Exercise 12

gRPC / Binary protocols

Pre-reqs
None

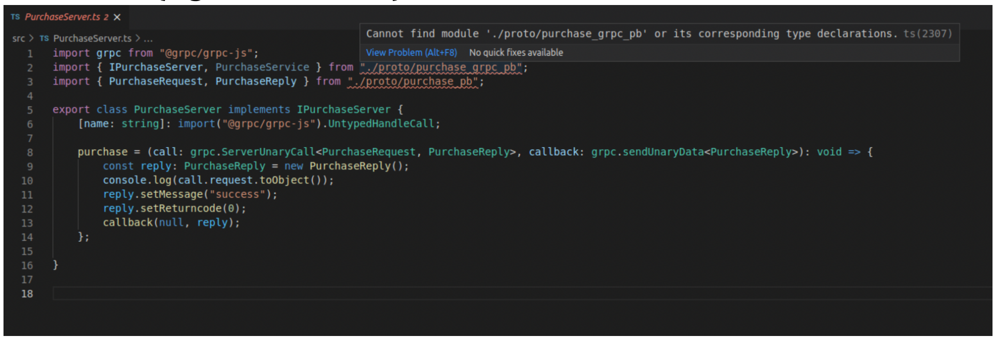
Objectives
Demonstrate the use of a fast binary protocol
Demonstrate a python client calling a typescript service via ProtoBuf messages
Use server-side streaming

Steps

1. Firstly, we need to checkout the code from Github.
   In a new terminal window, type the following:

   ```
   cd ~
   git clone https://github.com/pzfreo/grpc-sample-ts.git
   cd grpc-sample-ts
   yarn install
   ```

2. If you look at the code, you will see that there are some missing definitions (e.g. these errors:)

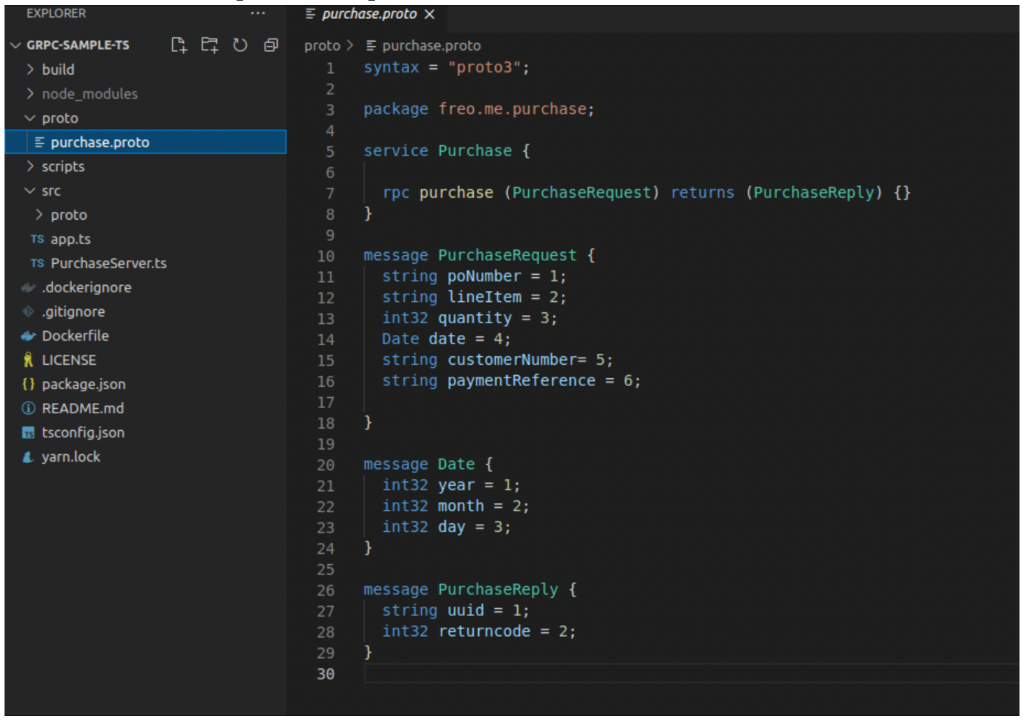


```
Cannot find module './proto/purchase_grpc_pb' or its corresponding type declarations. ts(2307)
```

```
import grpc from "@grpc/grpc-js";
import { IPurchaseServer, PurchaseService } from "./proto/purchase_grpc_pb";
import { PurchaseRequest, PurchaseReply } from "./proto/purchase_pb";

export class PurchaseServer implements IPurchaseServer {
    [name: string]: import("@grpc/grpc-js").UntypedHandleCall;

    purchase = (call: grpc.ServerUnaryCall<PurchaseRequest, PurchaseReply>, callback: grpc.sendUnaryData<PurchaseReply>): void => {
        const reply: PurchaseReply = new PurchaseReply();
        console.log(call.request.toObject());
        reply.setMessage("success");
        reply.setReturncode(0);
        callback(null, reply);
    };

}
```

