TikTok Tech Immersion 2023 Project

Description

Project description: [TikTok Tech Immersion] Server (Backend) Assignment

Code template: https://github.com/TikTokTechImmersion/assignment_demo_2023

In this assignment, you will design and develop an IM system implementing a set of specific APIs using Golang. You need only develop the backend side of the system, focusing on core message features without the front-end part and the account/authentication part. -> **Build a *scalable* messaging application backend.**

Discussion

- Can leave a comment on this doc (highlight a specific part of the text and the comment option will appear) if you have any questions. I will try to answer them
- Connect with me on Linkedin as well :D https://www.linkedin.com/in/weixingp/

Project setup

Golang

Step-by-step installation guide:

MacOS: https://www.geeksforgeeks.org/how-to-install-golang-on-macos/

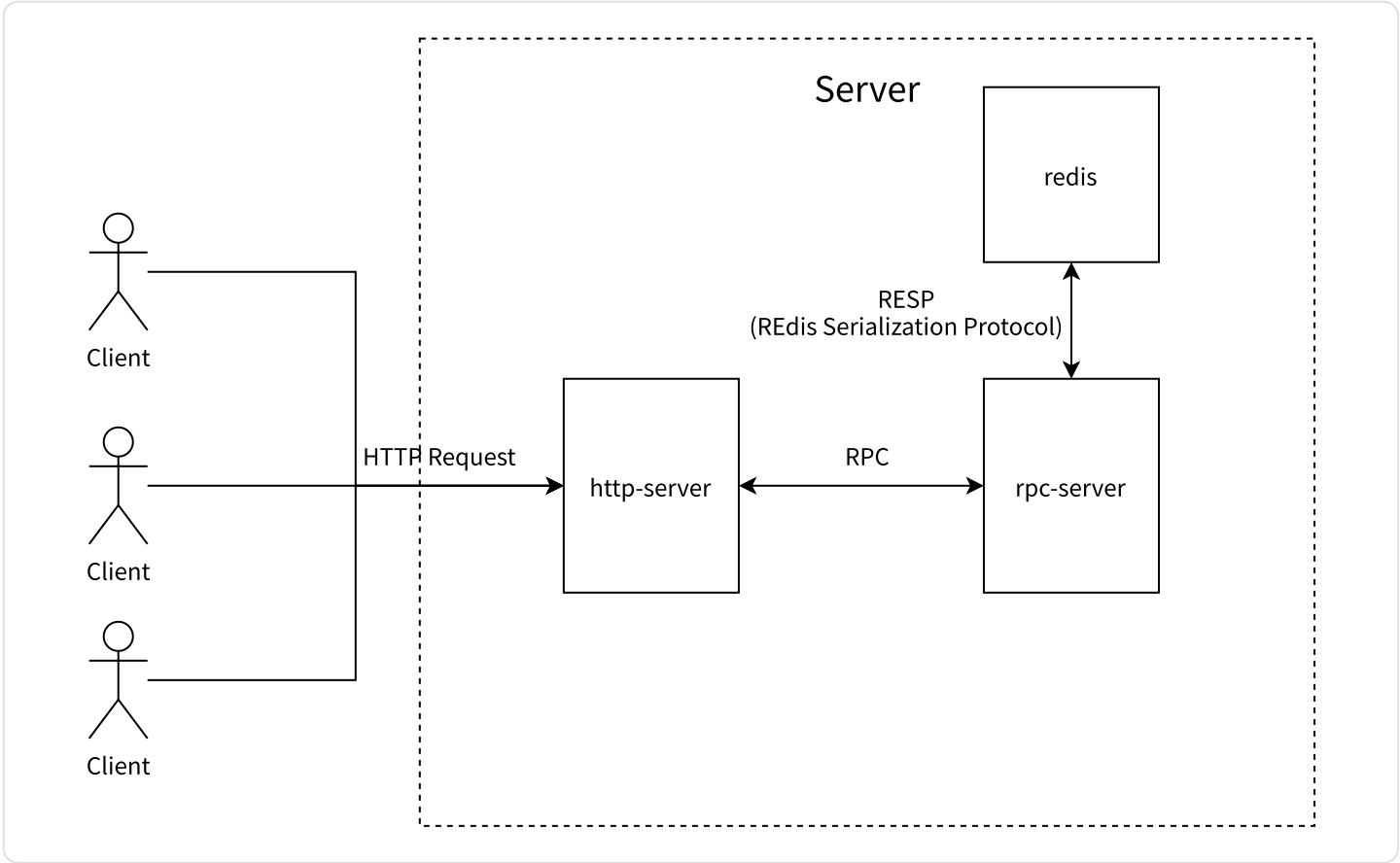
Windows: https://www.geeksforgeeks.org/how-to-install-go-on-windows/

Golang IDE

I recommend using Goland from JetBrains: https://www.jetbrains.com/go/

- Built-in integrations with other tools we are going to use in projects such as Docker
- Use the GitHub student pack for a free license: https://education.github.com/pack

System Architecture Design



Why Redis

- High-performance in-memory store
- Key-value store, easy to get started without caring about the database schema
- Distributed storage, for scaling part later
- Advanced queries are not needed in this project. Redis matches the use cases
- https://aws.amazon.com/redis/

Flow

1. The client (user) will first make an HTTP call to the HTTP server. The main purpose of the HTTP server is just to process HTTP requests and responses. Most of the business logic is done in microservices, which reside in the rpc-server for our design.
2. The HTTP server makes an RPC call to the rpc-server in the local network. (External clients can't access this rpc-server directly).
3. The rpc-server receives and processes the request. This can either be SendRequest() or PullRequest() in our case.
4. SendRequest will write messages to the Redis server while PullRequest will read messages from Redis.
5. This data is passed back to the HTTP server to return to the client as an HTTP response.

Project requirements & tasks

API Specifications

From API definitions in *idl_http.proto*

- User to send an HTTP request to /api/send to send a message

  POST /api/send

- **Request (send in the body):**

Param	Type	Description	Comment
sender	string	Sender name	
receiver	string	Receiver name	
text	string	The text message to send	

- **Request payload sample:**

```
{
    "chat": "a1:a2",
    "text": "hello",
    "sender": "a2"
}
```

Response: Empty

- User to send an HTTP request to /api/pull to pull messages from a room

  GET /api/pull

  Request(send in the body):

> ❗ It's not common to send a data body in a "Get" request, but the http code in the demo is given in this way, and the instruction states that we should not touch the http-server code. (Maybe the assessor has a specific way to test our program)

Param	Type	Description	Comment
chat	string	Chat ID	

		Format "<member1>:<member2>"	
cursor	int	Starting position of message's send_time, inclusively, 0 by default	
limit	int	The maximum number of messages returned per request, 10 by default	
reverse	boolean	If false, the results will be sorted in ascending order by time	

Request payload sample:

```
{
    "chat": "a1:a2",
    "cursor": 0,
    "limit": 2,
    "reverse": true
}
```

Response:

Query Param	Type	Description	Comment
messages	array	List of messages	
has_more	boolean	If true, can use next_cursor to pull the next page of messages	
next_cursor	int	Starting position of next page, inclusively	

Response data sample:

```
{
    "messages": [
        {
            "chat": "a1:a2",
            "text": "good morning",
            "sender": "a2",
            "send_time": 1684770951
        },
```

```
        {
            "chat": "a1:a2",
            "text": "hello",
            "sender": "a1",
            "send_time": 1684770116
        }
    ],
    "has_more": true,
    "next_cursor": 2
}
```

