README

Warehouse in Python

Author: Rahul Gupta
Date: Feb, 13.

Getting started

```
cd warehouse_python
pip install pipenv
pipenv install
```

Start the server

```
pipenv run python src/server.py
```

Start the client in a different terminal session

```
pipenv run python src/client.py
```

Explanation of the entire Project

What is gRPC

gRPC is a high-performance, open-source universal RPC (Remote Procedure Call) framework developed by Google0. It enables the server and client applications to communicate transparently and makes it easier to build connected systems. gRPC works across languages and platforms by using Protocol Buffers (Protobuf) as its interface definition language, allowing developers to define their service methods and message types in a language-neutral, platform-neutral way1. This approach ensures that services built in different languages can communicate with each other seamlessly.

Why Does It Work Across Languages and Platforms?

gRPC is designed to be language-agnostic and platform-independent2. This is achieved through the use of Protocol Buffers, which serve as a universal language for defining service interfaces and message structures. The Protobuf compiler (`protoc`) can generate client and

server code from `.proto` files for a variety of programming languages, enabling developers to create services and clients in their language of choice while ensuring interoperability[3].

RPC Lifecycle Starting With the RPC Client

1. **Service Definition**: Before any RPC calls are made, services and their methods are defined in a `.proto` file using Protocol Buffers syntax[4].
2. **Client Stub Initialization**: The client initializes a stub for the remote service. This stub provides methods that correspond to the service methods defined in the `.proto` file[5].
3. **Making the Call**: The client makes a remote procedure call by invoking a method on the stub, passing the required request parameters as arguments[6].
4. **Request Serialization**: The client stub serializes the request message into a binary format (using Protocol Buffers) and sends it to the server over the network[7].
5. **Server Handling**: The server receives the request, deserializes it, and invokes the appropriate service method[8].
6. **Response**: The server serializes the response into a binary format and sends it back to the client[9].
7. **Client Receives Response**: The client stub deserializes the response, and the call is completed – the client now has the result of the remote procedure call[10].

Workflow of Protocol Buffers

1. **Define Message and Service**: Developers define message formats and service interfaces in a `.proto` file using the Protobuf language[11].
2. **Code Generation**: Using the `protoc` compiler, source code is generated from the `.proto` file for the desired languages. This generated code includes data access classes for each message and client/server code for each service[12].
3. **Implementation**: Developers implement the generated server interfaces and use the client stubs in their application code to make and handle RPCs[13].
4. **Serialization/Deserialization**: Messages are automatically serialized/deserialized from/to the binary format by the client/server stubs when making or receiving RPC calls[14].

Benefits of Using Protocol Buffers

- **Efficiency**: Protobuf is more efficient in both size and speed compared to JSON and XML. It's a binary format, making it faster to serialize/deserialize and requiring less bandwidth[15].
- **Cross-Language Support**: Protobuf supports code generation in multiple languages, enabling seamless communication between services written in different languages[16].
- **Forward and Backward Compatibility**: Developers can add new fields to their message definitions without breaking backward compatibility with services that were compiled

against older definitions[17].

When Is the Use of Protocol Not Recommended?

- **Human Readability**: If human readability of the data format is a primary requirement, Protobuf might not be the best choice due to its binary nature[18].
- **Small Projects or Microservices**: For small-scale projects or microservices where the overhead of defining Protobuf schemas and generating code might not be justified[19].

3 Different Data Types That Can Be Used With Protocol Buffers

1. **Basic Types**: Such as `int32`, `float`, `double`, `bool`, and `string`, which cover most of the primitive data types needed in applications[20].
2. **Complex Types**: Custom defined `message` types, which allow the composition of basic types and other messages to create complex data structures[21].
3. **Enumerations (`enum`)**: These allow the definition of named constants, which can make protocols clearer and more readable by using symbolic names instead of integers[22].

Here is an example of my own code where I have used it:

```
message WarehouseData {
  string warehouse_id = 1;
  string warehouse_name = 2;
  string warehouse_country = 3;
  string warehouse_city = 4;
  string address = 5;
  string timestamp = 6;
  repeated ProductData product_data = 7;
}
```

An enum is very similar to an enum in java or other programming languages you might know. To explain in simple terms it is just a, like the name says, enumeration and is there to have a selection of values to choose from.
If I had to have such an enum in my code it would look like this:

```
proto enum ProductCategory {
  UNKNOWN = 0;
  BEVERAGE = 1;
  SNACK = 2;
  CLEANING_SUPPLY = 3;
}
```

Generate code out of .proto files

```
cd warehouse_python
pipenv run python -m grpc_tools.protoc -I./src --python_out=./src --
grpc_python_out=./src ./src/*.prot
```