3. That is because we need to generate code from a protobuf definition using some scripts.

Let's first look at purchase.proto



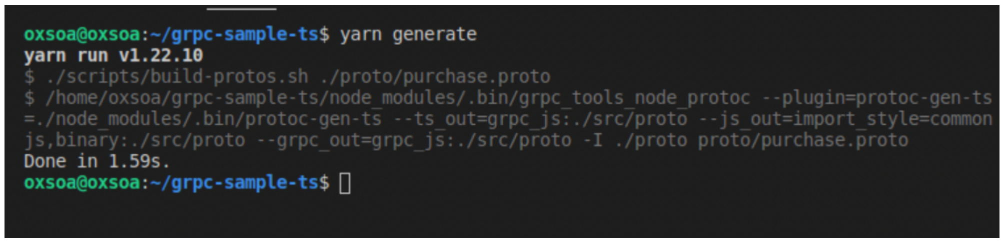
```
syntax = "proto3";

package freo.me.purchase;

service Purchase {

  rpc purchase (PurchaseRequest) returns (PurchaseReply) {}
}

message PurchaseRequest {
  string poNumber = 1;
  string lineItem = 2;
  int32 quantity = 3;
  Date date = 4;
  string customerNumber= 5;
  string paymentReference = 6;

}

message Date {
  int32 year = 1;
  int32 month = 2;
  int32 day = 3;
}

message PurchaseReply {
  string uuid = 1;
  int32 returncode = 2;
}

```

4. This contains the gRPC service definition in the ProtoBuf format.

   You can see there is a simple input message and output message. The information may be similar to the REST model, but the approach is definitely different. There are no resources, no well defined error/return codes, and no standard VERBS. Certainly no HATEOAS!

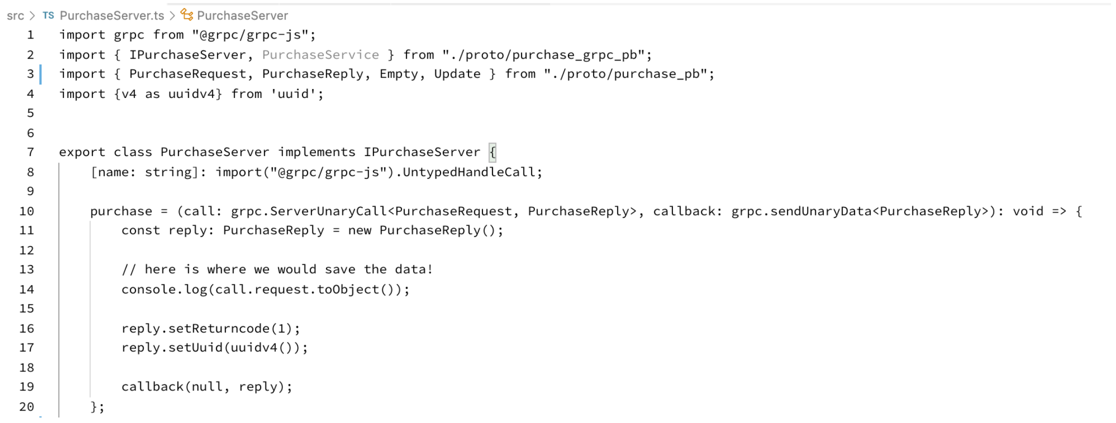
5. Let's generate the code from the proto. There is a yarn script "generate" that calls a script in ./scripts to generate the code using the typescript grpc tools. These are installed by the yarn install command (as they are defined in package.json).

