Readme

1. How to run this project?

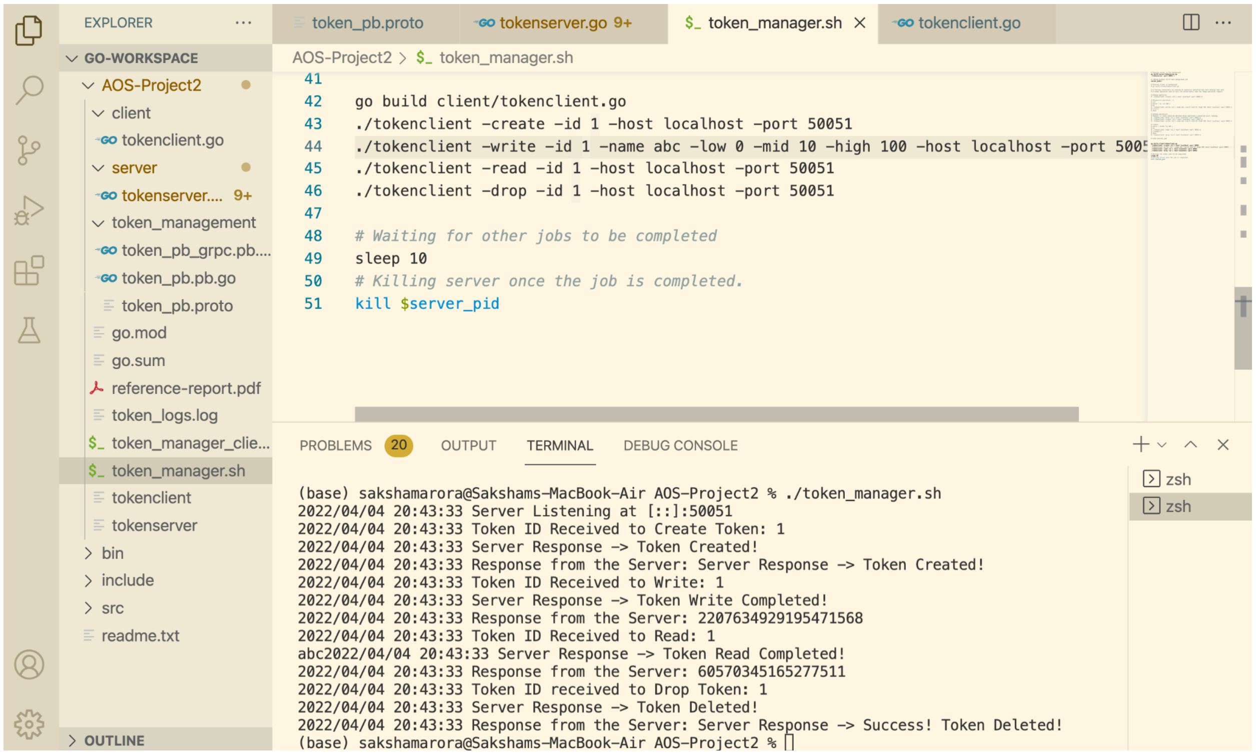
To implement a client-server application for the management of tokens. Firstly, we need to set up protocol buffers to serialize the data. It makes gRPC calls to efficiently use the messages from client to server and vice versa.

To compile the proto buffer file. Run the following command -

protoc --go_out=. --go_opt=paths=source_relative –go-grpc_out=. –go-grpc_opt=paths=source_relative token_management/token_pb.proto

Once the files are created. Run *token_manager.sh* file to start the server and run all the operations from the client end.

