

Getting Started with gRPC

This page explains how to start using gRPC in your Quarkus application. While this page describes how to configure it with Maven, it is also possible to use Gradle.

Let's imagine you have a regular Quarkus project, generated from the [Quarkus project generator](#). The default configuration is enough, but you can also select some extensions if you want.

Configuring your project

Edit the `pom.xml` file to add the Quarkus gRPC extension dependency (just under `<dependencies>`):

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-grpc</artifactId>
</dependency>
```

Make sure you have `generate-code` goal of `quarkus-maven-plugin` enabled in your `pom.xml`. If you wish to generate code from different `proto` files for tests, also add the `generate-code-tests` goal:

```
<build>
  <plugins>
    <plugin>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>generate-code</goal>
            <goal>generate-code-tests</goal>
            <goal>build</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

With this configuration, you can put your service and message definitions in the `src/main/proto` directory. `quarkus-maven-plugin` will generate Java files from your `proto` files. Alternatively to using the `prepare` goal of the `quarkus-maven-plugin`, you can use `protobuf-maven-plugin` to generate these files, more in [Generating Java files from proto with protobuf-maven-plugin](#)

Let's start with a simple *Hello* service. Create the `src/main/proto/helloworld.proto` file with the following content:

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.quarkus.example";
option java_outer_classname = "HelloWorldProto";

package helloworld;

// The greeting 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;
}
```

This `proto` file defines a simple service interface with a single method (`SayHello`), and the exchanged messages (`HelloRequest` containing the name and `HelloReply` containing the greeting message).

Before coding, we need to generate the classes used to implement and consume gRPC services. In a terminal, run:

```
$ mvn compile
```

Once generated, you can look at the `target/generated-sources/protobuf` directory:

```

target/generated-sources/protobuf
├── grpc-java
│   └── io
│       └── quarkus
│           └── example
│               └── GreeterGrpc.java
└── java
    └── io
        └── quarkus
            └── example
                ├── HelloReply.java
                ├── HelloReplyOrBuilder.java
                ├── HelloRequest.java
                ├── HelloRequestOrBuilder.java
                ├── HelloWorldProto.java
                └── MutinyGreeterGrpc.java

```

These are the classes we are going to use.



Every time you update the `proto` files, you need to re-generate the classes (using `mvn compile`).

Implementing a gRPC service

Now that we have the generated classes let's implement our *hello* service.

With Quarkus, implementing a service requires to *extend* the generated service base implementation and expose it as a `@Singleton` CDI bean.



Don't use `@ApplicationScoped` as the gRPC service implementation cannot be proxied.

Implementing a service

Create the `src/main/java/org/acme/HelloService.java` file with the following content:

```

package org.acme;

import io.grpc.stub.StreamObserver;
import io.quarkus.example.GreeterGrpc;
import io.quarkus.example.HelloReply;
import io.quarkus.example.HelloRequest;

import javax.inject.Singleton;

@Singleton
①
public class HelloService extends GreeterGrpc.GreeterImplBase {
②

    @Override
    public void sayHello(HelloRequest request,
StreamObserver<HelloReply> responseObserver) { ③
        String name = request.getName();
        String message = "Hello " + name;

responseObserver.onNext(HelloReply.newBuilder().setMessage(message)
.build()); ④
        responseObserver.onCompleted();
⑤
    }
}

```

1. Expose your implementation as bean
2. Extends the `ImplBase` class. This is a generated class.
3. Implement the methods defined in the service definition (here we have a single method)
4. Build and send the response
5. Close the response

Quarkus also provides an additional model with Mutiny, a Reactive Programming API integrated in Quarkus. Learn more about Mutiny on the [Getting Started with Reactive guide](#). A Mutiny implementation of this service would be:

```

package org.acme;

import io.quarkus.example.HelloReply;
import io.quarkus.example.HelloRequest;
import io.quarkus.example.MutinyGreeterGrpc;
import io.smallrye.mutiny.Uni;

import javax.inject.Singleton;

@Singleton
public class ReactiveHelloService extends
MutinyGreeterGrpc.GreeterImplBase {