   ```
   yarn generate
   ```



```
oxsoa@oxsoa:~/grpc-sample-ts$ yarn generate
yarn run v1.22.10
$ ./scripts/build-protos.sh ./proto/purchase.proto
$ /home/oxsoa/grpc-sample-ts/node_modules/.bin/grpc_tools_node_protoc --plugin=protoc-gen-ts
=./node_modules/.bin/protoc-gen-ts --ts_out=grpc_js:./src/proto --js_out=import_style=common
js,binary:./src/proto --grpc_out=grpc_js:./src/proto -I ./proto proto/purchase.proto
Done in 1.59s.
oxsoa@oxsoa:~/grpc-sample-ts$ 
```

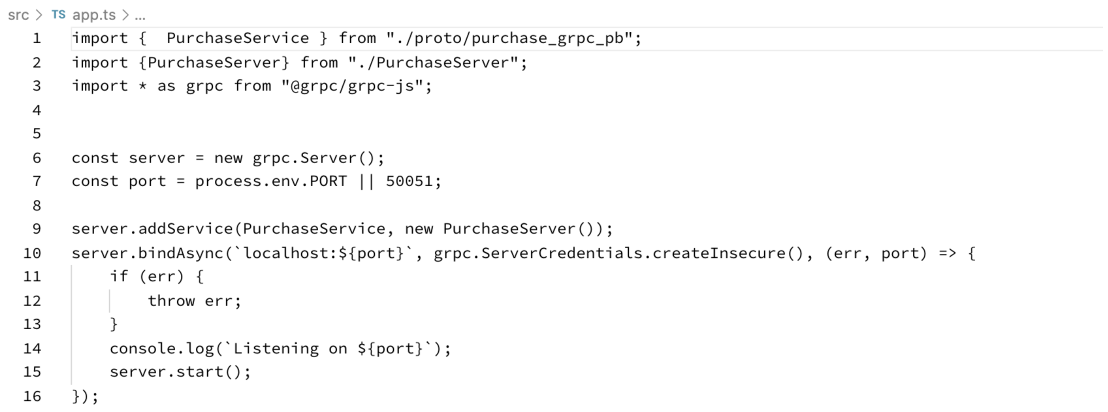
6. Look at the code. There are two key files:



```
import grpc from "@grpc/grpc-js";
import { IPurchaseServer, PurchaseService } from "./proto/purchase_grpc_pb";
import { PurchaseRequest, PurchaseReply, Empty, Update } from "./proto/purchase_pb";
import {v4 as uuidv4} from 'uuid';


export class PurchaseServer implements IPurchaseServer {
    [name: string]: import("@grpc/grpc-js").UntypedHandleCall;

    purchase = (call: grpc.ServerUnaryCall<PurchaseRequest, PurchaseReply>, callback: grpc.sendUnaryData<PurchaseReply>): void => {
        const reply: PurchaseReply = new PurchaseReply();

        // here is where we would save the data!
        console.log(call.request.toObject());

        reply.setReturncode(1);
        reply.setUuid(uuidv4());

        callback(null, reply);
    };
```

As you can see this defines a function "purchase" which creates a reply. You can see the code just prints out the input message and returns a new UUID and rc=1.