Tasks

Most of the server codes are implemented by the demo template. There are only a few tasks left:

- [ ] Setup and configure a datastore (Redis)
  - [ ] Edit docker-compose.yml to add a Redis server
  - [ ] Setup Redis client in the rpc-server
- [ ] Implement handlers in rpc-server

  areYouLucky() is a placeholder function. We need to replace it with our business logic here, i.e writing and reading messages from Redis

  - [ ] Send

```go
func (s *IMServiceImpl) Send(ctx context.Context, req *rpc.SendRequest) (*rpc.SendResponse, error) {  hyfunc
    resp := rpc.NewSendResponse()
    resp.Code, resp.Msg = areYouLucky()
    return resp, nil
}
```

  - [ ] Pull

```go
func (s *IMServiceImpl) Pull(ctx context.Context, req *rpc.PullRequest) (*rpc.PullResponse, error) {  hyfunc
    resp := rpc.NewPullResponse()
    resp.Code, resp.Msg = areYouLucky()
    return resp, nil
}
```

- [ ] Scaling (optional)

  To be completed

  - [ ] Configure a k8s cluster

Kitex and Hertz

RPC Server using Kitex

- TikTok's Golang RPC framework: https://www.cloudwego.io/docs/kitex/getting-started/
- **kitex_gen** includes codes generated from idl_rpc.thrift, it generates the RPC client for use on the HTTP server and the RPC server code for the rpc-server.
- If you change the definitions to idl_rpc.thrift, for example, adding a field in the Message struct, you need to regenerate it again.

  Example usage for our project:

  a. Install thrift compiler and Kitex

```
go install github.com/cloudwego/thriftgo
go install github.com/cloudwego/kitex/tool/cmd/kitex@latest
```

  b. Generate the code for http-server and rpc-server

```
cd ./rpc-server
kitex -module "github.com/TikTokTechImmersion/assignment_demo_2023/rpc-server" -service imservice ../idl_rpc.thrift
cp -r ./kitex_gen ../http-server # copy kitex_gen to http-server
```

  c. You can save the above commands as a bash script for easy re-generation later on.


```
bash kitex-regen.sh
```

HTTP Server using Hertz

- TikTok's Golang HTTP server framework: https://www.cloudwego.io/docs/hertz/getting-started/
- Protobuf protocol
  - Like JSON and XML, another format for data serialization
  - What and why protobuf: https://www.youtube.com/watch?v=9fh-XdUH7qw

- If you wish to change the API definitions, first update **idl_http.proto** then regenerate the code.
  - It's different from **idl_rpc.thrift**
    - **idl_rpc.thrift**: for generation of RPC client and server code
    - **idl_http.proto**: for generation API request and response definitions, and protobuf definitions for communication to rpc-server. (Looks like Kitex is using Protobuf/Thrift for data serialization)

  Example for this project:

  a. Install Protobuf

```
brew install protobuf #mac os with brew
```

  b. Install the **protoc-gen-go** plugin for generation of idl code

```
go install github.com/golang/protobuf/protoc-gen-go@latest
```

  c. Generate the new API definitions

```
protoc --go_out=./http-server/proto_gen/api --go_opt=paths=source_relative
./idl_http.proto
```

  d. Bash script for easy re-generation

Let's build :)

Spoilers below, try yourself first before referring to this section. There is no right or wrong in the design and code section as well. There are 1001 ways to implement a requirement.

✅ Completed project for reference: https://github.com/weixingp/tiktok-tech-immersion-2023

GoLand IDE is used for this section

Fork the project to your own GitHub and clone it locally.

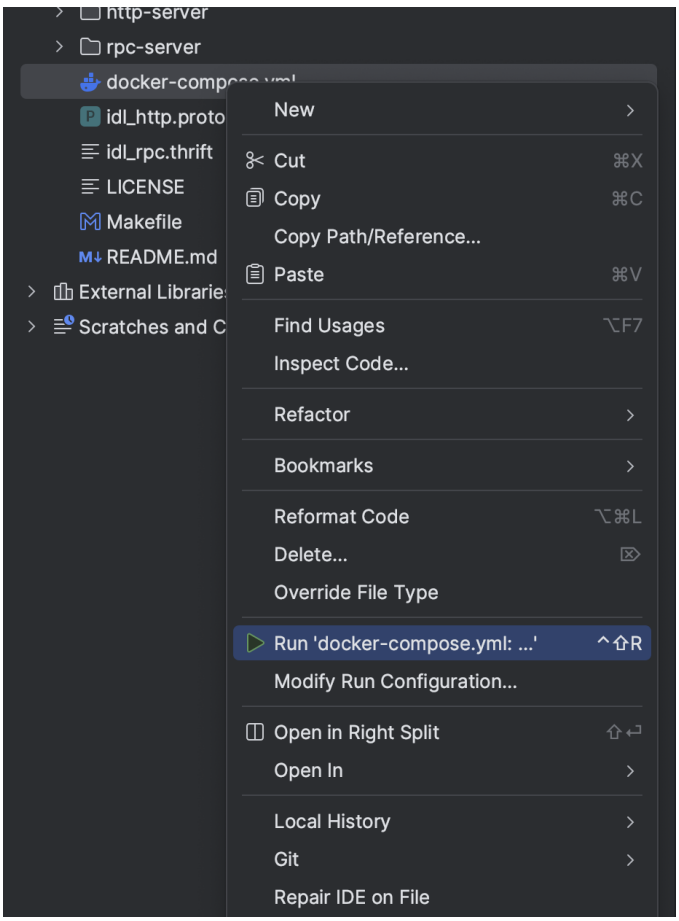
Running demo code with Docker

Make sure you have Docker installed:

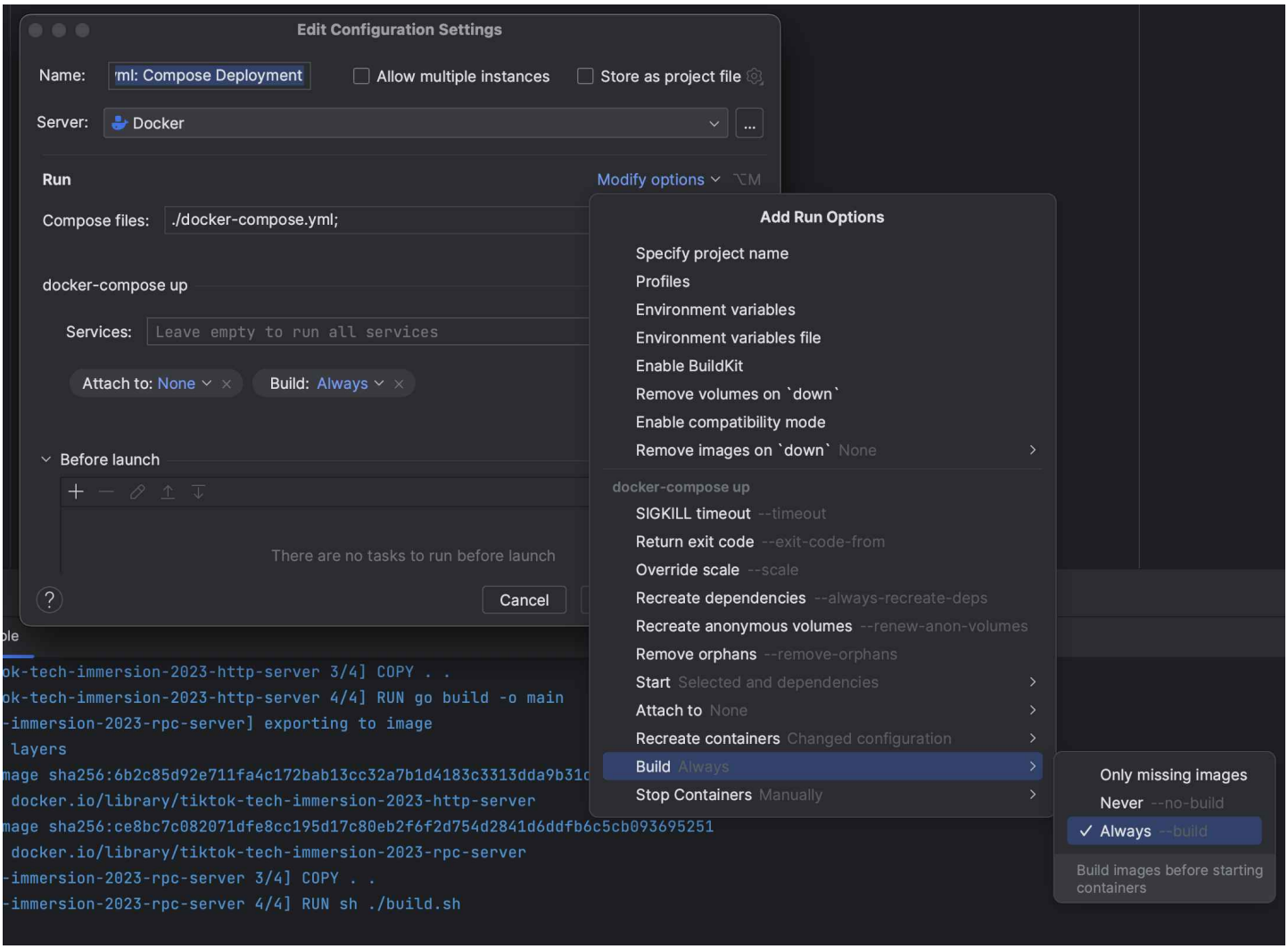
- Windows: https://docs.docker.com/desktop/install/windows-install/
- MacOS: https://docs.docker.com/desktop/install/mac-install/

Let's test if the demo template works nicely:

Right-click docker-compsoe.yml and run

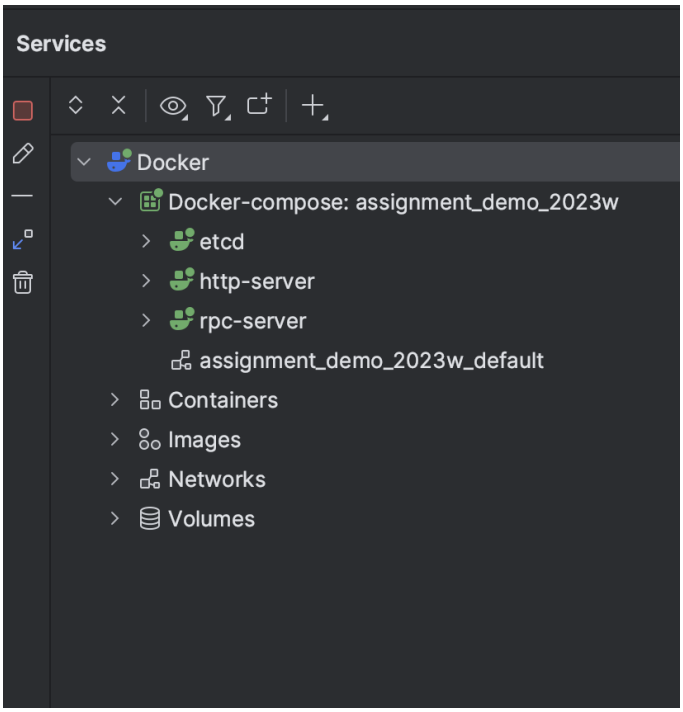


> ❗ Set *build* to **always** when you are developing with Docker, so your latest code is rebuilt every time you run it.



If you encounter the following error, the issue is related to: https://stackoverflow.com/questions/53165471/building-docker-images-on-windows-entrypoint-script-no-such-file-or-directory

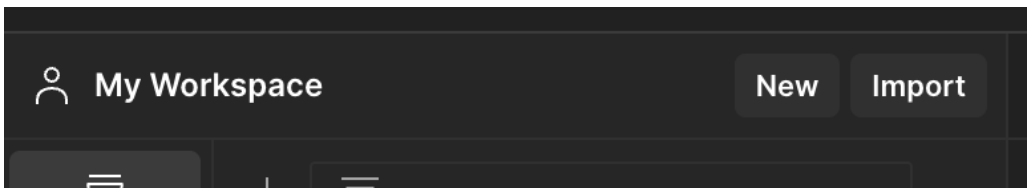
```
 => ERROR [4/4] RUN sh ./build.sh                                    0.5s
------
 > [4/4] RUN sh ./build.sh:
: not foundbuild.sh: 3:
#0 0.521 chmod: cannot access 'output/bootstrap.sh'$'\r': No such file or directory
```