    @Override
    public Uni<HelloReply> sayHello(HelloRequest request) {
        return Uni.createFrom().item(() ->
            HelloReply.newBuilder().setMessage("Hello " +
request.getName()).build()
        );
    }
}

```

The main differences are the following:

- it extends the `ImplBase` from `MutinyGreeterGrpc` instead of `GreeterGrpc`
- the signature of the method is using Mutiny types

The gRPC server

The services are *served* by a *server*. Available services (*CDI beans*) are automatically registered and exposed.

By default, the server is exposed on `localhost:9000`, and uses *plain-text* (so no TLS).

Run the application using: `mvn quarkus:dev`.

Consuming a gRPC service

In this section, we are going to consume the service we expose. To simplify, we are going to consume the service from the same application, which in the real world, does not make sense.

Open the existing `org.acme.ExampleResource` class, and edit the content to become:

```

package org.acme;

import io.quarkus.example.GreeterGrpc;
import io.quarkus.example.HelloRequest;
import io.quarkus.grpc.runtime.annotations.GrpcService;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class ExampleResource {

    @Inject
    @GrpcService("hello")
    GreeterGrpc.GreeterBlockingStub client;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "hello";
    }

    @GET
    @Path("/{name}")
    public String hello(@PathParam("name") String name) {
        return
client.sayHello(HelloRequest.newBuilder().setName(name).build()).ge
tMessage();
    }
}

```

1. Inject the service and configure its name. This name is used in the application configuration
2. Use the *blocking* stub (also a generated class)
3. Invoke the service

We need to configure the application to indicate where is the `hello` service. In the `src/main/resources/application.properties` file, add the following property:

```
quarkus.grpc.clients.hello.host=localhost
```

`hello` is the name of the service used in the `@GrpcService` annotation `host` configures the service host (here it's localhost).

Then, open <http://localhost:8080/hello/quarkus> in a browser, and you should get **Hello quarkus!**

Packaging the application

Like any other Quarkus applications, you can package it with: `mvn package`. You can also package the application into a native executable with: `mvn package -Pnative`.

Generating Java files from proto with protobuf-maven-plugin

Alternatively to using Quarkus code generation to generate stubs for `proto` files, you can also use `protobuf-maven-plugin`.

To do it, first define the 2 following properties in the `<properties>` section:

```
<grpc.version>1.30.2</grpc.version>
<protoc.version>3.12.3</protoc.version>
```

They configure the gRPC version and the `protoc` version.

Then, add to the `build` section the `os-maven-plugin` extension and the `protobuf-maven-plugin` configuration.

```
<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>${os-maven-plugin-version}</version>
    </extension>
  </extensions>

  <plugins>
    <plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId> ①
      <version>${protobuf-maven-plugin-version}</version>
      <configuration>

<protocArtifact>com.google.protobuf:protoc:${protoc.version}:exe:${
os.detected.classifier}</protocArtifact> ②
      <pluginId>grpc-java</pluginId>
      <pluginArtifact>io.grpc:protoc-gen-grpc-
java:${grpc.version}:exe:${os.detected.classifier}</pluginArtifact>
      <protocPlugins>
```

```

        <protocPlugin>
            <id>quarkus-grpc-protoc-plugin</id>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-grpc-protoc-
plugin</artifactId>
            <version>1.7.2.Final</version>

<mainClass>io.quarkus.grpc.protoc.plugin.MutinyGrpcGenerator</mainC
lass>
        </protocPlugin>
    </protocPlugins>
</configuration>
<executions>
    <execution>
        <id>compile</id>
        <goals>
            <goal>compile</goal>
            <goal>compile-custom</goal>
        </goals>
    </execution>
    <execution>
        <id>test-compile</id>
        <goals>
            <goal>test-compile</goal>
            <goal>test-compile-custom</goal>
        </goals>
    </execution>
</executions>
</plugin>

    <!-- ... -->
</plugins>
</build>

```

1. The `protobuf-maven-plugin` that generates stub classes from your gRPC service definition (`proto` files).
2. The class generation uses a tool named `protoc`, which is OS-specific. That's why we use the `os-maven-plugin` to target the executable compatible with the operating system.



This configuration instructs the `protobuf-maven-plugin` to generate the default gRPC classes and classes using Mutiny to fit with the Quarkus development experience.