./token_manager.sh



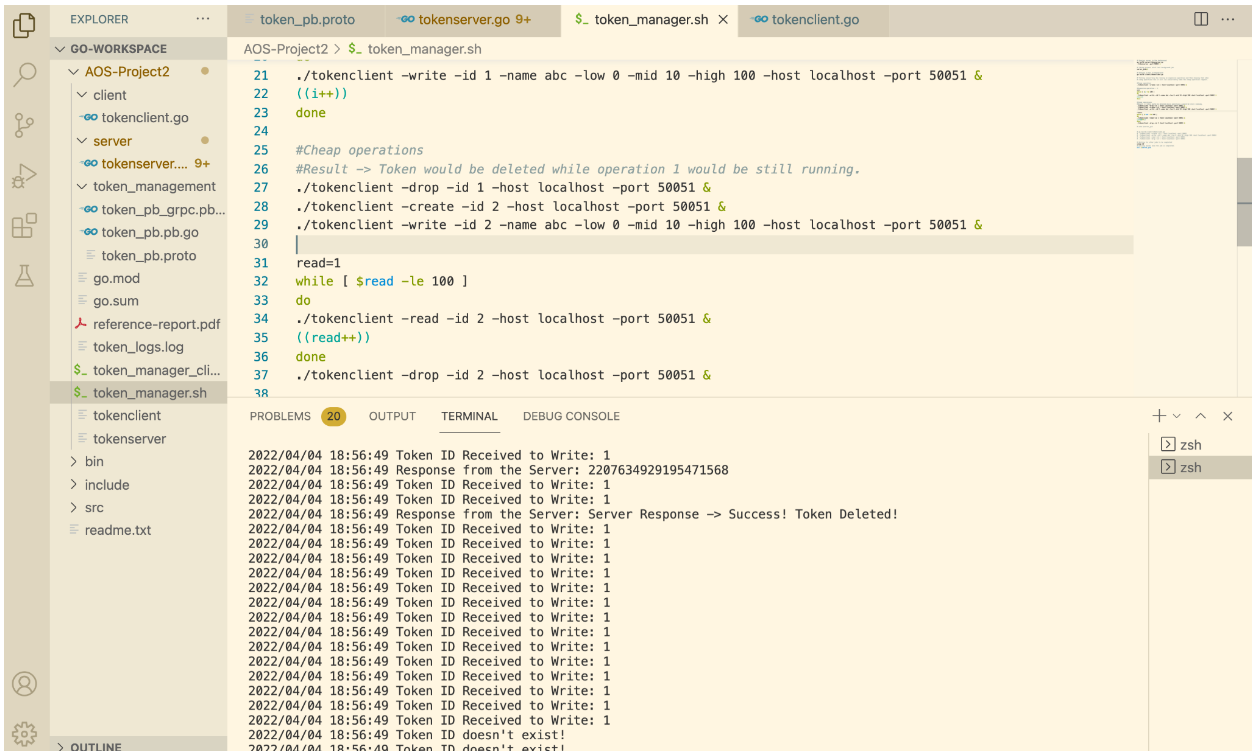
```

go build client/tokenclient.go
./tokenclient -create -id 1 -host localhost -port 50051
./tokenclient -write -id 1 -name abc -low 0 -mid 10 -high 100 -host localhost -port 5005
./tokenclient -read -id 1 -host localhost -port 50051
./tokenclient -drop -id 1 -host localhost -port 50051

# Waiting for other jobs to be completed
sleep 10
# Killing server once the job is completed.
kill $server_pid
```

```
(base) sakshamarora@Sakshams-MacBook-Air AOS-Project2 % ./token_manager.sh
2022/04/04 20:43:33 Server Listening at [::]:50051
2022/04/04 20:43:33 Token ID Received to Create Token: 1
2022/04/04 20:43:33 Server Response -> Token Created!
2022/04/04 20:43:33 Response from the Server: Server Response -> Token Created!
2022/04/04 20:43:33 Token ID Received to Write: 1
2022/04/04 20:43:33 Server Response -> Token Write Completed!
2022/04/04 20:43:33 Response from the Server: 2207634929195471568
2022/04/04 20:43:33 Token ID Received to Read: 1
abc2022/04/04 20:43:33 Server Response -> Token Read Completed!
2022/04/04 20:43:33 Response from the Server: 60570345165277511
2022/04/04 20:43:33 Token ID received to Drop Token: 1
2022/04/04 20:43:33 Server Response -> Token Deleted!
2022/04/04 20:43:33 Response from the Server: Server Response -> Success! Token Deleted!
(base) sakshamarora@Sakshams-MacBook-Air AOS-Project2 % 
```

This demonstrates the running of the bash script. Here the client executes CRUD operations serially.



