# Implementing a gRPC Service

gRPC service implementations exposed as beans are automatically registered and served by quarkus-grpc.

> ❗ Implementing a gRPC service requires the gRPC classes to be generated. Place your `proto` files in `src/main/proto` and run `mvn compile`.

## Implementation base

The generation has created 2 implementation bases:

1. One using the default gRPC API

2. One using the Mutiny API

The first classname is structured as follows: `${NAME_OF_THE_SERVICE}Grpc.${NAME_OF_THE_SERVICE}ImplBase`. The second classname is structured as follows: `Mutiny${NAME_OF_THE_SERVICE}Grpc.${NAME_OF_THE_SERVICE}ImplBase`.

For example, if you use `Greeter` as service name as in:

```
service Greeter {
    rpc SayHello (HelloRequest) returns (HelloReply) {}
}
```

The regular implementation base is: `GreeterGrpc.GreeterImplBase`. The second implementation base is: `MutinyGreeterGrpc.GreeterImplBase`.

Note that these classes are not interfaces but regular classes. When extending them, you need to override the service methods defined in the service definition.

## Implementing a service with the default gRPC API

To implement a gRPC service using the default gRPC API, create a class extending the default implementation base. Then, override the methods defined in the service interface. Finally, expose the service as a CDI bean using the `@Singleton` annotation:

```
import javax.inject.Singleton;

@Singleton
public class HelloService extends GreeterGrpc.GreeterImplBase {

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

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

## Implementing a service with the Mutiny API

To implement a gRPC service using the Mutiny gRPC API, create a class extending the Mutiny implementation base. Then, override the methods defined in the service interface. These methods are using Mutiny types. Finally, expose the service as a CDI bean using the @Singleton annotation:

```
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()
        );
    }
}
```

## Handling streams

gRPC allows receiving and returning streams:

```
service Streaming {
    rpc Source(Empty) returns (stream Item) {} // Returns a stream
    rpc Sink(stream Item) returns (Empty) {}   // Reads a stream
    rpc Pipe(stream Item) returns (stream Item) {}  // Reads a
streams and return a streams
}
```

Using Mutiny, you can implement these as follows:

```java
@Singleton
public class StreamingService extends
MutinyStreamingGrpc.StreamingImplBase {

    @Override
    public Multi<Item> source(Empty request) {
        // Just returns a stream emitting an item every 2ms and
stopping after 10 items.
        return
Multi.createFrom().ticks().every(Duration.ofMillis(2))
                .transform().byTakingFirstItems(10)
                .map(l ->
Item.newBuilder().setValue(Long.toString(l)).build());
    }

    @Override
    public Uni<Empty> sink(Multi<Item> request) {
        // Reads the incoming streams, consume all the items.
        return request
                .map(Item::getValue)
                .map(Long::parseLong)
                .collectItems().last()
                .map(l -> Empty.newBuilder().build());
    }

    @Override
    public Multi<Item> pipe(Multi<Item> request) {
        // Reads the incoming stream, compute a sum and return the
cumulative results
        // in the outbound stream.
        return request
                .map(Item::getValue)
                .map(Long::parseLong)
                .onItem().scan(() -> 0L, Long::sum)
                .onItem().transform(l ->
Item.newBuilder().setValue(Long.toString(l)).build());
    }
}
```

# Health check

For the exposed services, Quarkus gRPC exposes health information in the following format:

```
syntax = "proto3";

package grpc.health.v1;

message HealthCheckRequest {
  string service = 1;
}

message HealthCheckResponse {
  enum ServingStatus {
    UNKNOWN = 0;
    SERVING = 1;
    NOT_SERVING = 2;
  }
  ServingStatus status = 1;
}

service Health {
  rpc Check(HealthCheckRequest) returns (HealthCheckResponse);

