Akka gRPC Quickstart with Java

Akka gRPC is a toolkit for building streaming gRPC servers and clients on top of Akka Streams. This guide will get you started building gRPC based systems with Java. If you prefer to use Akka gRPC with Scala, switch to the **Akka gRPC Quickstart with Scala guide (https://developer.lightbend.com/guides/akka-grpc-quickstart-scala/)**.

After trying this example the **Akka gRPC documentation (https://developer.lightbend.com/docs/akka-grpc/current/index.html)** is a good next step to continue learning more about Akka gRPC.

Downloading the example

The Hello World example for Scala is a zipped project that includes a distribution of sbt, Maven and Gradle. You can choose any of these build tools. You can run it on Linux, MacOS, or Windows. The only prerequisite is Java 8.

Download and unzip the example:

1. Download the **zip file (https://example.lightbend.com/v1/download/akka-grpc-quickstart-java?name=akka-grpc-quickstart-java)**.
2. Extract the zip file to a convenient location:

- On Linux and OSX systems, open a terminal and use the command `unzip akka-grpc-quickstart-java.zip`. Note: On OSX, if you unzip using Archiver, you also have to make the build files executable:

Maven

```
Local installation of mvn is required.
```

- On Windows, use a tool such as File Explorer to extract the project.

Running the example

To run Hello World:

1. In a console, change directories to the top level of the unzipped project.

   For example, if you used the default project name, akka-grpc-quickstart-java, and extracted the project to your root directory, from the root directory, enter: `cd akka-grpc-quickstart-java`

2. Compile the project by entering:

   sbt **Maven** **Gradle**

   ```
   mvn compile
   ```

   Maven downloads project dependencies, generates gRPC classes from protobuf, and compiles.

3. Run the server:

   sbt **Maven** **Gradle**

   ```
   mvn compile dependency:properties exec:exec@server
   ```

Maven runs the `com.example.helloworld.GreeterServer` main class that starts the gRPC server. The `exec:exec@server` execution is defined in the Maven `pom.xml` build definition.

The output should include something like:

```
gRPC server bound to: /127.0.0.1:8080
```

4. Run the client, open another console window and enter:

sbt **Maven** **Gradle**

```
mvn compile dependency:properties exec:exec@client
```

Maven runs the `com.example.helloworld.GreeterClient` main class that starts the gRPC client. The `exec:exec@client` execution is defined in the Maven `pom.xml` build definition.

The output should include something like:

```
Performing request: Alice
Performing request: Bob
HelloReply(Hello, Bob)
HelloReply(Hello, Alice)
```

Congratulations, you just ran your first Akka gRPC server and client. Now take a look at what happened under the covers.

You can end the programs with `ctrl-c`.

What Hello World does

As you saw in the console output, the example outputs several greetings. Let’s take at the code and what happens at runtime.

Server

First, the `GreeterServer` main class creates an `akka.actor.typed.ActorSystem`, a container in which Actors, Akka Streams and Akka HTTP run. Next, it defines a function from `HttpRequest` to `CompletionStage<HttpResponse>` using the `GreeterServiceImpl`. This function handles gRPC requests in the HTTP/2 with TLS server that is bound to port 8080 in this example.

copy
```java
import akka.actor.typed.ActorSystem;
import akka.actor.typed.javadsl.Behaviors;
import akka.http.javadsl.*;
import akka.http.javadsl.model.HttpRequest;
import akka.http.javadsl.model.HttpResponse;
import akka.japi.function.Function;
import com.typesafe.config.Config;
import com.typesafe.config.ConfigFactory;

import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.security.KeyFactory;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.SecureRandom;
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;
import java.security.spec.PKCS8EncodedKeySpec;
import java.util.Base64;
import java.util.concurrent.CompletionStage;

public class GreeterServer {

  public static void main(String[] args) throws Exception {
    // important to enable HTTP/2 in ActorSystem's config
    Config conf = ConfigFactory.parseString("akka.http.server.preview.
      .withFallback(ConfigFactory.load());
    ActorSystem<Void> system = ActorSystem.create(Behaviors.empty(), '
    new GreeterServer(system).run();
  }

  final ActorSystem<?> system;
```

```
    public GreeterServer(ActorSystem<?> system) {
      this.system = system;
    }

    public CompletionStage<ServerBinding> run() throws Exception {

      Function<HttpRequest, CompletionStage<HttpResponse>> service =
          GreeterServiceHandlerFactory.create(
              new GreeterServiceImpl(system),
              system);

      CompletionStage<ServerBinding> bound =
              Http.get(system)
                      .newServerAt("127.0.0.1", 8080)
                      .enableHttps(serverHttpContext())
                      .bind(service);

      bound.thenAccept(binding ->
          System.out.println("gRPC server bound to: " + binding.localAdd
      );

      return bound;
    }
  }
```

`GreeterServiceImpl` is our implementation of the gRPC service, but first we must define the interface of the service in the protobuf file `src/main/proto/helloworld.proto`:

```
copysyntax = "proto3";

option java_multiple_files = true;
option java_package = "com.example.helloworld";
option java_outer_classname = "HelloWorldProto";

// The greeting service definition.
service GreeterService {
    // 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;
}
```

When compiling the project several things are generated from the proto definition. You can find the generated files in `target/generated-sources/` if you are curious.