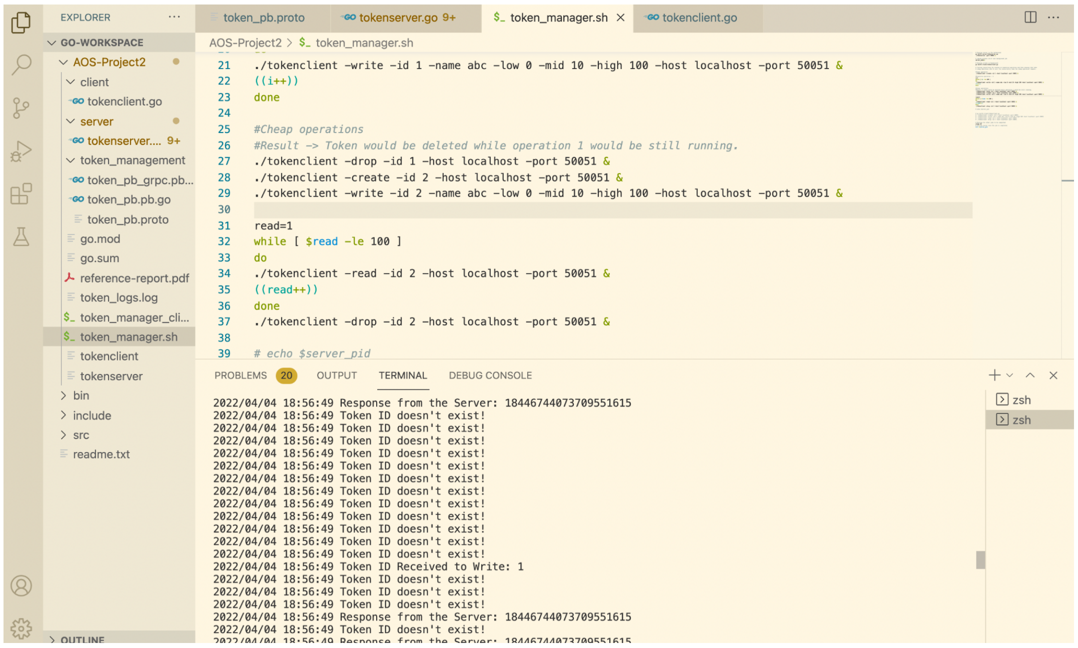
```
./tokenclient -write -id 1 -name abc -low 0 -mid 10 -high 100 -host localhost -port 50051 &
((i++))
done

#Cheap operations
#Result -> Token would be deleted while operation 1 would be still running.
./tokenclient -drop -id 1 -host localhost -port 50051 &
./tokenclient -create -id 2 -host localhost -port 50051 &
./tokenclient -write -id 2 -name abc -low 0 -mid 10 -high 100 -host localhost -port 50051 &

read=1
while [ $read -le 100 ]
do
./tokenclient -read -id 2 -host localhost -port 50051 &
((read++))
done
./tokenclient -drop -id 2 -host localhost -port 50051 &
```

```
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Response from the Server: 2207634929195471568
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Response from the Server: Server Response -> Success! Token Deleted!
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID doesn't exist!
```

This figure demonstrates the concurrency which was tested using the bash script. Here we can see when an expensive operation is working(write operation) and the user executes drop operation(cheap operation) from the new terminal the server will handle the request concurrently. We can see that the write operation was giving us the response but after the drop request the server logs started showing ***Token ID doesn't exist****. In the image below we can see the new partial value coming from the server.*



```
./tokenclient -write -id 1 -name abc -low 0 -mid 10 -high 100 -host localhost -port 50051 &
((i++))
done

#Cheap operations
#Result -> Token would be deleted while operation 1 would be still running.
./tokenclient -drop -id 1 -host localhost -port 50051 &
./tokenclient -create -id 2 -host localhost -port 50051 &
./tokenclient -write -id 2 -name abc -low 0 -mid 10 -high 100 -host localhost -port 50051 &

read=1
while [ $read -le 100 ]
do
./tokenclient -read -id 2 -host localhost -port 50051 &
((read++))
done
./tokenclient -drop -id 2 -host localhost -port 50051 &

# echo $server_pid
```

```
2022/04/04 18:56:49 Response from the Server: 18446744073709551615
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID Received to Write: 1
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Token ID doesn't exist!
2022/04/04 18:56:49 Response from the Server: 18446744073709551615
2022/04/04 18:56:49 Token ID doesn't exist!
```

