נושא הפרויקט:
מחברות משותפות

שם התלמיד: גל בן – שך

מספר תעודת זהות: 213261118

שם המנחה: יורם אביטוב

תאריך הגשה: 20.4.2022

Contents

1 ניהול שינויים

תיאור מלא ב-https://github.com/Benshcha/Cyber-Project-2022

פעילות	גרסה	תכולה / שינוי	תאריך סיום
יזום	0.0.0.1	הצעה ראשונית וארכיטקטורה	20.11
פיתוח	0.0.1	קנווס ראשוני	27.12
פיתוח	0.0.2	אימפלמנטציה של רישום וכניסת משתמשים	9.1
פיתוח	0.0.3	שמירת מחברות	27.1
פיתוח	0.0.4	עדכון מערכת ציור ומעבר לספרייה perfect handwriting	7.2
פיתוח	0.0.5	מעבר ל-HTTPS	12.2
פיתוח	0.0.6	הוספת פיצ'רים של ציור: צבע ועובי	13.2
פיתוח	0.0.7	עדכון מערכת העדכונים ומעבר למערכת המאפשרת תקשורת של מחברות גדולות	7.3
יזום	0.1	הוספת אתר דוקומנטציה וREADME	19.4

2 מבוא

בעזרת השרת משתמשים יוכלו לכתוב מחברות בכתב יד אשר ישמרו באופן בטוח על השרת ויוכלו לצפות בהם בכל רגע. בנוסף, יוכלו המשתמשים לשתף מחברות אלו עם חבריהם ולערוך אותם בו זמנית.

רוב תוכנות הכתיבה בכתב יד הינן איטיות ואינן פשוטות לניהול, על כן, ברצוני להקים אתר אשר יאפשר תפעול קל ונגיש של המחברות מבלי זמני טעינה ארוכים או הבלאגן של סידור העריכה.

נושא המחקר בפרויקט

ניהול בסיסי של גרפיקת ווקטורים. על מנת לשמור ולכתוב את המחברות יש לנהל את המידע בעזרת גרפיקת ווקטורים אשר תאפשר שמירה ואיפיון יעיל של כתיבת המשתמש.

גרפיקת ווקטורים ועקומות בייזיר משמשות כמעט תמיד בעיצוב וגרפיקת מחשב.

3 סביבת העבודה בפרויקט

3.1 טכנולוגיות בשימוש בפרויקט

- השרת רץ בעזרת פייתון בגרסה 3.10
- ספריות הפייתון הנמצאות בשימוש:
  - mysql.connector[1]
- תקשורת השרת והקליינטים תתבצע בעזרת socket ושרת מרובה משתמשים.
  והאינטראקציה של הקליינט תוצג בעזרת html ו-javascript ויוצג ב-browser. מידע המשתמש ישמר על השרת באופן מוצפן בעזרת mysql.
  - כדי לקבל את חוזק הלחיצה של משתמש אשר משתמש בעט, אשתמש בתוספת https://pressurejs.com/.
  - בכדי לנהל את עריכת ה-svg, אשתמש בספרייה svg.js
  - בכדי להקל על עריכת ה-html אשתמש בספרייה jquery

[1] ניתן להתקין ספרייה זו בעזרת השורה pip install mysql-connector-python

3.2 מדריך למשתמש

- שרת:
  - על מנת להריץ את השרת יש להתקין פייתון 3.10 לפחות ולהתקין את הספריות הנדרשות של פייתון.
  - בנוסף יש ליצור קובץ בפורמט json בשם "dbconfig.json" בתיקייה הראשית של הפרויקט אשר בה נמצא המידע של ה-database בו ישמר המידע. לדוגמא:

```
{
    "host": "localhost",
    "username": "Benshcha",
    "password": "Super secure and secret password",
    "database": "CyberProject2022",
    "pool_name": "updateNotebooks",
    "autocommit": "True"
}
```

כאשר על המשתנים "pool_name" ו-"autocommit" להיות בעלי בדיוק אותם ערכים והמשתנים האחרים הינם של ה-database והמשתמש אשר יצר אותו (במידה ויש משתמש כזה).

  - לבסוף יש להריץ את הקובץ main.py.
  - על מנת לצאת בבטחה מהשרת יש לכתוב ב-console את הפקודה exit אשר תסגור את השרת ותשמור את המשתמשים והמחברות שלהם.

