Consuming a gRPC Service

gRPC services can be injected in your application code.

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

Stubs and Injection

gRPC generation provides several stubs, providing different way to consume a service. Quarkus gRPC can inject:

* blocking stubs

* reactive stubs based on Mutiny

In addition, it also can inject the gRPC `io.grpc.Channel`, that lets you create other types of stubs.

```
@Inject @GrpcService("hello-service")
MutinyGreeterGrpc.MutinyGreeterStub mutiny;

@Inject @GrpcService("hello-service")
GreeterGrpc.GreeterBlockingStub blocking;

@Inject @GrpcService("hello-service")
Channel channel;
```

The stub class names are computed from the service name. For example, if you use `Greeter` as service name as in:

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

The Mutiny stub name is: `MutinyGreeterGrpc.MutinyGreeterStub` The blocking stub name is: `GreeterGrpc.GreeterBlockingStub`

Client injection must be qualified using `@GrpcService`. This annotation indicates the configuration prefix used to configure the service. For example, if you set it to `hello-service`, configuring the host of the service is done using `hello-service.host`.

Examples

Using a blocking and mutiny stubs

```
@Inject @GrpcService("hello") GreeterGrpc.GreeterBlockingStub
blockingHelloService;
@Inject @GrpcService("hello") MutinyGreeterGrpc.MutinyGreeterStub
mutinyHelloService;

@GET
@Path("/blocking/{name}")
public String helloBlocking(@PathParam("name") String name) {
    return
blockingHelloService.sayHello(HelloRequest.newBuilder().setName(nam
e).build()).getMessage();
}

@GET
@Path("/mutiny/{name}")
public Uni<String> helloMutiny(@PathParam("name") String name) {
    return
mutinyHelloService.sayHello(HelloRequest.newBuilder().setName(name)
.build())
            .onItem().transform(HelloReply::getMessage);
}
```

Note that in this example, the `quarkus.grpc.clients.hello.host` property must be set.

Handling streams

gRPC allows sending and receiving 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 the Mutiny stub, you can interact with these as follows:

```
package io.quarkus.grpc.example.streaming;

import io.grpc.examples.streaming.Empty;
import io.grpc.examples.streaming.Item;
import io.grpc.examples.streaming.MutinyStreamingGrpc;
import io.quarkus.grpc.runtime.annotations.GrpcService;
```

```java
import io.smallrye.mutiny.Multi;

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("/streaming")
@Produces(MediaType.APPLICATION_JSON)
public class StreamingEndpoint {

    @Inject @GrpcService("streaming")
    MutinyStreamingGrpc.MutinyStreamingStub client;

    @GET
    public Multi<String> invokeSource() {
        // Retrieve a stream
        return client.source(Empty.newBuilder().build())
                .onItem().transform(Item::getValue);
    }

    @GET
    @Path("sink/{max}")
    public Uni<Void> invokeSink(@PathParam("max") int max) {
        // Send a stream and wait for completion
        Multi<Item> inputs = Multi.createFrom().range(0, max)
                .map(i -> Integer.toString(i))
                .map(i -> Item.newBuilder().setValue(i).build());
        return
client.sink(inputs).onItem().ignore().andContinueWithNull();
    }

    @GET
    @Path("/{max}")
    public Multi<String> invokePipe(@PathParam("max") int max) {
        // Send a stream and retrieve a stream
        Multi<Item> inputs = Multi.createFrom().range(0, max)
                .map(i -> Integer.toString(i))
                .map(i -> Item.newBuilder().setValue(i).build());
        return
client.pipe(inputs).onItem().transform(Item::getValue);
    }

}
```

Client configuration

For each gRPC service you inject in your application, you can configure the following attributes:

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

Configures the gRPC clients	Type	Default
`quarkus.grpc.clients."service-name".port` The gRPC service port.	int	9000
`quarkus.grpc.clients."service-name".host` The host name / IP on which the service is exposed.	string	required ❗
`quarkus.grpc.clients."service-name".ssl.certificate` The file path to a server certificate or certificate chain in PEM format.	path	
`quarkus.grpc.clients."service-name".ssl.key` The file path to the corresponding certificate private key file in PEM format.	path	
`quarkus.grpc.clients."service-name".ssl.trust-store` An optional trust store which holds the certificate information of the certificates to trust	path	
`quarkus.grpc.clients."service-name".plain-text` Whether `plain-text` should be used instead of `TLS`. Enables by default, except it TLS/SSL is configured. In this case, `plain-text` is disabled.	boolean	
`quarkus.grpc.clients."service-name".keep-alive-time` The duration after which a keep alive ping is sent.	Duration ❓	
`quarkus.grpc.clients."service-name".flow-control-window` The flow control window in bytes. Default is 1MiB.	int	
`quarkus.grpc.clients."service-name".idle-timeout` The duration without ongoing RPCs before going to idle mode.	Duration ❓	

`quarkus.grpc.clients."service-name".keep-alive-timeout` The amount of time the sender of of a keep alive ping waits for an acknowledgement.	Duration ❓	
`quarkus.grpc.clients."service-name".keep-alive-without-calls` Whether keep-alive will be performed when there are no outstanding RPC on a connection.	boolean	`false`
`quarkus.grpc.clients."service-name".max-hedged-attempts` The max number of hedged attempts.	int	`5`
`quarkus.grpc.clients."service-name".max-retry-attempts` The max number of retry attempts. Retry must be explicitly enabled.	int	`5`
`quarkus.grpc.clients."service-name".max-trace-events` The maximum number of channel trace events to keep in the tracer for each channel or sub-channel.	int	
`quarkus.grpc.clients."service-name".max-inbound-message-size` The maximum message size allowed for a single gRPC frame (in bytes). Default is 4 MiB.	int	
`quarkus.grpc.clients."service-name".max-inbound-metadata-size` The maximum size of metadata allowed to be received (in bytes). Default is 8192B.	int	
`quarkus.grpc.clients."service-name".negotiation-type` The negotiation type for the HTTP/2 connection. Accepted values are: `TLS`, `PLAINTEXT_UPGRADE`, `PLAINTEXT`	string	`TLS`
`quarkus.grpc.clients."service-name".override-authority` Overrides the authority used with TLS and HTTP virtual hosting.	string	
`quarkus.grpc.clients."service-name".per-rpc-buffer-limit` The per RPC buffer limit in bytes used for retry.	long	

`quarkus.grpc.clients."service-name".retry` Whether retry is enabled. Note that retry is disabled by default.	boolean	`false`
`quarkus.grpc.clients."service-name".retry-buffer-size` The retry buffer size in bytes.	long	
`quarkus.grpc.clients."service-name".user-agent` Use a custom user-agent.	string	

ℹ️ *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.

The `service-name` is the name set in the `@GrpcService`.

Example of configuration

The 2 following examples uses *hello* as service name. Don't forget to replace it with the name you used in in the `@GrpcService` annotation.

Enabling TLS

To enable TLS, use the following configuration:

```
quarkus.grpc.clients.hello.host=localhost
quarkus.grpc.clients.hello.ssl.trust-
store=src/main/resources/tls/ca.pem
```

ℹ️ 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.clients.hello.host=localhost
quarkus.grpc.clients.hello.plain-text=false
quarkus.grpc.clients.hello.ssl.certificate=src/main/resources/tls/c
lient.pem
quarkus.grpc.clients.hello.ssl.key=src/main/resources/tls/client.ke
y
quarkus.grpc.clients.hello.ssl.trust-
store=src/main/resources/tls/ca.pem
```