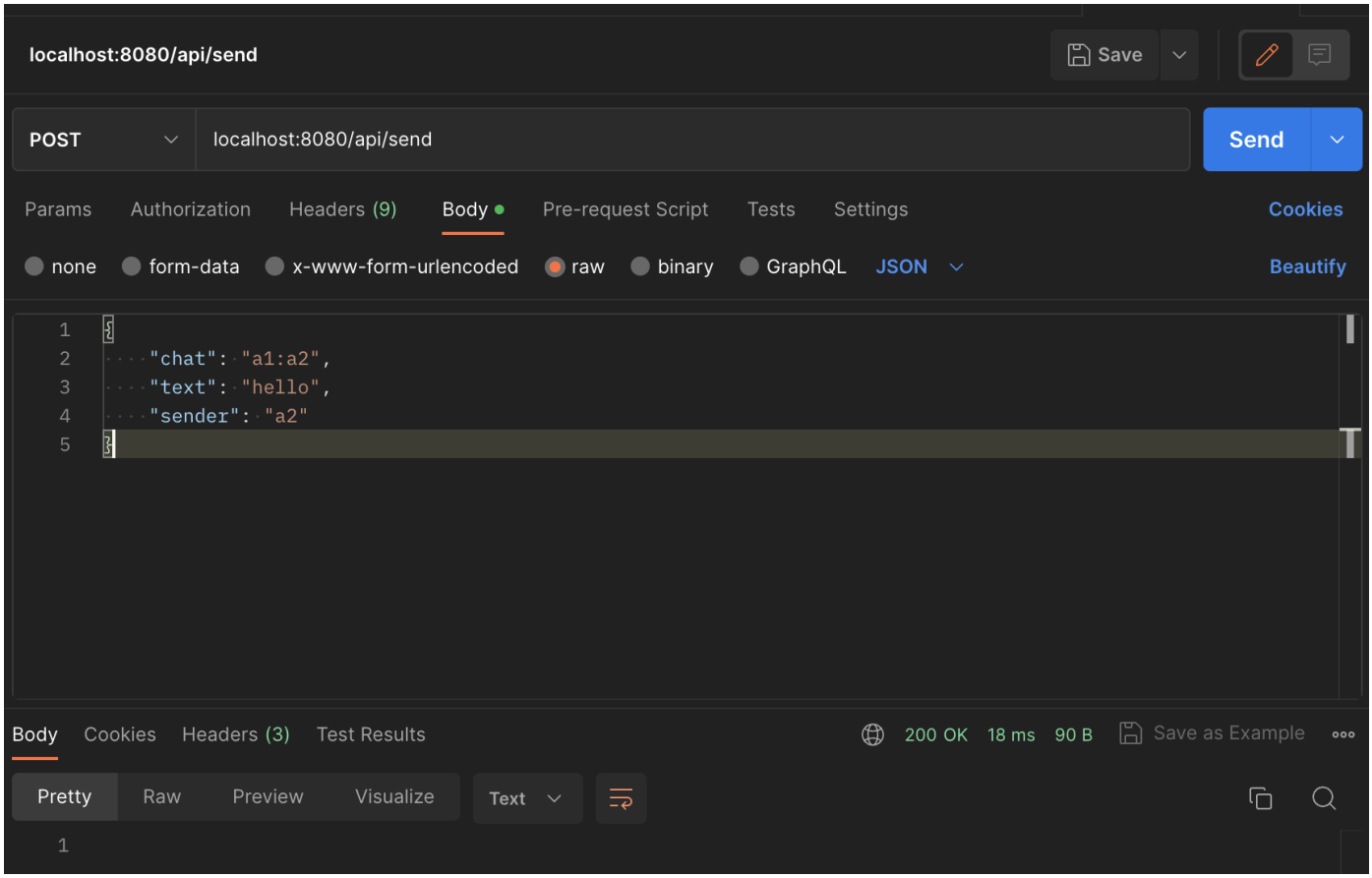
Looking good :) let's test the http-server

I am using **Postman** to test the API

- Download Postman: https://www.postman.com/downloads/
- Import the following request to Postman using the cURL format




```
curl --location 'localhost:8080/api/send' \
--header 'Content-Type: application/json' \
--data '{
    "chat": "a1:a2",
    "text": "hello",
    "sender": "a2"
}'
```



It works! (The demo code logic has a 50/50 chance of getting a 200 Success or 500 internal Server Error)

- I will be using Postman to test as well when I write my own logic

Database Design

- Since the requirement is simple, we just need one "table" to store data.
- Redis is a key-value store. We can get data from Redis given a key.
- We can set room ID as the key, and its value to the list of messages in the room
  - Room ID is just simply userA:userB, eg ***jack:marcus***
    - Messages from Jack to Marcus and Marcus to Jack will be stored in this room
    - Since the keys marcus:jack and jack:marcus are different, we need a function to sort the order of names using alphabetical order.
- We can use the **ZADD** command (https://redis.io/commands/zadd/) to add data
  - It adds data to the sorted set stored at key.
  - We can use the Unix timestamp as the score, so messages will be ordered in chronological order
  - Later on, we can use **ZRANGE** or **ZREVRANGE** command to get messages in order of timestamp

- Visualization of data stored:

```
{
    "jack:marcus": [
        {
            "score": 1684758444,
            "member": {
                "message": "hello!",
                "sender": "jack",
                "timestamp": 1684758444
            }
        },
        {
            "score": 1684758844,
            "member": {
                "message": "nice to meet you!",
                "sender": "marcus",
                "timestamp": 1684758844,
            }
        }
    ]
}
```

Setup Redis

1. Install Redis Go client for rpc-server

```
cd ../rpc-server
go get github.com/redis/go-redis/v9
```

2. Add Redis into docker-compose.yml

```
version: '3.9'
services:
  rpc-server:
    build: rpc-server
    ports:
      - "8888:8888"
    environment:
      - SERVICE_NAME=rpc-server
```

```
      - SERVICE_TAGS=rpc
    depends_on:
      - etcd
      - redis # Wait for Redis server to be ready before running rpc-server
  http-server:
    build: http-server
    ports:
      - "8080:8080"
    environment:
      - SERVICE_NAME=http-server
      - SERVICE_TAGS=http
    depends_on:
      - etcd
      - rpc-server
  etcd:
    image: quay.io/coreos/etcd:v3.5.0
    command: ["etcd", "--advertise-client-urls", "http://etcd:2379", "--listen-c
    ports:
      - "2379:2379"
  redis:
    image: 'bitnami/redis:latest'
    environment:
      - ALLOW_EMPTY_PASSWORD=yes
    ports:
      - "6379:6379"
```

With this config, you can access the Redis server at "redis:6379" inside the rpc-server, as all the services are connected via the "default" docker network for this app.

Create a Redis client in rpc-server

RedisClient

```
import (
    "context"
    "encoding/json"
    "github.com/redis/go-redis/v9"
    "time"
)

type RedisClient struct {
    cli *redis.Client
}
```

```
11
12 func (c *RedisClient) InitClient(ctx context.Context, address, password string)
13     r := redis.NewClient(&redis.Options{
14         Addr:     address,
15         Password: password, // no password set
16         DB:       0,        // use default DB
17     })
18
19     // test connection
20     if err := r.Ping(ctx).Err(); err != nil {
21         return err
22     }
23
24     c.cli = r
25     return nil
26 }
```

- Nothing fancy here, we write the Redis Client init code
  - A custom RedisClient struct is created here, so later we can define methods of this type. (https://gobyexample.com/methods)

Init Redis client in main()

```
1 var (
2     rdb = &RedisClient{} // make the RedisClient with global visibility in the
  'main' scope
3 )
4
5 func main() {
6     ctx := context.Background() //
  https://www.digitalocean.com/community/tutorials/how-to-use-contexts-in-go
7
8     err := rdb.InitClient(ctx, "redis:6379", "")
9     if err != nil {
10        errMsg := fmt.Sprintf("failed to init Redis client, err: %v", err)
11        log.Fatal(errMsg)
12    }
13
14    r, err := etcd.NewEtcdRegistry([]string{"etcd:2379"}) // r should not be
  reused.
15    if err != nil {
16        log.Fatal(err)
17    }
18
```

```
19      svr := rpc.NewServer(new(IMServiceImpl), server.WithRegistry(r),
    server.WithServerBasicInfo(&rpcinfo.EndpointBasicInfo{
20          ServiceName: "demo.rpc.server",
21      }))
22
23      err = svr.Run()
24      if err != nil {
25          log.Println(err.Error())
26      }
27  }
```

- Using our RedisClient init code earlier on, we connect to the Redis server on "redis:6379" and no password. You can't use "localhost:6379" here because it's inside the docker network. Use the service name defined in docker-compose as the hostname. (ref: https://docs.docker.com/compose/networking/)