7. And then another program file sets up the listener:



```
import {  PurchaseService } from "./proto/purchase_grpc_pb";
import {PurchaseServer} from "./PurchaseServer";
import * as grpc from "@grpc/grpc-js";


const server = new grpc.Server();
const port = process.env.PORT || 50051;

server.addService(PurchaseService, new PurchaseServer());
server.bindAsync(`localhost:${port}`, grpc.ServerCredentials.createInsecure(), (err, port) => {
    if (err) {
        throw err;
    }
    console.log(`Listening on ${port}`);
    server.start();
});
```

8. You can run this code using:

   yarn start

9. Before we create a client, let's try a generic client. There are a couple of useful ones.

   One is called grpcurl which is clearly meant to be like cURL.

   We can install it either by downloading the binary or using go:

```
go get github.com/fullstorydev/grpcurl/...
go install github.com/fullstorydev/grpcurl/cmd/grpcurl
```

10. In the directory with your grpc code in, try:

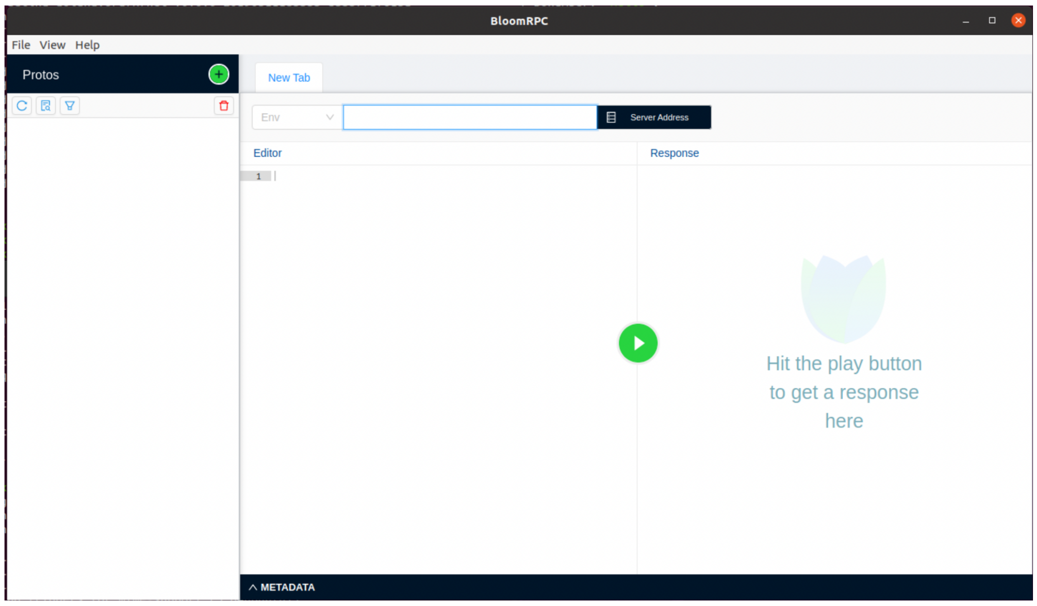
```
grpcurl -proto proto/purchase.proto list

freo.me.purchase.Purchase
```