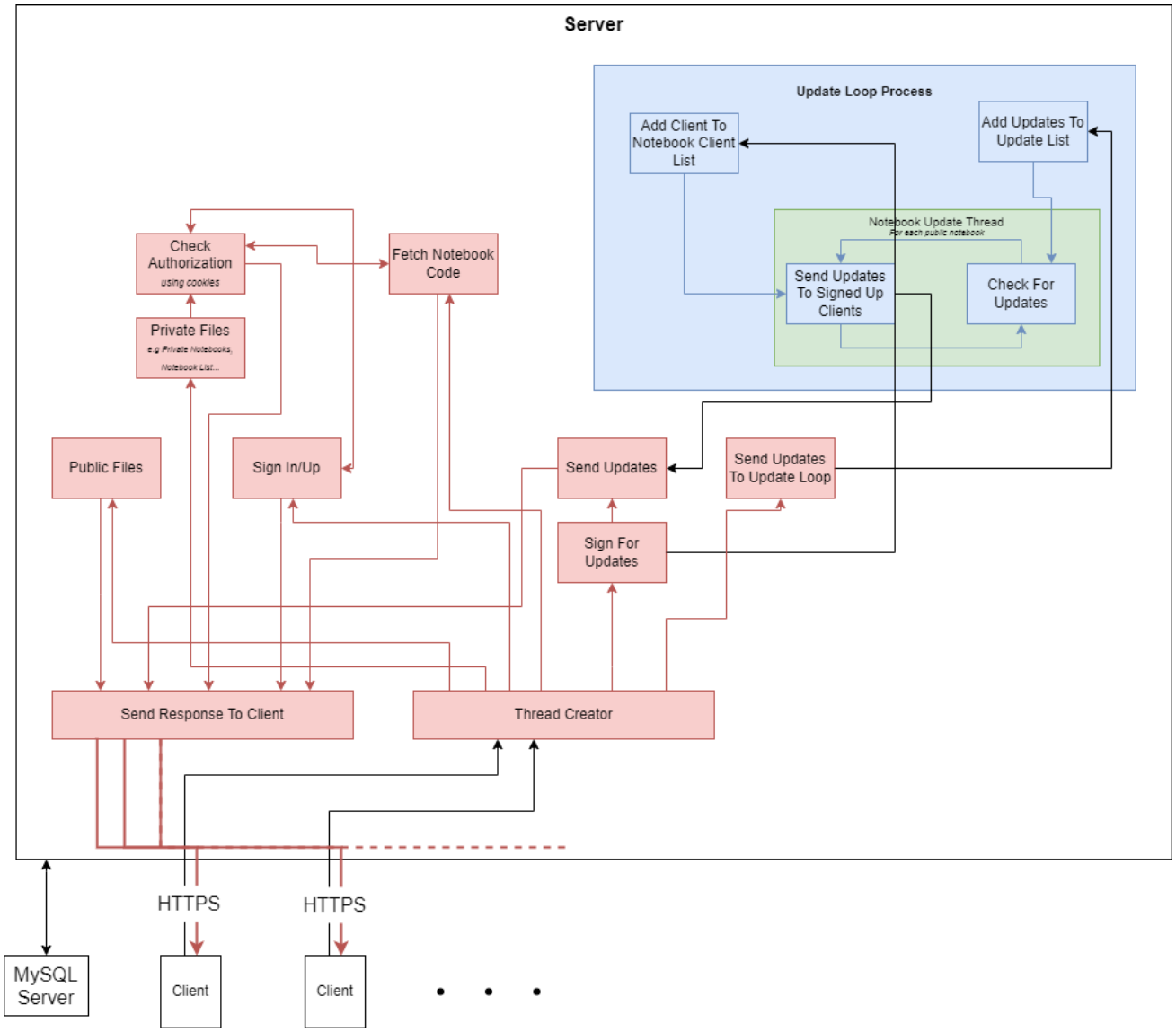
- משתמש:
  - כל שיש על המשתמש לעשות הוא להתחבר דרך המרשתת בדפדפן לכתובת השרת ולהיכנס ליוזר שלו (במידה ואין, יכול ליצור).
  - לאחר שנכנס, יכול ליצור מחברת חדשה, לכתוב בה ולשמור אותה.
  - לאחר ששמר, יכול ללחוץ על כפתור ה-share על מנת לפתוח את חלון השיתוף אשר יאפשר למשתמש ליצור קוד שיתוף.
  - את קוד השיתוף יכול המשתמש לשלוח לחבריו אשר בהיכנסתם יוכלו לערוך את המחברת בזמן אמת.

4 אפיון דרישות וארכיטקטורת המערכת

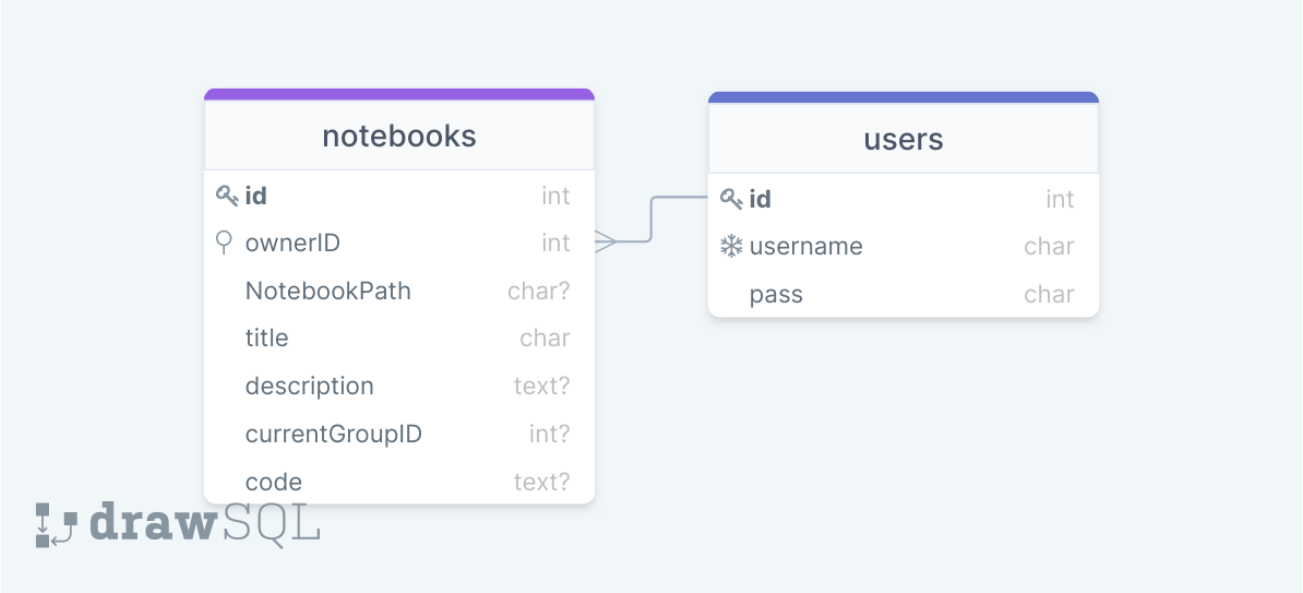
4.1 דרישות ושימושי מערכת – Use Cases

- אין דרישות למערכת מעבר לתמיכה בדפדפן.