Save messages to Redis

- Using the database design from earlier on, we can create a method **SaveMessage** for *RedisClient*
  - In Go, if you want to make your methods/functions/fields public, simply cap the first letter.
- We need to create another struct **Message** for JSON serialization

```
1  type Message struct {
2      Sender    string  `json:"sender"`
3      Message   string  `json:"message"`
4      Timestamp int64   `json:"timestamp"`
5  }
```

```
1  func (c *RedisClient) SaveMessage(ctx context.Context, roomID string, message
    *Message) error {
2      // Marshal the Go struct into JSON bytes
3      text, err := json.Marshal(message)
4      if err != nil {
5          return err
6      }
7
8      member := &redis.Z{
9          Score:  message.Timestamp, // The sort key
10         Member: text, // Data
11     }
12
```

```
    _, err = c.cli.ZAdd(ctx, roomID, *member).Result()
    if err != nil {
        return err
    }

    return nil
}
```

- We will save the message in JSON format (as opposed to the protobuf format from the RPC server)

Get messages from Redis

```
func (c *RedisClient) GetMessagesByRoomID(ctx context.Context, roomID string, start, end int64, reverse bool) ([]*Message, error) {
    var (
        rawMessages []string
        messages    []*Message
        err         error
    )

    if reverse {
        // Desc order with time -> first message is the latest message
        rawMessages, err = c.cli.ZRevRange(ctx, roomID, start, end).Result()
        if err != nil {
            return nil, err
        }
    } else {
        // Asc order with time -> first message is the earliest message
        rawMessages, err = c.cli.ZRange(ctx, roomID, start, end).Result()
        if err != nil {
            return nil, err
        }
    }

    for _, msg := range rawMessages {
        temp := &Message{}
        err := json.Unmarshal([]byte(msg), temp)
        if err != nil {
            return nil, err
        }
        messages = append(messages, temp)
    }

    return messages, nil
```

```
32 }
```

- If the order is reversed, we need to use the ZRevRange command instead of ZRANGE
- We need to deserialize the raw content into the Go *Message* struct for use later

Most of the hard work is done here :D, we just need to call these Redis functions in handler

Implement Send()

As mentioned in the database design, the key(chat ID) "a1:a2" and "a2:a1" are different in Redis. We need to write a function to standardize this.

- Using strings.Compare to sort the sender in asc order. This way we can make sure the room ID is identical.

```go
func getRoomID(chat string) (string, error) {
    var roomID string

    lowercase := strings.ToLower(chat)
    senders := strings.Split(lowercase, ":")
    if len(senders) != 2 {
       err := fmt.Errorf("invalid Chat ID '%s', should be in the format of user1:user2", chat)
       return "", err
    }

    sender1, sender2 := senders[0], senders[1]
    // Compare the sender and receiver alphabetically, and sort them asc to form the room ID
    if comp := strings.Compare(sender1, sender2); comp == 1 {
       roomID = fmt.Sprintf("%s:%s", sender2, sender1)
    } else {
       roomID = fmt.Sprintf("%s:%s", sender1, sender2)
    }

    return roomID, nil
}
```

It's also cool to validate the request first, in case a user sends unexpected data. For example, an invalid sender for a room.

```go
func validateSendRequest(req *rpc.SendRequest) error {
    senders := strings.Split(req.Message.Chat, ":")
    if len(senders) != 2 {
       err := fmt.Errorf("invalid Chat ID '%s', should be in the format of user1
       return err
    }
    sender1, sender2 := senders[0], senders[1]

    if req.Message.GetSender() != sender1 && req.Message.GetSender() != sender2
       err := fmt.Errorf("sender '%s' not in the chat room", req.Message.GetSend
       return err
    }

    return nil
}
```

Finally, we put all the code together

- Convert the request data into the *Message* type
- Save the message to Redis
- Resp.code is not HTTP code here, zero means success and non-zero means error.

```go
func (s *IMServiceImpl) Send(ctx context.Context, req *rpc.SendRequest)
(*rpc.SendResponse, error) {
    if err := validateSendRequest(req); err != nil {
       return nil, err
    }

    timestamp := time.Now().Unix()
    message := &Message{
       Message:   req.Message.GetText(),
       Sender:    req.Message.GetSender(),
       Timestamp: timestamp,
    }

    roomID := getRoomID(req.Message.GetChat())

    err := rdb.SaveMessage(ctx, roomID, message)
    if err != nil {
       return nil, err
    }

    resp := rpc.NewSendResponse()
    resp.Code, resp.Msg = 0, "success"
    return resp, nil
```

```
}
```

Implement Pull()

- When calculating the end position, we did not minus 1 on purpose for hasMore check
  - If we have 1 extra record in the returned result from Redis, it means there is at least 1 more record to be fetched, so we set hasMore to true, the nextCursor to the end position
- The whole chunk of for loop to convert Message struct to RPC struct.

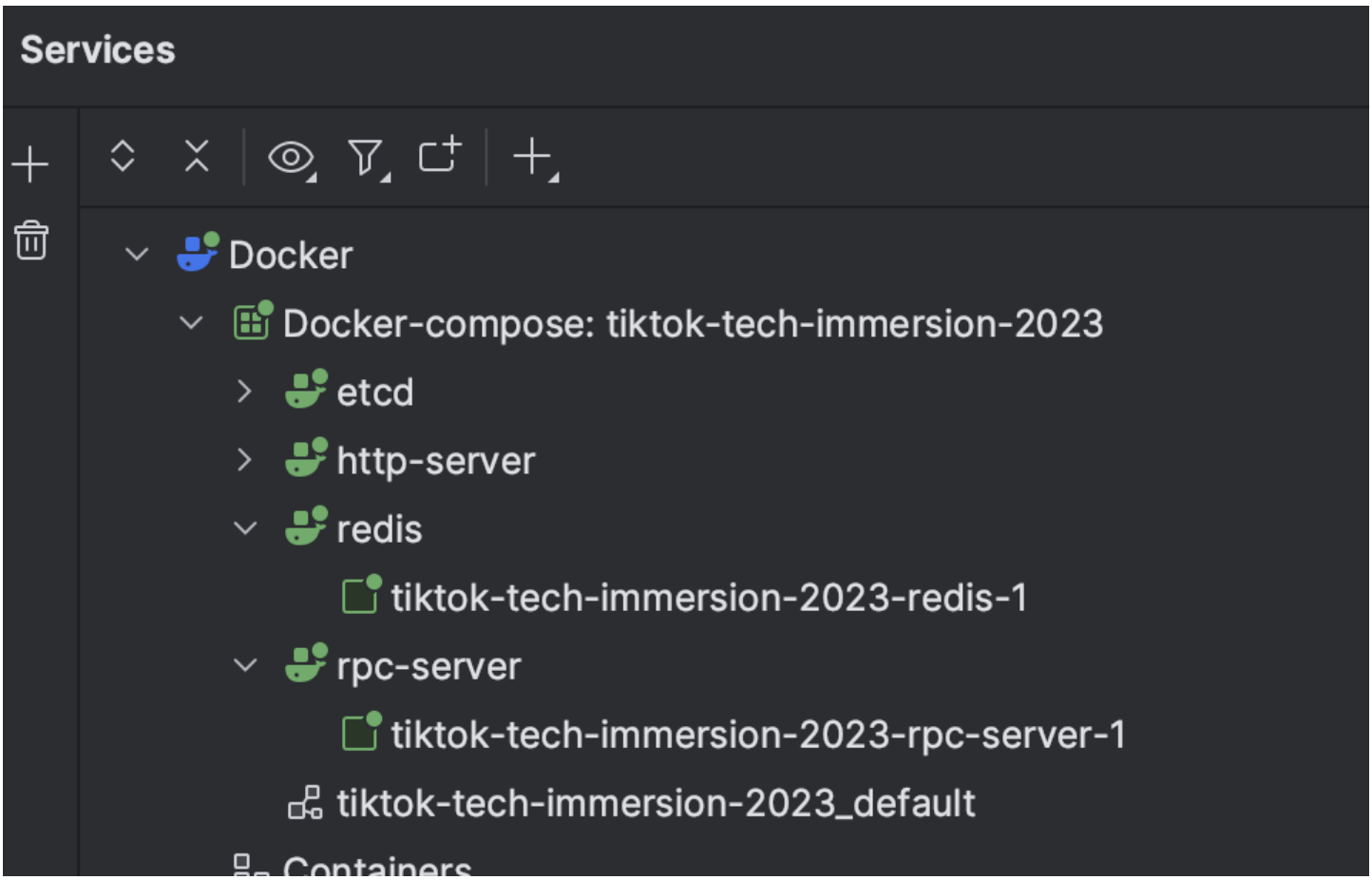
```
func (s *IMServiceImpl) Pull(ctx context.Context, req *rpc.PullRequest) (*rpc.PullResponse, error) {
    roomID, err := getRoomID(req.GetChat())
    if err != nil {
        return nil, err
    }

    start := req.GetCursor()
    end := start + int64(req.GetLimit()) // did not minus 1 on purpose for hasMore check later on

    messages, err := rdb.GetMessagesByRoomID(ctx, roomID, start, end, req.GetReverse())
    if err != nil {
        return nil, err
    }

    respMessages := make([]*rpc.Message, 0)
    var counter int32 = 0
    var nextCursor int64 = 0
    hasMore := false
    for _, msg := range messages {
        if counter+1 > req.GetLimit() {
            // having extra value here means it has more data
            hasMore = true
            nextCursor = end
            break // do not return the last message
        }
        temp := &rpc.Message{
            Chat:     req.GetChat(),
            Text:     msg.Message,
            Sender:   msg.Sender,
            SendTime: msg.Timestamp,
```

```
31          }
32          respMessages = append(respMessages, temp)
33          counter += 1
34      }
35
36      resp := rpc.NewPullResponse()
37      resp.Messages = respMessages
38      resp.Code = 0
39      resp.Msg = "success"
40      resp.HasMore = &hasMore
41      resp.NextCursor = &nextCursor
42
43      return resp, nil
44  }
```