11. Now try

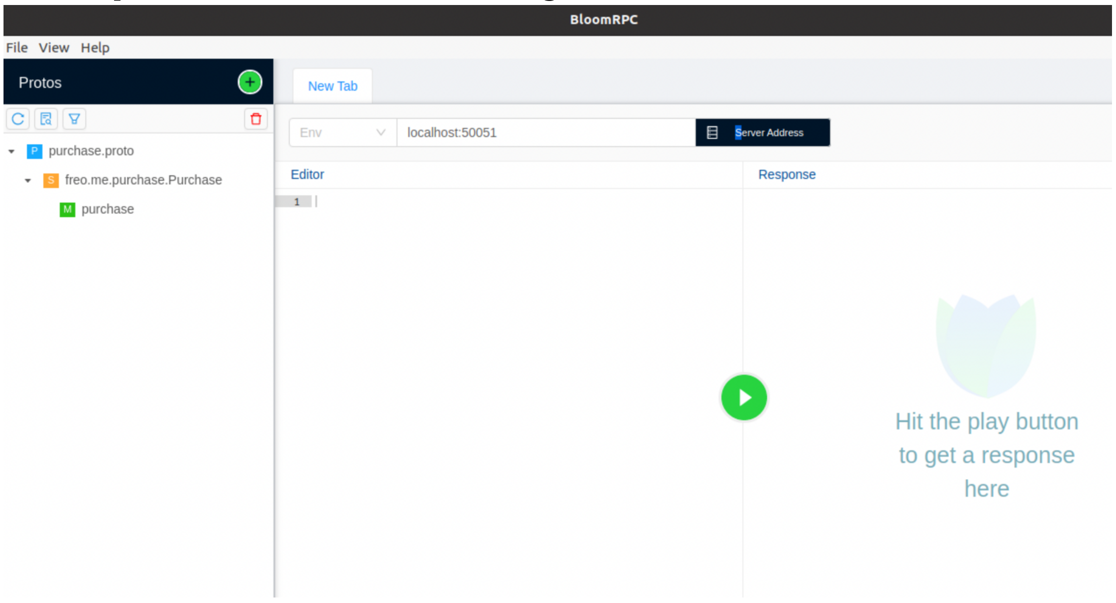
```
grpcurl -proto proto/purchase.proto describe

freo.me.purchase.Purchase is a service:
service Purchase {
  rpc purchase ( .freo.me.purchase.PurchaseRequest )
returns ( .freo.me.purchase.PurchaseReply );
}
```

12. This is quite hard work though. Wouldn't it be nice to have a graphical grpc client?

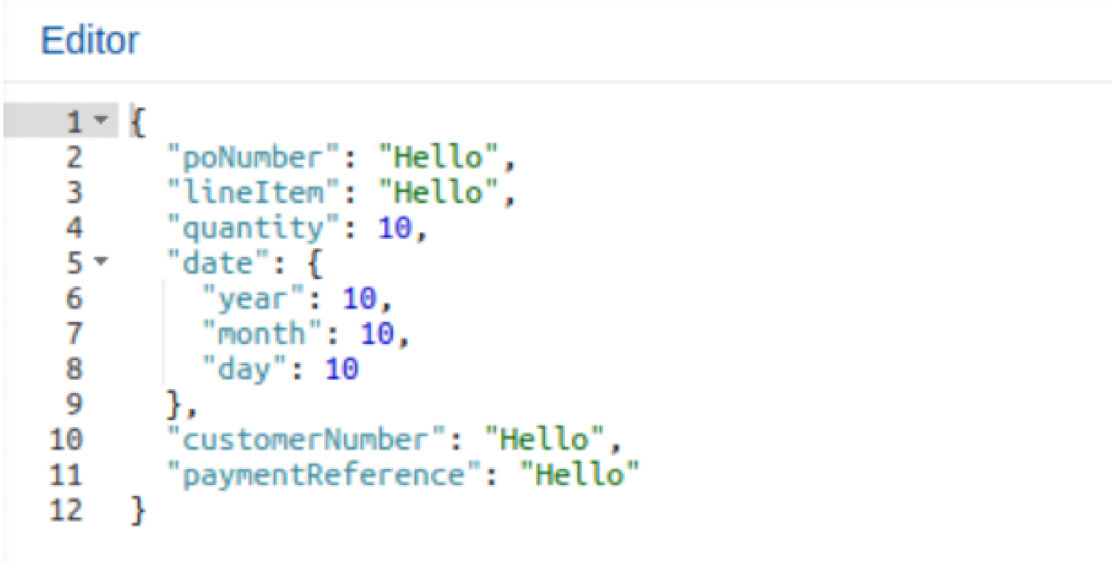
```
cd ~/Downloads
wget http://freo.me/bloomrpc-deb -O bloomrpc.deb
sudo dpkg -i bloomrpc.deb
bloomrpc
```