  rpc Watch(HealthCheckRequest) returns (stream
HealthCheckResponse);
}
```

Clients can specify the fully qualified service name to get the health status of a specific service or skip specifying the service name to get the general status of the gRPC server.

For more details, check out the gRPC documentation

Additionally, if Quarkus SmallRye Health is added to the application, a readiness check for the state of the gRPC services will be added to the MicroProfile Health endpoint response, that is `/health`.

# Reflection Service

Quarkus gRPC Server implements the reflection service. This service allows tools like grpcurl or grpcox to interact with your services.

The reflection service is enabled by default in *dev* mode. In test or production mode, you need to enable it explicitly by setting `quarkus.grpc-server.enable-reflection-service` to `true`.

# Scaling

By default, quarkus-grpc starts a single gRPC server running on a single event loop.

If you wish to scale your server, you can set the number of server instances by setting `quarkus.grpc.server.instances`.

# Server configuration

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

| Configure the gRPC server | Type | Default |
|---|---|---|
| `quarkus.grpc.server.port`<br><br>The gRPC Server port. | int | 9000 |
| `quarkus.grpc.server.host`<br><br>The gRPC server host. | string | 0.0.0.0 |
| `quarkus.grpc.server.handshake-timeout`<br><br>The gRPC handshake timeout. | Duration ❓ | |
| `quarkus.grpc.server.max-inbound-message-size`<br><br>The max inbound message size in bytes. | int | |
| `quarkus.grpc.server.ssl.certificate`<br><br>The file path to a server certificate or certificate chain in PEM format. | path | |
| `quarkus.grpc.server.ssl.key`<br><br>The file path to the corresponding certificate private key file in PEM format. | path | |
| `quarkus.grpc.server.ssl.key-store`<br><br>An optional key store which holds the certificate information instead of specifying separate files. | path | |
| `quarkus.grpc.server.ssl.key-store-type`<br><br>An optional parameter to specify the type of the key store file. If not given, the type is automatically detected based on the file name. | string | |
| `quarkus.grpc.server.ssl.key-store-password`<br><br>A parameter to specify the password of the key store file. If not given, the default ("password") is used. | string | password |

| | | |
|---|---|---|
| `quarkus.grpc.server.ssl.trust-store`<br><br>An optional trust store which holds the certificate information of the certificates to trust | path | |
| `quarkus.grpc.server.ssl.trust-store-type`<br><br>An optional parameter to specify type of the trust store file. If not given, the type is automatically detected based on the file name. | string | |
| `quarkus.grpc.server.ssl.trust-store-password`<br><br>A parameter to specify the password of the trust store file. | string | |
| `quarkus.grpc.server.ssl.cipher-suites`<br><br>The cipher suites to use. If none is given, a reasonable default is selected. | list of string | |
| `quarkus.grpc.server.ssl.protocols`<br><br>The list of protocols to explicitly enable. | list of string | `TLSv1.3,TLSv1.2` |
| `quarkus.grpc.server.ssl.client-auth`<br><br>Configures the engine to require/request client authentication. NONE, REQUEST, REQUIRED | `none`, `request`, `required` | `none` |
| `quarkus.grpc.server.plain-text`<br><br>Disables SSL, and uses plain text instead. If disabled, configure the ssl configuration. | boolean | `true` |
| `quarkus.grpc.server.alpn`<br><br>Whether ALPN should be used. | boolean | `true` |
| `quarkus.grpc.server.transport-security.certificate`<br><br>The path to the certificate file. | string | |
| `quarkus.grpc.server.transport-security.key`<br><br>The path to the private key file. | string | |

| | | |
|---|---|---|
| `quarkus.grpc.server.enable-reflection-service`<br><br>Enables the gRPC Reflection Service. By default, the reflection service is only exposed in `dev` mode. This setting allows overriding this choice and enable the reflection service every time. | boolean | `false` |
| `quarkus.grpc.server.instances`<br><br>Number of gRPC server verticle instances. This is useful for scaling easily across multiple cores. The number should not exceed the amount of event loops. | int | `1` |

> **About the Duration format**
>
> The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.
>
> You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, `PT` is implicitly prepended to the value to obtain a standard `java.time.Duration` format.

# Example of configuration

## Enabling TLS

To enable TLS, use the following configuration:

```
quarkus.grpc-server.ssl.certificate=src/main/resources/tls/server.pem
quarkus.grpc-server.ssl.key=src/main/resources/tls/server.key
```

> When SSL/TLS is configured, `plain-text` is automatically disabled.

## TLS with Mutual Auth

To use TLS with mutual authentication, use the following configuration:

```
quarkus.grpc-server.ssl.certificate=src/main/resources/tls/server.pem
quarkus.grpc-server.ssl.key=src/main/resources/tls/server.key
quarkus.grpc-server.ssl.trust-store=src/main/resources/tls/ca.jks
quarkus.grpc-server.ssl.trust-store-password=*****
quarkus.grpc-server.ssl.client-auth=REQUIRED
```