4.2 ארכיטקטורת המערכת

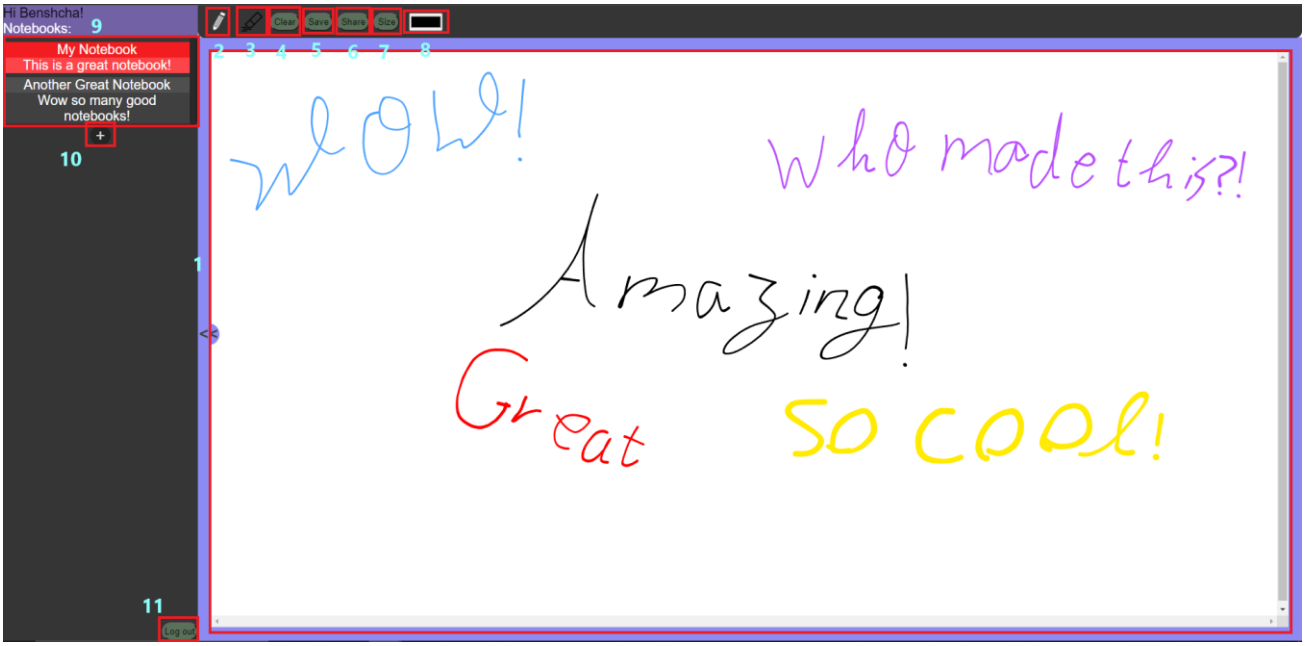


כאשר כל פנייה אל יוצר ה-Threads הינה כחלק משליחת Packet בפרוטוקול HTTPS בעזרת TLS.
בנוסף מבנה שרת ה-mySQL נראה כך:



4.3 ממשק משתמש - GUI

- השרת יוצג למשתמש בעזרת ה-web



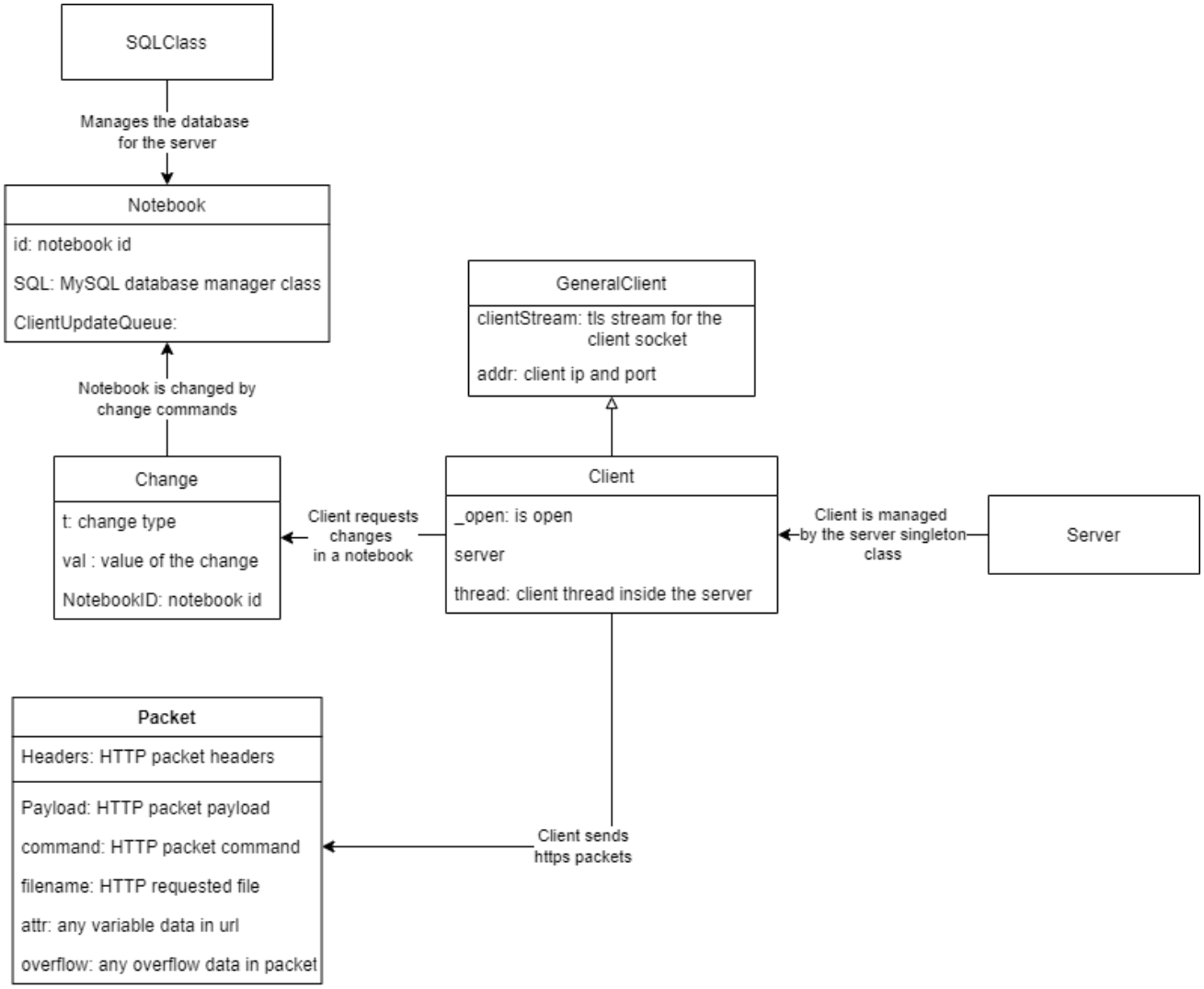
1. הלוח עליו המשתמש יצייר
2. כפתור המאפשר לבחור את כלי העט
3. כפתור המאפשר לבחור את כלי המחק
4. כפתור המאפשר לנקות את הלוח
5. כפתור המאפשר לשמור את המחברת
6. כפתור המאפשר לשתף את המחברת בעזרת לינק
7. כפתור המאפשר לשנות את גודל העט

8. כפתור הפותח את גלגל הצבע ומאפשר לשנות את צבע העט
9. רשימת המחברות של המשתמש
10. כפתור יצירת מחברת חדשה
11. כפתור כניסה או יציאה מהמשתמש

5 מדריך למפתח

ל-documentation בו מתוארות כל המתודות והמחלקות ניתן להיכנס בעזרת הלינק:
https://benshcha.github.io/Cyber-Project-2022/.