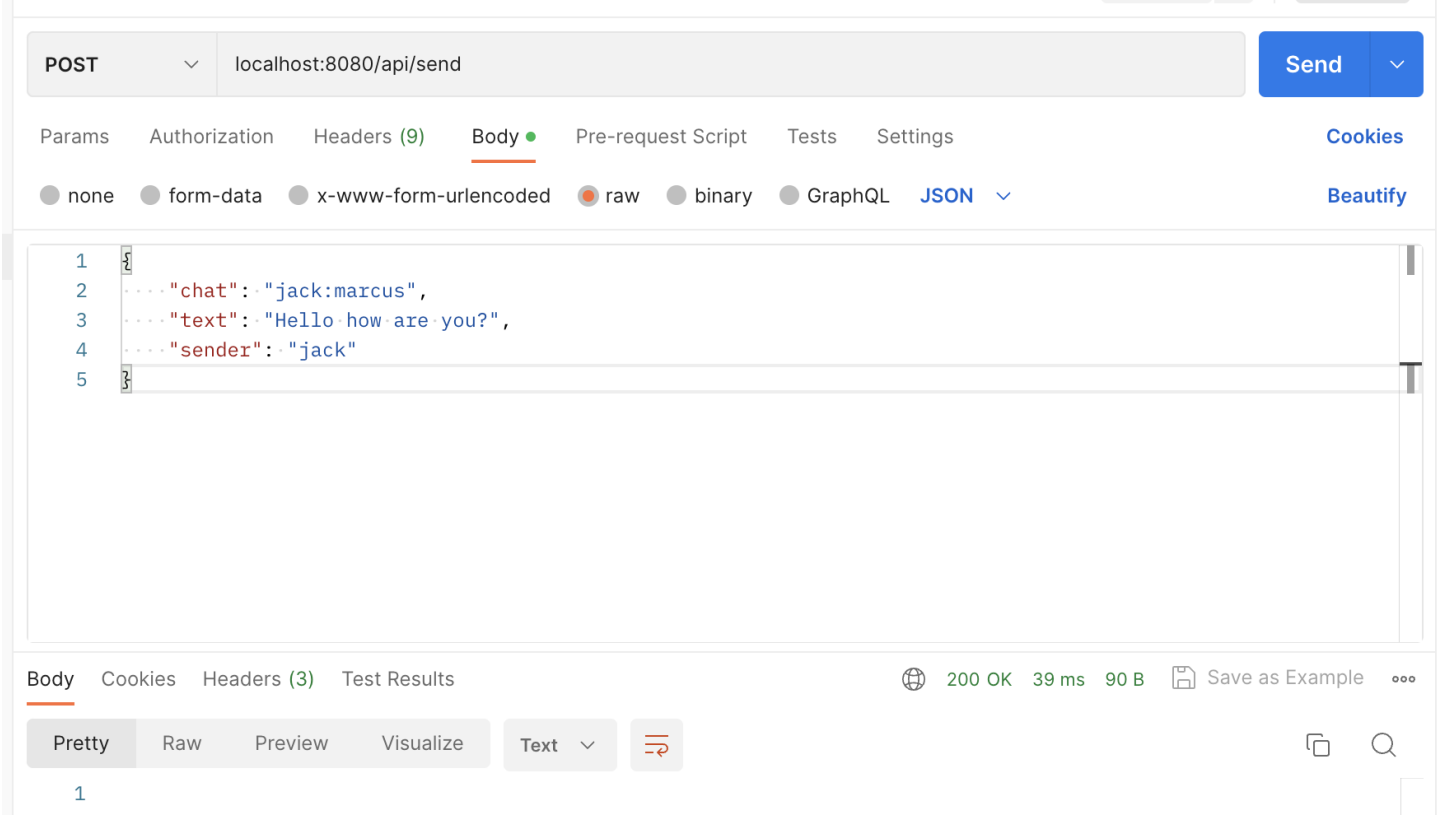
Until here, we are done with the basic project requirements.