13. Add the proto file, and set the server target to be localhost:50051

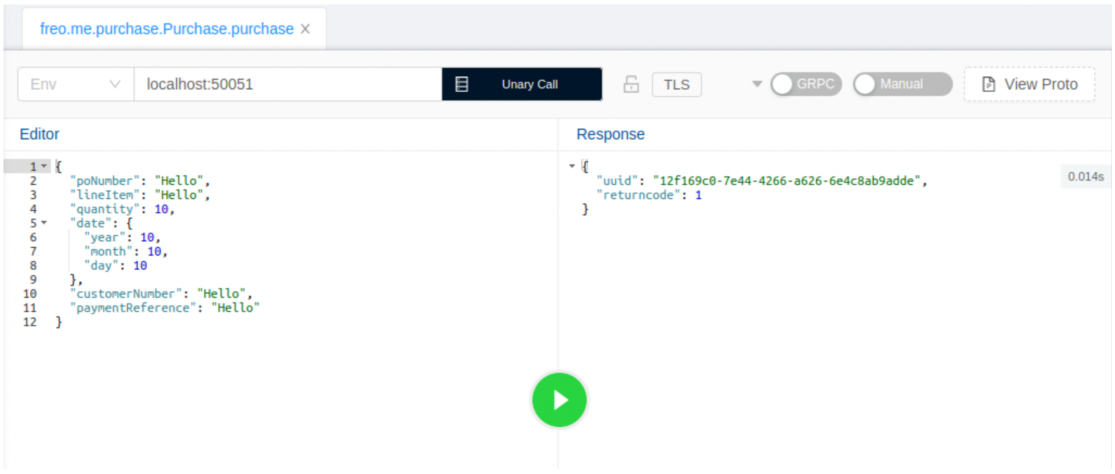


Click on the purchase method and it will autofill a sample:



```
{
    "poNumber": "Hello",
    "lineItem": "Hello",
    "quantity": 10,
    "date": {
      "year": 10,
      "month": 10,
      "day": 10
    },
    "customerNumber": "Hello",
    "paymentReference": "Hello"
}
```

Click play:



```
{
  "poNumber": "Hello",
  "lineItem": "Hello",
  "quantity": 10,
  "date": {
    "year": 10,
    "month": 10,
    "day": 10
  },
  "customerNumber": "Hello",
  "paymentReference": "Hello"
}
```

```
{
  "uuid": "12f169c0-7e44-4266-a626-6e4c8ab9adde",
  "returncode": 1
}
```

You will see how long it takes as well (14ms in my case)

Now let's create a Python client. In a new terminal window:

```
cd ~/grpc-sample-ts
mkdir python
cd python
cp ../proto/purchase.proto .
```

14. We need to install the python grpc tools:

```
sudo pip install grpcio
sudo pip install grpcio-tools
sudo pip install --upgrade protobuf
```

15. Now we can run the client code compiler (all on one line please)

```
python -m grpc_tools.protoc -I .   --python_out=.
--grpc_python_out=. ./purchase.proto
```