5.1 דיאגרמת מחלקות



6 סיכום אישי ורפלקציה

- העובדה על הפרויקט עבורי היתה בעיקר מהנה, למדתי התנהלות נכונה יותר בסביבה סובבת אובייקטים בפייתון אך זאת למעט העבודה וההתנהלות עם js והדפדפן. מצאתי את השפה לא אינטואיטיבית ואת הצורך באלפי ספריות שונות על מנת ליצור שרת נורמלי מיותר. יחד עם זאת, חוסר מבנה טבעי של תהליכונים ותהליכים והסידור האוטומטי של ההתנהלות איתם על ידי הדפדפן בלי יותר מידי הערות על כך מעיקים.
- מהפרויקט למדתי לעבוד עם js, html ו-css וכמובן הספריות הנכללות בהן. בנוסף למדתי להקים שרת HTTPS ואיך להשתמש ב-mysql ב-python.
- לו הייתי מתחיל את הפרוייקט היום הייתי מחלק את הקוד באופן ברור יותר. הייתי משתמש ביותר מחלקות במקום במערכים והייתי בוחר להשתמש בספרייה מוכנה להכין שרתי HTTPS במקום לכתוב את הכל מחדש עם ספריית הsocket. בנוסף הייתי רושם את האתר בעזרת ספרייה כמו React לעומת הjs הרגיל אשר בו השתמשתי.

```python
"""
Main file for Cyber-Project-2022 Gal Ben-Shach

HTTP is refering to the modules.py file.
"""

port = 443

# import global variables
from config import *

import socket, threading, ssl
from pprint import pprint, pformat
import modules as HTTP
from modules import colorText
from os.path import join
import os, sys
import json
import traceback, hashlib, time
import multiprocessing as mp
import xml.etree.ElementTree as ET
ET.register_namespace('', "http://www.w3.org/2000/svg")
from SQLModule import *

class InvalidLoginAttempt(Exception):
    """ # ! Do not insert sensitive information in the exception message
    """
    def __init__(self, msg: str):
        super().__init__(f"Invalid Login Attempt:\n{msg}")


class Client(HTTP.GeneralClient):
    """Main Client Class
    """
    def __init__(self, *args, server):
        super().__init__(*args)
        self._open = True
        self.thread = threading.Thread(target=self.manage)
        self.server = server
        self.UpdateCode = None

    @staticmethod
    def getUserAuth(packet):
        """Get authorization data from packet cookie"""
        if 'Cookie' in packet.Headers and 'user_auth' in packet.Headers['Cookie']:
            cookiesStr = [i.split("=") for i in packet.Headers['Cookie'].split(";")]
            cookies = {cookieStr[0]: cookieStr[1] for cookieStr in cookiesStr}
            username, password = tuple(json.loads(cookies['user_auth']).values())
        else:
            return None, None

        if "'" not in username and "'" not in password:
            return username, password
        else:
            raise InvalidLoginAttempt("Username or password contains invalid characters: '")

    def postResponse(self, packet: HTTP.Packet):
        """Manage response to post request

        Args:
            packet (HTTP.Packet): POST Request packet

        """
        file = packet.filename
        resp = None
        if file == "/SIGNUP":
            resp = self.SignUp(packet.Payload)
        elif file.startswith('/SAVENEWNB'):
            resp = self.NewNotebook(packet)
        elif file.startswith('/SAVE/'):
            resp = self.SaveNotebook(packet, file)
        elif file.startswith('/api/'):
            resp = self.APIPostResponse(packet)
        else:
            # TODO Add error response
            ...

        resp = json.dumps(resp)
        respPacket = HTTP.Packet()
```

```
        respPacket.Headers['Content-Type'] = "text/json"
        respPacket.setPayload(resp)
        self.SendPacket(respPacket)
        logger.debug(f"Sent response packet: {resp}")

    @staticmethod
    def SavePrivateNotebook(user_auth, notebookID, changes):
        """Save private notebook

        Args:
            user_auth (tuple): user authorization data
            notebookID (str): notebook ID
            changes (tuple): the changes the client requested (command, change data)

        Returns:
            dict: {'code': error code, 'data': relevent response data}
        """
        resp = SQL.DataQuery(*user_auth, "id", 'NotebookPath', 'currentGroupID', table="notebooks",
userIDString='ownerID', where=f"id={notebookID}", singleton=True, returnUserID=True)

        id = resp['UserID']
        logger.info(f"User {id} is saving notebook {notebookID}...")

        if resp['code'] == 1:
            errMsg = f"User {id} doesn't own notebook {notebookID}"
            logger.error(errMsg)
            return {'code': 1, 'data': errMsg}
        elif resp['code'] == 0:
            groupID = int(resp['data']['currentGroupID'])
            for i, change in enumerate(changes):
                Notebook.ChangeNotebook(resp['data']['NotebookPath'], groupID, change, SQL)

            logger.info(f"Succesfully saved user {id}'s notebook {notebookID}")
        return {'code': 0, 'data': "Changes saved"}

    def SavePublicNotebook(self, notebookCode, changes):
        """Save public notebook

        Args:
            notebookCode (str): notebook public code
            changes (tuple): the changes the client requested (command, change data)

        Returns:
            dict: {'code': error code, 'data': relevent response data}"""
        resp = SQL.Request('id', 'notebookPath', table="notebooks", where="code='%s'" % notebookCode,
singleton=True)

        if len(resp) == 0:
            return {'code': 1, 'data': "Unknown notebook code"}
        else:
            self.server.UpdatePipe.send((*list(resp.values()), changes))
            return {'code': 0, 'data': "Changes saved"}

    def SendUpdates(changes):
        ...

    def SaveNotebook(self, packet: HTTP.Packet, file: str = "") -> dict:
        """Save notebook

        Args:
            packet (HTTP.Packet): The packet requesting the save
            file (str, optional): The name of the notebook file from the post request. Defaults to "".

        Returns:
            dict: {'code': error code, 'data': relevent response data}"""
        user_auth = self.getUserAuth(packet)

        notebookCode = ""
        if 'nb' in packet.attr:
            notebookCode = packet.attr['nb']

        changes = json.loads(packet.Payload)
        if notebookCode == "":
            notebookID = file[6:]
            resp = self.SavePrivateNotebook(user_auth, notebookID, changes)
            return resp
        else:
            logger.debug(f"User {user_auth[0]} is saving public notebook {notebookCode}...")
            resp = self.SavePublicNotebook(notebookCode, changes)
```

```
            return resp

    def NewNotebook(self, packet: HTTP.Packet):
        """Create a new notebook

        Args:
            packet (HTTP.Packet): the packet which requested to create the notebook

        Returns:
            dict: {'code': error code, 'data': relevent response data}
