still writing messages to *their* message stream. The syntax for reading and writing here is exactly

the same as for our client-streaming method. Although each side will always get the other's messages in the order they were written, both the client and server can read and write in any order — the streams operate completely independently.

Try it out!

Follow the instructions in the example directory README to build and run the client and server.