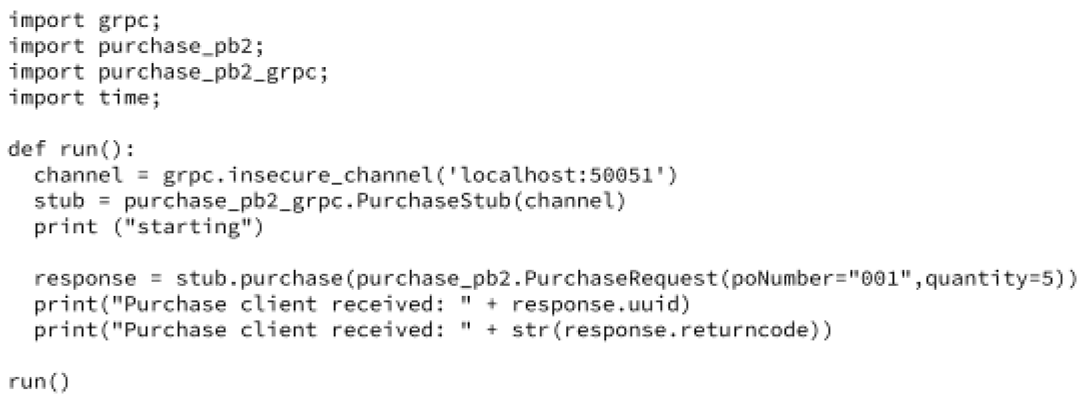
16. If you look at the directory (ls), you will see two new files:

```
> ls -l
total 20
-rw-rw-r-- 1 oxsoa oxsoa 1433 Jan 10 19:24 purchase_pb2_grpc.py
-rw-rw-r-- 1 oxsoa oxsoa 8269 Jan 10 19:24 purchase_pb2.py
-rw-rw-r-- 1 oxsoa oxsoa  446 Jan 10 19:21 purchase.proto
```

17. These files will let us call the purchase service. Create and edit a new python program:

```
code purchase-client.py
```

18. Type in the following code: (available here: http://freo.me/grpc-py)



```
import grpc;
import purchase_pb2;
import purchase_pb2_grpc;
import time;

def run():
  channel = grpc.insecure_channel('localhost:50051')
  stub = purchase_pb2_grpc.PurchaseStub(channel)
  print ("starting")

  response = stub.purchase(purchase_pb2.PurchaseRequest(poNumber="001",quantity=5))
  print("Purchase client received: " + response.uuid)
  print("Purchase client received: " + str(response.returncode))

run()
```

19. Run the program:

```
python purchase-client.py
```

20. Check the console log on your Typescript server and you should see it has been called. Congratulations - we have successfully built a heterogeneous gRPC app.

21. One of the key features of gRPC is the support for client, server and bidirectional streaming. This is really useful, especially for getting push notifications from servers without needing clients to be internet-addressable. Let's try this out.

    First, let's add the following protobuf message definitions to the grpc-sample-ts/purchase.proto files:

```
message Empty {}
message Update {
        string msg = 1;
}
```

22. Also add the following rpc definition:

```
rpc push (Empty) returns (stream Update) {}
```

23. Regenerate the server code:

```
yarn generate
```

24. You will see that the types no longer match in the PurchaseServer.ts
    We need to add a push function:

```
push = async function (call: grpc.ServerWritableStream<Empty, Update>): Promise<void> {

    for (let i = 0; i<10; i++) {
        const u:Update = new Update();
        u.setMsg(new Date().getTime().toString());
        call.write(u);
        await new Promise(resolve => setTimeout(resolve, 1000));
    }
    call.end();
};
```

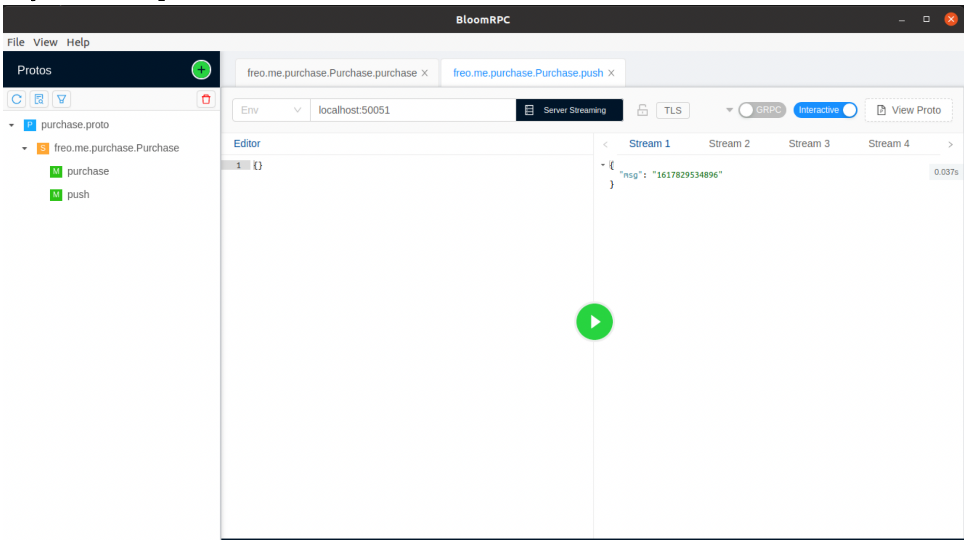
This code will write the time back as a message 10 times, with a 1 second (async) pause between calls. This is simulating a system that occasionally sends updates