        """
        user_auth = self.getUserAuth(packet)
        id = SQL.CheckAuth(*user_auth)

        if id != None:
            payloadDict = json.loads(packet.Payload)
            svgData = payloadDict['svgData']
            payloadDict.pop('svgData', None)
            payloadDict['ownerID'] = id

            try:
                insertResp = SQL.Insert('notebooks', **payloadDict)

                if insertResp['code'] != 0:
                    return insertResp

            except KeyError as e:
                logger.error("User request doesn't have enough data:")
                logger.error(e, exc_info=True)
                return {'code': 1, 'data': f'Missing key: {e}'}

            notebookID = insertResp['inserted_id']

            # Create svg file:
            newPath = f'Protected/Notebooks/{notebookID}.svg'
            with open(newPath, 'w') as FILE:
                svgData = '<?xml version ="1.0"?>\n' + svgData
                FILE.write(svgData)

            # Update file path to DB
            updateResp = SQL.Update(table="notebooks", where=f'id={notebookID}', NotebookPath=newPath)

            if updateResp['code'] == 0:
                return SQL.DataQuery(*user_auth, "id", table="notebooks", userIDString="ownerID", singleton=True)
            else:
                return updateResp
        else:
            return {'code': 1, 'data': 'Authorization denied!'}

    def SignUp(self, payload: str):
        """Manage signup request

        Args:
            payload (str): payload of the request

        Returns:
            dict: {'code': error code, 'data': relevent response data}
        """
        payloadDict = json.loads(payload)
        attemptUsername = payloadDict['username']
        attemptPassword = payloadDict['password']

        SQL.cursor.execute(f"SELECT EXISTS(SELECT 1 FROM users WHERE username='{attemptUsername}')")
        resp = SQL.cursor.fetchall()

        if resp[0][0] == 1:
            return {"code": 1, "description": "Username already exists!"}


        SQL.cursor.execute(f"INSERT INTO users (username, pass) VALUES ('{attemptUsername}',
'{attemptPassword}')")
        SQL.mydb.commit()

        return {"code": 0, "description": "Signed Up successfuly"}

    def RequestData(self, packet: HTTP.Packet, *attr: tuple[str], table="", userIDString="id", where: str=None,
**kwargs):
        """Request user's private data from the database.
```

```
        Args:
            packet (HTTP.Packet): the packet that contains the user's credentials.
            table (str, optional): the table from which to request the data. Defaults to "".
            userIDString (str, optional): the string which represents the owner id in the relevent table.
Defaults to "id".
            where (str | None, optional): additional where sql commands. Defaults to None.

        Returns:
            dict: dictionary containing the error code and data
            e.g: {'code': 0, 'data': somedata}.
        """
        try:
            user_auth = self.getUserAuth(packet)
            resp = SQL.DataQuery(*user_auth,  *attr, table=table, userIDString=userIDString, where=where,
**kwargs)
        except KeyError as e:
            resp = {"code": 1, "data": "No cookie was sent"}

        return resp

    def LoginAttempt(self, packet: HTTP.Packet, includePayload: bool=True):
        """Manage login attempt

        Args:
            packet (HTTP.Packet): packet whith the login request
            includePayload (bool, optional): Defaults to True.

        """
        # TODO: Make use of the "id" request
        try:
            resp = self.RequestData(packet, "id", table="users", userIDString="id")
            resp = json.dumps(resp)
            respPacket = HTTP.Packet()
            respPacket.Headers['Content-Type'] = "text/json"

            if includePayload:
                respPacket.setPayload(resp)
            else:
                respPacket.Headers['Content-Length'] = len(resp)

            self.SendPacket(respPacket)
            logger.debug(f"Sent login response packet: {resp}")
        except Exception as e:
            raise e

    def PublicResponse(self, file, includePayload=True):
        """manage response for public files.

        Args:
            file (str): the file which was requested by the user.
            includePayload (bool, optional): Defaults to True.
        """
        if file == '/':
            file = "/index.html"

        if not file.startswith('/node_modules') or '..' in file:
            filePath = "public/" + file
        else:
            filePath = file[1:]

        fileRespPacket = self.FileResponsePacket(filePath, includePayload=includePayload)
        sentBytes = self.SendPacket(fileRespPacket)

        msg = f"Sent {filePath} to {self.addr}"

        if not includePayload:
            if silentLog:
                return
            else:
                msg += " without payload"
        logger.info(msg)

    def SendNotebookList(self, packet: HTTP.Packet, includePayload=True):
        """Send notebook list as requested from client after checking authorization.

        Args:
            packet (HTTP.Packet): packet with the notebook list request
            includePayload (bool, optional): Defaults to True.
        """
```

```
        notebookList = self.RequestData(packet, "id", "ownerID", "title", "description", table="notebooks",
userIDString="ownerID")

        nbListPacket = HTTP.Packet(json.dumps(notebookList, indent=4), includePayload=includePayload)

        self.SendPacket(nbListPacket)

    def SendNotebook(self, packet: HTTP.Packet, includePayload=True):
        """Send notebook data after checking authorization.

        Args:
            packet (HTTP.Packet): packet requesting the notebook
            includePayload (bool, optional): Defaults to True.
        """
        isPublicNB = 'nb' in packet.attr

        if not isPublicNB:
            notebookID = packet.filename[10:]

            nbdatadict = self.RequestData(packet, "NotebookPath", "title", "currentGroupID", "code",
table="notebooks", userIDString="ownerID", where=f"id={notebookID}", singleton=True)

        else:
            code = packet.attr['nb']
            sqlReqResp = SQL.Request("NotebookPath", "title", 'id', "currentGroupID", "code", table="notebooks",
where=f"code='{code}'", singleton=True)
            nbdatadict = {'code': 0, 'data': sqlReqResp}

            notebookID = sqlReqResp['id']

        if 'code' in nbdatadict and nbdatadict['code'] == 0:
            filePath = nbdatadict['data'].pop('NotebookPath', None)
