# server.py

**Imports and Setup**

pythonCopy code

from concurrent import futures import grpc import helloworld\_pb2 import helloworld\_pb2\_grpc

* **concurrent.futures**: This module provides a high-level interface for asynchronously executing callables. The **ThreadPoolExecutor** is used to manage a pool of threads for executing calls asynchronously.
* **grpc**: This is the main gRPC module for Python, which provides the necessary functionality to start the server and handle requests.
* **helloworld\_pb2** and **helloworld\_pb2\_grpc**: These modules are generated from the **helloworld.proto** file using the **protoc** compiler with the gRPC Python plugin. **helloworld\_pb2** contains message classes, and **helloworld\_pb2\_grpc** contains server and client classes.

**Greeter Service Implementation**

pythonCopy code

class GreeterServicer(helloworld\_pb2\_grpc.GreeterServicer): def SayHello(self, request, context): return helloworld\_pb2.HelloReply(message='Hello, %s!' % request.name)

* **GreeterServicer** is a class that implements the service defined in the **helloworld.proto** file. It inherits from **helloworld\_pb2\_grpc.GreeterServicer**, which is an abstract base class for the service.
* **SayHello** is the method that actually processes the client requests. It takes a **HelloRequest** object as **request** and a **context** object. The method returns a **HelloReply** object, which is created by passing a greeting message back to the client. The **%s** in the message string is replaced with the **name** attribute from the **request** object.

**Server Setup and Execution**

pythonCopy code

def serve(): server = grpc.server(futures.ThreadPoolExecutor(max\_workers=10)) helloworld\_pb2\_grpc.add\_GreeterServicer\_to\_server(GreeterServicer(), server) server.add\_insecure\_port('[::]:50051') server.start() server.wait\_for\_termination()

* **serve** is a function that configures and starts the gRPC server.
* **grpc.server(futures.ThreadPoolExecutor(max\_workers=10))**: This creates a gRPC server instance with a thread pool executor. The server will use up to 10 threads to execute requests concurrently.
* **helloworld\_pb2\_grpc.add\_GreeterServicer\_to\_server(GreeterServicer(), server)**: This line adds the **GreeterServicer** implementation to the gRPC server, making it ready to handle incoming requests.
* **server.add\_insecure\_port('[::]:50051')**: This tells the server to listen for incoming connections on port 50051 on all interfaces (**[::]** is an IPv6 notation for "all interfaces"). The connection is insecure because it does not use SSL/TLS encryption.
* **server.start()**: Starts the server and begins listening for incoming connections.
* **server.wait\_for\_termination()**: Keeps the server running indefinitely, waiting for a signal to terminate (e.g., from the operating system or an interrupt from the keyboard).

Finally, the **if \_\_name\_\_ == '\_\_main\_\_':** block checks if the script is executed directly (not imported as a module) and, if so, calls the **serve** function to start the server.

This code effectively sets up a basic gRPC server that can handle **SayHello** requests from clients, responding with a personalized greeting message.

# client.py

**Imports**

pythonCopy code

import grpc import helloworld\_pb2 import helloworld\_pb2\_grpc

* **grpc** is the module providing the functionality to create a client that can connect to a gRPC server.
* **helloworld\_pb2** and **helloworld\_pb2\_grpc** are modules generated from the **helloworld.proto** file. **helloworld\_pb2** contains the message classes (**HelloRequest** and **HelloReply**), and **helloworld\_pb2\_grpc** contains client and server classes for the service.

**Client Implementation**

pythonCopy code

def run(): with grpc.insecure\_channel('localhost:50051') as channel: stub = helloworld\_pb2\_grpc.GreeterStub(channel) response = stub.SayHello(helloworld\_pb2.HelloRequest(name='world')) print("Greeter client received: " + response.message)

* **grpc.insecure\_channel('localhost:50051')** creates a channel to connect to the server. The channel is "insecure" because it does not use encryption, which is fine for examples and development environments. The server is expected to be running on **localhost** (the same machine) and listening on port **50051**.
* **helloworld\_pb2\_grpc.GreeterStub(channel)** creates a stub object. A stub is a client-side proxy that provides the same methods as the server. You use this stub to call the server methods as if they were local methods. In this case, the **GreeterStub** corresponds to the **Greeter** service defined in the **.proto** file.
* **stub.SayHello(helloworld\_pb2.HelloRequest(name='world'))** calls the **SayHello** method defined in the **Greeter** service. This method takes a **HelloRequest** object as input, which is created here with **name='world'**. The call is synchronous, meaning the client will wait for the server to respond.
* **print("Greeter client received: " + response.message)** prints the message received from the server. The **response** is an object of type **HelloReply**, which contains a **message** attribute. This message is constructed by the server based on the request sent by the client.

**Main Block**

pythonCopy code

if \_\_name\_\_ == '\_\_main\_\_': run()

* This block checks if the script is being run directly (not imported as a module) and, if so, calls the **run()** function. This is a common pattern in Python scripts to allow them to be used both as modules and as standalone programs.

In summary, this client code demonstrates how to set up a connection to a gRPC server, send a request using a method defined in a **.proto** file, and handle the response. It's a basic example of client-server communication using gRPC with Python.