25. You will need to fix the imports too.

26. Restart the typescript server.

27. Go back to Bloomrpc and reload the proto file.

28. Try out the push method:



localhost:50051

```
{}
```

```
{
  "msg": "1617829534896"
}
```

Notice the extra responses coming back one second apart:

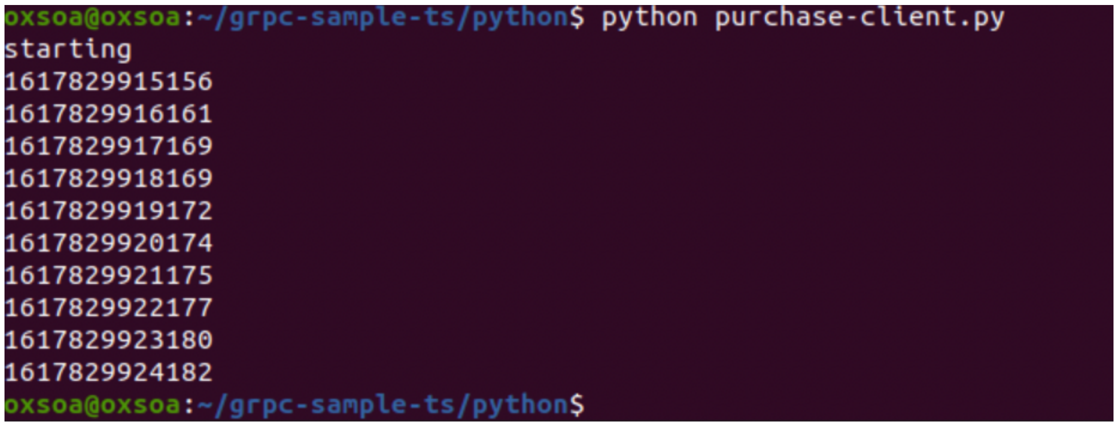
29. Copy the proto to the python directory and regenerate the python bindings.

30. Add this code to your purchase-client.py (and comment out the other call)

```
for update in stub.push(purchase_pb2.Empty()):
    print (update.msg)
```

This is really rather elegant code!

31. You should see streaming working:



```
oxsoa@oxsoa:~/grpc-sample-ts/python$ python purchase-client.py
starting
1617829915156
1617829916161
1617829917169
1617829918169
1617829919172
1617829920174
1617829921175
1617829922177
1617829923180
1617829924182
oxsoa@oxsoa:~/grpc-sample-ts/python$
```

32. Congratulations - you have completed the lab!

Extensions

33. Change the python code to call the server 10000 times. Add a timer around the code: e.g.

```
import time

t0 = time.time()
for x in range(0,1000):
    //code here

t1 = time.time()
t = t1-t0
print("elapsed seconds: " + str(t))
```

34. See how long it takes. Note that if you run it more than once the node server may “warm up”, so the numbers should improve and then stabilize.

35. You could also remove all the console logging in the Typescript and Python code and try again. What is the throughput of requests once the system is warm?

36. How does this compare to our throughput with autocannon? Is this a fair comparison?