For the server the following classes are generated:

- Message classes, such as `HelloRequest` and `HelloReply`
- `GreeterService` interface of the service
- `GreeterServiceHandler` utility to create the `HttpRequest` to `HttpResponse` function from the `GreeterServiceImpl`

The part that we have to implement on the server side is the `GreeterServiceImpl` which implements the generated `GreeterService` interface. It is this implementation that is bound to the `HTTP` server via the `GreeterServiceHandler` and it looks like this:

copy
```
import akka.NotUsed;
import akka.japi.Pair;
import akka.actor.typed.ActorSystem;
import akka.stream.javadsl.BroadcastHub;
import akka.stream.javadsl.Keep;
import akka.stream.javadsl.MergeHub;
import akka.stream.javadsl.Sink;
import akka.stream.javadsl.Source;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

class GreeterServiceImpl implements GreeterService {

  final ActorSystem<?> system;

  public GreeterServiceImpl(ActorSystem<?> system) {
    this.system = system;
  }

  @Override
  public CompletionStage<HelloReply> sayHello(HelloRequest request) {
    return CompletableFuture.completedFuture(
        HelloReply.newBuilder()
            .setMessage("Hello, " + request.getName())
            .build()
    );
  }

}
```

Client

In this example we have the client in the same project as the server. That is common for testing purposes but for real usage you or another team would have a separate project (different service) that is using the client and doesn't implement the server side of the service. Between such projects you would only share the proto file (by copying it).

From the same proto file that was used on the server side classes are generated for the client:

- Message classes, such as `HelloRequest` and `HelloReply`
- `GreeterService` interface of the service
- `GreeterServiceClient` that implements the client side of the `GreeterService`

On the client side we don't have to implement anything, the `GreeterServiceClient` is ready to be used as is.

We need an `ActorSystem` and then the `GreeterServiceClient` can be created and used like this:

```
copyimport akka.Done;
import akka.NotUsed;
import akka.japi.Pair;
import akka.actor.typed.ActorSystem;
import akka.actor.typed.javadsl.Behaviors;
import akka.grpc.GrpcClientSettings;
import akka.stream.javadsl.Source;

import java.time.Duration;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CompletionStage;

import static akka.NotUsed.notUsed;
class GreeterClient {

  public static void main(String[] args) {
    final ActorSystem<Void> system = ActorSystem.create(Behaviors.empt

    GreeterServiceClient client = GreeterServiceClient.create(
        GrpcClientSettings.fromConfig("helloworld.GreeterService", sys
        system
    );

    final List<String> names;
    if (args.length == 0) {
      names = Arrays.asList("Alice", "Bob");
    } else {
      names = Arrays.asList(args);
    }

    names.forEach(name -> {
      System.out.println("Performing request: " + name);
      HelloRequest request = HelloRequest.newBuilder()
          .setName(name)
          .build();
      CompletionStage<HelloReply> replyCS = client.sayHello(request);
```

```
        replyCS.whenComplete((reply, error) -> {
          if (error == null) {
            System.out.println(reply.getMessage());
          } else {
            System.out.println(error.getMessage());
          }
        });
      });
    }
  }
```

Note that clients and servers don't have to be implemented with Akka gRPC. They can be implemented/used with other libraries or languages and interoperate according to the gRPC specification.

Other types of calls

In this first example we saw a gRPC service call for single request returning a `CompletionStage` reply. The parameter and return type of the calls may also be streams in 3 different combinations:

- **client streaming call** - `Source` (stream) of requests from the client that returns a `CompletionStage` with a single response, see `itKeepsTalking` in above example
- **server streaming call** - single request that returns a `Source` (stream) of responses, see `itKeepsReplying` in above example
- **client and server streaming call** - `Source` (stream) of requests from the client that returns a `Source` (stream) of responses, see `streamHellos` in above example

As next step, let's try the **bidirectional streaming calls (streaming.html)**.

Next: *Streaming gRPC (streaming.html)*

TECH HUB	**LIGHTBEND**
Guides (https://developer.lightbend.com/guides/)	Subscription (https://www.lightbend.com/subscription/)
Docs (https://developer.lightbend.com/docs/)	Training (https://www.lightbend.com/services/training)
Blog (https://developer.lightbend.com/blog/)	Consulting (https://www.lightbend.com/services/consulting)
Forums (https://discuss.lightbend.com/)	About Us (https://www.lightbend.com/about)
Get Started (https://developer.lightbend.com/start/)	

FOLLOW LIGHTBEND

(http://www.facebook.com/lightbendOfficial)

(http://twitter.com/intent/follow?source=followbutton&variant=1.0&screen_name=lightbend)

(http://www.linkedin.com/company/lightbend-inc-) **(http://www.youtube.com/c/Lightbend-TV?sub_confirmation=1)**

(http://www.twitch.tv/lightbendtwitch) **(https://www.lightbend.com/blog/rss.xml)** **(https://github.com/lightbend)**