IT WORKS

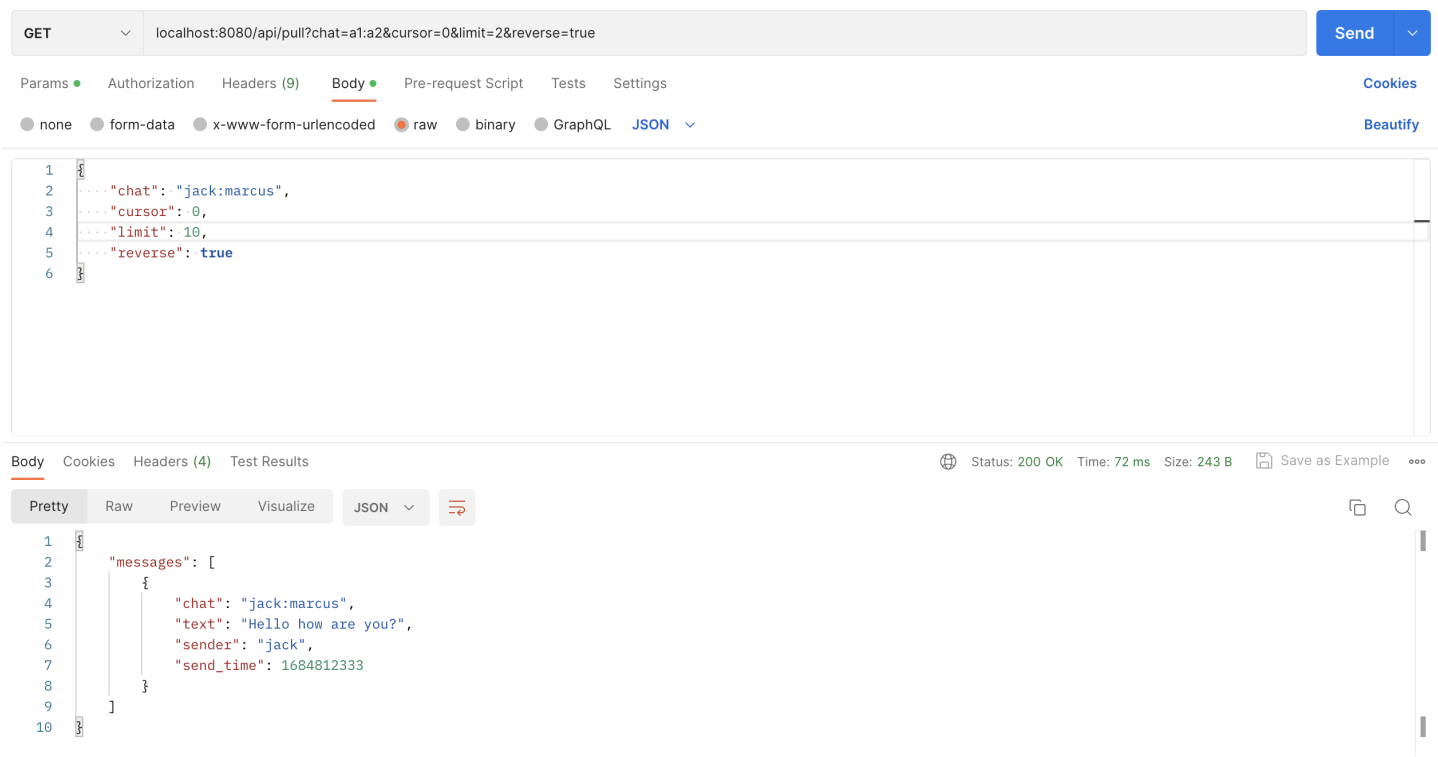
Notice that we now have a Redis service running alongside other services from demo code



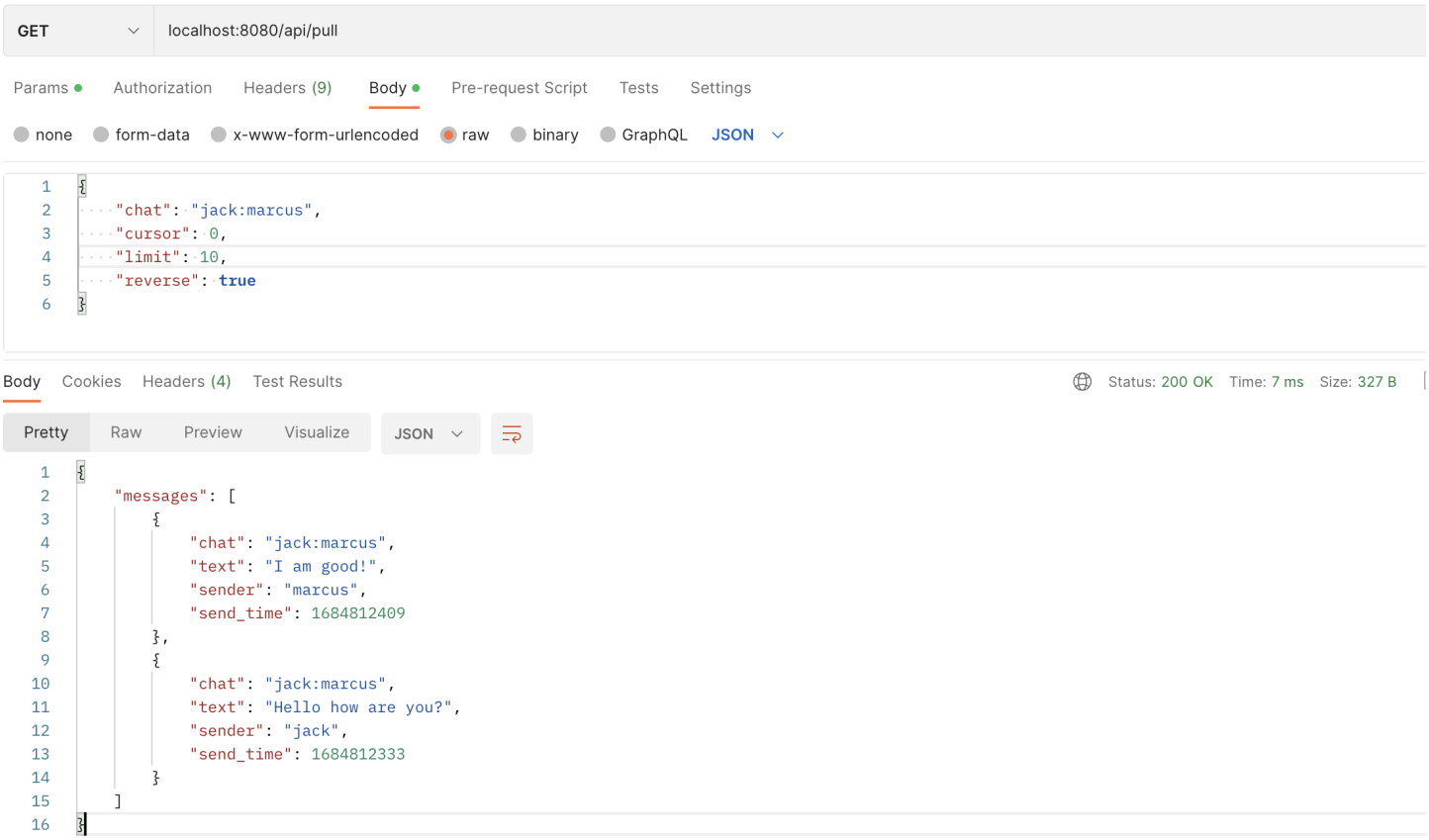
Sending a message from Jack to Marcus



And pulling messages from this room



And multiple messages in a room



Testing

Unit test

Since our program depends on Redis, we can't do a single unit test like the one given in demo, it will fail as it can't connect to the Redis server. So comment the test cases for now in handler_test.go

```
for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        //s := &IMServiceImpl{}
        //got, err := s.Send(tt.args.ctx, tt.args.req)
        assert.True(t, true)
    })
}
```

Concurrency and load test

Go is an efficient language. The 20QPS concurrency test can be passed without issue out of the box without code optimization.