            with open(filePath) as FILE:
                nbdatadict['data']["NotebookData"] = FILE.read()

        nbdataPacket = HTTP.Packet(nbdatadict, includePayload=includePayload, dataType='text/json')
        self.SendPacket(nbdataPacket)
        logger.info(f'Sent notebook {notebookID} to {self.addr} with groupid:
{nbdatadict["data"]["currentGroupID"]}')

    def APIPostResponse(self, APIpacket):
        """Manage code creation api post request and response

        Args:
            APIpacket (HTTP.Packet): packet requesting the code

        Returns:
            dict: the code for the notebook as {"code": code}
        """
        notebookID = APIpacket.Payload
        authResp = self.RequestData(APIpacket, "code", table='notebooks', userIDString='ownerID',
where=f"id={notebookID}", singleton=True)
        if authResp['code'] == 0:
            resp = self.UpdateNotebookCode(notebookID)
        return resp

    def getResponseManage(self, packet: HTTP.Packet, includePayload=True):
        """Manage GET request packet

        Args:
            packet (HTTP.Packet): the packet requesting the data.
            includePayload (bool, optional): wheather or not to include the payload in the response (for HEAD
requests). Defaults to True.
        """
        try:
            file = packet.filename
            if file == "/LOGIN":
                self.LoginAttempt(packet, includePayload=includePayload)
            elif file.startswith("/NotebookList"):
                self.SendNotebookList(packet, includePayload=includePayload)
            elif file.startswith("/Notebook"):
                self.SendNotebook(packet, includePayload=includePayload)
            elif file.startswith('/api/'):
                apiRequest = file[5:]
                self.APIGetResponse(packet, apiRequest)
            elif file.startswith('/UPDATE'):
                self.SignClientForUpdate(packet)
            else: # If file is public
                self.PublicResponse(file, includePayload=includePayload)
```

```
        except Exception as e:
            if isinstance(e, FileNotFoundError):
                logger.error(f"404:\n{e}")
                errorPacket = self.FileNotFoundMsgPacket(str(e).split()[-1][1:-1])
                self.SendPacket(errorPacket)
            elif isinstance(e, ConnectionAbortedError):
                logger.debug(f"{self.addr} Aborted Connection")
            elif isinstance(e, SQLException):
                logger.error(f"SQL Error:\n{e}\n{traceback.format_exc()}")
                errorPacket = HTTP.Packet(json.dumps({"code": 1, "data": "Internal Server Error"}), status="500")
            elif isinstance(e, InvalidLoginAttempt):
                logger.error(f"{e}\n{traceback.format_exc()}")
                errorPacket = HTTP.Packet({"code": 1, "data": f"invalid login attempt, {str(e)}"}, status="400")
            else:
                logger.error(f"{e}\n{traceback.format_exc()}")
                errorPacket = HTTP.Packet(f"Unknown Error: {e}", status="520")

            self.SendPacket(errorPacket)

    def getResponse(self, packet):
        self.getResponseManage(packet)

    def headResponse(self, packet):
        self.getResponseManage(packet, includePayload=False)

    def APIGetResponse(self, APIpacket: HTTP.Packet, apiUrl: str) :
        """Manage notebook API requests

        Args:
            APIpacket (HTTP.Packet): packet requesting the data
            apiUrl (str): requested url
        """
        if apiUrl == "notebook/code":
            notebookID =  APIpacket.attr['nbID']
            # Verify the opener is the owner of the notebook
            authResp = self.RequestData(APIpacket, "code", table='notebooks', userIDString='ownerID',
where=f"id={notebookID}", singleton=True)
            if authResp['code'] == 1:
                apiRespPacket = HTTP.Packet("Invalid credentials", filename=apiUrl, status="403")
            else:
                currentCode = authResp['data']['code']
                apiRespPacket = HTTP.Packet({'code': currentCode}, filename=apiUrl, dataType='text/json')


        self.SendPacket(apiRespPacket)

    def UpdateNotebookCode(self, notebookID):
        """Update notebook code to the sql database

        Args:
            notebookID (str): the id of the notebook to update

        Returns:
            dict: the code for the notebook as {"code": code}
        """
        code = GenerateNotebookCode()
        updateResp = SQL.Update(table="notebooks", where=f"id={notebookID}", code=code)
        return {"code": code}

    def SignClientForUpdate(self, packet: HTTP.Packet):
        """Sign client for update in the update process loop

        Args:
            packet (HTTP.Packet): packet requesting the sign
        """
        code = packet.attr['code']
        req = SQL.Request("id", table="notebooks", where="code='%s'" % str(code), singleton=True)
        nbid = req['id']

        # Signing the client for update
        logger.info(f"Signing client {self.addr} for update from notebook {nbid}")
        if nbid in self.server.onlineClients:
            self.server.onlineClients[nbid].append(self)
        else:
            self.server.onlineClients[nbid] = [self]

    def parseHttpPacket(self, packetByteData: bytes) -> HTTP.Packet:
        """parse the packet byte data and return the corresponding Packet instance
```

```
        Args:
            packetByteData (bytes): packet data as bytes

        Returns:
            HTTP.Packet: the returned parsed packet.
        """
        packetStr = packetByteData.decode()
        packet = HTTP.extractDataFromPacket(packetStr)
        if packet.command == "POST":
            while len(packet.Payload) < int(packet.Headers['Content-Length']):
                # Since the length is already set there is no need to use .setPayload
                packet.Payload += self.Recieve().decode()

        return packet

    def close(self):
        self._open = False
        self.stream.shutdown(socket.SHUT_WR)
        self.stream.close()

    def isOpen(self):
        return self._open

    def manage(self):
        """Manage packet and its response"""
        # Define all actions
        Actions = {"GET": self.getResponse, "POST": self.postResponse, "HEAD": self.headResponse}

        while True:
            try:
                packetByteData = self.Recieve()
                if packetByteData == b'':
                    logger.info(f'Recieved empty packet from {self.addr}')
                    try:
                        self.stream.send(b'\r\n')