Concurrency - Concurrency in this project has been achieved with the help of mutexes in golang. Mutex is basically a mutually exclusive lock that can be applied on an operation which may run into conflicting situations. For example - If 2 requests like read and drop arrive at the same time on the same token then the server can’t simply process both the requests parallely because doing so may result in inconsistency in the data.

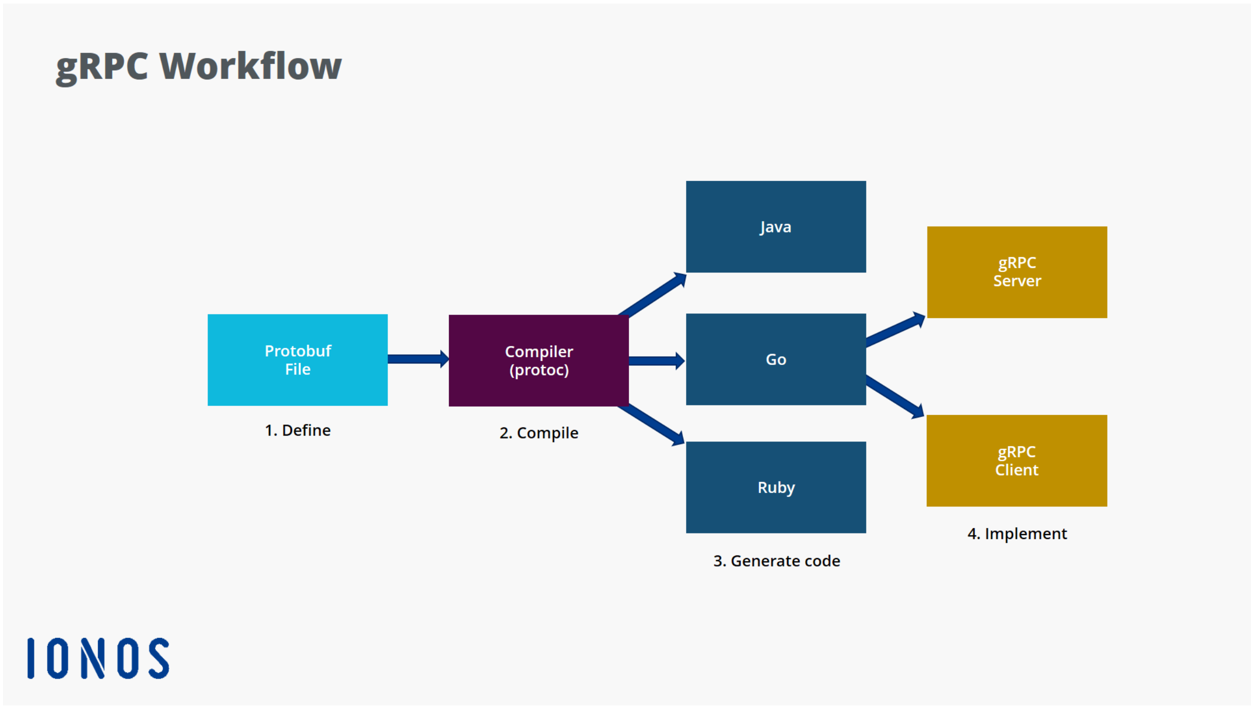
In order to resolve these conflicts we assign locks to each particular token so that whenever such conflicting requests arrive we lock the token first so that no other request can be executed apart from the one currently being executed. Thus it helps us to execute the requests in sequential manner in case of conflict. In case of non conflicting requests read-read we apply RLock rather than simple lock which would allow the read requests to be processed parallely.