Install and open JMeter

```
# For MacOS only, Windows users pls buy a Mac :)
# Jk, JMeter is available in Windows too
brew install jmeter
```

```
jmeter # open jmeter
```

Test setup

~~(yes I am an Apple fanboy)~~

JMeter test config file I am using:

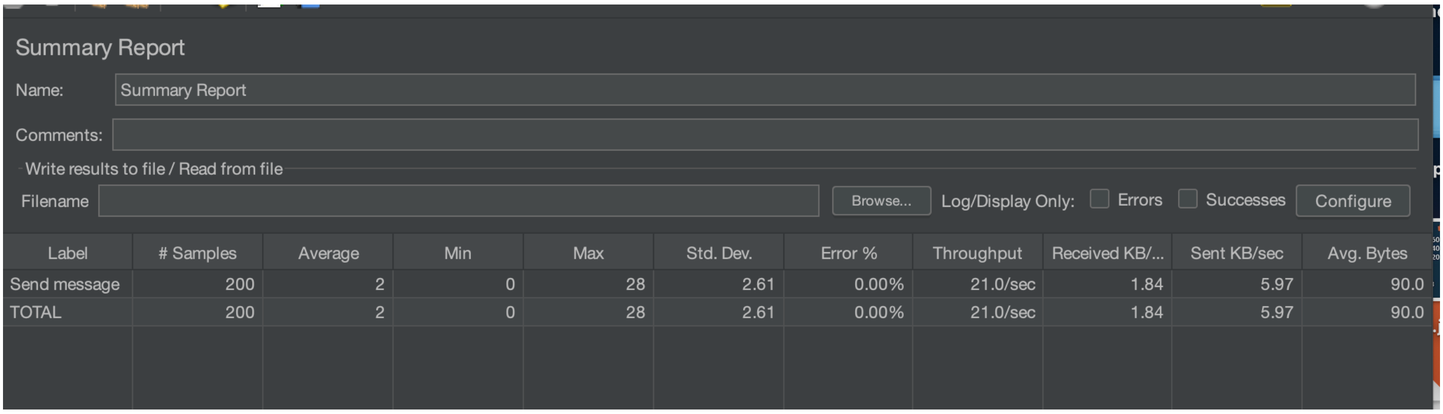
HTTP Request.jmx
7.32KB

- You can change the QPS in the thread group

Test results

Not breaking a sweat when running at 20qps

- Max 28ms response time (since I am testing it locally, no network overheads)



Summary Report

Name: Summary Report

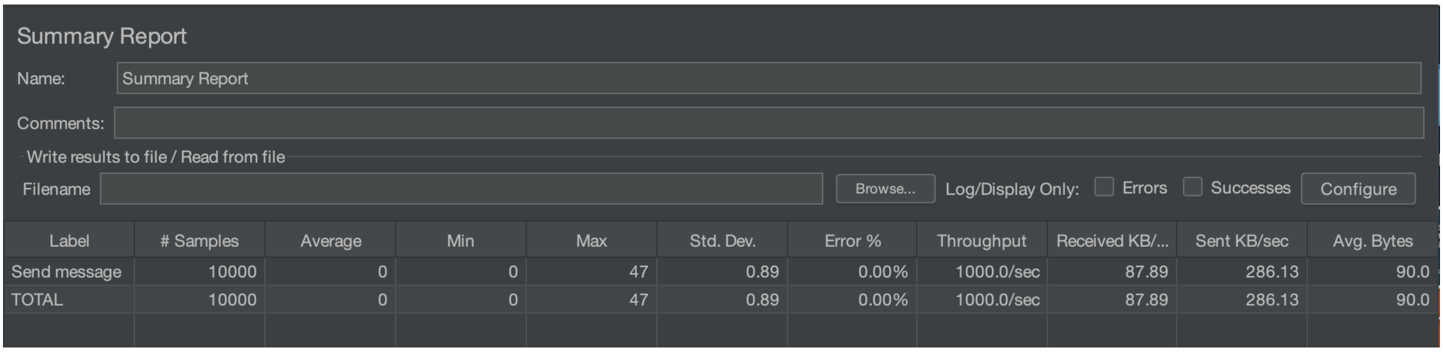
Comments:

Write results to file / Read from file

Filename | Browse... | Log/Display Only: | Errors | Successes | Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Send message	200	2	0	28	2.61	0.00%	21.0/sec	1.84	5.97	90.0
TOTAL	200	2	0	28	2.61	0.00%	21.0/sec	1.84	5.97	90.0

- Hmm, same performance for 1000qps



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename | Browse... | Log/Display Only: | Errors | Successes | Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Send message	10000	0	0	47	0.89	0.00%	1000.0/sec	87.89	286.13	90.0
TOTAL	10000	0	0	47	0.89	0.00%	1000.0/sec	87.89	286.13	90.0

- Starts to get some IO timeout > 5000QPS, not sure why

Summary Report

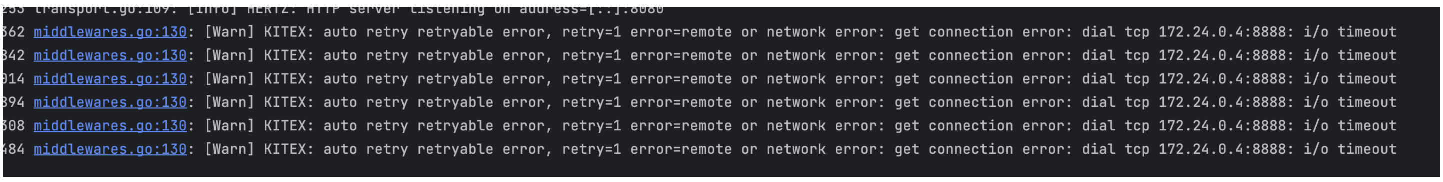
Name: Summary Report

Comments:

Write results to file / Read from file

Filename | Browse... | Log/Display Only: | Errors | Successes | Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Send message	45610	8	0	152	15.28	0.00%	5016.5/sec	440.90	1435.38	90.0
TOTAL	45610	8	0	152	15.28	0.00%	5016.5/sec	440.90	1435.38	90.0



```
362 middlewares.go:130: [Warn] KITEX: auto retry retryable error, retry=1 error=remote or network error: get connection error: dial tcp 172.24.0.4:8888: i/o timeout
342 middlewares.go:130: [Warn] KITEX: auto retry retryable error, retry=1 error=remote or network error: get connection error: dial tcp 172.24.0.4:8888: i/o timeout
014 middlewares.go:130: [Warn] KITEX: auto retry retryable error, retry=1 error=remote or network error: get connection error: dial tcp 172.24.0.4:8888: i/o timeout
394 middlewares.go:130: [Warn] KITEX: auto retry retryable error, retry=1 error=remote or network error: get connection error: dial tcp 172.24.0.4:8888: i/o timeout
308 middlewares.go:130: [Warn] KITEX: auto retry retryable error, retry=1 error=remote or network error: get connection error: dial tcp 172.24.0.4:8888: i/o timeout
484 middlewares.go:130: [Warn] KITEX: auto retry retryable error, retry=1 error=remote or network error: get connection error: dial tcp 172.24.0.4:8888: i/o timeout
```

Maybe if > 1000 QPS we can do some scaling

Scaling (Bonus)

KIV

Looking at it when I have time, should be using a Kubernetes cluster

Discussion

- Can leave a comment on this doc (highlight a specific part of the text and the comment option will appear) if you have any questions. I will try to answer them
- Connect with me on Linkedin as well :D https://www.linkedin.com/in/weixingp/