                        self.close()
                    except Exception:
                        pass
                    return
                    # continue

                packet = self.parseHttpPacket(packetByteData)


                command = packet.command
                if command != None:
                    if command in Actions:
                        Actions[command](packet)
                        if packet.getHeader('Connection') != 'keep-alive':
                            self.close()
                            return
                    else:
                        logger.error(f"command {command} is not supported!")
                else:
                    # print(packetStr)
                    ...
            except:
                try:
                    exc_type, exc_obj, exc_tb = sys.exc_info()
                    fname = os.path.split(exc_tb.tb_frame.f_code.co_filename)[1]

                    raise
                except ConnectionAbortedError as e:
                    logger.error(f"connection aborted with {self.addr}!")
                    self.close()
                    return 1
                except HTTP.ParsingError as e:
                    logger.error(f"{e}", exc_info=True)

                except Exception as e:
                    eText = f"{self.addr}\n========================\n\n\t" +
traceback.format_exc().replace('\n', '\n\t') + f"\n----------------------
\n\t{exc_obj}\n========================="
                    logger.error(eText)
                    # ! Generally dont need this, but it is here for debugging
                    # logger.debug("\n->->->->->->->->->->->\n" + pformat(packetByteData) + '\n->->->->->->->->->->->-
>\n')
            ...
```

```
    def start(self):
        self.thread.start()

    def __str__(self):
        return f"{self.addr}"

class Server:
    """Singleton class for server managment
    """
    def SendUpdates(self):
        """send updates from update queue.
        """
        while True:
            try:
                if not self.ClientUpdateQueue.empty():
                    nbID, changes = self.ClientUpdateQueue.get()
                    while len(self.onlineClients[nbID]) != 0:
                        client = self.onlineClients[nbID].pop()
                        logger.info(f"sending {client.addr} updates")
                        try:
                            changes = json.dumps([str(change) for change in changes])
                            if client.isOpen():
                                client.SendPacket(HTTP.Packet(changes, filename="/UPDATE", dataType="text/json"))

                        except Exception as e:
                            logger.error(f"{client.addr} was disconnected: {e}", exc_info=True)
                            self.onlineClients[nbID].append(client)
                            break
            except Exception as e:
                logger.error(e, exc_info=True)

    def start(self):
        self.consoleThread = threading.Thread(target=console)
        self.consoleThread.start()

        self.context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
        self.context.load_cert_chain(certfile="https/servercert.pem", keyfile="https/serverkey.pem")

        self.bindSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ADDR = ('', port)
        self.bindSocket.bind(ADDR)
        self.bindSocket.listen()
        self.hostname = socket.gethostname()
        self.local_ip = socket.gethostbyname(self.hostname)

        child_conn, self.UpdatePipe = mp.Pipe()
        self.ClientUpdateQueue = mp.Queue()
        self.UpdateNotebookProcess = mp.Process(target=UpdateOpenNotebooksLoop, args=(child_conn,
self.ClientUpdateQueue))
        self.UpdateNotebookProcess.start()

        self.onlineClients = {}
        self.UpdateClientsThread = threading.Thread(target=self.SendUpdates, )
        self.UpdateClientsThread.start()
        logger.info(f"[INIT] Server running on {self.local_ip, port, self.hostname = }")
        self.run()

    def run(self,):
        self.clients = []
        while True:
            clientSocket, addr = self.bindSocket.accept()
            try:
                connStream = self.context.wrap_socket(clientSocket, server_side=True)
                myClient = Client(connStream, addr, server=self)
                self.clients.append(myClient)
                myClient.start()
            except ssl.SSLError as e:
                if isinstance(e, ssl.AlertDescription):
                    logger.warning(e)
            except Exception as e:
                logger.error(f"{e}\n{traceback.format_exc()}\n\nClosing client {addr}")
                clientSocket.close()


class Change:
    """class for managing changes in the update process loop
    """
    def __init__(self, t: str, val: Any, NotebookID: str, code=None):
        """Initiator for Change Class
```

```
        Args:
            t (str): change type (e.g "a" = append/add, "e" = erase)
            val (Any): Value of the change (e.g the added group data or the removed group id)
            NotebookID (str): notebook id
            code (_type_, optional): Defaults to None.
        """
        self.t = t
        self.val = val
        self.code = code
        self.NotebookID = NotebookID

    def __str__(self):
        if self.code != None:
            strData = {"command": self.t, "data": self.val, "updateCode": self.code}
        else:
            strData = {"command": self.t, "data": self.val}

        return json.dumps(strData)


class Notebook:
    """Class for managing changes in public notebooks
    """
    def __init__(self, id: str, SQL: SQLClass, ClientUpdateQueue: mp.Queue):
        """Initiator for Notebook class.

        Args:
            id (str): notebook id
            SQL (SQLClass): SQL class relevent to notebook associated with a server
            ClientUpdateQueue (mp.Queue): Queue in charge of transfering changes between porocesses.
        """
        self.id = id
        self.path = ""
        self.Queue = mp.Queue()
        self.UpdateThread = None
        self.SQL = SQL
        self.clients = []
        self.ClientUpdateQueue = ClientUpdateQueue

    def addChanges(self, change: tuple):
        self.Queue.put(change)

    def hasPath(self,):
        return self.path != ""

    def setPath(self, path):
        self.path = path
        return self

    def UpdateNotebook(self):
        """Main loop for the public notebook updates
        """
        sentUpdates = True
        while True:
            try:
                if not self.Queue.empty():
                    sentUpdates = False

                    changes = self.Queue.get()
                    if changes == "stop":
                        return
                    sqlResp = self.SQL.Request("currentGroupID", table="notebooks", where=f"id={self.id}",
singleton=True)
                    currentGroupNumber = sqlResp['currentGroupID']
                    changesList = []
                    for change in changes:
                        changesList.append(self.ChangeNotebook(self.path, currentGroupNumber, change, self.SQL))

                elif not sentUpdates:
                    self.sendChanges(changesList)
                    sentUpdates = True
            except Exception as e:
                logger.error(f"{e}", exc_info=True)

    def sendChanges(self, changeTupleList):
        changeList = []
        for changeTuple in changeTupleList:
            changeList.append(Change(*changeTuple, self.id))
```

```
        self.ClientUpdateQueue.put((self.id, changeList))

    ns = "{http://www.w3.org/2000/svg}"

    # TODO: Update notebooks using the xml package
    @staticmethod
    def ChangeNotebook(path: str, currentGroupID: int, change: tuple, SQL: SQLClass):
        """Add changes to a notbook

        Args:
            path (str): notebook path
            currentGroupID (int): the current group id in the notebook
            change (tuple): the change whished upon the notebook
            SQL (SQLClass): SQL class associated with the notebook's server

        Returns:
            tuple: the final change enflicted upon the notebook
        """
        changeCMD = change[0]
        changeData = change[1]
        tree = ET.parse(path)
        root = tree.getroot()
        if changeCMD == 'a':
            newElement = ET.fromstring(changeData)
            root.append(newElement)
            currentGroupID += 1
            newElement.set('id', str(currentGroupID))
            SQL.Update('notebooks', 'NotebookPath=\'%s\'' % path, currentGroupID=currentGroupID)

            finalChange = (changeCMD, ET.tostring(newElement).decode())
        elif changeCMD == 'e':
            id = changeData['id']
            t = changeData['type']
            group = root.find(f".//{Notebook.ns + t}[@id='{id}']")
            if group != None:
                root.remove(group)

            finalChange = (changeCMD, changeData)

        tree.write(path)
        return finalChange

    def start(self):
        self.UpdateThread = threading.Thread(target=self.UpdateNotebook)
        self.UpdateThread.start()

def CodeEncryptionKey() -> str:
    """Encription for the code creation key, available for change.

    Yields:
        Iterator[byte]: key
    """
    key = 0
    while True:
        yield bytes(key) + str(time.time()).encode()
        key += 1

def GenerateNotebookCode() -> str:
    """Create a new notebook code using an encryption key and MD5

    Returns:
        str: code
    """
    keys = CodeEncryptionKey()
    code = hashlib.md5(next(keys)).hexdigest()[:5]
    return code

def UpdateOpenNotebooksLoop(child_conn, ClientUpdateQueue: mp.Queue):
    """Main loop for update process loop.

    Args:
        child_conn (Connection): child connection to the update pipe
        ClientUpdateQueue (mp.Queue): Queue for the client updates"""

    # Main function for the update process
    OpenNotebooks: dict[str, Notebook] = {}
    changesList = []
    connectedClients = {}
```

```
    # Connect to database
    logger.info("Connecting to database from update process")
    updateNBSQL = SQLClass()
    logger.info("Connected to database from update process")

    while True:
        try:
            changesList.append(child_conn.recv())

            while len(changesList) != 0:
                msg = changesList.pop(-1)

                NotebookID, NotebookPath, NBchanges = msg
                if NotebookID not in OpenNotebooks:
                    notebook = Notebook(NotebookID, updateNBSQL, ClientUpdateQueue).setPath(NotebookPath)
                    notebook.start()

                    OpenNotebooks[NotebookID] = notebook
                else:
                    notebook = OpenNotebooks[NotebookID]
                    if not notebook.hasPath():
                        notebook.setPath(NotebookPath)
                        notebook.start()

                notebook.addChanges(NBchanges)

        except Exception as e:
            logger.error(f"{e}\n{traceback.format_exc()}")


if __name__ == "__main__":
    # Load SQL
    logger.info("Initializing Database from main thread...")
    SQL = SQLClass()
    SQL.initMainSQL()
    logger.info("Database initialized")

    def exitFunc(*args):
        SQL.exitHandler()
        os._exit(0)

    def removeUser(username):
        SQL.Remove('users', username)

    def removeNotebook(notebookID):
        SQL.Remove('notebooks', notebookID)

    def Remove(*args):
        if len(args) == 0:
            logger.warning("Did not recieve arguments!")
        elif len(args) == 1:
            removeUser(args[0])
        elif args[0] == "ID":
            removeNotebook(args[1])
        else:
            logger.warning("No such command: ")

    def toggleSilentHeaderLog():
        global silentLog
        silentLog = not silentLog

    def printClientList():
        logger.debug(f"Client List:\n{pformat(server.clients)}")

    actions = {"exit": exitFunc, "remove": Remove, "save": SQL.saveDBToJson, "silent": toggleSilentHeaderLog,
"clients": printClientList}

    # Start console:
    def console():
        """
        Main I/O Console loop.
        Available functions:
        1. exit: exit the server safely
        2. remove: remove a user or a notebook, one argument is interpreted as a user and two, if the first is
"ID" then the second the notebook id (e.g remove ID 1 `removes the notebook` with id 1 and `remove 13` removes
user with id 13)
        3. save: save the database to the json files safely
        4. silent: silent header logs; for debug purposes
        5. clients: print online client list
```

```
    """
    while True:
        try:
            cmdtxt = input()
        except EOFError as e:
            exitFunc()

        cmd, *args = cmdtxt.split()
        logger.info(f"Executing Server Command: {cmd}")
        try:
            actions[cmd](*args)
        except KeyError as e:
            logger.warning(f"No Such Command: {cmd}")

server = Server()
server.start()


# ! Send SQL Module with the its global variables
# parent_conn.send(SQL)




# clientSocket.settimeout(10*60)
```