2. System Design

Server - Firstly, we create the server on the port mentioned from the command line. It will take care of the requests from the client for creating token, drop token, write token and delete token. Server will concurrently handle the calls made by multiple clients. Hence RWMutex is used to manage each and every token created by the server.

Client - Client will do the handshake with the Server on the defined port. Once a client is connected to the server it can start with operations create, drop, write and read. It will take the configuration from the command line as defined in the bash script.

Proto Buffers - *protobufs* basically provides the interface for the communication between server and client. The *protobuf* files define the structure of each and every object that can be exchanged while making a gRPC call. It provides a blueprint of which functions need to be implemented on the server side. These structures are compiled using the command which was stated above. Once this code is compiled, getters and interfaces are set to exchange messages between client and server.



Reference - https://www.ionos.com/digitalguide/server/know-how/an-introduction-to-grpc/

This figure shows how the protobuf file is compiled and generates interfaces which are used by the client and the server.

3. Documentation

Create Request -

From Client the following request would be made if the user has requested *create* operation.

```
create_req, err := c.CreateNewToken(ctx, &pb.Token{Id: int32(*id_input)})
```

This request would be handled on the server. As shown below-

```
func (s *TokenManagerServer) CreateNewToken(ctx context.Context, in *pb.Token)
(*pb.Response, error)
```

This function would return a response(“Success” or “Failed”). Which is defined as a message in the *token_pb.proto* file. Here *TokenManagerServer* is a pointer to the server’s runtime memory. *Context* provides additional information regarding the client(like *timeouts*). `in *pb.Token` is the request object that client sends obeying the interface structure defined in the proto buf file(*token_pb.proto*).

Drop Request -

From Client the following request would be made if the user has requested *drop* operation.

```
        drop_req, err := c.DropToken(ctx, &pb.Token{Id: int32(*id_input)})
```

This request would be handled on the server. As shown below-

```
func (s *TokenManagerServer) DropToken(ctx context.Context, in *pb.Token)
(*pb.Response, error) {
```

This function would return a response(“Success” or “Failed”). Which is defined as a message in the *token_pb.proto* file. Here *TokenManagerServer* is a pointer to the server’s runtime memory. *Context* provides additional information regarding the client(like *timeouts*). `in *pb.Token` is the request object that client sends obeying the interface structure defined in the proto buf file(*token_pb.proto*).

Write Request -

From Client the following request would be made if the user has requested *write* operation.

```
                        write_req, err := c.WriteToken(ctx, &pb.WriteTokenMsg{Id:
int32(*id_input), Name: *name_input, Low: uint64(*low_input), Mid: uint64(*mid_input),
High: uint64(*high_input)})
```

This request would be handled on the server. As shown below-

```
       func (s *TokenManagerServer) WriteToken(ctx context.Context, in
*pb.WriteTokenMsg) (*pb.WriteResponse, error) {
```

This function would return a partial value whether it’s success or failure. Which is defined as a message in the *token_pb.proto* file. The client sets low, mid and high domain values requested from the user using the command line. `*pb.WriteTokenMsg` is defined in the *token_pb.proto* file which is accessed using *pb* pointer which is pointing to the interface generated by *token_pb.proto* file.

Read Request -

From Client the following request would be made if the user has requested *read* operation.

```
read_req, err := c.ReadToken(ctx, &pb.Token{Id: int32(*id_input)})
```

This request would be handled on the server. As shown below-

```
func (s *TokenManagerServer) ReadToken(ctx context.Context, in *pb.Token)
(*pb.WriteResponse, error) {
```

This function would return a final value whether it's success or failure. Which is defined as a message in the *token_pb.proto* file. The client sets a token id which needs to be read and then sets the final value of the state as mentioned in the project. `*pb.WriteResponse` is defined in the *token_pb.proto* file which is accessed using *pb* pointer which is pointing to the interface generated by *token_pb.proto* file.